High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations

Steven J. Deitz

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

Doctor of Philosophy

University of Washington

2005

Program Authorized to Offer Degree: Computer Science & Engineering

University of Washington

Graduate School

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

Steven J. Deitz

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

_____

Carl Ebeling

_____

Lawrence Snyder

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral 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 the dissertation 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 Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, 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

High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations

by Steven J. Deitz

Chair of Supervisory Committee:

Professor Lawrence Snyder
Computer Science & Engineering

Call a parallel program *p-independent* if and only if it always produces the same output on the same input regardless of the number or arrangement of (virtual) processors on which it is run; otherwise call it *p-dependent*. Most modern parallel programming facilities let scientists easily and unwittingly write p-dependent codes even though the vast majority of programs that scientists want to write are p-independent. This disconnect—between the restricted set of codes that programmers want to write and the unrestricted set of codes that modern programming languages let programmers write—is the source of a great deal of the difficulty associated with parallel programming.

This thesis presents a combination of p-independent and p-dependent extensions to ZPL. It argues that by including a set of p-dependent abstractions into a language with a largely p-independent framework, the task of parallel programming is greatly simplified. When the difficulty of debugging a code that produces the correct results on one processor but incorrect results on many processors is confronted, the problem with the code is at least isolated to a few key areas in the program. On the other hand, when a programmer must write per-processor code or take advantage of the processor layout, this is possible.

Specifically, this thesis extends ZPL in three directions. First, it introduces abstractions to control processor layout and data distribution. Since these abstractions are first-class and

mutable, data redistribution is easy. Moreover, because these abstractions are orthogonal to the computation, they are p-independent. Second, it introduces abstractions for processor-oriented programming that relax ZPL's programming model and provides a mechanism for writing per-processor codes. Third, it introduces high-level abstractions for user-defined reductions and scans.

In addition, this thesis quantitatively and qualitatively evaluates ZPL implementations for three of the NAS benchmark kernels: EP, FT, and IS. ZPL code is shown to be easier to write than MPI code even while the performance is competitive with MPI.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACKNOWLEDGMENTS

I would like to start by thanking my advisor Larry Snyder. I couldn't have imagined a better advisor. I'd like to thank also my informal advisor Brad Chamberlain. This work would not have been possible without the work he did in his dissertation three years ago and his tireless and continued involvement in ZPL. Additionally, I'd like to thank Sung-Eun Choi, my practicum advisor at Los Alamos National Laboratory.

Thanks to Craig Chambers, Carl Ebeling, and Burton Smith for being on my reading and supervisory committees and for their valuable feedback and criticisms.

Thanks to the many people who contributed to the ZPL project in the past and present. In particular, Ruth Anderson, Brad Chamberlain, Sung-Eun Choi, Marios Dikaiakos, George Forman, Maria Gullickson, E Christopher Lewis, Vassily Litvinov, Douglas Low, Calvin Lin, Ton Ngo, Kurt Partridge, Jason Secosky, Larry Snyder, Peter Van Vleet, Derrick Weathersby, and Wayne Wong.

Thanks to Lindsay Michimoto and Judy Watson for, among other things, making attending graduate school as smooth and easy as it was.

I would like to thank and acknowledge the Department of Energy High Performance Computer Science Fellowship which I was supported by for four and a half years. The performance results in this dissertation were made possible by a grant of supercomputing time at the Arctic Region Supercomputing Center in Fairbanks, Alaska. I would like to thank the staff up there for their excellent service throughout the years.

And finally, I thank my mother, father, and brother.

DEDICATION

To my mother, who recently changed careers to teach grade school in the Bronx.

Chapter 1

INTRODUCTION

Parallel programs are notoriously difficult to write. They require a tremendous effort from the programmer, and when the inevitable errors show up, debugging can easily compromise the productivity of even the most seasoned programmer. When a program produces the correct results on one processor, or a few processors, but fails to work correctly on many processors, maybe hundreds, thousands, or hundreds of thousands, the reason is often difficult to ascertain. Moreover, the bug may not show up until years after the program was written and only when the program is run on a very large number of processors. As an example, consider the case of the NAS CG benchmark. Written in the nineties, a bug in the CG benchmark that showed up only on 1024 processors or higher was not uncovered until 2003 immediately before a paper deadline [DCCS03]. The details of this bug are discussed in the literature [CCDS04].

In order to improve the productivity of parallel programmers with respect to the debug process, researchers have proposed using a powerful technique, called *relative debugging* [AS96], for finding errors in parallel programs [WA00]. In relative debugging, a correct *reference* program is executed alongside an incorrect *suspect* program. In this way, an error in the suspect program can be detected by comparing its data to the reference program's data. Originally developed for porting an application from one language to another, relative debugging is also useful for comparing a correct sequential execution to a flawed parallel execution and thus finding the origin of an error in the parallel program.

This thesis addresses the same issues from the opposite side; rather than relative debugging, the subject of this thesis is *absolute programming*. Call a parallel program *p-independent* if and only if it always produces the same output on the same input regardless

of the number or arrangement of (virtual) processors on which it is run; otherwise call it *p-dependent*. For example, the canonical "hello world" program is trivially a p-independent program assuming it prints out the "hello world" message exactly once regardless of how many processors are executing it. Likewise, a program that prints out the number of processors on which it is being run is trivially a p-dependent program.

It is important to note that processors, as concerns this thesis, are virtual processors, not physical processors. Most parallel programming languages virtualize the processors and often the virtual processors are referred to by a different name to make the distinction clear. For example, Co-Array Fortran's images, Titanium's demesnes, ZPL and Chapel's locales, UPC's threads, and X10's places all refer to the virtual processors on which the program is running, be they threads or processes or some other implementation. The chief point is that they let the programmer reason about locality within the parallel program. They do not make it any easier to write p-independent programs, however.

In absolute programming, the programmer is restricted to writing p-independent programs. A programming language for which every program is p-independent is called an *absolute programming language*. Similarly, a programming abstraction that cannot be used to introduce p-dependent values into any program execution is called an *absolute programming abstraction*. This thesis ignores three important issues in which p-dependent values can be introduced in practice: round-off errors caused by reordering floating-point arithmetic operations, non-deterministic codes involving quantities such as timing or external machine state, and implementation issues such as out-of-memory errors stemming from finite memories.

Absolute programming languages are easier to use for developing and debugging because the difficulties associated with fragmenting a problem over a set of processors do not exist. For example, race conditions and deadlocks are absent from absolute programming languages. Moreover, if a program written in an absolute programming language produces the correct answer on one processor, it will produce the correct answer on any number of processors. The programmer need only hope that the answer will be produced faster on more processors, not that the answer will be correct. In short, only the performance can be impacted when the number or arrangement of processors is changed.

Performance is critical to parallel programming so a parallel programming language that significantly compromises high performance is useless, whether it is absolute or not. Therefore, a performance model is just as important as absolute programming abstractions. Since inter-processor communication is the bottleneck of most parallel programs, the performance model should be able to identify communication in its syntax, *i.e.*, the programming language should have *syntactically identifiable communication*. This point has been effectively argued in the literature [CCL+98b]. There is nothing inherent to absolute programming languages that makes syntactically identifiable communication unachievable even though absolute programming abstractions are, by definition, orthogonal to the processors.

Despite the advantages of absolute programming languages, they are not the vogue. They are widely though not universally believed to be insufficient. Indeed, high performance sometimes demands p-dependent programming. Important low-level optimizations that introduce temporary p-dependent values into a program's execution, *e.g.*, array contraction, are sometimes critical. Moreover, some problems allow for multiple correct solutions, *e.g.*, search, and a p-dependent program that computes a different correct solution on different numbers of processors can be significantly faster than a p-independent program that computes the same solution on all numbers of processors.

Ideally, then, a language should provide an absolute programming framework with syntactically identifiable communication and a small set of p-dependent abstractions. Optimizing programs is easy because communication bottlenecks are syntactically identifiable and low-level optimizations can be implemented where necessary using the p-dependent abstractions. Debugging programs is easy because there are a limited number of places in the code where p-dependent values can be introduced, namely, where the p-dependent abstractions are used.

The advantage to absolute programming languages, and mostly absolute programming languages, is enormous. The overwhelming majority of parallel programs are p-independent or, rather, are supposed to be p-independent. That said, the vast majority of today's parallel programming languages allow programmers to write p-dependent code easily and unwittingly. This disconnect—between the restricted set of codes that programmers want to write and the unrestricted set of codes that modern programming languages let programmers

write—is the source of a great deal of the difficulty associated with parallel programming.

Most of today's parallel programs are written in a sequential programming language such as C or Fortran coupled with the message passing library MPI [Mes94]. MPI is solidly in the tradition of Communicating Sequential Processes (CSP) [Hoa78]. The MPI programmer writes a single program that is independently executed by each processor. Communication is achieved through MPI library routines for sending and receiving data. These routines are called two-sided message-passing routines since the processors on both sides of the communication need to be involved in any exchange. One processor must call the send routine and the other processor must call the receive routine, though not necessarily at the same time. Other library calls are provided for collective communication.

The advantage to MPI is that the programmer is given great flexibility with relatively few abstractions. Communication is also clear to the programmer; it is induced only by MPI library calls. The disadvantages to MPI are numerous and are well recognized by the parallel programming community. The major disadvantage is that p-independent programs are truly difficult to develop, *i.e.*, the programs that scientists want to write are difficult to write. It should be noted that MPI was not originally designed to be used by the application programmer, but rather by high-level library developers and language designers.

A new library called ARMCI [NC99], which is similar to SHMEM [BK94] but is more portable, improves upon MPI, though it is also not intended to be used by the application programmer. Rather than providing library routines for sending and receiving data, ARMCI provides one-sided message-passing routines for putting or getting data. Only one processor is involved in the communication and the data simply changes on the other processor. Routines for synchronization are essential for both ARMCI and MPI. By providing such low-level per-processor abstractions, ARMCI also does little to ease the task of writing p-independent programs. It is arguably even more difficult to write p-independent code in ARMCI since the processors can work even more independently.

A class of languages, called Global Address Space (GAS) languages, has recently gained support. Co-Array Fortran [NR98], Titanium [YSP+98], and Unified Parallel C (UPC) [CDC+99] are all GAS languages. They provide a global address space which makes it significantly easier to program. All three languages, however, are still in the tradition of CSP. The

SPMD programs are executed on each processor and they communicate with one another in unstructured ways. This makes it difficult to write p-independent programs. It should be noted that Titanium does provide some support for p-independent programming and this will be discussed further in Section 4.9. Moreover, communication is not syntactically identifiable in either Titanium or UPC.

It is worth mentioning two higher-level parallel programming languages at this point. High Performance Fortran [Hig97] and OpenMP [DM98] both rely on directives to suggest parallel execution strategies. Unlike the languages mentioned so far, HPF and OpenMP (to an extent) are not in the tradition of CSP. There is a single thread of execution through the program and the statements of the program are executed in parallel. This change is significant and makes it much easier to write p-independent programs because synchronization is implicit.

Unfortunately, neither HPF nor OpenMP are as p-independent as they can be. If the directives the programmer writes are incorrect, the results could change depending on the number of processors executing the code. Because it is important to write aggressive directives, mistakes are easy to make and the resulting p-dependent errors are difficult to find [GMI03]. Moreover, communication is not syntactically identifiable in either OpenMP or HPF. They are both difficult to optimize, a chief reason HPF failed to be accepted and OpenMP only runs on shared memory computers.

Absolute programming languages have existed in the past, but they suffered from the same problems of HPF and OpenMP. Namely, they were difficult to optimize for computers with physically distributed memory. Examples of such languages include Sisal [MSA+85], NESL [Ble95], and Id [Nik91].

1.1 Contributions

This thesis extends the ZPL programming language to create a robust and general parallel programming language with syntactically identifiable communication and an absolute programming framework with four p-dependent abstractions. Specifically, this thesis proposes three extensions to ZPL:

- **Processor grids and data distributions.** Abstractions for managing processor layout and data distribution are essential for achieving high performance. In high-level languages, these abstractions can be orthogonal to a program's results and thus, counter-intuitively, p-independent. This work presents such abstractions with stricter p-independent semantics and performance guarantees than previously developed.

- **Free variables and grid dimensions.** Abstractions for per-processor coding are essential for optimizing complex algorithms and for taking advantage of faster implementations of problems that allow for multiple solutions. This work adds two lower-level abstractions to ZPL and shows they provide great power even while being safer and easier to use than their counterparts in other languages.

- **User-defined reductions and scans.** ZPL provides nine built-in reduction/scan operators–sum, product, minimum, maximum, logical and, logical or, bitwise and, bitwise or, and bitwise xor–that provide programmers with the ability to write optimized code for their application. This work allows programmers to define new operators for reductions and scans. This is shown to be essential in applications that could benefit from their use and these applications are shown to be quite common.

1.2 Outline

Chapter 2 provides a brief introduction to the ZPL programming language. It is shown to be a mostly p-independent programming language that makes parallel programming easier without sacrificing performance. Changes that were made to ZPL in order to support this thesis's extensions are highlighted.

Chapter 3 presents high-level, p-independent abstractions for managing processor layout and data distribution. These abstractions are shown to be sufficient for supporting data reallocation and redistribution at a high level. The related work of HPF is discussed.

Chapter 4 presents low-level, p-dependent abstractions for processor-oriented programming. They are shown to provide the flexibility needed to optimize code on a per-processor basis while still allowing a program to remain largely p-independent. Numerous examples

are presented.

Chapter 5 presents high-level, p-dependent abstractions for writing user-defined reductions and scans. They are shown to be essential for achieving high performance in applications that require them. Their potential to introduce p-dependent values into a program execution is shown to be limited.

Chapter 6 presents ZPL versions of the NAS EP, FT, and IS benchmarks and compares them against the provided MPI versions qualitatively and quantitatively.

Chapter 7 summarizes the contributions of this thesis and proposes directions for future work.

Chapter 2

THE ZPL APPROACH

ZPL is a parallel programming language developed at the University of Washington. Designed from first principles, it is unique in providing a mostly p-independent semantics alongside syntactically identifiable communication. This chapter presents a brief introduction to ZPL and provides a basis for the extensions described in later chapters; For more information on ZPL, the reader is referred to the literature [AGNS90, AGL+98, AS91, CCL+96, CCS97, CCL+98a, CLS98, CCL+98b, CLS99a, CLS99b, CDS00, CCL+00, CS01, Cha01, CCDS04, CS97, Cho99, CD02, DCS01, DCS02, DCCS03, Dei03, DCS04, DLMW95, GHNS90, LLST95, LLS98, LS00, Lew01, LS90, LS91, Lin92, LW93, LS93, LS94a, LS94b, LSA+95, NS92, NSC97, Ngo97, RBS96, Sny86, Sny94, Sny95, Sny99, Sny01, Wea99].

Throughout this chapter, changes to ZPL from the literature will be noted. In addition, how the current implementation differs from the features described will be noted. Lastly, it will be noted periodically that the base of ZPL is mostly p-independent.

2.1 ZPL's Parallel Programming Model

ZPL holds to a simple parallel programming model in which data is either replicated or distributed. In both cases, there is logically one copy of any given data object; replicated data is kept consistent via an implicit consistency guarantee brought to fruition by an array of static typechecking rules.

Figure 2.1 illustrates a good way of visualizing how ZPL achieves parallelism. Parallel arrays are distributed across the processors. Scalar variables and indexed arrays are replicated across the processors. This replicated data is kept consistent and can thus be viewed by the programmer as a single object.

One key feature of ZPL is its performance model. Communication between processors is only induced by a small number of parallel operators. Moreover, these parallel operators

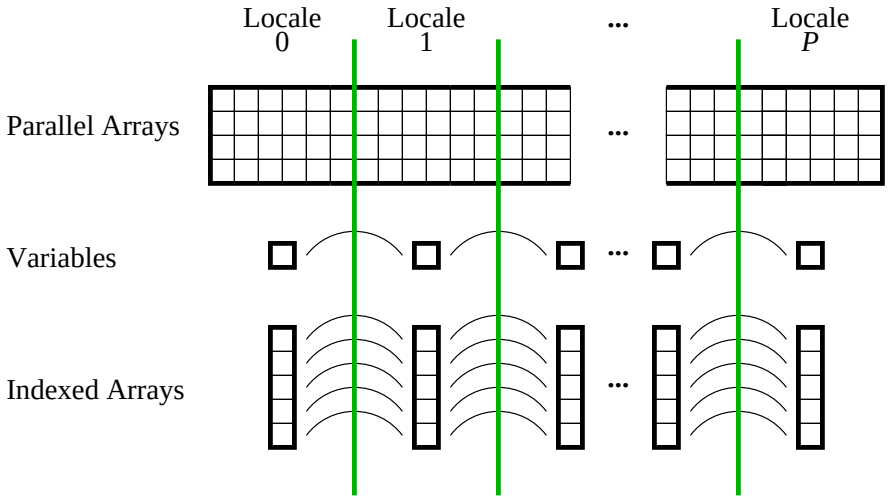

Figure 2.1: An Illustration of ZPL's Parallel Programming Model

signal the kind of communication that may take place.

Unlike most SPMD programming languages, most of ZPL is p-independent. This makes ZPL fundamentally different from languages like UPC and Titanium in which synchronization poses a serious challenge to the programmer. In ZPL, synchronization is implicit in the programming model. Race conditions and deadlocks are impossible.

2.2 Scalar Constructs

ZPL's parallelism stems from regions and parallel arrays, introduced in Section 2.3. This section presents ZPL's scalar constructs. They mimic Modula-2 [Wir83] and are described only briefly.

2.2.1 Types, Constants, and Variables

ZPL provides standard support for types, constants, and variables. In the parallel implementation, types and constants are replicated across the processors. Because they contain the same value on each processor, there is logically one of each type and constant. Variables are also replicated across the processors. By replicating the serial computation across the processor set as well, the replicated variables are kept consistent and can be thought of as

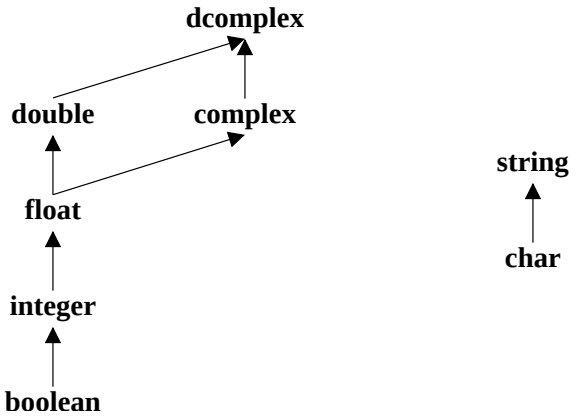

Figure 2.2: Type Coercion of a Selection of ZPL's Scalar Types

being a singular value.

2.2.2 Scalar Type Coercion

ZPL is a strongly typed language that supports type coercion on its base types. This is an important aspect to ZPL's base types that, as will be seen, extends to its parallel types as well. Figure 2.2.2 illustrates the implicit type coercions of ZPL. Expressions of a type lower in the hierarchy are automatically converted to expressions of a type higher in the hierarchy where necessary. Thus if a `double` value and an `integer` value are added together, the result is a `double` value. ZPL also supports explicit type coercions on most of the base types using a set of standard functions, *e.g.*, `to_integer`.

2.2.3 Scalar Operators and Bit Manipulation Procedures

ZPL provides the standard array of arithmetic, logical, relational, and assignment operators and bit manipulation procedures. These are listed in Table 2.1.

2.2.4 Configuration Constants and Variables

Configuration constants and variables are a unique concept to ZPL. They provide the same functionality as the `argc` and `argv` arguments to `main` in C, but do so in a simpler and less error-prone manner. Configuration constants and variables, which can only be declared

Table 2.1: ZPL's Scalar Operators and Bit Manipulation Procedures

Arithmetic Operators	
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`^`	exponentiation
`+`	unary plus
`-`	unary negation
`%`	modulus

Logical Operators	
`!`	unary negation
`&`	logical and
`|`	logical or

Relational Operators	
`=`	equality
`!=`	inequality
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

Assignment Operators	
`:=`	standard
`+=`	additive
`-=`	subtractive
`*=`	multiplicative
`/=`	divisive
`&=`	conjunctive
`|=`	disjunctive

Bit Manipulation Procedures	
`band(x,y)`	bitwise and
`bor(x,y)`	bitwise or
`bxor(x,y)`	bitwise exclusive or
`bnot(x)`	bitwise negation
`bpop(x)`	population count
`bsl(x,y)`	left shift
`bsr(x,y)`	right shift

Listing 2.1: Examples of Configuration Constants and Variables

```
1 config const
2   verbose : boolean = false; -- verbose mode if true
3   n : integer = 64;          -- problem size
4   lgn : integer = bpop(n-1); -- log of the problem size

6 config var
7   m : integer = 12;          -- initial problem value
```

Listing 2.2: Examples of Enumerated Types, Records, and Indexed Arrays

```
1 type
2   classes = (S, W, A, B, C, D);

4   valpos = record
5     value : float;
6     index : integer;
7   end;

9   vector = array[1..n] of float;
```

in the global scope of a program and must be initialized, may have their initialization overridden on the command line. Listing 2.1 shows some example configuration constants and variables.

Because configuration constants and variables may be initialized on the command line, their types are limited to the basic types, including strings and enumerated types. Configuration constants and variables may not be indexed arrays or records.

> Change Note. Configuration constants are new. In the manual, configuration variables cannot be changed after they are initialized. Now configuration variables can be changed, but configuration constants cannot be. This change unifies the rules on all variables and constants.

2.2.5 Enumerated Types, Records, and Indexed Arrays

ZPL provides standard support for enumerated types, records, and indexed arrays. Listing 2.2 shows an example enumerated type, record type, and indexed array type.

Indexed arrays are similar to arrays in most other programming languages; they are called indexed arrays because they can be indexed into. This distinguishes them from parallel arrays which cannot be directly indexed into (as will be seen). When referring to whole indexed arrays, programmers may leave the square brackets blank. In this case, a loop is automatically generated around the statement containing the array reference. For example, if `v` is of type vector, then the following code initializes all of its elements to zero:

```
v[] := 0.0;
```

In a statement, multiple blank array references must conform, *i.e.,* the blank dimensions must be the same size and shape.

Counterintuitively, indexed arrays are considered scalars in ZPL. This further distinguishes them from parallel arrays.

2.2.6 Input/Output

Basic input and output is handled in ZPL by three procedures, `write`, `writeln`, and `read`. The `write` procedure writes its arguments in the order that they are specified. The `writeln` procedure behaves identically except that a linefeed follows its last argument. The `read` procedure reads its arguments in the order that they are specified.

ZPL also provides a `halt` procedure to end a program's execution. The `halt` procedure behaves just like the `writeln` procedure except that the program is terminated after it completes.

2.2.7 Control Flow and Procedures

ZPL supports standard control structures for conditionals, `if`, and loops, `for`, `repeat`, and `while`. Listing 2.3 shows examples of each of ZPL's control structures.

ZPL provides a rich interface for procedures. Parameters may be passed with the following qualifiers: `const`, `in`, `out`, and `inout`. By default, parameters are passed with the `const` qualifier. These qualifiers specify whether a parameter will be changed by a procedure. Their meanings are as follows:

- If a variable is passed by `const`, the variable is not changed by the procedure. Moreover, the formal parameter may not be changed within the procedure. This allows the implementation to pass the variable by reference or by value.

- If a variable is passed by `in`, the variable is not changed by the procedure. However, the formal parameter may be changed within the procedure. The resulting change will simply not be reflected back to the call site. This is equivalent to a call by value.

- If a variable is passed by `out`, no value is taken into the procedure. The value that is assigned to the variable within the procedure is returned to the call site. This is

Listing 2.3: Examples of Control Flow

```
1  if n <= 0 then
2    halt("Problem size is too small.");
3  elsif bpop(n) != 1 then
4    halt("Problem size is not a power of two.");
5  else
6    writeln("Problem size = ", n);
7  end;

9  for i := 1 to n do
10   v[i] := 0.0;
11 end;

13 i := 1;
14 while i <= n do
15   v[i] := 0.0;
16   i := i + 1;
17 end;

19 i := 1;
20 repeat
21   v[i] := 0.0;
22   i := i + 1;
23 until i > n;
```

equivalent to a call by reference where the procedure assumes the variable is uninitialized.

- If a variable is passed by `inout`, the change in the procedure is reflected back to the call site. This is equivalent to pass by reference.

Implicit aliasing is not allowed in ZPL. If an `inout`, `out`, or `const` parameter may alias a different parameter or a global variable, the programmer must specify that this is acceptable. It may result in less optimized code, but the cost of copying to avoid aliasing may be greater.

Change Note. Previously, procedure parameters could either be passed by reference or value. The new parameter passing mechanism is more informative and an overall improvement. In addition, the alias keyword is new. Aliasing was previously allowed.

Implementation Note. At the time of this writing, alias analysis is not implemented and the alias keyword is not recognized.

P-Independent Note. In the definition of the language so far, the programmer can only write p-independent code. The code is completely sequential and each processor executes the identical computation. There is no possibility that p-dependent values can be introduced in to the execution, but there is also no potential for a parallel speedup.

2.3 Regions

A region is a convex index set that may be strided. For example, the declarations

```
region R         = [0..n-1, 0..n-1];
       InteriorR = [1..n-2, 1..n-2];
```

create the following two index sets, respectively:

$$\{(i,j)|0 \leq i \leq n-1, 0 \leq j \leq n-1\}$$

$$\{(i,j)|1 \leq i \leq n-2, 1 \leq j \leq n-2\}$$

Regions `R` and `InteriorR` are two-dimensional constant regions with unit stride. Regions with non-unit stride are discussed in Section 2.3.3 and variable regions are discussed in Chapter 3.

The rank of a region is the dimensionality of its indexes. Both `R` and `InteriorR` have rank two. The region `Line` declared by

```
region Line = [0..n-1];
```

has rank one. Regions can contain a single index in any dimension, and such dimensions are called *singleton dimensions*. A singleton dimension can be abbreviated with the single integer expression that would be duplicated if the dimension were written as a range. For example, the first row of `R` can be declared as in

```
region TopRow = [0, 0..n-1];
```

2.3.1 Anonymous Regions

Regions `R`, `InteriorR`, and `Line` are all named regions. A literal region may be used wherever a named region may be used. Such regions are called *anonymous regions* since there is no way to refer to them later. They are commonly used in computations where a

dynamic portion of a larger named region needs to be used. For example, the anonymous region

```
[i, 0..n-1]
```

refers to the `i`th row of `R` if `i` is an integer between `0` and `n-1` inclusive.

2.3.2 Implicit Distributions

Regions serve two main purposes: declaring parallel arrays and specifying the indexes of a parallel array computation. The distribution of indexes over the set of processors is implicit. Explicit distributions are discussed in Chapter 3.

All regions of the same rank share the same implicit distribution. The distributions serve to map the indexes in the regions to a set of processors. Because the distributions are implicit, ZPL is sometimes referred to as an implicitly parallel language. As will be seen, this is inaccurate. Though ZPL programmers do not manage the details of the parallel implementation, they are made well aware of the parallelism in their programs. Thus the parallelism is explicit.

2.3.3 Directions and Region Operators

Regions are manipulated with a set of specialized binary operators called *prepositions*. Prepositions apply to a region and a direction to create new regions.

Before defining regions and presenting the prepositions, it is worthwhile discussing the representation of regions. A region can be represented by a set of integers in each dimension. The region is then the cross-product of these sets. The integer sets are specified by a 4-tuple (l,h,s,a) where l is the low bound, h is the high bound, s is the stride, and a is the alignment. The integer set is then

$$\{i | l \leq i \leq h, i \equiv a \bmod s\}$$

For example, the set given by (2,10,3,1) is 4,7,10 and the region `R` is the cross-product of the integer set given by (`0`,`n-1`,`1`,`0`) and itself.

Directions

A direction is simply an offset vector. For example, the declarations

```
direction
  northwest = [-1, -1]; north = [-1,  0]; northeast = [-1, 1];
       west = [-1,  0];                        east = [ 0, 1];
  southwest = [-1,  1]; south = [ 1,  0]; southeast = [ 1, 1];
```

create offset vectors in the four cardinal and four radial directions. As with regions, literal directions may be inlined wherever directions are used, and they are called *anonymous directions*.

Of and In

The prepositions `of` and `in` are useful for boundary computations. They are binary operators that expect a direction on the left and a region on the right. The `of` preposition creates a new region on the outside of the region to which it applies and the `in` preposition creates a new region on the inside of the region to which it applies. The regions are created relative to the low bound of the region if the direction is negative and relative to the high bound of the region if the direction is positive. For example, the following equivalences hold:

$$\texttt{east of R} \approx \texttt{[0..n-1, n]}$$
$$\texttt{west of R} \approx \texttt{[0..n-1, -1]}$$
$$\texttt{east in R} \approx \texttt{[0..n-1, n-1]}$$
$$\texttt{west in R} \approx \texttt{[0..n-1, 0]}$$

The semantics for `of` and `in` are defined for each dimension in Figure 2.3.3.

At and By

The `at` and `by` prepositions expect a region on the left and a direction on the right (the opposite of `in` and `of`). The `at` preposition creates shifted regions. For example, the following equivalence holds for `at`:

$$\texttt{R at east} \approx \texttt{[0..n-1, 1..n]}$$

$$\delta \text{ of } (l,h,s,a) \Rightarrow \begin{cases} (l+\delta, l-1, s, a) & \text{if } \delta < 0 \\ (l,h,s,a) & \text{if } \delta = 0 \\ (h+1, h+\delta, s, a) & \text{if } \delta > 0 \end{cases}$$

$$\delta \text{ in } (l,h,s,a) \Rightarrow \begin{cases} (l, l-(\delta+1), s, a) & \text{if } \delta < 0 \\ (l,h,s,a) & \text{if } \delta = 0 \\ (h-(\delta-1), h, s, a) & \text{if } \delta > 0 \end{cases}$$

$$(l,h,s,a) \text{ at } \delta \Rightarrow (l+\delta, h+\delta, s, a+\delta)$$

$$(l,h,s,a) \text{ by } \delta \Rightarrow \begin{cases} (l, h, -\delta \cdot s, h + ((a-h) \bmod s)) & \text{if } \delta < 0 \\ (l,h,s,a) & \text{if } \delta = 0 \\ (l, h, \delta \cdot s, l + ((a-l) \bmod s)) & \text{if } \delta > 0 \end{cases}$$

$$(l,h,s,a) - \delta \Rightarrow \begin{cases} (l-\delta, h, s, a) & \text{if } \delta < 0 \\ (l,h,s,a) & \text{if } \delta = 0 \\ (l, h-\delta, s, a) & \text{if } \delta > 0 \end{cases}$$

$$(l,h,s,a) + \delta \Rightarrow \begin{cases} (l+\delta, h, s, a) & \text{if } \delta < 0 \\ (l,h,s,a) & \text{if } \delta = 0 \\ (l, h+\delta, s, a) & \text{if } \delta > 0 \end{cases}$$

Figure 2.3: Preposition Semantics. The per-dimension semantics for `of`, `in`, `at`, `by`, `+`, and `-` are shown for a direction where δ is the dimension-component and a region where (l,h,s,a) is the integer set 4-tuple dimension-component.

The `by` preposition creates strided regions. Strided regions can only be created with `by`. The stride of the region is simply multiplied by the direction for each dimension. The alignment is adjusted relative to the low bound if the direction is positive and relative to the high bound if the direction is negative. The following equivalences hold for the `by` preposition:

$$\texttt{R by [1,4]} \approx \texttt{R by [1,2] by [1,2]}$$

The semantics for `at` and `by` are defined for each dimension in Figure 2.3.3.

+ and -

The binary operators `+` and `-`, which are overloaded as unary operators, are also overloaded as prepositions. Like `at` and `by`, the `+` and `-` prepositions expect a region on the left and a direction on the right. They are used to expand and contract regions. The low bound of the region is modified if the direction is negative; otherwise the high bound is modified. For example, the following equivalences hold:

$$\texttt{[1..n-2] + [1]} \approx \texttt{[1..n-1]}$$

$$\texttt{[1..n-2] - [1]} \approx \texttt{[1..n-3]}$$

$$\texttt{[1..n-2] + [-1]} \approx \texttt{[0..n-1]}$$

$$\texttt{[1..n-2] - [-1]} \approx \texttt{[2..n-1]}$$

$$\texttt{InteriorR} \approx \texttt{R - northeast - southwest}$$

The semantics for `+` and `-` are defined for each dimension in Figure 2.3.3.

> Change Note. The`+` and `-` prepositions are new and add useful functionality especially given the extensions of Section 4.2.

> Implementation Note. At the time of this writing, there is no parser support for the + and - prepositions.

A Note on Region and Direction Literals

Region and direction literals may be indistinguishable if the region is singleton in every dimension. Nonetheless, it is easily identifiable as one or the other by both the programmer

and the compiler based on the context. For example, the following equivalences are easy to verify:

$$\texttt{[1,1] of [2,2]} \approx \texttt{[3,3]}$$
$$\texttt{[2,2] of [1,1]} \approx \texttt{[2..3,2..3]}$$

2.3.4 Region Types

Regions are first-class in ZPL. They are typed constructs and their rank is part of their type. As new concepts are introduced, the type of the region will be expanded. For example, in addition to a region's rank, the type of each of its dimensions is part of its type. So far, only one dimension type, *range*, has been introduced so this is uninteresting. Section 2.8 will introduce a second dimension type and Chapter 4 will introduce a third and final dimension type.

The syntax used for declaring regions previously is a useful syntactic sugar for declaring constant regions. Regions `R`, `InteriorR`, `TopRow`, and `Line` could also be declared using the following syntax:

```
const R         : region<..,..> = [0..n-1, 0..n-1];
      InteriorR : region<..,..> = [1..n-2, 1..n-2];
      TopRow    : region<..,..> = [0     , 0..n-1];
      Line      : region<..>    = [0..n-1];
```

The "`..`" portion of the region type specifies that a region is declared with dimension type *range* in the dimension. Two more dimension types will be introduced later in this thesis: flood dimensions in Section 2.8 and grid dimensions in Chapter 4. Note that for initialized regions it is legal to declare the type simply as `region` since the dimension types can be inferred. For variable regions that are not initialized this type information is necessary.

> Change Note. The extensions in this thesis made it necessary to distinguish between a region's value and type. Previously, it was unclear whether a region was a type or a value and regions were never variables or constants.

2.3.5 Sparse Regions

Sparse regions allow programmers to specify sparse computations over both sparse and dense arrays using syntax similar to masks. By separating the sparse index set from the array, programmers are able to express their sparse computations using a dense array syntax, making the code easier for programmers to read and compilers to optimize. Sparse regions are an important concept in ZPL. Their existence has influenced some of the choices made in this thesis. Nonetheless, they are a complicated subject that is largely orthogonal to the work in this thesis. The reader interested in learning more about ZPL's support for sparse computations is referred to the literature [CLS98, CS01, Cha01].

2.4 Parallel Arrays

As mentioned, regions are used to declare parallel arrays and to specify the indexes of a parallel array computation. ZPL is first and foremost a parallel array language. The region is its central abstraction, but regions are useless without parallel arrays. For simplicity, this thesis will refer to parallel arrays simply as arrays. Where the distinction between parallel arrays and indexed arrays is unclear, indexed arrays will be called indexed arrays.

2.4.1 Arrays and Regions

Regions have no associated data. Arrays are declared over regions and contain data for each index in its declaring region. For example, the declarations

```
var A, B, C : [R] integer;
    L : [Line] integer;
```

create three arrays, `A`, `B`, and `C`, that contain $\texttt{n}^2$ integers each and one array, `L`, that contains `n` integers. The scalar type component of these arrays, `integer`, is called the *base type*. The type of a parallel array consists of the base type and the type of the declaring region. The region itself is not part of the type of the array, although it is statically bound to the array.

Parallel arrays are distributed across the processors according to their region distribution. Thus, if explicit distributions are not used, all parallel arrays of the same rank are distributed in the same way. This allows all basic array computations to execute in paral-

lel without any interprocessor communication. Communication is only induced by ZPL's parallel operators (introduced in Section 2.5).

Basic array computations are elementwise and data-parallel. They are written with a region that specifies the indexes involved in the computation. For example, the statement

```
[InteriorR] A := B + C;
```

assigns the elementwise sums of the interior elements of `B` and `C` to the corresponding elements of `A`. Note that the computing region can be different from the declaring region.

2.4.2 Arrays and Structured Types

Unlike indexed arrays, parallel arrays may not be nested. However, parallel arrays can be nested inside indexed arrays and records and these structured types can be the base types of parallel arrays. For example, the declarations

```
var AofI : [R] array[0..n-1] of integer;
    IofA : array[0..n-1] of [R] integer;
```

create two structures. The parallel array of indexed arrays `AofI` distributes indexed arrays across the processors. The indexed array of parallel arrays `IofA` contains an indexed array of parallel arrays that distribute the integers. Their storage and parallel implementation is different, but they are referenced in the same way. For example, the statement

```
[R] AofI[] := IofA[];
```

assigns all of the elements of `IofA` to `AofI`.

2.4.3 Indexi Arrays

ZPL provides per-rank constant arrays called `Index`i where i is the rank of the array. The arrays can be used in any computation as long as the rank of the computing region is at least i. The `Index`i array contains the ith dimension component of the computing region.

For example, over region `[1..3,1..3]`, the `Index1` and `Index2` arrays are given by

```
Index1 = 1 1 1      Index2 = 1 2 3
         2 2 2               1 2 3
         3 3 3               1 2 3
```

P-Independent Note. In the definition of the language so far, all behavior is p-independent. This follows because the semantics of the language are orthogonal to the processors.

2.5 Parallel Operators

Communication between processors is an expensive part of parallel computing that should be avoided when possible. For interesting parallel computations, communication can be minimized, but it is impossible to avoid completely. In ZPL, unlike many other high-level parallel programming languages, communication is syntactically identifiable. Communication is induced only by the following operators: `@`, `@^`, *op*`<<`, *op*`||`, `>>`, `#`, and `'@`. The advantages of syntactically identifiable communication are great not only for programmers who can easily evaluate the quality of their code, but also for the compiler which can easily optimize local portions of the code. These operators are collectively referred to as the parallel operators.

This section presents the parallel operators and shows examples to acquaint the reader with their semantics. The most important point about ZPL's semantics is that it is an array language and thus the right hand side of an expression can be evaluated before the left hand side. Temporary variables declared over the computing region can be inserted for every expression. The one exception to this rule is with statements involving the prime at, (`'@`), operator. More exceptions will be discussed in Chapter 4. The semantics introduced here are a simpler form of the full semantics. They are complete as regarding parallel arrays with range dimensions and scalars.

2.5.1 The @ and Wrap @ Operators

The @ operator, (`@`), and the wrap @ operator, (`@^`), are useful for nearest neighbor and stencil computations. The @ operator shifts data in an array by a direction similarly to how the `at` preposition shifts indexes in a region. The wrap @ operator also shifts data in an array by a direction and, in addition, applies a toroidal boundary condition to the array.

Figure 2.5.1 shows several examples of applying the @ operator to parallel arrays. The first segment of code initializes arrays `A`, `B`, and `C` from Section 2.4.1. Array `A` is initialized with the `Index`*i* arrays so that each element in `A` is unique; arrays `B` and `C` are zeroed out.

```
[R] A := Index1 * n + Index2;
[R] B := 0;
[R] C := 0;

      A =  0  1  2  3   B =  0  0  0  0   C =  0  0  0  0
           4  5  6  7         0  0  0  0         0  0  0  0
           8  9 10 11         0  0  0  0         0  0  0  0
          12 13 14 15         0  0  0  0         0  0  0  0

[InteriorR] B := A@west;

                        B =  0  0  0  0
                             0  4  5  0
                             0  8  9  0
                             0  0  0  0

[InteriorR] C := A@north + A@east + A@south + A@west;

                                          C =  0  0  0  0
                                               0 20 24  0
                                               0 36 40  0
                                               0  0  0  0

[InteriorR] C@east := C;

                                          C =  0  0  0  0
                                               0 20 20 24
                                               0 36 36 40
                                               0  0  0  0

[R] A := A@^east;

      A =  1  2  3  0
           5  6  7  4
           9 10 11  8
          13 14 15 12
```

Figure 2.4: Examples of the @ and Wrap @ Operators

The next line shows a simple assignment statement from `A` to `B` where the @ operator has been applied to `A` and the direction `west`. Note that the assignment takes place over region `InteriorR`. Thus only the interior elements of `B` are assigned values. They are the values in the region `InteriorR at west`.

The third line shows a more interesting computation. Here the interior elements of `C` are assigned the sum of the four neighbors to the corresponding interior elements of `A`.

The fourth line shows how the @ operator can be applied to the left-hand side of a statement. The @ operator, wrap @ operator, '@ operator, and remap operator may all be used on the left-hand side. The other parallel operators can only be used on the right-hand side. In this example, the data in the interior of `C` is shifted to the right.

The last line shows a use of the wrap @ operator. In this line, the values of `A` are cyclically shifted to the left. As mentioned previously, because ZPL is an array programming language, the right hand side of a statement is evaluated before the left hand side.

2.5.2 The Reduce Operator

The reduce operator is used to compute reductions. It supports two types of reductions called *full reductions* and *partial reductions*. In full reductions, the elements of a parallel array read over some region are combined to form a single scalar value. In partial reductions, the elements of a parallel array read over some region are reduced in one or more dimensions to the same or another parallel array.

Figure 2.5.2 shows several examples of applying the reduce operator to parallel arrays. The first example of the reduce operator shows a full reduction of `A`. Since `R` specifies all the elements in `A`, the first example stores into `i` the sum of all of the elements in `A`. The second example shows a full reduction using the `max` operator instead of the `+` operator and using an alternative syntax. The result is the same; the scalar `i` is assigned the maximum element in `A`.

> Change Note. This new syntax for full reductions is new to ZPL. It makes for a more symmetric language definition.

> Implementation Note. At the time of this writing, this alternative syntax is not sup-

```
[R] A := Index1 * n + Index2;
[R] B := 0;
    i := 0;

      A =  0  1  2  3   B =  0  0  0  0   i = 0
           4  5  6  7        0  0  0  0
           8  9 10 11        0  0  0  0
          12 13 14 15        0  0  0  0

[R] i := +<< A;

                                           i = 120

i := max<<[R] A;

                                           i = 15

[0..n-1, 0] B := +<<[R] A;

                          B =  6  0  0  0
                              22  0  0  0
                              38  0  0  0
                              54  0  0  0

[0, 0] A := max<<[R] A;

      A = 15  1  2  3
           4  5  6  7
           8  9 10 11
          12 13 14 15
```

Figure 2.5: Examples of the Reduce Operator

ported.

The third and fourth examples show partial reductions. In the third example, the second dimension of array `A` is reduced. The first column of `B` is assigned the sum of the rows of array `A`. In the fourth example, the element at position (0,0) in array `A` is assigned the maximum element in `A`.

The semantics of the reduce operator are simple. Full reductions are always assigned to scalar variables; they are always applied to parallel arrays. Partial reductions are always assigned to and from parallel arrays. For partial reductions, there are two regions, a source region and a destination region. The source region applies to the array or array expression being reduced. The destination region applies to the result. A dimension is reduced if the source region is a range in that dimension and the destination region is a singleton in that dimension. If the destination dimension is a range, the source dimension must be the identical range. If the source dimension is a singleton, the destination dimension must be another, not necessarily identical, singleton.

The semantics do not require the source to be a range and the destination to be a singleton so as to allow for partial multi-dimensional reductions (such as the third reduction in Figure 2.5.2. Note that one can use the reduction operator to write a simple assignment between two arrays as in

```
[R] A := +<<[R] A;
```

In this case, since no dimensions are reduced, nothing is done.

2.5.3 The Scan Operator

The scan operator computes parallel prefix computations. The programmer specifies a dimension list that looks like a direction or region of singleton dimensions where every dimension has a unique integer from one to the rank. The length of the dimension list can be no longer than the rank of the computing region. The dimension list orders the scan over the dimensions. The scan operator then uses the operator to compute *exclusive* prefix reductions throughout the computing region. In the exclusive scan, as opposed to an inclusive scan, the last element is not included in the prefix reduction.

```
[R] A := Index1 * n + Index2;
[R] B := 1;
[R] C := 0;

      A =  0  1  2  3    B =  1  1  1  1    C =  0  0  0  0
           4  5  6  7         1  1  1  1         0  0  0  0
           8  9 10 11         1  1  1  1         0  0  0  0
          12 13 14 15         1  1  1  1         0  0  0  0

[R] C := +||[2,1] B;

                                                C =  0  1  2  3
                                                     4  5  6  7
                                                     8  9 10 11
                                                    12 13 14 15

[R] B := +||[1] C;

                           B =  0  0  0  0
                                0  1  2  3
                                4  6  8 10
                               12 15 18 21
```

Figure 2.6: Examples of the Scan Operator

Figure 2.5.3 shows a couple of examples of applying the scan operator to parallel arrays. In the first application, the array `B` is scanned in a row major traversal. Since `B` contains the value `1` everywhere, the resulting array has the same values as `A`. In the second application of the scan, the array C is scanned but only along the first dimension.

> Change Note. In ZPL, the scan operator previously computed an inclusive scan. The exclusive scan is a better choice because any inclusive scan can be easily computed after the exclusive scan has been computed. The reverse is not the case. For operators that do not have an inverse, *e.g.*, `max`, communication is necessary to compute the exclusive scan from the inclusive scan. The transformations presented in Chapter 4 are more efficient given exlusive scans.

2.5.4 The Flood Operator

The flood operator is used to replicate data throughout a distributed array. This broadcast style communication is useful in parallel computing even if it would be inefficient in sequential computing. Section 2.8 will present a new region dimension type that helps to minimize the impact of this replicated data.

```
[R] A := Index1 * n + Index2;
[R] B := 0;
    i := 0;

      A =  0  1  2  3   B =  0  0  0  0   i = 0
           4  5  6  7        0  0  0  0
           8  9 10 11        0  0  0  0
          12 13 14 15        0  0  0  0

[R] B := >>[0..n-1, 0] A;

                         B =  0  0  0  0
                              4  4  4  4
                              8  8  8  8
                             12 12 12 12

[R] A := >> [1, 1] A;

      A =  5  5  5  5
           5  5  5  5
           5  5  5  5
           5  5  5  5

i := >>[1,1] A;

                                          i = 5
```

Figure 2.7: Examples of the Flood Operator

Figure 2.5.4 shows several examples of applying the flood operator to parallel arrays. In the first application of the flood operator, the first column of `A` is replicated throughout the columns of `B`. Notice that with the flood operator, like with the reduce operator, there are two regions, a source region and a destination region. In the case of the flood, a dimension is replicated if the source dimension is a singleton and the destination dimension is a range. This is the opposite of reduce. In the case of flood, if the source dimension is a range and the destination dimension is a range, they must be identical ranges. If the destination dimension is a singleton, the source dimension must be another, not necessarily identical, singleton.

In the second application of the flood operator, the element at position (1,1) in array `A` is replicated throughout the whole of `A`. The third application of the flood operator shows a full flood. This is the logical counterpart to the full reduce. In the full flood, a single element in a parallel array is moved into a scalar variable. Note that in this case, `A` has the same value everywhere so the specified position does not matter. Nonetheless, even in the presence of a high-quality optimizing compiler, the flood operator could still need to induce communication here if, for example, there are processors that do not own any of `A`.

> Change Note. Previously there was no counterpart to the full reduction; the full flood did not exist. Rather, the full reduction operator was used in this case and the operator was ignored.

> Implementation Note. At the time of this writing, full floods are not supported.

> In the definition of the language so far, there are no p-dependent abstractions. This follows because the semantics are independent of the processors. Moreover, where communication does exist, there is no possibility that the logical values will change depending on the processors. Round-off error in reductions and scans is ignored as mentioned in the introduction. The next operator is different because of a many-to-one mapping which may change depending on the processors.

2.5.5 The Remap Operator

The remap operator is often referred to as the catch-all operator. It is used to compute a generalized gather if it is on the left-hand side of a statement or a generalized scatter if it

```
[R] A := Index1 * n + Index2;
[R] B := n - 1 - Index1;
[Line] L := Index1 * n;

      A =  0  1  2  3   B =  3  3  3  3   L =  0  4  8 12
           4  5  6  7         2  2  2  2
           8  9 10 11         1  1  1  1
          12 13 14 15         0  0  0  0

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

      A =  0  4  8 12
           1  5  9 13
           2  6 10 14
           3  7 11 15

[R] B := A#[B, ];

                         B =  3  7 11 15
                              2  6 10 14
                              1  5  9 13
                              0  4  8 12

[R] A := L#[Index2];

      A =  0  4  8 12
           0  4  8 12
           0  4  8 12
           0  4  8 12

[Line] L := B#[Index1,Index1];

                                          L =  3  6  9 12
```

Figure 2.8: Examples of the Remap Operators

is on the right-hand side of a statement. The remap operator also provides an efficient way to move data between parallel arrays of different ranks.

Figure 2.5.5 shows several examples of applying the remap operator to parallel arrays. In the first application of remap, the `Index`*i* arrays are used to transpose the data in `A`.

The second application of the remap uses the array `B` to index into `A` and return values. The second dimension of the remap is left blank so that the region applies directly to that dimension of `A`. If the map list's *i*th dimension is blank, it can be replaced by the array `Index`*i*. These arrays are simply elided when the dimension is blank. This is similar to region inheritance discussed in Section 2.7.3.

The third application of the remap operator shows an example of rank change. Since `L` is one-dimensional, the remap applies only a single map to it. This map, however, is two-dimensional and read over the current two-dimensional computing region. The result is stored in the two-dimensional array `A`. The last application of the remap operator shows an example of rank change from a higher-dimensional array to a lower-dimensional array. In this case, the major diagonal of `B` is copied into the one-dimensional array `L`.

> P-Independent Note. Scatter is a p-dependent abstraction since the result may depend on the number or arrangement of processors. For example, in a statement like `[R] A#[1,1] := B`, all the values in `B` will be stored in position $(0, 0)$ of array `A`. The last value stored will be the one that remains. Which value is last depends on the processor layout.

Scatter and Non-Standard Assignment

The non-standard assignment operators, *e.g.*, `+=`, interact with scatter in an intuitive but non-standard way. Typically, statements like `i += 1` can be interpreted with the following rewrite: `i := i + 1`. In the case of a many-to-one scatter, this rewrite does not apply. Instead, the multiple elements that map to the same position in the destination are all accumulated. For example, the statement

```
[R] A#[1,1] += A;
```

is equivalent to the reduction

```
[1,1] A := A + +<<[R] A;
```

```
[R] A := Index1 * n + Index2;
[R] B := 1;

      A =  0  1  2  3   B =  1  1  1  1
           4  5  6  7         1  1  1  1
           8  9 10 11         1  1  1  1
          12 13 14 15         1  1  1  1

[InteriorR] A := A'@west;

      A =  0  1  2  3
           4  4  4  7
           8  8  8 11
          12 13 14 15

[1..n-1, 1..n-1] B := B'@north + B'@west;

                         B =  1  1  1  1
                              1  2  3  4
                              1  3  6 10
                              1  4 10 20

[0..n-2, 0..n-2] interleave
                   A := B'@[1,1];
                   B := A + A'@east;
                 end;

      A = 41 17  4  3   B = 58 21  7  1
          30 31 10  7        61 41 17  4
           4 10 20 11        14 30 31 10
          12 13 14 15         1  4 10 20
```

Figure 2.9: Examples of the Prime @ Operator

It is not equivalent to the statement

```
[R] A#[1,1] := A#[1,1] + A;
```

2.5.6 The Prime @ Operator and Interleave

The prime @ operator, `'@`, provides language-level support for representing wavefront computations. This operator is briefly noted here. For a more thorough discussion of ZPL's support for pipelining wavefront computations, the reader is referred to the literature [CLS99a, LS00, Lew01].

The prime @ operator breaks the array language semantics of evaluating the right-hand

side before evaluating the left-hand side. For statements involving the prime @ operator, the statement cannot be rewritten with temporary arrays declared over the computing region used to store temporary expressions. For this reason, the prime @ operator is restricted to statements that contain only @, wrap @, and other prime @ operators. In addition, when multiple directions are used in these statements, there are restrictions on the values of these directions that make sense.

The prime @ operator denotes an array whose values should be reused in the computation after they are created. The directions thus induce an order on the computation called a wavefront. Multiple statements can be involved in these wavefronts using *interleaved blocks*. The `interleave` keyword requires multiple statements to be executed together.

In addition to being useful for wavefront computations, interleaving statements lets the programmer manually implement the scalarizing compiler optimization of loop fusion. Interleaving statements can be interpreted as executing the block of statements for each index separately. ZPL's sequential flow of parallel statements is violated in interleaved statement blocks. For this reason, communication is limited in interleave blocks to avoid the issues plaguing languages involving a parallel execution of communicating sequential processes. Such CSP style languages imply the possibility of deadlocks and race conditions which are impossible in ZPL. Interleaved statement blocks are subject to the same restrictions as single statements involving a prime @ operator. Other than the @, wrap @, and prime @ operators, no other parallel operators may be used in interleaved statements. In addition, scalars may not be assigned values in interleaved statement blocks.

> Change Note. Previously, multiple statements could be executed in a single wavefront using a `scan` block rather than an `interleave` block. The semantics were identical, but the keyword was different. By changing the keyword to `interleave` and giving it a more general meaning, the single concept can be reused. The `interleave` keyword is especially useful when combined with the extensions of Chapter 4.

Figure 2.5.6 shows several examples of applying the prime @ operator to parallel arrays. In the first application, the values in the interior of the first column of `A` are replicated throughout the interior columns of `A`. It is important to note that arrays to which the prime @ operator is applied must be assigned in the statement or group of interleaved statements.

Otherwise, there would be no newly created values to read.

In the second application of the prime @ operator, a wavefront computation is induced from the upper-left corner of `B`. Each element not in the first row or column of `B` is assigned the sum of its neighbor to the north and west as the wavefront passes by.

The last application of the prime @ operator shows a more complicated set of interacting wavefronts using two interleaved statements. A pipelined implementation of this code is very difficult to write in a lower-level language like Fortran+MPI. It is also very important to pipeline such wavefronts because they are important in many applications. The prime @ operator makes the code easy to write and guarantees a parallel implementation.

P-Independent Note. The Prime @ operator is p-independent.

2.5.7 ZPL's Parallel Performance Model

Knowing that the parallel operators induce communication lets programmers optimize their programs or choose between alternative algorithms if it can be determined that one program induces significantly more communication than the other. In addition to this knowledge, the programmer knows that some parallel operators are more expensive than others. The @ operator's nearest neighbor communication is significantly cheaper than the remap operator's remap communication in the worst case. The other operators lie between these two extremes. For a more detailed discussion of this important ZPL property, the reader is referred to the literature [CCL+98b].

2.5.8 A Note on Structured Parallel Arrays

The @, wrap @, prime @, and remap operators apply to parallel arrays and result in variables that can be assigned much like indexing into indexed arrays and selecting fields of records. However, unlike array indexing and field selection, these operators do not apply to the point in the type where the region is used to declare a parallel array. They always apply to the outside. For example, when the @ operator is applied to `AofI` and `IofA` from Section 2.4.2, the expressions are as follows:

```
[R] ... AofI[i]@east ... IofA[i]@east ...
```

This is particularly important for cases where the indexing is done by a parallel array. For example, if `i` is replaced by `A`, the @ operator applies to both `AofI`, `IofA`, and `A`. This allows us to maintain the performance model. If the @ operator did not apply to the `A` in either case, extra communication would be necessary to pass the values of `A` onto the processor that is reading or writing `AofI` and `IofA`. The excess communication is even worse in the case of the remap operator.

> Change Note. Previously, the @ and remap operators did not apply to the outside of the expression, but rather to the point where the region was used in the declaration. This was an oversight. It is not too difficult to derive an example that would violate the performance model if the operator did not apply to everything to its left. Though rare, examples of a similar nature are common with the extensions of Chapter 4.

2.6 Promotion

Promotion is a natural concept in ZPL in which scalar operators, values, procedures, and control structures seamlessly interact with parallel arrays. Virtually transparent, promotion has already been introduced. For example, in the simple statement

```
[R] C := A + B;
```

the scalar operators `:=` and `+` were *promoted* to operate element-wise on the parallel arrays.

2.6.1 Scalar Value Promotion

Promotion of scalar values has also been seen before. If a parallel array activates a statement, all scalar variables and values in that statement are transparently promoted. For example, in the statement

```
[R] C := C + 1;
```

the value `1` becomes a parallel array of `1`s. Similarly, a scalar variable would become a parallel array of the same base type with its value replicated.

Values can only be promoted for reading, not writing. Thus a parallel array cannot be assigned to a scalar. This error is caught by the type checker, ensuring that scalars are kept consistent across the processors. For example, the following line of code does not compute

Listing 2.4: An Example of Scalar Procedure Promotion

```
1 procedure add(a, b : integer; out c : integer);
2 begin
3   c := a + b;
4 end;

6 [InteriorR] add(A, B, C);
```

a reduction over parallel array `A`:

```
[R] i += A;
```

Instead it results in a type-checking error.

2.6.2 Scalar Procedure Promotion

Scalar procedures are procedures that do not involve parallel arrays and call only other scalar procedures. Scalar procedures can be promoted in a similar way to scalar operators if and only if they have no side effects. For example, Listing 2.4 shows an example in which a scalar add procedure that takes two integers and stores their sum in a third can be promoted to work on three parallel arrays.

A scalar procedure is promoted if it is called from a statement to which a region applies, *i.e.*, there are parallel arrays in that statement even if not in the arguments to the function.

> Change Note. A scalar procedure only used to be promoted if one or more of its arguments were parallel arrays. Thus if it was assigned to a parallel array, its result would be promoted but the procedure would not be. These old semantics do not mesh well with the extension of Chapter 4 or shattered control flow (introduced shortly).

When a scalar procedure is promoted, all its arguments are promoted. A scalar argument can thus not be passed to any of the promoted procedure's `out` and `inout` parameters. In addition, the procedure's return value is promoted and cannot be assigned to a scalar.

2.6.3 Shattered Control Flow

Control structures may be promoted as well. For example, Listing 2.5 shows how an `if` statement can apply to a parallel array rather than a scalar.

Listing 2.5: An Example of Shattered Control Flow

```
1 [R] if A < 0 then
2   B := -A;
3 else
4   B := A;
5 end;
```

Shattered control flow results in code similar to interleave statements, discussed in Section 2.5.6. The statements in the control flow block are executed for each index. The parallel operators may not be used in shattered control flow and scalars may not be assigned within shattered control flow.

Note that shattered control flow, interleaved statement blocks, and promoted scalar procedures are all similar. They all replace ZPL's sequential composition of parallel statements with a parallel execution of statements for each index. The restrictions associated with each of these concepts eliminates the possibility of a parallel composition of communicating sequential processes.

> P-Independent Note. Promotion and shattered control flow are p-independent because the semantics are orthogonal to the processors.

2.7 Region Scopes

Region scopes can be opened over single statements, compound statements, or control flow statements, but a region only actively applies to statements that contain parallel arrays. If a statement contains no parallel arrays, the region is ignored. Thus regions are said to be *passive*. For example, region `R` serves no purpose in the following line of code (assuming i is a scalar integer):

```
[R] i := i + 1;
```

It is good ZPL style to capitalize parallel arrays and regions while leaving non-parallel variables, *e.g.*, scalars and indexed arrays, lowercase. This serves as a visual clue to whether a region is active or passive for any given statement.

Listing 2.6: An Example of a Dynamic Region Scope

```
1 procedure Add(A, B : [R] integer; inout C : [R] integer);
2 begin
3   C := A + B;
4 end;

6 [InteriorR] Add(A, B, C);
```

Region scopes may be nested. In these cases, only the innermost region applies to statements involving parallel arrays. The outer regions are *eclipsed*.

2.7.1 Dynamic Region Scopes

A region scope may also span procedures. That is, regions are dynamically, not lexically, scoped. Listing 2.6 shows an example of a dynamically scoped region. In this example, region `InteriorR` applies to the parallel procedure call `Add`. Because it is not eclipsed in the procedure body, it is used to implement the array statement `C := A + B`.

2.7.2 Masked Region Scopes

Masks are parallel arrays of `Boolean` base type. They can be attached to computing regions using either the keyword `with` or `without`. They specify whether indexes are involved in the computation (`with`) or not (`without`). For example, given a `Boolean` parallel array `M` declared over `R`, the statement

```
[R with M] A := B + C;
```

computes the elementwise sum of `B` and `C` and stores the result in `A` wherever `M` is true. The mask must be readable over the region to which it applies but does not need to be declared over that region.

2.7.3 Region Inheritance

When region scopes are opened, they may inherit one or more dimensions from the eclipsed region. This is legal in the case where the eclipsed region is dynamically scoped. In this case, the procedure may have to be cloned depending on the types of the regions. There

are two forms of region inheritance: blank dimensions and ditto regions.

Blank Dimensions

Blank dimensions allow the programmer to open a new region scope based on the current region scope. For example, in the code

```
[R] begin
  A := B;
  [i, ] B := C;
end;
```

the values in `B` are assigned to `A` and then the values in the `i`th row of `C` are assigned to the `i`th row of `B`.

Blank dimensions are especially useful for writing reductions and floods since the reduce and flood operators require that non-singleton dimensions are identical, inheritance can save some redundant specifications. For example, writing

```
[R] A := >>[1, 0..n-1] A;
```

is equivalent to writing

```
[R] A := >>[1, ] A;
```

Ditto Scopes

The ditto region, `["]`, refers to the region scope. It is useful for creating new regions using prepositions based on the current region, *e.g.*, `[east of "]`. Because preprocessors sometimes complain about non-terminating strings, the ditto region may also be written with two apostrophes as in `['']`.

The ditto specifier may also be used to inherit masks from the current region scope. In this case, the ditto is used after the keyword `with` or `without`.

2.8 Flood Dimensions

Flood dimensions, `*`, are special dimensions of regions that specify a single value and conform to any index. They are used to hold data that is logically replicated across dimensions of

an array. For example, the declarations

```
region Row    = [*      , 0..n-1];
       Column = [0..n-1, *     ];
```

create two regions with one flood dimension and one range dimension. Region `Row`'s first dimension is a flood dimension; region `Column`'s second dimension is a flood dimension.

2.8.1 Flood Arrays

Flood arrays are arrays that are declared over flood regions. They have logically a single element in their flood dimension that can be read over any index. In the implementation, the single values are replicated over the flood dimensions and each processor contains a copy. These copies are kept consistent across the processors in the same way that scalars are kept consistent across the processors. Flood arrays are declared similarly to regular arrays. For example, the declarations

```
var FR : [Row] integer;
    FC : [Column] integer;
```

create flood arrays over `Row` and `Column`.

Since flood dimensions conform to any range dimension, flood arrays can be read over any region. The statement

```
[R] A := FR * FC;
```

computes the cross-product of the two replicated vectors `FR` and `FC`. Arrays with a flood dimension that are read over a region not flooded in that dimension are said to be promoted, in much the same way that scalars are promoted. Promotion of flood arrays is discussed in more detail in Section 2.8.5.

The inverse of this promotion is not allowed. Arrays not flooded in a dimension cannot be read or written over a region flooded in that dimension because only one value is associated with a flood dimension and there are multiple values in non-flooded dimensions. In addition, arrays not flooded in a dimension cannot be written over regions flooded in that dimension. This enforces their consistency. Like the consistency of scalars, the consistency of flood arrays is maintained not by extra communication but rather by a restricted use enforced by

type-checking.

2.8.2 Indexi Arrays and Flood Dimensions

The `Index`*i* arrays are closely related to flood arrays. For example, in a two-dimensional region, `Index1` has a single value that is replicated across the second dimension. The second dimension of `Index1` is a flood dimension. Similarly, the first dimension of `Index2` is a flood dimension. In general, every dimension other than the *i*th dimension of `Index`*i* is a flood dimension. Thus `Index`*i* is readable over any region as long as the *i*th region is not a flood dimension.

2.8.3 Parallel Operators and Flood Dimensions

When parallel operators are used on flood arrays, the same semantics as before apply if the regions do not have flood dimensions. The flood arrays are simply promoted. (The remap operator is an exception and this will be discussed shortly.) If the regions involved do have flood dimensions, the semantics are natural and are explained here.

The @, Wrap @, and Prime @ Operators

The @, wrap @, and prime @ operators are ignored over flood dimensions. Since flood arrays have infinitely replicated data over the flood dimension, accessing neighboring elements are naturally the same. For example, the statements

```
[Row] FR := FR@east + 1;
[Row] FR := FR + 1;
```

are identical for the same reason that

```
[R] A := FR@east + 1;
[R] A := FR + 1;
```

are identical. In the latter statements, `FR` is promoted and the elements in the rows are identical. In the former statements, `FR` is not promoted, but the single data value conforms across the arrays. Note that in the latter statement, `FR@east` does not result in an out-of-bounds error since the flood dimensions can be read over any index in that dimension.

The Reduce Operator

There are two cases to consider with the reduce operator: whether the destination region is flooded or whether the source region is flooded. Since flood dimensions have a logically singular value, they are similar to singleton dimensions. Values in an array can thus be reduced into flood dimensions. That is, a dimension is reduced if the destination region is flooded in that dimension and the source region is a range or a singleton in that dimension. Note that if the source region is singleton in that dimension, the reduction is over the one value.

In the other case, when the source region is flooded in a dimension, the reduction is legal, but it has no effect. The result of the reduction is an array that is flooded in the same dimension. This array can be read, by promotion, over a range or singleton dimension.

The Scan Operator

Like the @ operator, the scan operator has no effect over flooded dimensions. It is treated the same way it is treated over singleton dimensions. Since there is only one logical value in that dimension, there is nothing to scan.

The Flood Operator

The flood operator, as its name suggests, is intimately related to the flood dimension. Although the reduce operator can be used to move data in a regular array into a flood array, this is a degenerate case. It is much like a full reduction over a singleton region. The flood operator does a better job of this because there is no operator to ignore.

In a flood dimension, there are two cases to consider. If the destination region is flooded in a dimension, then the source region must either be a singleton in that dimension or flooded in that dimension. If it is a singleton, the value is replicated into the flood dimension. This is the common usage. If it is flooded as well, there is no effect. In the second case, if the source region is flooded in a dimension, the source expression, which must be readable over the flood dimension, is unchanged. It can then be promoted to either a singleton or range dimension of the destination region.

The Remap Operator

The remap operator is more complicated than the other operators. Even if the region is not flooded in any dimension, the semantics may be impacted if the array being remapped is flooded. This is different than the other operators in which the array could just be promoted.

If the region is flooded in some dimension, the effect on the remap operator is minor. The remapped array or expression does not need to be flooded in that dimension. It is read with the indexes generated by the maps. All other expressions and arrays, as well as the map expressions, do need to be flooded in that dimension (or they can be promoted scalars).

The interesting changes show up when the remapped array is flooded in some dimension. (Note that if the remapped array is flooded in all of its dimensions, the remap operator has no effect whatsoever and is trivially uninteresting. The interesting case is when the array is flooded in some dimensions, but not others.)

In the case of the gather, the map array that applies to a dimension in which the remapped array is flooded is, in a sense, unnecessary. No matter what values that map array contains, the result will be the same. The map array, however, can be useful in specifying an optimized parallel implementation. Since the indexes are distributed over the processors, different values in the map array could cause different processors to send the same value to other processors. In the case of the scatter, if the destination array is flooded in some dimension, there is a difficulty. This is because the maps will generate indexes to write the destination array and this would break the consistency of flood dimensions. For this reason, a special flood map is provided to read or write flood arrays.

For example, in the gather

```
[Column] FC := FR#[Index2,Index2];
```

the replicated row array is moved into a replicated column array. The second map array is unnecessary in the sense that it has no effect on the computation. Since `FR` is flooded in the second dimension, no matter what the values fed by the second map, the remapped values would be the same. The flood map allows the programmer to make this explicit. For

example, the gather above and the three gathers below are all equivalent:

```
[Column] FC := FR#[Index2,1];
[Column] FC := FR#[Index2,*];
[Column] FC := FR#[Index2, ];
```

The scalar map array `1` reads from the first position in the second dimension of `FR`. This is the same as all the positions. The flood map `*` reads from `FR` as if the region `FR` is read over is flooded in the second dimension. In the final gather, the map array is elided and the result is similar to using `Index2`.

A scatter can be used for the same computation but, in this case, the only legal implementation is to use a flood map:

```
[Row] FC#[*,Index1] := FR;
```

Since `FC` can only be written over a region flooded in the first dimension, the flood map is necessary. Gather is only different because the array is being read, not written.

> Change Note. The flood map is new and has not appeared in previous literature. Useful on its own, it is symmetric to an extension of Chapter 4.

> Implementation Note. At the time of this writing, the flood map is not implemented.

2.8.4 Region Operators and Flood Dimensions

The region operators `of`, `in`, `at`, `by`, `+`, and `-` can be applied to regions with flood dimensions. The flood dimensions of the region are unchanged; the range dimensions are changed as always.

2.8.5 Beyond Promotion: Basic Parallel Type Coercion

Promotion, discussed in Section 2.6, is more formally parallel type coercion. In promotion, scalars are changed into parallel arrays just as integers are changed into doubles in scalar type coercion (See Section 2.2.2).

Both scalars and flood arrays can be promoted and read over regions with range dimensions. For example, the statement

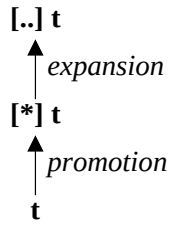

Figure 2.10: Basic Type Coercion of ZPL's Parallel Types

```
[R] A := F + i;
```

promotes the flood array `F` and the scalar `i` and then assigns their sum to `A`. Figure 2.8.5 shows a hierarchy for basic parallel type coercion. As with scalar type coercion, the coercion only applies to reading values, not writing. In the hierarchy, promotion refers to a scalar being changed to a flood array. Coercing a flood array into a range array is called *expansion*. This notes the actual replication of data on each processor. Informally, or wherever the distinction is unimportant, both coercions are called promotion.

Chapter 4 will introduce two more parallel types and complete this hierarchy. As it stands, however, it applies very well to ZPL as it is defined. Without the extensions of Chapter 4, there is no need for an expanded parallel type hierarchy.

> P-Independent Note. Flood dimensions are p-independent since there is logically a single value. The number of processors distributing a flood dimension is immaterial. Of course, the scatter remap is still p-dependent even if it involves flood dimensions.

2.9 Summary

This chapter presented a brief introduction to ZPL. Changes since the last major discussions of ZPL [Sny99, Cha01] have been highlighted. As a research project, ZPL has evolved significantly since its first release in 1997. With any luck, it will evolve further still.

ZPL is unique in the field of parallel programming in its support of both syntactically identifiable communication and a mainly p-independent semantics. The extensions described in this thesis will present some challenges and exceptions to these properties but, in general, will maintain them. They are a great strength to programmers trying to write

high-performance parallel programs in a reasonable amount of time.

Chapter 3

SUPPORT FOR DYNAMIC DATA DISTRIBUTIONS

The abstractions discussed in the last chapter make parallel programming easier by making data distribution and processor layout implicit. With the exception of the scatter remap operator, all of Chapter 2's constructs are p-independent. In addition, the ZPL programmer largely ignores the underlying processors. This chapter's constructs are also p-independent (See discussion in Section 3.5.), but the programmer no longer ignores the underlying processors.

In ZPL, processors are called locales to emphasize that, as noted before, they are virtual, not physical, processors. Two important intrinsic functions, `numLocales` and `localeID`, expose the underlying processors to the ZPL programmer. The `numLocales` function returns the number of processors executing the program. The `localeID` function returns a unique integer between zero and `numLocales()-1` that identifies the processors executing the code. Each processor returns the same value for `numLocales`, but a different value for `localeID`. Chapter 4 will discuss rules limiting how these procedures may be used. Note that a trivially p-dependent program is easy to write. It need only contain the following line:

```
writeln("Number of processors = ", numLocales());
```

Exposing the underlying processors is important because high performance sometimes demands programmer management of data distribution and virtual processor layout even if the semantics of a program are best kept orthogonal to the virtual processors. This chapter provides a high-level, p-independent means of managing data distribution and processor layout. Specifically, it adds to ZPL a hierarchy of high-level language abstractions: grids, distributions, regions, and arrays. A key property of this support is the mutability of grids, distributions, and regions, enabling the dynamic redistribution of data and rearrangement of processor layout.

Both low-level and high-level parallel programming languages must support dynamic data distributions because important applications such as Adaptive Mesh Refinement (AMR) require data to be redistributed on the fly. In low-level languages, programmers can implement dynamic distributions with low-level abstractions; changing the organization of data is complicated but possible with the same abstractions necessary for managing the normal details of data reallocation, interprocessor communication, and intraprocessor copying. On the other hand, high-level languages, which rely heavily on high-level abstractions to insulate programmers from low-level per-processor details, do not necessarily provide abstractions that support dynamic data distributions. For example, HPF 1.0 provided only static data distributions while SISAL and other absolute programming languages left data distribution entirely to the implementation.

Consider a simple approximation of AMR called *Computational Zoom*. This example will be returned to later in this chapter. In computational zoom, there is some array computation—it's details are immaterial—with the following property: the more time spent computing on some portion of the array, the more precise the solution is on that portion of the array. The problem then asks the programmer to compute an approximate solution over the entire array using all the processors and then to compute a more precise solution over a dynamically selected portion of the array again using all the processors. Because the portion of the array is dynamically selected, support for dynamic data distributions is necessary in order to use all the processors to compute the more precise solution.

Without any abstractions for processor layouts or data distributions, there is no way of ensuring that all the processors would be used for the smaller computation. Consider the classic ZPL solution:

```
region R = [1..nx, 1..ny];
var A : [R] double;
    x1, x2, y1, y2 : integer;

[R] computeApprox(A, x1, x2, y1, y2);
[x1..x2, y1..y2] computeMore(A);
```

The problem with the above code is that the distribution of the anonymous compute region used in the last line is unspecified. Without programmer support for processor layout

and distributions, there is no way for the programmer to control this computation. The performance implications are too great to ignore in a parallel programming language.

This chapter is organized as follows. The next section presents grids, an abstraction for managing processor layout. Section 3.2 presents distributions, an abstraction for managing data distribution. Section 3.3 unifies grids and distributions with regions and parallel arrays to form a hierarchy of abstractions for managing processor layout and data distribution. Section 3.4 presents two assignment operators that enable dynamic processor layouts and data distributions by letting the programmer change the values of grids, distributions, and regions. Section 3.5 discusses the impact of this Chapter's extensions on ZPL's performance model and p-independent framework. In particular, these extensions introduce several exceptions. Section 3.6 discusses related work done in the context of High Performance Fortran. Lastly, Section 3.7 concludes this chapter with a summary.

3.1 Grids

The indexes of regions are distributed over an arrangement of virtual processors called *grids*. A grid is simply a processor layout. For example, if `p` is the number of virtual processors running a program, then the following three two-dimensional grids are `1` × `p`, `p` × `1`, and `p/2` × `2` processor layouts, respectively:

```
grid G1 = [1      , 1..p];
     G2 = [1..p   , 1   ];
     G3 = [1..p/2, 1..2];
```

Like regions, grids can be of any rank. Notice the similarity between grid and region literals. Any region literal without flood dimensions is also a grid literal. Figure 3.1 illustrates `G1`, `G2`, and `G3` given eight processors.

> Implementation Note. At the time of this writing, grid dimensions are specified by singleton expressions. The lower bound is zero and the upper bound is the expression less one. For example, `G = [i]` is equivalent to `G = [0..i-1]`.

Specific virtual processors are mapped to grids in row-major order starting with processor zero. It is possible for the programmer to specify both more and less precise grids. Indexed

arrays are used to position particular processors in any order. The `auto` keyword is used to let the compiler decide how many processors to allocate to a given dimension.

3.1.1 Indexed Arrays as Grid Specifications

To assign specific processors to grids, an indexed array may be used as a grid literal. In this case, the shape of the grid is inherited from the shape of the indexed array. The values in the indexed array must be processor IDs, and no repeats are allowed; otherwise a runtime error will alert the programmer that an invalid grid has been created. For example, the following pair of grids are `2` × `2` processor layouts that split the first eight processors into two disjoint sets based on whether the processor ID is even or odd:

```
const a : array[1..2, 1..2] of integer = {{0, 2}, {4, 6}};
      b : array[1..2, 1..2] of integer = {{1, 3}, {5, 7}};
grid G4 = a;
     G5 = b;
```

The ability to create disjoint grids extends the applicability of ZPL into problems traditionally characterized as *task parallel* [Dei03]. Note that for grids `G4` and `G5` the program must be run on at least eight processors. Grids `G4` and `G5` are illustrated in Figure 3.1.

> Implementation Note. At the time of this writing, there is no support for assigning or initializing grids with indexed arrays.

3.1.2 "Auto" Grid Specifications

The keyword `auto` asks the compiler and runtime to heuristically allocate processors to a grid dimension. For example, the following grid declaration uses the `auto` keyword to divide the processors between the first two dimensions of grid `G6`, leaving the last dimension degenerate:

```
grid G6 = [auto, auto, 1];
```

With the `auto` keyword, a grid dimension's lower bound is one and its upper bound is the number of processors allocated to that dimension. The processors are still assigned to the grid in row-major order starting with processor zero. Grid `G6` is illustrated in Figure 3.1, assuming eight processors.

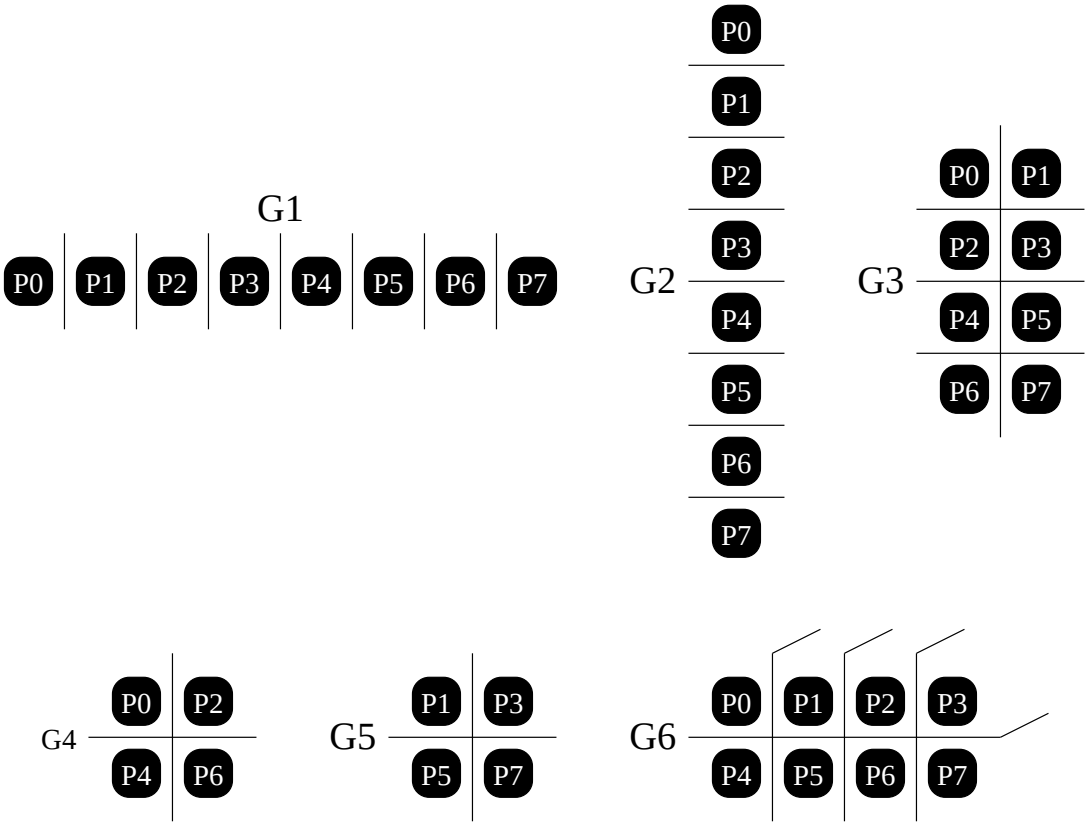

Figure 3.1: An Illustration of Grids `G1` to `G6` from Section 3.1. For illustrative purposes, eight processors are assumed.

The `auto` heuristic divides the processors among the `auto` dimensions as even as possible. A smarter heuristic might analyze the program to determine a processor layout that would minimize communication. Such a smart analysis could be part of the compiler or, more appropriately, could be developed as a separate tool.

> Implementation Note. At the time of this writing, a blank dimension implements `auto` and the `auto` keyword is not recognized.

3.1.3 The Implicit Grids

Regions are always distributed over grids of the same rank, whether the grid is implicitly or explicitly declared. (In the next section, a construct for mapping indexes in regions

to processors in grids will be introduced.) Grids are not a necessary concern of the ZPL programmer. In many cases, heuristically defined grids are sufficient, and programmers do not want to concern themselves with their declarations. Thus the explicit declaration of grids is optional in ZPL. This allows for rapid code development.

For each rank of region used in a ZPL program, an implicit grid of that rank is defined as if it were declared using the `auto` keyword in every dimension. For arrays that are declared over regions that are not explicitly linked to grids, the implicit grid of the rank of the array is used.

> Implementation Note. At the time of this writing, only the implicit grid of maximum rank is defined as above. The implicit grids of lower ranks simply use the first dimensions of the maximum rank implicit grid. For example, if a program needs implicit grids for arrays of rank two and three, and the program is run on 32 processors, then the 3D is `[4, 4, 2]` and the 2D grid is `[4, 4]`, not `[8,4]` as it would be given the above definition.

3.1.4 Grid Types

The syntax used so far for declaring grids is syntactic sugar for constant grid declarations. This syntactic sugar parallels that for regions. Grids have types and values associated with them and can be declared as constants or variables. The following declarations of the six grids introduced in this section retain the same semantics as before:

```
const G1 : grid<.,..>     = [1      , 1..p];
      G2 : grid<..,.>     = [1..p   , 1   ];
      G3 : grid<..,..>    = [1..p/2, 1..2];
      G4 : grid<..,..>    = a;
      G5 : grid<..,..>    = b;
      G6 : grid<..,..,.> = [auto, auto, 1];
```

Notice that the type of a grid is not simply `grid`, but also includes the rank of the grid. In addition, whether the dimension *may* be allocated to more than one processor (`..`) or not (`.`) is part of the type.

While the casual programmer can use "`..`" everywhere, using "`.`" could result in a more optimized program. The type of a grid can be (conservatively) inferred from its initializer; this inference is always done with the syntactic sugar. In this case, the keyword `grid` is a sufficient type specification. Changing the keyword `const` to `var` creates mutable grids that

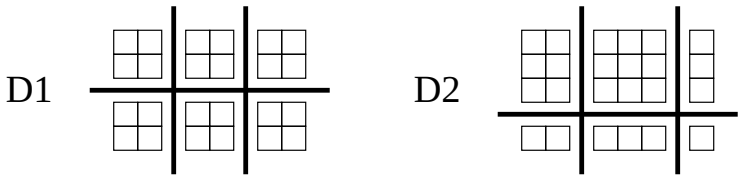

Figure 3.2: An Illustration of Distributions `D1` and `D2` from Section 3.2. For illustrative purposes, take `G` to be a 2 by 3 grid of processors and assume values of 4 and 6 for `n` and `m` respectively and illustrate the 4 by 6 index set over these two distributions. The array `a1` is taken to contain the single integer 3, and array `a2` is taken to contain the two integers 2 and 5.

do not need initializers. The reassignment of variable grids will be discussed in Section 3.4.

3.2 Distributions

The indexes in regions are mapped to the virtual processors in grids by an abstraction called a *distribution*. The distribution is a mapping of logical indexes of some rank to processors in a grid of the same rank. Thus the distribution that maps a region of a given rank to a grid of the same rank must also have the same rank. For example, if `G` is a 2D grid where neither dimension is degenerate, `n` and `m` are integers, and `a1` and `a2` are 1D indexed arrays of integers, then the following distributions map indexes to grid `G` using two built-in distributions:

```
distribution D1 : G = [blk(1,n), blk(1,m)];
              D2 : G = [cut(a1), cut(a2)];
```

The `blk` distribution takes as its arguments upper and lower index bounds and distributes the indexes between these bounds over `G` in a block fashion. The `cut` distribution takes as its argument a one-dimensional indexed array of integers. The indexed array must be defined over the grid dimension's range (excluding the last processor). Its elements, which must be monotonically increasing, contain the highest indexes mapped to the grid processor referred to by its index. Figure 3.2 illustrate distributions `D1` and `D2`.

3.2.1 The Implicit Distributions

Grids and distributions extend ZPL and require the programmer to manage data decomposition. To ease program development, neither grids nor distributions need to be declared. When arrays are declared over regions that are not declared over distributions, implicit grids and distributions are used. The implicit grids were already discussed.

The implicit distributions are defined on a per-rank basis as well. An array declared over a region without a distribution uses the implicit grid and distribution of its rank. The default implicit distribution is defined by using the `blk` distribution function in each dimension. The bounds of the `blk` distribution are the minimum low and maximum high bounds of every constant region of the distribution's rank in that dimension that is declared in the global scope of the program.

> Implementation Notice. At the time of this writing, the implicit distributions take their bounds for the `blk` distribution in each dimension from the minimum low and maximum high bounds of every constant region in that dimension that is declared in the global scope of the program, not just from the regions of the same rank as the distribution.

Note that an implicit grid may be used even if the implicit distribution is not used. This is the case when the distribution that the region is declared over is not declared over a grid. It is useful if the distribution is important but the grid is not. In many cases, the implicit grid is sufficient even if the implicit distribution is not.

3.2.2 Distribution Types

The syntax used so far for declaring distributions is syntactic sugar for constant distribution declarations. Like regions and grids, distributions have associated types and values. Without changing the semantics, the following declarations create the same distributions `D1` and `D2`:

```
const D1 : [G] distribution<block,block> = [blk(1,n), blk(1,m)];
      D2 : [G] distribution<block,block> = [cut(a1), cut(a2)];
```

Notice that the type of a distribution is not simply `distribution`, but also includes the rank of the distribution (which must match the rank of its grid). In addition, each dimension adds to the type of the distribution a *distribution dimension type*. Both `cut` and `blk` are

`block`-type dimensions. Note that the grid is not part of the distribution's type. It is statically linked to the distribution in the same way that a declaring region is statically linked to a parallel array. These static links will be discussed in Section 3.3. As with grids, changing the keyword `const` to `var` creates distributions that may be changed and need not be initialized.

There are five distribution dimension types: `block`, `cyclic`, `multiblock`, `nondist`, and `irregular`. The distribution dimension type determines the code that the compiler needs to generate. The types have been chosen because they capture the different types of code that the compiler would have to generate. Within each type, the compiler can generate efficient code for the distributions that fall into its type. For example, the compiler can generate the same code for the `blk` and `cut` distributions without compromising their efficiency.

> Implementation Note. At the time of this writing, only the `block` distribution dimension type is supported.

Though the other distribution dimension types are not implemented, it is worthwhile mentioning them. The `cyclic` type implements the standard cyclic distribution instantiated by the built-in distribution `cyc`. The `multiblock` distribution function type implements distributions like block-cyclic with `blkcyc`. It is also easy to image a cut-cyclic distribution implemented with `cutcyc` of the `multiblock` type. The `nondist` distribution function type with the `null` distribution optimizes the degenerate case where the grid dimension is allocated to only one processor. Lastly, the `irregular` distribution type is a catch-all for distributions that do not fall into the other categories. Another type, or set of types, is necessary to capture distributions that are not dimension-orthogonal, *e.g.*, recursive coordinate bisection.

By separating a distribution's type from its value, the amount of code needed to add new distributions to a given type is limited. This opens the possibility for supporting user-defined distributions. In adding the `cut` distribution, as well as several other `block` type distributions, it became apparent that the code in the runtime needed to add a new distribution is small. For example, code for the cut distribution is listed and discussed in Appendix A.

3.3 Grid-Distribution-Region-Array Hierarchy

Grids and distributions are fully integrated with regions. A region's distribution is controlled through the distribution it is declared over. Regions are bound to distributions in the same way that distributions are bound to grids. For example, the following declarations define two `n` × `n` regions, one bound to `D1` and one bound to `D2`:

```
region R1 : D1 = [1..n, 1..n];
       R2 : D2 = [1..n, 1..n];
```

Without the syntactic sugar, they can also be declared as follows:

```
const R1 : [D1] region = [1..n, 1..n];
      R2 : [D2] region = [1..n, 1..n];
```

To declare parallel arrays over `R1` and `R2`, a programmer writes

```
var A : [R1] double;
    B : [R2] double;
```

This creates a static hierarchy from grids to distributions to regions and finally to arrays. In this hierarchy, the elements of an array are associated with the indexes of a region which are mapped to a processor in a grid via a distribution.

3.3.1 Distribution Change

It is a runtime error to use arrays with different distributions in the same statement. Such statements could result in large amounts of communication which, if allowed, would violate ZPL's performance model. In simple cases, the compiler can warn the programmer about the likelihood of this runtime error. More complicated cases arise when grids and distributions are dynamically changed using the special assignment operators discussed in Section 3.4.

Since `A` and `B` are distributed using different distributions, they cannot interact without explicit use of the remap operator. The statement

```
[R1] A := B#[Index1, Index2];
```

copies the values in `B` to `A` preserving the logical positions of the values. This movement of data, from arrays of one distribution to arrays of another, is called *distribution change*.

Recall that the default ith map arrays can be elided, making the following code equivalent:

```
[R1] A := B#[,];
```

Note that rank change is a special case of distribution change. In rank change, data is moved from an array of one rank to an array of another rank. Since arrays of different ranks necessarily have different distributions, rank change also results in a distribution change. This is why it has always been impossible to use arrays of different rank in the same statement without using the remap operator. Rank change is simply a special case of distribution change.

3.3.2 Gridless Distributions and Distributionless Regions

Unlike the region-array binding, the grid-distribution and distribution-region bindings are optional. When a distributionless region is used to control computation, the distribution is inherited from the arrays involved. For example, in the statement

```
[1..n, 1..n] A1 := A2;
```

the anonymous region inherits its distribution from either array `A1` or `A2`. They must have the same distribution.

Gridless distributions and distributionless regions do not need to inherit a single distribution. That is, different statements can force the region to inherit different distributions. It is worthwhile to elaborate this note. For example, consider the statement block:

```
[1..n, 1..n] begin
  A1 := A2;
  A3 := A4;
end;
```

Arrays `A1` and `A2` must have the same distribution, and arrays `A3` and `A4` must have the same distribution. However, these distributions can be different. The compiler is thus limited on the optimizations it can apply to these statements. For example, the statements could not be fused.

In an alternative design, a distributionless region scope could require all of its statements to use the same distribution. This would enable more optimizations, but the code that the

ZPL programmer writes would be more complicated. Specifically the programmer would have to open the identical region multiple times if the distribution was changing. For example, this would make the grid implementation of the NAS FT benchmark described in Section 6.2.3 needlessly complicated by requiring the programmer to open the same anonymous region scope many times.

> Implementation Note. At the time of this writing, the implementation is based on this alternative design.

3.3.3 Parallel Array Types

Grid types, distribution types, and region types consist of their rank and the type of each of their dimensions. Though distributions and regions are statically linked to grids and distributions, the types of those grids and distributions are not part of the types of the distributions and regions. The type of a parallel array, however, does consist of the type of its declaring region, distribution, and grid. The compiler needs to know these attributes of an array for the cases where the array determines the distribution of the current region. It is not necessary for a programmer to explicitly define the type of an array when it is a parameter to a procedure, but the implementation must clone on the differences.

> Implementation Note. At the time of this writing, no cloning is done. Separate compilation would also pose problems in how procedure prototypes are defined. There is currently no support for separate compilation.

3.4 Assignment of Grids, Distributions, and Regions

The mutability of grids, distributions, and regions provides great power and flexibility. Their assignment is slightly different from assignment of, say, integers because changes propagate down the hierarchy. For example, changing a grid would result in changes to distributions, regions, and arrays declared over that grid. To make this distinction clear, two new assignment operators are introduced. These assignment operators are called *preserving assignment* (`<=#`) and *destructive assignment* (`<==`).

As the name suggests, in preserving assignment the data in the arrays is preserved. That

is, if a position in the original array still exists in the new array, the data at that position in the original array will be in that position in the new array. Positions in the new array that did not exist in the original array are uninitialized. Data values in positions in the original array that do not exist in the new array are lost.

In destructive assignment the data values in the arrays are destroyed. The advantage to destructive assignment is speed. Preserving assignment potentially requires significant communication to shuffle the data between processors. This communication could be all-to-all communication. It fits in with ZPL's performance model since the remap operator (`#`) is embedded in the preserving assignment operator (`<=#`) but not in the destructive assignment operator (`<==`). Note that communication is only necessary in preserving assignment of grids and distributions. When preserving assignment is applied to regions, data is moved around within processors but not between them.

Figure 3.4 shows examples of applying preserving and destructive assignment statements to a simple hierarchy. In the hierarchy, there is a single grid and distribution. Two regions, `R` and `S`, are declared over the distribution. The single array `A` is declared over region `R` and the two arrays `B` and `C` are declared over region `S`.

The first two lines of code initialize arrays `A` and `B`. Array `C` remains uninitialized, as indicated by the question marks. The third line of code copies the data from `A` to `C`. Note that because `A` and `C` have the same distribution, they can interact. If not, the remap operator would need to be used to move data though the maps could be left blank to indicate that no logical data change is happening.

The fourth line reassigns the value of region `R`. It makes `R` smaller by deleting the last row and column. Because destructive assignment is used, the data in `A` is lost. It becomes equivalent to data that is uninitialized. The fifth line reassigns the value of region `S`. It deletes the last row and column too. Note that the literal value does not have to be used all the time and this is equivalent to the statement `S <=# R`. In this statement, because preserving assignment is used, the data in arrays `B` and `C` is preserved.

The final line of code reassigns the value of the distribution `D`. Because destructive assignment is used, the data in all of the arrays is destroyed and thus uninitialized.

```
var G : grid = [auto, auto];
    D : [G] distribution = [blk(0,n-1), blk(0,n-1)];
    R : [D] region = [0..n-1, 0..n-1];
    A : [R] integer;
    S : [D] region = [0..n-1, 0..n-1];
    B, C : [S] integer;

[R] A := Index1 * n + Index2;
[S] B := 0;

      A =  0  1  2  3   B =  0  0  0  0   C =  ?  ?  ?  ?
           4  5  6  7        0  0  0  0        ?  ?  ?  ?
           8  9 10 11        0  0  0  0        ?  ?  ?  ?
          12 13 14 15        0  0  0  0        ?  ?  ?  ?

[S] C := A;

                                          C =  0  1  2  3
                                               4  5  6  7
                                               8  9 10 11
                                              12 13 14 15

R <== [0..n-2, 0..n-2];

      A =  ?  ?  ?
           ?  ?  ?
           ?  ?  ?


S <=# [0..n-2, 0..n-2];

                         B =  0  0  0      C =  0  1  2
                              0  0  0           4  5  6
                              0  0  0           8  9 10


D <== [blk(0,n-2), blk(0,n-2)];

      A =  ?  ?  ?      B =  ?  ?  ?      C =  ?  ?  ?
           ?  ?  ?           ?  ?  ?           ?  ?  ?
           ?  ?  ?           ?  ?  ?           ?  ?  ?
```

Figure 3.3: Examples of Preserving and Destructive Assignment

Listing 3.1: Computational Zoom in ZPL

```
1  const G : grid = [auto, auto];
2        D : [G] distribution = [blk(1,nx), blk(1,ny)];
3        R : [D] region = [1..nx, 1..ny];
4  var A : [R] double;
5      DZoom : [G] distribution<block,block>;
6      RZoom : [DZoom] region<..,..>;
7      Zoom : [RZoom] double;
8      x1, x2, y1, y2 : integer;

10 [R] computeApprox(A, x1, x2, y1, y2);
11 DZoom <== [blk(x1,x2), blk(y1,y2)];
12 RZoom <== [x1..x2, y1..y2];
13 [RZoom] Zoom := A#[,];
14 [RZoom] computeMore(Zoom);
15 [x1..x2, y1..y2] A := Zoom#[,];
```

3.4.1 Computational Zoom

Recall the problem called Computational Zoom introduced at the beginning of this chapter. It required the creation of a dynamic region and distribution which the extensions just presented make possible.

Listing 3.1 shows a ZPL solution for Computational Zoom. This code starts with declarations for grid `G`, distribution `D`, region `R`, and array `A`. It then adds declarations for variable distribution `DZoom`, variable region `RZoom`, and array `Zoom`. The more precise solution will be calculated on array `Zoom`.

The approximate solution is computed as in the naive code at the beginning of the chapter. The next two lines differ, however, and they set up the smaller region. When these lines are executed, the array `Zoom` is allocated and distributed across the processors. The data is moved into `Zoom` using the remap operator in the next line. Sine the area in `R` that corresponds to `RZoom` may only exist on a subset of the processors and/or be unevenly distributed, a potentially large amount of data may need to be transferred between processors. After procedure `computeMore` is called to compute a more precise solution, the remap operator is used again to move the data back from `Zoom` to `A`. Notice that region `RZoom` is not used since distribution `D` needs to apply.

3.4.2 Multiple Assignments

It is a common idiom to change more than just one of an array's region, distribution, and grid at the same time. An implementation in which these were changed sequentially, as specified by the programmer, would be highly inefficient. Indeed, by doing it sequentially, given a realistic machine with a finite memory, certain changes could even be impossible. For example, consider the declarations

```
var D : distribution = [blk(1, nx), blk(1, ny)];
    R : [D] region = [1..nx, 1..ny];
```

in which a rectangular region `R` is declared over a distribution `D` so that the dimensions are distributed using a balanced block distribution. If the following two lines were applied sequentially, in either order, there is potential for an out-of-memory error if `nx` is not equal to `ny`:

```
D <== [blk(1, ny), blk(1, nx)];
R <== [1..ny, 1..nx];
```

Figure 3.4 illustrates this memory problem. It shows that applying either statement alone could result in an out-of-memory error because the allocated data is too great on a single processor. While it is always possible to work around such a difficulty, as in the code given by

```
R <== [1, 1];
D <== [blk(1, ny), blk(1, nx)];
R <== [1..ny, 1..nx];
```

it is unreasonable to require the programmer to write such rigmarole. Moreover, in the case of preserving assignment, sequentially applying the statements could result in excessive inter-processor communication. If both the grid and distribution are changed, it is best to move data only once.

To avoid the memory problem, and in addition, to achieve better performance overall, consecutive changes are aggregated by the compiler and runtime. In this aggregation, the arrays act as spectators during the execution of the statements. Then, after the grids, distributions, and regions have been updated, the arrays are updated to reflect the changes. In destructive assignments, the arrays are simply reallocated. In preserving assignments,

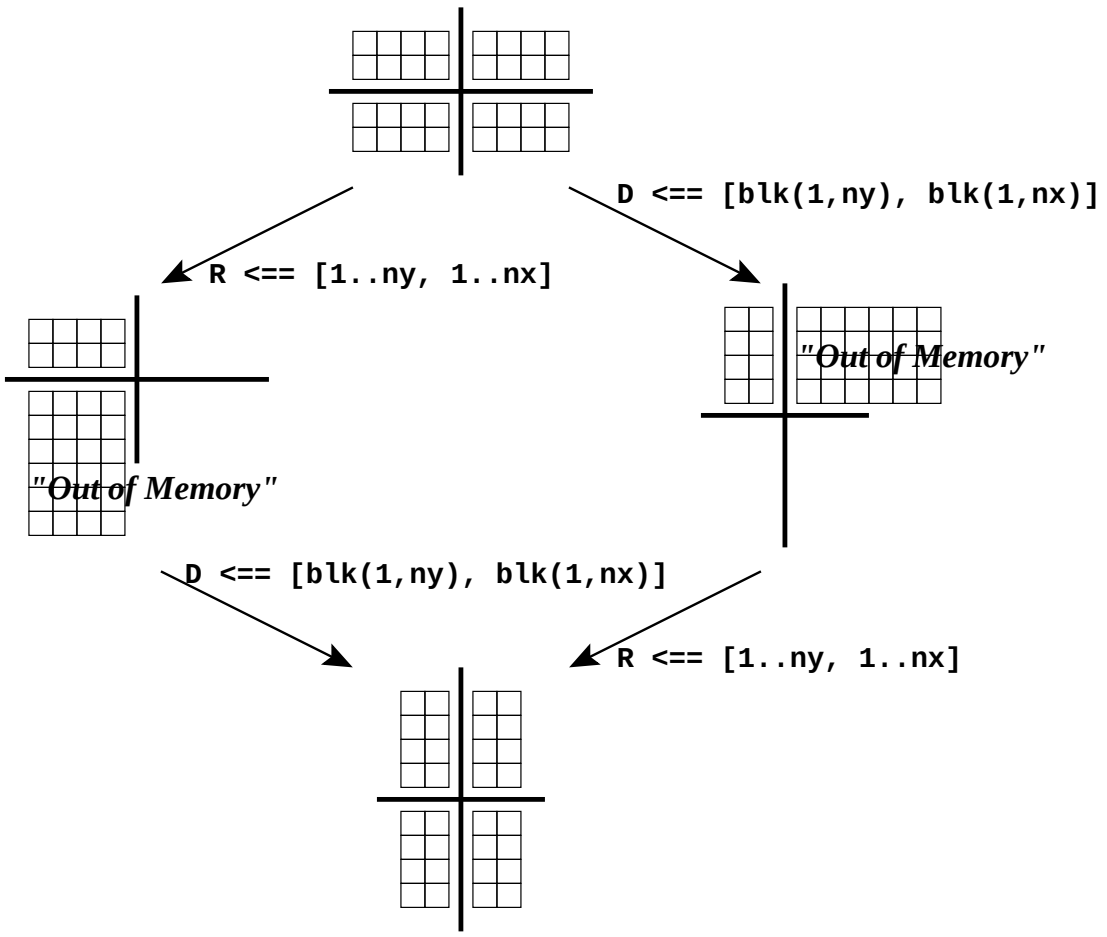

Figure 3.4: A Motivating Example for Aggregating Multiple Hierarchy Changes. In this illustration, a distribution and region are destructively changed. It is seen that an out-of-memory error would occur if either destructive assignment is done without the other.

data is shuffled around, possibly between processors.

3.4.3 Preserving Assignment and the Remap Operator

Preserving assignment and the remap operator are related in more than just syntax. They share the same implementation. The compiler generates a procedure for copying data between two instances of any parallel array that may have its data changed by a preserving assignment statement. This procedure simply moves the data between two arrays using the remap operator. The runtime implementation of preserving assignment then calls this

function to copy the data. The region over which the data is copied is taken to be the intersection of the old region and the new region.

Although originally seen as an easy way to implement preserving assignment, this approach has two important advantages and is likely to remain. First, by relying on code the compiler already generates, the runtime is kept small. This has great advantages for porting ZPL to different computer systems. Second, optimizations done for the remap operator are applied to preserving assignment as well. Some of these optimizations are discussed in the literature [DCCS03].

3.5 WYSIWYG Performance and P-Independent Semantics

This section discusses exceptions to ZPL's WYSIWYG performance model and p-independent semantics. They are considered benign exceptions that should be noted, but that do not adversely impact the properties of the language.

3.5.1 Communication in Destructive Assignment of Grids under MPI

In the MPI implementation of ZPL, there is communication when grids are assigned via the destructive assignment statement. In the MPI implementation, each grid contains an MPI communicator that consists of all the processors in the grid. The grid's communicator is initialized via the MPI routine `MPI_Comm_split`. This routine induces communication and results in a global synchronization.

One way to get around this communication is to not initialize the MPI communicator until it is needed by a parallel operator. In short, it is flagged as uninitialized. This would also prove problematic to ZPL's performance model since it would cause global synchronization even in the case of the @ operator, which does not require global synchronization.

In other implementations, this excess communication is unnecessary. Indeed, if MPI provided another way to create communicators, *e.g.*, by having each processor specify all the processors that are involved, then this communication could be avoided even in the MPI implementation.

3.5.2 Communication in Destructive Assignment of Sparse Arrays

Destructive assignment can induce communication when it applies to a grid or distribution that has a sparse child region. When a grid or distribution is changed via destructive assignment and its hierarchy contains a sparse region, communication is required to rearrange the sparsity pattern. It is a very similar rearrangement as the one induced by preserving assignment for dense arrays. For this reason, destructive assignment on sparse arrays requires roughly the same amount of time as preserving assignment since the values in the array can piggy-back the sparse index structure.

Destructive assignment can also induce communication when it applies to regions because processors that own pieces of sparse regions also keep track of some of the indexes on neighboring processors. When a region changes, these indexes may change and the processors need to keep them up-to-date.

3.5.3 P-Dependent Errors in Grid and Distribution Setup

The grid and distribution constructs are said to be p-independent even though they directly involve the processors. This is because the only p-dependent behavior is simple and straightforward. If a grid or a distribution is setup incorrectly, *e.g.*, more processors than exist are allocated to a grid, a runtime error will occur. There is no possibility that, barring these errors, changing the number of processors will produce the wrong results or cause other runtime errors.

This is substantively different from the concerns of p-dependent constructs. For example, incorrectly using the `INDEPENDENT` directive in HPF would result in an incorrect answer with no indication that something went wrong.

3.6 Related Work

High Performance Fortran provides data mapping directives for handling dynamic data distributions that are also orthogonal to the processors. These data mapping directives are central to HPF and they differ significantly from the abstractions discussed in this chapter. The simplest significant difference is that the directives in HPF are only suggestive. In ZPL,

grids and distributions are imperative.

HPF's `PROCESSORS` directive creates processor arrangements that correspond to ZPL's grids. HPF is only required to accept processor arrangements of a single processor or processor arrangements that include exactly the number of processors returned by the HPF `NUMBER_OF_PROCESSORS()` procedure (equivalent to ZPL's `numLocales()` procedure). Moreover, HPF's processor arrangements are not first class and cannot be changed. In ZPL, the grid only need not exceed the value returned by `numLocales()` and its value can be changed.

HPF provides two directives for specifying how data is mapped across processor arrangements: `ALIGN` and `DISTRIBUTE`. They correspond to the implicit links between regions and distributions and between distributions and grids. The `ALIGN` directive lets the HPF programmer advise the compiler to distribute multiple arrays in the same way. In a group of arrays that are aligned to one another, the distribute directive is applied to one of these arrays and the other arrays can follow this distribution. Performance is improved in HPF programs by aligning arrays, though because it is just advisory, arrays that are specified as being aligned may not actually be aligned.

In ZPL, all arrays that are declared over regions that are declared over the same distributions or equal distributions are aligned. HPF's alignment possibilities are more complicated because HPF does not have a performance model comparable to ZPL's. Thus HPF has a long list of legal alignment expressions. For example, the `M` $\times$ `N` arrays `X` and `Y` are aligned to one another by reversing both axes with the following odd alignment statement:

```
!HPF$ ALIGN X(J,K) WITH Y(M-J+1,N-K+1)
```

HPF's `DISTRIBUTE` directive is similar to ZPL's `distribution` abstraction. It maps an index set given by an array or a group of arrays to a processor arrangement. One of the interesting differences between ZPL and HPF is in the choice of the default distributions. In HPF, arrays without specified distributions are assumed to have their own distribution over their own processor arrangement. This is in sharp contrast to ZPL's per-rank implicit distributions and grids. Since HPF does not have any requirements that communication be easy to identify, there is no need for this restriction.

HPF's `ALLOCATABLE` arrays are similar to arrays in ZPL whose regions have not been initialized even if their distributions and grids have. When the HPF array is allocated, the array is distributed. Unlike in ZPL, care must be taken in HPF not to allocate an array that is aligned to another array that has not yet been allocated. In ZPL, grids, distributions, and regions can be initialized in any order.

Procedure calls in HPF are complicated by the distributions of arrays. Unlike in ZPL, data may be automatically remapped by either the caller or the callee. In ZPL, there is no automatic remapping and it is a runtime error to have two arrays with different distributions interact.

It is tempting to compare HPF's `TEMPLATE` directive with ZPL's distribution and region abstractions, but this would be misleading. Like a region, the template is an abstract index set, an "array of nothings" as the HPF manual describes it. Its similarity to the region, however, ends there. Like a distribution, the template is targeted by arrays that are aligned to it. But its similarity to distributions ends there. The template is merely an abstract target for the `ALIGN` directive.

Although HPF has no abstractions that correspond to ZPL's distributions and regions and even though HPF's processor arrangements cannot be changed like ZPL's grids can be, HPF still provides a mechanism for changing data distributions. As approved extensions to HPF 2.0, the directives `REDISTRIBUTE` and `REALIGN` let HPF programmers arbitrarily change the data distributions of arrays.

The mechanisms for data distribution and processor layout in HPF and ZPL differ primarily because the languages to which they were added differ. HPF added directives to the sequential language Fortran. Without a parallel performance model, communication induced by different distributions was not an issue. Indeed, if programmers do not concern themselves with this issue, it is easier to write programs. (It is almost as easy as it is to write sequential programs.)

Given ZPL's central abstraction of the region, it makes sense that more abstractions could extend ZPL to handle processor layout and data distribution. With this cleaner set of abstractions, ZPL's mechanisms are easier to use. With the extra benefit of its performance model, efficient code is easier to write. That said, alignment options are limited in ZPL.

Whether this is a problem or not remains to be seen. In addition, for parts of a parallel code that are not performance-critical, there is no way in ZPL to easily write the code. This concern is partially addressed in this thesis's discussion on future work in Chapter 7.

3.7 Summary

This chapter presented two abstractions for managing virtual processor layout and data distribution in ZPL: grids and distributions. Grids are processor layouts and distributions are mappings from indexes in regions to processors in grids. Both of these abstractions extend ZPL by creating a hierarchy of abstractions that also includes regions and parallel arrays. Two new assignment operators for changing grids, distributions, and regions enable support for dynamic data distributions and processor layouts.

The power of these abstractions is demonstrated in Chapter 6 where they are used to implement the NAS FT and IS benchmarks. These are shown to be cleaner and more high-level than the MPI code provided by NAS. Moreover, performance is shown to be comparable.

By providing high-level, p-independent abstractiosn, it is easy to use and experiment with different data distributions. It is a worthwhile exercise to write, or imagine writing, the computational zoom example in MPI. The benefits of ZPL is then more clear. In the MPI code, the programmer is responsible for shuffling data between the processors.

Chapter 4

SUPPORT FOR PROCESSOR-ORIENTED PROGRAMMING

This chapter presents two low-level p-dependent abstractions: free scalars and grid dimensions. In sharp contrast to other ZPL abstractions, the programmer using free scalars or grid dimensions writes per-processor SPMD code. These abstractions are shown to extend ZPL semantics naturally and to behave more simply than similar constructs in other languages. In addition, they are shown to be surprisingly versatile.

This chapter is organized as follows. The next two sections present free scalars and grid dimensions respectively. Section 4.3 discusses a simple interpretation of parallel types in ZPL called parallel type coercion. Section 4.4 discusses some compiler transforms and optimizations implemented by the ZPL compiler that, given free scalars and grid dimensions, can now be seen as source-to-source transformations. Although it is possible for programmers to write these codes, they are not encouraged to unless the case is too complicated for the compiler. Section 4.5 discusses two more complicated source-to-source transformations for implementing multi-dimensional scans. Section 4.6 shows an example of using free scalars to implement a fast pseudorandom number generator. This example is taken from the initialization stage of the NAS parallel benchmarks. The compiler is shown to be helpful to programmers adding free qualifiers to their code. Section 4.7 presents advanced timing routines made possible with the introduction of free scalars. Section 4.8 shows how it is possible to call MPI routines from ZPL programs even if it is undesirable. Section 4.9 discusses related work and Section 4.10 concludes with a summary.

4.1 Free Scalars

In ZPL, scalars are replicated on every processor. The typechecking scheme ensures that these scalars always contain the same values on every processor. For example, there is no way to assign a value in a distributed parallel array on some processor to the scalar on that

processor. Reductions that assign values from parallel arrays to scalars always assign the same value to all the scalars.

Free scalars in ZPL are similar to regular scalars in SPMD languages like Co-Array Fortran, Titanium, UPC, and Fortran+MPI. However, their usage in ZPL is more limited. These limitations maintain ZPL's high-level semantics as well as its performance model of syntactically identifiable communication. The advantages and differences are discussed more in Section 4.9. Free variables provide two key advantages over the defaults of the lower-level SPMD languages: (1) the programming model is simpler since race conditions and deadlocks are impossible and (2) code can be easily optimized by the programmer and the compiler since communication is identifiable in the syntax of the code.

Free scalars are declared with the `free` qualifier. For example, the declaration

```
free var lsum : integer;
```

creates a free integer variable `lsum`. Free scalars are replicated across the processors and may contain different values on different processors. Type checking ensures that scalars remain consistent in the presence of free scalars. For example, free scalars may not be assigned to scalars. A parallel type coercion hierarchy, discussed in Section 4.3, formalizes these rules.

Note that the keyword `free` qualifies `var` and not the type. This decision was made since `free` only makes sense when it is in the outermost position of structured types. Logically, a free array of integers has the same semantics as an array of free integers. Whether this was the correct decision or not, it is of little importance. In Titanium, for example, their related keyword of `single` (described further in Section 4.9) qualifies a type and is defined to be outward leaning.

Despite the restrictions on free scalars, they are surprisingly robust. There are no restrictions on writing to free scalars. Assigning values to free variables within shattered control flow and from parallel arrays are common idioms. Additionally, free scalars, like regular scalars, may be assigned to parallel arrays. These idioms allow for codes that were previously impossible to write.

It is important to note that statements assigning values to free scalars do not have array

semantics. For example, in the following code the free scalar `lsum` accumulates the sum of the elements in `A` that are local to the processor on which `lsum` resides:

```
[R] lsum += A;
```

The right hand side is not evaluated first. Instead, for every index in `R`, the value in `A` is added to `lsum`. To avoid nonsense, restrictions are placed on the code that can be written with free variables. Specifically, a free scalar that is assigned in a statement or interleaved block of statements may not be promoted in that statement or interleaved block of statements.

4.1.1 Parallel Operators and Free Scalars

The parallel operators apply to free scalars without promoting the free scalars. If, however, the expression to which the parallel operator applies requires the free scalar to be promoted, it is. As with the application of the parallel operators on parallel arrays, the parallel operators applied to free scalars cannot be used within shattered control flow, interleave blocks (except @, wrap @, and prime @) and within promoted procedures.

The @, Wrap @, and Prime @ Operators

The @, Wrap @, and Prime @ operators apply to free scalars in a natural way. The direction must be one-dimensional and it refers to a number of processors. For example, the statement

```
lsum := lsum@[-1];
```

assigns to processor i's copy of `lsum` the value in processor $i-1$'s copy of `lsum`. Processor 0 retains its original value. In this case, and this case only, reading or writing off the edge of the processors is automatically guarded against. While this may not be orthogonal or logical, it is pragmatic. Indeed, this useful operator would otherwise be useless.

Both the wrap @ and prime @ operator are similarly extended. The wrap @ operator cyclically shifts the values in free scalars and the prime @ operator induces one-dimensional sequential wavefronts across the processors.

Implementation Note. At the time of this writing, the @, Wrap @, and Prime @ operators are not implemented over free variables.

The Reduce Operator

The reduce operator can also be applied to free scalars. In this case, the reduction is calculated over the values in every processor. For example, the code

```
lsum := 0;
[R] lsum += A;
lmax := max<< lsum;
```

determines the maximum local sum of the values in `A`. Note that `lmax` does not need to be a free scalar. It is, however, a decidedly p-dependent value. A counterpart to the partial reduction does not exist (because there is no sense of dimensionality). Only full reductions are supported over the free scalars.

The Scan Operator

The scan operator can compute the prefix sums over the processor IDs going from low ID to high ID or high ID to low ID depending on whether the single value in the dimension list is -1 or 1. For example, a max scan across the local sums given by

```
lsum := 0;
[R] lsum += A;
lmax := max||[1] lsum;
```

would require `lmax` to be free. The values in `lmax` are assigned the maximum local sum of any processor ID lower than it.

Implementation Note. At the time of this writing, the scan operator is not implemented over free variables.

The Flood Operator

The flood operator takes a processor ID and replicates the value at that processor ID across all the processors. For example, the code

```
z := >>[0] lval;
```

replicates the value of free variable `lval` on processor zero and stores the result on all processors in regular variable `z`.

> Implementation Note. At the time of this writing, the flood operator is not implemented over free variables.

The Remap Operator

The remap operator takes a single map when applied to free scalars. The map specifies a processor ID that identifies a copy of the free scalar on a possibly different processor. For example, the code

```
lsum := lsum#[(localeID() + 1) % numLocales()];
```

cyclically shifts the values in `lsum`. Note that for this computation, the wrap @ operator is a better choice.

4.1.2 Free Regions

A free region is a region whose bounds are free. Free regions may only contain data local to a processor. Parallel operators may not be used in statements to which free regions apply. Free regions may be opened within free control flow (see below), but not within shattered control flow.

Because of the limitations on free regions, they are more of a convenience. When activated, they have the semantics of free control flow in the form of loops.

4.1.3 Free Procedures

Parameters to procedures may be qualified as free. A regular scalar cannot be passed as `inout` or `out` if the parameter is qualified as free since this would constitute assigning a free scalar to a regular scalar.

The introduction of free scalars changes how procedure promotion is defined. Procedure promotion was previously limited to scalar procedures without side effects. This limitation is too broad. Side effects to free variables are allowed. The only reason side effecting procedures were not allowed to be promoted was so that scalars would be kept consistent.

Free variables do not need to remain consistent. Note that other side effects, such as input or output, still disable promotion.

Similar to promotion, i.e., scalar procedures without side effects (except free side effects), may also be *relaxed*. That is, free variables may be passed to scalar parameters. Just like in promotion, however, all of the parameters that are written must be free.

4.1.4 Free Control Flow

It is legal to write to free variables in shattered control flow. In addition, free variables may be written in *free control flow*. Free control flow is control flow that is relaxed by free variables. In general, the same rules that apply to shattered control flow apply to free control flow. Parallel operators may not be used, scalars may not be written, and region scopes (except free region scopes) may not be opened.

4.1.5 An Expanded Standard Context

The procedure `localeID`, introduced in the last Chapter, was deliberately not defined. The reason for this can now be made clear. The `localeID` procedure returns a free integer. Each processor by definition has a different number. On the other hand, the procedure `numLocales`, introduced at the same time, was defined. This procedure returns a single integer value on every processor.

The distinction between `numLocales` and `localeID` is an important one. Both introduce p-dependent values into the flow of a program, but only the latter one introduces free values.

ZPL provides a set of intrinsic procedures, called the *standard context*, that are automatically included in every ZPL program. With the addition of free variables, it is beneficial to expand the standard context with procedures for querying the processors, grids, and distributions. Examples of these new procedures are listed here: (Generic entities refer to grids, distributions, regions, and arrays where the supplemental type information is irrelevant.)

Querying Grids and Locales

```
procedure numLocales() : integer;
procedure numLocales(G : GenericGrid) : integer;
procedure numLocales(G : GenericGrid; dim : integer) : integer;
```

The `numLocales` procedure is overloaded based on its arguments in much the same way that `write` is overloaded. Functions in the standard context can be overloaded, even if overloading is illegal in ZPL. When passed zero parameters, the `numLocales` procedure returns the number of locales executing the program. When passed a grid, the procedure returns the number of locales in the grid. When passed a grid and a dimension, the procedure returns the number of locales in the dimension of the grid.

```
free procedure localeID() : integer;
free procedure localeID(G : GenericGrid; dim : integer) : integer;
```

When passed zero parameters, the `localeID` procedure returns the rank of the locale, an integer value between zero and one less than the number of locales. When passed a grid and a dimension, the procedure returns the location in the dimension of the grid.

Querying Block-Type Distributions

```
free procedure blockLocalLo(X : GenericArray; dim : integer) : integer;
free procedure blockLocalLo(X : GenericRegion; dim : integer) : integer;
```

Given a block distribution in dimension `dim`, the procedure `blockLocalLo` returns the local lower bound of an array's declaring region or a region.

```
free procedure blockLocalHi(X : GenericArray; dim : integer) : integer;
free procedure blockLocalHi(X : GenericRegion; dim : integer) : integer;
```

Given a block distribution in dimension `dim`, the procedure `blockLocalHi` returns the local upper bound of an array's declaring region or a region.

There are many other procedures for querying grids, distributions, regions, and arrays that should be added to the standard context. They are not explored further here.

4.2 Grid Dimensions

Grid dimensions analogically unify the concepts of flood dimensions, free scalars, and scalars. Indeed, they can be called free flood dimensions, being to flood dimensions as free scalars are to scalars and being to free scalars as flood dimensions are to scalars. A unified parallel type coercion hierarchy, discussed in Section 4.3, makes these analogies clearer.

Denoted by `::` instead of `*`, a grid dimension associates a single value with *each* processor rather than with all processors. Like flood dimensions, grid dimensions conform to any indexes. Grid arrays, arrays with grid dimensions, can be read over any region (with the same distribution) and return the value associated with that processor.

4.2.1 Non-Grid Arrays and Grid Dimensions

Arrays declared over regions with a range dimension cannot be read over regions with the same dimension flooded. However, if the dimension is a grid dimension rather than a flood dimension, the array can be read. A value is chosen from the parallel array's values by indexing into the parallel array as if it were a parallel array of indexed arrays declared over a region with a grid dimension. The indexes used to index into the indexed array are the global indexes of the position in the array if it were read over a singleton dimension.

For example, the code

```
[0..n-1, ::]
  for i := blockLocalLo(A, 2) to blockLocalHi(A, 2) do
    A[i] := A[blockLocalHi(A,2) - i + blockLocalLo(A, 2)];
  end;
```

reverses the order of the local elements in the rows of `A`. If the second dimension is known to be local to a processor, *i.e.*, its grid allots the processors to the first dimension, then more meaningful computations can be written. For example, the rows can be sorted or an FFT can be applied to the rows.

Parallel arrays can be indexed into over a range region as well. The `::` operator can be

applied to any parallel array in the same way that the remap operator can be. In the case of `::`, however, the indexes used to read the parallel array must be local to the processor. For example, the code

```
[0..n-1, 0..n-1]
  A := A::[ , blockLocalHi(A,2) - Index2 + blockLocalLo(A, 2)];
```

also reverse the order of the local elements in the rows of `A`. Note that the elided index is taken straight from the region and is equivalent to Index1.

4.2.2 Region Operators and Grid Dimensions

The region operators interact naturally with grid dimensions. Unlike with flood dimensions, for which the region never changed, the region operators do change grid dimensions. Just like the `by` preposition was useful for defining regions that could not otherwise be created (strided regions), all of the prepositions change regions with grid dimensions in ways that are otherwise impossible.

The `in`, `by`, and `-` prepositions are the most useful. The `in` preposition creates a new region on the inside of the region to which it applies. The direction counts out how many processors to keep in the region. The `by` preposition is useful for striding a grid dimension. The direction indicates the number of processors to skip. The `-` preposition creates the opposite region as `in`. It removes border processors from a grid dimension.

The `of`, `at`, and `+` prepositions should never be applied to a grid dimension that was not previously altered with `in`, `by`, or `-`. Otherwise, the region that is being created would extend the parallel region over processors that do not exist.

The prepositions are useful for creating regions over which the parallel operators may apply. The next section presents some examples.

4.2.3 Parallel Operators and Grid Dimensions

The @, Wrap @, and Prime @ Operators

The @, Wrap @, and Prime @ operators apply when the compute region is a grid dimension. The directions are interpreted over the processors in the grid dimension. The directions have

a similar interpretation as with the prepositions. For example, the statement

```
var AG : [::] integer;
[:: - east] AG := AG@east;
```

shifts the data in the array. Note the use of the - preposition to guard against reading off the edge of the grid. It is possible to apply the @ operators to non-grid arrays. These arrays are indexed into as per the discussion above.

The Reduce Operator

The reduce operator is extended to support grid dimensions in either the source or destination region. If a grid dimension is used in the destination region, a grid dimension must be used in the source region. There is no reduction of values in this case. On the other hand, if a grid dimension is used in the source region, the dimension may be reduced to either a singleton or flood dimension. The per-processor values are then reduced.

The Scan Operator

The scan operator can apply to regions with grid dimensions without a significant change to the semantics. Whereas it was ignored in the flood dimension case because there was a single value in that dimension, it applies to the per-processor values of grid dimensions. In this case, the per-processor values along that dimension are simply scanned.

The Flood Operator

The flood operator is extended to support grid dimensions in either the source or destination region. The semantics are the opposite of the reduce semantics. A grid dimension source region implies a grid dimension destination region and no flood. A grid dimension in the destination region, on the other hand, may be the result of a flood of the value in a singleton dimension.

The Remap Operator

As with flood dimensions, the remap operator supports more interesting extensions. If the region is a grid dimension, this only impacts the map arrays and non-remapped arrays. If they are not grid arrays, they must be indexed into. The maps, however, apply to the remapped array in the normal way. For example, the statement

```
[R] A := AG#[Index1];
```

moves values in `AG` to `A`. The values in `AG` are specified by the indexes in `Index1` as they map to processors in `AG` based on the distribution. They do not specify processor numbers in the grid.

The interesting changes show up when the remapped array has grid dimensions or when the programmer wants to index into the remapped array as if it were declared over grid dimensions using the extension discussed in Section 4.2.1. This parallels the flood maps discussed in Section 2.8.3. Rather than flood maps, however, grid maps, specified by `::` and a processor number in the grid, are used. For example, the statement

```
[R] A := AG#[::Index1%numLocales(grid(AG))];
```

also moves values in `AG` to `A`, but the processor numbers in the grid distributing `AG` are used to determine the values of `AG`.

> Implementation Note. At the time of this writing, grid maps are implemented even though flood maps are not.

4.3 Parallel Type Coercion

Figure 4.3 shows a type coercion hierarchy for ZPL's parallel types. The coercions are identified as *promotion*, *expansion*, and *relaxation*. As before, informally any change is called a promotion. Only when a distinction is necessary are the terms expansion and relaxation used.

The type coercions work the same as base type coercions. If, for example, a free scalar integer and a flood array of integers are added, the resulting expression is a grid array of integers. Generally, any expression may be read over a region higher in the hierarchy.

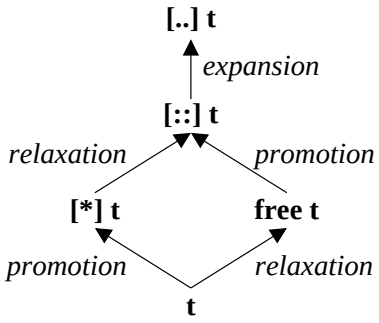

Figure 4.1: Type Coercion of ZPL's Parallel Types

It is important to note that the parallel operators do not apply to regular scalars. Although scalars can be assigned to free scalars, they are not automatically relaxed to free scalars just so the parallel operators will apply. They must be explicitly used in an expression with free scalars or assigned to a free scalar. Thus the code

```
var numProcs : integer;
numProcs := +<< 1;
```

is illegal rather than an inefficient calculation of `numLocales()`. On the other hand, the code

```
free var one : integer = 1;
numProcs := +<< one;
```

is legal. If this were not the case, it would be easy to write p-dependent code without explicitly using any of the p-dependent abstractions in ZPL.

4.4 Source-to-Source Compiler and Programmer Optimizations

A major advantage of the introduction of free scalars and grid dimensions to the ZPL languages is that it allows the compiler to implement many of its optimizations as source to source transformations. In addition, programmers are able to hand-code these optimizations for difficult or important cases. Note that the code rewrites in this section should not be written by the programmer. They are illustrative only. Better examples of free variables and grid dimensions will be seen in Chapter 6 where the implementations of the NAS

benchmarks in ZPL are discussed.

4.4.1 Manual Array Contraction

The free qualifier lets the ZPL programmer manually contract large parallel arrays and lets the ZPL compiler do so with a source-to-source transformation. Previously, the programmer would have to rely on the compiler to implement array contraction. Although the compiler does a good job contracting arrays [LLS98], there are cases that the compiler cannot handle. Section 6.1.3 discusses an example in which manual contraction is necessary. The following code illustrates an example ripe for contraction:

```
var A, B, Temp : [R] integer;
[R] begin
  Temp := A + B;
  A := Temp / A;
  B := Temp / B;
end;
```

In the above code, note that the array `Temp` cannot simply be eliminated. Were it replaced by the sum `A+B`, the program would produce a different result. Now, although the ZPL compiler contracts `Temp` in the above code, the programmer can write ZPL code, in which `Temp` is contracted, that is just as efficient. The following code contracts array `Temp` and replaces it by a free scalar `temp`:

```
free var temp : integer;
[R] interleave
  temp := A + B;
  A := temp / A;
  B := temp / B;
end;
```

In the above code, the statements are interleaved. This is equivalent to the compiler optimization of fusion. Fusion is necessary for enabling contraction. The code works because, for each index, the free scalar `temp` is assigned and then written. Defining all the values of `Temp` before using them is correct but inefficient.

The free qualifier is a low-level p-dependent abstraction. It can easily be used to write p-dependent code. For example, the statement

```
[R] A := temp;
```

executed after the manual contracted computation above would result in `A` being assigned a single value on each processor that is not necessarily the same value on all processors. Depending on the number or arrangement of processors, `A` could contain different values. A p-dependent program can easily be written by simply outputting `A`.

The free qualifier provides great power but only at great cost: the loss of p-independence. A program that works on some number of processors but fails to work on all numbers of processors is a real possibility. For example, consider what would happen if the programmer accidentally failed to interleave the statements in the example above. The code would then only produce the correct results on $\mathtt{n}^2$ processors, and depending on the values in `A` and `B`, in cases where the sums of the values of `A` and `B` stored on a given processor are the same. In the future work section of this thesis, a mechanism for automatically detecting such behavior is discussed.

4.4.2 Manual Decomposition of Full Reductions

The ZPL compiler decomposes a reduction into a local computation and a global computation. In the local computation, the values on a given processor are reduced. In the global computation, the locally reduced values are reduced into a globally consistent value. The free qualifier lets the ZPL programmer manually decompose a full reduction and lets the ZPL compiler do so with a source-to-source transformation. For example, the full reduction

```
var sum : integer;
[R] sum := +<< A;
```

is equivalent to the code

```
free var lsum : integer = 0;
[R] lsum += A;
sum := +<< lsum;
```

in which the full reduction has been manually decomposed.

In the declaration, `lsum` is initialized to zero. In the second line, the values in `A` on each processor are accumulated into `lsum`. In the last line, the reduction operator is used to compute the sum of every processor's copy of the free variable `lsum`. (See Section 4.1.1.)

The benefit of decomposing a reduction is immediate. The local computation can be fused with computation above. This increases the potential of fusion and contraction. It is typically unnecessary for the ZPL programmer to manually decompose a reduction since the compiler does so.

4.4.3 Manual Decomposition of Partial Reductions

Grid dimensions let the ZPL programmer manually decompose a partial reduction into a local computation and a global computation in much the same way that the free qualifier let the ZPL programmer manually decompose a full reduction. They also let the compiler implement manual decomposition of partial reductions using a source-to-source transformation. For example, the partial reduction

```
[0..n-1, 0] A := +<< [R] B;
```

is equivalent to the code

```
var T : [0..n-1, ::] integer = 0;
[R] T += B;
[0..n-1, 0] A := +<< [0..n-1, ::] T;
```

in which the partial reduction has been manually decomposed.

In the declaration, `T`'s second dimension is a grid dimension. It is thus able to hold the local sum within the second dimension. In the partial reduction of the third line, the grid dimension is used in the reduction to calculate the sum of the per-processor values in the second dimension.

This transformation is similar to what the compiler does for all partial reductions. It has the same benefits as the transformation on full reductions that was discussed earlier. Most notably, it increases the potential for fusion and contraction. Since the compiler implements this transformation, it is typically unnecessary for a programmer to write this code. ZPL users should think of this and all the other examples of manually decomposing scans and reductions as pedagogical.

4.4.4 Manual Decomposition of Scans

A similar transformation applies to scans. For example, the following codes are equivalent:

```
[1..n, 1..n] A := +|| [2, 1] B;
```

and

```
[1..n,  :: ] T := 0;
[1..n, 1..n] T += B;
[1..n,  :: ] T := +|| [2, 1] T;
[1..n, 1..n] interleave
               A := T;
               T += B;
             end;
```

4.5 Decomposition of Multidimensional Scans

The implementation of multidimensional scans is non-trivial. The following two source-to-source transformations simplify the code that needs to be generated by changing multidimensional scans into multiple scans over single dimensions. There is an apparent tradeoff between space and time but, as will be seen, the grid dimensions eliminate this tradeoff, making the second transformation the clear favorite.

4.5.1 Transformation 1: Reduce, Scan, and Flood

Since a scan is fundamentally a prefix reduction, Transformation 1 first computes the results of partial reductions on the dimensions already scanned. If it is the first dimension to be scanned, the partial reduction is the null partial reduction and the result is the array. Next, the transformation scans the result of the partial reduction. Finally, the results are flooded across these dimensions in order for the inner scans to be combined into the result.

As an example, consider the 3D scan given by

```
var A, B : [1..n, 1..n, 1..n] float;
[1..n, 1..n, 1..n] B := +||[3, 2, 1] A;
```

The transformed form, written pedagogically, is given by

```
[1..n, 1..n, 1..n]
  B := >>[1..n, 1..n, 1..n] +||[3] +<<[1..n, 1..n, 1..n] A +
```

```
      >>[1..n, 1..n, 1   ] +||[2] +<<[1..n, 1..n, 1..n] A +
      >>[1..n, 1   , 1   ] +||[1] +<<[1..n, 1..n, 1..n] A;
```

This is equivalent to the code (since the first flood and reduction neither flood nor reduce any values)

```
[1..n, 1..n, 1..n]
  B := +||[3] A +
       >>[1..n, 1..n, 1] +||[2] +<<[1..n, 1..n, 1..n] A +
       >>[1..n, 1   , 1] +||[1] +<<[1..n, 1..n, 1..n] A;
```

which is equivalent to the following using temporaries:

```
var
  T1 : [1..n, 1..n, 1] float;
  T2 : [1..n, 1   , 1] float;
[1..n, 1..n, 1   ] T1 := +<<[1..n, 1..n, 1..n] A;
[1..n, 1..n, 1   ] T1 := +||[2] T1;
[1..n, 1   , 1   ] T2 := +<<[1..n, 1..n, 1..n] A;
[1..n, 1   , 1   ] T2 := +||[1] T2;
[1..n, 1..n, 1..n] B := +||[3] A + T1 + T2;
```

The temporaries required by this transformation are small compared to the size of the arrays assuming that the first scanned dimension is non-trivial. In this case, all the temporaries can either be flooded or a singleton in this dimension. On square arrays, this saves an order of magnitude of storage.

The disadvantage of this transformation is that the scan duplicates the computation that was done in the reduction. Rather than just using a single log-tree communication pattern for the parallel-prefix computation, an additional one is used for the reduction. Transformation 2 solves this problem but requires additional storage. As will be discussed, the additional storage can be eliminated using grid dimensions.

4.5.2 Transformation 2: Scan and Flood

Transformation 2 computes the scans from the inside out. It uses the partial results to avoid having to recompute the log-tree communication patterns. As an example, consider again the 3D scan given by

```
var A, B : [1..n, 1..n, 1..n] float;
[1..n, 1..n, 1..n] B := +||[3, 2, 1] A;
```

The result of transformation 2 is as follows:

```
var T1 : [1..n, 1..n, 1..n] float;
    T2 : [1..n, 1..n, *   ] float;
    T3 : [1..n, *   , *   ] float;
[1..n, 1..n, 1..n] T1 := +||[3] A;
[1..n, 1..n, *   ] T2 := >>[1..n, 1..n, n] (+||[2] (A + T1));
[1..n, *   , *   ] T3 := >>[1..n, n   , n] (+||[1] (A + T1 + T2));
[1..n, 1..n, 1..n] B := T1 + T2 + T3;
```

Note that the reduction is eliminated, but that the temporary `T1` allocates substantial storage. To eliminate this storage, the trick is to manually decompose the scan before making this transformation. Note, fortunately, that the compiler implements this transformation, not the programmer.

4.5.3 Manual Decomposition of Parallel Text I/O

Grid dimensions let programmers write their own implementations of parallel text i/o routines. For example, the following code uses the sprintf, fseek, and fprintf functions of C to implement efficient parallel text output to a file:

```
var outfile                 :              file;
    A                       : [1..n, 1..n] float;
    BytesPerRow, ByteOffset : [1..n,  :: ] integer;
    str                     :              string;
[1..n,  :: ] BytesPerRow := 0;
[1..n, 1..n] BytesPerRow += sprintf(str, "%f ", A);
[1..n,  :: ] ByteOffset := +|| [2, 1] BytesPerRow;
[1..n,  :: ] interleave
               fseek(outfile, ByteOffset, SEEK_SET);
               fprintf(outfile, "%f ", A[]);
             end;
```

Because of the difficulty of determining how many bytes of data will be printed before a given position in the array is reached, this computation is non-trivial. The scan operator is ideal for the parallel computation and the grid dimensions make it efficient in memory: $O(n * p)$.

Note that line feeds could be printed at the end of each row if this were accounted for in the computation of `BytesPerRow` before the scan as in the following line of code:

```
[1..n, n] BytesPerRow += 1;
```

4.6 The High-Level Initialization of the NAS Benchmarks

Generating pseudo-random numbers is a non-trivial task in parallel programs. Given a problem with n numbers distributed over p processors, each processor may be asked to generate n/p pseudo-random numbers. If each processor uses the same random number seed, then each processor will generate the same numbers, presumably not what is desired. If each processor uses a random number seed to generate p random numbers and then uses its ID to select one of these random numbers as a seed to generate its n/p numbers, each processor will generate random numbers. This may or may not be acceptable depending on the quality of the random number generator. In addition, both of these techniques create a p-dependent set of pseudo-random numbers. A p-independent solution is desirable. Unfortunately, without a clever trick, this is non-scalable.

The NAS Parallel Benchmarks use a linear congruential pseudo-random number generator for initialization. The routine `randlc(x)` returns the next pseudorandom number given a real number `x`. The routine `ipow46(x,i)` initializes the variable `x` assuming `i` calls to `randlc`. It uses a doubling scheme to find this number using log calls to `randlc`. This makes the generation of a p-independent sequence of random numbers fast and scalable.

There are many ways to write this code in ZPL. This section will describe a modular implementation of a wrapper procedure that generates the `i`th random number quickly; this procedure is currently used in the ZPL implementations of the NAS MG, CG, and FT benchmarks. As an illustration of the compiler support for writing code using free variables, this section will present code that does not use any free qualifiers and show how the compiler can help the programmer add free qualifiers where necessary.

Listing 4.1 shows code for generating p-independent random numbers in a two-dimensional array `A` without the necessary free qualifiers. The wrapper function `random` looks like `ipow46` in that it takes an integer `i` along with a temporary variable `x`. Unlike `ipow46`, however, it returns the `i`th random number and can therefore be called just as in line 15.

The simplest way to write this wrapper function, and in a p-independent way, would be to call `ipow46` every time. This, however, is too slow of a solution. In the code in Listing 4.1, a static variable, a variable that exists in the global scope of the program but is

Listing 4.1: NAS Initialization in ZPL Without Free Qualifiers

```
1  procedure random(inout x : double; i : integer) : double;
2  static var
3    last : integer = -2;
4  begin
5    if i != last + 1 then
6      ipow46(x, i);
7    end;
8    last := i;
9    return randlc(x);
10 end;

12 var A : [0..n-1, 0..n-1] double;
13 var x : double;

15 [0..n-1, 0..n-1] X := random(x, Index1*n + Index2);
```

only accessible in the lexical scope of the procedure, is used to determine whether `ipow46` needs to be called. If `randlc` was just used to compute `i-1`, then `randlc` can be called immediately.

The code in Listing 4.1 is incorrect because the free qualifier has been omitted. The programmer who writes this code would probably at least realize that `last` needs to be free since it takes on different values on different processors. The programmer may even realize that `random` cannot be promoted because it has a side effect on a static scalar (which is equivalent to a side effect on a global scalar).

For illustrative purposes, consider trying to compile this code. The ZPL compiler returns error messages similar to the ones that follow:

```
ERROR: Parallel procedure random may not be promoted (arg 2) (15)
ERROR: Scalar cannot be passed by inout to promoted function (15)
ZPL compile failed -- exiting.
```

The compiler also returns the following information about `random`:

```
Procedure random is parallel because of line 8
```

The first error message says that the function random may not be promoted because parallel functions may not be promoted. The procedure is parallel because of the assignment in line 8. The static variable must be made free. The second error message tells the programmer

Listing 4.2: NAS Initialization in ZPL With Free Qualifiers

```
1  free procedure random(free inout x : double; i : integer) : double;
2  static free var
3    last : integer = -2;
4  begin
5    if i != last + 1 then
6      ipow46(x, i);
7    end;
8    last := i;
9    return randlc(x);
10 end;

12 var A : [0..n-1, 0..n-1] double;
13 free var x : double;

15 [0..n-1, 0..n-1] X := random(x, Index1*n + Index2);
```

that `x` cannot be passed by `inout` since the procedure is promoted. It too must be made free since it will be assigned within a promoted procedure. Changing the declarations of `x` and `last` to free and recompiling results in the following error messages:

```
ERROR: Cannot pass scalar by inout in free control flow (6)
ZPL compile failed -- exiting.
```

This error message indicates that `x` must be free in the procedure header. Whether it is free or not when passed into the routine, it is assigned a value within free control flow. Changing the procedure header so that `x` is free and recompiling results in one last error message:

```
ERROR: Can only return free expressions in free procedures (9)
ZPL compile failed -- exiting.
```

Because `random` returns a value dependent on `x`, it returns a free value. The procedure must thus be declared as free. Listing 4.2 shows the code of Listing 4.1 with the required free qualifiers.

The free qualifier is simple to work with. Knowing that values can take on different values on different processors is useful to the programmer trying to understand the code. ZPL's type checker is essential to making programming with free qualifiers useful.

4.7 Timing Parallel Programs

Timing parallel programs is not as easy as timing sequential programs because each processor acts independently and may take a different amount of time to execute the same part of a code. In addition to deciding whether to report the minimum, average, or maximum amount of time, the programmer must decide whether to synchronize the processors before starting to time a segment of code. Note that when timing nested sections of a code, synchronizing the processors before timing the inner section could seriously impact the timing of the outer section.

ZPL provides two functions for timing parallel programs that do not use free scalars or grid dimensions. Unfortunately, they have serious limitations making them only appropriate for coarse-grained timings. The two procedures, `ResetTimer` and `CheckTimer`, are defined as follows:

```
procedure ResetTimer() : double;
procedure CheckTimer() : double;
```

The `ResetTimer` procedure resets the program timer. It returns the elapsed time (in seconds) since the last call to `ResetTimer`. The `CheckTimer` procedure checks the program timer, but does not reset it. It returns the elapsed time (in seconds) since the last call to `ResetTimer`.

Since `ResetTimer` and `CheckTimer` return scalar values, the same values must be returned on different processors. Otherwise ZPL's scalar consistency model would break. Therefore the maximum time is always returned. The processors implicitly synchronize to compute this maximum reduction. Additionally, processors are synchronized whenever ResetTimer() is called.

Because of this synchronization, these procedures are only appropriate for coarse-grained timings. The results of timing fine-grained sections of code or nested sections of code will be impacted by this synchronization.

With the free scalars extension, a more robust facility for timing parallel programs is possible. This support complements, rather than replaces, the above procedures.

Rather than relying on a single program timer, the new timing support adds a `timer`

type. The `timer` type cannot be read, written, or operated on except through the following four procedures, which are appropriate for taking both coarse-grained and fine-grained timings:

```
procedure ClearTimer(free out t : timer);
procedure StartTimer(free inout t : timer; sync : boolean);
procedure StopTimer(free inout t : timer);
free procedure ReadTimer(free t : timer) : double;
```

The `ClearTimer` procedure resets timer `t` and turns it off. It's elapsed time is set to zero seconds. Until `StartTimer` is called with timer `t`, any call to `ReadTimer` with timer `t` will return zero on every processor. As a convenience, the `ClearTimer` procedure is implicitly called on all timers when they are declared.

The `StartTimer` procedure turns timer `t` on. The elapsed time thus starts increasing. If `sync` is passed in as `true`, the processors are synchronized before the timer is turned on. The issue of whether to synchronize is left in the hand of the programmer. It is a difficult decision, though there are some simple guidelines the programmer can follow. For example, if the code segment starts with a reduction, it is important to synchronize since otherwise the time that some processors spend waiting for other processors to reach the reduction could impact the time for the code segment. On the other hand, if the code segment of interest is short but repeated many times, the overhead of synchronization could adversely impact the total time of a segment that contains this measured code segment of interest.

The `StopTimer` procedure turns timer `t` off. The elapsed time is frozen. Note that it is not necessary to call `StopTimer` before calling `ReadTimer`. There is no decision about whether to synchronize after or before a call to `StopTimer`. It is sufficient to make this decision for all calls to `StartTimer`.

The `ReadTimer` procedure returns the elapsed time of timer `t` in seconds. Its result can be different on different processors. A reduction can be used to find a set time. For example, the maximum, minimum, and average timings are easy to compute using the maximum, minimum, and sum reductions.

The power of these timers should not be underestimated. They make the difficult task of accurately timing and profiling parallel programs easy.

4.8 The Potential for Using MPI in ZPL

The next section discusses the related work of Titanium and discusses what should already be known: ZPL limits the nature of the parallel codes. The advantage is the coupling of two properties, the WYSIWYG performance model and the mostly p-independent semantics. The disadvantage is that ZPL might not be sufficient.

This thesis does not attempt to definitively determine whether ZPL is sufficient although it argues that, for most cases, ZPL's model is simpler, easier-to-use, and sufficient for achieving performance as high or higher than any less restricted model can manage. That said, this section begrudgingly notes that free scalars are a powerful extension to ZPL because, in addition to the myriad other reasons discussed, they make it possible to provide an MPI interface in ZPL. The programmer is strongly cautioned against reverting to MPI.

The following ZPL prototypes and their C procedures might be used to implement MPI's blocking send and receive routines on integers:

```
extern prototype MPI_Send(free i, locale, tag : integer);
extern prototype MPI_Recv(free inout i : integer;
                          free locale, tag : integer);


void MPI_Send(int i, int locale, int tag) {
  MPI_Send(&i, 1, MPI_INT, locale, tag, MPI_COMM_WORLD);
}

void MPI_Recv(int *i, int locale, int tag) {
  MPI_Recv(i, 1, MPI_INT, locale, tag, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
}
```

It is a small jump to see how the other routines in MPI could be prototyped in ZPL.

Note that these functions cannot be implemented as ZPL procedures, at least not without changing their semantics. As a proof by contradiction, consider that their use can cause deadlocks. More directly, they cannot both be called in shattered or free control flow and be implemented with any of ZPL's communication-inducing operators. Since they are meant to be called within free control flow, *e.g.*, within a conditional to guard some processors from making the call, any ZPL implementation of them would be insufficient.

4.9 Related Work

ZPL is unique in coupling a performance model as powerful as its own with mostly p-independent semantics. This chapter's extension, which adds two low-level p-dependent abstractions to ZPL, is not similar to any other languages. To this end, it is useful to discuss Titanium.

Titanium provides a keyword `single` that is the exact opposite of ZPL's keyword `free`. In Titanium, the default is free whereas in ZPL the default is single. When data is single, it is consistent across the processors; when it is free, it is not. Similarly, control can be described as free or single. In single control, the processors all execute the same program points. In free control, they do not.

Shattered and free control flow in ZPL are free. The default is single in ZPL. In Titanium, free control is the default. All of the GAS languages, UPC, Co-Array Fortran, and Titanium, allow for communication within free control. This is the fundamental difference between low-level languages and ZPL, HPF, and high-level languages. In ZPL, communication is only legal in single control.

Titanium is one of the few languages that allows for communication in free control and also provides a mechanism for single control. By restricting global synchronization and broadcast routines to be used within single control, Titanium guarantees to the programmer at compile-time that these constructs are being used correctly. While it does restrict the possible codes (There are correct codes that the compiler would reject.), this restriction is probably acceptable and wise.

Though using different terminology, Luc Bougé[Bou96] argued that single control languages, such as ZPL and HPF, are superior to free control languages, such as Fortran+MPI and UPC, by making an analogy to sequential languages. He reasoned that structured programming places restrictions on what the programmer can write, as opposed to unstructured programming with the `goto` statement, but these restrictions greatly simplify the programming process and do not severely restrict the kinds of codes that programmers write.

4.10 Summary

This chapter presented two abstractions that add processor-oriented coding capabilities to the high-level parallel programming language ZPL. Sometimes, high performance demands processor-oriented programming. This chapter has shown some examples of this. While making it possible for the programmer to write per-processor code, the abstractions presented in this chapter let the programmer write most of the code at a higher level, independent of the processors.

Chapter 6 will show more examples of how these abstractions are used in more complicated benchmarks. Always, the nature of these abstractions lets the programmer limit the impact of the processor-oriented coding. This makes it possible for the programmer to ignore issues of synchronization that plague other languages.

Chapter 5

SUPPORT FOR USER-DEFINED REDUCTIONS AND SCANS

ZPL's parallel reduce operator, *op*<<, and parallel scan operator, *op*||, let scientists easily write consistently efficient code for computing the reductions and scans of several scalar operators. For many other scalar operators, however, it is impossible to write an efficient reduction or scan. This chapter addresses this inability by presenting language-level support for implementing arbitrary *user-defined* reductions and scans. A user-defined reduction or scan is one in which the scalar operator used by the reduction or scan is defined by the programmer. The mechanism for user-defined reductions and scans subsumes that for built-in reductions and scans since any built-in operator can be written as a user-defined operator. Nonetheless, wherever possible, a built-in reduction or scan is preferred since the compiler can better analyze the built-in reductions and scans.

Like free variables and grid dimensions, user-defined reductions and scans are p-dependent abstractions. Sometimes, the p-dependent nature of the code can be desirable. For example, a user-defined reduction to find the position of the minimum element in an array may be underspecified, leaving multiple correct solutions. This program then produces different correct solutions on different processor arrangements; it is p-dependent but correct. To make the code p-independent, it can be written to find the minimum position of one of many minimum values rather than any position of one of many minimum values. In this example, there may be no performance loss, however, in other cases, it is conceivable that there would be greater costs associated with p-independence.

More frequently, the impact of the p-dependent nature of user-defined reductions and scans is incorrect code. By providing this mechanism in a largely p-independent framework, however, this impact is limited and significantly easier to handle. Knowing there are exactly three ways that a user-defined reduction or scan could cause p-dependent results makes it easier to find the source of the error.

This chapter is organized as follows. The next section presents basic support for user-defined operators that is sufficient for simple reductions and scans such as the built-in reductions and scans. Section 5.2 presents advanced support for user-defined operators that enables more complex reductions and scans. Section 5.3 discusses the impact of user-defined scans and reductions on ZPL's performance model and p-independent framework. Section 5.4 discusses related work, and Section 5.5 summarizes.

5.1 Basic User-Defined Operators for Scans and Reductions

This section presents basic support for user-defined scans and reductions. It is powerful enough to define all of the built-in scans and reductions. The next chapter will extend this support and show more complex examples.

In the reduction and scan, each processor first computes its local contribution. Then, a collective log-tree communication pattern is employed to compute the global contribution, in the case of the reduction, or the prefix contribution, in the case of the scan. Finally, in the case of the scan, the prefix contributions are used to compute the scan.

As seen in Chapter 4, reductions and scans can be decomposed. To create a user-defined reduction, the programmer is essentially defining functions to replace parts of the code in the decomposition. For example, Section 4.4.2 shows the decomposition of a full reduction. In this code, `lsum` is assigned zero. This is the identity for the sum reduction. The `+` operator was then used twice. First to accumulate values into `lsum` and then to combine the values in each processor's copy of `lsum`.

To define a new reduction or scan operator, the programmer must specify two functions: the identity function and the accumulator function. In the case of redefining a sum reduction, the identity function returns zero and the accumulator function adds its arguments. The accumulator is used for both the accumulate and combine parts of the decomposed code. In the next section, the notion of an initial type, a state type, and a final type will be introduced and the programmer will be able to define two new functions including a separate function to be used for the combine part of the decomposition. In this basic support, it is assumed that the types of the elements being operated on and the resulting elements

are the same.

To define an operator, the programmer must implement the identity and accumulator functions and attach them to an operator. The BNF for specifying this operator is as follows:

```
operator-definition ::=
  ( 'associative' | 'nonassociative' )
  ( 'commutative' | 'noncommutative' )
  'operator' identifier
    'identity' '=' procedure-header
    'accumulator' '=' procedure-header
    [ 'combiner' '=' procedure-header ]
    [ 'scan-generator' '=' procedure-header ]
    [ 'reduce-generator' '=' procedure-header ]
  'end' ';'
```

Note that the combiner and generator functions will be introduced in the next section.

> Implementation Note. At the time of this writing, user-defined reductions in ZPL are specified using procedure overloading as described in the literature [DCS02]. User-defined scans are not supported.

5.1.1 The Identity Function

The identity function should return the identity element for a reduction or scan. Its value should have the identity property. Namely, that when operated on by the accumulator function along with another value, the other value is returned. If the type of the elements is `t`, then the identity function is specified by either of the following two function forms:

```
procedure fi() : t;
procedure fi(out x : t);
```

In the first form, the function returns the identity element. In the second form, the function passes the identity element as a reference parameter. Both forms are provided as a convenience. In the accumulator function, where both forms are also provided, the latter version can be more efficient if `t` is a complex type and copying is expensive.

The identity function is useful for a fast implementation. There are two principal advantages to requiring an identity function. First, the reduction over an empty region is well-defined. It is simply the identity. Second, if some processors do not contain any elements, passing the identity to other processors results in a simple implementation strategy.

ZPL is unique in its demand that the programmer supply a procedure that returns the identity element of the reduction or scan operator. While it is expected that the vast majority of user-defined reductions and scans will have identities, it is important to note that this demand does not preclude the definition of operators that do not have an identity. In this case, the programmer can use a fake identity by defining the state type with an identity bit flag. The accumulator or combiner can then deal with this case ignoring the argument that has its identity bit flag set. It can simply pass the other argument.

5.1.2 The Accumulator Function

The accumulator function takes an element from the array and accumulates it into a tally on each processor. The tallies are originally initialized with the identity functions. As with the identity function, the accumulator function can take two forms:

```
procedure fa(x : t; y : t) : t;
procedure fa(x : t; inout y : t);
```

The latter form is especially useful if `t` is complex and should not be copied. For simple types, the former function is just as efficient.

In a full reduction, the tally is simply a free scalar of type `t`. A basic full reduction resulting in a scalar value `result` computed over a region `R` of an array `A` are then computed as follows:

```
free var tally : t  = fi();
tally := fi();
[R] tally := fa(A, tally);
result := fa<< tally;
```

In the last line, the accumulator function is used a second time to combine the processors' copies of the free scalar `tally`.

In partial reductions and scans, there are multiple tallies. They are easily understood as parallel arrays with grid dimensions. For example, to decompose a partial reduction with the user-defined operator `f` having identity `fi` and accumulator `fa` given by

```
[0..n-1, 0] A := f<< [R] B;
```

the compiler would generate the following code:

```
var Tally : [0..n-1, ::] integer = fi();
[R] Tally := fa(B, Tally);
[0..n-1, 0] A := fa<< [0..n-1, ::] Tally;
```

In the last line, the values in `T` over the `::` dimensions are combined using the accumulator function.

5.1.3 Associativity and Commutativity

The built-in reductions are all associative and commutative, but there are important reductions that may not have these properties. For example, a reduction to find the longest sequence of `true` values in a `boolean` array is associative but not commutative and a scan to fill an array with random numbers using a linear congruence relation is neither associative nor commutative.

It is essential for the compiler to know the associativity of a particular user-defined reduction or scan. If the reduction or scan is not associative, the implementation cannot use a fan-in tree. Thus non-associative scans have worse performance than ZPL's worst-case performance model indicates. Note that all the built-in reductions are associative.

It is less essential for the compiler to know the commutativity of a particular user-defined reduction, because non-commutative reductions can still use a fan-in tree for their implementation. Hence non-commutative reductions are almost as fast as commutative reductions. On some machines, commutativity could give us a better implementation since data can be combined immediately when it is received. For example, if the fan-in tree has each processor receiving data from more than two processors, the data can be used immediately if the reduction is commutative even if it comes in in an arbitrary order.

The programmer must specify whether the operator is associative or nonassociative as well as commutative or noncommutative. There is no default because neither default seems acceptable. If the default is associative and commutative, it seems wrong that not specifying nonassociative or noncommutative would result in a difficult-to-find p-dependent error. If the default is nonassociative and noncommutative, it seems wrong that not specifying associative or commutative would result in significantly diminished performance. Programmers that define their own reductions or scans need to think through the associativity and

Listing 5.1: A User-Defined Sum Operator With Return Statements

```
1  associative commutative operator sum
2    identity = sumIdentity() : float;
3    accumulator = sumAccumulator(x1, x2 : float) : float;
4  end;

6  procedure sumIdentity() : float;
7  begin
8    return 0;
9  end;

11 procedure sumAccumulator(x1, x2 : float) : float;
12 begin
13   return x1 + x2;
14 end;
```

commutativity.

5.1.4 Handling Nonassociative Reductions and Scans

Reductions and scans that are either non-associative or non-commutative need to be evaluated in a fixed order. For scans, the order is normally specified, but for reductions it is not. At the time of this writing, there is no mechanism for fixing the order of non-associative reductions.

Nonassociative reductions and scans are handled by a source-to-source transformation that attempts to take advantage of pipelining. In the case of partial reductions and scans, pipelining is possible because there is an independent computation in given dimensions. For example, the partial scan given by

```
[R] B := f|| [2] A;
```

can be transformed into the following lines of code:

```
[0..n-1, 0     ] B := A;
[0..n-1, 1..n-1] B := fa(A, B'@[0,-1]);
```

5.1.5 The Sum Reduction Redefined

All of the built-in reduction and scan operators can be written as user-defined operators

Listing 5.2: A User-Defined Sum Operator With Reference Parameters

```
1  associative commutative operator sum
2    identity = sumIdentity(out x : float);
3    accumulator = sumAccumulator(x1 : float; inout x2 : float);
4  end;

6  procedure sumIdentity(out x : float);
7  begin
8    x := 0;
9  end;

11 procedure sumAccumulator(x1 : float; inout x2 : float);
12 begin
13   x2 += x1;
14 end;
```

for reductions and scans. Listing 5.1 and 5.2 show two ways to define a sum operator for the sum reduction and scan. Figure 5.1 uses return statements in the procedure definitions and Figure 5.2 uses reference parameters in the procedure definitions.

5.2 Advanced User-Defined Operators for Scans and Reductions

The previous section provided support for basic user-defined scans and reductions. This section looks at more complex scans and reductions. In particular, it looks at scans and reductions in which the array expression, the tally, and the result are of different types. Instead of a single type `t`, there are three types: an initial type `t1`, a state type `t2`, and a result type `t3`.

A general full reduction given by `[R] result := f<< A` has three distinct types: the type of `A` is `t1`, the type of `result` is `t3`, and the type of the tally is `t2`. The identity function `fi`, the accumulator `fa`, the combiner `fc`, and the generator `fg` are given by the following procedures:

```
procedure fi() : t2;
procedure fa(x : t1; y : t2) : t2;
procedure fc(y1 : t2; y2 : t2) : t2;
procedure fg(y1 : t2) : t3; -- reduction (scan is different)
```

The decomposition of the full reduction then is as follows:

```
free var local_tally : t2 = fi();
```

```
var global_tally : t2;
[R] local_tally := fa(A, local_tally);
global_tally := fc<< local_tally;
result := fg(global_tally);
```

5.2.1 The Combiner Function

When the array expression type `t1` differs from the state type `t2`, the combiner function is necessary for an efficient implementation that takes advantage of associativity. The accumulator operates on the array expression (type `t1`) and the tally (type `t2`). After the accumulation is complete, the processors' copies of tally are combined, so the combiner function must operate on arguments of type `t2`.

As with the identity function and the accumulator function, the combiner function can take two forms:

```
procedure fc(y1, y2 : t2) : t2;
procedure fc(y1 : t2; inout y2 : t2);
```

Note that there is sometimes a tradeoff between potential implementations of the accumulator and the combiner. When this happens, the programmer should always optimize the accumulator. It is executed much more frequently than the combiner.

The Minimums Reduction Example

The minimums reduction is a good example of a complex reduction in which the array expression type `t1 = float` is different from the state type `t2 = vector`. The result type `t3` is the same as the state type `t2`. The minimums reduction computes the `k` minimum values in a parallel array. Without a mechanism for user-defined reductions, this is possible but inefficient using `k` consecutive applications of a minimum reduction.

Listing 5.3 shows ZPL code for defining the minimums reduction. The identity function is defined in lines 9–12. It returns a vector initialized to the maximum float values. This ensures that the ten smallest values in the array expression are found. The accumulator function is defined in lines 14–29. This procedure adds values to the vector of minimums if they are less than the smallest minimum already found. By keeping the vector in sorted order, this code is optimized. The combiner function is defined in lines 31–38. This code is

Listing 5.3: A User-Defined Minimums Operator

```
1  type vector = array[1..k] of float;

3  associative commutative operator minimums
4    procedure minimumsIdentity(out v : vector);
5    procedure minimumsAccumulator(x : float; inout v : vector);
6    procedure minimumsCombiner(v1 : vector; inout v2 : vector);
7  end;

9  procedure minimumsIdentity(out v : vector);
10 begin
11   v[] := FLOAT_MAX;
12 end;

14 procedure minimumsAccumulator(x : float; inout v : vector);
15 var
16   i : integer;
17   tmp : float;
18 begin
19   if x < v[1] then
20     v[1] := x;
21     for i := 1 to k-1 do
22       if v[i] < v[i+1] then
23         tmp := v[i+1];
24         v[i+1] := v[i];
25         v[i] := tmp;
26       end;
27     end;
28   end;
29 end;

31 procedure minimumsCombiner(v1 : vector; inout v2 : vector);
32 var
33   i : integer;
34 begin
35   for i := 1 to k do
36     minimumsAccumulator(v1[i], v2);
37   end;
38 end;
```

not as fast as it could be. Namely, a merging of the sorted vectors could be more efficient. In a real implementation, this would be desirable. Nonetheless, because the accumulator is called many more times than the combiner, this implementation may be acceptable in production codes.

The minimums reduction is representative of useful user-defined reductions. It is comparable to a reduction that is used in the initialization phase of the NAS MG benchmark. In this benchmark, the ten smallest and largest elements, as well as their locations, are found in an array. The ZPL implementation using user-defined reductions proved to perform significantly better than the low-level Fortran+MPI implementation using built-in minimum reductions [DCS02].

5.2.2 The Extended Accumulator Function

It is sometimes important that the accumulator function have access to more than just the array expression data. The accumulator may be defined to take additional arguments. In this case, the additional arguments must appear just before the state type in the function definition, *i.e.*, they must appear at the location of the ellipsis in the following accumulator prototype alternatives:

```
procedure fa(x : t; ...; y : t) : t;
procedure fa(x : t; ...; inout y : t);
```

If the user-defined operator `f` has accumulator function `fa`, then this reduction could be called over `R` on array `A` as in the following code:

```
[R] B := f<< (A, ...);
```

Note that the expressions represented by the ellipsis are passed into the user-defined reduction. These expressions correspond to the expressions that the accumulator function expects in place of its ellipsis.

The Minimum Index Reduction Example

These extra arguments are useful in cases where, for example, the position of an element needs to be known as well. The minimum index reduction is a good example. In this common reduction, which is a built-in reduction in MPI and HPF, the programmer finds

Listing 5.4: A User-Defined Minimum Index Operator

```
1  type valpos = record
2    value : float;
3    index : integer;
4  end;

6  associative commutative operator mini
7    identity = miniIdentity(out vp : valpos);
8    accumulator =
9      miniAccumulator(v : float; i : integer; inout vp : valpos);
10   combiner = miniCombiner(vp1 : valpos; inout vp2 : valpos);
11 end;

13 procedure miniIdentity(out vp : valpos);
14 begin
15   vp.value := FLOAT_MAX;
16 end;

18 procedure miniAccumulator(v : float; i : integer; inout vp : valpos);
19 begin
20   if v < vp.value then
21     vp.value := v;
22     vp.index := i;
23   end;
24 end;

26 procedure miniCombiner(vp1 : valpos; inout vp2 : valpos);
27 begin
28   if vp1.value < vp2.value then
29     vp2 := vp1;
30   end;
31 end;
```

the position of the minimum value as well as the minimum value itself.

Listing 5.4 shows ZPL code for creating the user-defined reduction for the minimum index reduction. In addition to the array expression, the accumulator function, defined in lines 18–24, takes an integer expression. For a reduction over `R`, the extra parameter passed to this reduction could be `Index1*n+Index2`. The combiner function does not need to be passed this integer. Indeed, there is no integer it could be passed. It compares two per-processor minimums and their locations, and returns the smaller minimum and its location.

5.2.3 The Generator Function

The generator function is useful when types `t2` and `t3` are different. This function is especially important in the case of the scan when the state type `t2` is large. The result of a reduction is relatively small whereas the result of a scan is the same size as the source. Thus if there is no generator, the scan may produce a result which is larger than necessary.

The generator function is different for reductions and scans. The reduction generator function turns a state type element into a result type element. It is straightforward to define, and its function prototype alternatives are as follows:

```
procedure fg(y : t2) : t3;
procedure fg(y : t2; out z : t3);
```

The results of the combiner have state type `t2`. The reduction generator function simply turns these elements into elements having result type `t3`.

The scan generator function is more complicated since it may do different things with the prefix result of the combiner depending on where it is in the scan. That is, the value that was sent to the accumulator is important. The prototypes for the scan generator function are as follows:

```
procedure fg(x : t1; ...; y : t2) : t3;
procedure fg(x : t1; ...; y : t2; inout z : t3);
```

They are identical to the accumulator except that the state type prefix result is taken as input and a result of type `t3` is returned.

The Counts Scan Example

The counts scan is a good example of how the generator function could be used. It is a useful operator, and its reduction is applicable to the NAS IS benchmark.

The counts reduction takes an array expression that contains integer values between `1` and `k`. The reduction returns an array of size `k` that contains the number of times each integer between `1` and `k` appears in the array expression. Without a generator function, the counts scan would return a parallel array where each element was an indexed array of size `k` that contains the prefix reductions. The generator function lets us change the definition of the counts scan so that it returns a parallel array of integers instead. It returns the number of times the integer at that position in the array has already appeared. Notice that the state type for this scan needs to be an indexed array of size `k`, but the sizable parallel array of indexed arrays is never necessary with the generator function.

Listing 5.5 shows a ZPL definition of a counts operator. Notice that the generator function takes the integer `i` and uses it to pull out the particular count of interest to that position in the array.

5.3 P-Independent Discussion

When p-dependent values emerge in a user-defined reduction or scan it is most probably an error. (The other case is that there are multiple correct solutions.) If it is an error, there are only three possible ways it could have arisen. This limit on the number of possibilities makes debugging significantly easier. The three sources of p-dependent errors in user-defined reductions and scans are as follows:

- The identity function does not produce the identity. Since the identity procedure is called exactly once by each processor involved in the reduction, if the result is not the identity, the reduction or scan will produce p-dependent values. For example, if the accumulator and combiner compute addition and the identity function returns one instead of zero, the result of a reduction will be the sum of the inputs plus the number of processors.

Listing 5.5: A User-Defined Counts Operator

```
type list = array[1..k] of integer;

associative commutative operator counts
  identity = procedure countsIdentity(out l : list);
  accumulator =
    procedure countsAccumulator(i : integer; inout l : list);
  combiner = procedure countsCombiner(l1 : list; inout l2 : list);
  scan-generator =
    procedure countsScanGenerator(l1 : list) : integer;
end;

procedure countsIdentity(out l : list);
begin
  l[] := 0;
end;

procedure countsAccumulator(i : integer; inout l : list);
begin
  l[i] += 1;
end;

procedure countsCombiner(l1 : list; inout l2 : list);
begin
  l2[] += l1[];
end;

procedure countsScanGenerator(i : integer; l : list) : integer;
begin
  return l[i];
end;
```

- The combiner computes a different function than the accumulator. For example, if the accumulator computes a sum and the combiner computes a product then, on more than one processor, the result will be the product of the per-processor sums, a potentially p-dependent value.

- The operator was incorrectly identified as being associative or commutative. In this case, an error in evaluation is likely on more than one processor.

The importance of limiting the possibilities of p-dependent values should not be underestimated. In the porting of the MG benchmark from Fortran+MPI to ZPL, the ZPL program initially had a bug that showed up on more than one processor. After finally getting the program to work correctly on one processor, it failed on two. The MG benchmark does not use any free scalars, grid dimensions, or scatter remaps. It does, however, use a user-defined reduction to find the ten smallest elements in a parallel array. This reduction is similar to the minimums reduction discussed earlier, but also finds the positions of each of the minimum values.

When the ZPL program failed on two processors, it was clear that the error had to be in the user-defined reduction. Since it was obvious that the operator was associative and commutative, and since the identity function was trivial, the problem had to be in the combiner. A quick read over this function revealed a minor bug.

Nonassociative user-defined reductions and scans are p-independent. Changing a p-dependent associative operator to a nonassociative operator will make it p-independent, but will severely degrade performance. This can be useful for debugging, however.

5.4 Related Work

MPI provides robust support for user-defined reductions and scans. It differs substantially from ZPL's support because the MPI programmer writes per-processor code. Thus programmers only need to specify an equivalent of the combiner function. The identity function, the accumulator function, and the generator function are not applicable because the MPI programmer naturally writes these functions using the low-level abstractions of MPI. The

single combiner function is written using the `MPI_Op_create` routine. It is assumed to be associative and is identified as being commutative or not by the programmer.

HPF did not provide support for user-defined reductions because it was thought to be too difficult to implement. However, a draft proposal for supporting user-defined reductions was defined. It consisted mainly of an additional directive to the compiler.

The C** language supports user-defined reductions [VL96]. Its support is quite similar to the support discussed here, however there are some significant differences. There was no support for an identity function, for multiple parameters to the accumulator function, and for state types and the generator function. Also, the mechanism introduced the possibility of data races which stemmed mostly from C**'s per-processor view of the computation.

Previous work on support for user-defined reductions in ZPL was based on overloading procedures of the same name in order to specify the identity, the accumulator, and the combiner [DCS02]. (There was no generator.) There were three main problems with the approach of overloading procedures. First, the error messages that arose when the procedures were not defined exactly as they should be were often misleading or uninformative. Second, the user-defined reductions could only be used in simple assignment statements so that the correct result type could be determined. Third, if the state type was the same as the initial type, there was no way of supplying a combiner that was different from the accumulator. Although this is not common, it seems possible given a fairly complex reduction and initial type.

5.5 Summary

High performance is difficult to achieve without good support for user-defined reductions because their implementation can be highly optimized. The performance penalty of working around poor support for user-defined reductions is exacerbated in high-level languages so good support is imperative. ZPL provides such support without retreating to a per-processor view of computation.

ZPL's support for user-defined reductions was previously shown to be crucial to applications [DCS02]. The same analysis holds now even though the abstractions have been

modified.

Chapter 6

NAS PARALLEL BENCHMARKS

The NAS parallel benchmarks are a suite of scientific applications and kernels implemented in Fortran or C and MPI. According to their authors, "these implementations, which are intended to be run with little or no tuning, approximate the performance a typical user can expect for a portable parallel program on a distributed memory computer." [BHS+95] In truth, the authors are being modest; these codes are well-designed, highly-tuned examples and represent the potential performance that a seasoned programmer can achieve with substantial time and effort. As such, they represent a golden opportunity for designers of parallel programming languages to strut their high-level abstractions. For typical scientists, who are not programmers, high-level languages offer even greater advantage.

This chapter presents ZPL versions for three of the five NAS kernels: EP (Embarrassingly Parallel), FT (Fast Fourier Transform), and IS (Integer Sort). CG and MG are omitted; the reader interested in ZPL versions of CG and MG is referred to the literature [CS01, CDS00, Cha01]. For each benchmark, the ZPL version is qualitatively compared to the NAS provided Fortran or C and MPI version.

Performance results are shown after each version has been presented for a benchmark. A Cray T3E was used to measure the performance of each benchmark. Speedup graphs are used to show the performance of Class C of each version of the benchmarks, and memory graphs show the total amount of memory used by each version of the benchmarks. For all the speedup graphs, the speedup was calculated against the version that produced the fastest time on the lowest number of processors that any version ran on. Memory constrains how few processors can run any of the benchmarks. Class C did not run on one processor for any of the benchmarks.

The Cray T3E and Class C were selected after careful consideration. First, although the Cray T3E is an older machine, it is representative of many parallel machines. It is

Table 6.1: NAS EP Classes

Class	*Pseudo-random Numbers*
A	2^{28}
B	2^{30}
C	2^{32}
D	2^{36}

well-designed and capable of achieving high performance. Thus problems that a code may have with scaling show up well on the T3E. Second, the T3E is a basic machine. Using a vector machine like the Cray X1 or a machine of clustered SMPs like the IBM p655 may not reflect the standard behavior of a code since there are many issues to consider when analyzing codes run on these machines. Third, class C was chosen because it represents a large class size that programmers would need a machine with the T3E's stature to run. Class D was determined to be too large of a class for the T3E to handle. Its age shows in the amount of memory on each processor.

Information on the machine, the compilers, and the raw data used to conduct the experiments in this chapter is listed in Appendix B.

6.1 Embarrassingly Parallel (EP)

The NAS EP (Embarrassingly Parallel) benchmark computes pairs of normally distributed numbers. It derives its name from its loosely coupled nature. Each processor works independently until the very end.

The basic execution of EP is as follows: Each processor independently generates batches of pseudo-random numbers and uses these numbers to compute pairs of normally distributed numbers. The rounded norms of these pairs are tallied for each batch. After all the batches have been generated and all the norms tallied, reductions are used to combine each processor's tallies.

The EP benchmark comes in four classes: A, B, C, and D. These classes determine the size of the problem, the total number of pseudo-random numbers that are generated. Table 6.1 shows the class specifications.

Listing 6.1 shows the core of the Fortran+MPI version and Listings 6.2, 6.3, and 6.4 show the equivalent code in three ZPL versions. In these versions, `m` is assumed to be the log of the total number of pseudo-random numbers. In addition, the following integers are common to these versions:

```
mk = 16
mm = m - mk
nk = 2**mk = 2^mk
nn = 2**mm = 2^mm
nq = 10
```

The size of each batch of pseudo-random numbers, nk, is the same for all problem classes while the number of batches, nn, increases. The number of counts, nq, is 10.

6.1.1 The Fortran+MPI Version

The Fortran+MPI version of NAS EP, the core of which is shown in Listing 6.1, is a typical, though simple, example of processor-oriented programming. The code tells what each processor computes, obfuscating the overall computation. The problem space is explicitly divided between the processors. Lines 6–16 divide the iteration space, the number of batches of pseudo-random numbers, `nn`, between the processors, `np`. Notice the care that is taken to cope with the situation where `np` does not evenly divide `nn` (Lines 10–16).

The main computation is done in lines 17–38. The norms, `sx` and `sy`, are initialized in lines 17 and 18, and the counts are initialized in lines 19–21. The norms and counts are then accumulated in the main loop (lines 22–38) while the pairs of normally distributed numbers are generated. The final reductions of the norms and counts are computed in lines 39–49.

6.1.2 A ZPL Version Using Free Variables

The ZPL version of NAS EP called ZPL Free, the core of which is shown in Listing 6.2, uses free variables to closely mimic the Fortran+MPI version. This version avoids ZPL's high-level abstractions. In sharp contrast to almost all ZPL programs ever presented, this code does not contain any regions. Nonetheless, it is an important pedantic example of how free variables can be used to create a processor-oriented ZPL program.

Lines 1–9 of ZPL Free show the declarations. They correspond directly to lines 1–4

Listing 6.1: The Fortran+MPI Implementation of EP

```
      double precision t1, t2, t3, t4, x, x1, x2, q, sx, sy
      integer          np, ierr, node, no_nodes, i, kk, l, k,
     >                 no_large_nodes, np_add
      common/storage/ x(2*nk), q(0:nq-1)

      call mpi_init(ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,node,ierr)
      call mpi_comm_size(MPI_COMM_WORLD,no_nodes,ierr)
      np = nn / no_nodes
      no_large_nodes = mod(nn, no_nodes)
      if (node .lt. no_large_nodes) then
         np_add = 1
      else
         np_add = 0
      endif
      np = np + np_add
      sx = 0.d0
      sy = 0.d0
      do 110 i = 0, nq - 1
         q(i) = 0.d0
 110  continue
      do 150 k = 1, np
c        Initialization of x omitted to save space
         do 140 i = 1, nk
            x1 = 2.d0 * x(2*i-1) - 1.d0
            x2 = 2.d0 * x(2*i) - 1.d0
            t1 = x1 ** 2 + x2 ** 2
            if (t1 .le. 1.d0) then
               t2   = sqrt(-2.d0 * log(t1) / t1)
               t3   = (x1 * t2)
               t4   = (x2 * t2)
               l    = max(abs(t3), abs(t4))
               q(l) = q(l) + 1.d0
               sx   = sx + t3
               sy   = sy + t4
            endif
 140     continue
 150  continue
      call mpi_allreduce(sx, x, 1, MPI_DOUBLE_PRECISION, MPI_SUM,
                         MPI_COMM_WORLD, ierr)
      sx = x(1)
      call mpi_allreduce(sy, x, 1, MPI_DOUBLE_PRECISION, MPI_SUM,
                         MPI_COMM_WORLD, ierr)
      sy = x(1)
      call mpi_allreduce(q, x, nq, MPI_DOUBLE_PRECISION, MPI_SUM,
                         MPI_COMM_WORLD, ierr)
      do i = 1, nq
         q(i-1) = x(i)
      enddo
      call mpi_finalize(ierr)
```

of the Fortran+MPI. The ZPL declarations are divided into two parts: variables and free variables. This provides more information to the compiler and the programmer in the ZPL version. It also requires duplicated variables. For example, `sx_part`, `sy_part`, and `q_part` are free counterparts of `sx`, `sy`, and `q`. In this case, the extra memory is not an issue because it is relatively small. In general, such duplicated variables are unlikely to cause problems since scalable parallel programs must limit the amount of replicated memory anyway.

Lines 11–20 of ZPL Free correspond to lines 6–16 of the Fortran+MPI. These lines divide the batches of pseudo-random numbers between the processors. Notice just how similar the codes are. The ZPL contains slightly more information since `nn`, `no_nodes`, and `no_large_nodes` are guaranteed to be identical on each processor. This is the case in the Fortran+MPI code, but a Fortran+MPI compiler would not enforce it even if it could infer it.

The main computation is done in lines 21–40, corresponding to lines 17–38 of the Fortran+MPI. The code here is even more similar. The final reductions are computed in lines 41–43. These three lines of code are equivalent to the eleven lines, lines 39–49, of the Fortran+MPI. Compared to the potential advantages ZPL has over Fortran+MPI, this is a small but moderately important win. As an operator in the language, the reduction is easier to write. Note that the reduction of all the values in array `q` is aggregated into a single reduction in both codes.

6.1.3 A ZPL Version Using Regions and Parallel Arrays

The ZPL version of NAS EP called ZPL Classic, the core of which is shown in Listing 6.3, uses regions and parallel arrays. Unlike the Fortran+MPI and ZPL Free versions, it is p-independent. It is a high-level, potentially efficient version that suffers because the key optimization of array contraction fails to work on all of the parallel arrays.

This ZPL version uses a region to divide the batches of pseudo-random numbers between the processors. The declaration of region `R` in lines 1 and 2 is thus equivalent to lines 11–20 of ZPL Free and lines 6–16 of the Fortran+MPI.

The declarations in lines 3–10 correspond to the previous version though `no_nodes`,

Listing 6.2: The ZPL Free Implementation of EP

```
var
  no_nodes, no_large_nodes : integer;
  sx, sy : double;
  q : array[0..nq-1] of double;
free var
  x : array[1..2*nk] of double;
  q_part : array[0..nq-1] of double;
  sx_part, sy_part, t1, t2, t3, t4, x1, x2 : double;
  node, np_add, np, kk, i, l, k : integer;

no_nodes := numLocales();
node := localeID();
np := nn / no_nodes;
no_large_nodes := nn % no_nodes;
if node < no_large_nodes then
  np_add := 1;
else
  np_add := 0;
end;
np := np + np_add;
sx_part := 0.0;
sy_part := 0.0;
q_part[] := 0.0;
for k := 1 to np do
  -- Initialization of x omitted to save space
  for i := 1 to nk do
    x1 := 2.0 * x[2*i-1] - 1.0;
    x2 := 2.0 * x[2*i] - 1.0;
    t1 := x1^2 + x2^2;
    if t1 <= 1.0 then
      t2        := sqrt(-2.0 * log(t1) / t1);
      t3        := x1 * t2;
      t4        := x2 * t2;
      l         := max(abs(t3), abs(t4));
      q_part[l] := q_part[l] + 1.0;
      sx_part   := sx_part + t3;
      sy_part   := sy_part + t4;
    end;
  end;
end;
sx := +<< sx_part;
sy := +<< sy_part;
q[] := +<< q_part[];
```

`no_large_nodes`, `node`, `np_add`, and `np` are conspicuously absent. In addition, the ZPL Free distinction between variables and free variables has been replaced by a distinction between scalars and parallel arrays. The sums `sx` and `sy`, the counts `q`, and the iterator `i` remain scalars although `i` is no longer free. The rest of the variables are declared as parallel arrays over the iteration space, one value for every batch of pseudo-random numbers. (The memory implications are discussed shortly.) Note that the variables `sx_part`, `sy_part`, and `q_part` have been replaced by the parallel arrays `SX`, `SY`, and `Q`. This harmless duplication of variables and storage is identical to the one mentioned in the ZPL Free version.

The rest of ZPL Classic corresponds closely to the Fortran+MPI and ZPL Free. Lines 12–30 perform the main computation (Fortran+MPI Lines 17–38, ZPL Free Lines 21–40). Lines 31–33 perform the reductions (Fortran+MPI Lines 39–49, ZPL Free Lines 41–43).

The memory require by a naive implementation of ZPL Classic is catastrophically larger than either of the previous versions. Contraction of the parallel arrays is essential. In the current compiler for ZPL, most of the parallel arrays are contracted but several are not. Specifically, `X`, `Q`, `SX`, and `SY` are not contracted.

There are two reasons contraction currently fails. First, fusion is not done across the intrinsic functions `StartTimer` and `StopTimer`. These functions do not appear in Listing 6.3, but in the actual version they surround the initialization of `X` and the loop on Line 17. They thus effectively block the contraction of `X`, `Q`, `SX`, and `SY` since these arrays would have to appear in multiple loops in a scalarized execution. This is a major concern for array languages that do not provide a means for programmers to contract arrays. Even if fusion did span `StartTimer` and `StopTimer`, a reasonable decision for when `StartTimer` does not cause synchronization, it is certainly true that statements should not be moved before or after these timing functions. This immobility can also prevent contraction, though not in this case.

Second, the scalar iterator `i` prevents the statements before and after lines 17 and 30 to be fused since the scalar could potentially become inconsistent across the processors. Unlike the first reason contraction fails, this reason is more implementation dependent. Either live variable analysis or a simple transformation that replicates the scalar portion of the computation could be used by the compiler to fuse the statements.

Listing 6.3: The ZPL Classic Implementation of EP

```
region
  R = [1..nn];
var
  sx, sy : double;
  q : array[0..nq-1] of double;
  i : integer;
  X : [R] array[1..2*nk] of double;
  Q : [R] array[0..nq-1] of double;
  X1, X2, T1, T2, T3, T4, SX, SY : [R] double;
  L : [R] integer;

[R] begin
  SX := 0.0;
  SY := 0.0;
  Q[] := 0.0;
  -- Initialization of X omitted to save space
  for i := 1 to nk do
    X1 := 2.0 * X[2*i-1] - 1.0;
    X2 := 2.0 * X[2*i] - 1.0;
    T1 := X1^2 + X2^2;
    if T1 <= 1.0 then
      T2   := sqrt(-2.0 * log(T1) / T1);
      T3   := X1 * T2;
      T4   := X2 * T2;
      L    := max(abs(T3), abs(T4));
      Q[L] := Q[L] + 1.0;
      SX   := SX + T3;
      SY   := SY + T4;
    end;
  end;
  sx := +<< SX;
  sy := +<< SY;
  q[] := +<< Q[];
end;
```

6.1.4 A ZPL Version Using Grid Dimensions

The ZPL version of NAS EP called ZPL Grid, the core of which is shown in Listing 6.4, uses regions with grid dimensions to manually contract all of the parallel arrays. It is a high-level version like ZPL Classic, but its use of grid dimensions makes it potentially p-dependent.

There are few differences between the ZPL Classic and ZPL Grid versions. In the declarations, `R` is the same but the parallel arrays are declared over `[::]` instead of `R`. Also, the scalar iterator `i` has been promoted to a parallel array declared over `[::]`. In the code, an outer region of `[::]` is used for the initialization and reductions. The statements in the main computation have been interleaved. These simple changes effectively create a single loop in the implementation and thus serve to manually contract the parallel arrays.

Although ZPL Grid is p-dependent, it is qualitatively less p-dependent than both the ZPL Free and the Fortran+MPI codes. It is a simple transformation away from ZPL Classic, a p-independent code. Moreover, this transformation can potentially be automated given the discussion in Section 6.1.3. Unlike in ZPL Free and the Fortran+MPI codes, ZPL Grid does not require the programmer to divide the work and data between the processors. This is done by using the high-level region abstraction.

6.1.5 Performance

Figure 6.1.5 shows speedup graphs comparing the performance of the Fortran+MPI, ZPL Free, and ZPL Grid versions of the NAS EP benchmark on a Cray T3E for the Class C problem size. (The raw data and information on the machine and the compilers is listed in Appendix B.) Note that ZPL Classic required too much memory to run the Class C problem size even on 256 processors of the T3E; the other versions of code use a minimal amount of memory that is independent of the number of processors and the problem size.

The speedup graphs show that the ZPL Free and ZPL Grid codes are competitive with the Fortran+MPI version. The time in the benchmark is split between the initialization of the random numbers and the computation of the Gaussian pairs in a roughly 3:2 ratio. ZPL Free and ZPL Grid are competitive with the Fortran+MPI in both phases. The biggest gap shows up in the Gaussian pairs computation in which ZPL Free suffers the most. (ZPL

Listing 6.4: The ZPL Grid Implementation of EP

```
region
  R = [1..nn];
var
  sx, sy : double;
  q : array[0..nq-1] of double;
  X : [::] array[1..2*nk] of double;
  Q : [::] array[0..nq-1] of double;
  X1, X2, T1, T2, T3, T4, SX, SY : [::] double;
  L, I : [::] integer;

[::] begin
  SX := 0.0;
  SY := 0.0;
  Q[] := 0.0;
  [R] interleave
    -- Initialization of X omitted to save space
    for I := 1 to nk do
      X1 := 2.0 * X[2*I-1] - 1.0;
      X2 := 2.0 * X[2*I] - 1.0;
      T1 := X1^2 + X2^2;
      if T1 <= 1.0 then
        T2   := sqrt(-2.0 * log(T1) / T1);
        T3   := X1 * T2;
        T4   := X2 * T2;
        L    := max(abs(T3), abs(T4));
        Q[L] := Q[L] + 1.0;
        SX   := SX + T3;
        SY   := SY + T4;
      end;
    end;
  end;
  sx := +<< SX;
  sy := +<< SY;
  q[] := +<< Q[];
end;
```

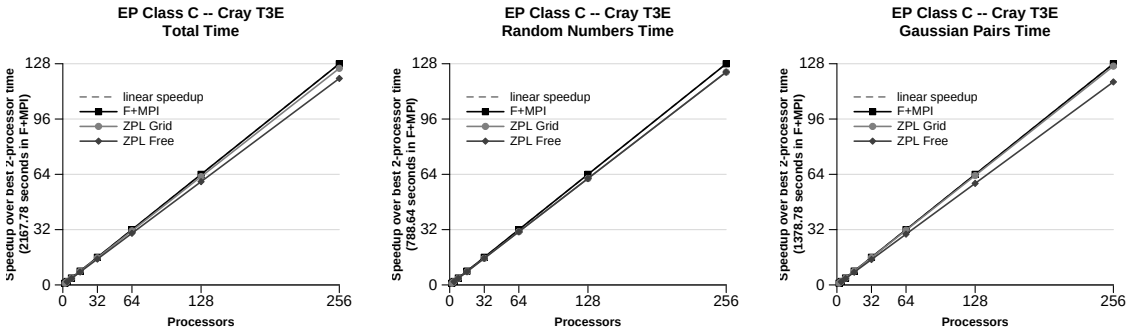

Figure 6.1: Parallel Speedup of NAS EP

Grid achieves performance as high as the Fortran+MPI here.) The slightly-degraded performance in ZPL Free stems from uninteresting low-level implementation details regarding C's array support. Since ZPL is compiled to C, differences between C and Fortran that do not necessarily reflect inherent differences between ZPL and Fortran are manifest in all performance results. There is no reason ZPL Free should perform worse than ZPL Grid.

6.2 Fast Fourier Transform (FT)

The NAS FT (Fast Fourier Transform) benchmark numerically solves a 3D partial differential equation using forward and backward Fast Fourier Transforms (FFTs). The computation involves solving 1D FFTs on each dimension of a 3D array. The basic idea is to always leave at least one dimension of the array local to a processor in order to keep the complicated access patterns required by a 1D FFT from inducing communication.

The FT benchmark comes in four classes: A, B, C, and D. These classes determine the size of the three-dimensional `nx`×`ny`×`nz` arrays and the number of iterations of the computation. Table 6.2 shows the class specifications.

6.2.1 The Fortran+MPI Version

The Fortran+MPI version of NAS FT computes FFTs to solve a partial differential equation using a standard method. There are three main arrays of complex numbers, `u0`, `u1`, and `u2`, as well as an array of real numbers for computing the time evolution, `twiddle`. The

Table 6.2: NAS FT Classes

Class	*nx*	*ny*	*nz*	*iterations*
A	128	256	256	6
B	256	256	512	20
C	512	512	512	20
D	1024	1024	2048	25

main loop of the FT benchmark is shown in Listing 6.5. Initialization involves setting up the time evolution array `twiddle`, initializing array `u1`, and computing the roots of unity for computing one-dimensional FFTs. A forward 3D FFT is then computed on `u1`, and the result is left in `u0`.

The main loop is iterated a number of times depending on the problem class. Array `u0` is evolved to array `u1` using the time evolution array `twiddle`. A backward 3D FFT is then computed on `u1`, and the result is left in `u2`. The last line in the main loop computes a checksum over `u2`. After the main loop, the checksums are verified.

The 3×3 array `dims` contains the per-processor upper bounds of the arrays `u0`, `u1`, `u2`, and `twiddle` for different times in the computation. Depending on the processor layout, specifically whether zero, one, or two dimensions of the parallel arrays are distributed, the arrays may or may not be transposed. In any case, `dims(1,1)`, `dims(2,1)`, and `dims(3,1)` contain the local bounds of an array before the first transpose (if there is one); `dims(1,2)`, `dims(2,2)`, and `dims(3,2)` contain the local bounds of an array after the first transpose (if there is one); and `dims(1,3)`, `dims(2,3)`, and `dims(3,3)` contain the local bounds of an array after the second transpose (if there is one).

To keep the 1D FFT computations local to processors, the FFT procedure is dependent on the layout. Listing 6.6 shows part of the FFT procedure. In a 0D layout, no dimensions are distributed, the 1D FFTs are called on the arrays in each of the three dimensions. The 0D layout only applies to sequential executions. Note that the second dimension of the `dims` array is increased on each call. This corresponds to a logical view of the dimensions changing after each transpose. Since there is no transpose, this is unnecessary; The values in `dims` were initially set to be the same independent of the second dimension.

Listing 6.5: The Fortran+MPI Implementation of FT Main

```
1       call compute_indexmap(twiddle, dims(1,3))
2       call compute_initial_conditions(u1, dims(1,1))
3       call fft_init (dims(1,1))
4       call fft(1, u1, u0)
5       do iter = 1, niter
6          call evolve(u0, u1, twiddle, dims(1, 1))          (sic.)
7          call fft(-1, u1, u2)
8          call checksum(iter, u2, dims(1,1))
9       end do
10      call verify(nx, ny, nz, niter, verified, class)
```

Listing 6.6: The Fortran+MPI Implementation of FT FFT

```
2 c     0D Layout
3       call cffts1(1, dims(1,1), x1, x1, scratch)
4       call cffts2(1, dims(1,2), x1, x1, scratch)
5       call cffts3(1, dims(1,3), x1, x2, scratch)

7 c     1D Layout
8       call cffts1(1, dims(1,1), x1, x1, scratch)
9       call cffts2(1, dims(1,2), x1, x1, scratch)
10      call transpose_xy_z(2, 3, x1, x2)
11      call cffts1(1, dims(1,3), x2, x2, scratch)

13 c     2D Layout
14      call cffts1(1, dims(1,1), x1, x1, scratch)
15      call transpose_x_y(1, 2, x1, x2)
16      call cffts1(1, dims(1,2), x2, x2, scratch)
17      call transpose_x_z(2, 3, x2, x1)
18      call cffts1(1, dims(1,3), x1, x2, scratch)
```

In the 1D and 2D layouts, the changes in the second dimension of dims are critical. The 2D layout, the second and third dimensions are distributed, applies when the program is run on more processors than could be used to distribute the third dimension; otherwise the 1D layout applies, and only the third dimension is distributed. In the 1D layout, 1D FFTs are applied to the first and second dimensions of the array. The array is then transposed and a final set of 1D FFTs are applied to the first dimension, which now contains the logical values of the third dimension. In the 2D layout, the array is transposed between the application of the FFTs to each dimension.

Listing 6.7: The ZPL Implementation of FT Declarations

```
1  grid
2    G = [np1, np2, 1];

4  distribution
5    DXYZ : G = [blk(0,nx-1), blk(0,ny-1), blk(0,nz-1)];
6    DYZX : G = [blk(0,ny-1), blk(0,nz-1), blk(0,nx-1)];
7    DXZY : G = [blk(0,nx-1), blk(0,nz-1), blk(0,ny-1)];

9  region
10   RXYZ : DXYZ = [0..nx-1, 0..ny-1, 0..nz-1];
11   RYZX : DYZX = [0..ny-1, 0..nz-1, 0..nx-1];
12   RXZY : DXZY = [0..nx-1, 0..nz-1, 0..ny-1];

14 var
15   DU0       : [G]         distribution<block,block,block>;
16   RU0       : [DU0]       region<..,..,..>;
17   U0        : [RU0]       dcomplex;
18   DU1       : [G]         distribution<block,block,block>;
19   RU1       : [DU1]       region<..,..,..>;
20   U1        : [RU1]       dcomplex;
21   DU2       : [G]         distribution<block,block,block>;
22   RU2       : [DU2]       region<..,..,..>;
23   U2        : [RU2]       dcomplex;
24   DTwiddle  : [G]         distribution<block,block,block>;
25   RTwiddle  : [DTwiddle]  region<..,..,..>;
26   Twiddle   : [RTwiddle]  double;
```

Note that the code shown in Listing 6.6 is only for the forward FFT. The backward FFT is omitted. There is a different transpose procedure call in the backward FFT and the logic is reversed, but symmetric. The code to transpose the arrays is nontrivial and is omitted due to its length (206 lines of code). In the ZPL versions that follow, the transposes, which involve 26 and 6 lines of code, are shown.

6.2.2 A ZPL Version Using Distribution Destructive Assignment

The ZPL version of FT uses dynamic regions and distributions to closely mimic the ideas in the Fortran+MPI version. The next section will show a second ZPL version of FT that is also in the spirit of the NAS version but which makes several substantial changes.

The Fortran+MPI version of FT made the assumption that each processor owns the same number of elements of each of the arrays `u0`, `u1`, `u2`, and `twiddle` independent of the

Listing 6.8: The ZPL Implementation of FT Main

```
1   RU0 <== RXYZ; DU0 <== DXYZ;
2   RU1 <== RXYZ; DU1 <== DXYZ;
3   RU2 <== RXYZ; DU2 <== DXYZ;
4   compute_indexmap(DTwiddle, RTwiddle, Twiddle);
5   compute_initial_conditions(U1);
6   fft_init();
7   fft(1, DU1, RU1, U1, DU0, RU0, U0);
8   for iter := 1 to niter do
9     if np1 > 1 | np2 > 1 then
10      DU1 <== DYZX; RU1 <== RYZX;
11    end;
12    [RU1] evolve(U0, U1, Twiddle);
13    fft(-1, DU1, RU1, U1, DU2, RU2, U2);
14    checksum(iter, U2);
15  end;
16  verified := verify();
```

array's orientation, *i.e.*, whether before or after any given transpose. The ZPL version makes no such assumption. There are still some restrictions on the number and arrangement of processors that can be used to run the benchmark because the 1D FFTs are still computed in blocks. Nevertheless, the ZPL version is not restricted to a power-of-two number of processors.

The main declarations are listed in Listing 6.7. The main arrays `U0`, `U1`, `U2`, and `Twiddle` are declared over their own variable regions and distributions. They are allocated, and reallocated for transposing, by applying destructive assignment on their distributions and regions and using the remap operator to move the data to the redistributed array. The constant grid is never changed. The constant distributions and regions are always the source of the destructive assignments to the distributions and regions of the array.

Listing 6.8 shows the main loop for the FT procedure of the FT benchmark. Arrays `U0`, `U1`, and `U2` are initialized in lines 1–3. Constant distributions `DXYZ`, `DXZY`, and `DYZX` and constant regions `RXYZ`, `RXZY`, and `RYZX` correspond to `dims(*,1)`, `dims(*,2)`, and `dims(*,3)` in the Fortran+MPI. The arrays are initially set to their pre-transpose orientation. Note that the Fortran's column major layout and ZPL's row major layout should be considered when comparing the Fortran+MPI and ZPL versions of FT. The `Twiddle` array is initialized with the `RXYZ` or `RYZX` region and corresponding distribution depending on whether a 0D

Listing 6.9: The ZPL Implementation of FT FFT

```
1  -- 0D Layout
2  cffts3(dir, nz, X1, X1, x, y);
3  cffts2(dir, ny, X1, X1, x, y);
4  cffts1(dir, nx, X1, X2, x, y);

6  -- 1D Layout
7  cffts3(dir, nz, X1, X1, x, y);
8  cffts2(dir, ny, X1, X1, x, y);
9  DX2 <== DYZX; RX2 <== RYZX;
10 [RX2] X2 := X1#[Index3, Index1, Index2];
11 DX1 <== DYZX; RX1 <== RYZX;
12 cffts3(dir, nx, X2, X2, x, y);

14 -- 2D Layout
15 cffts3(dir, nz, X1, X1, x, y);
16 DX2 <== DXZY; RX2 <== RXZY;
17 [RX2] X2 := X1#[Index1, Index3, Index2];
18 cffts3(dir, ny, X2, X2, x, y);
19 DX1 <== DYZX; RX1 <== RYZX;
20 [RX1] X1 := X2#[Index3, Index2, Index1];
21 DX2 <== DYZX; RX2 <== RYZX;
22 cffts3(dir, nx, X1, X2, x, y);
```

or a 1D or 2D layout applies. The rest of the main loop is similar to the Fortran except instead of reading `dims`, the ZPL reads and destructively assigns distributions and regions.

Listing 6.9 shows code for the forward FFTs in the FFT procedure. This code closely mimics the Fortran+MPI. However, it should be noted that the 206 lines of code used to transpose the arrays have been replaced by 26 lines of code, 13 of which are omitted being in the symmetric backward FFT.

6.2.3 A ZPL Version Using Grid Preserving Assignment

This section presents another ZPL version of FT called ZPL Grid. This version uses preserving grid assignment to change the grid and distribute different dimensions of the array. It is logically cleaner, and as we will see, results in faster code.

The declarations for ZPL Grid are shown in Listing 6.10. In contrast to the previous ZPL version, the arrays in this version have variable grids, distributions, and regions. Only the grid is ever changed. It is assigned the constant grids `G1`, `G2`, or `G3`. The future work section of this thesis will discuss a way around declaring the variable distributions and

Listing 6.10: The ZPL Grid Implementation of FT Declarations

```
1  grid
2    G3 = [np1, np2, 1  ];
3    G2 = [np2,   1, np1];
4    G1 = [  1, np2, np1];

6  distribution
7    D = [blk(0,nx-1), blk(0,ny-1), blk(0,nz-1)];

9  region
10   R = [0..nx-1, 0..ny-1, 0..nz-1];

12 var
13   GU0       :              grid<..,..,..>;
14   DU0       : [GU0]        distribution<block,block,block> = D;
15   RU0       : [DU0]        region<..,..,..> = R;
16   U0        : [RU0]        dcomplex;
17   GU1       :              grid<..,..,..>;
18   DU1       : [GU1]        distribution<block,block,block> = D;
19   RU1       : [DU1]        region<..,..,..> = R;
20   U1        : [RU1]        dcomplex;
21   GTwiddle  :              grid<..,..,..>;
22   DTwiddle  : [GTwiddle]   distribution<block,block,block> = D;
23   RTwiddle  : [DTwiddle]   region<..,..,..> = R;
24   Twiddle   : [RTwiddle]   double;
```

regions that are never used.

Figures 6.11 and 6.12 show the ZPL code that corresponds to the code shown for the Fortran+MPI and ZPL versions. The logical simplification can be seen most easily in Listing 6.12. For each layout, the same calls to the 1D FFT procedure are made. This contrasts to the code that computed FFTs on the same dimension of an array multiple times because it contained data from a different logical dimension.

The grid change is also significantly simpler than the remap transpose—6 lines of code versus 26 lines of code. Indeed, it is a bit of a puzzle to write the transposes of the previous versions. As an example, consider that the 1D layout forward transpose, in which the data in an `nx`×`ny`×`nz` array `B` is moved to an `ny`×`nz`×`nx` array `A`, is given by that statement

```
[1..ny, 1..nz, 1..nx] A := B#[Index3, Index1, Index2];
```

rather than the statement

```
[1..ny, 1..nz, 1..nx] A := B#[Index2, Index3, Index1];
```

Listing 6.11: The ZPL Grid Implementation of FT Main

```
1   GU0 <== G3;
2   GU1 <== G3;
3   compute_indexmap(GTwiddle, Twiddle);
4   compute_initial_conditions(U0);
5   fft_init();
6   fft(1, GU0, U0);
7   for iter := 1 to niter do
8     if np1 > 1 | np2 > 1 then
9       GU1 <== G1;
10    end;
11    evolve(U0, U1, Twiddle);
12    fft(-1, U1, GU1);
13    checksum(iter, U1);
14  end;
15  verified := verify();
```

To get the logical data in the first dimension local to a processor, changing the `p×1×1` grid using the statement

```
G <=# [1, p, 1];
```

is significantly simpler to reason about.

6.2.4 Performance

Figure 6.2.4 shows speedup graphs comparing the performance of the Fortran+MPI, ZPL, and ZPL Grid versions of the NAS FT benchmark on a Cray T3E for the Class C problem size. (The raw data and information on the machine and the compilers is listed in Appendix B.) The speedup graphs show the total time and the time of the benchmark divided into the significant parts. The setup time accounts for the initialization of the array, the initialization of the time evolution array, and the computation of the roots of unity used in the FFT computation. The evolve time accounts for the evolution of the array between iterations. The FFT time is broken into two parts: FFT Low accounts for the time in the actual FFT computation and FFT Copy accounts for the time to move blocks of data from the main array to a scratch array. The transpose time accounts for the time to transpose the arrays in the 1D and 2D layout. In ZPL Grid, the transpose time accounts for the logically equivalent grid change time.

Listing 6.12: The ZPL Grid Implementation of FT FFT

```
1  -- 0D Layout
2  cffts3(dir, nz, X, x, y);
3  cffts2(dir, ny, X, x, y);
4  cffts1(dir, nx, X, x, y);

6  -- 1D Layout
7  cffts3(dir, nz, X, x, y);
8  cffts2(dir, ny, X, x, y);
9  GX <=# G1;
10 cffts1(dir, nx, X, x, y);

12 -- 2D Layout
13 cffts3(dir, nz, X, x, y);
14 GX <=# G2;
15 cffts2(dir, ny, X, x, y);
16 GX <=# G1;
17 cffts1(dir, nx, X, x, y);
```

Overall, the versions are competitive with one another. The ZPL Grid and Fortran+MPI versions both outperform the ZPL version by a small margin. The roughly equal advantages, however, stem from different sections of the code. The setup time accounts for a small amount of the total time. The Fortran+MPI version greatly outperforms both ZPL versions, which are identical in this section of code. The modular implementation of the random initialization, discussed in Section 4.6, is seen to be moderately slower.

The evolve time is interesting, though it also accounts for a small amount of the total time. Both ZPL and ZPL Grid perform marginally better than the Fortran+MPI, though ZPL Grid degrades on higher numbers of processors. This code should be completely scalable for all three versions. However, the implementation of destructive grid assignment in the MPI implementation of ZPL sets up an MPI communicator for the entire grid, resulting in a significant but limited amount of communication. This exception to ZPL's performance model was discussed in Chapter 3.

The FFT computation is similar in all three versions. ZPL Grid uses a different set of FFT copy functions than the other versions because it changes the underlying grid rather than transposing the arrays. The Fortran+MPI and ZPL versions rely more heavily on a copy that blocks the middle dimension and copies the inner dimension. This copy is slightly

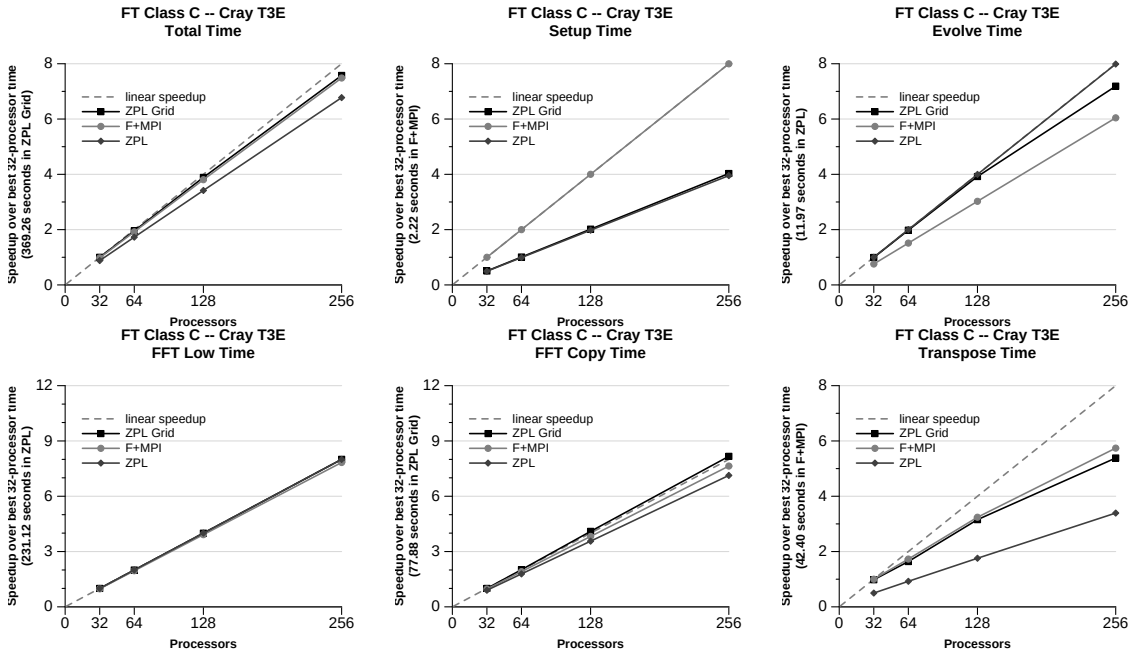

Figure 6.2: Parallel Speedup of NAS FT

slower than the other copies because the dimensions traversed are switched between the scratch and main arrays.

The transpose code shows the advantage of ZPL Grid over ZPL. In this case, ZPL Grid is on par with the Fortran+MPI and even slightly ahead on 256 processors. This is significant especially when considering that the Fortran+MPI transpose code blocks data to achieve better performance. The ZPL codes rely on several remap optimizations that have profound effects. In both ZPL versions, the indexes are exchanged only once for the forward and backward transposes (by saving and reusing the maps) and run length encoding minimizes the communication and makes for efficient loops over the indexes. The performance could be improved further by avoiding this exchange of indexes all together and taking advantage of the basic `Index`i maps; this further improvement is small since the exchange of indexes is greatly amortized even on the small number of iterations that the benchmark is run.

Both the ZPL and ZPL Grid versions of the code are able to take advantage of a remap optimization that avoids allocating extra memory for buckets for exchanging the data. In

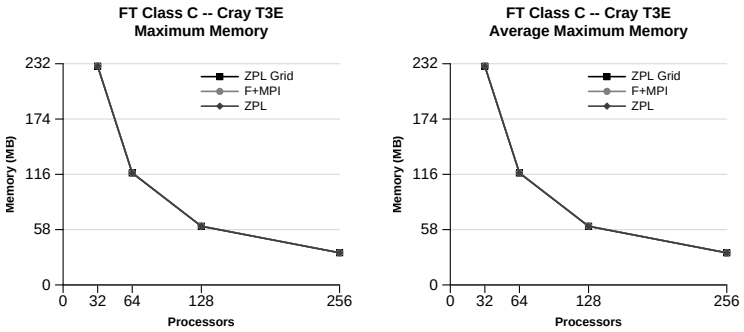

Figure 6.3: Memory Usage of NAS FT

addition, the direct sending and receiving optimization applies to the ZPL Grid version for the backwards and forwards grid changes. Only one applies to each, so data shuffling is only done on one side of the communication rather than both as is done in the Fortran+MPI and ZPL versions.

Figure 6.2.4 shows the amount of memory used by each of the versions of FT. The memory graphs show the maximum memory used by any of the processors and the average maximum memory used by all of the processors. These graphs are equivalent and show perfect scaling for all three benchmark implementations. The memory management for ZPL Grid is more complicated because of the destructive and preserving grid assignment statements, although this does not show up here. This issue will be discussed further in the performance discussion of the NAS IS benchmark.

6.3 Integer Sort (IS)

The NAS IS (Integer Sort) benchmark implements a bucket sort algorithm on normally distributed positive integers. The integers are divided into buckets, the buckets are distributed across the processors, and one major communication phase is used to shuffle the integers between the processors so that the integers are partially sorted. The benchmark is iterated ten times. Within each iteration, the partial sorting takes place and all the integers are ranked. The actual sorting of the data is only done for the last iteration for full verification.

The IS benchmark comes in three classes: A, B, and C. These classes determine the

Table 6.3: NAS IS Classes

Class	*Total Keys*	*Maximum Key*	*Number of Buckets*
A	2^{23}	2^{19}	2^{10}
B	2^{25}	2^{21}	2^{10}
C	2^{27}	2^{23}	2^{10}

number and range of integers that are sorted and the number of buckets that should be used. Table 6.3 shows the class specifications. Note that the maximum value of a key is four orders of magnitude less than the number of keys, allowing for a radix-style optimization of the parallel bucket sort.

Unlike the other NAS benchmarks, which are written in Fortran+MPI, the NAS provided version of IS is written in C+MPI. This section describes the C+MPI version and three ZPL versions. Two portions of the code are shown for each version: the core of the rank procedure and the full verify procedure. The rank procedure constitutes the major portion of the benchmark. In it, the keys are first repartitioned across the processors so that they are partially sorted, *i.e.*, the keys on any given processor are greater than the keys on the lower processors and less than the keys on the higher processors. In the sort procedure, keys are sorted based on their ranks and then the whole array is examined to make sure it has been sorted correctly.

6.3.1 The C+MPI Version

There are three large arrays in the C+MPI program: `key_array`, `key_buff1`, and `key_buff2`. These arrays are allocated with extra space on each processor to deal with the situation after sorting when the problem is not evenly load-balanced. When running the program on less than 256 processors, the buffers are fifty percent bigger than necessary. When running on 256 or more processors, the buffers are 500% bigger than necessary.

The rank procedure, shown in Listing 6.13, is typical of C+MPI programs, being low-level and optimized for its special case. The first step of the rank procedure is to determine the number of keys that will fall into each bucket. Lines 1–5 of Listing 6.13 compute this first step. Two arrays of sums are computed: `bucket_size` and `bucket_size_totals`. The

Listing 6.13: The C+MPI Version of IS Rank

```
for (i=0; i<NUM_BUCKETS; i++)
  bucket_size[i] = 0;
for (i=0; i<NUM_KEYS; i++)
  bucket_size[key_array[i] >> shift]++;
MPI_Allreduce(bucket_size, bucket_size_totals, NUM_BUCKETS, ...);
bucket_sum_accumulator = 0;
local_bucket_sum_accumulator = 0;
send_displ[0] = 0;
tmp_bucket_ptr1[0] = 0;
for( i=0, j=0; i<NUM_BUCKETS; i++ ) {
  bucket_sum_accumulator       += bucket_size_totals[i];
  local_bucket_sum_accumulator += bucket_size[i];
  if( bucket_sum_accumulator >= (j+1)*NUM_KEYS ) {
    send_count[j] = local_bucket_sum_accumulator;
    if( j != 0 ) {
      send_displ[j] = send_displ[j-1] + send_count[j-1];
      tmp_bucket_ptr1[j] = tmp_bucket_ptr2[j-1]+1;
    }
    tmp_bucket_ptr2[j++] = i;
    local_bucket_sum_accumulator = 0;
  }
}
min_key_val = tmp_bucket_ptr1[my_rank] << shift;
max_key_val = ((tmp_bucket_ptr2[my_rank] + 1) << shift)-1;
m = 0;
for(k=0; k<my_rank; k++)
  for(i=tmp_bucket_ptr1[k]; i<=tmp_bucket_ptr2[k]; i++)
    m += bucket_size_totals[i];
j = 0;
for(i=tmp_bucket_ptr1[my_rank]; i<=tmp_bucket_ptr2[my_rank]; i++)
  j += bucket_size_totals[i];
bucket_ptrs[0] = 0;
for(i=1; i<NUM_BUCKETS; i++)
  bucket_ptrs[i] = bucket_ptrs[i-1] + bucket_size[i-1];
for(i=0; i<NUM_KEYS; i++) {
  key_buff1[bucket_ptrs[key_array[i] >> shift]++] = key_array[i];
}
MPI_Alltoall(send_count, 1, ..., recv_count, 1, ...);
recv_displ[0] = 0;
for( i=1; i<comm_size; i++ )
  recv_displ[i] = recv_displ[i-1] + recv_count[i-1];
MPI_Alltoallv(key_buff1, send_count, send_displ, ...,
              key_buff2, recv_count, recv_displ, ...);
for(i=0; i<max_key_val-min_key_val+1; i++)
  key_buff1[i] = 0;
key_buff_ptr = key_buff1 - min_key_val;
for( i=0; i<j; i++ )
  key_buff_ptr[key_buff2[i]]++;
key_buff_ptr[min_key_val] += m;
for(i=min_key_val; i<max_key_val; i++)
  key_buff_ptr[i+1] += key_buff_ptr[i];
```

`bucket_size` array contains different values on each processor. Namely, each processor's copy of `bucket_size` contains the number of keys that it will contribute to each bucket. This is calculated by right-shifting the keys by `shift`, the difference between the log of the maximum key value and the log of the number of buckets. This divides the keys into buckets based on the high-order bits. The `bucket_size_totals` array, on the other hand, contains the total number of keys that will be contributed to each bucket

Line 6–31 compute several important values. Each processor's copy of `min_key_val` and `max_key_val` contains the minimum and maximum values that it will hold after the keys are partially sorted. Each processor's copy of `m` contains the number of keys stored on the processors containing the lower buckets. Lastly, each processor's copy of `j` contains the total number of keys that it will contain after the sort. Note that the sort may result in a distribution of the keys that is not perfectly load balanced like before the sort. In addition to `min_key_val`, `max_key_val`, `m`, and `j`, these lines of code compute counts and displacements used by the `MPI_Alltoallv` function.

Lines 32–37 partially sort the local keys from `key_array` into `key_buff1`. After these lines, the keys in `key_buff1` that will be sent to the same processors are blocked together and ready to be sent using `MPI_Alltoallv`. The previously computed array `send_displ` contains the offsets into these blocks.

In line 38, the number of elements each processor is sending to each other processor are exchanged so that each processor now knows how many keys to expect. In lines 39–41, the `recv_displ` array is computed. It corresponds to the `send_displ` array and is set to receive the data in blocks depending on the processors. The call to `MPI_Alltoallv` does the actual rearrangement of keys between processors.

The ranking of keys on each processor is done in lines 44–51. The array `key_buff1` will contain the ranks of all the numbers between `min_key_val` and `max_key_val`. Note that because the range of values of keys is less than the number of keys, this is generally less than the number of keys divided by the number of processors. The loop of lines 47–48 counts the number of keys for each value. The number of keys on lower processors, `m`, is added to the ranks in the loop of lines 50–51.

Listing 6.14: The C+MPI Version of IS Full Verify

```
1  for(i=0; i<total_local_keys; i++)
2    key_array[--key_buff_ptr_global[key_buff2[i]]-
3                                total_lesser_keys] = key_buff2[i];
4  if (my_rank > 0)
5    MPI_Irecv(&k, 1, ..., my_rank-1, 1000, ...);
6  if (my_rank < comm_size-1)
7    MPI_Send(&key_array[total_local_keys-1], 1,
8             ..., my_rank+1, 1000, ...);
9  if (my_rank > 0)
10   MPI_Wait(...);
11 j = 0;
12 if (my_rank > 0)
13   if (k > key_array[0])
14     j++;
15 for (i=1; i<total_local_keys; i++)
16   if (key_array[i-1] > key_array[i])
17     j++;
18 if (j == 0)
19   verified++;
```

The full verify procedure is shown in Listing 6.14. The loop in lines 1–3 sorts the local data. The variables computed in the rank procedure are given more descriptive names: `j` is `total_local_keys`, `m` is `total_lesser_keys`, and `key_buff_ptr` which points into `key_buff1` is `key_buff_ptr_global`.

Lines 4–17 make sure the array is correctly sorted. First the last element on each processor is compared against the first element on the logically next processor. Then, in lines 15–17, the local portion of each array is checked.

6.3.2 A ZPL Version Using A P-Size Region

The first ZPL version of IS, called ZPL Pin, uses a p-size region, *i.e.*, a region containing the indexes between zero and one less than the number of processors, to store the keys and ranks after the sorting process. Listing 6.15 shows the core of the rank procedure for ZPL Pin and Listing 6.16 shows the full verify procedure for ZPL Pin. Tables 6.4 and 6.5 show the declarations of selected variables in this version of code as they compare to the C+MPI. The C+MPI declarations are listed in the first column; the ZPL Pin declarations are listed in the second column.

Table 6.4: IS Variables of C+MPI, ZPL Pin, ZPL Free, ZPL Cut

C+MPI	ZPL Pin	ZPL Free	ZPL Cut
int key_array [SIZE_OF_BUFFERS]	KeyArray : [R] **integer**	KeyArray : [R] **integer**	KeyArray : [K] **integer**
int key_buff1 [SIZE_OF_BUFFERS]	KeyBuff1 : [P] **array** [0..sizeOfBuffers−1] **of integer**	*free* keyBuff1 : **array** [0..sizeOfBuffers−1] **of integer**	DBuff1 : **distribution**<**block**> RBuff1 : [DBuff1] **region** = V KeyBuff1 : [RBuff1] **integer**
int key_buff2 [SIZE_OF_BUFFERS]	KeyBuff2 : [P] **array** [0..sizeOfBuffers−1] **of integer**	*free* keyBuff2 : **array** [0..sizeOfBuffers−1] **of integer**	DBuff2 : **distribution**<**block**> RBuff2 : [DBuff2] **region** = K KeyBuff2 : [RBuff1] **integer**
	KeyBuff3 : [P] **array** [0..sizeOfBuffers−1] **of integer**	*free* keyBuff3 : **array** [0..sizeOfBuffers−1] **of integer**	
int bucket_size [NUM_BUCKETS]	*free* bucketSize : **array** [0..numBuckets−1] **of integer**	*free* bucketSize : **array** [0..numBuckets−1] **of integer**	*free* bucketSize : **array** [0..numBuckets−1] **of integer**
int bucket_size_totals [NUM_BUCKETS]	bucketSizeTotals : **array** [0..numBuckets−1] **of integer**	bucketSizeTotals : **array** [0..numBuckets−1] **of integer**	bucketSizeTotals : **array** [0..numBuckets−1] **of integer**

Table 6.5: More IS Variables of C+MPI, ZPL Pin, ZPL Free, ZPL Cut

C+MPI	**ZPL Pin**	**ZPL Free**	**ZPL Cut**
int m	M : [P] **integer**	*free* m : **integer**	*free* m : **integer**
int min_key_val	MinKeyVal : [P] **integer**	*free* minKeyVal : **integer**	
int max_key_val	MaxKeyVal : [P] **integer**	*free* maxKeyVal : **integer**	cutsMaxKey : **array** [0..numLocales()−1] **of integer**

ZPL Pin is written at a higher level than the C+MPI code. There are two distribution and region pairs given by the following declarations:

```
config const numProcs : integer = numLocales();

distribution DR = [blk(0,totalKeys-1)];
             DP = [blk(0,numProcs-1)];

region R = [0..totalKeys-1];
       P = [0..numProcs-1];
```

Region `R` is a high-level p-independent region whose size is based on the number of keys, the problem size, not the number of keys divided by the number of processors. On the other hand, region `P` is a p-dependent region since its size is based on the number of processors. Its p-dependency can be changed at the command line by overriding the configuration constant `numProcs`.

Note that because `KeyArray` is declared over `R`, it uses no extra space, unlike `key_array` in the C+MPI version. The two other large arrays in this version of ZPL IS, `KeyBuff1` and `KeyBuff2`, are declared over `P` with an indexed array base type with the same amount of extra space as in the C+MPI version. A fourth large array, `KeyBuff3`, not necessary in the C+MPI versions, is used in this ZPL version in the full verify procedure. Memory usage will be discussed further in Section 6.3.5.

The ZPL Pin version of the rank procedure is similar to the C+MPI version but exhibits important differences. In the code in Listing 6.15, there is one free variable, `bucketSize`. The other variables, also local to a processor, are instead declared over region `P` since each index of `P` is meant to be distributed to a different processor, though it is not necessary that this is the case.

Lines 1–3 of ZPL Pin are equivalent to lines 1–5 of the C+MPI. The `bucketSize` indexed array is free rather than being a parallel array of indexed arrays declared over `P` because it must interact with the array `KeyArray` declared over `R` and `R` and `P` have different distributions. If `numProcs` is `numLocales()`, the effect is the same. If not, `bucketSize` is meaningless. The sum array, `bucketSizeTotals`, is the same in either case.

Lines 4–22 compute a similar set of values to those computed in lines 6–31 of the C+MPI. The variable `MinKeyVal` is equivalent to `min_key_val`, `MaxKeyVal` is equivalent

Listing 6.15: The ZPL Pin Version of IS Rank

```
1  bucketSize[] := 0;
2  [R] bucketSize[bsr(KeyArray,shift)] += 1;
3  bucketSizeTotals[] := +<< bucketSize[];
4  bucketSumAccumulator := 0;
5  bucketProc[] := 0;
6  proc := 0;
7  [P] M := 0;
8  [0] MinKeyVal := 0;
9  for i := 0 to numBuckets - 1 do
10   bucketSumAccumulator += bucketSizeTotals[i];
11   bucketProc[i] := proc;
12   [P] if proc < Index1 then
13     M += bucketSizeTotals[i];
14   end;
15   if bucketSumAccumulator >= (proc + 1) * numKeys then
16     [P] MaxKeyVal := bsl(i+1,shift) - 1;
17     proc += 1;
18     if proc < numProcs then
19       [proc] MinKeyVal := bsl(i+1,shift);
20     end;
21   end;
22 end;
23 [P] J := -1;
24 [R] KeyBuff2[inc(J)]#[bucketProc[bsr(KeyArray,shift)]] := KeyArray;
25 [P] for I := 0 to MaxKeyVal - MinKeyVal do
26   KeyBuff1[I] := 0;
27 end;
28 [P] for I := 0 to J - 1 do
29   KeyBuff1[KeyBuff2[I] - MinKeyVal] += 1;
30 end;
31 [P] KeyBuff1[0] += M;
32 [P] for I := 1 to MaxKeyVal - MinKeyVal do
33   KeyBuff1[I] += KeyBuff1[I - 1];
34 end;
```

to `max_key_val`, and `M` is equivalent to `m`. The array `bucketProc` contains an index number for the region `P`. This identifies which processor, assuming `numProcs` is equivalent to `numLocales()`, the bucket should map to. This is a smaller computation than the equivalent C+MPI. Notably, the local counts and displacements are not calculated. In addition, the value of `j`, the total number of keys each processor will have after the sort is completed, is not computed.

Lines 23–24 in ZPL Pin are equivalent to lines 29–43 of the C+MPI. The advantage of the higher-level ZPL code is evident in the savings of the thirteen lines here. The ZPL remap operator is used to move the values from `KeyArray` into `KeyBuff2`. The map, specified by the `bucketProc` array, says to which processor, or which index in `P`, the data in `KeyArray` is moved to. The parallel array `J` is incremented within the index of the parallel array of indexed arrays `KeyBuff2`. The value in `J` local to `KeyBuff2` is incremented for every value that is mapped to the same location.

There is a trade-off in the high-level ZPL version. Although it is significantly shorter, cleaner, and easier to write, it contains some inherent inefficiencies. The most inefficient part of the statement, and what hurts performance the most, is that the indexes are exchanged between processors as is standard in all remaps. However, if `numProcs` is equal to `numLocales()`, there is only one index in each processor. Although ZPL's run length encoding reduces the volume of communication, the cost of the run length encoding and the small amount of communication is significant enough especially considering that it should be completely avoided. In the next ZPL version of IS, discussed shortly, it will be shown how to avoid this inefficiency. There are several other small inefficiencies in the version since the implementation of the remap operator requires recomputing some of the values calculated in lines 4–22 of the ZPL source.

The final portion of the code, lines 25–34, computes the ranks of the keys on each processor. It is equivalent to lines 44–51 of the C+MPI. The comparison is straight-forward.

Listing 6.16 shows the full verify code for the ZPL Pin version. The comparison between the C+MPI and this code is more straight-forward than it was for the rank code. Lines 2–5 sort the integers. In the ZPL version, a third buffer is used. It has the same extra elements as the other buffer arrays and it is necessary since the values are now rebalanced. In the

Listing 6.16: The ZPL Pin Version of IS Full Verify

```
[P] begin
  for I := 0 to TotalLocalKeys-1 do
    KeyBuff3[dec(KeyBuff1[KeyBuff2[I]-GlobalMinKeyVal])-
                                      TotalLesserKeys] := KeyBuff2[I];
  end;
  K := KeyBuff3[TotalLocalKeys-1];
  [1..numProcs-1] K := K@[-1];
  J := 0;
  if Index1 != 0 then
    if K > KeyBuff3[0] then
      J += 1;
    end;
  end;
  for I := 1 to TotalLocalKeys - 1 do
    if KeyBuff3[I-1] > KeyBuff3[I] then
      J += 1;
    end;
  end;
  if J = 0 then
    Verified += 1;
  end;
end;
```

last version of IS in ZPL, called ZPL Cut, this extra buffer will be eliminated by using an array that changes from a block distribution to a cut distribution.

The rest of the sort procedure is used to verify that the array is sorted. The @ operator is used rather than MPI sends and receives. The advantage of clarity is gained, and in addition, race conditions and deadlocks possible with the MPI, is impossible in the ZPL.

6.3.3 A ZPL Version Using Free Indexed Arrays

The second ZPL version of IS, called ZPL Free, uses free indexed arrays rather than parallel arrays of indexed arrays declared over the p-size region `P`. It uses the same variable `KeyArray` and region `R` to achieve a higher-level version. Listing 6.17 shows the core of the rank procedure for ZPL Free and Listing 6.18 shows the full verify procedure for ZPL Free.

Due to the similarity of the ZPL Pin and ZPL Free versions, there is no reason to walk through the code. Instead, note the implications of the difference. First, there are more free scalars. All of the variables in ZPL Pin declared as parallel arrays over `P` have been

Listing 6.17: The ZPL Free Version of IS Rank

```
1  bucketSize[] := 0;
2  [R] bucketSize[bsr(KeyArray,shift)] += 1;
3  bucketSizeTotals[] := +<< bucketSize[];
4  bucketSumAccumulator := 0;
5  bucketProc[] := 0;
6  proc := 0;
7  m := 0;
8  minKeyVal := 0;
9  for i := 0 to numBuckets - 1 do
10   bucketSumAccumulator += bucketSizeTotals[i];
11   bucketProc[i] := proc;
12   if proc < localeID() then
13     m += bucketSizeTotals[i];
14   end;
15   if bucketSumAccumulator >= (proc + 1) * numKeys then
16     maxKeyVal := bsl(i+1,shift) - 1;
17     proc += 1;
18     if proc = localeID() then
19       minKeyVal := bsl(i+1,shift);
20     end;
21   end;
22 end;
23 j := -1;
24 [R] keyBuff2[inc(j)]#[bucketProc[bsr(KeyArray,shift)]] := KeyArray;
25 for fi := 0 to maxKeyVal - minKeyVal do
26   keyBuff1[fi] := 0;
27 end;
28 for fi := 0 to j - 1 do
29   keyBuff1[keyBuff2[fi] - minKeyVal] += 1;
30 end;
31 keyBuff1[0] += m;
32 for fi := 1 to maxKeyVal - minKeyVal do
33   keyBuff1[fi] += keyBuff1[fi - 1];
34 end;
```

Listing 6.18: The ZPL Free Version of IS Full Verify

```
1  for i := 0 to totalLocalKeys-1 do
2    keyBuff3[dec(keyBuff1[keyBuff2[i]-globalMinKeyVal])-
3                                    totalLesserKeys] := keyBuff2[i];
4  end;
5  k := keyBuff3[totalLocalKeys-1];
6  k := k@[-1];
7  j := 0;
8  if localeID() > 0 then
9    if k > keyBuff3[0] then
10     j += 1;
11   end;
12 end;
13 for i := 1 to totalLocalKeys - 1 do
14   if keyBuff3[i-1] > keyBuff3[i] then
15     j += 1;
16   end;
17 end;
18 if j = 0 then
19   verified += 1;
20 end;
```

replaced by free scalars. Tables 6.4 and 6.5 compare declarations from the versions. The ZPL Pin variables are declared in the second column; the ZPL Free variables are declared in the third column.

A good question to ask is whether this basic replacement makes the program more p-dependent. For the most part, no. Because the region `P` has a p-dependent upper bound, all of the arrays declared over `P` are inherently p-dependent. One small advantage to ZPL Pin is that errors can be isolated more easily by setting the value in `numProcs` to a value other than the number of processors. For example, if `numProcs` is two in a one-processor execution and the program works, seeing that program fails on two processors limits the site of the failures to the uses of typical p-dependent abstractions such as the free indexed array `bucketSize`.

The major advantage to ZPL Free, and one which has a substantial impact on performance, is the change to the remap operator. The map no longer specifies indexes into `P` but rather specifies processor IDs. This eliminates a mathematical computation to change the index into a processor ID. In addition, the potentially all-to-all communication pat-

Listing 6.19: The ZPL Cut Version of IS Rank

```
1  bucketSize[] := 0;
2  [K] bucketSize[bsr(KeyArray,shift)] += 1;
3  bucketSizeTotals[] := +<< bucketSize[];
4  bucketSumAccumulator := 0;
5  bucketProc[] := 0;
6  proc := 0;
7  m := 0;
8  for i := 0 to numBuckets - 1 do
9    bucketSumAccumulator += bucketSizeTotals[i];
10   bucketProc[i] := proc;
11   if proc < localeID() then
12     m += bucketSizeTotals[i];
13   end;
14   if bucketSumAccumulator >= (proc + 1) * numKeys then
15     cutsTotalKeys[proc] := bucketSumAccumulator - 1;
16     cutsMaxKey[proc] := bsl(i+1,shift) - 1;
17     proc += 1;
18   end;
19 end;
20 DBuff1 <== [cut(cutsMaxKey)];
21 DBuff2 <== [cut(cutsTotalKeys)];
22 j := blockLocalLow(KeyBuff2, 1) - 1;
23 [K] KeyBuff2[inc(j)]#[::bucketProc[bsr(KeyArray,shift)]] := KeyArray;
24 [V] KeyBuff1 := 0;
25 [K] KeyBuff1::[KeyBuff2] += 1;
26 [V] interleave
27   KeyBuff1 += m;
28   m := KeyBuff1;
29 end;
```

tern stemming from an exchange of indexes is avoided as is the costly run length encoding process.

6.3.4 A ZPL Version Using Cut Distributions

The third ZPL version of IS, called ZPL Cut, achieves an even higher level of abstraction by taking advantage of the `cut` distribution to avoid the indexed array buffers necessary to achieve a load balanced solution with `blk` distributed arrays. This version also avoids the extra buffer in the full verify procedure. Despite its tighter implementation in memory, memory management is an issue and it will be discussed further in the performance discussion. Listing 6.19 shows the core of the rank procedure for ZPL Cut and Listing 6.20 shows the full verify procedure for ZPL Cut.

Listing 6.20: The ZPL Cut Version of IS Full Verify

```
1 [K] KeyArray::[dec(KeyBuff1::[KeyBuff2])] := KeyBuff2;
2 j := 0;
3 [1..totalKeys-1] if KeyArray@[-1] > KeyArray then
4   j += 1;
5 end;
6 if j = 0 then
7   verified += 1;
8 end;
```

The ZPL Cut version of IS uses two regions. The region `R` is renamed `K` to denote that it is over the keys. The second region `V` is over the possible values that the keys can take on. These regions are declared as follows:

```
region K = [0..totalKeys-1];
       V = [0..maxKey-1];
```

In the other ZPL versions, `KeyBuff2` was used to contain keys and `KeyBuff1` was used to rank the keys and thus expanded the values of the keys. This distinction is made clearer in ZPL Cut using the two regions.

The arrays are declared over their own distributions and regions which are changed when necessary. For example, the names of the distributions of `KeyBuff1` and `KeyBuff2` are `DBuff1` and `DBuff2`, and these distributions are changed in lines 20 and 21 The declarations of ZPL Cut appear in the fourth column of Tables 6.4 and 6.5.

The ZPL Cut version is similar to the other versions of IS already discussed, especially in the beginning. Lines 1–3 of ZPL Cut are identical to lines 1–3 of ZPL Pin and ZPL Free. There are some differences in the computation of global values. Though `m`, the number of keys on smaller processors, and `bucketProc`, the map, are the same as before, the other values are calculated for every processor on every processor. Instead of the two values `MinKeyVal` and `MaxKeyVal` on each processor, the indexed array `cutsMaxKey` contains `MaxKeyVal` for all processors. This and the array `cutsTotalKeys` are used for the unbalanced distributions of keys after the keys are remapped.

Lines 20–21 are unique to this version. They update the distribution of the buffers to account for the unbalanced distribution of keys. The remap is similar to the remap of ZPL

Free. Again, the map does not contain indexes into a region but rather processors to which the data should map, thus avoiding the expensive exchange and encoding of indexes. Note in line 22 that `j` is set to local low index on each processor. The reason for this is that indexing into the parallel array is done using the global indexes. So each processor has a different low index.

The ZPL Cut version of the full verify procedure shows the advantages to the higher level version. The values are sorted in the first line which is equivalent to lines 1–4 of ZPL Free. Note that the values which were subtracted in the previous versions are no longer necessary since global indexing is already assumed.

The big advantage of the full verify procedure shows up in Lines 2–5 in which the array is tested to make sure it is sorted. This is equivalent to lines 5–17 of ZPL Free, lines 6–18 of ZPL Pin, and lines 4–17 of the C+MPI. The test code is p-independent and written in such a way that ignores how the array is broken up between the processors. The IS benchmark generally involves significantly more processor-oriented programming than the other NAS benchmarks. This even shows up in ZPL Cut. In a longer application, one in which a sort plays a small role, ZPL's high-level abstractions would really pay off.

The Possibility of A P-Independent ZPL Version of IS

The versions of IS discussed in this section all require a significant amount of p-dependence. Unlike EP, for which a p-independent ZPL code was presented, and FT, for which two largely p-independent codes were presented, the versions of IS use significantly more p-dependent abstractions. Although they are p-independent programs, there is a great deal of room left to make p-dependent errors.

Another ZPL version using different `cut` distributions is possible. It is an even higher-level version of IS that is wholly p-independent. Certain implementation limitations prevent its completion, but it is outlined here.

Rather than the region `V` in ZPL Cut, this new version would use the region `K` and `B` as follows:

```
region K = [0..totalKeys-1];
       B = [0..numBuckets-1];
```

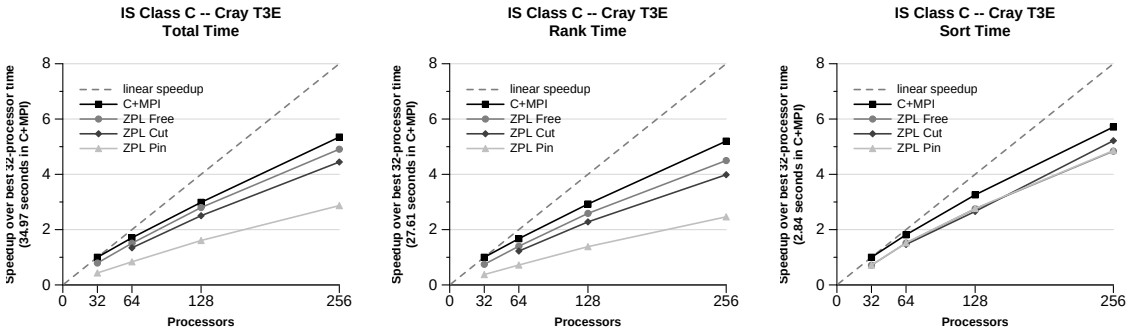

Figure 6.4: Parallel Speedup of NAS IS

The buffers can then be allocated over `B`. Note that the big buffer size in the C+MPI, ZPL Pin, and ZPL Free versions could not be used here. The buckets would be of very different sizes and need to use indexed arrays that can be resized. (ZPL currently does not support variable-size indexed arrays.)

The advantages of this version are numerous. For one, the remap would be significantly simpler and p-independent. Rather than using an array like `bucketProc` to implement the remapping, the version would use the values from the expression `bsr(KeyArray,shift)` to determine which bucket a key should be remapped to. The buckets are distributed across the processors using a `cut` distribution.

The disadvantage of this version is that it is likely that the remap operator would be as expensive, if not more expensive, than the remap of ZPL Pin. Performance results will be discussed shortly, but ZPL Pin should be treated as an upper bound of performance of this p-independent version. This is because the indexes would have to be sent between the processors as well as the data so that the receiving processors would be able to put the data into the right buckets.

6.3.5 Performance

Figure 6.3.5 shows speedup graphs comparing the performance of the C+MPI, ZPL Pin, ZPL Free, and ZPL Cut versions for class C of the NAS IS benchmark on a Cray T3E. (The raw data and information on the machine and the compilers is listed in Appendix B.) The

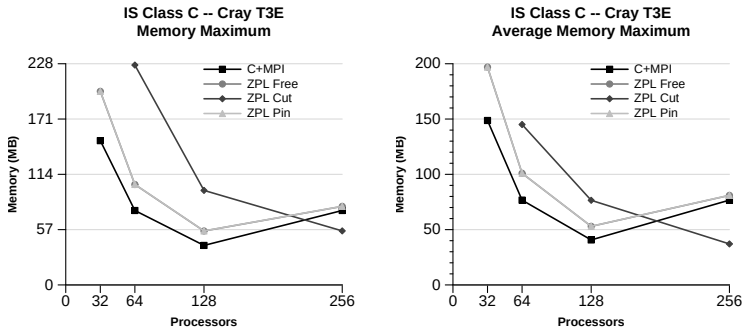

Figure 6.5: Memory Usage of NAS IS

speedup graphs show total time as well as the time used for the rank procedure (rank time) and the full verify procedure (sort time). The rank procedure accounts for the bulk of the total time; the sort time is relatively small.

The ZPL Free and ZPL Cut versions of IS are competitive with the C+MPI version. The ZPL Pin version is significantly slower because of the inefficiencies of the remap operator application discussed before. The ZPL Cut version suffers from poor memory management and is unable to run on 32 processors. It actually uses less memory than the other versions (using the minimum possible), but limitations on the ZPL memory management implementation show up in this code.

Figure 6.3.5 shows the amount of memory used by each of the versions of IS. The memory graphs show the maximum memory used by any of the processors and the average maximum memory used by all of the processors. The ZPL Pin and ZPL Free versions are identical in memory usage. The C+MPI version uses noticeably less memory on less than 256 processors and roughly the same amount of memory on 256 (and more) processors. These three versions allocate buffers that are significantly larger than necessary. Their size is cased on whether there are less than 256 processors. If there are less than 256 processors, the buffers are allocated to contain 150% of the total number of keys divided by the number of processors. Otherwise, they contain 600% of the total number of keys divided by the number of processors.

The C+MPI version uses three large buffers for `key_array`, `key_buff1`, and `key_buff2`.

The ZPL Free and ZPL Pin versions use exactly the necessary storage for `KeyArray`, but still declare three large buffers for `KeyBuff1`, `KeyBuff2`, and `KeyBuff3`. On 256 processors and more, the three buffers dominate the amount of storage and the three implementations use roughly the same amount of memory. On less than 256 processors, the extra storage required for `KeyArray` is significant for ZPL Free and ZPL Pin. In addition, there is a modest amount of storage used for implementing the remap movement. The C+MPI program uses the storage in `key_buff1` for this movement, but the ZPL versions do not.

The memory used by ZPL Cut is more interesting. The graphs show the total memory that the program has requested from the system. The current ZPL memory management solution is less than ideal. The main problem is that the buffers, which are declared to be only as big as necessary, are destroyed using `free` and recreated using `malloc` sometime later in the program. On some processors, the array size is increased. In these cases, the area of memory may not be big enough and a new area of memory may need to be requested. This is the reason that IS cannot even run on 32 processors. It is an implementation issue that is not inherent in ZPL. A smarter memory manager could be used for the array data that may entail moving array data if memory is grossly fragmented. It also accounts for some of the great difference in memory requirements between the maximum for any processor and the average maximum for all processors. The rest of the difference is accounted for by the slightly unbalanced data distribution. Note that this difference does not show up with the other three versions.

6.4 Summary

This chapter presented ZPL versions for NAS EP, FT, and IS. These kernel benchmarks were chosen because they illustrate the extensions introduced in this thesis. Through a qualitative discussion, the ZPL versions were shown to be written at a higher level of abstraction. Quantitative performance results were shown for each benchmark demonstrating that the ZPL code is highly competitive.

Chapter 7

CONCLUSIONS

The difficulty of writing parallel codes has proven an insurmountable obstacle for the majority of scientists wanting to use supercomputers or high-performance clusters. The most widely available tool for parallel programming is the unequivocally processor-oriented MPI library. It needlessly complicates the vast majority of scientific programming tasks, burdening them with issues of synchronization, race conditions, and deadlocks. Although it is possible to eke out high performance on the parallel machines using MPI, generally the required time and effort is too great.

7.1 Summary

This thesis has extended ZPL, a high-level parallel programming language. It has added abstractions for managing processor layout and data distribution, for writing processor-oriented code, and for defining new reduction and scan operators. The performance model has been maintained, though several new exceptions were detailed. In addition, the distinction between p-independent and p-dependent abstractions has been introduced. The clean separation between these types of abstractions is the source of great potential in ZPL.

7.2 Future Work

7.2.1 User-Defined Distributions

Code for defining distributions, as shown in Appendix A for the `cut` distribution, is very difficult to get right. It is important, therefore, to limit the amount of code that needs to be written in order to define a new distribution. If it can be kept as small as this preliminary implementation suggests, it could be useful to let programmers define their own distributions. This could be especially useful for defining block distributions that mesh

with block distributions for non-ZPL software. There are simply too many variations on block distributions to support them all. When the number of processors does not divide the index space evenly, the excess indices can be mapped to the processors in numerous ways. For example, they could all map to the last processor or they could be divided as evenly as possible among all the processors.

As important as it is to limit the amount of code that needs to be written, it is equally important to allow for a significant amount of optional code to be written. Such code could be used to further optimize the code. For example, this idea is discussed in Appendix A.

7.2.2 Automatically Testing User-Defined Distributions, Reductions, and Scans

User-defined distributions could be defined similarly to user-defined reductions and scans. They also have the potential to be used to create incorrect, difficult-to-debug, p-dependent code.

The difficulty of implementing correct distributions has already been noted. Although reductions and scans seem to be easier to get right, they are difficult to correct, if they are wrong, because they are p-dependent. A tool to prove they are correct or, at least, p-independent does not seem easy to build. However, a tool to test the procedures with some random inputs to increase the programmer's confidence in their correctness seems like a good idea.

Such a tool would work similarly for user-defined distributions, reductions, and scans. It would involve testing the procedures to make sure the same answer is computed independently of the number of processors. In the case of the reduction and scan, the accumulator and combiner need to be called in different sequences to check their correctness. In the case of the distribution, the to-processor procedure needs to be checked against the setup and init procedures.

7.2.3 Anonymous Grids and Distributions in Declarations

One problem with grids and distributions in ZPL is that there are too many structures that must be named even if they never have to be referred to again. In the ZPL Grid version of

the FT benchmark, this problem can be seen. Recall the declarations:

```
var
  GU0 :        grid<..,..,..>;
  DU0 : [GU0]  distribution<block,block,block> = D;
  RU0 : [DU0]  region<..,..,..> = R;
  U0  : [RU0]  dcomplex;
  GU1 :        grid<..,..,..>;
  DU1 : [GU1]  distribution<block,block,block> = D;
  RU1 : [DU1]  region<..,..,..> = R;
  U1  : [RU1]  dcomplex;
```

The variable regions and distributions above are never referenced. The distributions of the arrays are modified only when the grids are changed. As an extension, it may be possible to use anonymous or distributionless regions and anonymous or gridless distributions in the declarations. This would make it possible to not have to name the distributions and regions. For example, the declarations could be rewritten as follows:

```
var
  GU0 :        grid<..,..,..>;
  U0  : [GU0]  [D] [R] dcomplex;
  GU1 :        grid<..,..,..>;
  U1  : [GU1]  [D] [R] dcomplex;
```

In this code, only the variables that are changed are named. The arrays have their own regions and distributions, but these cannot be accessed by name. It may be worthwhile to allow them to be accessed through intrinsic functions that, for example, return a distribution given a region. In this case, it could be nice to avoid declaring the grids since each grid only has one array. They could be declared anonymously, but there is currently no way of having the counterpart of a distributionless region or a gridless distribution for grids.

7.2.4 Extending Grid Dimensions

In the discussion of the extension to the remap operator for grid dimensions, the remap maps were allowed to be grid dimensions, `::`, followed by a processor number. There is no reason to limit this number to the remap case. It should always be possible to follow a grid dimension by a number. For example, the region scope `[::2]` would refer to only processor two rather than each processor.

7.2.5 ZPL Extensions to C and Fortran

ZPL is syntactically based on Modula-2. Given that most scientific programming today is done in C and Fortran, it makes sense to build ZPL on top of either of these languages. There is nothing particularly special about Modula-2 that makes ZPL possible.

7.2.6 Unification of Indexed Arrays and Parallel Arrays

ZPL is complicated by its two forms of arrays: indexed and parallel arrays. Unifying these two arrays is the grand challenge for ZPL. Such a unification would greatly simplify the language—programmers are often confused by the existence of two kinds of arrays.

This unification is difficult because of ZPL's performance model and its disallowing of nested parallelism. Such a unification would allow for nested parallelism, but might require loosening some of ZPL's restrictions.

7.2.7 Fast Code Development

ZPL's performance model and syntactically identifiable communication make it possible for the programmer to develop highly optimized code. In some cases, however, such strictness is a liability. For sections of code where performance is not important and the communication pattern is complex, *e.g.*, initialization, ZPL programmers have to deal with optimizations that they may rather ignore.

One idea for working around this problem is to let the ZPL programmer open a distribution or region scope that would allow for arbitrary communication that is not syntactically identifiable. This idea deserves more exploration as it could enable even faster code development.

7.2.8 Conservative P-Independent Analysis

In the execution of a parallel program, values can be thought of as being p-dependent. P-dependent values can be seen to flow through the variables, and thus a tool to detect where code is p-independent could be based on a data-flow analysis. However, while a simple data-flow analysis can be used to locate p-dependent values, it alone may be too conservative.

Recall the manual contraction example of Section 4.4.1. The free variable `temp` is assigned a p-dependent value, but this p-dependent value does not flow to `A` or `B`. If the code used `begin` instead of `interleave`, the p-dependent values would flow to `A` and `B`. A tool that detected this presumed mistake could be useful in keeping bugs out of codes.

A good analysis must be based on a p-dependent data-flow analysis, but must also consider other properties of the language. Our hypothesis is that even a very conservative analysis could help locate a small number of areas where p-dependent values could emerge. By limiting the section of code that has the potential for parallel bugs, the task of parallel programming could be greatly eased.

Recall the discussion of relative debugging at the beginning of this thesis. The focus of this thesis has been similar, but from the opposite side. For this reason, it is sometimes nice to refer to p-independent programming as *absolute* programming, making the comparison to relative debugging clear. In an absolute programming language, relative debugging would only be beneficial for porting codes. However, for any parallel language with p-dependent abstractions, relative debugging could be used to find parallel bugs. Coupling a good relative debugging tool with a good p-dependence detection tool may lead to a system that could help a programmer find or fix a bug.

Note that the analyses of relative debuggers are, in a sense, the reverse of the analyses for p-dependence detection tools. In relative debuggers, the location of where a p-dependent value was introduced is found by backtracing through a parallel execution rather than forward tracing through the program statically.

Language abstractions, compiler analyses, and programming tools that foster the development of p-independent parallel programs have the potential to radically improve the current state of the art. By guaranteeing to the programmer that a code working on one processor will work on any number of processors or, at least, by limiting where in a program a parallel bug could occur, development becomes substantially easier. This thesis has argued that ZPL is a mostly p-independent parallel programming language and has introduced several p-dependent abstractions that provide powerful flexibility.

BIBLIOGRAPHY

[AGL+98] Gail Alverson, William Griswold, Calvin Lin, David Notkin, and Lawrence Snyder. Abstractions for portable, scalable parallel programming. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):71–86, January 1998.

[AGNS90] Gail Alverson, William Griswold, David Notkin, and Lawrence Snyder. A flexible communication abstraction for nonshared memory parallel computing. In *Proceedings of the ACM Conference on Supercomputing*, 1990.

[AS91] Richard J. Anderson and Lawrence Snyder. A comparison of shared and nonshared memory models of parallel computation. *Proceedings of the IEEE*, 79(4):480–487, April 1991.

[AS96] David Abramson and R. Sosic. A debugging and testing tool for supporting software evolution. *Automated Software Engineering*, 3(3–4):369–390, August 1996.

[BHS+95] David Bailey, Tim Harris, William Saphir, Rob F. Van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report RNR-95-020, NASA Ames Research Center, Moffett Field, CA, December 1995.

[BK94] Ray Barriuso and Allan Knies. SHMEM user's guide. Technical Report SN-2516, Cray Research Inc., May 1994.

[Ble95] Guy E. Blelloch. NESL: A nested data-parallel language (Version 3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, Pittsburgh, PA, September 1995.

[Bou96] Luc Bouge. The data parallel programming model: A semantic perspective. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*. Springer-Verlag, 1996.

[CCDS04] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[CCL+96] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. Factor-join: A unique approach to compiling array languages for parallel machines. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1996.

[CCL+98a] Bradford L. 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 *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 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.

[CCS97] Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. A compiler abstraction for machine independent parallel communication generation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1997.

[CD02] Sung-Eun Choi and Steven J. Deitz. Compiler support for automatic checkpointing. In *Proceedings of the International Symposium on High Performance Computing Systems and Applications*, 2002.

[CDC+99] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, Center for Computing Sciences, Bowie, MD, May 1999.

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

[Cha01] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.

[Cho99] Sung-Eun Choi. *Machine Independent Communication Optimization*. PhD thesis, University of Washington, March 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, Seattle, WA, November 1998.

[CLS99a] Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. Array language support for wavefront and pipelined computations. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1999.

[CLS99b] Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. Problem space promotion and its evaluation as a technique for efficient parallel computation. In *Proceedings of the ACM International Conference on Supercomputing*, 1999.

[CS97] Sung-Eun Choi and Lawrence Snyder. Quantifying the effects of communication optimizations. In *Proceedings of the IEEE International Conference on Parallel Processing*, 1997.

[CS01] Bradford L. Chamberlain and Lawrence Snyder. Array language support for parallel sparse computation. In *Proceedings of the ACM International Conference on Supercomputing*, 2001.

[DCCS03] Steven J. Deitz, Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. The design and implementation of a parallel array operator for the arbitrary remapping of data. In *Proceedings of the ACM Conference on Principles and Practice of Parallel Programming*, 2003.

[DCS01] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the ACM International Conference on Supercomputing*, 2001.

[DCS02] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. High-level language support for user-defined reductions. *Journal of Supercomputing*, 23(1), 2002.

[DCS04] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. Abstractions for dynamic data distribution. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2004.

[Dei03] Steven J. Deitz. Renewed hope for data parallelism: Unintegrated support for task parallelism in ZPL. Technical Report UW-CSE-2003-12-04, University of Washington, Seattle, WA, December 2003.

[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 *Proceedings of the ACM International Conference on Supercomputing*, 1995.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: an industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January–March 1998.

[GHNS90] William Griswold, Gail Harrison, David Notkin, and Lawrence Snyder. Scalable abstractions for parallel programming. In *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990.

[GMI03] Haoqiang Jin Stephen Johnson Gregory Matthews, Robert Hood and Constantinos Ierotheou. Automatic relative debugging of openmp programs. In *Proceedings on the European Workshop on OpenMP*, 2003.

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

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Lew01] E Christopher Lewis. *Achieving Robust Performance in Parallel Programming Languages.* PhD thesis, University of Washington, February 2001.

[Lin92] Calvin Lin. *The Portability of Parallel Programs Across MIMD Computers.* PhD thesis, University of Washington, 1992.

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

[LLST95] E Christopher Lewis, Calvin Lin, Lawrence Snyder, and George Turkiyyah. A portable parallel n-body solver. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, 1995.

[LS90] Calvin Lin and Lawrence Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the IEEE International Conference on Parallel Processing*, 1990.

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

[LS93] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1993.

[LS94a] Calvin Lin and Lawrence Snyder. Accommodating polymorphic data decompositions in explicitly parallel programs. In *Proceedings of the International Parallel Processing Symposium*, 1994.

[LS94b] Calvin Lin and Lawrence Snyder. SIMPLE performance results in ZPL. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1994.

[LS00] E Christopher Lewis and Lawrence Snyder. Pipelining wavefront computations: Experiences and performance. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, May 2000.

[LSA+95] Calvin Lin, Lawrence Snyder, Ruth Anderson, Bradford L. Chamberlain, Sung-Eun Choi, George Forman, E Christopher Lewis, and W. Derrick Weathersby. ZPL vs. HPF: A comparison of performance and programming style. Technical Report UW-CSE-95-11-05, University of Washington, Seattle, WA, November 1995.

[LW93] Calvin Lin and W. Derrick Weathersby. Towards a machine-independent solution of sparse cholesky factorization. In *Proceedings of the Third International Conference on Parallel Computing*, 1993.

[Mes94] Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4):169–416, 1994.

[MSA+85] J. R. McGraw, S. K. Skedzielawski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2. Technical Report Manual M-146, Lawrence Livermore National Laboratory, March 1985.

[NC99] Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Workshop on Runtime Systems for Parallel Programming*, 1999.

[Ngo97] Ton Anh Ngo. *The Role of Performance Models in Parallel Programming and Languages*. PhD thesis, University of Washington, 1997.

[Nik91] R. S. Nikhil. Id language reference manual (version 90.1). Technical Report 284-2, Massachusetts Institute of Technology, July 1991.

[NR98] Robert W. Numrich and John K. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, August 1998.

[NS92] Ton Ahn Ngo and Lawrence Snyder. On the influence of programming models on shared memory computer performance. In *Proceedings of the Scalable High Performance Computing Conference*, 1992.

[NSC97] Ton Ahn Ngo, Lawrence Snyder, and Bradford L. Chamberlain. Portable performance of data parallel languages. In *Proceedings of the ACM Conference on Supercomputing*, 1997.

[RBS96] Wilkey Richardson, Mary L. Bailey, and William H. Sanders. Using ZPL to develop a parallel chaos router simulator. In *Proceedings of the Winter Simulation Conference*, 1996.

[Sny86] Lawrence Snyder. Type architecture, shared memory and the corollary of modest potential. *Annual Review of Computer Science*, 1:289–317, 1986.

[Sny94] Lawrence Snyder. Foundations of practical parallel programming languages. In *Portability and Performance for Parallel Processing*, pages 1–19. John Wiley & Sons, Ltd., 1994.

[Sny95] Lawrence Snyder. Experimental validation of models of parallel computation. In *Computer Science Today: Recent Trends and Developments, Lecture Notes in Computer Science*, volume 1000, pages 78–100. Springer Verlag, 1995.

[Sny99] Lawrence Snyder. *Programming Guide to ZPL.* MIT Press, Cambridge, MA, 1999.

[Sny01] Lawrence Snyder. Parallel computation: MM +/- X. In *Informatics: 10 Years Back, 10 Years Ahead, Lecture Notes in Computer Science*, volume 2000, pages 234–250. Springer Verlag, 2001.

[VL96] Guhan Viswanathan and James R. Larus. User-defined reductions for efficient communication in data-parallel languages. Technical Report 1293, University of Wisconsin, Madison, WI, January 1996.

[WA00] Greg Watson and David Abramson. The architecture of a parallel relative debugger. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, 2000.

[Wea99] W. Derrick Weathersby. *Machine-Independent Compiler Optimizations for Collective Communication.* PhD thesis, University of Washington, August 1999.

[Wir83] Nicholas Wirth. *Programming in Modula-2.* Springer-Verlag, New York, NY, 1983.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *ACM Workshop on Java for High-Performance Network Computing*, 1998.

Appendix A

CUT DISTRIBUTION

This appendix contains the C code necessary to add the `cut` distribution to the ZPL runtime. It is the implementation used for the timings of Chapter 6.

There are four procedures that must be defined to implement a block-type distribution: setup, init, copy, and to-processor. The setup procedure for the `cut` distribution, called `_cutSetup`, and the init procedure for the `cut` distribution, called `_cutInit`, set up and initialize the distribution dimension from indexed array `a`. The setup procedure is called when the distribution is declared or assigned in the ZPL code. The init procedure is only called after the grid is initialized. The setup procedure merely copies the values from `a` into an array `cuts`. The init procedure calculates the local low and high bounds of indices for each processor.

```
void _cutSetup(_distribution_fnc dist, int dim,
               const int * const restrict a) {
  int i;
  _grid grid = _DIST_GRID(dist);
  const int numCuts = _GRID_SIZE(grid, dim) - 1;

  _DIST_LO(dist, dim) = INT_MIN>>1;
  _DIST_HI(dist, dim) = INT_MAX>>1;
  ((_cut)_DIST_DIMDIST(dist, dim))->numCuts = numCuts;
  if (numCuts) {
    ((_cut)_DIST_DIMDIST(dist, dim))->cuts =
      malloc(sizeof(int) * numCuts);
    for (i = 0; i < numCuts; i++) {
      ((_cut)_DIST_DIMDIST(dist, dim))->cuts[i] = a[i];
    }
  }
  _DIMDIST_TO_PROC(*_DIST_DIMDIST(dist, dim)) = _cutToProc;
  _DIMDIST_INIT(*_DIST_DIMDIST(dist, dim)) = _cutInit;
  _DIMDIST_COPY(*_DIST_DIMDIST(dist, dim)) = _cutCopy;
}


void _cutInit(_distribution_fnc dist, int dim) {
```

```
  _grid grid = _DIST_GRID(dist);
  const int * const restrict cuts =
    ((_cut)_DIST_DIMDIST(dist, dim))->cuts;

  if (_GRID_SIZE(grid, dim) == 1) {
    _DIST_MYLO(dist, dim) = INT_MIN>>1;
    _DIST_MYHI(dist, dim) = INT_MAX>>1;
  }
  else if (_GRID_LOC(grid, dim) == 0) {
    _DIST_MYLO(dist, dim) = INT_MIN>>1;
    _DIST_MYHI(dist, dim) = cuts[_GRID_LOC(grid, dim)];
  }
  else if (_GRID_LOC(grid, dim) == _GRID_SIZE(grid, dim) - 1) {
    _DIST_MYLO(dist, dim) = cuts[_GRID_LOC(grid, dim)-1]+1;
    _DIST_MYHI(dist, dim) = INT_MAX>>1;
  }
  else {
    _DIST_MYLO(dist, dim) = cuts[_GRID_LOC(grid, dim)-1]+1;
    _DIST_MYHI(dist, dim) = cuts[_GRID_LOC(grid, dim)];
  }
}
```

The copy procedure for the `cut` distribution, called `_cutCopy`, copies one distribution dimension to another. It is used when a distribution is initialized with a distribution that already exists and when the preserving or destructive assignment operators are applied to distributions.

```
_dimdist_info* _cutCopy(_dimdist_info* dimdist) {
  int i;
  _cut_info* new = malloc(sizeof(_cut_info));
  new->localBound = dimdist->localBound;
  new->globalBound = dimdist->globalBound;
  new->numCuts = ((_cut)dimdist)->numCuts;
  new->cuts = malloc(sizeof(int) * new->numCuts);
  for (i = 0; i < new->numCuts; i++) {
    new->cuts[i] = ((_cut)dimdist)->cuts[i];
  }
  new->toProc = dimdist->toProc;
  new->init = dimdist->init;
  new->copy = dimdist->copy;
  return (_dimdist_nc)new;
}
```

The to-processor procedure for the `cut` distribution, called `_cutToProc`, returns the processor that a given index belongs too. A more efficient implementation might use a binary search to determine the processor of an index.

```
int _cutToProc(_distribution dist, int dim, int i) {
```

```
  const int numCuts = ((_cut)_DIST_DIMDIST(dist, dim))->numCuts;
  const int * const restrict cuts =
    ((_cut)_DIST_DIMDIST(dist, dim))->cuts;
  int j;

  if (numCuts == 0) {
    return 0;
  }
  for (j = 0; j < numCuts; j++) {
    if (i <= cuts[j]) {
      return j;
    }
  }
  return j;
}
```

There is no more code necessary to add the `cut` distribution. (Code to free memory is omitted.) In a sense, this is the *minimum* amount of code. It is possible that later implementations could add optional procedures that result in a faster implementation. For example, it is possible to write code to determine the processors a range of indices map to that is more efficient than code using multiple calls to the `_cutToProc` procedure.

Appendix B

EXPERIMENTAL TIMINGS

This appendix contains tables showing raw timings (in seconds) and memory reports (in MB) for the experiments of Chapter 6 that compared ZPL and NAS version of the NAS benchmarks EP, FT, and IS.

Table B.1: Raw Experimental Data for NAS EP

EP Class C – Cray T3E (Total Time)

processors	2	4	8	16	32	64	128	256
F+MPI	2167.777	1084.030	541.990	271.000	135.500	67.760	33.880	16.940
ZPL Grid	2212.823	1106.560	553.263	276.633	138.319	69.170	34.586	17.295
ZPL Free	—.—	1160.631	580.303	290.166	145.083	72.551	36.277	18.141

EP Class C – Cray T3E (Random Numbers Time)

processors	2	4	8	16	32	64	128	256
F+MPI	788.635	394.354	197.174	98.586	49.297	24.650	12.325	6.163
ZPL Grid	819.282	409.706	204.851	102.424	51.213	25.609	12.805	6.403
ZPL Free	—.—	409.724	204.840	102.427	51.215	25.609	12.805	6.403

EP Class C – Cray T3E (Gaussian Pairs Time)

processors	2	4	8	16	32	64	128	256
F+MPI	1378.776	689.478	344.727	172.367	86.184	43.101	21.551	10.778
ZPL Grid	1393.198	696.664	348.329	174.167	87.083	43.551	21.778	10.890
ZPL Free	—.—	750.735	375.362	187.689	93.844	46.930	23.466	11.737

EP Class C – Cray T3E (Maximum Memory)

processors	2	4	8	16	32	64	128	256
F+MPI	6.240	6.240	6.240	6.240	6.240	6.240	6.240	6.240
ZPL Grid	5.440	5.440	5.440	5.440	5.440	5.440	5.440	5.440
ZPL Free	—.—	5.440	5.440	5.440	5.440	5.440	5.440	5.440

EP Class C – Cray T3E (Average Maximum Memory)

processors	2	4	8	16	32	64	128	256
F+MPI	6.254	6.254	6.254	6.254	6.254	6.254	6.254	6.254
ZPL Grid	5.410	5.410	5.410	5.410	5.410	5.411	5.411	5.410
ZPL Free	—.—	5.406	5.406	5.406	5.406	5.406	5.407	5.406

Table B.2: Raw Experimental Data for NAS FT

FT Class C – Cray T3E (Total Time)

processors	32	64	128	256
ZPL Grid	369.261	189.023	94.966	48.736
F+MPI	377.890	192.219	97.020	49.340
ZPL	419.589	213.664	108.073	54.506

FT Class C – Cray T3E (Setup Time)

processors	32	64	128	256
ZPL Grid	4.399	2.200	1.100	0.551
F+MPI	2.220	1.110	0.555	0.278
ZPL	4.482	2.243	1.122	0.561

FT Class C – Cray T3E (Evolve Time)

processors	32	64	128	256
ZPL Grid	12.047	6.044	3.052	1.665
F+MPI	15.821	7.911	3.957	1.980
ZPL	11.967	5.984	2.993	1.498

FT Class C – Cray T3E (FFT Low Time)

processors	32	64	128	256
ZPL Grid	231.195	115.758	57.845	28.879
F+MPI	236.145	118.029	59.018	29.511
ZPL	231.121	115.495	57.758	28.893

FT Class C – Cray T3E (FFT Copy Time)

processors	32	64	128	256
ZPL Grid	77.883	38.639	19.022	9.538
F+MPI	80.605	40.758	20.384	10.196
ZPL	86.523	43.598	21.856	10.923

FT Class C – Cray T3E (Transpose Time)

processors	32	64	128	256
ZPL Grid	43.240	25.871	13.443	7.876
F+MPI	42.400	24.539	13.083	7.386
ZPL	85.009	45.906	24.126	12.501

FT Class C – Cray T3E (Maximum Memory)

processors	32	64	128	256
ZPL Grid	229.440	117.440	61.520	33.760
F+MPI	229.760	117.760	61.760	33.760
ZPL	229.360	117.440	61.520	33.680

FT Class C – Cray T3E (Average Maximum Memory)

processors	32	64	128	256
ZPL Grid	229.352	117.410	61.535	33.660
F+MPI	229.750	117.750	61.750	33.750
ZPL	229.375	117.371	61.496	33.621

Table B.3: Raw Experimental Data for NAS IS

IS Class C – Cray T3E (Total Time)

processors	32	64	128	256
C+MPI	34.970	20.500	11.710	6.540
ZPL Free	44.183	23.187	12.493	7.124
ZPL Cut	—.—	26.052	13.966	7.866
ZPL Pin	80.795	41.823	21.759	12.192

IS Class C – Cray T3E (Rank Time)

processors	32	64	128	256
C+MPI	27.613	16.503	9.439	5.310
ZPL Free	37.054	19.736	10.674	6.139
ZPL Cut	—.—	22.539	12.110	6.928
ZPL Pin	73.684	38.380	19.944	11.213

IS Class C – Cray T3E (Sort Time)

processors	32	64	128	256
C+MPI	2.837	1.565	0.871	0.497
ZPL Free	3.978	1.874	1.031	0.587
ZPL Cut	—.—	1.938	1.067	0.544
ZPL Pin	3.961	1.867	1.027	0.584

IS Class C – Cray T3E (Maximum Memory)

processors	32	64	128	256
C+MPI	148.720	76.720	40.720	76.720
ZPL Free	199.520	103.520	55.600	80.960
ZPL Cut	—.—	226.640	97.440	55.680
ZPL Pin	199.600	103.600	55.680	81.040

IS Class C – Cray T3E (Average Maximum Memory)

processors	32	64	128	256
C+MPI	148.688	76.684	40.684	76.688
ZPL Free	196.953	100.953	52.971	80.977
ZPL Cut	—.—	145.047	76.516	37.068
ZPL Pin	196.977	100.988	53.014	81.055

VITA

Steve Deitz was born in New York City on April 13, 1976. He grew up in Hastings-On-Hudson, New York, graduating from Hastings High School in the spring of 1994. He attended Bowdoin College from 1994 to 1998, earning his B.A. in Computer Science and Mathematics. After graduating college, he immediately began graduate studies in the Department of Computer Science and Engineering at the University of Washington, earning his M.S. in 2000 and his Ph.D. in 2005.