The Design and Implementation of a Region-Based Parallel Programming Language

Bradford L. Chamberlain

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

Doctor of Philosophy

University of Washington

2001

Program Authorized to Offer Degree: Computer Science & Engineering

University of Washington

Graduate School

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

Bradford L. Chamberlain

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

Chair of Supervisory Committee:

Lawrence Snyder

Reading Committee:

Craig Chambers

Burton Smith

Lawrence Snyder

Date:

University of Washington

Abstract

The Design and Implementation of a Region-Based Parallel Programming Language

by Bradford L. Chamberlain

Chair of Supervisory Committee:

Professor Lawrence Snyder
Computer Science & Engineering

Programming parallel computers is an extremely challenging task for expert computer programmers, let alone for scientists in other disciplines whose computations often drive the acquisition of such machines. This dissertation describes the design and implementation of ZPL, a programming language created to simplify the task of programming parallel computers. ZPL allows programmers to write code using a global view that describes their algorithms at a high level rather than implementing per-processor behavior. However, unlike other global-view languages, ZPL permits users to reason about the parallel implementation of their code at the syntactic level, allowing them to make informed algorithmic decisions based on the program's parallel implementation. The language feature that supports this duality is called the region. A region is simply a language-level index set that programmers can define, name, and manipulate using high-level operators. Regions constitute a unique means of specifying array computation, serving as an alternative to traditional array indexing and slicing. By distributing each region's indices across a processor set, a parallel interpretation of a ZPL program is achieved.

This dissertation studies the impact of the region concept throughout the design and im-

plementation of ZPL. It begins by defining the region concept and its use in the language. It then gives a parallel interpretation of regions, which results in ZPL's syntax-based performance model. ZPL's implementation and runtime libraries are described in detail to show how regions are represented and used at runtime. The design and implementation of a paradigm-neutral interface for efficient portable communication is also described. Finally, two extensions to the basic region concept are given: parameterized regions which can be used to implement hierarchical algorithms such as the multigrid method, and sparse regions which can be used to specify sparse computation over sparse or dense arrays. Throughout the dissertation, regions are evaluated by comparing ZPL programs to other languages in terms of clarity and performance. The conclusion is that regions are a crisp and powerful mechanism for array-based parallel programming.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACKNOWLEDGMENTS

I would like to thank everyone who made the completion of this dissertation possible and enjoyable, in spite of the challenges.

First and foremost to my immediate family, who have been a constant support network as well as my best friends throughout my graduate school years. Mom, Dad, Jeff, and Kathleen: thanks for all the encouragement, support, friendship, and love.

Thanks also to my grandparents, uncles, aunts, and cousins who have managed to keep up with me a bit throughout the process, especially Granny, Gramps, Big John, Ed, Lee, Phil, and Harriet. Thanks also to Helen, Rob, and Connie who didn't get to see me cross the finish line, but were an important part of the process.

It's been interesting re-enacting the transformation from childhood through adolescence to adulthood in my relationship with my advisor, Larry Snyder. But like my actual childhood, I could not have imagined a better "parent" to guide me through those changes. Larry, thanks for your constant advice, wisdom, and encouragement throughout the years. I could not have wished for an advisor more suited to my style and temperment.

Burton Smith and Craig Chambers, the other members of my supervisory committee, have also provided valuable feedback and criticism throughout the years, and have my deepest gratitude. Thanks also to Mike Eisenberg for stepping in as my GSR committee member at the last minute, and to Mark Damborg for serving faithfully in the years preceding. I would also like to thank Calvin Lin for serving as an advisor during my early years, and from a distance thereafter.

I would like to recognize and thank the faculty of UW CSE, who have made this department enjoyable and comfortable throughout the years. Most particularly: Paul Beame, Alan Borning, Gaetano Borriello, Tony DeRose, Martin Dickey, Carl Ebeling, Anna Karlin, Richard Ladner, and Larry Ruzzo. Thanks also to the other faculty members at UW whose collaborations have been useful and educational in the course of this work, particularly Randy LeVeque, Tom Quinn, and George Turkiyyah. And a tip of the hat to John Lewis, for providing an external and informed opinion on several occasions throughout the years.

Thanks to all of the members of the ZPL project that I've had the opportunity to work with closely over the years: Ruth Anderson, Sung-Eun Choi, Steve Deitz, George Forman, E Lewis, Ton Ngo, and Derrick Weathersby.

I would like to thank my roommate Ketan Dalal for subjecting himself to an early draft of this document, and to everyone who was coerced into reading earlier incarnations of the work in preparing it for publication.

The work in this dissertation would not have been possible without grants of supercomputing time from the Arctic Region Supercomputer Center, Los Alamos National Laboratory, the Maui High Performance Computing Center, and the University of Texas at Austin. Thanks especially to Tom Baring and Guy Robinson at ARSC for providing excellent service, feedback, and e-friendship throughout the years.

Perhaps most importantly, to the great friends that I've had the opportunity to know and laugh with during my years at the University of Washington. Without you, it would not have been nearly as fun or rewarding: Ruth Anderson, Paul, Beth, and Hannah Barton/Davis, A. J. Bernheim Brush, Brett Bruyere, Sung-Eun Choi, Tina Chung, Mike Cloyd, Susannah Andre Cloyd, Kathy Cowles, Ketan Dalal and Allison Klein, Brian and Amy Dragoo, Pam Emerson, Jim Fix, Howard Goldstein, Suzzin Gygi, Kevin Hinshaw, Becca Jones Johnson, Samantha Johnston, Jean Kaiser, Eric Kratsa, Tashana Landray, Sonny Lemmons, Neal Lesh, Lee Lim, Nick Manning, Jen Marlowe, Carrie McCarthy and Kent Chasson, Ellen Pan and Muriel Wood, Caroline Park, Sasha Peters, Denise Pinnel and Mike Salisbury, Joanna and Eva Power, Pete and Beverly Richardson, John Roberts, Amy Robison, Julia Sanders, Sean Sandys, Geoff and Jenny Skillman, Chris Svara, Geoff Voelker, Alec and Yvonne Wolman, and last (but far from least) Wayne Wong.

Though frivolous when compared to family, friends, and advisors, I tend to believe that this work would not have been possible without the accompanying music of the following artists: The Afghan Whigs, American Music Club, Tori Amos, Laurie Anderson, the Arsonists, Belle and Sebastian, Belly, Big Star, Frank Black, Black Anger, Black-Eyed Peas, Black Tape for a Blue Girl, Luka Bloom, Bovox Clown, Billy Bragg, the Breeders, Michael Brook, Richard Buckner, Buffalo Tom, Built to Spill, Cadallaca, Johnny Cash, Nick Cave and the Bad Seeds, Leonard Cohen, Common, Dead Can Dance, Diamond Fist Werney, Ani DiFranco, Dirty Three, Mark Eitzel, Enya, Foo Fighters, Fugazi, Peter Gabriel, Ganger, Lisa Germano, Lisa Gerrard, Henryk Gorecki, PJ Harvey, Hated, Kristin Hersh, His Name is Alive, Husker Du, Ida, Jawbreaker, Juno, Mark Lanegan, Led Zeppelin, the Leland Stanford Junior (pause) University Marching Band, Liquorice, Live, Low, Nusrat Fateh Ali Khan, Matthew Good Band, Dave Matthews Band, Mavis Piggot, the Modern Lovers, Modest Mouse, Mogwai, the Moon Seven Times, Morphine, Bob Mould, Neutral Milk Hotel, Nine Inch Nails, Nirvana, OutKast, Palace, Parini, Avro Part, Pearl Jam, a Perfect Circle, Joel R. L. Phelps and the Downer Trio, the Pixies, the Pogues, Mary Prankster, Public Enemy, Rage Against the Machine, Red House Painters, R.E.M., the Roots, Sage, Screaming Trees, Shudder to Think, Sky Cries Mary, Sleater-Kinney, Smashing Pumpkins, Elliot Smith, Son Volt, Sonic Youth, Soundgarden, Spoon, Bruce Springsteen, Sugar, Sunny Day Real Estate, Swallow, Swervedriver, Talisman, Tarnation, Team Dresch, Three Fish, Three Shades of Dirty, Throwing Muses, Steve Tibbets, Tool, Tuatara, U2, Unrest, Tom Waits, Whiskeytown, Wilco, Dar Williams, Lucinda Williams, the Wedding Present, Wu-Tang Clan, Yo La Tengo, and Hukwe Zawose.

DEDICATION

This dissertation is dedicated to my grandmother, Connie Huneke, who did not get to see its conclusion, yet whose curiosity and passion for seemingly everything in life left its indeliable mark on me.

Chapter 1

INTRODUCTION TO PARALLEL PROGRAMMING

The past few decades have seen large fluctuations in the perceived value of parallel computing. At times, parallel computation has optimistically been viewed as the solution to all of our computational limitations. At other times, many have argued that it is a waste of effort given the rate at which processor speeds and memory prices continue to improve. Perceptions continue to vacillate between these two extremes due to a number of factors, among them: the continual changes in the "hot" problems being solved, the programming environments available to users, the supercomputing market, the vendors involved in building these supercomputers, and the academic community's focus at any given point and time. The result is a somewhat muddied picture from which it is difficult to objectively judge the value and promise of parallel computing.

In spite of the rapid advances in sequential computing technology, the promise of parallel computing is the same now as it was at its inception. Namely, if users can buy fast sequential computers with gigabytes of memory, imagine how much faster their programs could run if p of these machines were working in cooperation! Or, imagine how much larger a problem they could solve if the memories of p of these machines were used cooperatively!

The challenges to realizing this potential can be grouped into two main problems: the hardware problem and the software problem. The former asks, "how do I build a parallel machine that will allow these p processors and memories to cooperate efficiently?" The software problem asks, "given such a platform, how do I express my computation such that it will utilize these p processors and memories effectively?"

In recent years, there has been a growing awareness that while the parallel community can build machines that are reasonably efficient and/or cheap, most programmers and scientists are incapable of programming them effectively. Moreover, even the best parallel programmers cannot do so without significant effort. The implication is that the software problem is currently lacking in satisfactory solutions. This dissertation focuses on one approach designed to solve that problem.

In particular, this work describes an effort to improve a programmer's ability to utilize parallel computers effectively using the ZPL parallel programming language. ZPL is a language whose parallelism stems from operations applied to its arrays' elements. ZPL derives from the description of Orca C in Calvin Lin's dissertation of 1992 [Lin92]. Since that time, Orca C has evolved to the point that it is hardly recognizable, although the foundational ideas have remained intact. ZPL has proven to be successful in that it allows parallel programs to be written at a high level, without sacrificing portability or performance. This dissertation will also describe aspects of Advanced ZPL (A-ZPL), ZPL's successor language which is currently under development.

One of the fundamental concepts that was introduced to Orca C during ZPL's inception was the concept of the *region*. A region is simply a user-specified set of indices, a concept which may seem trivially uninteresting at first glance. However, the use of regions in ZPL has had a pervasive effect on the language's appearance, semantics, compilation, and runtime management, resulting in much of ZPL's success. This dissertation defines the region in greater depth and documents its role in defining and implementing the ZPL language.

This dissertation's study of regions begins in the next chapter. The rest of this chapter provides a general overview of parallel programming, summarizing the challenges inherent in writing parallel programs, the techniques that can be used to create them, and the metrics used to evaluate these techniques. The next section begins by providing a rough overview of parallel architectures.

1.1 Parallel Architectures

1.1.1 Parallel Architecture Classifications

This dissertation categorizes parallel platforms as being one of three rough types: *distributed memory*, *shared memory*, or *shared address space*. This taxonomy is somewhat coarse given the wide variety of parallel architectures that have been developed, but it provides a useful characterization of current architectures for the purposes of this dissertation.

Distributed memory machines are considered to be those in which each processor has a local memory with its own address space. A processor's memory cannot be accessed directly by another processor, requiring both processors to be involved when communicating values from one memory to another. Examples of distributed memory machines include commodity Linux clusters.

Shared memory machines are those in which a single address space and global memory are shared between multiple processors. Each processor owns a local cache, and its values are kept coherent with the global memory by the operating system. Data can be exchanged between processors simply by placing the values, or pointers to values, in a predefined location and synchronizing appropriately. Examples of shared memory machines include the SGI Origin series and the Sun Enterprise.

Shared address space architectures are those in which each processor has its own local memory, but a single shared address space is mapped across the distinct memories. Such architectures allow a processor to access the memories of other processors without their direct involvement, but they differ from shared memory machines in that there is no implicit caching of values located on remote machines. The primary example of a shared address machine is Cray's T3D/T3E line.

Many modern machines are also built using a combination of these technologies in a hierarchical fashion, known as a *cluster*. Most clusters consist of a number of shared memory machines connected by a network, resulting in a hybrid of shared and distributed memory characteristics. IBM's large-scale SP machines are an example of this design.

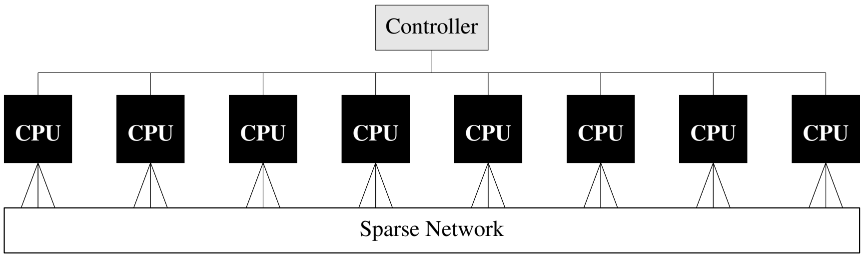


Figure 1.1: The Candidate Type Architecture (CTA)

1.1.2 The CTA Machine Model

ZPL supports compilation and execution on these diverse architectures by describing them using a single machine model known as the *Candidate Type Architecture* (CTA) [Sny86]. The CTA is a reasonably vague model, and deliberately so. It characterizes parallel machines as a group of von Neumann processors, connected by a sparse network of unspecified topology. Each processor has a local memory that it can access at unit cost. Processors can also access other processors' values at a cost significantly higher than unit cost by communicating over the network. The CTA also specifies a controller used for global communications and synchronization, though that will not be of concern in this discussion. See Figure 1.1 for a simple diagram of the CTA.

Why use such an abstract model? The reason is that parallel machines vary so widely in design that it is difficult to develop a more specific model that describes them all. The CTA successfully abstracts the vast majority of parallel machines by emphasizing the importance of locality and the relatively high cost of interprocessor communication. This is in direct contrast to the overly idealized PRAM [FW78] model or the extremely parameterized LogP model [CKP+93], neither of which form a useful foundation for a compiler concerned with portable performance. For more details on the CTA, please refer to the literature [Sny86, Sny95, Lin92].

1.2 Challenges to Parallel Programming

Writing parallel programs is strictly more difficult than writing sequential ones. In sequential programming, the programmer must design an algorithm and then express it to the computer in some manner that is correct, clear, and efficient to execute. Parallel programming involves these same issues, but also adds a number of additional challenges that complicate development and have no counterpart in the sequential realm. These challenges include: finding and expressing concurrency, managing data distributions, managing interprocessor communication, balancing the computational load, and simply implementing the parallel algorithm correctly. This section considers each of these challenges in turn.

1.2.1 Concurrency

Concurrency is crucial if a parallel computer's resources are to be used effectively. If an algorithm cannot be divided into groups of operations that can execute concurrently, performance improvements due to parallelism cannot be achieved, and any processors after the first will be of limited use in accelerating the algorithm. To a large extent, different problems inherently have differing amounts of concurrency. For most problems, developing an algorithm that achieves its maximal concurrency requires a combination of cleverness and experience from the programmer.

As motivating examples, consider matrix addition and matrix multiplication. Mathematically, we might express these operations as follows:

Matrix addition: Given matrices A and B $(m \times n)$,
$A + B = C\ (m \times n),$ where $C_{i,j} = A_{i,j} + B_{i,j}.$

Matrix multiplication: Given matrices A $(m \times n)$ and B $(n \times o)$,
$A \times B = C\ (m \times o),$ where $C_{i,k} = \sum_{j=1}^{n} A_{i,j} \cdot B_{j,k}$

Consider the component operations that are required to implement these definitions. Matrix addition requires $m \cdot n$ pairwise sums to be computed. Matrix multiplication requires the evaluation of $m \cdot n \cdot o$ pairwise products and $\Omega(m \cdot \log n \cdot o)$ pairwise sums. In considering the parallel implementation of either of these algorithms, programmers must ask themselves, "can all of the component operations be performed simultaneously?" Looking at matrix addition, a wise parallel programmer would conclude that they can be computed concurrently—each sum is independent from the others, and therefore they can all be computed simultaneously. For matrix multiplication, the programmer would similarly conclude that all of the products could be computed simultaneously. However, each sum is dependent on values obtained from previous computations, and therefore they cannot be computed completely in parallel with the products or one another.

As a result of this analysis, a programmer might conclude that matrix addition is inherently more concurrent than matrix multiplication. As a second observation, the programmer should note that for matrices of a given size, matrix multiplication tends to involve more operations than matrix addition.

If the programmer is designing an algorithm to run on p processors where $p \ll m, n, o$, a related question is "are there better and worse ways to divide the component operations into p distinct sets?" It seems likely that there are, although the relevant factors may not be immediately obvious. The rest of this section describe some of the most important ones.

1.2.2 Data Distribution

Another challenge in parallel programming is the distribution of a problem's data. Most conventional parallel computers have a notion of *data locality*. This implies that some data will be stored in memory that is "closer" to a particular processor and can therefore be accessed much more quickly. Data locality may occur due to each processor having its own distinct local memory—as in a distributed memory machine—or due to processor-specific caches as in a shared memory system.

Due to the impact of data locality, a parallel programmer must pay attention to where data is stored in relation to the processors that will be accessing it. The more local the values are, the quicker the processor will be able to access them and complete its work. It should be evident that distributing work and distributing data are tightly coupled, and that an optimal design will consider both aspects together.

For example, assuming that the $m \times n$ sums in a matrix addition have been divided between a set of p processors, it would be ideal if the values of A, B, and C were distributed in a corresponding manner so that each processor's sums could be computed using local values. Since there is a one-to-one correspondence between sums and matrix values, this can easily be achieved. For example, each processor p_k could be assigned matrix values $A_{i,j}$, $B_{i,j}$, and $C_{i,j}$, $\forall i \equiv k \pmod p, \forall j \in 1 \ldots n$.

Similarly, the implementor of a parallel matrix multiplication algorithm would like to distribute the matrix values, sums, and products among the processors such that each node only needs to access local data. Unfortunately, due to the data interactions inherently required by matrix multiplication, this turns out to be possible only when matrix values are explicitly replicated on multiple processors. While this replication may be an option for certain applications, it runs counter to the general goal of running problems that are p times bigger than their sequential counterparts. Such algorithms that rely on replication to avoid communication are not considered *scalable*. Furthermore, replication does not solve the problem since matrix products are often used in subsequent multiplications and would therefore require communication to replicate their values across the processor set after being computed.

To create a scalable matrix multiplication algorithm, there is no choice but to transfer data values between the local memories of the processors. Unfortunately, the reality is that most interesting parallel algorithms require such communication, making it the next parallel programming challenge.

1.2.3 Communication

Assuming that all the data that a processor needs to access cannot be made exclusively local to that processor, some form of data transfer must be used to move remote values to a processor's local memory or cache. On distributed memory machines, this communication typically takes the form of explicit calls to a library designed to move values from one processor's memory to another. For shared memory machines, communication involves cache coherence protocols to ensure that a processor's locally cached values are kept consistent with the main memory. In either case, communication constitutes work that is time-consuming and which was not present in the sequential implementation. Therefore, communication overheads must be minimized in order to maximize the benefits of parallelism.

Over time, a number of algorithms have been developed for parallel matrix multiplication, each of which has unique concurrency, data distribution, and communication characteristics. A few of these algorithms will be introduced and analyzed during the course of the next few chapters. For now, we return to our final parallel computing challenges.

1.2.4 Load Balancing

The execution time of a parallel algorithm on a given processor is determined by the time required to perform its portion of the computation plus the overhead of any time spent performing communication or waiting for remote data values to arrive. The execution time of the algorithm as a whole is determined by the longest execution time of any of the processors. For this reason, it is desirable to balance the total computation and communication between processors in such a way that the maximum per-processor execution time is minimized. This is referred to as *load balancing*, since the conventional wisdom is that dividing work between the processors as evenly as possible will minimize idle time on each processor, thereby reducing the total execution time.

Load balancing a matrix addition algorithm is fairly simple due to the fact that it can be implemented without communication. The key is simply to give each processor approximately the same number of matrix values. Similarly, matrix multiplication algorithms are typically load balanced by dividing the elements of C among the processors as evenly as possible and trying to minimize the communication overheads required to bring remote A and B values into the processors' local memories.

1.2.5 Implementation and Debugging

Once all of the parallel design decisions above have been made, the nontrivial matter of implementing and debugging the parallel program still remains. Programmers often implement parallel algorithms by creating a single executable that will execute on each processor. The program is designed to perform different computations and communications based on the processor's unique ID to ensure that the work is divided between instances of the executable. This is referred to as the *Single Program, Multiple Data* (SPMD) model, and its attractiveness stems from the fact that only one program must be written (albeit a nontrivial one). The alternative is to use the *Multiple Program, Multiple Data* (MPMD) model, in which several cooperating programs are created for execution on the processor set. In either case, the executables must be written to cooperatively perform the computation while managing data locality and communication. They must also maintain a reasonably balanced load across the processor set. It should be clear that implementing such a program will inherently require greater programmer effort than writing the equivalent sequential program.

As with any program, bugs are likely to creep into the implementation, and the effects of these bugs can be disastrous. A simple off-by-one error can cause data to be exchanged with the wrong processor, or for a program to deadlock, waiting for a message that was never sent. Incorrect synchronization can result in data values being accessed prematurely, or for race conditions to occur. Bugs related to parallel issues can be nondeterministic and show up infrequently. Or, they may occur only when using large processor sets, forcing the programmer to sift through a large number of execution contexts to determine the cause. In short, parallel debugging involves issues not present in the sequential world, and it can often be a huge headache.

1.2.6 Summary

Computing effectively with a single processor is a challenging task. The programmer must be concerned with creating programs that perform correctly and well. Computing with multiple processors involves the same effort, yet adds a number of new challenges related to the cooperation of multiple processors. None of these new factors are trivial, giving a good indication of why programmers and scientists find parallel computing so challenging.

The design of the ZPL language strives to relieve programmers from most of the burdens of correctly implementing a parallel program. Yet, rather than making them blind to these details, ZPL's regions expose the crucial parallel issues of concurrency, data distribution, communication, and load balancing to programmers, should they care to reason about such issues. These benefits of regions will be described in subsequent chapters. For now, we shift our attention to the spectrum of techniques that one might consider when approaching the task of parallel programming.

1.3 Approaches to Parallel Programming

Techniques for programming parallel computers can be divided into three rough categories: parallelizing compilers, parallel programming languages, and parallel libraries. This section considers each approach in turn.

1.3.1 Parallelizing Compilers

The concept of a parallelizing compiler is an attractive one. The idea is that programmers will write their programs using a traditional language such as C or Fortran, and the compiler will be responsible for managing the parallel programming challenges described in the previous section. Such a tool is ideal because it allows programmers to express code in a familiar, traditional manner, leaving the challenges related to parallelism to the compiler. Examples of parallelizing compilers include SUIF, KAP, and the Cray MTA compiler [HAA$^+$96, KLS94, Ter99].

Listing 1.1: Sequential C Matrix Multiplication

```
for (i=0; i<m; i++) {
  for (k=0; k<o; k++) {
    C[i][k] = 0;
  }
}
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    for (k=0; k<o; k++) {
      C[i][k] += A[i][j] * B[j][k];
    }
  }
}
```

The primary challenge to automatic parallelization is that converting sequential programs to parallel ones is an entirely non-trivial task. As motivation, let us return to the example of matrix multiplication. Written in C, a sequential version of this computation might appear as in Listing 1.1.

Well-designed parallel implementations of matrix multiplication tend to appear very different than this sequential algorithm, in order to maximize data locality and minimize communication. For example, one of the most scalable algorithms, the SUMMA algorithm [vdGW95], bears little resemblance to the sequential triply nested loop. SUMMA consists of n iterations. On the i^{th} iteration, `A`'s i^{th} column is broadcast across the processor columns and `B`'s i^{th} row is broadcast across processor rows. Each processor then calculates the cross product of its local portion of these values, producing the i^{th} term in the sum for each of `C`'s elements. Figure 1.2 shows an illustration of the SUMMA algorithm.

The point here is that effective parallel algorithms often differ significantly from their sequential counterparts. While having an effective parallel compiler would be a godsend, expecting a compiler to automatically understand an arbitrary sequential algorithm well enough to create an efficient parallel equivalent seems a bit naive. The continuing lack of such a compiler serves as evidence to reinforce this claim.

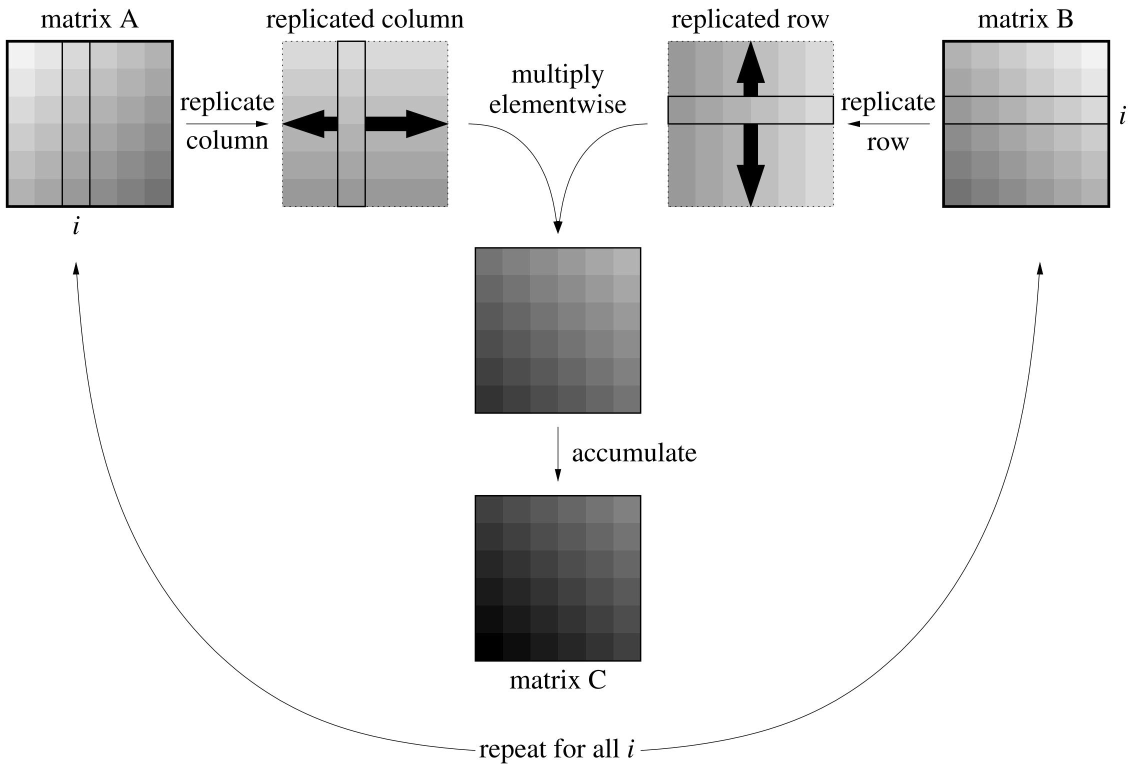


Figure 1.2: The SUMMA Algorithm For Matrix Multiplication

Many parallelizing compilers, including those named above, take an intermediate approach in which programmers add directives to their codes to provide the compiler with information to aid it in the task of parallelizing the code. The more of these that need to be relied upon, the more this approach resembles a new programming language rather than a parallelizing compiler, so further discussion of this approach is deferred to the next section.

1.3.2 Parallel Programming Languages

A second approach to parallel programming is the design and implementation of parallel programming languages. These are languages designed to support parallel computing better than sequential languages, though many of them are based on traditional languages in the hope that existing code bases can be reused. This dissertation categorizes parallel languages as being either *global-view* or *local-view*.

Global-view Languages

Global-view languages are those in which the programmer specifies the behavior of their algorithm as a whole, largely ignoring the fact that multiple processors will be used to implement the program. The compiler is therefore responsible for managing all of the parallel implementation details, including data distribution and communication.

Many global-view languages are rather unique, providing language-level concepts that are tailored specifically for parallel computing. The ZPL language and its regions form one such example. Other global-view languages include the directive-based variations of traditional programming languages used by parallelizing compilers, since the annotated sequential programs are global descriptions of the algorithm with no reference to individual processors. As a simple example of a directive-based global-view language, consider the pseudocode implementation of the SUMMA algorithm in Listing 1.2. This is essentially a sequential description of the SUMMA algorithm with some comments (directives) that indicate how each array should be distributed between processors.

Listing 1.2: Pseudo-Code for SUMMA Using a Global View

```
double A[m][n];
double B[n][o];
double C[m][o];
double ColA[m];
double RowB[o];

// distribute C [block,block]
// align A[:,:]  with C[:,:]
// align B[:,:]  with C[:,:]
// align ColA[:] with C[:,*]
// align RowB[:] with C[*,:]

for (i=0; i<m ; i++) {
  for (k=0; k<o; k++) {
    C[i][k] = 0;
  }
}

for (j=0; j<n; j++) {
  for (i=0; i<m; i++) {
    ColA[i] = A[i][j];
  }
  for (k=0; k<o; k++) {
    RowB[k] = B[j][k];
  }

  for (i=0; i<m ;i++) {
    for (k=0; k<o; k++) {
      C[i][k] += ColA[i] * RowB[k];
    }
  }
}
```

The primary advantage to global-view languages is that they allow the programmer to focus on the algorithm at hand rather than the details of the parallel implementation. For example, in the code above, the programmer writes the loops using the array's global bounds. The task of transforming them into loops that will cause each processor to iterate over its local data is left to the compiler.

This convenience can also be a liability for global-view languages. If a language or compiler does not provide sufficient feedback about how programs will be implemented, knowledgeable programmers may be unable to achieve the parallel implementations that they desire. For example, in the SUMMA code of Listing 1.2, programmers might like to be assured that an efficient broadcast mechanism will be used to implement the assignments to `ColA` and `RowB`, so that the assignment to `C` will be completely local. Whether or not they have such assurance depends on the definition of the global language being used.

Local-view Languages

Local-view languages are those in which the implementor is responsible for specifying the program's behavior on a per-processor basis. Thus, details such as communication, data distribution, and load balancing must be handled explicitly by the programmer. A local-view implementation of the SUMMA algorithm might appear as shown in Listing 1.3.

The chief advantage of local-view languages is that users have complete control over the parallel implementation of their programs, allowing them to implement any parallel algorithm that they can imagine. The drawback to these approaches is that managing the details of a parallel program can become a painstaking venture very quickly. This contrast can be seen even in short programs such as the implementation of SUMMA in Listing 1.3, especially considering that the implementations of its `Broadcast...()`, `IOwn...()`, and `GlobToLoc...()` routines have been omitted for brevity. The magnitude of these details are such that they tend to make programs written in local-view languages much more difficult to maintain and debug.

Listing 1.3: Pseudo-Code for SUMMA Using a Local View

```
int m_loc = m/proc_rows;
int o_loc = o/proc_cols;
int n_loc_col = n/proc_cols;
int n_loc_row = n/proc_rows;

double A[m_loc][n_loc_col];
double B[n_loc_row][o_loc];
double C[m_loc][o_loc];
double ColA[m_loc];
double RowB[o_loc];

for (i=0; i<m_loc ; i++) {
  for (k=0; k<o_loc; k++) {
    C[i][k] = 0;
  }
}

for (j=0; j<n; j++) {
  if (IOwnCol(j)) {
    BroadcastColSend(A,GlobToLocCol(j));
    for (i=0; i<m_loc; i++) {
      ColA[i] = A[i][j];
    }
  } else {
    BroadcastColRecv(ColA);
  }
  if (IOwnRow(j)) {
    BroadcastRowSend(B,GlobToLocRow(j));
    for (k=0; k<o_loc; k++) {
      RowB[k] = B[j][k];
    }
  } else {
    BroadcastRowRecv(RowB);
  }

  for (i=0; i<m_loc ;i++) {
    for (k=0; k<o_loc; k++) {
      C[i][k] += ColA[i] * RowB[k];
    }
  }
}
```

1.3.3 Parallel Libraries

Parallel libraries are the third approach to parallel computing considered here. These are simply libraries designed to ease the task of utilizing a parallel computer. Once again, we categorize these as global-view or local-view approaches.

Global-view Libraries

Global-view libraries, like their language counterparts, are those in which the programmer is largely kept blissfully unaware of the fact that multiple processors are involved. As a result, the vast majority of these libraries tend to support high-level numerical operations such as matrix multiplications or solving linear equations. The number of these libraries is overwhelming, but a few notable examples include the NAG Parallel Library, ScaLAPACK, and PLAPACK [NAG00, BCC$^+$97, vdG97].

The advantage to using a global-view library is that the supported routines are typically well-tuned to take full advantage of a parallel machine's processing power. To achieve similar performance using a parallel language tends to require more effort than most programmers are willing to make.

The disadvantages to global-view libraries are standard ones for any library-based approach to computation. Libraries support a fixed interface, limiting their generality as compared to programming languages. Libraries can either be small and extremely special-purpose or they can be *wide*, either in terms of the number of routines exported or the number of parameters passed to each routine [GL00]. For these reasons, libraries are a useful tool, but often not as satisfying for expressing general computation as a programming language.

Local-view Libraries

Like languages, libraries may also be local-view. For our purposes, local-view libraries are those that aid in the support of processor-level operations such as communication between processors. Local-view libraries can be evaluated much like local-view languages: they give the programmer a great deal of explicit low-level control over a parallel machine, but by nature this requires the explicit management of many painstaking details. Notable examples include the MPI and SHMEM libraries [Mes94, BK94].

1.3.4 Summary

This section has described a number of different ways of programming parallel computers. To summarize, general parallelizing compilers seem fairly intractable, leaving languages and libraries as the most attractive alternatives. In each of these approaches, the tradeoff between supporting global- and local-view approaches is often one of high-level clarity versus low-level control. The goal of the ZPL programming language is to take advantage of the clarity offered by a global-view language without sacrificing the programmer's ability to understand the low-level implementation and tune their code accordingly. Further chapters will develop this point and also provide a more comprehensive survey of parallel programming languages and libraries.

1.4 Evaluating Parallel Programs

For any of the parallel programming approaches described in the previous section, there are a number of metrics that can be used to evaluate its effectiveness. This section describes five of the most important metrics that will be used to evaluate parallel programming in this dissertation: performance, clarity, portability, generality, and a programmer's ability to reason about the implementation.

1.4.1 Performance

Performance is typically viewed as the bottom line in parallel computing. Since improved performance is often the primary motivation for using parallel computers, failing to achieve good performance reflects poorly on a language, library, or compiler.

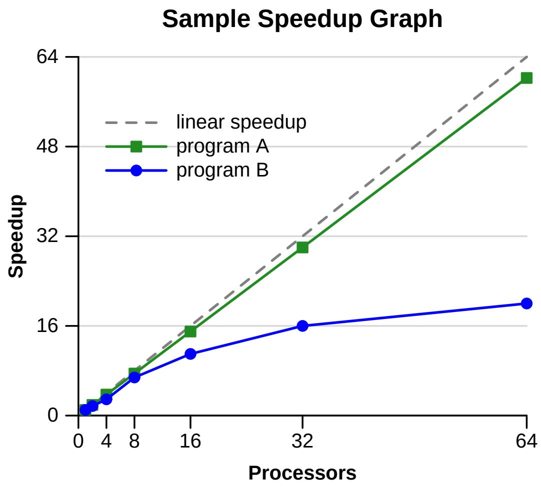


Figure 1.3: A Sample Speedup Graph. The dotted line indicates linear speedup ($speedup_p = p$), which represents ideal parallel performance. The "program A" line represents an algorithm that scales quite well as the processor set size increases. The "program B" line indicates an algorithm that does not scale nearly as well, presumably due to parallel overheads like communication. Note that these numbers are completely fabricated for demonstration purposes.

This dissertation will typically measure performance in terms of *speedup*, defined to be the fastest single-processor execution time (using *any* approach) divided by the execution time on p processors:

$$speedup_p = T_{1_{\min}}/T_p$$

If the original motivating goal of running a program p times faster using p processors is met, then $speedup_p = p$. This is known as *linear speedup*. In practice, this is challenging to achieve since the parallel implementation of most interesting programs requires work beyond that which was required for the sequential algorithm: in particular, communication and synchronization between processors. Thus, the amount of work per processor in a parallel implementation will typically be more than $1/p$ of the work of the sequential algorithm.

On the other hand, note that the parallelization of many algorithms requires allocating approximately $1/p$ of the sequential program's memory on each processor. This causes the working set of each processor to decrease as p increases, allowing it to make better use of the memory hierarchy. This effect can often offset the overhead of communication, making linear, or even *superlinear* speedups possible.

Parallel performance is typically reported using a graph showing speedup versus the number of processors. Figure 1.3 shows a sample graph that displays fictional results for a pair of programs. The speedup of program A resembles a parallel algorithm like matrix addition that requires no communication between processors and therefore achieves nearly linear speedup. In contrast, program B's speedup falls away from the ideal as the number of processors increases, as might occur in a matrix multiplication algorithm that requires communication.

1.4.2 Clarity

For the purposes of this dissertation, the clarity of a parallel program will refer to how clearly it represents the overall algorithm being expressed. For example, given that listings 1.2 and 1.3 both implement the SUMMA algorithm for matrix multiplication, how clear is each representation? Conversely, how much do the details of the parallel implementation interfere with a reader's ability to understand an algorithm?

The importance of clarity is often brushed aside in favor of the all-consuming pursuit of performance. However, this is a mistake that should not be made. Clarity is perhaps the single most important factor that prevents more scientists and programmers from utilizing parallel computers today. Local-view libraries continue to be the predominant approach to parallel programming, yet their syntactic overheads are such that clarity is greatly compromised. This requires programmers to focus most of their attention on making the program work correctly rather than spending time implementing and improving their original algorithm. Ideally, parallel programming approaches should result in clear programs that can be readily understood.

1.4.3 Portability

A program's portability is practically assured in the sequential computing world, primarily due to the universality of C and Fortran compilers. In the parallel world, portability is not as prevalent due to the extreme differences that exist between platforms. Parallel architectures vary widely not only between distinct machines, but also from one generation of a machine to the next. Memory may be organized as a single shared address space, a single distributed address space, or multiple distributed address spaces. Networks may be composed of buses, tori, hypercubes, sparse networks, or hierarchical combinations of these options. Communication paradigms may involve message passing, single-sided data transfers, or synchronization primitives over shared memory.

This multitude of architectural possibilities may be exposed by local-view approaches, making it difficult to implement a program that will run efficiently, if at all, from one machine to the next. Architectural differences also complicate the implementation of global-view compilers and libraries since they must run correctly and efficiently on all current parallel architectures, as well as those that may exist in the future.

Ideally, portability implies that a given program will behave consistently on all machines, regardless of their architectural features.

1.4.4 Generality

Generality simply refers to the ability of a parallel programming approach to express algorithms for varying types of problems. For example, a library which only supports matrix multiplication operations is not very general, and would not be very helpful for writing a parallel quicksort algorithm. Conversely, a global-view functional language might make it simple to write a parallel quicksort algorithm, but difficult to express the SUMMA matrix multiplication algorithm efficiently. Ideally, a parallel programming approach should be as general as possible.

Listing 1.4: Two matrix additions in C. Which one is better?

```
double A[m][n];
double B[m][n];
double C[m][n];

for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    C[i][j] = A[i][j] + B[i][j];
  }
}

for (j=0; j<n; j++) {
  for (i=0; i<m; i++) {
    C[i][j] = A[i][j] + B[i][j];
  }
}
```

1.4.5 Performance Model

This dissertation defines a *performance model* as the means by which programmers understand the implementations of their programs. In this context, the performance model need not be a precise tool, but simply a means of weighing different implementation alternatives against one another.

As an example, C's performance model indicates that the two loop nests in Listing 1.4 may perform differently in spite of the fact that they are semantically equivalent. C specifies that two-dimensional arrays are laid out in row-major order, and the memory models of modern machines indicate that accessing memory sequentially tends to be faster than accessing it in a strided manner. Using this information, a savvy C programmer will always choose to implement matrix addition using the first loop nest.

Note that C does not say how much slower the second loop nest will be. In fact, it does not even guarantee that the second loop nest *will* be slower. An optimizing compiler may reorder the loops to make them equivalent to the first loop nest. Or, hardware prefetching may detect the memory access pattern and successfully hide the memory latency normally associated with strided array accesses. In the presence of these uncertainties, experienced C programmers will recognize that the first loop nest should be no worse than the second. Given the choice between the two approaches, they will choose the first implementation every time.

C's performance model gives the programmer some idea of how C code will be compiled down to a machine's hardware, even if the programmer is unfamiliar with specific details like the machine's assembly language, its cache size, or its number of registers. In the same way, a parallel programmer should have some sense of how their code is being implemented on a parallel machine—for example, how the data and work are distributed between the processors, when communication takes place, what kind of communication it is, *etc*. Note that users of local-view languages and libraries have access to this information, because they specify it manually. Ideally, global-view languages and libraries should also give their users a parallel performance model with which different implementation alternatives can be compared and evaluated.

1.5 This Dissertation

This dissertation was designed to serve many different purposes. Naturally, its most important role is to describe the contributions that make up my doctoral research. With this goal in mind, I have worked to create a document that examines the complete range of effects that regions have had on the ZPL language, from their syntactic benefits to their implementation, and from their parallel implications to their ability to support advanced parallel computations. I also designed this dissertation to serve as documentation for many of my contributions to the ZPL compiler for use by future collaborators in the project. As such, some sections contain low-level implementation details that may not be of interest to those outside the ZPL community. Throughout the process of writing, my unifying concept has been to tell the story of regions as completely and accurately as I could in the time and space available.

In telling such a broad story, some of this dissertation's contributions have been made as a joint effort between myself and other members of the ZPL project—most notably Sung-Eun Choi, Steven Deitz, E Christopher Lewis, Calvin Lin, Ton Ngo, and my advisor, Lawrence Snyder. In describing aspects of the project that were developed as a team, my intent is not to take credit for work that others have been involved in, but rather to make this treatment of regions as complete and seamless as possible.

The novel contributions of this dissertation include:

- A formal description and analysis of the region concept for expressing array computation, including support for replicated and privatized dimensions.

- A parallel interpretation of regions that admits syntax-based evaluation of a program's communication requirements and concurrency.

- The design and implementation of a runtime representation of regions which enables parallel performance that compares favorably with hand-coded parallel programs.

- The design of the Ironman philosophy for supporting efficient paradigm-neutral communications, and an instantiation of the philosophy in the form of a point-to-point data transfer library.

- A means of parameterizing regions that supports the concise and efficient expression of hierarchical index sets and algorithms.

- Region-based support for sparse computation that permits the expression of sparse algorithms using dense syntax, and an implementation that supports general array operations, yet can be optimized to a compact form.

The chapters of this dissertation have a consistent organization. The bulk of each chapter describes its contributions. Most chapters contain an experimental evaluation of their ideas along with a summary of previous work that is related to their contents. Each chapter concludes with a discussion section that addresses the strengths and weaknesses of its contributions, mentions side issues not covered in the chapter proper, and outlines possibilities for future work.

This dissertation is organized as follows. The next three chapters define and analyze the fundamental region concept. First, Chapter 2 describes the role of the region as a syntactic mechanism for sequential array-based programming, using ZPL as its context. Then, Chapter 3 explains the parallel implications of regions, detailing their use in defining ZPL's performance model. The implementation of regions and of ZPL's runtime libraries is covered in Chapter 4. The two chapters that follow each describe an extension to the basic region concept designed to support more advanced parallel algorithms. The notion of a parameterized region is defined in Chapter 5 and its use in implementing multigrid-style computations is detailed. Chapter 6 extends the region to support sparse sets of indices, and demonstrates its effectiveness in a number of sparse benchmarks. Finally, Chapter 7 presents my concluding remarks and summarizes opportunities for future work.

Chapter 2

REGIONS AND THE ZPL LANGUAGE

This chapter describes the ZPL language, concentrating on the role of regions in its design. To make this discussion both readable and precise, some language concepts are introduced informally at first and are then reconsidered with increased formality as the chapter progresses. While this format would not be appropriate for a language reference manual, it is designed to provide an appropriate mixture of clarity and precision for this presentation.

Note that this chapter focuses on the sequential interpretation of ZPL, largely ignoring the parallel implications of regions and the language itself. Since parallelism is inherent in the definition and use of regions, this will leave some questions unanswered at the chapter's conclusion. These questions will be addressed in the following chapter, which describes the parallel implications of regions.

This chapter's description of ZPL is meant to provide a general overview of the language. For a more complete description, refer to the ZPL Programmer's Guide and the ZPL web page [Sny99, ZPL01].

The structure of this chapter is as follows. Sections 2.1–2.14 describe the ZPL language, including such fundamental concepts as regions, arrays, and array operators. Section 2.15 illustrates ZPL's use in a number of small sample applications that will be used in subsequent chapters. Section 2.16 describes other sequential approaches for array programming including vector indexing and slicing. Finally, Sections 2.17 and 2.18 provide an evaluation of ZPL's features in the sequential context, listing both benefits and liabilities of the region as it currently exists. This chapter's contents serve as an expanded discussion of work that was published previously [CLLS99, CLS99].

2.1 ZPL's Guiding Principles

Languages are for Communicating

One of the primary principles that has guided ZPL's development is the notion that programming languages are meant to be a means of communication between human and computer. Programmers have algorithms in their minds that they would like to execute on a computer. Computers have finite resources and an extremely limited capacity for understanding high-level languages. Programming languages should form a bridge between these two points, spanning the gap between programmer and computer using a direct, natural route that complements the abilities of both. When this principle is violated, communication is broken and a heroic effort is required by the user and/or compiler if the program is to have its intended effect.

Such broken languages can result in *macho compiling*, in which tremendous effort is put into helping a compiler recognize idioms that are not made apparent by the language and to implement them efficiently. These efforts tend to result in brittle optimizations that are easily broken if the programmer does not stick to the specific set of idioms that the compiler recognizes [Lew00]. When the optimization does not fire, programmers must expend great effort to achieve their desired results. Frustration abounds for both programmers and compiler implementors.

In contrast, creating a language that is natural to compile to a given architecture allows implementors more time to work on general improvements and optimizations, rather than worrying about particular syntactic patterns or corner cases. It should be noted that most programming languages which have enjoyed widespread use have not relied on sophisticated compiler optimizations to achieve acceptable baseline performance.

ZPL strives to implement this principle for parallel programming by providing a syntax that directly reflects parallelism. This allows users to express the parallelism that is inherent in their algorithms and to evaluate the parallel overheads of their programs. It also allows ZPL's implementors to detect parallelism trivially and create a straightforward baseline implementation. By avoiding the recognition problem, implementors can concentrate their efforts on optimizations that improve the baseline implementation.

The False Seduction of Legacy Code Reuse

Many parallel computing approaches have been designed in hopes of taking advantage of existing sequential codes with minimal programmer effort. For example, a perfect parallelizing compiler would transform sequential programs into parallel code automatically. Similarly, languages such as High Performance Fortran (HPF) [Hig94] and Co-Array Fortran (CAF) [NR98] were designed with the idea of leveraging existing code as a primary goal. Ideally, programmers can take their existing sequential programs, make minimal modifications to them, and end up with a good parallel implementation.

While this is a laudable goal, the assumption that incremental changes can turn a good sequential algorithm into a good parallel one is naive. The seductive pitch of these approaches is that the compiler will do all of the hard work for you once you add a line of code here or there to help it out. The reality of the situation is that the work required to transform sequential programs into an optimal parallelizable form is often nontrivial for both the programmer and the compiler [FJY98]. This effect is demonstrated by the conceptual leap between the sequential and SUMMA matrix multiplication implementations of Chapter 1. Often, a parallel code bears little resemblance to its sequential counterpart. In such cases, the effort required to convert a sequential program into an effective parallel one can be greater than that which would have been required to write a new program from scratch with parallelism in mind.

Starting from First Principles

ZPL approaches this problem from the opposite direction. Rather than starting with a sequential language and striving to detect the parallelism inherent in its (sequential) syntax, ZPL's design starts with nothing and incrementally adds concepts and operations that are

Listing 2.1: Simple Type, Constant, and Variable Declarations in ZPL

```
type
    age = shortint;
    coord = record
              x: integer;
              y: integer;
            end;

constant
        pi: double = 3.14159265;
        tabsize: integer = 1000;
        maxage: age = 128;

var done: boolean;
    length: integer;
    name: string;
    origin: coord;
    table: array [1..tabsize] of complex;
```

implicitly parallel. By starting from first principles in this way, ZPL was able to avoid supporting language constructs that disable parallelism. As an example, ZPL does not permit traditional scalar indexing of its parallel arrays, due to the fact that it is an inherently sequential construct. This approach forces programmers to consider the opportunities for parallelism in a program from its inception, rather than doing the minimal amount of work to get the compiler to accept their sequential code, and then spending hours with feedback tools trying to determine why it is not achieving good parallel performance.

ZPL's syntax is based on Modula-2 [Wir83] rather than a more popular language like C or Fortran. This decision reinforces the idea of "starting from scratch" by forcing C and Fortran users to confront the notion that certain features of those languages are not present in ZPL due to their interference with parallelism (*e.g.*, pointers, scalar array indexing, and common blocks). It also reinforces the idea that programmers should consider their sequential algorithms afresh when implementing them in parallel by making it difficult for existing C and Fortran codes to be tweaked slightly and run through the compiler.

Listing 2.2: Sample Configuration Variable Declarations in ZPL

```
config var
          n: integer = 100;          -- a sample problem size
          verbose: boolean = true;   -- use to control output

          logn: integer = lg2(n);    -- log of the problem size
          nsq: integer = n^2;        -- the problem size squared
          npi: double = pi*n;        -- n times the constant pi
```

A second reason for choosing Modula-2 was to support a language whose syntax is both readable and intuitive. While it would be possible to create C and Fortran dialects of ZPL, no such effort has been made at this point. The primary challenge would be to ensure that the features of C and Fortran which have been deliberately omitted from ZPL would interact appropriately with its parallel concepts (or simply outlaw them altogether).

As Chapter 4 will discuss, ZPL is compiled by translating it to C. For this reason, C's influence is occasionally seen in the language's syntax. For example, the names of ZPL's data types and its formatting of I/O both strongly reflect C.

2.2 Scalar ZPL Concepts

ZPL's scalar concepts are largely un-original and uninteresting, but form an important foundation for the rest of the language, so are described here quite briefly.

2.2.1 Data types, Constants, and Variables

To start with the basics, ZPL supports standard data types, type declarations, and declarations for constants and variables, as in most languages. It supports integers of varying sizes as well as floating point and complex values of varying precision. ZPL supports homogeneous array types (referred to as *indexed arrays*) and heterogeneous record types. For some sample type, constant, and variable declarations, refer to Listing 2.1.

Table 2.1: A Summary of ZPL's Scalar Operators

Arithmetic Operators		**Relational Operators**		**Assignment Operators**	
+	addition	=	equality	:=	standard
−	subtraction	!=	inequality	+=	accumulative
*	multiplication	<	less than	−=	subtractive
/	division	>	greater than	*=	multiplicative
%	modulus	<=	less than/equal	\=	divisive
ˆ	exponentiation	>=	greater than/equal	&=	conjunctive
				|=	disjunctive
Logical Operators		**Bitwise Operators**			
&	and	band	and		
|	or	bor	or		
!	not	bnot	complement		
		bxor	xor		

2.2.2 *Configuration Variables*

ZPL's *configuration variables* are a somewhat more unique scalar concept. Each configuration variable represents a *loadtime constant*—a value that can be defined at the outset of a program's execution but which cannot be changed thereafter. This allows users to define values that they may not want to constrain at compile time, such as problem sizes, verbosity levels, or tolerance values. The advantage of making such values configuration variables rather than traditional variables is that it allows the compiler to treat the variable as a constant of unknown value during analysis and optimization.

Configuration variables are defined similarly to normal constants, except that their initializing values are merely defaults that can be overridden on the program's command-line. Configuration variable initializers may be defined using expressions composed of constants, scalar procedures, and other configuration variables. Currently, ZPL only supports configuration variables of scalar types (including records and indexed arrays). Listing 2.2 shows some sample configuration variable declarations.

Listing 2.3: Sample Uses of ZPL's Control Structures

```
if (age > maxage) then
  writeln("Age too large!");
end;
...
for i := 1 to tabsize do
  table[i] := 0;
end;
...
repeat
  length /= 2;
  done := (length < 100);
until (done);
...
while (origin.x > origin.y) do
  leftshift(origin);
end;
```

2.2.3 Scalar Operators

ZPL supports a standard set of scalar arithmetic, logical, relational, bitwise, and assignment operators. See Table 2.1 for an overview.

2.2.4 Control Structures

ZPL supports standard control structures such as conditionals, for loops, while loops, and repeat-until loops. See Listing 2.3 for some simple examples.

2.2.5 Blank Array References

To encourage array-based thinking, ZPL's indexed arrays support a shorthand notation to operate over their entire index range without a loop. This is done by omitting the indexing expression for an array reference. For example, the assignment to `table` in Listing 2.3 could be written as follows using blank array references:

```
table[] := 0;
```

Listing 2.4: Sample ZPL Procedures

```
prototype mycomp(x: double; y: double): integer;

procedure leftshift(var pt: coord);
begin
  pt.x -= 10;
end;

procedure mycomp(x: double; y: double): integer;
begin
  if (x < y) then
    return -1;
  elsif (x = y) then
    return 0;
  else
    return 1;
  end;
end;
```

This syntactic shortcut is designed to aid with the common case of performing purely elementwise operations on indexed arrays. In many codes, blank array references can eliminate a number of trivial and uninteresting loops over an array's indices.

2.2.6 Procedures

ZPL's primary functional unit is the procedure, which can accept value or reference parameters and return a single value of arbitrary type. Procedures strongly resemble their Modula-2 counterparts and may be recursive. ZPL also supports prototypes that allow a procedure's signature to be declared for use before the procedure is defined. Listing 2.4 contains some sample prototype and procedure definitions.

2.2.7 Interfacing with External Code

Though ZPL's choice of Modula-2 as a base syntax limits the amount of code re-use that can take place within the parallel portion of a ZPL program, existing scalar code can be

Listing 2.5: An Example of Using `extern` in ZPL

```
extern constant M_PI: double;
extern var errno: integer;

extern type timezone = opaque;
            timeval = record
                        tv_sec: longint;
                        tv_usec: longint;
                      end;

extern prototype gettimeofday(var tv: timeval; var tz: timezone);
```

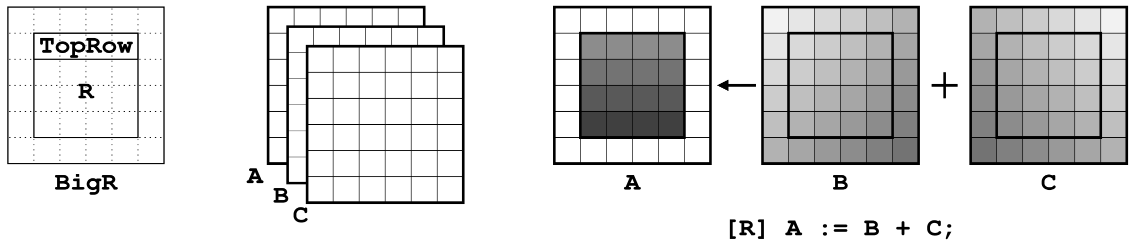
integrated into a ZPL program if it can be called by and linked into a C program. This is achieved using the `extern` keyword which can be applied to types, constants, variables, and procedures. External types may be partially specified or omitted completely using the `opaque` keyword, which allows the programmer to store variables of external types and pass them around, but not to operate on them directly or modify them. See Listing 2.5 for some sample external declarations.

2.3 Regions and Parallel Arrays

As mentioned in the introduction, ZPL's fundamental concept is that of the region. A region is simply an *index set*—a set of indices in a coordinate space of arbitrary dimensions. ZPL's regions are regular and rectilinear in nature. In this sense they are much like traditional arrays with no associated data. This similarity is emphasized syntactically: simple regions are defined using syntax that resembles a traditional array's bounds. For example, the following shows a simple two-dimensional region and the set of indices that it describes:

$$\texttt{[1..m, 1..n]} = \{(1,1),(1,2),\ldots,(1,n),(2,1),\ldots,(m,n)\}$$

Regions may contain *singleton dimensions* which describe only a single index value. These are defined by replacing the degenerate range with a single index (*e.g.*, `[1, 1..n]` rather than `[1..1, 1..n]`).



(a) *(b)* *(c)*

Figure 2.1: Using Regions and Arrays. (a) An illustration of the three regions declared in Section 2.3: `R`, `TopRow`, and `BigR`. (b) Three parallel integer arrays of size `BigR`—`A`, `B`, and `C`. (c) An example of how a statement's enclosing region scope restricts the range of its operators. Only indices within `R` (interior to the arrays) are referenced in this statement.

ZPL programmers can name regions. For example, the following declarations name the simple regions above "`R`" and "`TopRow`." They also create a third region, "`BigR`", which extends both dimensions of `R` by a single index in each direction.

```
region R = [1..m, 1..n];
       TopRow = [1, 1..n];
       BigR = [0..m+1, 0..n+1];
```

See Figure 2.1a for an illustration of these regions.

The dimension bounds of named regions must be expressions composed of constants or configuration variables. The *rank* or *dimensionality* of a region refers to the number of dimensions that it contains. For example, all of the regions above have rank 2.

Regions have two primary purposes. The first is to declare *parallel arrays*. This is done by specifying a region and an *element type* as a variable's type declaration. Such declarations result in the allocation of an array with an element of the specified type for each index described by the region. For example, the following declaration creates three $(m + 2) \times (n + 2)$ arrays of integers named `A`, `B`, and `C` (Figure 2.1b):

```
var A, B, C: [BigR] integer;
```

The rank of a parallel array is defined to be the rank of its region. For example, all of the parallel arrays above have a rank of 2. Parallel arrays may not be nested. That is, the element type of a parallel array may not contain a parallel array itself.

Parallel arrays are the primary data structure in ZPL, and will generally be referred to as "arrays" within this dissertation. The traditional scalar arrays described in Section 2.2 will always be referred to as "indexed arrays" to avoid confusion. Note that this chapter does not explain *why* parallel arrays are so named, but merely uses the term as a label. The following chapter provides the justification for the name (though discerning readers will possibly figure it out on their own).

The second purpose of regions is to provide indices for array references within a ZPL statement. Unlike indexed arrays, ZPL's parallel array elements cannot be referenced using traditional indexing mechanisms. Instead, regions are required to specify the indices for an array reference. As an example, consider the following statement:

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

This statement says to add arrays `B` and `C` elementwise, assigning their resulting sums to the corresponding values in `A`. The statement is prefixed by the *region scope* "`[R]`" which specifies that the addition and assignment operations should be performed for all indices described by `R`—namely, the interior $m \times n$ elements. Thus, this statement describes the matrix addition computation from Chapter 1. Region scopes serve as a form of universal quantification. For example, the statement above is equivalent to:

$$A_{i,j} \leftarrow B_{i,j} + C_{i,j}, \forall (i,j) \in \mathrm{R}$$

See Figure 2.1c for an illustration.

Using region scopes, any of ZPL's standard scalar operators can be applied to arrays in an elementwise manner. The chief constraint is that arrays cannot be read or written at indices that were not in their defining region (since no data is allocated for those indices).

Region definitions may also be specified explicitly within a region scope. These are called *dynamic regions*, since their bounds are typically based on expressions whose values are not known until runtime. For example, the following code fragment adds row `i` of arrays `B` and `C`, where `i` may be computed during the program's execution.

```
[i, 1..n] A := B + C;
```

Note that technically, this region scope should contain another set of square brackets to be consistent with the region specification syntax described previously. However, ZPL allows programmers to drop the redundant square brackets for readability.

Subsequent sections will describe regions in more depth, but for now this example-based overview of the ZPL language continues.

2.4 Array Operators

If ZPL could only express elementwise computations on its arrays, it would not be a very useful language. More general computations are supported by using *array operators* to modify a region scope's indices for a given array variable or expression. This section provides a brief introduction to the most important array operators: the *@ operator*, *floods*, *reductions*, and *remaps*.

2.4.1 The @ Operator

The @ operator (`@`) is ZPL's simplest array operator, providing a means for translating array references using constant offset vectors known as *directions*. Directions are specified and named in ZPL as follows:

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

These declarations create four vectors, one for each of the cardinal directions (Figure 2.2a).

The @ operator is applied to an array reference in a postfix manner, taking a direction as its second operand. Applying the @ operator to an array causes the indices of the enclosing region scope to be translated by the direction vector as they are applied to the array

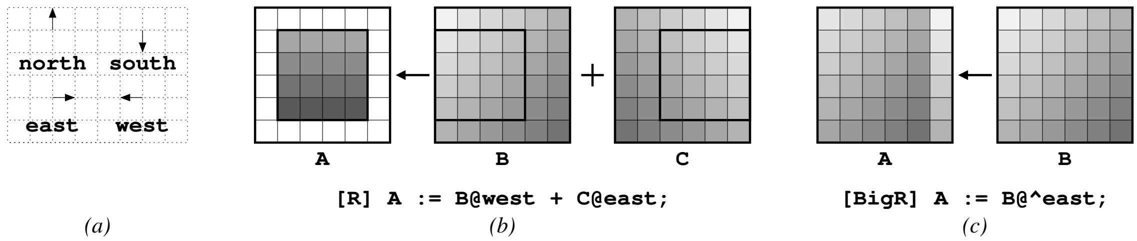


Figure 2.2: The @ Operator. (a) An illustration of the directions declared in Section 2.4.1. (b) A use of the @ operator to add shifted references of B and C, storing the result in region R of A. (c) A diagram illustrating the application of the wrap-@ operator to assign a cyclically-shifted version of B to A.

reference. For example, the expression A@south would increment all indices in the region by 1 in the first dimension. As a slightly more interesting example, consider the following statement:

```
[R] A := B@west + C@east;
```

This replaces each interior element of A with the element just to its left in B and just to its right in C. More formally:

$$A_{i,j} \leftarrow B_{i,j-1} + C_{i,j+1}, \forall (i,j) \in \mathtt{R}$$

Refer to Figure 2.2b for an illustration.

Note that the legality of this code hinges on the fact that B and C are declared using region BigR, causing the @-references to access declared values. Had they been declared using region R, the @-references would refer to values outside of their declared boundaries, which would be illegal.

Expressions using the @ operator may be used on either side of an assignment, but may not be passed by reference to a procedure. This dissertation will primarily concentrate on reading @-references and not writing them.

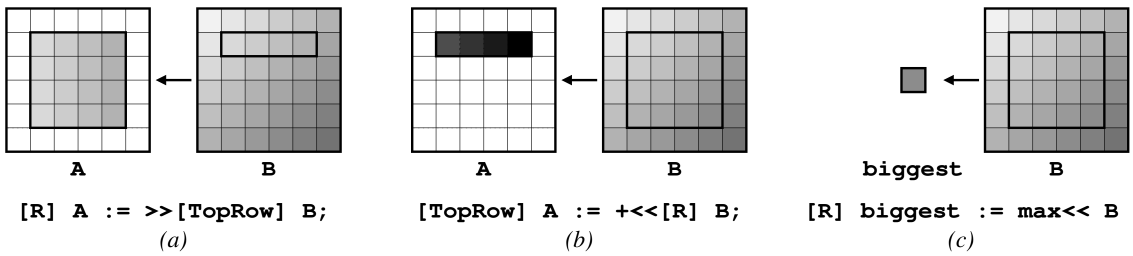


Figure 2.3: The Flood and Reduction Operators. (a) An illustration of the flood operator, causing the top row of `B` within `R` to be replicated across all rows of `A` within `R`. (b) An application of the sum reduction operator, which totals the values of `B` within each column of `R` and assigns the sum to the corresponding value of `A` within `TopRow`. (c) A full reduction which finds the biggest value of `B` within `R` and assigns the result to the scalar `biggest`.

The Wrap-@ Operator

One variation on the @ operator is the *wrap-@ operator* (**@^**), which causes accesses to the array that fall outside of its declared boundaries to wrap around and access the opposite side. Thus a statement like:

```
[BigR] A := B@^east;
```

would cyclically shift `B` one position to the left, assigning it to `A`.

2.4.2 The Flood Operator

The *flood operator* (**>>**) provides a means for replicating a slice of an array's values, either explicitly or implicitly. Symbolically, it can be viewed as taking a small piece of the array expression to its right and expanding it to make it bigger when used to the left. The flood operator is a prefix operator which is followed by a region to indicate the slice of the array to be replicated. This region is referred to as the *source region*, while the enclosing region of matching rank is called the *destination region*. As an example, consider the following assignment:

```
[R] A := >>[TopRow] B;
```

This statement assigns the first row of `B` (restricted to columns 1 through `n`) to rows 1 through `m` of `A`. See Figure 2.3a for an illustration.

In this statement, the flood operator's role is to replicate the values of `B` described by the source region (`TopRow` or `[1, 1..n]`) such that they conform to the destination region (`R`). This action can be interpreted in either an active or a passive way. Actively, the flood operator is taking the row of values described by `TopRow` and using it to create an array of size `R` for assignment to `A`. Passively, the operator can be thought of as causing the first dimension of indices in `R` to be ignored when accessing `B`, replacing them by the index 1. Formally, this statement can be interpreted as follows:

$$A_{i,j} \leftarrow B_{1,j}, \forall (i,j) \in \mathtt{R}$$

The main legality issues for the flood operator concern the conformability of the source and destination regions. First, they must be the same rank. In addition, each dimension of the source region must either be a singleton (as in this example's first dimension), or it must be identical to the destination region (as in the second dimension). The former case results in replication of the values described by the singleton index. The second results in a traditional array reference.

2.4.3 The Reduction Operator

The *reduction operator* (**<<**) is the dual of the flood operator. It compresses an array's values down to form a smaller array. As with the flood operator, it uses prefix notation and expects a source region to describe the values that should be reduced. The resulting size of the expression is described by the enclosing region scope of matching rank.

Because multiple values are being collapsed into a single item, some sort of reduction operation must also be specified to indicate how this collapsing should take place. These operations are typically commutative and associative, and they precede the reduction operator syntactically. Built-in reduction operations include addition, multiplication, min, and max, as well as logical and bitwise operators. Users may also create custom reduction operations using scalar ZPL procedures.

As a simple example, consider the following statement which uses a plus reduction:

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

This statement computes the sum of each column of B (for the rows and columns specified by R), storing each result in the first row of the corresponding column of A. See Figure 2.3b for an illustration. Again, this operator has both an active and a passive interpretation. Actively, it compresses B from rows 1 through m down to a single row (the first). Passively, it expands the reference to row 1 of B so that it refers to rows 1 through m, as combined using addition. Formally:

$$A_{i,j} \leftarrow \sum B_{k,l}, \forall(i,j) \in \texttt{TopRow}, \forall(k,l) \in \texttt{R}, \text{ such that } l = j$$

The legality rules for reductions are similar to those for the flood operator. The source and destination regions must have the same rank. In addition, each dimension of the source and destination regions must either be the same (causing the dimension to be read normally), or the destination dimension must contain a singleton (causing the values in that dimension to be reduced).

Full Reductions

One special case for reductions collapses an entire array to a single scalar value. This is known as a *full* or *complete reduction*, in contrast with the *partial reductions* described previously. Full reductions require only a single covering region since the scalar reference requires no indices. A simple example is shown here:

```
var biggest:integer;

[R] biggest := max<< B;
```

This statement finds the maximum value of B within the indices described by R and assigns it to the scalar value biggest. See Figure 2.3c for an illustration. Note that full reductions

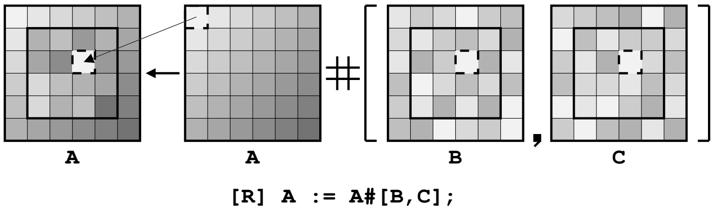


Figure 2.4: The Remap Operator. The B and C arrays serve as the map arrays for the remap of A in this assignment, thus they must contain values from 0 to 5 within region R (displayed here using varying levels of grey). As a specific example, consider the assignment to row 2, column 3, outlined with a dotted line. The corresponding values in B and C are both 0, indicating that element [0,0] of A should be assigned to this location.

compute the same value as a partial reduction over all dimensions, but they store the result in a scalar rather than an array element. For example, the full reduction above is similar to the following partial reduction:

```
[k1, k2] A := max<<[R] B;
```

2.4.4 The Remap Operator

The *remap operator* (#) serves as a catch-all operator, supporting parallel random array accesses. Unlike traditional array indexing, the remap operator requires an entire array of indices per dimension rather than a single index. The following is a simple example:

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

This use of the remap operator randomly accesses the *source array* A as specified by the *map arrays* B and C. In this statement, the result is assigned back into A. The B array provides the indices in the first dimension for each access to A, while C provides the indices for the second dimension. This is actually easiest to see in the formal version:

$$A_{i,j} \leftarrow A_{B_{i,j},C_{i,j}}, \forall(i,j) \in \mathtt{R}$$

This statement is illustrated in Figure 2.4.

The main legality constraint for the remap operator is that the number of map arrays must be equal to the rank of the source array so that each of its dimensions has an index. In addition, the map arrays must not refer to indices that are outside of the source array's defining region, since that would refer to values with no allocated storage. As Section 2.15.2 will demonstrate, remap operators can be used to operate on arrays of different ranks (and are in fact ZPL's only mechanism for doing so). Remap operators may be applied to expressions on either side of an assignment, though this dissertation focuses on uses on the right-hand side.

2.4.5 Other Array Operators

ZPL has a few other array operators that will not be described in this thesis, most notably the *scan operator* for performing *parallel prefix* operations, and the *wrap* and *reflect* operators for supporting boundary conditions. These are omitted in this discussion for brevity and because they do not pose significant challenges or intrigues in ZPL's design and implementation beyond the array operators described here. For more information on these operators, please refer to the literature [Sny99].

2.5 Formal Region Definition

Given the intuitive definitions of array operators, we now reconsider regions more formally. Each dimension of a region can be represented by a 4-tuple *sequence descriptor*, $r = (l, h, s, a)$. The variables l and h represent the low and high bounds of the sequence. The s value represents the sequence's stride, and a encodes its alignment. A sequence descriptor, r, is interpreted as defining a set of integers, $S(r)$, as follows:

$$S(r) \quad = \quad \{x | l \leq x \leq h \text{ and } x \equiv a \pmod{s}\} \tag{2.1}$$

For example, the descriptor $(1, 6, 2, 0)$ describes the set of even integers between one and six, inclusive: $\{2, 4, 6\}$.

A d-dimensional region, $\mathbf{r}$, is defined as a list of d sequence descriptors, $r_1 \ldots r_d$, where r_i represents the indices of the region's i^{th} dimension:

$$\mathbf{r} \quad = \quad \langle r_1, r_2, \ldots, r_d \rangle$$

The index set, $I(\mathbf{r})$, defined by a region $\mathbf{r}$ is simply the cross-product of the sets specified by each of its sequence descriptors:

$$I(\mathbf{r}) \quad = \quad S(r_1) \times S(r_2) \times \ldots \times S(r_d)$$

For example, the index set of the 2-dimensional region $\langle(1,6,2,0),(1,6,2,1)\rangle$ would be defined as follows:

$$\begin{aligned} I(\langle(1,6,2,0),(1,6,2,1)\rangle) &= S(1,6,2,0) \times S(1,6,2,1) \\ &= \{2,4,6\} \times \{1,3,5\} \\ &= \{(2,1),(2,3),(2,5),(4,1),(4,3),(4,5),(6,1) \\ &\qquad (6,3),(6,5)\} \end{aligned}$$

Recall the simple region declarations described in Section 2.3 which take the following general form:

R = [l_1..h_1, l_2..h_2, ..., l_d..h_d]

Such declarations correspond to the following formal region definition:

$$\mathbf{r} \quad = \quad \langle(l_1, h_1, 1, 0), (l_2, h_2, 1, 0), \ldots, (l_d, h_d, 1, 0)\rangle$$

These sequence descriptors specify that each dimension i contains all indices from l_i to h_i, due to the trivial values used for the stride and alignment. Note that while ZPL could allow programmers to express regions in a sequence descriptor format, the syntax used here allows the common case to be described in a clearer, more intuitive manner.

2.6 Region Operators

In addition to the simple region declarations of Section 2.3, ZPL provides a set of *region operators* that allow new regions to be created relative to existing ones. These are provided to give the user a more descriptive way of creating regions than specifying them by hand. They also provide the only means of changing a region's stride or alignment.

Region operators are defined using a set of *prepositional operators*—*of*, *in*, *at*, and *by*—that are defined for sequence descriptors. Each of these operators modifies a sequence descriptor using an integer value, δ. The operators are defined as follows:

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

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

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

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

To summarize, the **`of`** and **`in`** operators modify the sequence bounds relative to the existing bounds, leaving the stride and alignment unchanged. The **`of`** operator describes a range adjacent to the original range, whereas **`in`** describes a range interior to the previous range. The **`at`** operator translates the sequence bounds and alignment of a sequence. The **`by`** operator is used to scale the stride of the sequence and possibly shift the alignment, leaving the bounds unchanged.

Listing 2.6: Applications of Region Operators

```
direction north = [-1, 0];
          east2 = [ 0, 2];
          n2e3  = [-2, 3];
          step2 = [ 2, 2];

region R = [1..m, 1..n];
       NorthernBoundary = north of R;
       EasternInterior  = east2 in R;
       ShiftedN2E3      = R at n2e3;
       OddCols          = R by east2;
```

ZPL defines a region operator for each prepositional operator. Each region operator takes a *base region* and an offset vector in the form of a direction. The operator is evaluated by distributing each component of the direction to the region's corresponding sequence descriptor and applying the prepositional operator. For example, the **at** operator would be distributed as follows:

$$\begin{aligned}
\textbf{r at } [\delta_1, \delta_2] &= \langle (l_1, h_1, s_1, a_1), (l_2, h_2, s_2, a_2) \rangle \textbf{ at } [\delta_1, \delta_2] \\
&= \langle (l_1, h_1, s_1, a_1) \text{ at } \delta_1, (l_2, h_2, s_2, a_2) \text{ at } \delta_2 \rangle \\
&= \langle (l_1 + \delta_1, h_1 + \delta_1, s_1, a_1 + \delta_1), (l_2 + \delta_2, h_2 + \delta_2, s_2, a_2 + \delta_2) \rangle
\end{aligned}$$

As a more concrete example, the code in Listing 2.6 shows some direction declarations followed by region declarations that use the region operators. These regions, as well as several others, are illustrated relative to the base region `R` in Figure 2.5. In each case, the role of the direction in defining the new region is indicated. Though the formulas defining the prepositional operators seem fairly complex at first glance, they define regions which intuitively match the English definition of the preposition, making the mathematical definitions simply a formality. Intuitively, the **of** operator defines regions that are adjacent to the base region while **in** defines regions that are just within the base region. The **at** operator shifts the base region, while **by** strides the base region. In each case, the offset vector provides the notion of the direction and magnitude of the operation.

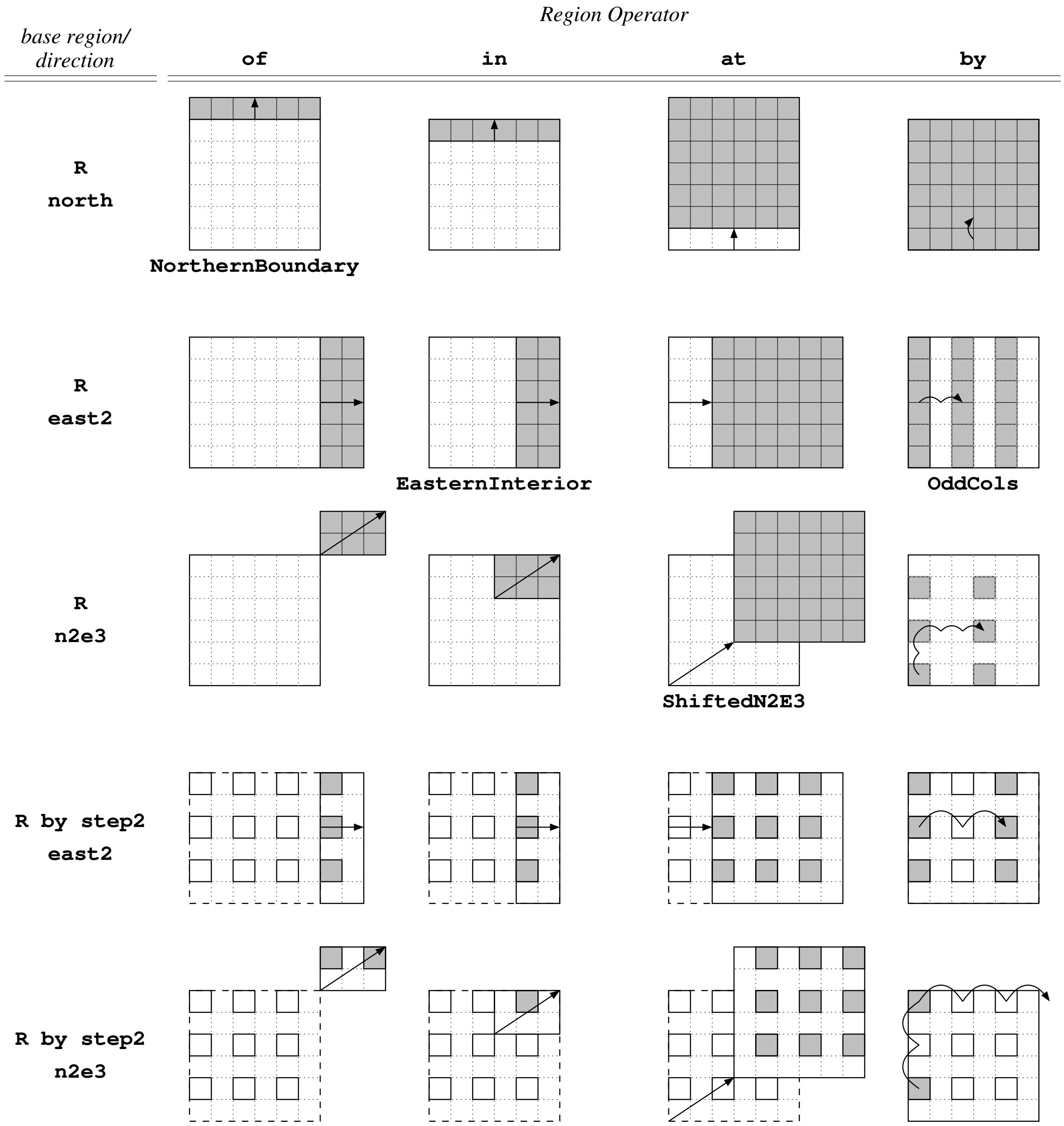


Figure 2.5: The Region Operators. This diagram illustrates the region operators applied using the declarations of Listing 2.6. Each column of pictures represents a single region operator (**of**, **in**, **at**, and **by**), while each row illustrates a different base region/direction pair. The indices of the regions created by applying the region operator to the base region and direction are shown in grey. Arrows gives a sense of the directions' roles in the definition. Those regions that were given names in Listing 2.6 are labeled.

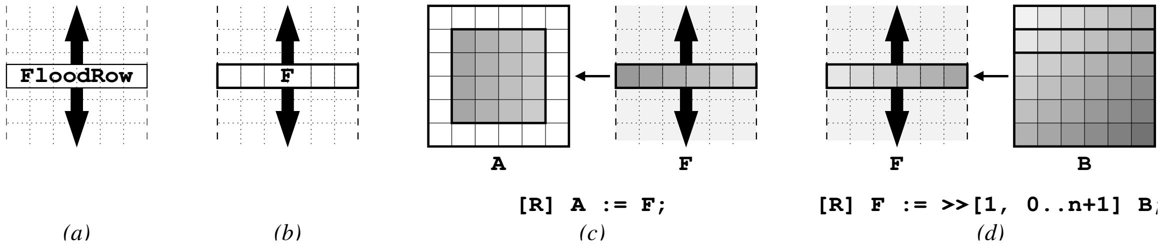


Figure 2.6: Flood Dimensions and Flood Arrays. (a) An illustration of a region whose first dimension is flooded. It represents a single row of values that are conformable to any row. (b) An array `F` declared using region `FloodRow`. (c) An assignment from `F` to `A` within region `R`. (d) An assignment from a row of `B` to `F` using the flood operator.

Although there are certainly other region operators that could be useful to a programmer, those listed here were selected as a basis set since they support common array references and are closed over our region notation. This chapter's discussion section considers this topic further.

2.7 Flood Dimensions

2.7.1 Introduction to Flood Dimensions

In addition to traditional dimensions (*e.g.*, `l..h`) and singleton dimensions (*e.g.*, `i`), regions can have a third type of dimension, the *flood dimension*. Flood dimensions are syntactically represented using an asterisk (`*`), and they represent a single index that conforms to any other index in the dimension. As an example, consider the following code fragment:

```
region FloodRow = [*, 0..n+1];

var F:[FloodRow] integer;

[R] A := F;
```

This code begins by declaring a region which is floodable in the first dimension and contains columns 1 through `n` in the second (Figure 2.6a). It then uses the region to declare a row of integers named F (Figure 2.6b). The assignment to A reads from the appropriate column of F for all rows in R. That is, the single row of values from F is explicitly replicated in rows 1 through m of A. See Figure 2.6c for an illustration.

Note that FloodRow differs from a row declared using a singleton dimension like TopRow. In particular, if F was declared using TopRow in the example above, the assignment would attempt to read from F in rows other than the first. This constitutes an error since F did not allocate storage for those rows. The use of the flood dimension in FloodRow allows it to conform to all indices, making the assignment legal.

2.7.2 Relationship with the Flood Operator

The code above illustrates a similarity between flood dimensions and the flood operator—both are used to represent replicated values. In fact, the flood operator can be used to assign to the values of a flood array. Consider the following example:

```
[FloodRow] F := >>[1, 0..n+1] B;
```

In this code fragment, row 1 of B is replicated by the flood operator to conform to the flood dimension of FloodRow (Figure 2.6d). Similar assignments without the flood operator would be illegal:

```
[FloodRow]   F := B;
[1, 0..n+1] F := B;
```

The first assignment is illegal because B is defined for rows 1 through m, making it ambiguous which row of B should be stored in F. Even if B was declared to be a single row (*e.g.*, [1, 0..n+1]), this assignment would remain illegal since the right-hand side of the assignment needs to conform to "all" row indices, not simply a particular one. For a standard array like B, this can only be achieved using the flood operator. The second assignment is illegal because it attempts to assign to a single row of F rather than assigning all of its rows using a flood dimension.

2.7.3 Formal Definition

As described above, an array with a flood dimension can intuitively be thought of as having a single set of values in that dimension which conform to all indices. Equivalently, the flood dimension can be thought of as representing an infinite number of indices, all of which are constrained to contain the same values.

Flood dimensions are represented using a special sequence descriptor: ($-\infty$, ∞, 0, 0). This states that the dimension covers all indices ($-\infty \ldots \infty$). The stride and alignment of 0 reflects the fact that there is a single implementing set of values and therefore no way to step from one index to the next. The flood sequence descriptor cannot be interpreted like those of traditional dimensions due to the nonsensical nature of working in a modulo-0 system. Rather, it serves as a placeholder that readily distinguishes flood dimensions from traditional ones. By convention, $S(-\infty, \infty, 0, 0)$ is defined to be $\{-\infty, \ldots, \infty\}$. The index defining the single set of values, will be referred to as $*$. For example, the element in the fourth column of `F` would be referred to as $F_{*,4}$.

Only the identity forms ($\delta = 0$) of the prepositional operators for sequence descriptors are defined for flood dimension sequence descriptors. This matches the intuitive sense that a dimension which represents an infinite number of indices cannot have adjacent or interior indices, cannot be shifted, and cannot be strided. Thus, only direction components of 0 may be applied to a flood dimension using ZPL's region operators.

The legality of interactions between flood dimensions, traditional dimensions, and array operators will be summarized in Section 2.12, which contains a more formal treatment of these subjects.

2.8 Index Constants

ZPL provides a set of built-in array constants referred to collectively as the *index constants*. These are a group of built-in virtual parallel arrays named **`Index1`**, **`Index2`**, **`Index3`**, *etc*. When read, each element of `Index`i evaluates to its index in the i^{th} dimen-

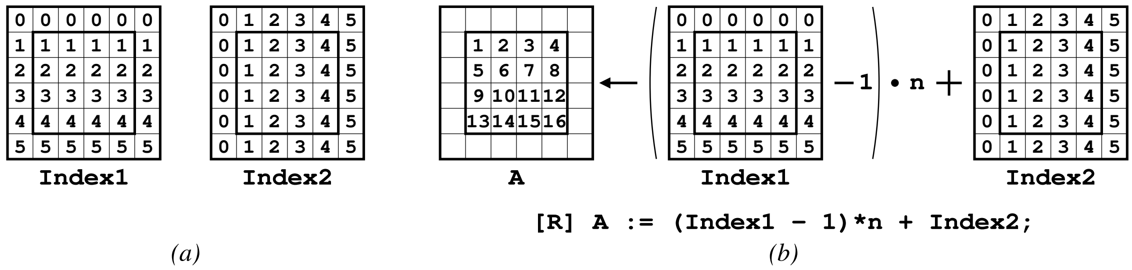


Figure 2.7: The Index Constants. (a) Pictures depicting **Index1** and **Index2**. BigR and R are indicated by the outlines. (b) A diagram showing the unique numbering of elements in R using **Index1** and **Index2**.

sion. Thus, **Index1**'s elements evaluate to their row indices, **Index2**'s elements to their column indices, *etc*. More formally:

$$\texttt{Index}i_{j_1,j_2,\ldots,j_d} = j_i$$

Figure 2.7a gives a pictorial depiction of **Index1** and **Index2** within regions R and BigR.

As a sample use, the following code fragment numbers the values of A within R from 1 to m·n in row-major order:

```
[R] A := (Index1 - 1)*n + Index2;
```

Using the formal definition of index constants, this assignment is interpreted as follows:

$$\begin{aligned} A_{i,j} &\leftarrow (Index1_{i,j} - 1) * n + Index2_{i,j} \\ &\leftarrow (i-1) * n + j \end{aligned}$$

See Figure 2.7b for an illustration.

As a second example, the following code uses the remap operator to assign the transpose of B to A within region R, assuming that m = n (if it did not, the map arrays might refer to values outside of B's declared size).

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

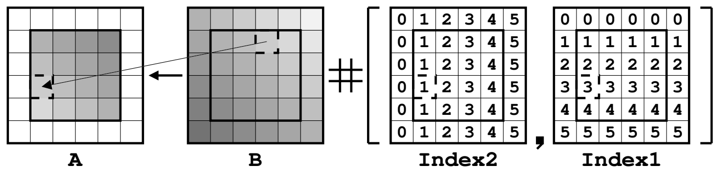


Figure 2.8: An Array Transpose. This diagram illustrates how the Index constants can be used to transpose arrays when used as the map arrays in a remap operation. By using column indices as the map array for `B`'s rows and row indices for its columns, the elements of `B` are transposed during their assignment to `A`. The dotted lines indicate how element $(3, 1)$ of `A` is assigned element $(1, 3)$ of `B`.

Using the formal definition of index constants, this assignment is interpreted as follows:

$$\begin{aligned} A_{i,j} &\leftarrow B_{Index2_{i,j},Index1_{i,j}} \\ &\leftarrow B_{j,i} \end{aligned}$$

See Figure 2.8 for an illustration of this assignment.

Each index constant can be thought of as being floodable in every dimension other than the i^{th}, since its size is arbitrarily large and its values only differ in dimension i. However, note that `Index`i conforms to arrays of rank i, $i + 1$, $i + 2$, *etc.*, making it more flexible than a similar user-defined flood array.

2.9 Masks

As defined in Section 2.3, regions must be rectilinear. This results in index sets that are very rectangular and regular, though possibly strided. In many applications, programmers may want to refer to a more arbitrary set of indices than those permitted by regions. For this reason, regions may be modified by boolean *masks* to select a subset of indices from the region. The mask must have the same rank as the region that it is modifying and must be allocated for all indices described by the region.

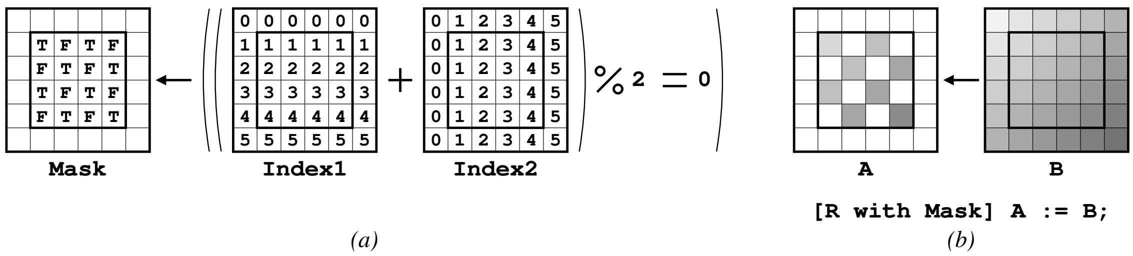


Figure 2.9: An Example of Using Masks. (a) The mask is assigned **true** for all locations in R where the sum of the row and column indices is even. (b) The mask is used to restrict the indices of R when assigning from B to A.

Here is a simple example that uses masks:

```
var Mask: [R] boolean;

[R] Mask := ((Index1 + Index2)%2 = 0);
[R with Mask] A := B;
```

The first assignment initializes the values of Mask to be true wherever the sum of the row and column indices is even (Figure 2.9a). The second assignment is prefixed by a region scope in which R is modified by Mask. This causes the assignment of B to A to take place only at those indices where the sum of the row and column indices is even. More formally:

$$A_{i,j} \leftarrow B_{i,j}, \forall(i,j) \in \text{R such that } Mask_{i,j} = \text{true}$$

See Figure 2.9b for an illustration. Masks can also be applied using the **without** keyword, which reverses the sense of the mask, computing only at indices where the mask's value is false.

Masks will not be covered with any more depth or formality in this chapter, as they are fairly intuitive and not a central concept in this dissertation. In general, any region scope can be masked, and the mask has the effect of filtering the region's indices as they are applied to array expressions within the region's scope.

Listing 2.7: A Demonstration of Region Scoping

```
1 [R] begin
2       A := B@west + C@east;
3      [BigR] A := B@^east;
4       A := >>[TopRow] B;
5      [TopRow] A := +<<[R] B;
6       biggest := max<< B;
7      [k1, k2] A := max<<[R] B;
8       A := A#[B, C];
9     end;
```

2.10 Region Scoping

2.10.1 Region Scoping Overview

Up to this point, each statement that has referred to a parallel array has been prefixed by a region scope to provide the statement's base indices. In general, region scopes can be applied to an entire block of statements using compound statements like control flow or a simple **begin**. . .**end** block. Moreover, new region scopes can be applied to individual statements within the compound statement, eclipsing the enclosing scope for that statement but no others.

As an example, all of the array statements in Section 2.4 could be written in a single block of statements (though an admittedly nonsensical one) as shown in Listing 2.7. The outermost region scope, `[R]`, serves as the enclosing region for lines 2, 4, 6, and 8. Lines 3, 5, and 7 are enclosed by an overriding region scope. Floods and partial reductions (as in lines 4, 5, and 7) open additional region scopes that enclose their array arguments (`B`, for each statement in this example).

Region scopes should be thought of as being passive rather than active objects. They do not cause things to occur, but merely supply indices, if needed, for the array references that they enclose. To this end, statements may be enclosed by multiple region scopes of different ranks, each of which can provide indices for array references of matching rank.

Listing 2.8: An Example of Multiple Enclosing Region Scopes

```
region R1D = [1..m];
       R2D = [1..n, 1..p];

var x: integer;
    Y: [R1D] integer;
    Z: [R2D] integer;

[R1D] [R2D] begin
              x := 1;
              Y := 2;
              Z := 3;
            end;
```

As an example, consider Listing 2.8. In this fragment, the assignment to `x` is a scalar and therefore does not require the enclosing region scopes at all. The assignment to `Y` refers to a 1-dimensional array and therefore makes use of the enclosing 1-dimensional region scope `[R1D]`. Likewise, the assignment to `Z` is 2-dimensional and uses `[R2D]` as its enclosing region scope. The enclosing region scope that controls an expression's array references is referred to as its *covering region*.

2.10.2 Dynamic Region Scoping

Region scoping occurs not only within static blocks of code, but also across procedure calls. As an example, consider Listing 2.9. The `addmat()` procedure takes three array variables as arguments, adding two of them and assigning to the third. Note that since no region scope is specified within the procedure, each procedure call's enclosing 2D region scope will be used during execution. Thus, the first call performs the computation for all indices within `R`, the second performs it for the top row of `R`, and the third performs it for the k^{th} column of `R`.

Listing 2.9: A Demonstration of Dynamic Region Scoping

```
procedure addmat(var X, Y, Z: [BigR] integer);
begin
  X := Y + Z;
end;

[R]       addmat(A,B,C);
[TopRow] addmat(A,B,C);
[1..m,k] addmat(A,B,C);
```

2.10.3 Region Inheritance

Due to the scoped nature of regions, it is often useful to inherit aspects of the enclosing region scope when opening a new region scope. ZPL provides two mechanisms for inheritance, the *blank dimension* and the *double-quote reference* (`"`). Each is described here.

Blank Dimensions

When opening a dynamic region, one or more dimensions may be inherited from the enclosing region scope by omitting their definitions. As an example, consider that line 4 of Listing 2.7 can be re-written using a dynamic region as follows:

```
A := >>[1, 1..n] B;
```

However, since this statement is enclosed by region `R`, which also spans columns 1. . .`n`, the second dimension can be inherited using a blank dimension as follows:

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

Since floods require that all non-replicated dimensions match, this syntax can save some redundant specification. It is especially useful when the source region's indices are computed dynamically. The same technique can be used to rewrite the partial reduction of line 5 in Listing 2.7 as follows:

```
[1, ] A := +<< [1..m, ] B;
```

Listing 2.10: Region Inheritance Using Double-Quote References

```
[R] begin
      [north of "] A := 0;    -- " refers to R
      [south of "] A := 0;    -- " refers to R
      [east of "]  A := 0;    -- " refers to R
      [west of "]  A := 0;    -- " refers to R
    end;
```

Listing 2.11: Mask Inheritance Using a Double-Quote Reference

```
[R with Mask] begin
                A := 0;
                [[k, ] without "] A := 1;    -- " refers to Mask
              end;
```

Blank dimensions can inherit from a procedure's dynamically enclosing scope. In addition, they can be used to leave the size of formal array parameters unspecified. For example, the `addmat()` procedure of Listing 2.9 could be written in a more general manner using blank dimensions as follows:

```
procedure addmat(var X, Y, Z: [ , ] integer);
```

This specifies that `addmat()` takes three 2-dimensional parallel arrays as its parameters, but does not specify their size or indices.

Double-Quote References

Double-quote references are used within region scopes to refer to the enclosing region as a whole. This is especially useful with region operators. For example, the code fragment in Listing 2.10 zeroes out the four boundaries of variable `A` (Figure 2.10a). The rank of the inherited region is inferred from the direction supplied to the region operator. For example, in this listing, since `north` is 2-dimensional, the enclosing 2-dimensional region, `R`, is inherited. As with blank dimensions, the double-quote can be used to refer to a procedure's dynamically enclosing region scope.

(a) *(b)*

Figure 2.10: Region Inheritance Examples. In both diagrams, white is used to represent 0, black to represent 1, and grey to indicate values that are untouched. (a) The result of the assignments using double-quote references in Listing 2.10. (b) The result of the statements in Listing 2.11 using the same checkerboard mask as Figure 2.9.

The double-quote can also be used to inherit a mask from the enclosing region scope. For example, in Listing 2.11, the inner region scope restricts the enclosing scope `R` down to its k^{th} row. It then inherits the mask from the enclosing scope, determining its rank using that of the dynamic region. Thus, this code first zeroes out `A` for all indices in `R` for which `Mask` is true. It then assigns the value 1 to all elements for which it is false in the k^{th} row of `R`. See Figure 2.10b for an illustration.

2.11 Scalar Promotion

Scalar promotion is the idea of permitting a concept that is scalar in nature to interact naturally with a parallel array concept. Scalar promotion is an intrinsic concept in ZPL. For example, most of the sample codes in this chapter have made use of scalar promotion by using the scalar assignment operator, `:=`, to assign one array expression to another. Similarly, the codes have applied scalar addition, subtraction, multiplication, and modulus operators to array expressions with the understanding that the operator would be applied to corresponding elements of the arrays. In these instances, scalar promotion causes the operator to be applied to the array expressions one scalar at a time for all indices in the enclosing region. The use is so intuitive that it is virtually transparent.

Listing 2.12: An Example of Scalar Procedure Promotion

```
var W, V: [R] double;
    Res:  [R] integer;

[R] Res := mycomp(W, V);
[R] Res := mycomp(W, 0);
```

The rest of this section explores the concept of promotion and its uses in ZPL, beginning with a discussion of scalar conformability.

2.11.1 Conformability of Scalar Promotion

When a scalar operator is promoted and applied to two array arguments, ZPL requires that the expressions be of the same rank. This means, for example, that scalar addition cannot be used to add a one-dimensional array to a two-dimensional array (although a similar effect can be achieved by storing the one-dimensional values in a two-dimensional array with a flooded dimension). Furthermore, the result of any promoted scalar operator is an array expression with the same rank as its operands. These are the requirements for array conformability in ZPL.

Just as scalar operators can be promoted, so can scalar values. As an example, in Listing 2.8, the scalar constants 2 and 3 were assigned to parallel arrays `Y` and `Z`. In these assignments, the scalar is promoted much like a scalar operator. The scalar value is treated as an array of appropriate rank that stores the scalar value in every location. Scalar variables are much like arrays that are flooded in every dimension: they are conformable with arbitrary indices in any dimension, and they hold the same value at all locations. However, scalars are strictly more powerful than flood arrays in that they are conformable with arrays of arbitrary rank. That is, scalar values may interact with arrays of rank 1, 2, *etc*., whereas any user-defined flood array will have a fixed rank.

Listing 2.13: Using Shattered Control Flow to Compute an Array's Absolute Value

```
[R] if (A < 0) then
      B := -A;
    else
      B := A;
    end;
```

2.11.2 Procedure Promotion

Just as scalar operators can be promoted using array operands, so can scalar procedures be promoted using array actual parameters. As an example, the scalar procedure `mycomp()` in Listing 2.4 can be applied to array arguments as shown in Listing 2.12. In the first call, arrays `W` and `V` are passed to `mycomp()` an element at a time for all indices in `R`, with the return value being assigned to the corresponding value of `Res`. In the second call, only the first argument is promoted, forcing the second argument, a scalar, to be promoted to act as a 2D array, making the parameters conformable.

A promoted scalar procedure's actual parameters must have the same rank. For example, it would be illegal to call `mycomp()` with array arguments that were 2D and 3D, respectively. As expected, the return value of a promoted scalar procedure will be promoted to the rank of its array parameters.

Note that procedure promotion only applies to scalar procedures. That is, procedures which refer to regions, parallel arrays, or ZPL's array operators may not be promoted. In addition, regions that use I/O, modify global variables, or call other parallel procedures are considered to be parallel to ensure deterministic execution.

2.11.3 Shattered Control Flow

Just as scalar operators and functions can be promoted, so can control structures (conditionals, loops) that are traditionally scalar in nature. For example, consider the conditional in Listing 2.13 which branches based on an array expression rather than a scalar value.

Listing 2.14: Using Promoted Procedures Instead of Shards

```
procedure abs(x: integer): integer;
begin
  if (x < 0) then
    return -x;
  else
    return x;
  end;
end;

[R] B := abs(A);
```

This conditional is evaluated for each element of A described by region R. Array references within the body of the conditional refer to elements with the same indices at which the conditional was evaluated. Thus, the conditional in this example will assign each element of B the absolute value of its corresponding element in A for all indices in R.

This promotion of control structures is referred to as *shattered control flow* because the single thread of control which is implicit in traditional ZPL statements may now take different actions on an element-by-element basis. In effect, it is "shattered," giving each index its own logical thread of control. At the end of the shattered control flow statement (or *shard* for short), the conceptual threads are joined and a single thread of execution resumes.

It should be noted that shards are similar to inlining a promoted scalar function. For example, the code in Listing 2.13 could be rewritten as shown in Listing 2.14. For this reason, the bodies of shattered control flow statements have many restrictions similar to those for promoted scalar procedures. In particular, they may not contain regions or parallel array references whose rank differs from that of the controlling expression. Most array operators are also disallowed in shattered control flow expressions, though limited uses of the @ operator are allowed (corresponding to passing @-references to a procedure by value).

Table 2.2: Formal Definition of Writing an Array Within a Region Scope

`[r]` A `:= ...` (where A is defined over $\mathbf{a}$)

	$\mathbf{a}$ is normal or singleton	$\mathbf{a}$ is flooded
$\mathbf{r}$ is normal or singleton	Legal if $I(\mathbf{r}) \subseteq I(\mathbf{a})$. Writes $A_i, \forall i \in I(\mathbf{r})$.	Illegal since $I(\mathbf{r}) \subset I(\mathbf{a})$ and A's values must be identical.
$\mathbf{r}$ is flooded	Illegal, since $I(\mathbf{r}) \supset I(\mathbf{a})$.	Legal. Writes A_*.

2.12 Array Operators, More Formally

Now that regions have been formally defined, and their uses have been described more completely, the array operators can be defined more formally. This section gives a more precise definition of the array operators, and also for the legality of reading and writing arrays within an enclosing region. For simplicity, these definitions are given for the single-dimensional case. Multidimensional cases simply extend these rules by applying them to each dimension independently in the natural manner. We begin by defining simple array writes and reads.

2.12.1 Array Writes

Arrays can be modified by being on the left-hand side of an assignment operator or by being passed by reference to a promoted scalar procedure. To test the legality of a write to an array, each dimension of its defining region, $\mathbf{a}$, must be compared to that of the enclosing region scope of matching rank, $\mathbf{r}$. Table 2.2 summarizes the different cases that are possible, classifying them based on whether the dimensions of the enclosing region and the array's defining region are singleton, flooded, or a normal range of indices.

In the case that neither dimension is flooded, the write is legal so long as the array is declared over the indices referenced by the region. When both dimensions are flooded, the

Table 2.3: Formal Definition of Reading an Array Within a Region Scope

`[r]` ... `:=` $\ldots A \ldots$ (where A is defined over $\mathbf{a}$)

	$\mathbf{a}$ is normal or singleton	$\mathbf{a}$ is flooded
$\mathbf{r}$ is normal or singleton	Legal if $I(\mathbf{r}) \subseteq I(\mathbf{a})$. Reads $A_i, \forall i \in I(\mathbf{r})$.	Legal. Reads $A_*, \forall i \in I(\mathbf{r})$.
$\mathbf{r}$ is flooded	Illegal, since $I(\mathbf{r}) \supset I(\mathbf{a})$.	Legal. Reads A_*.

write is legal, and the single replicated value will be modified. As described in Section 2.7, the cases in which one dimension is flooded but the other is not are illegal due to the fact that one index set represents an infinite index range while the other is finite.

2.12.2 Array Reads

The legality of an array read is defined in Table 2.3. In most cases, array reads are identical in legality to array writes. The one exception is that it is legal to read an array's flood dimension within a non-flooded region dimension. In this case, the programmer is specifying that a finite subset of the infinite index space be read, which makes sense. All of the other cases match their array write counterparts.

2.12.3 The @ Operator

To be legal, the array and direction supplied to an @ operator must match in dimensionality. This rank is also used to determine the enclosing region $\mathbf{r}$. As with array reads and writes, the dimensions of $\mathbf{r}$, and the array's defining region, $\mathbf{a}$, must be considered. Table 2.4 defines the legality of each case. For each legal reference, the transformation from the array's data space to the region's index space is also given, indicating which array values are read for each index in the enclosing region.

Table 2.4: Formal definition of the @ Operator

`[r]` $\ldots A$`@[`δ`]`$\ldots$ (where A is defined over $\mathbf{a}$)

	$\mathbf{a}$ is normal or singleton	$\mathbf{a}$ is flooded
$\mathbf{r}$ is normal or singleton	Legal if $I(\mathbf{r}$ **at** $\delta) \subseteq I(\mathbf{a})$. Returns $A_{i+\delta}$ at $i, \forall i \in I(\mathbf{r})$.	Legal. Returns A_* at $i, \forall i \in I(\mathbf{r})$.
$\mathbf{r}$ is flooded	Illegal, since $I(\mathbf{r}) \supset I(\mathbf{a})$.	Legal. Returns A_* at $*$.

The cases in which one or both of the regions have a flood dimension are identical to a traditional array read. This implies that applying the @ operator to a flood dimension has no effect, as one would expect. When neither dimension is flooded, the legality condition is similar to that of an array read: if the array is declared for the region's indices shifted by the offset, the reference is legal. The array reference evaluates to the values located at those shifted indices.

2.12.4 The Flood Operator

Evaluating a flood operator differs from previous sections in that two regions are involved—the source ($\mathbf{s}$) and destination ($\mathbf{d}$) regions of the flood. The first legality constraint is that the array expression being flooded must match the source region in dimensionality. This rank is also used to determine the enclosing region, so these will match as well. In addition, it must be legal to read the array expression using the source region as its covering region.

If these conditions are met, corresponding dimensions of the source and destination region are compared, with the possible outcomes summarized in Table 2.5. Floods are typically used to replicate a single value across a range of indices, making the cases where $\mathbf{s}$ is a singleton and $\mathbf{d}$ is normal or flooded the interesting ones. These uses of the flood operator cause the value at the index indicated by the source region to be referenced for

Table 2.5: Formal Definition of the Flood Operator

`[`$\mathbf{d}$`] ...>>[`$\mathbf{s}$`]` $A\ldots$

	$\mathbf{s}$ is normal	$\mathbf{s}$ is singleton	$\mathbf{s}$ is flooded
$\mathbf{d}$ is normal	Legal if $I(\mathbf{s}) = I(\mathbf{d})$. Returns A_i at i, $\forall i \in I(\mathbf{d})$.	Legal. Returns A_k at i, where $I(\mathbf{s}) = \{k\}$, $\forall i \in I(\mathbf{d})$.	Legal. Returns A_* at i, $\forall i \in I(\mathbf{d})$.
$\mathbf{d}$ is singleton	Illegal, since $\lvert I(\mathbf{d})\rvert < \lvert I(\mathbf{s})\rvert$.	Legal. Returns A_k at i, where $I(\mathbf{s}) = \{k\}$, $I(\mathbf{d}) = \{i\}$.	Legal. Returns A_* at i, where $I(\mathbf{d}) = \{i\}$.
$\mathbf{d}$ is flooded	Illegal, since $\lvert I(\mathbf{d})\rvert > \lvert I(\mathbf{s})\rvert > 1$.	Legal. Returns A_k at $*$, where $I(\mathbf{s}) = \{k\}$.	Legal. Returns A_* at $*$.

all indices in the destination region. The case where $\mathbf{d}$ is also a singleton is considered a degenerate case—the value is replicated over that single index. When $\mathbf{s}$ is flooded or $\mathbf{s}$ and $\mathbf{d}$ are both normal and equal, the reference is treated as a traditional array read. When $\mathbf{s}$ is normal but $\mathbf{d}$ is not, replication is nonsensical, so these cases are illegal.

2.12.5 The Reduce Operator

The reduce operator also utilizes a source and destination region. Once again, the argument expression and the source region must match in rank, and it must be legal to read the argument in the context of the source region.

Table 2.6 summarizes the different cases for reduction operators. The cases are essentially the dual of the flood operator, as one would expect. For simplicity, the table describes a sum reduction, though other operators may be substituted by replacing the summations in the definitions. The interesting cases reduce a range of values to a single value, and these occur when $\mathbf{s}$ is normal and $\mathbf{d}$ is either a singleton or flood dimension. The degenerate case

Table 2.6: Formal Definition of the (plus) Reduce Operator

`[`$\mathbf{d}$`] ...+<<[`$\mathbf{s}$`]` $A\ldots$

	$\mathbf{s}$ is normal	$\mathbf{s}$ is singleton	$\mathbf{s}$ is flooded
$\mathbf{d}$ is normal	Legal if $I(\mathbf{s}) = I(\mathbf{d})$. Returns A_i at i, $\forall i \in I(\mathbf{d})$.	Illegal, since $\lvert I(\mathbf{s})\rvert = 1$ and $\lvert I(\mathbf{d})\rvert > 1$.	Legal. Returns A_* at i, $\forall i \in I(\mathbf{d})$.
$\mathbf{d}$ is singleton	Legal. Returns $\sum_{j \in I(\mathbf{s})} A_j$ at i, where $I(\mathbf{d}) = \{i\}$.	Legal. Returns A_j at i, where $I(\mathbf{s}) = \{j\}$ and $I(\mathbf{d}) = \{i\}$.	Legal. Returns A_* at i, where $I(\mathbf{d}) = \{i\}$.
$\mathbf{d}$ is flooded	Legal. Returns $\sum_{j \in I(\mathbf{s})} A_j$ at $*$.	Legal. Returns A_j at $*$, where $I(\mathbf{s}) = \{j\}$.	Legal. Returns A_* at $*$.

occurs when $\mathbf{s}$ is a singleton, causing the reduction to be trivial. If $\mathbf{s}$ is flooded or $\mathbf{s}$ and $\mathbf{d}$ are normal and equal, the dimension is treated as a traditional array read. The only case that is illegal is trying to reduce a single value down to a range of values, which is nonsensical.

Full reductions are less complicated than partial reductions. Legality is determined by whether the array can be legally read within the covering region of matching rank. If it can, all values described by that region are combined by the given operation and returned as a scalar.

2.12.6 The Remap Operator

The covering region for a remap expression is determined not by the rank of the source array, but by that of the map arrays being applied to it (all of which must have the same rank). The number of map arrays must equal the rank of the source array, so that they provide an index for each of its dimensions. In addition, it must be legal to read each map array within the context of the covering region.

Table 2.7: Formal Definition of the Remap Operator

`[r]` $\ldots A$`#[`M`]`$\ldots$ (where A is defined over $\mathbf{a}$)

	$\mathbf{a}$ is normal or singleton	$\mathbf{a}$ is flooded
$\mathbf{r}$ is normal or singleton	Legal if $M_i \in I(\mathbf{a}), \forall i \in I(\mathbf{r})$. Returns A_{M_i} at $i, \forall i \in I(\mathbf{r})$.	Legal. Returns A_* at $i, \forall i \in I(\mathbf{r})$.
$\mathbf{r}$ is flooded	Legal if $M_* \in I(\mathbf{a})$. Returns A_{M_*} at $*$.	Legal. Returns A_* at $*$.

The single-dimensional case is defined by Table 2.7. If neither the covering region, $\mathbf{r}$, nor the source array's defining region, $\mathbf{a}$, are flooded, the reference is legal as long as the array is declared for the indices described by the map array. The values corresponding to the map indices will be returned by the reference.

If $\mathbf{r}$ is flooded, the map array must be flooded as well in order to be read. The reference will therefore be legal if the source array is defined for the index stored in the map array's unique location, and the value corresponding to that index will be returned. If $\mathbf{a}$ is flooded, the value of the map array is inconsequential. Regardless of its value, the single defining value of the source array will be returned. This case corresponds to a traditional array read.

The multidimensional case is handled using the obvious extension: the legality of each dimension is tested independently, and the indices for each dimension are determined by reading each map array in turn. For each index in the covering region, the single value in the source array defined by the map arrays' indices is referenced.

2.13 Files and Input/Output

Console and file I/O have not been a primary focus of research in ZPL, nor will they serve a large role in this dissertation, but they deserve the very briefest mention. ZPL supports the ability to open files for reading and writing, and also supports the standard console input, output, and error streams (**zin**, **zout**, and **zerr**, respectively). ZPL supports **read**(), **write**(), and **writeln**() statements that can be used to read or write expressions to one of these streams or to a file. Expressions can be formatted using control strings like those accepted by C's `printf()` and `scanf()` routines. Binary I/O is supported using the **bread**() and **bwrite**() statements. Array expressions are read or written for all indices in the enclosing region scope of the same rank, in row-major order (with some minimal formatting in the case of text output).

2.14 ZPL Summary

This chapter's description of ZPL concludes with a brief recap of its contents. To summarize, ZPL contains traditional scalar language constructs using a Modula-based syntax. In addition, ZPL supports configuration variables that serve as runtime constants and can be set on the resulting executable's command line.

ZPL supports array-based programming using the concept of the region to represent a regular, rectilinear set of indices. Regions may be named or specified in-line. A region's dimensions can represent a range of indices (potentially strided), a single index, or a replicated index using a flood dimension. Region operators may also be used to create new regions from existing ones. Regions are used to declare parallel arrays, which are the primary unit of computation in ZPL. The language also supplies built-in `Index`*i* array constants which evaluate to their own indices in a particular dimension.

Regions are also used to define region scopes, which passively provide indices for parallel array references and expressions of matching rank. Array operators are used to transform a region's indices as applied to a particular array expression. Array operators support translation, replication, reduction, or general remapping of an array's values. Region scopes are dynamically scoped and may inherit from their enclosing scopes of matching rank. Masks can be applied to region scopes to filter out a subset of their indices.

ZPL allows the promotion of scalar operators, values, functions, and control flow to interact with arrays in a natural manner. It also contains support for binary and text I/O of scalar and array expressions to files or the console.

Nagging Questions

At this point, it is likely that there are several aspects of ZPL which seem arbitrary or strange. For example: Why does ZPL prevent interactions between regions and arrays of different rank if they are the same shape? Since the remap operator can be used to express translations, floods, and reductions, why does ZPL bother supporting other array operators? Why can ZPL regions only be applied to statements and certain array operators rather than arbitrary expressions? Why are flood dimensions non-conformable with singleton dimensions, given that they each represent a single set of defining values? Why are @-references not allowed to be passed by reference to parallel procedures?

The answers to these questions are based on the parallel interpretation of regions and arrays, and therefore will have to wait until the following chapter. For now, let us turn our attention to some sample applications written in ZPL.

2.15 Sample Codes

This section contains several sample applications written in ZPL. The problems considered are the Jacobi iteration, matrix-vector multiplication, matrix multiplication, and tridiagonal matrix multiplication. These applications were chosen because they are simple, well-known, and useful for demonstrating the language features described in this chapter. Most of the problems have a few different implementations to illustrate different approaches in ZPL. For a larger variety of application domains in ZPL, please consult the literature [WGS00, DLMW95, RBS96, LLST95, Sny99].

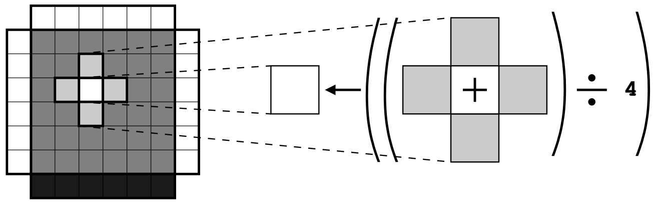


Figure 2.11: The Jacobi Iteration

2.15.1 Jacobi Iteration

The Jacobi iteration is a simple relaxation method for solving Laplace's equation on a regular grid [BBC$^+$94]. Given an initial approximate solution, it refines the values using a *five-point stencil* until the solution converges within some tolerance ϵ. The five-point stencil simply replaces each value by the average of its neighbors in the four cardinal directions. See Figure 2.11 for an illustration. The Jacobi iteration can be used, for example, to approximate the electric potential in a flat metal sheet whose edges have a fixed electric potential.

Listing 2.15 shows an implementation of the Jacobi iteration in ZPL. This code makes use of many of the concepts that this chapter introduced: configuration variables, regions, directions, and parallel arrays; region inheritance using blank dimensions and double-quote references; the @ operator and full reductions; promotion of scalars, operators, and procedures; and I/O.

The code begins with the **`program`** statement, which names the program and identifies the code's entry procedure. Lines 3–5 declare three configuration variables: `n`, which serves as the size of the grid; `epsilon` which specifies the termination condition; and `verbose` which indicates whether or not to print verbose output during the program's run.

Lines 7–8 declare two regions for the program. The first, `R`, is the region which specifies the size of the regular grid. The second region, `BigR`, is used to declare the main data array, which requires an extra row and column in each direction to store boundary values.

Listing 2.15: The Jacobi Iteration

```
program jacobi;

config var n: integer = 100;            -- the problem size
           epsilon: double = 0.00001;   -- the convergence condition
           verbose: boolean = false;    -- verbose output?

region R = [1..n, 1..n];                -- the computation indices
       BigR = [0..n+1, 0..n+1];         -- the declaration indices

var A: [BigR] double;                   -- the main data values
    New: [R] double;                    -- the new iteration's values
    delta: double;                      -- change between iterations

direction north = [-1, 0];              -- the four cardinal directions
          south = [ 1, 0];
          east  = [ 0, 1];
          west  = [ 0,-1];

procedure init(var X: [ , ] double);    -- array initialization routine
begin
  X := 0;
 [north of "] X := 0.0;
 [south of "] X := 1.0;
 [east of "]  X := 0.0;
 [west of "]  X := 0.0;
end;

procedure jacobi();                     -- the main entry point
[R] begin
      init(A);

      repeat
        New := (A@north + A@south +     -- five-point stencil on A
                A@east  + A@west)/4.0;

        delta := max<< fabs(A - New);   -- find maximum change

        A := New;                       -- copy back
      until (delta < epsilon);          -- continue while change is big

      if (verbose) then
        writeln("A:\n", A);             -- write data if desired
      end;

      writeln("delta: %le": delta);     -- always write delta
    end;
```

Lines 10–12 declare the variables for the problem. Array `A` serves as the primary data array, which is declared over region `BigR` to store the boundary values. Array `New` stores the new values computed during each iteration and requires no boundary values, so it is declared using region `R`. The variable `delta` is a scalar value that is used to store the maximum change that an array value undergoes in a single iteration.

Lines 14–17 declare the four cardinal directions, used to express the five-point stencil.

Lines 19–26 declare a procedure `init()` that is used to initialize the data array `A`. Note that this procedure is written in a generic manner for two-dimensional arrays, taking a 2D array of any size as its input parameter and containing statements that rely on dynamic region inheritance. The procedure zeroes out the array for all indices specified by the dynamically enclosing region scope, as well as its north, east, and west boundaries. The southern boundary is initialized to 1.0.

The main procedure spans lines 28–46. It opens a region scope using `R` that supplies indices to all parallel expressions within the procedure. It also serves as the enclosing region for the call to `init()` on line 30.

The main computation takes place in lines 32–39. Lines 33-34 compute the 5-point stencil on `A` using the @ operator and the four cardinal directions. The result is stored in the array `New`. Next, in line 36, the scalar `fabs()` routine is promoted across the array expression `A - New`. The `fabs()` routine is part of the standard C library and is included in ZPL's standard context. This computes the absolute value of the difference between corresponding elements of `A` and `New`. The resulting array of values is then collapsed to a scalar using the max reduction operator, and assigned to `delta`. The new values are assigned back into `A` in preparation for the next iteration in line 38. This loop is repeated until `delta` falls below the convergence value, `epsilon`.

Lines 41–45 output the results. If the `verbose` flag is true, line 42 prints the values of `A` described by `R` to the console in row-major order. The final value of `delta` is printed using exponential notation in line 45 and the program exits.

2.15.2 Matrix-Vector Multiplication

Matrix-vector multiplication is a fundamental operation that is used in a wide variety of numerical computations. This section considers two possible implementations using 2D and 1D vector representations.

2D Vector Implementation

Listing 2.16 shows an implementation of matrix-vector multiplication in ZPL. Though a fairly simple program, it demonstrates the use of flood dimensions, file I/O, partial reductions, and the remap operator.

Typically, matrices are thought of as being 2-dimensional while vectors are considered 1-dimensional. However, since ZPL makes interactions between 1D and 2D arrays non-trivial, this program represents all vectors using 2D arrays with either a flood or singleton dimension. In particular, it uses a flooded row array to store the vector argument so that its values will conform to all rows of the matrix.

Lines 3–5 declare the configuration variables. The values `m` and `n` are used to represent the number of rows and columns of the matrix, respectively. The third configuration variable is of the **`string`** type and stores the filename for reading the matrix and vector.

Lines 7–10 declare the regions for this program. Region `R` is the base region which describes the matrix indices. Lines 8–9 declare two row regions: `TopRow`, a singleton row, and `RowVect`, a flooded row. Line 10 declares a singleton column region, `ColVect`, that describes the result of the multiplication. Arrays are declared for each region in lines 12–15.

The `matvectmult()` procedure itself spans lines 17–34. Lines 20–23 open the file specified by the `filename` configuration variable and read values for matrix `M` and input vector `I` from it. Line 25 uses the flood operator to assign a replicated copy of the input vector to `V`, the flooded vector. Note that the source region for the flood is a dynamic region that inherits its second dimension from `RowVect`. Equivalently, the region `TopRow` could have served as the source region. The dynamic region is used here for illustrative purposes.

Listing 2.16: Matrix-Vector Multiplication Using 2D Vectors

```
program matvectmult;

config var m: integer = 100;           -- number of matrix rows
           n: integer = 100;           -- number of matrix columns
           filename: string = "MV.dat"; -- input filename

region R = [1..m, 1..n];               -- matrix index set
       TopRow =  [1, 1..n];            -- top row of the matrix
       RowVect = [*, 1..n];            -- row vector index set
       ColVect = [1..m, n];            -- col vector index set

var M: [R] double;                     -- the matrix
    I: [TopRow]  double;               -- the input vector
    V: [RowVect] double;               -- the vector flooded
    S: [ColVect] double;               -- the solution vector

procedure matvectmult();
var infile: file;
begin
  infile := open(filename, file_read);  -- open file
  [R]      read(infile, M);            -- read matrix values
  [TopRow] read(infile, I);            -- read vector values
  close(infile);                       -- close file

  [RowVect] V := >>[1, ] I;            -- flood the input vector

  [ColVect] begin
              S := +<<[R] (M * V);     -- matrix-vector mult.

              writeln(S);
            end;

  -- [RowVect] V := S#[Index2, n];     -- transpose solution?
end;
```

The actual matrix-vector multiplication takes place on line 28. Since `V` is flooded in its dimension, all of the vector values are aligned with the appropriate matrix values in `M`. Thus, they can simply be multiplied elementwise using scalar multiplication over region `R`. Since the solution vector is formed by summing the products in each row, a partial sum reduction is used to reduce the data from `R` down to the singleton column, `ColVect`. This represents the solution, which is written to the console in line 30.

In many matrix-vector multiplications, the matrix is square, and the solution vector must be used in subsequent multiplications. With this in mind, line 33 indicates how the solution vector could be re-assigned to a row vector using the remap operator. In particular, the column index (**`Index2`**) of the row is used to access the first dimension of `S` while the configuration variable `n` is promoted to access the second dimension.

It should be noted that region `RowVect` and array `V` could be completely omitted from this program by inlining the flood expression into the matrix-vector multiplication statement as follows:

```
S := +<<[R] (M * (>>[1, ] I));
```

For this discussion, this version was not used due to the fact that it is somewhat less clear, and does not demonstrate the use of flood dimensions.

Alternatively, region `TopRow` and array `I` could be removed from the program by reading directly into array `V`. While this would make the program even clearer, it was not used for this discussion in order to demonstrate the flood operator.

1D Vector Implementation

What if users really want to store their vectors as 1-dimensional arrays—is it possible in ZPL? Certainly, although the next chapter demonstrates that there may be compelling reasons to avoid such an implementation. This section illustrates matrix-vector multiplication using a 1D vector representation. For this program and all that follow, I/O will be omitted for brevity.

Listing 2.17: Matrix-Vector Multiplication Using 1D Vectors

```
1  program matvectmult;
2
3  config var m: integer = 100;             -- number of matrix rows
4             n: integer = 100;             -- number of matrix columns
5
6  region R = [1..m, 1..n];                 -- matrix index set
7         InVect = [1..n];                  -- 1D input vector indices
8         OutVect = [1..m];                 -- 1D output vector indices
9
10 var M: [R] double;                       -- the matrix
11     V: [InVect] double;                  -- the input vector
12     P: [R] double;                       -- an array of products
13     S: [OutVect] double;                 -- the solution vector
14
15 procedure matvectmult();
16 [R] begin
17       P := M * V#[Index2];               -- compute the mults
18                                          -- then sum the rows:
19      [OutVect] [ , n] S := (+<<[R] P)#[Index1, n];
20     end;
```

Listing 2.17 shows one way of writing such a code. To make a rather complex operation somewhat more readable, it has been broken into two lines (17 and 19). Line 17 computes the $m \cdot n$ products, storing them in array `P`. These products are computed using the remap operator to read the 1D input vector `V` as though it was a 2D array. Recall that the number of map arrays in a remap must match the rank of the source array (1 in this case), and that the rank of the result is inferred by the rank of the map arrays. This program uses **`Index2`** as its map array which has ambiguous rank since it is conformable to arrays of rank 2 or greater. However, in this case it must be 2D to allow the remap expression to conform to the multiplication with 2D array `M`.

Line 19 adds up the rows of `P`, assigning the result to `S`. This is done using a partial reduction as in the previous version using source region `R` and destination region `[ , n]`, which inherits rows `1..m` from `R`. Since storing the result in a 2D column vector seems contrary to the spirit of this approach, it is immediately remapped for assignment to `S` using **`Index1`** and `n` as its map arrays. As in the previous statement, **`Index1`** and the scalar `n`

Listing 2.18: The SUMMA Algorithm in ZPL

```
program summa;

config var m: integer = 100;              -- first dimension
           n: integer = 100;              -- inner dimension
           o: integer = 100;              -- last dimension

region RA = [1..m, 1..n];                 -- indices for A
       RB = [1..n, 1..o];                 -- indices for B
       RC = [1..m, 1..o];                 -- indices for C

var A: [RA] double;                       -- matrix A
    B: [RB] double;                       -- matrix B
    C: [RC] double;                       -- result matrix C

procedure summa();
var i: integer;
[RC] begin
       C := 0;                            -- zero C

       for i := 1 to n do                 -- loop over inner dim
         C += (>>[ , i] A) * (>>[i, ] B); -- cross ith col of A
       end;                               -- ...with ith row of B
     end;
```

have ambiguous rank, but they can be inferred to be 1D by context due to the assignment to `S`. The assignment itself is controlled by the enclosing 1D region scope `OutVect`.

That was fairly painful. The next chapter will show that this is not without good reason.

2.15.3 Matrix Multiplication

Matrix multiplication is yet another fundamental operation, and one that was used as motivation throughout Chapter 1. This section presents three different algorithms for matrix multiplication.

The SUMMA Algorithm

As described in the introduction, the SUMMA algorithm for matrix multiplication is considered one of the most scalable parallel approaches [vdGW95]. It has a fairly straightforward implementation in ZPL due to the support for replication provided by the flood operator. See Listing 2.18 for an implementation.

The program is fairly simple. The size of the matrices is specified by three configuration variables `m`, `n`, and `o`. A region is declared for each of the matrix sizes, and an array declared for each matrix. The execution is controlled by region `RC`, since all computations are done with respect to the result matrix, `C`. First `C` is zeroed out in line 18. Then, a loop is opened which specifies the n iterations of the algorithm. On iteration i, column i of `A` and row i of `B` are flooded across `RC` and multiplied elementwise, accumulating into `C`. At the end of the loop, `C` holds the result.

Cannon's Algorithm

Cannon's algorithm takes a systolic approach to matrix multiplication, illustrated in Figure 2.12. The algorithm begins by skewing the rows of A and the columns of B. In particular, each row i of `A` is cyclically shifted $i-1$ columns to the left. Similarly, column i of `B` is shifted $i-1$ rows upward. This has the effect of shifting `A`'s main diagonal into its first column and `B`'s main diagonal into its first row. Matrix `C` is initialized to contain all zeroes.

The main algorithm consists of n iterations. On each iteration, the initial $m \times o$ elements of each matrix are multiplied elementwise and accumulated into `C`. The `A` matrix is then cyclically shifted one row to the left and `B` is cyclically shifted one column upward. When all n iterations have completed, `C` contains the resulting matrix.

Listing 2.19 shows an implementation of Cannon's algorithm written in ZPL. The declarations are identical to those of the SUMMA algorithm, except that additional copies of `A` and `B` are declared to hold the skewed versions of the arrays. This was done in order to leave the original arrays unperturbed. Note that these copies could be eliminated by skewing the original matrices and then un-skewing them at the end of the computation. Here, the extra copies are used for simplicity. In addition to the extra arrays, two directions are declared for use in the shifting.

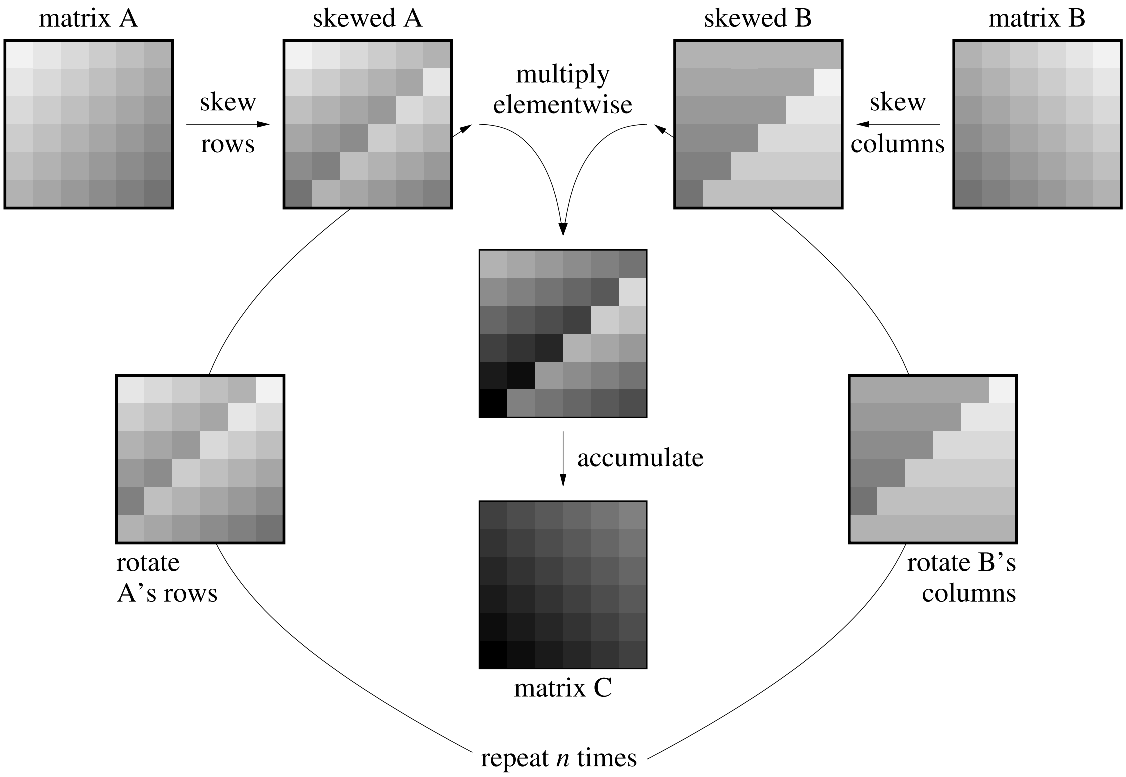


Figure 2.12: Cannon's Algorithm For Matrix Multiplication

Listing 2.19: Cannon's Algorithm in ZPL

```
program cannon;

config var m: integer = 100;                -- first dimension
           n: integer = 100;                -- inner dimension
           o: integer = 100;                -- last dimension

region RA = [1..m, 1..n];                   -- indices for A
       RB = [1..n, 1..o];                   -- indices for B
       RC = [1..m, 1..o];                   -- indices for C

var A: [RA] double;                         -- matrix A
    ASkew: [RA] double;                     -- skewed matrix A
    B: [RB] double;                         -- matrix B
    BSkew: [RB] double;                     -- skewed matrix B
    C: [RC] double;                         -- result matrix C

direction nextcol = [0, 1];                 -- directions for shifting
          nextrow = [1, 0];

procedure cannon();
var i: integer;
[RC] begin
       /* Skew A's rows and B's columns */
      [RA] ASkew := A#[Index1, ((Index2 + Index1 - 2)%n) + 1];
      [RB] BSkew := B#[((Index1 + Index2 - 2)%n) + 1, Index2];

       C := 0;                              -- zero C

       for i := 1 to n do
         C += ASkew * BSkew;                -- accumulate into C

        [RA] ASkew := ASkew@^nextcol;      -- shift ASkew
        [RB] BSkew := BSkew@^nextrow;      -- shift BSkew
       end;
     end;
```

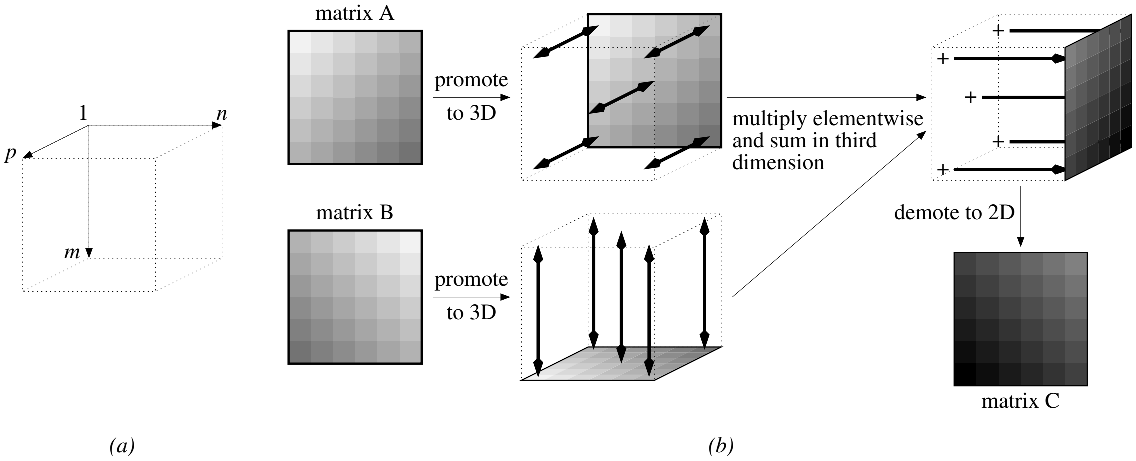


Figure 2.13: The PSP Algorithm For Matrix Multiplication

The initial skewing of the arrays is implemented in lines 24–25 using the remap operator and map expressions involving the **`Index1`** and **`Index2`** constant arrays. Matrix `C` is zeroed out in preparation for the main computation.

Within the main loop, line 30 performs a single elementwise multiplication of the skewed matrices, accumulating the products into `C`. Lines 32–33 use the wrap-@ operator to shift `ASkew` and `BSkew` for the next iteration. At the end of the program, `C` will contain the result matrix as expected.

PSP Algorithm

A third algorithm to consider is an instance of *problem space promotion* (PSP) [CLS99]. Problem space promotion is the idea of turning instances of iterations in an algorithm into explicit data dimensions. In particular, the PSP matrix multiplication algorithm converts the loop from 1 to n in the SUMMA and Cannon algorithms into a third data dimension. By doing so, the $m \cdot n \cdot o$ multiplications required for the matrix product are represented by a 3D index space (Figure 2.13a). Conceptually, matrix A represents one face of the box while matrix B represents a second perpendicular face. The algorithm proceeds by

Listing 2.20: PSP Matrix Multiplication in ZPL

```
program matmultpsp;

config var m: integer = 100;              -- first dimension
           n: integer = 100;              -- inner dimension
           o: integer = 100;              -- last dimension

region RA = [1..m, 1..n];                 -- 2D indices for A
       RB = [1..n, 1..o];                 -- 2D indices for B
       RC = [1..m, 1..o];                 -- 2D indices for C
       R3D  = [1..m, 1..n, 1..o];         -- 3D index space
       RA3D = [1..m, 1..n,  *  ];         -- 3D indices for A
       RB3D = [ *  , 1..n, 1..o];         -- 3D indices for B
       RC3D = [1..m, 1   , 1..o];         -- 3D indices for C

var A: [RA] double;                       -- matrix A
    B: [RB] double;                       -- matrix B
    C: [RC] double;                       -- result matrix C
    A3D: [RA3D] double;                   -- matrix A in 3D
    B3D: [RB3D] double;                   -- matrix B in 3D
    C3D: [RC3D] double;                   -- matrix C in 3D

procedure matmultpsp();
begin
  [RA3D] A3D := A#[Index1, Index2];       -- promote A to 3D
  [RB3D] B3D := B#[Index2, Index3];       -- promote B to 3D

  [RC3D] C3D := +<<[R3D] (A3D * B3D);     -- compute C in 3D

  [RC] C := C3D#[Index1, 1, Index2];      -- demote C to 2D
end;
```

replicating these faces throughout the box, computing their elementwise products, and then summing along the third dimension to form C. This idea is illustrated in Figure 2.13.

In ZPL, the $m \cdot n \cdot o$ elementwise products need not be represented explicitly, but can be expressed using flood dimensions and a partial reduction. See Listing 2.20 for an implementation. The code declares the same 2D configuration variables, regions, and arrays as in the previous codes. However, it also declares a 3D region to represent the 3-dimensional computation space and three faces within that space—two flood regions for the argument arrays and a third singleton region for the result.

The algorithm begins by using the remap operator to align the 2-dimensional `A` and `B` matrices in the 3D space (lines 24–25). The computation itself is expressed in line 27, which multiplies values of `A` and `B` within `R3D` and then reduces the products to the third plane of the space. Finally in line 29, the result array is mapped from 3D back to 2D.

2.15.4 Tridiagonal Matrix Multiplication

As a final application area, consider the multiplication of two tridiagonal matrices. Though any of the algorithms from the previous section can be used for this problem, the presence of so many zeroes allows more specialized techniques to be used. In particular, the resulting product will be a pentadiagonal matrix whose values are formed from the products of neighboring values in the tridiagonal argument matrices. See Figure 2.14 for an illustration. Since this code only needs to reference nearby neighbors, our implementations will use the @ operator rather than the floods and reductions of the previous matrix multiplication algorithms.

Mask-based Solution

One approach for implementing tridiagonal matrix multiplication in ZPL is to use a mask to restrict computation to one of the five resulting diagonals at a time. An implementation of this approach is given in Listing 2.21.

The implementation begins by declaring the problem size in line 3 and a region to describe the matrix indices in line 5. A larger region, `BigR` is also declared to store the argument matrices such that @-references can spill outside of the main problem area. The mask, arrays, and directions are declared in lines 8–16.

The implementation begins by zeroing out `C` so that all values not lying on the pentadiagonal will be correct. This could be done using a mask over all non-pentadiagonal indices, but the approach shown is asymptotically equivalent and used for simplicity. Next, each diagonal is computed one at a time by setting the mask using an expression that compares the

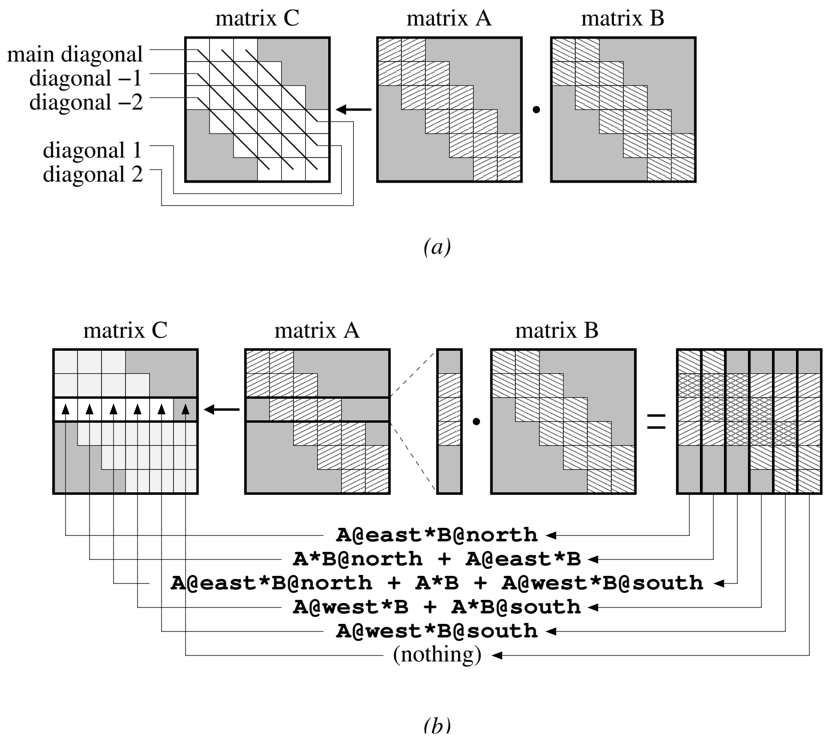


Figure 2.14: Tridiagonal Matrix Multiplication

Listing 2.21: Tridiagonal Matrix Multiplication in ZPL Using Masks

```
program trimask;

config var n: integer = 100;          -- assume n x n arguments

region R = [1..n, 1..n];              -- the base matrix size
       BigR = [0..n+1, 0..n+1];       -- matrix with boundaries

var A: [BigR] double;                 -- matrix A
    B: [BigR] double;                 -- matrix B
    C: [R] double;                    -- the product matrix, C
    Mask: [R] boolean;                -- mask for selecting diagonals

direction north = [-1, 0];            -- the four cardinal directions
          south = [ 1, 0];
          east  = [ 0, 1];
          west  = [ 0,-1];

procedure trimask();
[R] begin
      /* Assume we've zeroed A and B's boundaries */

      C := 0;                         -- zero out C

      /* Mask lowest diagonal (-2) and compute */
      Mask := (Index1 = Index2 + 2);
      [" with Mask] C := A@east * B@north;

      /* compute diagonal -1 */
      Mask := (Index1 = Index2 + 1);
      [" with Mask] C := (A * B@north) + (A@east * B);

      /* compute main diagonal */
      Mask := (Index1 = Index2);
      [" with Mask] C := (A@west * B@north) + (A * B) +
                         (A@east * B@south);

      /* compute diagonal 1 */
      Mask := (Index1 = Index2 - 1);
      [" with Mask] C := (A@west * B) + (A * B@south);

      /* compute diagonal 2 */
      Mask := (Index1 = Index2 - 2);
      [" with Mask] C := A@west * B@south;
    end;
```

Index1 and **Index2** arrays. The computation of each diagonal's values is expressed in a straightforward manner, using the mask to restrict it to the appropriate values. At the end of the program, `C` contains the matrix product.

Shattered Control Flow Solution

A second implementation is very similar to the first, but uses shattered control flow rather than a mask. The obvious advantage is that no time or space are required to compute and store the mask.

The declarations are identical to the mask-based version. The main computation consists of a shattered conditional that branches based on the relative values of **Index1** and **Index2**. Since the comparison of these arrays implies that they can be of any rank greater than 1, the body of the conditional is examined to determine that this is a 2-dimensional conditional, due to its references to `A`, `B`, and `C`. Each branch of the conditional simply assigns `C` using that diagonal's definition. The **else** clause at the end causes all non-pentadiagonal values to be zeroed out.

Compact Solution

The final implementation uses a more compact representation for the banded matrices. In particular, it uses an $n \times 3$ region, `Tri`, to represent the tridiagonal matrices and an $n \times 5$ region, `Pent`, to represent the resulting pentadiagonal. The regions' second dimensions refer to the diagonal numbers rather than matrix columns, and are therefore numbered between $-2 \ldots 2$, as appropriate. Note that directions `north` and `south` have been transformed to `ne` and `sw` to reflect this index space transformation. The tridiagonal region is also extended by an additional row in each direction to handle @-references that spill outside of the array's bounds.

The computation proceeds by opening a single dynamic region per diagonal which inherits its row dimension from the enclosing region, `Pent`. The expression to compute each

Listing 2.22: Tridiagonal Matrix Multiplication in ZPL Using Shattered Control Flow

```
program trishard;

config var n: integer = 100;           -- assume n x n arguments

region R = [1..n, 1..n];               -- the base matrix size
       BigR = [0..n+1, 0..n+1];        -- matrix with boundaries

var A: [BigR] double;                  -- matrix A
    B: [BigR] double;                  -- matrix B
    C: [R] double;                     -- the product matrix, C

direction north = [-1, 0];             -- the four cardinal directions
          south = [ 1, 0];
          east  = [ 0, 1];
          west  = [ 0,-1];

procedure trishard();
[R] begin
      /* Assume A and B's boundaries are zeroed out */

      /* shatter control flow based on the row and column indices */
      if (Index1 = Index2 + 2) then      -- compute diagonal -2
        C := A@east * B@north;
      elsif (Index1 = Index2 + 1) then   -- compute diagonal -1
        C := (A * B@north) + (A@east * B);
      elsif (Index1 = Index2) then       -- compute main diagonal
        C := (A@west * B@north) + (A * B) + (A@east * B@south);
      elsif (Index1 = Index2 - 1) then   -- compute diagonal 1
        C := (A@west * B) + (A * B@south);
      elsif (Index1 = Index2 - 2) then   -- compute diagonal 2
        C := A@west * B@south;
      else                               -- zero all other indices
        C := 0;
      end;
    end;
```

Listing 2.23: Tridiagonal Matrix Multiplication in ZPL Using Compact Arrays

```
program tridense;

config var n: integer = 100;          -- assume n x n arguments

region Tri  = [0..n+1, -1..1];        -- dense tridiagonal storage
       Pent = [1..n,   -2..2];        -- dense pentadiagonal storage

var A: [Tri] double;                  -- matrix A
    B: [Tri] double;                  -- matrix B
    C: [Pent] double;                 -- the product matrix, C

direction ne    = [-1, 1];            -- northeast (acts as north)
          sw    = [ 1,-1];            -- southwest (acts as south)
          east  = [ 0, 1];            -- east
          west  = [ 0,-1];            -- west

procedure tridense();
[Pent] begin
         /* Assume A and B's boundaries are zeroed out */

         /* one statement per diagonal in the product */
         [ ,-2] C := A@east * B@ne;
         [ ,-1] C := (A * B@ne) + (A@east * B);
         [ , 0] C := (A@west * B@ne) + (A * B) + (A@east * B@sw);
         [ , 1] C := (A@west * B) + (A * B@sw);
         [ , 2] C := A@west * B@sw;
       end;
```

diagonal is the same as in the previous codes, but substitutes `ne` for `north` and `sw` for `south`.

This implementation is attractive because it uses an amount of memory proportional to the number of interesting values in the problem, rather than the conceptual problem space. However, this has the disadvantage of making it more awkward to operate on tridiagonal matrices in conjunction with traditional $n \times n$ matrices. For example, adding a traditional matrix to a tridiagonal matrix in this format would require the remap operator to transform one index space to the other.

The next chapter will re-examine all of the sample codes in this section and further evaluate their strengths and weaknesses in the context of a parallel implementation.

2.16 Related Work

This section describes alternatives to region-based programming that are used to express array computations in other languages. Its focus is restricted to the indexing mechanisms of sequential languages. Parallel languages will be covered in the related work section of the following chapter.

2.16.1 Scalar Indexing

The oldest and most prevalent form of expressing array computation is *scalar indexing* or *array subscripting*, as found in languages such as FORTRAN, its later incarnation as FORTRAN 77 (F77) and more recent languages such as C [Mac87, Bac98, Sec78, KR88]. In each of these languages, basic operations such as assignment and addition are defined only for scalar values, and promotion is not supported. As a result, operations on arrays must be written to explicitly loop over the index space and reference the array's values one at a time. Scalar indexing allows the programmer to specify a single value per dimension in order to specify a single array value. For example, adding two arrays might appear as follows in F77:

```
do j = 1, n
  do i = 1, m
    C(i, j) = A(i, j) + B(i, j)
  enddo
enddo
```

The primary disadvantage of scalar indexing is that it burdens the programmer with the task of performing the explicit looping and subscripting required to express array computations. This can quickly become a tedious task that requires more keystrokes than it does intelligence. Moreover, the loops required by scalar indexing describe a sequential ordering on the operations that runs counter to parallelism. This is not a problem in a sequential context, but can complicate the parallelization of languages using scalar indexing in the parallel domain.

Scalar indexing does have the advantage of being a very simple and general mechanism for expressing array computations. For instance, there is no need for ZPL's array operators, nor its restrictions governing what types of expressions can and cannot interact. Moreover, conformability rules in such a language are simple: since all arguments to an operator must be scalars, the only check required is that all array dimensions are being indexed.

2.16.2 Vector Indexing

In the late 1950's Kenneth Iverson developed a mathematical notation that was designed to clarify some of the ambiguities that he felt standard mathematical notation contained. Shortly thereafter, this notation evolved into *APL* (*A Programming Language*), one of the earliest higher-level programming languages [Ive62, Mac87]. APL was the first pure array-based programming language, since *all* data items in the language are arrays (scalars are simply arrays of rank 0). APL's arrays support *vector indexing* in which the programmer supplies a vector of indices per dimension. The outer product of these vectors specifies the indices of the array. In this way, array references no longer refer to a single value of the array, but a subarray of values, or possibly the entire array. For example, matrix addition would appear as follows in APL:

$$C[1 + \iota M; 1 + \iota N] \leftarrow A[1 + \iota M; 1 + \iota N] + B[1 + \iota M; 1 + \iota N]$$

In this notation, ιk represents a k-element vector containing values from 0 to $k - 1$ (similar to ZPL's `Index`*i* arrays). Each element is incremented to use 1-based indexing to be consistent with the Fortran implementation. Thus, the outer product of these vectors causes each array reference to refer to elements $\{(1, 1), \ldots, (m, n)\}$. Addition (+) and assignment ($\leftarrow$) are promoted across the elements as in ZPL, and the sum is computed.

Although APL contained many elegant and revolutionary ideas, it has not remained in widespread use over the years. Detractors find it too terse and unreadable, due primarily to its large set of unique operators, most of which require non-standard characters (like the "$\leftarrow$" used for assignment above). Though enthusiasts are quick to rush to its defense, it remains largely unused and unknown today. Even so, its use of arrays as operators and vector subscripting have had an influence on more modern languages including ZPL.

2.16.3 Array Slicing

Modern array-based languages have adopted syntax to support APL's array operands without so much generality and built-in support for mathematical operators. Many of these languages have provided a syntax for accessing a regularly strided set of indices within an array. This is known as *array slicing* or *array sectioning*, and represents an excellent example of optimizing for the common case. Consider, for example, how few of the ZPL programs from the previous section would require an APL-style index vector that was not a simple ι-expression.

One language with support for array slicing is Fortran 90 (F90) [ABM+92], a successor to F77. F90 allows the user to specify indices using a 3-tuple per dimension: `[l:h:s]`. These values are identical to the low, high, and stride values in ZPL's sequence descriptors, and they can be used to express a wide variety of simple APL-style ι-expressions. In F90, l serves as the alignment value, which was the fourth value in ZPL's sequence descriptors.

Another language that supports slicing is Matlab, an interactive, interpreted matrix manipulation language [Mat93]. Matlab supports a slice notation similar to F90's, but without the stride value. In both languages, the low and high bounds may be omitted, which cause the array's declared bounds to be used. Omitting the stride in F90 results in a stride of 1. Omitting the slice notation altogether causes the entire array to be referenced. Matrix addition would appear as follows in F90 and Matlab:

```
C(1:n, 1:n) = A(1:n, 1:n) + B(1:n, 1:n)
```

As in APL, both F90 and Matlab allow the programmer to use vector indexing, though in practice this rarely seems to be used. For example, a five-element vector could be permuted in F90 using the following assignment:

```
X(1:5) = Y(/ 2,5,1,4,3 /)
```

Naturally, traditional scalar indexing may be used as well, should slicing or vector indexing fail to express a desired array reference.

The primary advantage of array slicing over traditional indexing is that it is a more concise means of specifying operations over arrays or subarrays, eliminating the need for explicit loops for many array operations. Like scalar indexing, slices allow for more general array interactions than ZPL's regions, yet use a notation that is more concise and optimized for the common case than that of APL's vector indexing.

The chief disadvantage of array slicing is the syntactic overhead of specifying a region-like set of indices for each array reference. Though small examples like matrix addition are not so bad, slice notation can become rather cumbersome and error-prone in larger array codes. In particular, the conformability requirements of array slices require the size and shape of each array operand to match, causing redundant information to be supplied with each array reference. Moreover, these conformability rules require more analysis than scalar indexing or region-based indexing, both of which can be satisfied by simple checks of the arguments (note that the related problem of bounds checking is common to all approaches, and is not made simpler by any scheme).

2.16.4 Forall loops

A final array access mechanism that is often supported by languages to simplify scalar indexing is the *forall* loop. This structure iterates over a multidimensional index range or an arbitrary index set, typically in an unspecified order. In this sense, forall loops represent universal quantification much like regions, in that they generate a set of indices with which to operate. Unlike regions, forall loops tend to use iterator variables like traditional for loops. These iterators are typically used to access an array's values using scalar indexing. Thus, forall loops can be considered a compact representation of a nested loop whose iteration order is unconstrained.

FIDIL (FInite DIfference Language) [SH89, HC93] is an example of an array language that uses forall loops. FIDIL was designed for use in scientific computation and supports general index sets called *domains*. Domains need neither be rectangular nor dense, and FIDIL supports computation over them using set-theoretic union, intersection, and difference operations. Domains are used both to specify the structure of arrays (*maps*), and to provide index sets for FIDIL's forall loops.

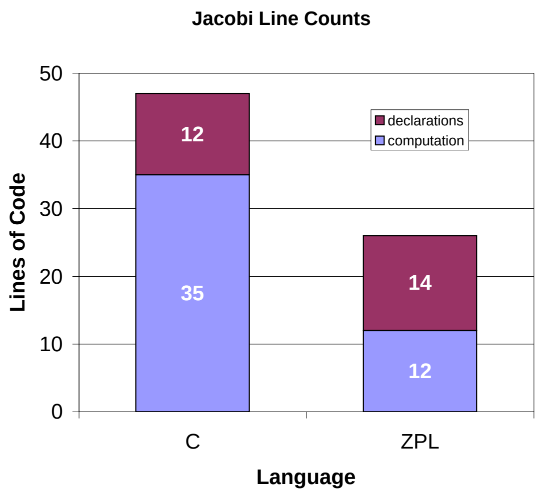
Fortran 95 [Geh96] also supports a forall loop structure that takes an array slice as its bounds and iterates over the indices that it describes. In this context, the forall loop is much more of a syntactic sugar, as there is no language-level support for index sets as in FIDIL and ZPL.

2.17 Evaluation

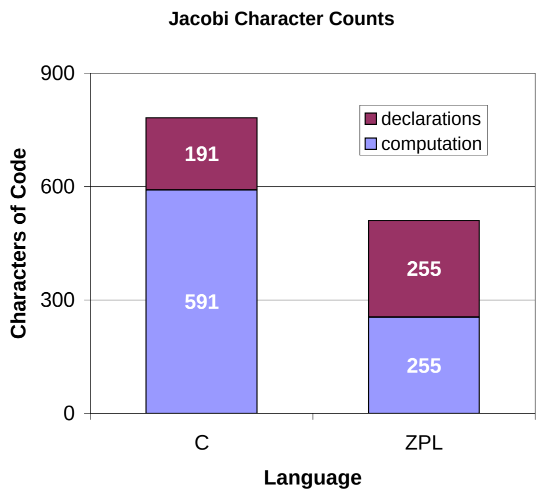
To evaluate the impact of region-based programming on a program's clarity, the sample codes of this chapter are compared to *sequential*, hand-written implementations in C. Appendix A contains the source code for these C implementations for reference.

Each line of code in the ZPL and C implementations of the benchmarks is classified as serving one of three purposes: (1) declaring a variable, procedure, or other identifier; (2) computing the benchmark's result; or (3) performing other non-essential work such as I/O, initialization, timing, *etc*. This study only considers lines in the first two categories. The graphs in Figures 2.15 and 2.16 indicate the number of useful lines and characters of code used by each implementation. Characters were counted by removing all extraneous spaces and whitespace from the codes, other than that which is required to represent the algorithms in a simple, readable form.

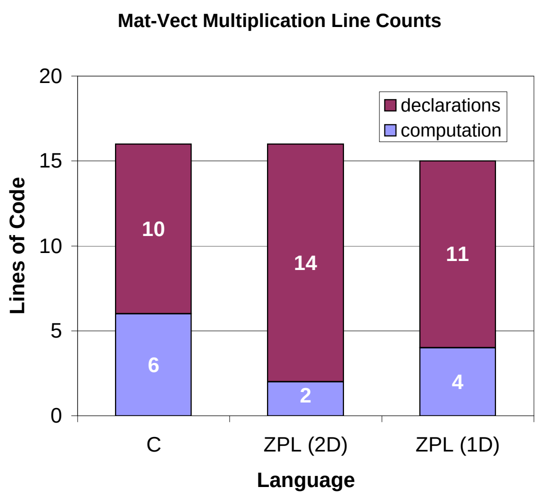
The general trend shown in these graphs is that ZPL codes express computation with a conciseness similar to C, both in terms of line- and character counts. These benchmarks also indicate that the coding effort in the ZPL versions of these benchmarks tends to be weighted more heavily towards declarations than computation. This is a result of ZPL's use of high-level named concepts like regions and directions to replace traditional loops and indexing. Presumably, the overhead of these declarations will be lessened in longer codes



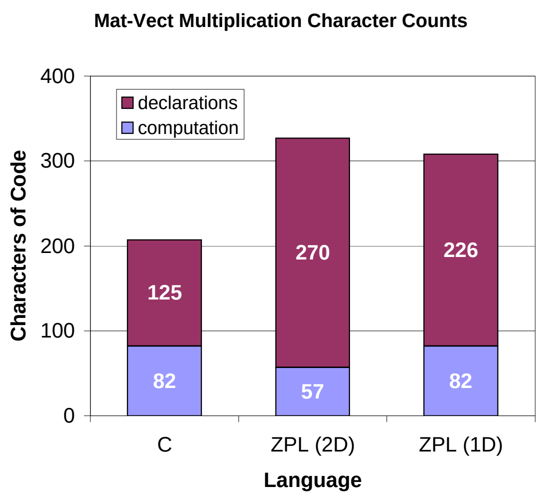
(a)



(b)



(c)



(d)

Figure 2.15: Conciseness of Sample Codes. These graphs display the number of useful lines and characters required for the sample Jacobi and matrix-vector multiplication codes in ZPL and C.

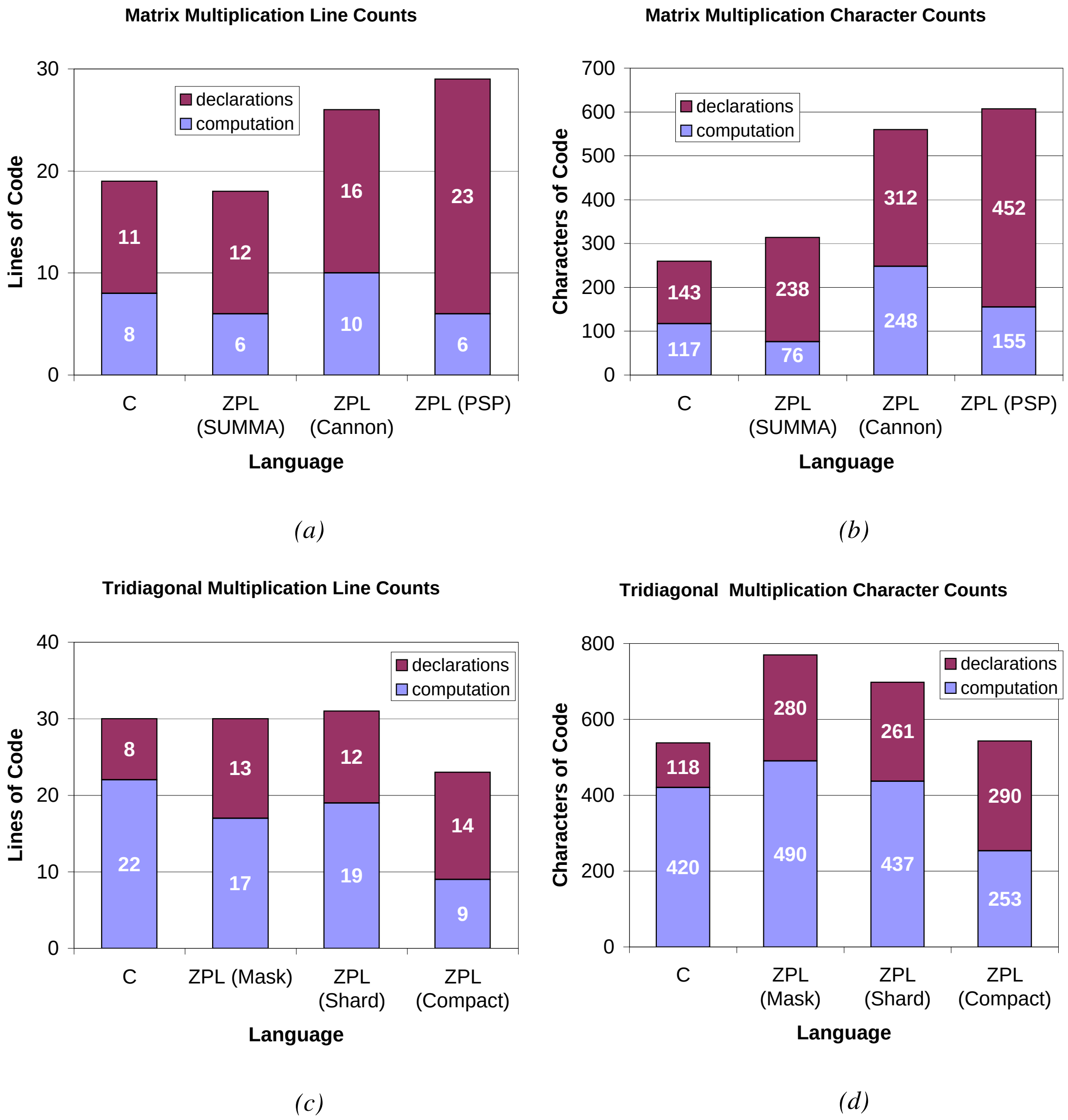


Figure 2.16: Conciseness of Sample Codes (continued). These graphs display the number of useful lines and characters required for the sample normal and tridiagonal matrix multiplication codes in ZPL and C.

where the same declarations can be used again and again, amortizing the cost of declaring them over a larger code base. The experiments in Chapters 5 and 6 support this hypothesis.

The following paragraphs give a few notes for each benchmark.

The Jacobi Iteration

The Jacobi Iteration best demonstrates the benefits of region-based programming. While most of the other benchmarks consist of a small number of array statements surrounded by a single region, the Jacobi benchmark uses a number of diverse regions to establish its boundary conditions. In the C code, each of these regions requires its own loop, demonstrating the region's concise support for array computation. Since most real-world codes will tend to require many regions/loop nests to express a computation, this benchmark best demonstrates the concise power that a few appropriate ZPL declarations can have.

Matrix-Vector Multiplication

C and ZPL represent matrix-vector multiplication using reasonably equivalent code sizes. Once again, the ZPL representations tend to be slightly more concise in terms of computation due to the use of regions rather than loops.

Matrix Multiplication

Matrix multiplication represents a worst-case for ZPL, simply due to the difference in complexity between sequential and parallel matrix multiplication algorithms. In particular, sequential C algorithms can simply iterate over the matrices and compute on them in-place. In contrast, all of the parallel algorithms described by this chapter require some amount of data movement and copying.

The SUMMA algorithm is ZPL's most concise implementation and the only one that is more compact than C's triply-nested loop. Cannon's algorithm requires significantly more computation due to its skewing operation and cyclic shifts. It also requires additional declarations to create the directions used to implement the shifting. The PSP algorithm requires the greatest amount of declarations due to its use of 2D *and* 3D regions to describe its two computational domains. The PSP computation itself is fairly concise in terms of lines, but these lines are long due to the `Index`*`i`* expressions used to implement the alignment of arrays from 2D to 3D.

Tridiagonal Matrix Multiplication

The hand-coded C implementation of tridiagonal matrix multiplication uses a compact representation of the matrices similar to the third ZPL implementation. As a result, these two codes are the most comparable in size due to their similar approach. ZPL's mask- and shard-based approaches tend to require more computation due to the overhead of restricting computation to the diagonals within the logical 2D index space. In contrast, the compact ZPL and C implementations can trivially isolate a single diagonal.

Summary

In summary, ZPL can represent these simple algorithms as concisely as C. ZPL tends to require slightly less syntax to specify computation due to its use of regions to replace looping and indexing. This effect is offset somewhat by ZPL's more verbose syntax (*e.g.*, `Index`*`i`* constants, the use of **`begin...end`** rather than curly braces). For the benchmarks studied here, ZPL tends to require more declarations than C, though it is expected that these declarations will be amortized in larger benchmarks by the savings in computation.

It is crucial to keep in mind that the ZPL implementations of this section differ from the C codes in one crucial way: they represent fully-functional parallel implementations of the benchmarks, whereas the C codes can only be run on a single processor. For this reason, regions must be considered a benefit to clarity, since they support the expression of a more complex program using syntax that tends to be as concise as sequential C. As the next chapter will show, this cannot be said for most parallel languages.

2.18 Discussion

2.18.1 Benefits of Regions

Though Section 2.16 argued that regions are somewhat less flexible and adaptable than array indexing and slicing, they are not without their benefits. This section describes the advantages that regions give programmers, syntactically and semantically.

Cleaner Elementwise Operations

Performing strict elementwise operations on arrays remains an extremely common case in array-based programming. Though interesting programs will require more complex interactions between their arrays, most large programs will still require many elementwise operations in addition to the more complex ones. In these cases, regions represent a positive evolution in array reference syntax. Array slices can be seen as a factoring of F77 loop bounds into the array references in order to optimize the common case of iterating over an array in a regular manner. In the same spirit, regions can be thought of as factoring a set of indices that describe the size and shape of slice-based array references into a single prefixing slice—the region scope.

Table 2.8 shows a number of simple array statements written in F77, F90, and ZPL. In the first row, a simple array addition is demonstrated. The F77 version requires explicit loops and repetitive array indexing. The F90 version eliminates the loops, but requires identical slices to be applied to each individual array reference. The ZPL version factors this common slice into the region scope, leaving the array references unadorned and eliminating a lot of redundant typing. Since elementwise operations constitute a common case, the result is that many array references will be unadorned in ZPL.

Table 2.8: Language Syntax Comparison

F77	F90	ZPL

`do j = 1, n`
`  do i = 1, m`
`    C(i, j) = A(i, j) + B(i, j)`
`  enddo`
`enddo`
	`C(1:m, 1:n) = A(1:m, 1:n) +`
	`              B(1:m, 1:n)`
		`[1..m, 1..n] C := A + B;`
		
		`-- or --`
		
		`[R] C := A + B;`

`do j = 1, n`
`  do i = 1, m`
`    A(i, j) = B(i, j-1) +`
`              C(i, j+1)`
`  enddo`
`enddo`
	`A(1:m, 1:n) = B(1:m, 0:n-1) +`
	`              C(1:m, 2:n+1)`
		`[1..m, 1..n] A := B@[0, -1] +`
		`                  C@[0, 1];`
		
		`-- or --`
		
		`[R] A := B@west + C@east;`

`do j = 1, n`
`  do i = 1, m`
`    A(i, j) = B(1, j)`
`  enddo`
`enddo`
	`do i = 1, m`
	`  A(i:i, 1:m) = B(1:1, 1:n)`
	`enddo`
		`[1..m, 1..n] A := >>[1, ] B;`
		
		`-- or --`
		
		`[R] A := >>[TopRow] B;`

`do j = 1, n`
`  A(1, j) = 0`
`  do i = 1, m`
`    A(1, j) = A(1, j) + B(i, j)`
`  enddo`
`enddo`
	`A(1:1, 1:n) = 0`
	`do i = 1, m`
	`  A(1:1, 1:n) = A(1:1, 1:n) +`
	`                B(i:i, 1:n)`
	`enddo`
		`[1, 1..n] A := +<<[1..m, ] B;`
		
		`-- or --`
		
		`[TopRow] A := +<<[R] B;`

`do j = 1, n`
`  do i = 1, m`
`    A(i, j) = A(B(i, j),`
`                C(i, j))`
`  enddo`
`enddo`
	`do j = 1, n`
	`  do i = 1, m`
	`    A(i, j) = A(B(i, j),`
	`                C(i, j))`
	`  enddo`
	`enddo`
		`[1..m, 1..n] A := A#[B, C];`
		
		`-- or --`
		
		`[R] A := A#[B, C];`

Clearer Array Reference Patterns

When array operations are not strictly elementwise, regions still serve a purpose by describing the size and shape of the subarray accesses. Array operators express any modifications to these base indices for a particular array expression. This has the effect of syntactically factoring the redundant part of each array reference out of the main computation, leaving only indications of how each reference differs.

As an example, consider the second row of Table 2.8, in which shifted references to B and C are summed. In F77 and F90, the array indices and slices encode redundant information. In particular, F77 specifies that each access is based on index (i, j), while F90 specifies three slices, each of which are $m \times n$ in size. In ZPL, the common aspects of these array references—the $m \times n$ base indices—are factored into the region, leaving only the differences in how the arrays are accessed, using the @ operator. By removing much of the redundant clutter, the meaning of the statement is clearer.

As further evidence, consider each column of the table one at a time to see how long it takes you to identify the operation that is being performed by each statement. Note that very different array operations end up looking rather similar in F77 and F90, whereas ZPL does a better job of distinguishing them.

Fewer Loops Required

In F77, programmers expect to use loops and indices. In F90, many array operations no longer require loops due to the availability of array slicing and vector indexing. However, as the last two entries in Table 2.8 show, these mechanisms are not strong enough to express floods, partial reductions, or remap operations without using loops. The floods and partial reductions fail due to the requirement that the two sides of an operation must have the same size and shape. This makes it illegal to add an arbitrary number of rows to a single row without a loop. The remap operator cannot be written succinctly due to the fact that F90 has no mechanism for taking the dot product of vector indices rather than the cross product.

It should be noted that F90 supports intrinsic functions such as `SUM`, `RESHAPE`, and `TRANSPOSE` that may be used to write such statements in a single line. However, these functions should be considered part of a standard library context rather than a syntax-based means for expressing array computation. Similarly, F95's forall loops allow such statements to be written on a single line, but still rely on a loop-style concept (albeit one that syntactically begins to resemble the region).

Naming Improves Readability

The fact that regions can be named greatly improves the readability of ZPL code, since it allows index sets to be given identifiers that are meaningful to the programmer. Column 3 in Table 2.8 shows each ZPL statement written both with and without identifiers. These examples demonstrate that names can improve the clarity of each statement. Note that the ability to name array slices or index ranges could improve the readability of F90 codes somewhat, but would not produce a ZPL-equivalent syntax.

Regions Promote Code Reuse

The fact that regions are dynamically scoped allows procedures to be written in a more generic way. As a simple example, note that the ZPL implementation of the SUMMA algorithm (Figure 1.2) could be moved into a procedure that takes only the size of the inner dimension as an argument and inherits the matrix size from the callsite. In contrast, a generic F90 implementation would require the bounds for each dimension to be passed in as arguments. Furthermore, even when an algorithm does require more than a single inherited region (as in the Cannon and PSP algorithms), regions represent a concise means of bundling index information for passing to another procedure.

2.18.2 Region Deficiencies

Though regions have many benefits, there are also some deficiencies that should be addressed in future region-based languages including Advanced ZPL. This section briefly describes some of them.

Regions are Rectangular

One obvious limitation of regions is their regular and rectilinear nature. Though masks and shattered control flow can be used to restrict a region to an arbitrary subset of indices, these concepts tend to require time and space proportional to the region's size in order to compute the irregularity. For example, the mask- and shattered control flow-based implementations of tridiagonal matrix multiplication require $\Theta(n^2)$ time and space rather than the $\Theta(n)$ time and space that the computation requires.

One partial solution would be to expand the region specification to allow `Index`*i* expressions in a region's index ranges. For example, a lower triangular matrix could be represented using the region `[1..n, Index1..n]`. Similarly, a tridiagonal matrix could be represented using `[1..n, (Index1 - 1)..(Index1 + 1)]`. The main challenge to such an approach is the fact that such regions cannot be specified using a cross product of sequence descriptors. Therefore, they would require a different formal specification and implementation. Legality issues would also be a concern in such a scheme.

Chapter 6 presents a different solution to this problem, in the form of sparse regions.

Inability to Capture Dynamic Regions

As currently defined, ZPL allows users to open dynamic region scopes, but not to "capture" them in a way that allows them to be reused again later. Rather, they must be explicitly redefined for each use. As a motivating example, imagine that a program dynamically calculates three subregions of an array in which it wants to perform further computation. It would be nice to have some way of snapshotting these three index sets using named regions so that they could be reused later, rather than explicitly maintaining their bounds using scalar variables.

This lack also makes it difficult to declare persistent arrays using a dynamic region. Arrays that are local to a procedure may be declared using the dynamically covering region or an explicit dynamic region, but such arrays will not persist once the procedure returns. This restricts the user's ability to declare an array over the three subregions of the previous paragraph, for example, such that they could be used throughout the program naturally.

Inability to Redefine Regions

A related problem is that named regions cannot be redefined. In a sense, ZPL's regions can be viewed as constant sets of indices that cannot be altered during the program's execution. For some applications in which a problem size cannot be known at configuration time, it would be nice to incrementally grow regions to meet a program's specific requirements dynamically. For example, while a 1D region can be used to implement an array-based list in ZPL, the list cannot be grown using standard array resizing techniques because the bounds of its defining region cannot be modified.

Regions are Inflexible

In ZPL, regions are technically neither a type, nor a first-class object. This limits their use in a number of ways: programmers may not declare collections of regions using indexed arrays or records; they may not assign regions; they may not pass regions to a procedure; *etc*. This design choice was made in order to provide the compiler with as much information about the regions as possible. The idea was to start with a restricted region definition and then broaden it as much as the compiler could tolerate without sacrificing performance or the ability to effectively parallelize a ZPL program. During the past decade, virtually no relaxation of these restrictions has taken place, though it has seemed increasingly feasible to do so.

2.18.3 Proposed Support for Regions as Values

One solution to many of the previous section's problems would be to promote the region concept to that of a first-class value. In doing so, traditional region declarations would be interpreted as declarations of constant or configuration regions. That is, the following two declarations would be considered equivalent to one another:

```
region R = [1..m, 1..n];
config var R: region = [1..m, 1..n];
```

Any regions for which ZPL's current rules are overly strict could be declared as variables of type region, allowing them to be assigned dynamically, modified, or grown. Parallel arrays declared using region variables would be reallocated after the region was assigned, preserving any values in the intersection of the old and new index sets. Procedures could be written with formal parameters of type region. Moreover, types could be created that have region components, such as indexed arrays of regions and records with region fields.

The primary liability of this scheme is that current ZPL optimizations may be compromised due to an increased amount of confusion over a region's definition. For example, in the presence of region assignments and aliasing, will the compiler be able to determine whether a region's dimension is floodable, singleton, or normal? What about its rank? It seems reasonable to be optimistic about these issues since the absence of pointers should make most of a region's salient features statically detectable using interprocedural analysis. Even in the worst-case, such an approach is worthy of more study in order to attempt to support more general region-based programming.

2.18.4 Proposed Support for User-Defined Region Operators

One feature that I believe is missing from the ZPL language is the ability for users to define their own region operators. While the region operators supported by ZPL form a useful basis set, it is not difficult to conceive of other operators that might also benefit a programmer. Rather than hoping to supply all region operators that a user could want, it

Listing 2.24: Proposed Syntax for User-Defined Region Operators

```
1 postfixregionop grow(delta: integer; var l, h, s, a: integer);
2 begin
3   if (delta < 0) then
4     l += delta;
5   else
6     h += delta;
7   end;
8 end;
9
10 direction nw = [-1,-1];
11           se = [ 1, 1];
12
13 region R = [1..m, 1..n];
14        BigR = [R grow nw grow se]
```

makes more sense to give the user the ability to define custom operators by describing the effect of a δ value on a sequence descriptor.

For example, I might define a `grow` region operator that pulls a region's corner in the specified direction without changing its stride or alignment. Listing 2.24 shows proposed syntax for such an operator. Lines 1–8 define the grow operator by indicating the delta value's effect on the four-tuple sequence descriptor (which could be expressed using a record type rather than four scalar variables). The region operator could then be used to define `BigR` as shown in line 14, rather than by explicitly specifying its bounds.

Such support seems like a useful and simple extension to ZPL as it currently stands. One important side-effect that this might have is to require parenthesization to indicate operator precedence in a region expression. ZPL's built-in region operators are associative, so parenthesization is neither required nor allowed.

2.18.5 *Implicit Storage Considered Frustrating*

One feature of ZPL that has not been described in this chapter is its support for implicit storage. The implicit storage concept causes certain specifications of an array's boundary conditions to implicitly extend the amount of memory allocated for it. For example, in the Jacobi iteration of Listing 2.15, variable `A` can be declared over region `R`, and the initialization of its borders using **`of`** regions would cause its storage to automatically be extended by a row and column in each direction.

This feature was motivated by the observation that many ZPL programs use two regions for each problem size, one for the computation space and a second that extends the computation space by a few extra rows or columns to describe the array's data space with boundaries. For example, most of the sample programs of Section 2.15 exhibit this characteristic. Therefore, it was believed that implicit storage would reduce the number of regions that programmers would have to declare, saving them some trouble.

In the long-run, it has turned out quite the opposite. Implicit storage allocation continues to be a confusing issue to most programmers due to the fact that (1) the rules are not as clear to them as they should be, (2) the rules are not always as general or intuitive as they ought to be, and/or (3) the fact that implicit storage is invisible in the program makes it hard to debug or even feel reassured that the expected behavior is going to take place. More questions and bugs have probably been addressed due to misunderstandings related to implicit storage than any other concept in ZPL. Most programmers eventually give up on the idea and simply explicitly declare the complete memory that their arrays require as I have done in this chapter's sample codes.

As a result of these experiences, I believe that implicit storage allocation is a bad idea. Users are accustomed to explicitly declaring the type of their variables, which includes the sizes of their arrays. While giving them a mechanism to handle the common case with one less identifier is an admirable idea, it has caused more confusion than it is worth. In this sense, implicit storage seems much like optional variable declarations [Mac87] and should be similarly avoided.

2.18.6 Scalar Issues

For the most part, ZPL's scalar concepts are not particularly interesting or inventive. They provide basic functionality without many surprises. This section touches on a few noteworthy characteristics.

Configuration Variables

The configuration variable is ZPL's most interesting scalar concept. It has proven extremely useful as a means of specifying a value that a programmer will want to change from run to run, but which the compiler can use as a basis for optimization. This makes programmers' lives easier by not requiring them to create and maintain a separate executable per program configuration. Yet, it provides more semantic flexibility than the `const` keyword in C, which requires its initializer to be statically specifiable. Having worked with ZPL for a number of years, I often find myself wishing that other languages had a concept that was equivalent to the configuration variable.

The Lack of Pointers

Up to this point, the lack of pointers in ZPL has kept the language clean and easy to analyze. Furthermore, the ZPL applications that have been studied to this date have not suffered due to the lack of pointers. As the language strives to support more irregular, graph-based data structures, some sort of pointer mechanism will be required. It will be an interesting challenge to see whether such a mechanism can be supported in a clean, high-level way analogous to regions, or whether such data structures necessarily require pointers and the difficulties that they entail. An interesting starting point for anyone approaching this problem would be Vassily Litvinov's exploratory work in *Graph ZPL* [Lit95].

Parameter Specifications

One key place where I find ZPL's scalar syntax lacking is in its lack of richness for describing how a procedure's parameter will be used. In particular, the by-reference **`var`** specification might mean any of the following: (1) I will be sending this value into the procedure, modifying it there, and want the resulting modification to be reflected in the actual parameter; (2) The current value of this parameter is unimportant, but I will be using this parameter as a means of returning a new value calculated within the procedure; or (3) This is a very large data structure and the compiler should not make a copy of it when passing it into the procedure, though I will not be modifying it. There are many instances during ZPL compilation in which the compiler would like to differentiate between these meanings for optimization purposes. While many cases can be differentiated by analyzing procedure definitions, aliasing can complicate matters and foil the analysis. My sense is that a better-designed set of parameter tags, perhaps similar to those provided by Ada [TD97], would not represent a hardship to the user and would support better communication between the programmer and compiler.

2.19 Summary

This chapter has defined the concept of the region and explained its use in defining ZPL. Regions represent a succinct means of describing a set of indices for use in declaring arrays and expressing array computation. This chapter argues that regions have many syntactic benefits, including the elimination of redundant indexing expressions and an emphasis on different array access styles. However, the most important benefits of regions are related to their use in parallel computing. The next chapter describes these benefits.

Chapter 3

REGIONS AND PARALLELISM

As mentioned in the previous chapter, the most important benefits of regions pertain to their role in parallel computation. This chapter addresses the parallel interpretation of regions and their impact on parallel programming. In doing so, it justifies some of ZPL's rules that seem somewhat arbitrary in the sequential context.

This chapter is organized as follows: Sections 3.1–3.5 interpret regions, arrays, and scalars in the parallel context. Section 3.6 explains how this implementation forms the basis of ZPL's syntax-based performance model. The nagging questions of the previous chapter are re-examined in light of this performance model in Section 3.7. Sections 3.8 and 3.9 apply the performance model to Chapter 2's sample codes and validate it experimentally. Section 3.10 defines a new type of region dimension for use in the parallel context. Section 3.11 contains a survey of other parallel programming languages and libraries, and Section 3.12 discusses issues related to this chapter's contributions. This chapter presents an expanded discussion of work that was published previously [CCL$^+$98, CLLS99].

3.1 Regions Imply Parallelism

The key to parallelism in ZPL is that each region's indices are distributed among the processors that execute a ZPL program. This has two implications for the parallel interpretation of ZPL programs. First, since arrays are declared using regions, the distribution of a region's indices determines the distribution of array elements between processors (hence the term "parallel array"). Second, since regions supply indices for array operations, the distribution of their indices determines the distribution of array computation between processors.

Listing 3.1: Matrix Addition in ZPL

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

var A, B, C: [R] integer;

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

As a simple example, reconsider matrix addition in ZPL as shown in Listing 3.1. Assume that this code is run on two processors. Globally, there are $m \cdot n$ indices in region R. Assume that these indices are distributed such that half of them reside on processor 0 and the other half on processor 1. Since R is used to declare A, B, and C, half of each array's elements will be allocated on each processor. More importantly, any two elements with matching indices will be allocated on the same processor. Therefore, the elementwise addition and assignment of elements from A, B, and C will be executed perfectly in parallel, with each processor doing half of the work.

Note that understanding *how* a region's indices are distributed across the processor set is crucial to understanding how a ZPL program is executed. For example, in the matrix addition code, if one processor was given $m \cdot n - 1$ of the indices and the other received the remaining index, the execution of the program would not be load balanced. Or, if A, B, and C were each declared over different regions, knowing the distribution of those regions would be crucial to understanding the communication required to bring like-indexed elements into one processor's local memory. Statements with array operators raise additional questions, since they represent more complex array interactions.

Fortunately, ZPL defines the distribution of regions in a very precise manner that makes such analysis straightforward. However, to understand ZPL's region distribution, one must first understand how ZPL views a machine's processors.

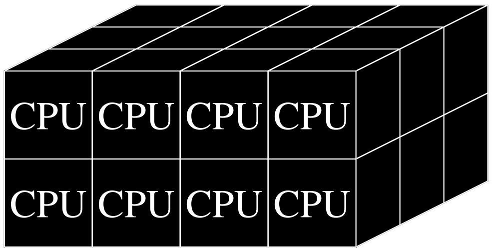


Figure 3.1: A $2 \times 4 \times 3$ Processor Grid

3.2 The Processor Grid

This section describes ZPL's model of the processor set. One outcome of using the CTA machine model is that the physical topology of a machine's processors is not as important as whether data is local to a processor or not. This matches the behavior of modern parallel machines, in which the latency for transferring data between processors tends to overshadow the latency difference in communicating with different processors. This assumption is corroborated by the evolution in parallel algorithm research over the years from "how to write algorithm x on parallel architecture y" to simply "how to write algorithm x in a portable, scalable manner."

ZPL takes advantage of the CTA's de-emphasis of network topology by representing a machine's physical processors using logical *processor grids* (or *grids* for short). A processor grid is simply an arrangement of the processor set in a rectilinear grid of arbitrary dimension, d. For example, a set of p processors could be represented using a $p_1 \times p_2 \times \ldots \times p_d$ grid, where $p_1 \cdot p_2 \cdot \ldots \cdot p_d = p$. The value of d typically corresponds to the rank of the regions being used with the processor grid. Programs that utilize regions with different ranks or different scales may use multiple processor grids to represent a different view of the processor set for each computational domain. See Figure 3.1 for a sample 3-dimensional 24-processor grid.

Users specify the number of processors and the dimensions of the processor grid(s) on the command line of their ZPL executables. Note that since ZPL is a global-view language, processor grids are not apparent in the source code of a ZPL program. This has the advantage that a correct ZPL program can be developed and debugged on a single processor and then run on a machine with multiple processors effortlessly. Typically, the only differences between single-processor and multiple-processor runs are due to precision issues stemming from the reordering of arithmetic operators in the parallel setting.

The ability to specify processor grids on the command line also allows programs to be executed on multiple processor grid configurations without recompilation. This provides a degree of convenience to programmers who have to share processors with other users, or who want to experiment with different configurations, since there is no need for recompilation or the maintenance of multiple executables. While it might be assumed that compile-time knowledge of the processor grid would allow the ZPL compiler to generate more efficient code, the fact that region bounds are typically defined using configuration variables tends to thwart these efforts. While specifying both of these factors at compile time would open up optimization opportunities for ZPL, these avenues have not been pursued, primarily due to the belief that such inflexibility is virtually intolerable to a sophisticated programmer.

3.3 Region Distribution

ZPL has two rules about how regions are distributed. They are:

Distribution Rule 1: Regions must be distributed in a *grid-aligned* manner.

Distribution Rule 2: If two regions *interact*, they must have the same distribution.

This section describes these rules in further detail.

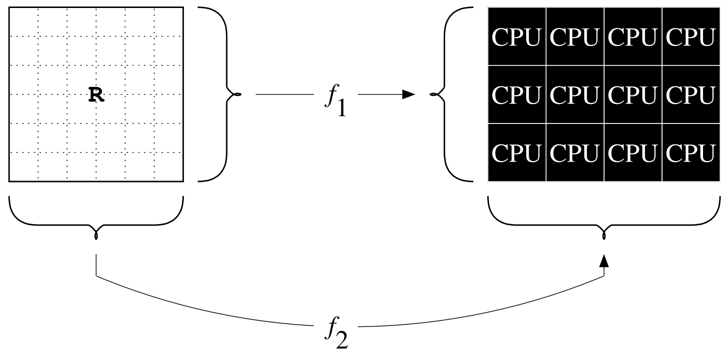


Figure 3.2: Grid-Aligned Distribution in 2D. For 2D regions and processor grids, one function (f_1) is supplied to map row indices to processor rows. A second function (f_2) maps column indices to processor columns.

3.3.1 Grid-Aligned Distribution

To understand grid-aligned distribution, consider the problem of distributing the indices of a d-dimensional region onto a d-dimensional processor grid. Such a distribution might be defined using a distribution function $f : \mathbb{I}^d \rightarrow \mathbb{N}$ that maps each index $\mathbf{i} = (i_1, i_2, \ldots, i_d)$ to a processor $p_{\mathbf{i}} \in 1 \ldots p$. Such a scheme permits the expression of arbitrary distributions of indices to processors.

Grid-aligned distribution takes a more constrained approach. In particular, it uses a vector of distribution functions, $\mathbf{f} = f_1, f_2, \ldots, f_d,\ f_i : \mathbb{I} \rightarrow \mathbb{N}$. In this representation, function f_i describes the mapping of the indices in the i^{th} dimension of a region to the i^{th} dimension of a processor grid (Figure 3.2). Thus, index $\mathbf{i}$ is mapped to the processor at location $(f_1(i_1), f_2(i_2), \ldots, f_d(i_d))$ of the processor grid. Intuitively, this means that each row of a region will be mapped to a single row of the processor grid, each column to a single column of the grid, *etc*. Note that the most common index distribution schemes—blocked, cyclic, and block-cyclic distributions—are all grid-aligned distributions.

3.3.2 Interacting Regions

Interacting regions are those that ZPL requires to be conformable (have the same rank). This includes those that are referenced by a statement explicitly—the source regions of floods and partial reductions—as well as those that are referenced implicitly—the statement's covering regions and the regions used to define the arrays it references.

For example, referring to a 2D array A within the context of a 2D region R implies that A's defining region *interacts* with R. Similarly, the source and destination regions of a flood are considered to be interacting. One interesting result of this rule is that the defining region of a remap operator's source array does not necessarily interact with any other regions in a statement. Note, however, that the map arrays *do* interact with the expression's covering region, since they are read in its context.

To summarize, consider the following ZPL statement which contains all of the array operations described in Chapter 2:

```
[RG] [R] A := B + C@dir + D@^dir2 + (>>[RF] E) +
                    (+<<[RR] F) + (+<< G) + H#[I1,I2,...];
```

Assume that R and A are the same rank and that RG and G are the same rank. ZPL's conformability requirements specify that R, A, B, C, D, E, F, and the Ii arrays must all have the same rank. Array G may have a different rank, since the full reduction evaluates to a scalar. Similarly, H may have a different rank, so long as it matches the number of map arrays in the remap operator. Therefore, R, RF, RR and the defining regions of arrays A, B, C, D, E, F, and Ii are considered to be interacting. If G has the same rank as R, then its defining region also interacts with them since G is read within the context of R. Otherwise, the defining region of G interacts with RG. This statement does not cause H's defining region to interact with anything.

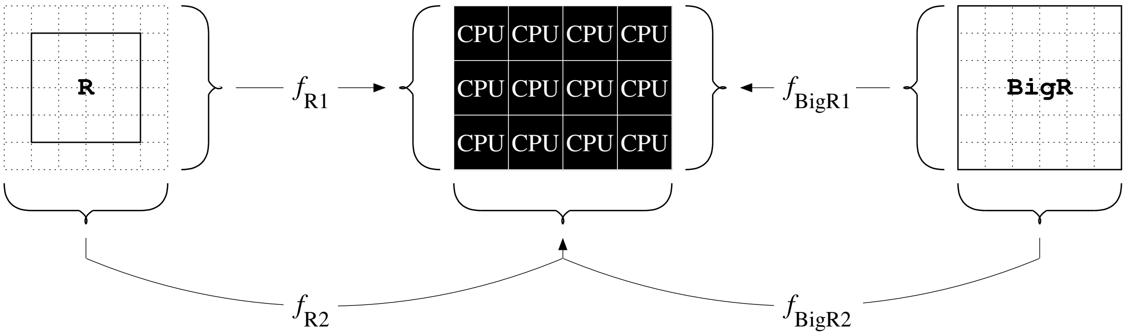


Figure 3.3: Region Interaction Constraints. Assuming that regions `R` and `BigR` interact (as they have in most sample codes in this dissertation), they must be distributed to the same processor grid using the same functions. That is, in this diagram, f_{R1} must equal f_{BigR1} and f_{R2} must equal f_{BigR2}.

Constraints on Interacting Regions

Rule 2 for region distributions states that any two regions which interact must share the same distribution. As an example, consider two regions `R` and `S`, distributed using $\mathbf{f}_R$ and $\mathbf{f}_S$, respectively. This rule imples that if `R` and `S` interact, they must view the processor set using the same processor grid. It also means that $\mathbf{f}_R$ must equal $\mathbf{f}_S$. See Figure 3.3 for an illustration.

One implication of this rule is that any index which belongs to both `R` and `S` must be distributed to the same processor for both regions. Thus, returning to the matrix addition example of Listing 3.1, even if `A`, `B`, and `C` were declared using different regions, corresponding elements of each array would still reside on the same processor since the addition of the arrays classifies their regions as interacting.

3.3.3 Region Distribution Summary

The primary effect of ZPL's two region distribution rules is that the parallel overheads of any ZPL statement can be reasoned about at the source level. This forms the basis of ZPL's performance model since it gives the programmer the ability to reason about the relative performance of different algorithms during a program's implementation. The performance model is described in more detail in Section 3.6. First, however, the following sections describe the parallel view of scalars and flood dimensions.

3.4 The Parallel Implementation of Scalars

Scalars have a simple representation in ZPL. Each scalar variable is allocated redundantly on every processor. Similarly, all scalar computations are performed redundantly on every processor. This reflects the fact that scalars are not a source of parallelism in ZPL, since they are not defined using regions.

An alternative to performing this redundant computation would be to give one processor the task of performing scalar computation and communicating its results to the others as needed. ZPL does not take this approach in order to avoid the overhead of communication that would be required to keep the processors' scalar values coherent. In addition, the redundant computation serves to keep the processors more tightly synchronized, eliminating any need to wait for the processor performing scalar computations to catch up.

3.4.1 Indexed Arrays and Records

Like basic scalar types, indexed arrays and records are allocated redundantly on each processor. Thus, the declaration of a variable of type `array [1..100] of integer` will result on 100 integers being allocated on each processor.

Special mention should be made of the interpretation of combining parallel arrays with indexed arrays or records. For example, consider the following two declarations:

```
var IofP: array [1..3] of [R] integer;
    PofI: [R] array [1..3] of integer;
```

While both variables will result in the same number of elements being allocated on a given processor, their interpretations and implementations differ. The first declaration results in

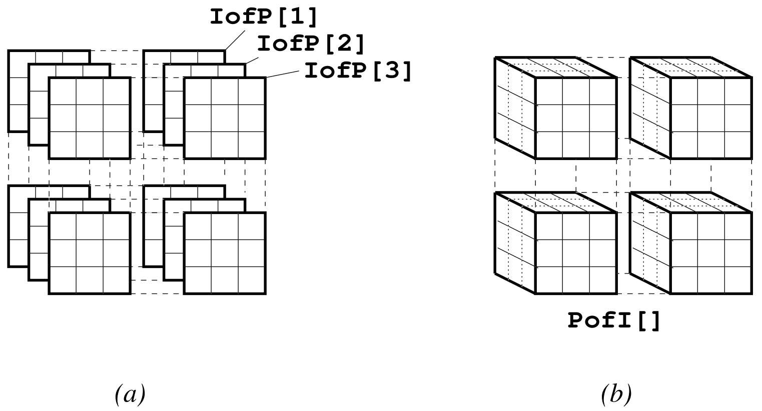


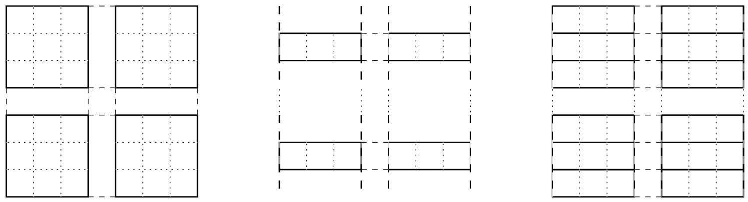
Figure 3.4: Combinations of Parallel and Indexed Arrays (shown for a 2×2 processor grid). (a) The 3-element indexed array of parallel arrays declared in Section 3.4.1. (b) The parallel array of 3-element indexed arrays.

three parallel arrays being declared, each of which stores integer elements. The second declaration results in a single parallel array, each element of which is a 3-element vector of indices. See Figure 3.4 for an illustration. Arrays of records and records of arrays have a similar interpretation. All of these data structures may be mixed to any degree so long as a parallel array never contains another parallel array as part of its element type.

3.5 The Parallel Implementation of Flood Dimensions

Flood dimensions are distributed much like regular region dimensions in that flooded rows correspond to processor grid rows, flooded columns to grid columns, *etc*. However, the distribution of flood dimensions differs due to the fact that they represent a single set of values that can conform to arbitrary indices.

As a concrete example, consider the 2D floodable row `[*, 1..n]`. Columns 1 to `n` of this region are mapped to processor columns in the traditional manner. However, the fact that the row is conformable to *any* row index causes the region to be stored on *all* rows of the processor grid rather than merely a single one. Note that the processors do not need



(a) (b) (c)

Figure 3.5: Parallel Implementation of Flood Dimensions. (a) A traditional region distributed on a 2×2 processor grid. (b) A floodable region that interacts with the traditional region of part a. Note that only a single copy of the defining indices are assigned to each processor, even though they conform to all indices (as shown conceptually in part c).

to store a copy of the flood row's values for every row index that they are assigned. A single copy of the floodable row suffices, since its values are constrained to be the same everywhere.

Figure 3.5a shows the distribution of a traditional region `R` to a 2D processor grid. Note that each processor owns 3 rows and 3 columns of `R`. Figure 3.5b shows the distribution of a floodable row `F` that interacts with `R` and must therefore be distributed in the same manner. Since `F`'s row must conform to all rows of `R`, both processor rows are used to store `F`. However, since `F` represents a single row of values, each processor stores 1 row and 3 columns for it rather than an explicitly replicated 3×3 block as shown in Figure 3.5c. The same holds true for arrays declared over these regions: in this example, an array declared using `R` would result in 9 elements being allocated per processor whereas a region declared over `F` would cause each processor to allocate 3 elements.

This parallel representation is the cause for some of the flood dimension rules in Chapter 2 which may have seemed strange at the time. In order to fully understand the justification, however, one must first understand the WYSIWYG nature of ZPL's performance model.

3.6 *ZPL's Performance Model*

ZPL's performance model has three aspects: communication, concurrency, and scalar performance. This section addresses each of these topics in turn.

3.6.1 Communication

The most important result of ZPL's region distribution rules is that a program's communication is easily visible in the language's syntax. In particular, every use of an array operator potentially requires communication of a specific type. Furthermore, code that does not use array operators will never require communication. This ability to determine a program's communication requirements simply by examining its array operators is referred to as *ZPL's WYSIWYG performance model (What You See Is What You Get)*, since the communication required by each expression is clearly reflected in its syntax.

Elementwise Operations

Any array statement that lacks array operators must be composed purely of elementwise operations—promoted scalar operators and promoted scalar functions. Recall that ZPL requires the operands of a promoted scalar operator or function to be equal in rank. This implies that the regions defining the array operands are interacting, which means that elements with matching indices will be assigned to the same processor. Thus, ZPL's region distribution rules imply that elementwise operations will never require communication.

Reversing this argument, all array operators are used to refer to array elements with different indices. Thus, the use of any array operator may require interprocessor communication to bring data values with different indices to a single processor's memory. The following paragraphs consider each array operator in turn.

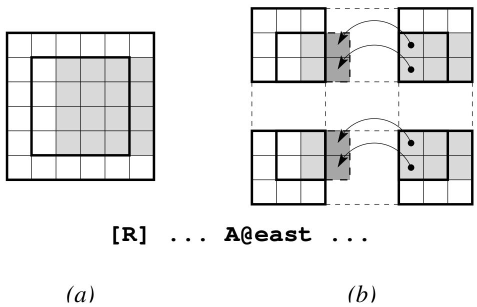


Figure 3.6: The @ Operator in Parallel. (a) The global view of an @-reference. The shaded area will be read for the indices in the center outlined area. (b) The local view of an @-reference. The processors on the left are missing a column of values, so these are sent from the processors on the right using point-to-point communication.

The @- and Wrap-@ Operators

ZPL's @- and wrap-@ operators translate an array's references by a constant offset, relative to the enclosing region scope. This typically causes processors to refer to non-local array values. Due to the one-to-one correspondence between the covering region's indices and the accessed values, point-to-point communication can be used to bring any remote values into local memory. Note that for small direction vectors using the standard distribution schemes, this will typically require communication with a processor's nearest neighbors in the processor grid. Figure 3.6 illustrates the parallel implementation of an @-reference using a blocked distribution.

The Flood Operator

The flood operator replicates a subarray of values across one or more region dimensions. Since grid-aligned distributions map region dimensions to corresponding processor grid dimensions, this implies that the flood operator can be implemented using a broadcast along one or more dimensions of the processor grid. In particular, the processors owning the source region's indices will broadcast their values along all grid dimensions in which repli-

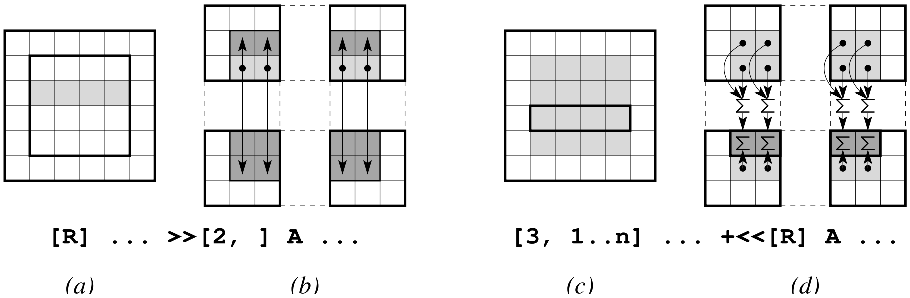


Figure 3.7: The Flood and Reduce Operators in Parallel. (a) The global view of a flood operator. The shaded row will be read for all rows within the center outline. (b) The local view of a flood operator. The lightly shaded values will be broadcast to the processor's column to be read for the darkly shaded indices. (c) The global view of a reduction. The shaded values are those that will be read and reduced to the outlined row. (d) The local view of a reduction. Reductions will be performed along processor columns, summing values and storing the result in the darkly shaded row.

cation is required. See Figure 3.7a–b for an illustration of a flood operator as it would be implemented using a blocked distribution.

The Reduction Operator

Reductions are the dual of floods, collapsing an array of values along one or more dimensions to the destination region. Thus, reductions can be implemented by reducing values along the processor grid dimensions that correspond to the array dimensions being reduced. Figure 3.7c–d illustrates the parallel implementation of a partial reduction.

Certain reductions also require a broadcast along one or more grid dimensions to ensure that the result of the reduction ends up on the appropriate processors. In particular, full reductions require the resulting scalar value to be broadcast to all processors so that they can update their local copy of the scalar. Similarly, partial reductions into flood dimensions require broadcasts for those dimensions of the processor grid. Conversely, reductions to a singleton dimension require no broadcast, since only the processor slice owning that index will require the resulting values.

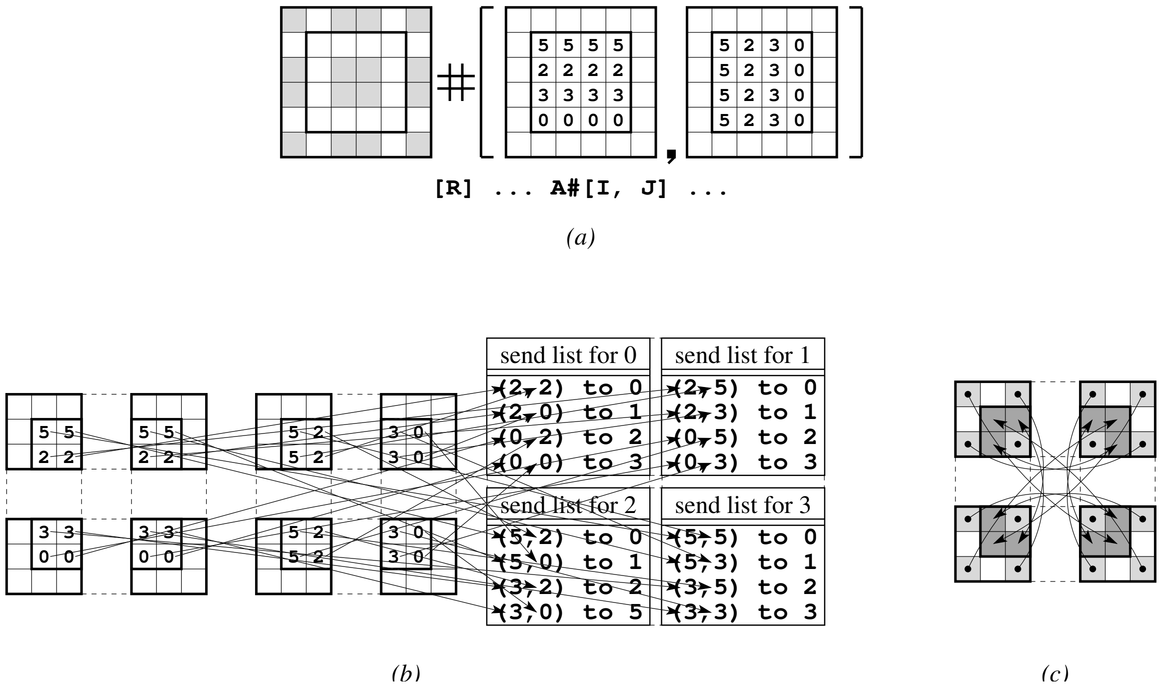


Figure 3.8: The Remap Operator in Parallel. (a) The global view of a remap operator. The shaded values are ones that will be read for the indices within the outline (given the map arrays as shown). (b) The communication required to let every processor know which elements it needs to send where. This requires an all-to-all communication step in general. (c) The communication required to actually move the values from their sources (lightly shaded) to their destinations (dark shading). In general, this step requires a second all-to-all communication.

Table 3.1: Summary of Array Operator Communication Styles

Operator	Effect	Sample	Communication Style
elementwise	operate on corresponding values	`A + B`	none
@	translate array references	`A@dir`	point-to-point
flood	replicate array values	`>>[i,] A`	sub-grid broadcast
reduction	collapse array values	`+<<[R] A`	sub-grid reduction (+ possible broadcast)
remap	arbitrarily access array values	`A#[I, J]`	all-to-all

The Remap Operator

The remap operator allows programmers to randomly access an array. Since the map arrays are aligned with the destination region, this means that communication is required both to tell other processors what values they need to communicate, and to transfer the values themselves. In the general case, each step requires all-to-all communication to implement, since the values required by a processor may be spread across the entire processor grid. Figure 3.8 shows the parallel implementation of a remap operator.

Summary

Table 3.1 summarizes each array operator, describing its effect, giving a sample expression, and summarizing the communication style that it requires. Note that the order in which the operators have been described throughout this dissertation corresponds roughly to their expected cost. While all of the operators scale proportionally to the number of elements that they reference, their communication overheads differ based on the number of processors involved. In particular, elementwise operations require no communication while the @ operator requires point-to-point communication which has $O(1)$ communication steps. The flood operator requires sub-grid broadcasts, which tend to require $O(\log p)$ communication steps, where p is the number of processors involved. The reduction operator also tends to be $O(\log p)$, and has the additional overhead of applying the reduction function to the values being combined. Reductions may also have to broadcast the final result for an additional $O(\log p)$ cost. Finally, the remap operator requires all-to-all communication which results in $O(p)$ communications per processor. For many applications, trading off an instance of one of the more expensive operators for a handful of the less expensive ones can be worthwhile.

Table 3.1 also justifies the existence of multiple array operators rather than simply using the remap operator to implement all array references: namely, there is a one-to-one correspondence between array operators and common communication paradigms. This allows users not only to detect whether or not their programs require communication, but also to classify the types of communication that it requires. If the remap operator was used to express all non-elementwise computations, users could not be assured of the cost of the implementing communication, nor would they have any visual cues to help discern between the different styles.

This correspondence between operators and communication styles also aids compilers in implementing a ZPL program because they do not have to analyze array subscripts, slices, or vectors of indices to determine the style of communication required by a program. While some classification is still needed, such as determining the grid dimensions in which a broadcast is required, these tasks are comparatively simple. The result is that compiler implementors can spend less effort classifying communication and more time implementing it efficiently and optimizing it.

3.6.2 Concurrency

The concurrency of a ZPL program is fairly easy to determine, though not quite as visible in the syntax as its communication. Each operator applied to an array reference implies some amount of parallel computation that is required, and its enclosing region scope indicates the

Listing 3.2: Four Matrix Additions with Differing Amounts of Concurrency

```
-- (a) add a value at a time              (b) add a row at a time
   for i := 1 to m do                     for i := 1 to m do
     for j := 1 to n do                     [i, 1..n] C := A + B;
       [i, j] C := A + B;                 end;
     end;
   end;

-- (c) add a column at a time             (d) add whole arrays at once
   for i := 1 to n do                     [R] C := A + B;
     [1..m, i] C := A + B;
   end;
```

available concurrency. How well or poorly that computation is balanced depends on how the indices of the region are mapped to the processors. Since the processors are not visible within a program's text, neither is the concurrency. What *is* visible are the region scopes themselves. Thus, as long as programmers ensure that the regions they use are distributed across the processors as evenly as possible, they can trust that the available concurrency is load-balanced.

As a trivial example, consider the four code excerpts in Listing 3.2. Each of these performs the matrix addition of Listing 3.1 in a different way. For example, the first excerpt shows how a naive user who is accustomed to scalar indexing might write matrix addition in ZPL, adding a pair of elements at a time using a dynamic region that describes a single index. The second two variations use regions that describe a row or column of indices at a time. The final variation uses a region to describe the entire index space.

Of these four solutions, the last is the best. Since its region describes the largest number of indices and the codes are otherwise equivalent, it maximizes the amount of concurrency available to the compiler. Assuming that `R`'s indices are distributed across all processors in a load-balanced manner, this code will achieve maximal parallelism. On the other end of the spectrum, the first code sample is the worst, since its regions merely describe a single index at a time. Since each index will be assigned to a single processor, there is no concurrency in the matrix addition statement. Moreover, loops are not a source of parallelism in ZPL, so there is no basis for hoping that different iterations of the loop will execute concurrently.

The second and third implementations have more concurrency than the first, since each region describes a number of indices. Whether or not this will result in maximal concurrency depends on the processor grid being used. For example, each region in the second version would result in concurrent execution when run on a $1 \times p$ processor grid, but would be completely sequential on a $p \times 1$ grid, since grid-aligned distribution implies that index rows are mapped to processor rows. The dual would be true for the third code sample.

As a final performance note, the first three code snippets also have the overhead of creating a new dynamic region on each iteration, which represents a non-negligible amount of overhead. For this reason, even when either of the second two codes achieve full concurrency, they would still be expected to execute more slowly than the last one.

Although this example is an extremely trivial one, its lesson is crucial: programmers should make regions as big as possible in order to maximize the available concurrency in an array computation. When programmers find that part of their program uses a number of small regions, they should rethink that part of the algorithm to determine if a different solution would admit more concurrency. This example also shows that programmers should pay close attention to their processor grids' dimensions and to how regions are distributed to the grids, since both of these factors play a role in concurrency.

3.6.3 Scalar Performance

Though ZPL is an array language, it is typically implemented on scalar machines. This implies that ZPL compilers must implement the language by translating array operations into scalar loops and scalar operations. As a result, many traditional principles for efficiently using sequential languages also apply to ZPL. For example: using large amounts of memory can be expensive; paying attention to the memory order of an array's elements can be useful (ZPL uses row-major order); function calls have a certain amount of overhead associated with them; floating point operations are more expensive than integer operations; *etc.*

There is no magic to evaluating a ZPL program's scalar performance, other than to keep in mind that there is also no magic used to implement ZPL programs. Thus, the savvy ZPL programmer will be sure to pay attention to traditional scalar performance issues in addition to the parallel issues of communication and concurrency.

3.6.4 Performance Model Summary

It should be noted that ZPL's performance model is not a means for computing the absolute running time of a program as a function of its problem size and processors. Rather, it gives programmers a means for understanding how a program will be implemented so that they can decide between different algorithmic choices.

In many ways, ZPL's performance model is similar to the cues that are provided by scalar languages like C. Returning to the matrix addition example of Listing 1.4, C does not indicate what the running time of either loop ordering will be, or even that one will necessarily be slower than the other. However, it does give the programmer a basis for deciding between the two. Most good C programmers do not know the precise sequence of assembly instructions that will be generated for their code, yet they do have a general sense of the instructions and overheads that their code requires and can make informed decisions as a result. For example, C programmers know that using an iterative approach will tend to require less space and time than a recursive one.

In an analogous manner, ZPL strives to give parallel programmers a sense of how their arrays will be distributed and what communication their code will require at the source level. They may not know precisely how the communication will be implemented or where it will take place, but they will have a general sense of their program's communication requirements and available concurrency. The hope is that this information can be used by programmers to make informed implementation choices without understanding the compiler or delving into its implementation of their code. Section 3.8 uses ZPL's performance model to evaluate the sample programs of Section 2.15. First, however, the following section reconsiders some of the nagging questions from the previous chapter.

Listing 3.3: (Illegal) ZPL Assignment of a Vector to an Array

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

var A1: [R1] integer;
    A2: [R2] integer;

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

3.7 *Open Questions Reconsidered*

Now that ZPL's performance model has been explained, the nagging questions of Section 2.14 should be less confusing. This section considers those that have not already been addressed.

Why ZPL Prevents Interactions Between Regions and Arrays of Different Rank

The reason for this is that such interactions would break the WYSIWYG model of communication evaluation. For example, consider the code in Listing 3.3. One could imagine that the assignments of `A1` to `A2` might be considered legal, since both sides of the assignment describe an n-ary vector of values. However, when `R2` is distributed across the processor grid, one of the two assignments would have to result in communication since `A1` cannot be aligned with both row 1 and column 1 of `A2`. Since ZPL's performance model dictates that communication must be visible to programmers in their programs' syntax, such assignments are not legal. To perform such an assignment, the 1-dimensional vector would have to be explicitly aligned with one of the array's dimensions, either using the remap operator or by declaring it using a 2D region with a flood or singleton dimension.

Why Regions Cannot be Applied to Arbitrary Expressions

The rationale for this is similar to that for the previous question. As an example, consider the following statement which tries to assign the first row of A2 to its first column:

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

Grid-aligned distributions dictate that elements in the rows and columns of A2 will not be aligned on a processor grid. Thus, communication would be required to perform this assignment. Since there is no communication visible in the assignment, applying regions to arbitrary expressions has been made illegal. This decision also has the benefit of eliminating a number of subtle conformability questions such as, "is a $1 \times 2n$ region conformable with a $2 \times n$ region?"

Why Flood Dimensions are Non-Conformable with Singleton Dimensions

Once again, the rationale is due to the WYSIWYG performance model. Reconsider the illegal assignments from Section 2.7:

```
[FloodRow]  F := B;
[1, 0..n+1] F := B;
```

Since F's implementation requires multiple copies of its values to be stored on multiple processors, assignments of this style would either require communication to broadcast a single row of B's values to all processor rows, or they would result in different copies of F on different processors. The latter interpretation is untenable due to the fact that F only represents a single set of values. But the former interpretation violates the WYSIWYG property by once again hiding communication from the programmer. This is the reason for strict rules about reading or writing flood arrays. It also explains why the flood operator represents a legal means of assigning to a flood array, since that operator is implemented using broadcasts.

In a similar vein, note that it is only legal to assign scalar expressions to scalar variables. This once again ensures that all communication required to update each processor's copy of the scalar is visible in the code. This is the most important reason that ZPL chooses to replicate scalars and compute on them redundantly.

Why @ References Cannot be Passed by Reference to Parallel Procedures

Any time an array with an @-reference is read or written, point-to-point communication may be required to move values from one processor's memory to another. Since modifications to a formal array parameter must be reflected in the actual parameter, this implies that any writes to such a parameter would require communication if its actual parameter contained an @-reference. In order to keep each processor's view of the array up-to-date, this communication would have to be performed within the procedure body in case another array reference aliased the array in question. However, this would require communication to take place within the function without an @-reference to alert the programmer.

3.8 Analysis of Sample Programs

This section reconsiders the sample programs of Chapter 2.15, analyzing their parallel implementations using ZPL's performance model.

3.8.1 The Jacobi Iteration

The Jacobi iteration of Listing 2.15 uses the @ operator to implement its 5-point stencil and the max-reduction operator to test for convergence. These operators imply that each iteration of the main loop will require point-to-point communication to read non-local values in the four cardinal directions as well as a reduction and broadcast across all processors to calculate `delta`.

Ignoring the initialization code, the entire Jacobi computation contains a single region scope, `R`. This implies that distributing `R`'s indices across all processors in an even manner should result in maximal concurrency for the program. Using a 2D block distribution should minimize the number of border elements that need to be transferred by the @ operators due to surface-to-volume arguments. Moreover, the reduction and broadcast are unlikely to be affected by the processor grid's dimensions since they involve all processors.

The initialization code writes to the boundaries of `A` using four **`of`** regions. While these will not result in much concurrency for a 2D block distribution, the fact that they are only executed once and require $\Theta(n)$ work implies that the suboptimal concurrency should be negligible as compared to a few iterations of the $\Theta(n^2)$ main loop.

As far as scalar concerns go, the division by 4.0 could be changed to a multiplication by 0.25 to turn a floating-point division into a multiplication. As a somewhat more subtle issue, the fact that ZPL allocates its data in row-major order implies that the values communicated for the east and west @-references will be strided in memory. In contrast, references to the north and south refer to values that are adjacent in memory. Thus, if the processor set cannot be organized in a perfect square, specifying a rectangular grid with more rows than columns would be preferable in order to minimize the number of strided references that each processor has to communicate.

3.8.2 Matrix-Vector Multiplication

A useful application of ZPL's performance model is to predict the better of two algorithms for a problem. This section does so for the two matrix-vector multiplication algorithms presented in Section 2.15.2.

The array operators used in the 2D vector implementation (Listing 2.16) are a row flood and a column reduction. These operators should scale logarithmically with the number of processor columns and rows, respectively. If the result vector must be transposed at the end, there is also a use of the remap operator, which could involve all-to-all communication.

The 1D implementation (Listing 2.17) uses two remap operators and a column reduction. Thus, the difference between the communication requirements of the two programs is roughly a flood versus a remap. Given the overhead of the remap operator, programmers should expect the 2D implementation to spend less time in communication. The use of the constant map expressions in the 1D implementation's extra remap operator provides some hope that the compiler could optimize it to perform competitively with a flood operator. However, the wise programmer will probably choose the guaranteed communication schedule that the first implementation provides rather than hoping for a particular compiler optimization to fire.

In terms of concurrency, both programs perform the bulk of the computation—the floating point multiplies and adds—over the same regions, so they are identical in this respect. The only real difference in the two algorithms is how data values are moved from their 1D representations to interact with the 2D matrix.

Considering the scalar factors of each program, memory-conscious programmers will note that the 1D implementation requires two $m \times n$ arrays to store data, as well as m- and n-element vectors. In contrast, the 2D version uses a single $m \times n$ array, similar m- and n-element vectors, and a flooded array that will require $p_1 \times n$ elements, globally. The more modest memory requirements of the 2D implementation should result in better performance.

As an additional memory-related consideration, note that the remap of `V` on line 17 of the 1D code may require the use of a 2D temporary array to store the result of the remap. While it is possible for the compiler to use flooded storage to hold the result of the remap, this once again relies on a particular compiler optimization rather than a guarantee from the performance model. While the floodable storage could be explicitly introduced and used by the programmer, it would break the purity of the 1D implementation.

Due to its communication and memory requirements, the 2D version appears to be preferable. By explicitly specifying how the 1D vectors should be aligned with the 2D array, the user maintains tight control over the execution and avoids relying on compiler optimizations for good performance.

It should be noted that there may be cases when the 1D version is useful. In particular, if the multiplication is part of a larger program that spends the bulk of its time computing on 1D vectors, storing them as actual 1D arrays could be worthwhile, as it would allow them to be mapped to the processor grid independently of the 2D array. However, note that for such a program the 2D version could be run on a $1 \times p$ processor grid in order to maximize the parallelism of the row vector operations. This would have the effect of increasing the number of elements that line 26 would have to reduce, but it might be an agreeable tradeoff if the 1D computation dominates the cost of the program.

3.8.3 Matrix Multiplication

In comparing the matrix multiplication algorithms of Section 2.15.3, we begin by focusing on the 2D algorithms.

The SUMMA algorithm (Listing 2.18) consists of n iterations, each of which uses a row flood and a column flood. Cannon's algorithm (Listing 2.19) also consists of n iterations, each of which uses two wrap-@ operators. It also uses the remap operator during initialization. This represents a situation where ZPL's WYSIWYG performance model indicates *what* communication the programs require, yet does not make it clear *which* approach is better. Wrap-@s tend to be cheaper than floods, so Cannon's inner loop should have lower communication costs. Yet, remap operators typically have the highest cost. Are the two uses of the remap operator costly enough that they could offset the efficiency of the wrap-@s? Unfortunately, the answer to this question probably varies with the choice of problem size, compiler, and parallel machine.

Turning to the concurrency of the algorithms, the bulk of the work is the same for both programs: $m \times o$ multiplications and adds are performed per iteration, as described by region `RC`. However, note that Cannon's algorithm also requires $m \times n$ assignments per iteration to shift `ASkew` and $n \times o$ assignments to shift `BSkew`. In comparison, the SUMMA algorithm's flood operators should require at most $p_1 \times o$ and $m \times p_2$ assignments to implement, since they result in a floodable expression. Thus, while the concurrency of the floating point operations in both algorithms is similar, the SUMMA algorithm requires fewer assignments per iteration, assuming $p_1 \ll m$ and $p_2 \ll n$.

This observation also reveals an important scalar performance difference in these algorithms. Although the number of multiplies and adds per iteration is the same for both

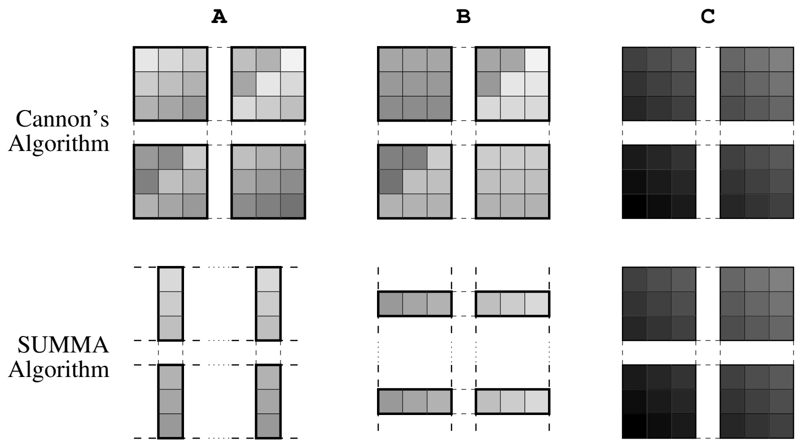


Figure 3.9: 2D Matrix Multiplication Memory Accesses. This illustration shows the approximate amount of memory touched per processor in each iteration of Cannon's Algorithm and the SUMMA algorithm. The SUMMA algorithm has a smaller memory footprint due to the fact that it refers to a single column of `A` and row of `B` in each iteration using a flood. In contrast, Cannon's algorithm has to read all of `A` and `B`'s elements to perform shifts and its elementwise multiplications. Both algorithms touch all elements of `C`, so there is no difference there.

codes, the number of values accessed to perform the multiplications differs greatly. In particular, the SUMMA algorithm reads flooded rows of `A` and `B` to perform the multiplications while Cannon's algorithm reads $m \times o$ unique elements from the `ASkew` and `BSkew` arrays. Figure 3.9 depicts these memory requirements visually. The asymptotic difference in the amount of memory accessed should translate into better scalar performance for the SUMMA algorithm.

Thus, while ZPL's performance model does not make it obvious which algorithm will have lower communication costs, the SUMMA algorithm is likely to end up being superior given the smaller working set of each iteration.

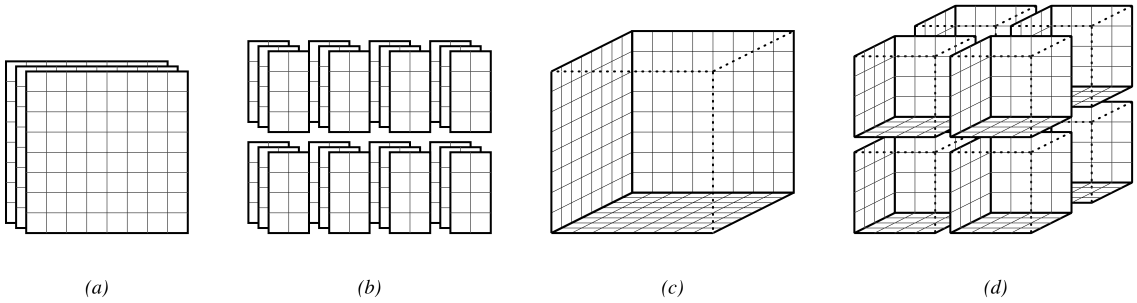


Figure 3.10: Matrix Multiplication Memory Requirements. (a) The three 2D arrays that are required for the SUMMA and Cannon Algorithms. (b) The arrays partitioned across 8 processors, resulting in 24 elements per processor. (c) The three 2D planes in a 3D space required by the PSP matrix multiplication algorithm. (d) The arrays partitioned across 8 processors. Note that the flood dimensions of the arrays result in replication and that each processor must allocate 48 elements. This is a ratio of $1 : 2$ between the two algorithms, which matches the $p^{2/3} : p$ ratio described by Section 3.8.3's analysis.

The PSP Algorithm

Unfortunately, the PSP algorithm (Listing 2.20) is not much easier to compare. To its advantage, the fact that it has no inner loop means that there are only a constant number of communications in the program. However, its communications are induced by three remap operators and a partial reduction—not the cheapest operators in ZPL. This makes it difficult to compare the communication overhead of the PSP algorithm with the other algorithms.

PSP does contain greater concurrency than the other algorithms, since it uses a single region to simultaneously represent all of its multiplications and additions. This gives the implementation a larger piece of work to divide between the processors and avoids the synchronization induced by the per-iteration communication of the 2D algorithms. Thus, PSP's concurrency is preferable.

On the other hand, the PSP algorithm requires more memory per processor due to the fact that the problem space being distributed is 3D rather than 2D. The implementation avoids the unacceptable possibility that a $m/p_1 \times n/p_2 \times o/p_3$ cube of memory will be allocated per processor by using floodable arrays. However, simply allocating the faces of the cube will require more memory per processor than in the 2D case. For example, assuming that $m = n = o$ and that the processor grid for each algorithm is as square as possible, the 2D algorithms requires $(n/\sqrt{p})^2 = n^2/p$ elements per array per processor, whereas the PSP algorithm requires $(n/\sqrt[3]{p})^2 = n^2/p^{2/3}$ elements per array per processor. Figure 3.10 illustrates this difference.

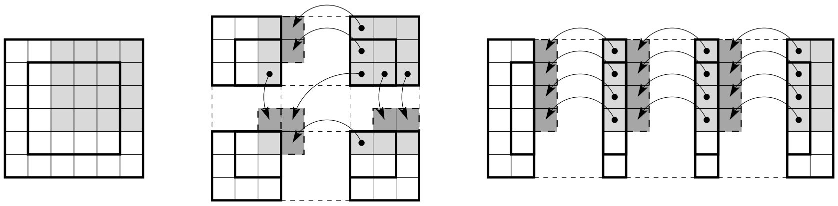
As a result, PSP represents an interesting tradeoff. It requires more memory per processor to run, but if that memory is available, it maximizes concurrency, minimizes synchronization, and uses a constant amount of (expensive) communication. Will this pay off? Without sufficient experience, programmers cannot be sure. However, at the very least, they do have a clear picture of how each implementation will execute in parallel, and the tradeoffs involved.

3.8.4 Tridiagonal Matrix Multiplication

Fortunately, the tridiagonal matrix multiplication programs of Section 2.15.4 are somewhat easier to evaluate. To achieve load balancing and ensure that each processor owns part of the diagonal matrices, it makes sense to run the programs on a $p \times 1$ processor grid (or a $1 \times p$ grid for the non-compact implementations).

Each of the three algorithms contains twelve @-references. Thus, each has approximately the same communication requirements. The diagonal references in the compact version could require communication with more processors to get boundary values in both dimensions (see Figure 3.11). However, when using a $p \times 1$ grid, no communication is required for the components to the east and west. Thus, the three algorithms should behave similarly in terms of communication.

In terms of concurrency, the shattered control flow implementation (Listing 2.22) is ideal since it performs the entire computation within a single shattered conditional enclosed by region `R`. However, to its disadvantage, its explicit representation of the matrices requires $\Theta(n^2)$ work to update array `C`.



`[R] ... A@ne ...`

(a) (b) (c)

Figure 3.11: Communication Required by Diagonal @ References. (a) A global view of a diagonal @-reference. (b) The communication required to fulfill this reference on a 2×2 processor grid. Note that the processor at the lower left needs to communicate with three processors rather than the one required by cardinal directions (Figure 3.6). (c) If one of the grid dimensions is flattened, each processor only needs to communicate with a single processor, though size of each communication is greater.

The compact version (Listing 2.23) improves upon this by only performing $\Theta(n)$ work, but requires five different regions to do so. Although the user cannot expect these five statements to execute concurrently, running the program on a $p \times 1$ grid will cause each of the five regions to span all of the processors, maximizing concurrency for each statement. Although some amount of overhead is required to set up the regions, the asymptotic savings in work should ensure that this version outperforms the previous one. A better implementation would transpose the array to produce long rows of consecutive values and avoid operating on array columns.

The mask-based implementation (Listing 2.21) also has good concurrency for its diagonals, since they will span all processors when using a $p \times 1$ grid. However, the fact that it completely rewrites and reads the mask for each diagonal results in far more work than either of the previous two implementations. As a result, it cannot be expected to compete with either of them in terms of overall performance.

Thus, ZPL's performance model indicates that the compact version should be fastest due to its $\Theta(n)$ implementation that maximizes concurrency for each region. As mentioned in the previous chapter, this implementation does make it difficult to transparently operate on tridiagonal matrices in conjunction with traditional $n \times n$ matrices. For such applications, the shattered control flow version might be preferable since its representation matches that of a traditional dense matrix. Chapter 6 reconsiders this issue by trying to achieve the best of both worlds: $\Theta(n)$ computation using a representation that conforms to traditional 2D matrices.

3.9 Evaluation

This section evaluates ZPL's performance model by running the sample codes against their sequential C implementations and computing the resulting speedups. The problem size for each benchmark was chosen by repeatedly doubling a base problem size of 10 until it exceeded the memory capacity of a single processor or required an excessive amount of time to compute. This problem size was then run on larger processor sets, doubling the number of processors each time. Speedups are computed by comparing the best parallel time over a number of runs to the best single-processor time. Figures 3.12–3.15 show the resulting speedups. Each graph indicates the base time used to compute its speedups in its y-axis label. The raw timings can be found in Appendix D.

The Jacobi Iteration

Figure 3.12a shows the performance of the Jacobi iteration written in both ZPL and C. As expected, the C implementation is fastest on one processor, due primarily to its lack of communication calls. A second difference is that its loops are fused more aggressively than those generated for the ZPL program. In particular, the ZPL compiler fails to fuse the loop that implements the reduction with those of its neighboring array statements, which would serve to increase locality and eliminate loop overheads. ZPL's current implementation of loop fusion focuses on fusing loops that enable array contractions [LLS98]. Since neither of the arrays in the Jacobi iteration can be contracted to a scalar, it fails to fuse the m-loops resulting in slightly worse cache utilization.

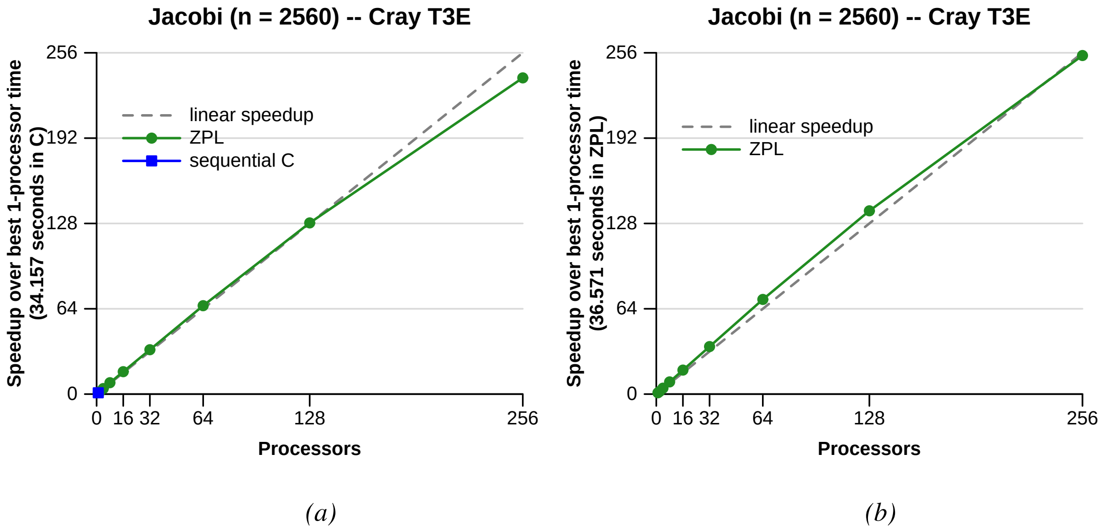


Figure 3.12: Performance of the Jacobi Iteration. (a) The speedup of the ZPL implementation of the Jacobi Iteration as compared to a hand-coded C version. (b) The speedup of the ZPL implementation against itself.

Nevertheless, the ZPL code scales superlinearly for most of the processor sets, due to its improved use of the cache as the working set per processor decreases. As the ratio of communication to computation increases, the communication overheads gradually bring the speedup back to being sub-linear. Figure 3.12b shows how the ZPL version scales against its own single-processor time, demonstrating nearly linear speedup for all processor sets.

Matrix-Vector Multiplication

Figure 3.13a shows that ZPL's matrix-vector implementations do not scale quite as ideally, due to the more expensive forms of communication that they use. Once again the C version is faster than the sequential ZPL version, this time by a factor of three. This difference is not surprising given the simplicity of the sequential code and its complete lack of data copying.

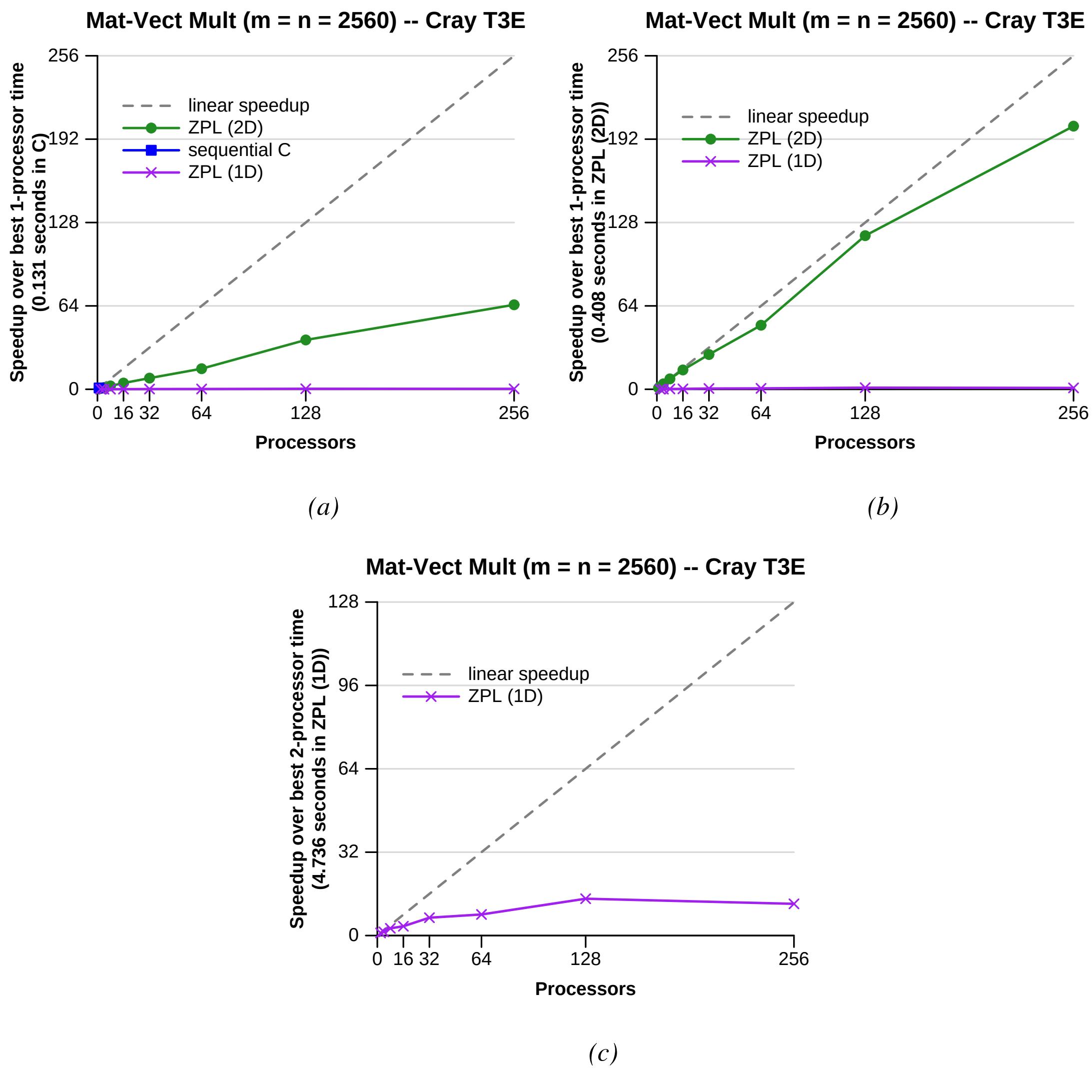


Figure 3.13: Performance of Matrix-Vector Multiplications. (a) The speedup of the 1D and 2D ZPL implementations of matrix-vector multiplication as compared to a hand-coded C version. (b) The speedup of the ZPL implementations against themselves. (c) The speedup of the ZPL 1D version in isolation.

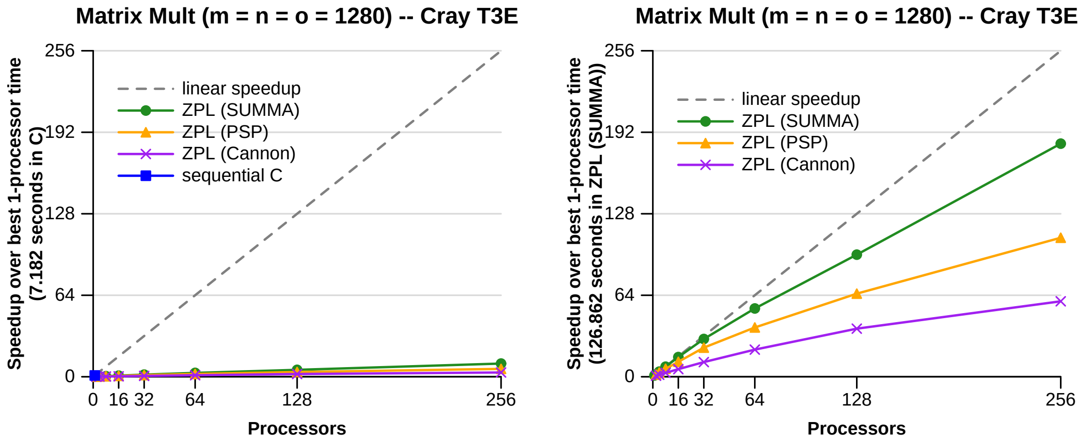
As predicted by the performance model, the 2D ZPL implementation outperforms the 1D implementation due to its reduced memory and communication requirements. It also scales quite uniformly with the processor set, achieving a speedup of 64 on 256 processors. When the speedups are plotted relative to the fastest single-processor ZPL time (Figure 3.13b), the 2D implementation achieves speedup that is closer to linear, while the 1D implementation fails to compete.

Figure 3.13c shows the speedup of the 1D implementation relative to itself. Note that its additional memory requirements prevent it from running this problem size on a single processor. Although the algorithm scales somewhat, its speedup tapers off at 128 processors. This is caused by its use of the remap operator. Remap currently uses a first-generation implementation that was developed with correctness rather than performance in mind. The next chapter describes some of its shortcomings. An effort is currently underway to implement the remap operator more aggressively to take advantage of constant map arrays and reduce their overheads. While it is expected that these changes will benefit the 1D algorithm, its memory requirements and the inherent complexity of the remap operator will probably prevent it from rivaling the 2D implementation.

Matrix Multiplication

ZPL's matrix multiplication codes scale the worst as compared to the sequential C code, as shown in Figure 3.14a. With a bit of reflection, this is not terribly surprising. Each of the ZPL matrix multiplication codes requires a total of $\Theta(n^2)$ communication. In contrast, the C implementation is a trivial triply-nested loop with no data movement or copying. As a result, the ZPL versions simply cannot compete.

Once the C code is removed from the picture (Figure 3.14b), the ZPL codes demonstrate reasonable speedups. No implementation achieves linear speedup, but each scales very consistently with the number of processors. As anticipated, the SUMMA and PSP algorithms outperform Cannon's algorithm due to their reduced per-iteration working sets. The PSP algorithm fails to keep pace with SUMMA due to its use of the remap operator



(a) *(b)*

Figure 3.14: Performance of Matrix Multiplications. (a) The speedup of the Cannon, SUMMA, and PSP Matrix Multiplication algorithms as compared to a sequential hand-coded C version. (b) The speedup of the ZPL implementations in isolation.

to re-align arrays. As with matrix-vector multiplication, its map arrays admit simple optimizations that are currently unimplemented in the compiler. It is expected that once they are there, PSP will be more competitive with SUMMA.

Tridiagonal Matrix Multiplication

Figure 3.15a shows the tridiagonal matrix multiplication codes on the largest problem size that the C code could run on one processor using the recursive doubling scheme. For this problem size, the two $n \times n$ ZPL implementations are unable to run due to their excessive memory requirements, leaving only the compact ZPL version. While this implementation scales quite consistently, the C version is sufficiently simple that ZPL only achieves a speedup of 64 on 256 processors. Once the C code is removed from the picture (Figure 3.15b), the compact ZPL implementation is shown to scale near linearly with respect to itself.

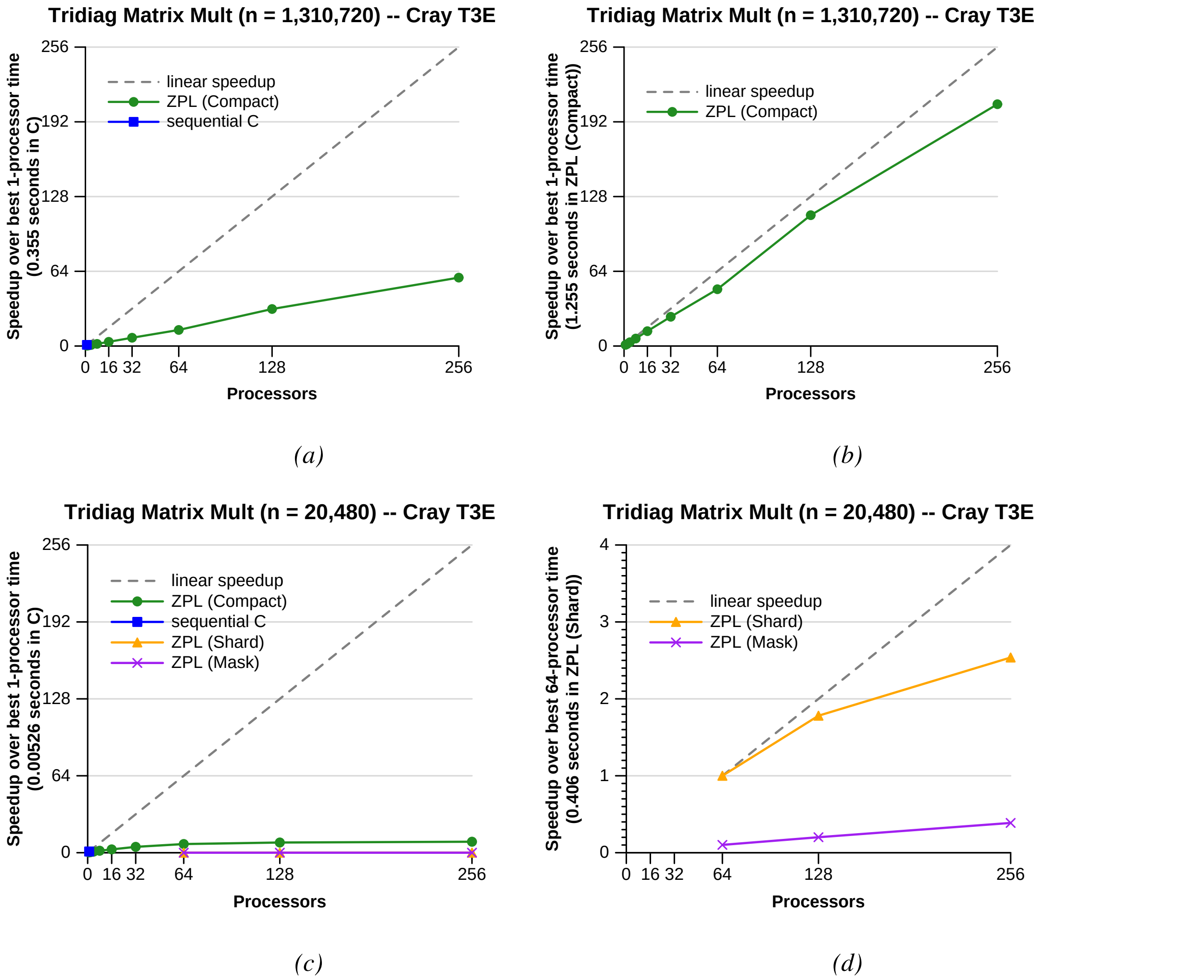


Figure 3.15: Performance of Tridiagonal Matrix Multiplications. (a) The speedup of the compact ZPL algorithm as compared to a sequential hand-coded C version that also uses compact storage. Note that there is not enough memory for the $n \times n$ algorithms to run this problem size. (b) The speedup of the compact ZPL implementation in isolation. (c) The same algorithms with a smaller problem size that demonstrates that the $n \times n$ algorithms cannot compete with the compact versions. (d) The $n \times n$ algorithms in isolation. Note that they do scale, just not at a rate that compares with the more compact algorithms.

Figure 3.15c switches to a smaller problem size to illustrate the behavior of the two $n \times n$ algorithms. At this smaller size, the speedup of all three algorithms is pitiful with respect to the C version due to the increased ratio of communication to computation. The compact ZPL version can be seen to be outpacing the $\Theta(n^2)$ algorithms by a significant margin, as expected by their asymptotic differences. Figure 3.15d shows the two $n \times n$ algorithms isolated for the same problem size. As anticipated, the shard-based implementation outperforms the mask-based implementation due to its reduced memory requirements and accesses.

Summary

To summarize, most of the hand-coded C implementations vastly outperform the ZPL algorithms on a single processor. This should not be surprising, since the C implementations represent sequential algorithms that are inherently simpler than their parallel counterparts. The experiments of Chapters 5 and 6 will remove this difference by comparing ZPL algorithms against other parallel implementations.

In spite of having strictly worse scalar performance, most of the ZPL algorithms scale quite consistently across the range of processor sets. More importantly, the use of ZPL's performance model in Section 3.8 produced accurate predictions of each algorithm's performance (and in the spirit of fair play, that section was written before these experiments were performed). The next chapter will explain how ZPL is implemented to achieve this performance.

3.10 Grid Dimensions

As mentioned previously, Advanced ZPL (A-ZPL) is a successor language to ZPL that is currently under development. The goal of A-ZPL is to expand ZPL's capabilities to provide support for more general parallel programming paradigms including pipelined parallelism, task parallelism, sparse computation, and irregular data structures. Advanced ZPL adds one additional type of region dimension to those supported by ZPL, called the *grid dimension*. Grid dimensions are uninteresting in a sequential context, which is why they were not described in the previous chapter. They also give the programmer access to the local view that implements a ZPL program. Since this is counter to the spirit of ZPL, grid dimensions are reserved for use in A-ZPL programs, where the programmer is expected to have greater awareness of the parallel machine.

3.10.1 Definition

Grid dimensions are specified using a double colon notation (`::`). A grid dimension causes the allocation of a special singleton index per processor in the corresponding dimension of the processor grid. This index conforms to any other index that the processor owns in that dimension. In effect, a grid dimension can be thought of as a flood dimension that is local to a processor. Unlike flood dimensions, the values stored in a grid dimension can vary from one processor to the next.

Arrays that are declared using a grid dimension for each dimension (*grid arrays*) are similar to a private scalar value per processor. In this sense, each processor's unique ID from 0 to p can be thought of as being stored using a grid array. Moreover, a processor's location within a processor grid can be represented using an array with a single grid dimension and flood dimensions everywhere else. For example, a processor would store its row within the processor grid using an array whose first dimension is a grid dimension, and whose other dimensions are floodable.

3.10.2 A Simple Example

A simple example illustrating the use of grid dimensions is shown in Listing 3.4. This code implements a full reduction by explicitly splitting it into local and global steps. The grid array `LocSum` is used to represent each processor's contribution to the global sum. The program begins by initializing each processor's `LocSum` value to 0. It then adds `A`'s values

Listing 3.4: A Reduction Using a Grid Array

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

var A: [R] integer;
    LocSum: [G] integer;
    sum: integer;

...

[G] LocSum := 0;
[R] LocSum += A;
[G] sum := +<< LocSum;
```

to LocSum using region R. This statement causes each processor to iterate over its local indices in R, adding the corresponding value of A to its copy of LocSum. Since LocSum is conformable to all of the processor's indices, it can be read and written for every local index in R. When this statement has completed, LocSum will contain the sum of each processor's local A values. The last statement uses a traditional full reduction over the grid region G to add up each processor's LocSum value. This assigns the global sum of A to the scalar variable sum. See Figure 3.16a for an illustration.

Although this example is somewhat artificial, it illustrates how the ZPL compiler actually implements reductions, since it must typically allocate an array with one or more grid dimensions to store a processor's local contribution to the reduction.

Note that grid dimensions allow users to create programs whose output varies with the number of processors. For example, consider adding the following line to the end of Listing 3.4:

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

This would cause each processor to overwrite its local values of A with its local sum. As a result, A's definition would vary depending on the number of processors as well as the mapping of indices to processors. Figure 3.16b illustrates this effect.

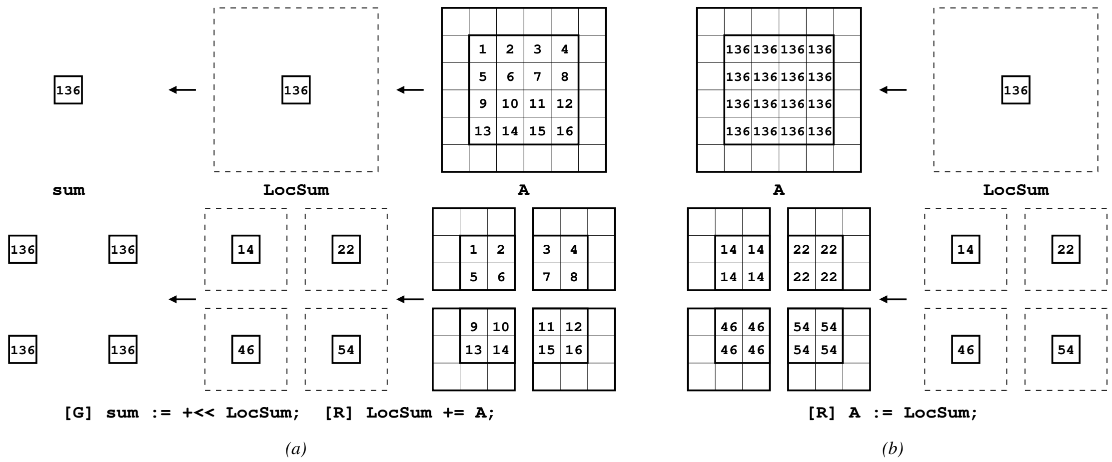


Figure 3.16: Grid Dimension Assignments. Each assignment is shown as it would execute both on a single processor and a 2×2 processor grid. (a) A full reduction written using grid dimensions. First, values of `A` are accumulated into the grid array `LocSum`. Then a full reduction of `LocSum` is performed over grid region `G`. Note that the scalar `sum` has the correct value on all processors in both versions. (b) An illustration of how grid reductions can result in programs whose output varies with the processor grid. The value of `LocSum` computed in part a is assigned back into `A` using region `R`. Note that the four-processor version results in a different global result than the one-processor version. This is caused by the fact that grid dimensions are conformable to multiple indices but are not constrained to have the same value on every processor. It also gives an indication of how grid processors give the user a peek into a program's local view.

Note that simple mistakes can have a big impact when using grid dimensions. For example, if the full reduction in Listing 3.4 was specified using region `R` rather than `G`, each processor would read its value of `LocSum` once per local index from `R`, erroneously inflating the intended value of the sum.

Other mistakes may only impact a program's performance. For example, if the initialization of `LocSum` had been performed using region `R` rather than `G`, it would still have been zeroed out. However, the time required by the assignment would have been proportional to its local `R` indices rather than $\Theta(1)$.

3.10.3 Parallel Implementation

Grid dimensions are distributed and implemented identically to flood dimensions, simply without the constraint that all values must be kept coherent. This tends to be simply a matter of ignoring most of the legality checks that flood dimensions require. As a result, grid dimensions result in no unique runtime implementation issues.

3.10.4 Legality

Grid dimensions have slightly different legality constraints than flood dimensions due to the fact that they may take on different values on different processors. Appendix E contains copies of the legality tables from the previous chapter, amended to include grid dimensions. To avoid repeating content from the previous tables, Appendix E's tables describe each case using a practical parallel interpretation rather than a formal sequential one.

3.10.5 A More Interesting Example

The code in Listing 3.5 shows a more useful application of grid arrays. This code reads a number of integer values and stores each one in a single processor's "bucket" based on its value. The code begins by declaring its configuration variables in lines 1–3. The `bucketsize` variable indicates the capacity of each processor's bucket, while `minval` and `maxval` indicate the expected range of values.

The program declares two regions on lines 5–6: `R`, a 1D region that spans the range of legal indices; and `G`, a 1D grid region. The main array declared by the program is `Bucket`, a grid array of indexed arrays that is used to store each processor's values. A second grid array, `NumVals` keeps track of the number of values that each processor is currently storing. A scalar variable `value` is also declared to hold the value that is currently being considered.

The program begins by initializing `NumVals` to 0 on each processor. It then reads scalar values from the console one at a time, continuing until it finds one that is outside

Listing 3.5: Bucketing Using Grid Arrays

```
config var bucketsize: integer = 100;
           minval: integer = 0;
           maxval: integer = 100;

region R = [minval..maxval];
       G = [ :: ];

var Bucket: [G] array [1..bucketsize] of integer;
    NumVals: [G] integer;
    value: integer;

[G] begin
      NumVals := 0;

      read(value);
      while (value >= minval & value <= maxval) do
       [value] if (NumVals < bucketsize) then
                 NumVals += 1;
                 Bucket[NumVals] := value;
               end;
        read(value);
      end;
    end;
```

of the expected range. Each scalar value is used to define a dynamic singleton region in line 17. This region controls a shattered conditional which is entered only if `NumVals` has not exceeded the bucket size. Since `NumVals` is a 1D array, the enclosing region scope for this conditional is the singleton region `[value]`. This implies that the conditional will only be evaluated on the processor that owns index `value`, and that its unique `NumVals` element will be read. If the processor's bucket is not yet full, `NumVals` is incremented and the scalar value is stored in its bucket at the corresponding location. Then a new value is read and the loop continues.

Note that such a program would be difficult to write without grid dimensions. For example, in traditional ZPL, it would be difficult to store a unique count of values per processor due to the fact that scalars and flood dimensions must contain consistent values for all processors. Possible approaches would be to use a traditional region and have multiple buckets per processor or to set its size to be equal to the number of processors. However, neither of these solutions is quite as elegant as the one supported by grid dimensions.

3.11 Related Work

This section describes other approaches to parallel programming—in particular, libraries and languages that support local and global views. Due to the sheer volume of parallel programming systems that have been developed in the past few decades, this section concentrates on those approaches that are most notable, that are in active use and development today, and that are most closely related to ZPL. The array access mechanisms and support for a performance model are considered for each approach. In addition, most descriptions include a code sample excerpting the main loop from the Jacobi iteration as written in the language to serve as an example. These codes were written by experts in the languages and have been modified only for readability. References for the original code sources are given in their captions. Many of the implementations vary slightly in terms of the algorithm's details, but the expression of the general Jacobi technique will typically be recognizable.

3.11.1 Local-View Libraries

As described in the introduction, local-view libraries are those that give the programmer an interface which allows them to specify the per-processor behavior of a parallel machine. These libraries typically provide several methods of transferring data between processors. This section considers some of the primary examples.

Note that since local-view libraries are used in conjunction with traditional programming languages like C or Fortran 90, the array access methods available to the programmer are those supported by the base language. In addition, as local-view languages, since programmers are responsible for explicitly specifying both the distribution of data and work as well as the program's communication, the performance model is inherent in the code's implementation.

MPI

MPI [Mes94] stands for the *Message Passing Interface*. It is a library specification for message-passing-based routines between cooperating processes. The initial specification of MPI (version 1.1) supports point-to-point communications (sends and receives) in a multitude of varieties (including blocking, non-blocking, buffered, ready-send, and send-receive). In addition, MPI supports higher-level operations such as broadcasts, all-to-all scatters and gathers, reductions, scans, and barrier synchronizations. MPI is a highly structured interface, allowing users to specify new datatypes and to organize processors into different groups. MPI version 2 adds support for additional features such as one-sided communications, process creation and management, extended collective operations, and I/O [Mes97].

MPI has succeeded in becoming the *de facto* standard for parallel programming due to its portability and widespread availability. MPI is supported on almost every parallel platform, and free implementations are available for most commodity platforms [GLDS96, Ohi96].

Listing 3.6: MPI Jacobi Implementation Excerpt [Gro01]

```
/* Main loop of Jacobi Iteration */
do {
  /* Send up unless I'm at the top, then receive from below */
  /* Note the use of xlocal[i] for &xlocal[i][0] */
  if (rank < size - 1)
    MPI_Send(xlocal[maxn/size], maxn, MPI_DOUBLE, rank + 1, 0,
             MPI_COMM_WORLD);
  if (rank > 0)
    MPI_Recv(xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0,
             MPI_COMM_WORLD, &status);

  /* Send down unless I'm at the bottom */
  if (rank > 0)
    MPI_Send(xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1,
             MPI_COMM_WORLD);
  if (rank < size - 1)
    MPI_Recv(xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank + 1, 1,
             MPI_COMM_WORLD, &status);

  /* Compute new values (but not on boundary) */
  diffnorm = 0.0;
  for (i=i_first; i<=i_last; i++)
    for (j=1; j<maxn-1; j++) {
      xnew[i][j] = (xlocal[i][j+1] + xlocal[i][j-1] +
                    xlocal[i+1][j] + xlocal[i-1][j]) / 4.0;
      diffnorm += (xnew[i][j] - xlocal[i][j]) *
                  (xnew[i][j] - xlocal[i][j]);
    }
  /* Only transfer the interior points */
  for (i=i_first; i<=i_last; i++)
    for (j=1; j<maxn-1; j++)
      xlocal[i][j] = xnew[i][j];

  MPI_Allreduce(&diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM,
                MPI_COMM_WORLD);
  gdiffnorm = sqrt(gdiffnorm);
} while (gdiffnorm > 1.0e-2);
```

One of MPI's primary disadvantages is that message-passing is not always the cheapest means of moving data around a parallel machine. For example, message-passing semantics often require buffering and synchronization that are not inherently required by shared memory or shared address space machines. This can result in overheads beyond those required for communication on a given architecture.

This problem tends to be combatted by expanding the MPI interface, particularly by the addition of one-sided communications in MPI-2. The problem is that such generalizations are at odds with portability. In particular, while an MPI program may be written in a style that is suitable for a particular machine (*e.g.*, a one-sided message passing style on the Cray T3E), its suitability for other machines is not guaranteed, forcing the programmer to consider changing the MPI calls for each platform.

Note that this problem is not particular to MPI, but is rather a seemingly inevitable problem with any interface that specifies particular communication semantics and yet wants to run on a diverse set of parallel architectures. Chapter 4 introduces ZPL's paradigm-neutral interface that manages to avoid this paradigm-mismatch problem.

In spite of its drawbacks, MPI should be considered a success, due not only to its popularity, but also the fact that it is enabling more people to write parallel programs today than any other approach described in this dissertation (including ZPL, regrettably).

Listing 3.6 shows the inner loop of the Jacobi iteration written using C and MPI for a 1D processor grid. Lines 5–18 implement the communication required to exchange boundary values between processors to the north and south. Note that a 2D decomposition would not only require similar lines for east and west communications, but also code to marshall the data in and out of temporary buffers, since those elements would not be adjacent in memory. Lines 21–32 perform the main computation and update. Lines 34–35 use an MPI reduction routine to calculate the global normalized difference.

PVM

PVM stands for *Parallel Virtual Machine* [BDG$^+$91] and could be considered the awkward cousin of MPI. A rough characterization of PVM might describe it as being a quickly-built, practical solution to parallel programming which gradually became more sophisticated as time passed. In contrast, MPI was developed as a standard for which complete implementations have gradually been developed (for example, a complete implementation of MPI-2 is not yet available as of this writing). One effect of this is that while each MPI routine takes a dozen parameters allowing for type information, error conditions, processor sets, identifiers, and the actual data to be described, PVM's interface tends to be somewhat simpler and less general. In spite of this, PVM supports many of the same communication types as MPI, including blocking and non-blocking sends and receives, process creation and management, broadcasts, reductions, and barrier synchronizations. PVM has similar portability problems as MPI, but has not taken the same approach of diversifying the available communication styles. This may prevent it from performing as well on diverse architectures, but prevents the interface from being as confusing.

PVM is freely available for most standard platforms [PVM01], and should be considered a reasonable and somewhat simpler alternative to using MPI. A PVM implementation of Jacobi would look extremely similar to the MPI version in Listing 3.6, simply replacing the MPI calls with their equivalent PVM routines.

SHMEM

The SHMEM interface [BK94] is another inter-process communication library that was developed for the Cray T3D in order to expose the low-overhead one-sided communications supported by its architecture. In addition to one-sided *puts* and *gets*, SHMEM supports higher-level operations such as broadcasts, reductions, and barrier synchronizations.

One disadvantage of the SHMEM interface is that some characteristics of the Cray T3D architecture are embedded in the routines themselves. For example, collective operations such as reductions and broadcasts can only be performed on subsets of processors that are strided by powers of two. Thus, to perform a reduction or broadcast over a general set of processors, users must write their own routines using puts and gets.

Another disadvantage of SHMEM is again one of paradigm/architecture mismatches. SHMEM's routines are ideal for shared address space machines like the Cray T3D and T3E which support one-sided communications. It is also appropriate for shared memory machines, on which such routines can easily be implemented. However, on distributed memory architectures that have no hardware support for writing directly to a processor's local memory, some amount of overhead is required to watch for incoming messages and store them to memory appropriately. Though there have been some early attempts to support such implementations [BB00], it has yet to be proven that one-sided communication interfaces can support low-overhead data transfers on such architectures.

A SHMEM version of Jacobi would also look very similar to the MPI version, except that each pair of send/receive calls would be replaced by a single `shmem_put()` or `shmem_get()` call that specifies both the local and remote addresses. SHMEM's symmetric memory allocation routines would be used to avoid explicitly transferring addresses between processors.

Other Local-View Libraries

Two other local-view libraries that deserve brief mention are Intel's NX library [Pie93] and the Active Messages work by Culler, von Eicken, *et al.* [vECGS96]. NX is yet another message-passing standard, developed for parallel Intel machines like the Paragon. It has largely faded from use with the passing of those platforms. NX differs somewhat from MPI and PVM in that it provides routines not only for blocking and non-blocking communications, but also for sends and receives with *callback routines* that are executed when a message is successfully sent or received. These routines are ideally run on a dedicated co-processor to perform actions such as unpacking the message into memory while the original program thread continues unhindered.

Active Messages represents a related idea in which a function pointer is bundled into a message along with its data. Upon receiving the message, the receiving processor calls the specified function, allowing the message to be handled in a way that best suits the details of the architecture. As such, Active Messages have been touted as an extremely portable, low-overhead communication strategy, suitable for use in implementing higher-level languages such as Split-C (described in the next section).

3.11.2 Local-View Languages

Split-C

Split-C [CDG+93, CDG+95] is an extension to C that was designed to support parallel programming. In doing so, its designers were motivated to provide efficient access to the underlying machine with no surprises, much as in traditional C (and ZPL). Split-C supports a global address space across all processors as well as global pointers that can refer to objects located anywhere in the address space (*i.e.*, on any processor). Global pointers are represented using a 2-tuple containing a processor number and a local address on that processor. Arrays, whether pointer-based or C-style arrays, can be declared to be *spread*, which causes indices in the spread dimensions to be distributed cyclically across the processor set.

Split-C also supports a number of novel assignment operators that reduce the synchronization requirements of a traditional assignment. For example, the split-phase assignment (`:=`) allows a non-local value to be assigned to a local value or vice-versa. This operator specifies that such an assignment should take place and initiates the communication required to carry it out. However, rather than waiting for the assignment to complete, it simply proceeds to the next statement. To ensure that a split-phase assignment has completed, programmers use synchronization calls that block until all preceding split-phase assignments are done. In this sense, split-phase assignment offers a language-level equivalent to one-sided communication.

In spite of being a relatively high-profile and well-published language, Split-C has never become widely used in the community, possibly due to the fact that its one-sided communication style was difficult to implement efficiently on distributed memory platforms.

Co-Array Fortran

Developed at Cray Research, Co-Array Fortran (CAF) [NR98] is another parallel language developed by extending a traditional language, in this case Fortran 90. The primary addition to the language is the concept of the *co-array*. Co-array dimensions are simply extra array dimensions that refer to processor space rather than data space. For example, a scalar variable declared with a co-array dimension causes each processor to allocate a copy of that variable, much like ZPL's grid arrays. Appending co-array dimensions to traditional arrays results in a blocked parallel array. Remote data is referred to by indexing into a co-array dimension with the remote processor's index. This serves as a concise representation of interprocessor communication that is simple, yet extremely elegant and powerful. CAF also provides a number of synchronization operations which are used to maintain a consistent global view of the problem.

Listing 3.7 shows a Jacobi implementation in CAF. The co-array references are the expressions in square brackets on lines 8–11, 30, and 35. All of these references are on the right-hand side of the assignment, indicating that a get-style communication is used to access the remote processor's memory. The initial lines exchange boundary values between processors, while the final pair are used to implement an $\Theta(p)$ reduction and broadcast. Synchronization is used in lines 3–7, 27, and 34 to ensure that all computations have completed before the data transfers take place. The rest of the code is a straightforward local-view implementation.

Listing 3.7: CAF Jacobi Implementation Excerpt [Wal01]

```
      DO
!       update halo.
        IF (MIN(P, Q) >= 3) THEN
          CALL SYNC_ALL( WAIT=NEIGHBORS )
        ELSE
          CALL SYNC_ALL()
        ENDIF ! neighbor images have ANS(1:NN, 1:MM) up to date
        ANS(1:NN, MM+1) = ANS(1:NN, 1   )[ME_P,  ME_QP]  ! north
        ANS(1:NN,    0) = ANS(1:NN,   MM)[ME_P,  ME_QM]  ! south
        ANS(NN+1, 1:MM) = ANS(1,    1:MM)[ME_PP, ME_Q ]  ! east
        ANS(   0, 1:MM) = ANS(  NN, 1:MM)[ME_PM, ME_Q ]  ! west

!       5-point stencil is correct everywhere,
!       since halo is up to date.
        DO J= 1, MM
          DO I= 1, NN
            WRK(I, J) = (1.0/4.0) * (ANS(I-1, J  ) + &
                                     ANS(I+1, J  ) + &
                                     ANS(I  , J-1) + &
                                     ANS(I  , J+1)   )
          ENDDO
        ENDDO

!       calculate global maximum residual error.
        PMAX = MAXVAL( ABS( WRK(1:NN, 1:MM) - ANS(1:NN, 1:MM)))
        CALL SYNC_ALL()  ! protects both PMAX and ANS
        IF (ME == 1) THEN
          DO I= 2, NUM_IMAGES()
            PMAXI = PMAX[I]
            PMAX  = MAX( PMAX, PMAXI )
          ENDDO
        ENDIF
        CALL SYNC_ALL( WAIT=(/1/) )  ! protects PMAX[1]
        RESID_MAX = PMAX[1]

!       update the result, note that above SYNC_ALL() guarantees
!       that the old ANS(1:NN,1:MM) is no longer needed for halo
!       update.
        ANS(1:NN, 1:MM) = WRK(1:NN, 1:MM)
      UNTIL (RESID_MAX <= TOL)
```

Local Language Summary

The local views promoted by Split-C and CAF represent both a blessing and a curse. To their credit, they give programmers the means to control the details of an algorithm's parallel implementation, relying on the compiler only to insert and implement the actual communication. Use of certain language primitives like split-phase assignment and co-array dimensions make it reasonably clear where communication is required by the program (though it should be noted that traditional assignments using global pointers in Split-C may hide communication). Combining these cues with the explicit user-specified distributions of data and computation, programmers have a reasonably good performance model for evaluating their codes.

On the other hand, local-view languages have the disadvantage that the programmer must manage all of these details by hand, which can become tedious even for trivial codes like Jacobi. This is especially true for cases in which a distributed array's size does not divide evenly amongst the processors. As another example, consider the amount of work that would be required to rewrite the CAF reduction in Listing 3.7 to use a general $\Theta(\log p)$ reduction scheme for arbitrary values of p. These are the sorts of details that global-view languages like ZPL strive to manage on the user's behalf. Moreover, ZPL's performance model does so without letting programmers lose sight of the lower-level details of their program's parallel implementation.

3.11.3 Global-View Languages

High Performance Fortran

High Performance Fortran (HPF) [Hig94] is another extension to Fortran 90 that was developed by the High Performance Fortran Forum, a coalition of academic and industrial experts. HPF is based on the earlier parallel Fortran dialects Fortran-D and Vienna Fortran [FHK+90, ZBC+92]. These languages support parallel computation through the use of programmer-inserted compiler directives. HPF's directives allow users to give hints

Listing 3.8: HPF Jacobi Implementation Excerpt [Joi01]

```
      PROGRAM jacobi1
!
! Solve the Poisson equation with the Jacobi method
!
      PARAMETER (nx = 100, ny=100)
      PARAMETER (tol = 1.0e-8)
      REAL u(0:nx,0:ny), unew(0:nx,0:ny)
      REAL dx, dy, error, tol
      INTEGER i, j
!HPF$ DISTRIBUTE u(BLOCK,*)

      ! initialize all data
      ...

      ! Jacobi iteration
      error = tol + 1.0
      DO WHILE (error > tol)
        FORALL ( i=1:nx-1, j=1:ny-1 )
          unew(i,j) = (u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)) / 4
        END FORALL
        error = MAXVAL( ABS(unew-u) )
        u = unew
      END DO
      STOP
      END
```

for array distribution and alignment, loop scheduling, and other details relevant to parallel computation. The hope is that with a minimal amount of effort, programmers can modify existing Fortran codes by inserting directives that will enable HPF compilers to generate an efficient parallel implementation of the program. HPF supports a global view of computation, managing parallel implementation details such as array distribution and interprocessor communication in a manner that is invisible to the user without the use of compiler feedback or analysis tools. HPF also adds a forall loop construct as in F95 and FIDIL.

The primary disadvantage of HPF is that its specification makes no guarantees as to how compilers will interpret and implement its directives. In particular, no guidelines are provided for programs with conflicting or underspecified directives, and the compiler may choose to ignore a program's directives altogether. The result is that the programmer has little basis for evaluating a code's parallel implementation. Moreover, since each compiler may implement its own interpretation of the directives, programmers may have to re-tune their programs from one compiler or architecture to the next [NSC97, Ngo97].

Listing 3.8 shows a Jacobi iteration kernel written in HPF. Line 10 contains the only HPF directive, specifying that the first dimension of `u` should be distributed in a blocked manner, and that the second dimension should not be distributed. This implementation does not specify `unew`'s distribution, relying on the compiler to align it with `u` for optimal performance. A more cautious programmer would add a second directive specifying its alignment explicitly: `ALIGN unew(:,:)` **`WITH`** `u(:,:)`. Note that as a global-view language, HPF's references to `nx` and `ny` describe the global size of the problem rather than a processor's local block. Finally, note that the reduction is implemented using HPF's intrinsic `MAXALL()` function.

OpenMP

OpenMP is not technically a language, but rather a set of compiler directives and library routines that can be used in C or Fortran programs to specify their parallel execution on shared memory platforms [Ope01]. Nevertheless, OpenMP's directives resemble those of

Listing 3.9: OpenMP Jacobi Implementation Excerpt [KAI01]

```
      do while (error.gt. tol)

         error = 0.0

!* Copy new solution into old
!$omp paralleldo schedule(static)
!$omp& shared(n,m,uold,u)
!$omp&private(i,j)
         do j=1,m
            do i=1,n
               uold(i,j) = u(i,j)
            enddo
         enddo
!$omp end paralleldo

!* Compute stencil, residual, & update

!$omp paralleldo schedule(static)
!$omp& shared(omega,error,tol,n,m,ax,ay,b,alpha,uold,u,f)
!$omp&private(i,j,resid)
!$omp&reduction(+:error)
         do j = 2,m-1
            do i = 2,n-1
!*     Evaluate residual
               resid = (ax*(uold(i-1,j) + uold(i+1,j))
     &                  + ay*(uold(i,j-1) + uold(i,j+1))
     &                   + b * uold(i,j) - f(i,j))/b
!* Update solution
               u(i,j) = uold(i,j) - omega * resid
!* Accumulate residual error
               error = error + resid*resid
            enddo
         enddo
!$omp end paralleldo

!* Error check

         error = sqrt(error)/dble(n*m)

      enddo                     ! End iteration loop
```

a global-view language like HPF closely enough that this section is most appropriate for its discussion.

OpenMP's model is to spawn threads for structured blocks of a program to implement them in parallel. The hope is that this can be done incrementally to convert a sequential program into a parallel one. As in most languages described here, parallelism is achieved by assigning a loop's iterations to different threads. OpenMP's directives give cues to the compiler, such as how loops should be parallelized and whether the code contains operations such as reductions. As in most shared memory programming, explicit locks and synchronization are required to prevent race conditions and data conflicts, and OpenMP's library routines support such mechanisms.

OpenMP has rapidly gained support by the industry, primarily due to the large number of symmetric multiprocessors being produced by traditional sequential hardware vendors. At this point, the main question regarding its success is whether its directives are rich enough and well-defined enough to make OpenMP programs readable and portable. In addition, the fact that OpenMP is not intended for distributed memory platforms will restrict its portability, especially with the community's recent enthusiasm for commodity distributed memory clusters. Even in its target domain of shared memory machines, OpenMP makes data locality seductively invisible. This supports the myth that shared memory systems make all memory equally available to all processors. In reality, such a model disguises the fact that data locality is as crucial to performance on shared memory architectures as it is on distributed memory machines.

Listing 3.9 shows the main loop of an OpenMP Jacobi implementation. OpenMP directives are applied in lines 6–8 and 18–20 to specify the parallel execution of the do loops. In addition, these directives indicate that the arrays and problem sizes should be shared amongst the threads while the loop iterators and local reduction value `resid` should be private for each thread. Line 21 specifies that the sum and assignment into `error` on line 29 should be implemented as a reduction across threads. The rest of the code describes the Jacobi iteration using a global view.

Unified Parallel C (UPC) [CDC$^+$99] is the latest in the evolution of C-based parallel programming languages. In many ways it resembles Split-C or CAF, except that it supports a global view of computation rather than a local view. This distinction is subtle, but can best be characterized by the fact that UPC codes tend to declare variables using their global size rather than a local size crossed with a number of processors. Furthermore, UPC codes are typically written without referring to a processor's unique index.

UPC provides a *distributed shared memory programming model* that gives each thread a private local memory, but also supports a logically shared portion of the address space that any thread can access. Thus, pointer variables may be private or shared, and may point to memory that is either private to the thread, or shared between all of them. Parallelism is typically expressed in UPC using a forall-style loop that not only specifies the loop's bounds, but also the assignment of loop iterations to processors (*affinity*). This mechanism allows the expression of parallel computations over shared arrays using blocked, cyclic, or other traditional distributions. Due to its distributed shared memory model, UPC programmers must specify synchronization explicitly using barrier, split-phase barrier, and locking primitives provided by the language. The current UPC specification has limited support for operations like reductions.

UPC is the youngest of the languages considered in this section, and as such, it has not yet had much of an opportunity to prove itself. The biggest hurdle it seems to face is the fact that moving data from the shared portion of the address space to a processor's physical memory occurs rather transparently in the language, disguising the overhead of communication on both shared and distributed memory platforms. This will obfuscate the performance model for the programmer, though careful programmers may be able to use the affinity field of UPC's forall loops to carefully manage their programs' locality.

Single-Assignment C

Single Assignment C (SAC) is a functional variation of ANSI C developed at the University of Kiel [Sch98b, Sch94]. Its extensions to C support multidimensional arrays, APL-like operators for dynamically querying array properties, and functional semantics. SAC also supports a *with-loop* construct that superficially resembles ZPL's regions. Like regions, with loops are used to specify the indices involved in a computation for a statement or group of statements. However, unlike regions, the with loop provides a mechanism for iterating over an index set by generating indices that can be used to index into an array rather than replacing array indexing altogether. In this sense, the with loop resembles the forall construct of languages like F95 and FIDIL rather than ZPL's regions. SAC currently runs only on shared memory machines, and issues like array distribution and interprocessor communication are invisible to the programmer.

Listing 3.10 shows a SAC implementation of Jacobi designed to run for `LOOP` iterations. Line 10 shows the use of a with-loop to iterate over `B`'s bounds, applying the five-point stencil using the `modarray` operator to modify `B`. Note that `B` can serve as both the source and destination of the stencil due to SAC's functional semantics. Lines 35–36 show a sum reduction that is used by this program as a checksum.

NESL

NESL is a data-parallel programming language developed at Carnegie Mellon that implements nested parallelism using functional semantics [Ble95, Ble96]. Functional semantics dictate that many functions can be executed in parallel due to the complete lack of aliasing between sibling function calls. NESL supports nested parallelism by allowing these functions to spawn parallel function calls of their own.

NESL also supports the concept of data parallelism using its *sequence* concept—a one-dimensional distributed array that can consist of data items or other sequences. NESL provides a parallel *apply-to-each* construct to operate on a sequence's elements in parallel.

Listing 3.10: SAC Jacobi Implementation Excerpt [SAC01]

```
/**************************************************************
 * description:
 *
 *   This SAC demo program implements 2-dimensional relaxation
 *   on double precision floating point numbers applying a
 *   5-point stencil and fixed boundary conditions.
 *************************************************************/

inline double[] onestep(double[] B) {
  A = with ( . < x < . )
      modarray(B, x, 0.25*(B[x+[1,0]]
                         + B[x-[1,0]]
                         + B[x+[0,1]]
                         + B[x-[0,1]]) );

  return(A);
}

inline double[] relax(double[] A, int steps) {
  for (k=0; k<steps; k++) {
    A = onestep(A);
  }

  return(A);
}

int main () {
  A = with( . <= x <= .)
      genarray([SIZE1, SIZE2], 0.0d);

  A = modarray(A, [0,1], 500.0d);

  A = relax( A, LOOP);

  z = with( 0*shape(A) <= x < shape(A))
      fold(+, A[x]);

  printf("%.10g\n", z);

  return(0);
}
```

Listing 3.11: NESL Jacobi Implementation Excerpt [NES01]

```
function sparse_MxV(mat,vect) =
let l = {#row: row in mat};
    i,v = unzip(flatten(mat))
in {sum(row): row in partition({x*v: x in vect->i; v},l)} $

% Each Jabobi iteration is just a matrix vector product,
  this will just repeat it n times. %
function Jacobi_Iterate(Mat,vect,n) =
  if (n == 0) then vect
  else Jacobi_Iterate(Mat,sparse_MxV(Mat,vect),n-1) $

% THE FOLLOWING TWO FUNCTIONS ARE MESH SPECIFIC AND ONLY GET
  CALLED ONCE TO INITIALIZE THE MESH AND VECTOR %

function make_2d_n_by_n_mesh(n) =
  let
    % A sequence of indices of the internal cells. %
    intrnl_ids = flatten({{i+n*j: j in [1:n-1]}: i in [1:n-1]});

    % Creates a matrix row for each internal cell.
      Each row points left, right, up and down with weight .25 %
    internal = {(i,[((i+1), .25), ((i-1), .25),
                    ((i+n), .25), ((i-n), .25)]):
                 i in intrnl_ids};

    % Creates a default matrix row (used for boundaries).
      Each points to itself with weight 1 %
    default = {[(i,1.0)]: i in [0:n^2]}

  % Insert internal cells into defaults %
  in default <- internal $

% Assumes mesh is layed out in row major order %
function make_initial_vector(n) =
  dist(0.0,n^2) <- { i, 50.0: i in [n^2-n:n^2]} $

% ENTRY POINT %
function runit(n, steps) =
let matrix = make_2d_n_by_n_mesh(n);
    vector = make_initial_vector(n);
in Jacobi_Iterate(matrix,vector,steps) $
```

The language's combination of nested parallelism and data parallelism allows the expression of parallel divide-and-conquer algorithms such as Quicksort, as well as traditional data parallel algorithms such as matrix additions and multiplications.

NESL has a well-defined performance model that uses a work/depth scheme to calculate asymptotic bounds for the execution time of NESL programs on parallel computers. Although this model is well-suited to the language's functional paradigm and allows users to make coarse-grained algorithmic decisions, it reveals very little about the lower-level impact of one's implementation choices and how they will be mapped to the target machine. In particular, issues of data locality and interprocessor communication are completely invisible in NESL's syntax and performance model, and are therefore inscrutable to the user.

Listing 3.11 gives a NESL implementation of the Jacobi iteration that runs for `steps` iterations. This version implements Jacobi by multiplying the data array (stored in the n^2-element sequence `vect`) by an extremely sparse matrix `Mat` that describes the 5-point stencil. This matrix is set up in the `make_2d_n_by_n_mesh()` routine in lines 15–31. The actual computation is expressed recursively in lines 1–10.

3.11.4 Global-View Libraries

Mathematical Libraries

As described in the introduction, the vast majority of global-view libraries export interfaces for high-level mathematical operations such as matrix multiplications or matrix solvers. While such routines tend to be highly optimized, they strive to serve a more restricted purpose than the other work described in this section. As a result, these libraries should not be viewed as competitors with languages so much as resources that can be used within a parallel language. The primary challenge to doing so is that each language and library tends to have its own array format and distribution scheme. As a result, trying to get them to interoperate often requires remapping an array from one allocation to another, which can be quite expensive.

KeLP

KeLP [BCSvS01, BFS01, FKB98] is a C++ library that evolved from LPARX [Koh95] and more distantly from FIDIL. It supports rectangular index set classes called *regions* that are used for iteration and to declare arrays. Rather than distributing a single region between processors as in ZPL, KeLP breaks a problem's global index set down into a collection of regions, each of which is assigned to a specific processor to achieve parallelism. Groups of cooperating regions are managed and distributed across physical processors using the *FloorPlan* class. FloorPlans are also used to declare parallel array instance variables, known as *XArrays*. Data transfers between XArrays are captured using *MotionPlan* objects that describes the indices that need to be communicated between processors for a particular FloorPlan, while *Mover* objects implement the specific schedule that a MotionPlan describes for an XArray. KeLP's regions support high-level operators such as shift, intersect, and grow. KeLP expresses iteration over a region's indices using an *indexIterator* class that provides forall-style semantics.

KeLP is unique among the languages described in this section due to the fact that it takes a very practical stance for solving extremely hard problems utilizing multiple arrays at multiple scales. While other languages are trying hard to get a simple Jacobi iteration or matrix multiplication to run well, KeLP is on the front lines working on a crucial class of problems that most languages (including ZPL) hope to be able to handle "someday." This practicality is not without a certain amount of unwieldiness, however, as KeLP's Jacobi implementation demonstrates. Furthermore, KeLP is targeted heavily at grid-based computing, and therefore may not be as general as other languages described here.

The KeLP implementation of Jacobi is shown in Listings 3.12 and 3.13. Listing 3.13 contains the code which declares the main Region (line 19) and then partitions it into subregions, capturing them in a FloorPlan (line 20). It then declares two arrays, `grid1` and `grid2` using that FloorPlan (line 22). Lines 2–15 set up the MotionPlan required to transfer boundary values between processors for the FloorPlan. Line 26 establishes pointers to

Listing 3.12: KeLP Jacobi Implementation Excerpt [KeL01]

```
/* perform one 5-point jacobi stencil operation on oldgrid,      *
 * storing the result in newgrid                                 */
void ComputeLocal(XArray2<Grid2<double> >& oldgrid,
                  XArray2<Grid2<double> >& newgrid) {
   int i;

   for (nodeIterator ni(oldgrid); ni; ++ni) {
      i = ni();

      Grid2<double>& OG = oldgrid(i);
      Grid2<double>& NG = newgrid(i);

      Region2 interior = grow(OG.region(),-1);

      FortranRegion2 Foldgrid(OG.region());

      f_j5relax(OG.data(), FORTRAN_REGION2(Foldgrid),
             NG.data());
   }
}

/* compute the error norm max(abs(newgrid - oldgrid))            *
 * Note: for better performance this should be done in Fortran   */
double computeNorm(XArray2<Grid2<double> >& oldgrid,
                   XArray2<Grid2<double> >& newgrid) {
  double result = 0.0;
  int i;
  Point2 p;

  for (nodeIterator ni(oldgrid); ni; ++ni) {
    int i = ni();
    const Region2 interior = grow(oldgrid(i).region(), -1);
    for (indexIterator2 ii(interior); ii; ++ii) {
      p = ii();
      result = MAX(ABS(newgrid(i)(p)-oldgrid(i)(p)),result);
    }
  }

  mpReduceMax(&result,1);
  return(result);
}
```

Listing 3.13: KeLP Jacobi Implementation Excerpt (continued) [KeL01]

```
/* fill in a one-cell ghost region for each Grid in X          */
void initMotionPlan(FloorPlan2& X, MotionPlan2 &M) {
  int i;
  int j;

  for (indexIterator1 ii(X); ii; ++ii) {
    i = ii(0);
    Region2 inside = grow(X(i), -1);

    for (indexIterator1 jj(X); jj; ++jj) {
      j = jj(0);
      if (i != j) M.CopyOnIntersection(X,i,X,j,inside);
    }
  }
}

void main() {
  MotionPlan2 M;
  Region2 domain(1,1,N,N);
  FloorPlan2 T = UniformPartition(domain);

  XArray2<Grid2<double> > grid1(T), grid2(T);
...
  initMotionPlan(T,M);

  XArray2<Grid2<double> > *oldgrid = &grid1, *newgrid = &grid2;

  Mover2<Grid2<double>, double>* pDM1 =
            new  Mover2<Grid2<double>, double>(grid1,grid1,M);
  Mover2<Grid2<double>, double>* pDM2 =
            new  Mover2<Grid2<double>, double>(grid2,grid2,M);

  do {
    /* Exchange boundary data with neighboring processors */
    pDM1->execute();
    SwapPointer(&pDM1, &pDM2);

    /* Perform the local jacobi computation */
    ComputeLocal(*oldgrid,*newgrid);

    /* Compute the stopping criterion */
    stop = computeNorm(*oldgrid, *newgrid);

    /* Swap the pointers to the grids */
    SwapPointer(&oldgrid,&newgrid);
  } while (stop > epsilon);
}
```

each grid to support quick swapping on each iteration, while lines 28–31 establish a Mover for each grid. The main loop takes place in lines 33–46. The Mover is executed on line 35, causing boundary values to be updated for the current grid. The five point stencil is expressed using a simple Fortran routine for efficiency (not shown here), and is applied to the arrays using the `ComputeLocal()` function of Listing 3.12. Then the normal value is computed, the pointers are swapped, and the next iteration begins.

3.11.5 SIMD Programming Languages

Parallaxis-III and C∗

Though the SIMD (Single Instruction, Multiple Data) programming model seems to be in a state of indefinite retirement, two SIMD programming languages deserve mention here for their index set concepts: Parallaxis-III and C∗ [Brä95, Thi91]. Both languages support dense multidimensional index spaces that are used to declare parallel arrays. Parallaxis-III array statements are performed over the entire array, and therefore do not use index sets to describe computation. In contrast, C∗ does use its index sets (*shapes*) to designate parallel computation over entire arrays. However, it enforces a tight correspondence between the shapes of the computation and the arrays being used. Due to this restriction, its shapes are more of a type modifier than a general index set for expressing array computation. Both languages allow for individual elements to be masked. Neither provides support for strided index sets.

3.11.6 Summary

Table 3.2 provides an overview of the main approaches considered in this section. Its columns indicate: whether each approach is a library- or language-based approach; whether it supports a local-view or global-view of computation; whether it assumes a distributed memory (DM), shared address space (SAS), or shared memory (SM) memory model; its supported notation for array accesses; how its syntax indicates concurrency and communi-

Table 3.2: Summary of Main Programming Approaches

Name	Type of Approach	Programmer View	Memory Model	Array Access Style	Concurrency Indicator	Communication Indicator	Synchro-nization
CAF	language	local	SAS/SM	F90	explicit	co-array reference	explicit
HPF	language	global	DM/SM	F90+forall	directives	none	implicit
KeLP	library	both	DM/SM	forall+indexing	forall	motion plans	implicit
MPI	library	local	DM/SM	base language	explicit	explicit	implicit
NESL	language	global	DM/SM	forall	forall; work/depth	none	implicit
OpenMP	directives	local	SM	base language	directives	none	explicit
PVM	library	local	DM/SM	base language	explicit	explicit	implicit
SAC	language	global	SM	forall+indexing	forall	none	implicit
Split-C	language	local	DM/SM	C	explicit	novel assignments/none	explicit
SHMEM	library	local	SAS/SM	base language	explicit	explicit	explicit
UPC	language	global	DM/SM	forall+indexing	forall	none	explicit
ZPL	language	global	DM/SM	regions+array ops	region scope	array operators	implicit

cation; and whether synchronization is implicit in the approach or explicitly specified by the programmer.

Generally characterizing the space of related work, the trend seems to be that local-view libraries and languages support good flexibility and a performance model for programmers, but require a greater programming effort to explicitly manage the details of parallel programming. By contrast, global-view languages relieve programmers of much of this burden, but often hide it so well that users cannot make informed decisions about their programs. One such example is the lack of cues to indicate communication and data distribution information in languages like HPF and NESL. In ZPL, the goal is that the distribution of regions and their use with array operators will provide the programmer with sufficient information to understand the program's parallel implementation without explicit compiler feedback or performance analysis tools.

3.12 Discussion

3.12.1 Current Support for Region Distribution

The current implementation of ZPL's regions matches this chapter's content very closely with two important limitations. First, a region's dimensions can currently only be distributed in a blocked manner. Second, all of a program's regions are considered interacting. This second limitation implies that there will only be a single processor grid per program.

These limitations exist primarily due to insufficient motivation rather than technical challenges. In particular, most of the applications that have been written in ZPL to date only use a single index space to describe their algorithms. Furthermore, the algorithms' characteristics have always been amenable to block distribution. As a result, the mechanisms for specifying alternative distribution schemes or additional processor grids have never been developed. This is not to say that support for other distributions or multiple problem sizes is not important in a language like ZPL, but simply that the problems which require such mechanisms have not yet been studied in ZPL.

Summary of Current Scheme

Currently, ZPL programmers specify the number of processors and the dimensions of the processor grid on the command line of their ZPL executables. The $\texttt{-p}p$ flag is used to specify the number of processors, while $\texttt{-r}p_1$, $\texttt{-c}p_2$, and $\texttt{-l}p_3$ can be used to specify the number of processor rows, columns, or levels in grids that are 1D–3D. Processor grids greater than 3D can be specified using $\texttt{-g}p_1\texttt{x}p_2\texttt{x}\ldots\texttt{x}p_d$. If the processor grid's dimensions are underspecified, the ZPL runtime heuristically uses a configuration that divides the processors between the grid dimensions as evenly as possible.

At execution time, the ZPL runtime evaluates the global bounds of each region, computes the resulting bounding region, and distributes each dimension of that bounding region over the processor grid in a blocked fashion. This distribution is then propagated to each of the program's regions to compute its distribution.

3.12.2 Proposed Support for Region Distribution

In this section, I propose syntax for supporting interacting regions, multiple processor grids, and alternative index distributions. Runtime support for these mechanisms is discussed in Chapter 4.

Support For Interacting Regions

One question that should be raised is whether the compiler should detect interacting regions on the programmer's behalf (using a disjoint set algorithm, for example [CLR92]), or whether the programmer should specify sets of interacting regions and have the compiler "typecheck" the sets to ensure that they are independent. I believe that the second approach is more attractive, and follows the spirit of disallowing automatic variable declarations.

To be more specific, I believe that programmers have a sense of what the "problem spaces" in their algorithms are. For example, "my program uses a large 2D domain, a smaller 2D domain, and some 1D vectors." It therefore makes sense to have users declare

Listing 3.14: Proposed Syntax for Specifying Region Interactions (Domains)

```
region
      R = [1..m, 1..n];
      BigR = [0..m+1, 0..n+1];
      North = [north of R];
      South = [south of R];
      S = [1..x, 1..y];
      V = [1..numelems];
      LongV = [0..numelems+1];
      InnerV = [2..numelems-1];

grid Grid2D: 2;
     Grid1D: 1;

domain Big2D:   Grid2D = R, BigR;
       Small2D: Grid2D = S;
       Vect1D:  Grid1D = V, LongV, InnerV;
```

these problem spaces to the compiler explicitly, naming the groups of regions that they believe describe each domain.

As a proposed syntax, consider Listing 3.14. A number of regions are declared by the programmer. These declarations are followed by two processor grid declarations, `Grid2D` which is 2-dimensional and `Grid1D` which is 1-dimensional. Next, three domains are declared, each of which defines a group of interacting regions that will share the same distribution and processor grid. Prepositional regions belong to the same domain as their base region. Similarly, dynamic regions are grouped in domains based on the regions and arrays with which they interact.

The compiler's job is therefore to typecheck region and array uses to ensure that two regions from different domains are not used in an interacting way. For example, assuming array `A` is declared using region `R` in Listing 3.14, the compiler would emit an error for the following statement which mixes domains:

```
[S] A := 0;
```

To support backwards compatibility with current programs, it can be assumed that if no domains are specified, all of a program's regions would belong to the same domain. Programs that use regions of different ranks would either be forced to declare domains, or could be interpreted as having an implicit domain per rank.

Returning quickly to the grid dimension bucketing example of Listing 3.5, note that while its intended meaning was logically clear to us as readers, there is nothing in the code to indicate that regions `R` and `G` interact—the only interacting regions are `G` and the dynamic region `[value]`. This does not offer ZPL's runtime any indication of how the indices of the dynamic region `[value]` should be distributed across the processors, since `G` has no actual bounds itself. Intuitively, we were relying on `R` to describe the global problem space, and to determine the location of region `[value]`. Using this section's proposal, the intended effect would happen automatically due to the lack of domain declarations. Or, to be more explicit, one could declare:

```
domain D: Grid1D = R, G;
```

Support for Multiple Processor Grids

The proposed grid and domain concepts for grouping regions also provide the user with nice identifiers for specifying each processor grid. In particular, the current command-line support for specifying grids could be expanded to allow domains to be named. For example, the processor grids for the domains of Listing 3.14 could be specified as follows:

```
-gGrid2D=2x8 -gVect1D=16
```

Support for Non-Blocked Distributions

The remaining challenge is how to support distributions of region dimensions other than blocked. I believe that region distributions should be specified at compile-time rather than on the command-line. My reasons for this are three: First, region distribution cues can be valuable for the compiler to generate efficient code. For example, it is easy to generate

Listing 3.15: Proposed Syntax for Declaring Domain Distributions

```
prototype logdist(i: integer; p: uinteger): uinteger;

domain Big2D: Grid2D(blocked,cyclic) = R, BigR;
       Small2D: Grid2D = S;
       Vect1D: Grid1D(logdist) = V, LongV, InnerV;

/* The following distribution gives the first half of the
   indices to the first processor, a quarter to the second,
   an eighth to the third and so on */

procedure logdist(i: integer;      -- the index to be distributed
                  lo: integer;     -- the low bound for this dim
                  hi: integer;     -- the high bound for this dim
                  p: uinteger      -- number of procs in this dim
                 ): uinteger;      -- return the proc i maps to
var
   numelems: integer;              -- number of indices in this dim
   top: integer;                   -- upper bound for current proc
   denom: integer;                 -- 2, 4, 8, etc.
begin
  numelems := hi - lo + 1;         -- calculate number of elements
  top := lo;                       -- set top to low bound
  denom := 2;                      -- start with 1/2 the elements
  for i := 0 to p-1 do             -- loop over processors
    top += numelems/denom;         -- increase top by 1/denom elems
    if (i < top) then              -- if i is less than top...
      return i;                    -- it's on proc i
    end;
    denom *= 2;                    -- otherwise double denominator
  end;
  return p-1;                      -- in case we slip through
end;
```

efficient loops over blocked, cyclic, or block-cyclic dimensions. However, generating a single loop that can efficiently iterate over all three (or any other arbitrary distribution) is much more challenging. Secondly, region distribution decisions tend to be based on an algorithm and not a particular problem size or number of processors. While this claim may not be true for every program, it is sufficiently accurate to require recompilation when a programmer wants to experiment with different distributions. Third, expressing an arbitrary distribution is much easier to do within the context of a ZPL program than it would be on a program's command line.

Thus, I propose the following scheme: For each domain, the user can specify either a standard distribution (block, cyclic, or block-cyclic) or a custom distribution per dimension. Custom distributions correspond to the f_i functions in Section 3.3 and would be expressed using a procedure that accepts an index, a number of processors, p_i, and the low and high bounds of that dimension's indices as its arguments. The procedure returns a processor number from 0 to $p_i - 1$ to indicate where the index resides. If a domain's distribution is not specified, it defaults to a blocked distribution in every dimension. Under this interpretation, all existing ZPL programs are being implemented correctly.

As an example, consider Listing 3.15. This code re-specifies the domains of Listing 3.14 with the following distributions: The `Big2D` domain should be distributed in a blocked manner for its first dimension and a cyclic manner for its second. No distribution is specified for domain `Small2D`, causing it to be blocked in both dimensions. `Vect1D` is distributed using a custom distribution function, `logdist()`, which distributes half as many elements to each successive processor.

As a final note, consider that language support for variations on the standard distributions could be very helpful. In particular, one might want a blocked distribution in which the blocks are not all of equal size, as in the `logdist()` function. This represents information that could be helpful to the compiler, yet is virtually undetectable simply by analyzing a user's custom functions. Some means of specifying such simple variations on standard distributions could benefit both the compiler and the programmer.

3.12.3 The (In-)Completeness of Array Operators

In an ideal world, one could imagine that ZPL would give programmers the ability to specify their own array operators. However, ZPL's array operators vary greatly in syntax and effect from one another—much more so than its region operators, for example. For this reason, it is difficult to imagine a clean scheme for allowing users to specify custom array operators, especially given that each would tend to require a custom communication implementation. Otherwise, user array operators would simply be syntactic sugar for specific applications of the remap operator.

For this reason, it is tempting to argue that ZPL's array operators represent a complete, fundamental basis set of operators that a programmer might want to use. Unfortunately, this seems a bit naive, and in any case it is beyond my abilities to prove.

However, there is a simple and quick argument that ZPL's array operators form a "good" basis set. In particular, the fact that there is a one-to-one correspondence between the array operators and the different communication styles provided by libraries such as MPI, PVM, SHMEM, *etc.* implies that ZPL's operators match the communication styles that practical libraries have identified as being fundamental to parallel computing. I believe that a new array operator should be considered only if (1) it involves a useful communication style that is not succinctly described by an existing ZPL array operator, and (2) it has an implementation that is more efficient than its remap equivalent. Judging from the styles supported by current libraries, the existing array operators seem to match current user requirements.

3.12.4 Reconsidering the WYSIWYG Performance Model

This section reconsiders the wisdom and usefulness of ZPL's performance model.

Does WYSIWYG Thwart Compiler Optimization?

One of the main arguments against ZPL's WYSIWYG performance model is that telling the user what to expect from a program limits the amount of optimization that a compiler can do. In my interpretation of ZPL's performance model, this could be true in the extreme case, but not in general. In particular, I do not read ZPL's performance model as saying "a point-to-point communication shall be invoked for every @-reference" so much as "@-references may refer to remote values, and point-to-point style communication is sufficient for transferring those values to local memory." That is, the performance model does not dictate an absolute cause and effect, but rather describes the data access pattern and a likely solution. This does not strike me as being a barrier to optimization.

As evidence, the ZPL compiler performs several optimizations on the communication used to implement @-references, including message vectorization, latency hiding, redundancy elimination, and combining of messages [CS97, Cho99]. None of these break ZPL's performance model, because the array values are still organized in the same way, and the non-local references are still made. The communication implementing a program's @-references is simply being performed in a more subtle way than a naive reading of the performance model might indicate.

On the other hand, there are other optimizations that should be considered a direct violation of ZPL's performance model—for example, distributing an array in a skewed, non-grid-aligned manner. Though such a distribution might improve performance for certain programs, I am not in favor of such optimizations, as I think that languages must have some invariants upon which a user can rely. For example, as a C programmer, it is useful to know that its arrays are laid out in row-major order. While I can imagine compiler optimizations that use a different order, I would worry that such optimizations would disrupt programmers' views of what was happening to the extent that it thwarts their ability to evaluate a program's implementation. The same holds true in ZPL.

ZPL's Invisible Performance Factors

Another potential trouble with the WYSIWYG performance model is that not everything is visible to the programmer. This I believe to be a real problem. For example, savvy programmers may want to know how the compiler is inserting and optimizing communication, so that they might reorganize their code to enable additional optimizations. This information is not made available to them in the language.

As a second example, ZPL compilers often have to insert temporary arrays into a user's code in order to store the results of complex expressions like remap operators. These temporary arrays consume precious memory resources and result in memory reference overheads, yet are invisible to programmers who lack an intimate knowledge of their ZPL compiler. Such effects seem at odds with the WYSIWYG principle.

As a final example, ZPL contains an *array contraction* optimization that replaces array references with scalars in order to reduce memory usage and overheads [LLS98, Lew00]. However, the user has no visual cues in the program to indicate when contraction is possible or whether the optimization fires as expected.

All of these cases involve parallel overheads that programmers might like to have more information about at the source level. However, ZPL's global view does not allow the language to expose such information. These cases also represent code that the programmer typically cannot hand-optimize using a "back door" in the language to ensure a particular implementation. For example, programmers have no means of specifying communication or the contraction of a variable by hand, if they have some insight as to what would be best for the program.

This seems like an obvious area for continued research. One approach might be to supply edit-time tools that are integrated with the compiler and can indicate such invisible factors as communication placement or unexpected array storage. Another approach would be to develop lower-level concepts, perhaps using grid dimensions as a foundation, that would give users a means of specifying lower-level behavior when the compiler fails (as indicated by compiler feedback or profiling tools).

While ZPL's performance model is not perfect, it does provide the user with a reasonably accurate low-level interpretation of their code without sacrificing high-level, global-view semantics. This makes ZPL unique among current parallel programming languages, and it is a direct result of the language's use of regions.

3.12.5 Unifying Parallel and Indexed Arrays

There has been some attempt in the ZPL project to do away with ZPL's indexed arrays by unifying them with parallel arrays. The motivations for this are understandable: having two distinct array types can be somewhat confusing to users. In addition, many of the parallel array operators would be useful if supported for indexed arrays. Since they are not, explicit loops and indexing must be used instead. The proposal would be to declare all arrays using regions, and then to give the user the ability to mark each dimension as being distributed or not.

The problem with this approach is that it complicates certain aspects of regions and array operators. For example, imagine that a ZPL code contains the following 2D parallel array declarations:

```
var A: [1..n, 1..n] integer;
    B: [1..n, 1..n] array [1..3] of integer;
    C: array [1..3] of [1..n, 1..n] integer;
```

Using these declarations, the same regions and directions can be applied to all three arrays, since they are each declared using a 2D region. For example, a simple operation might appear as follows:

```
[1..n, 1..n] A@east := B@east[i] + C[j]@east;
```

If parallel and indexed dimensions were unified into the region concept, the declarations would appear as follows:

```
var A: [1..n, 1..n] of integer;
    B: [1..n, 1..n, 1..3] of integer;
    C: [1..3, 1..n, 1..n] of integer;
```

This approach has several disadvantages. First, my array assignment would need to be covered by 2D and 3D regions to provide indices for all three arrays. Furthermore, three different versions of direction `east` would have to be declared since the logical dimension that `east` was modifying is now different in each array. Third, `B` and `C` would require different 3D regions to describe the correct elements in each of their references. Finally, the programmer would need some mechanism to indicate to the compiler which dimensions of A, B, and C it should align. All of these problems obfuscate what was originally a very simple statement.

The other approach would be to allow the user to declare parallel arrays of parallel arrays, resulting in declarations similar to the original ones. This has its own set of problems, however. For example, each nested array expression would require multiple covering regions to specify its indices. In addition, nested arrays would make it ambiguous which array an operator was being applied to.

All of these problems might be solvable given enough time and imagination. However, they also motivate a re-evaluation of the existing syntax to determine whether it is all that bad. In my opinion, it is not. ZPL supplies users with traditional sequential arrays with familiar access styles. It also supplies them with parallel arrays with unfamiliar operators. This forces users to think about parallel arrays differently than sequential arrays. And they should. It also gives users a clear syntactic indication of when communication is required and when it is not. In short, I believe parallel arrays should look different because they are different.

This is not to say that additional support for indexed arrays is unwarranted. In particular, ZPL could easily support more advanced indexed array accesses such as array slicing or vector subscripting. Both of these mechanisms would eliminate additional loops from existing ZPL codes.

3.13 Summary

This chapter described the parallel interpretation of regions, and the effect that their distribution has when interpreting ZPL programs. Distributing interacting regions in a grid-aligned manner supports the concept of a WYSIWYG performance model in ZPL, allowing programmers to evaluate their codes' communication and concurrency. As a result, regions impart a unique mix of high-level semantics and low-level implementation information. In contrast, other programming languages either require users to program at the low-level using a local-view, or they abstract away important details like communication to provide a high-level global-view. Regions allow ZPL to achieve an ideal balance between these two approaches.

The next chapter describes the implementation of regions in ZPL.

Chapter 4

IMPLEMENTING REGIONS AND ZPL

This chapter describes the implementation of ZPL, focusing on its runtime data structures and library interfaces. This discussion describes the current implementation of ZPL as accurately as possible, though in many cases the identifiers are changed for reasons of clarity.

The chapter begins by providing an overview of the compilation and execution models used to implement ZPL programs. It then describes the data structures that represent ZPL concepts at runtime, including regions, arrays, processor grids, data distributions, and directions. Section 4.3 introduces several of the code idioms used to iterate over regions and access arrays in the compiler-generated code and runtime libraries. Section 4.4 summarizes the interface for ZPL's runtime libraries, detailing its guiding philosophy for communication routines and the application of this philosophy in its point-to-point communication interface. Section 4.5 provides a high-level overview of ZPL's compilation. Related work is described in Section 4.6, and proposed support for alternative data distributions is discussed in Section 4.7. Parts of this chapter elaborate on previously published results [CCS97].

4.1 Models of ZPL's Compilation and Execution

This section gives a brief overview of ZPL's compilation and execution models in order to set the stage for the rest of the chapter.

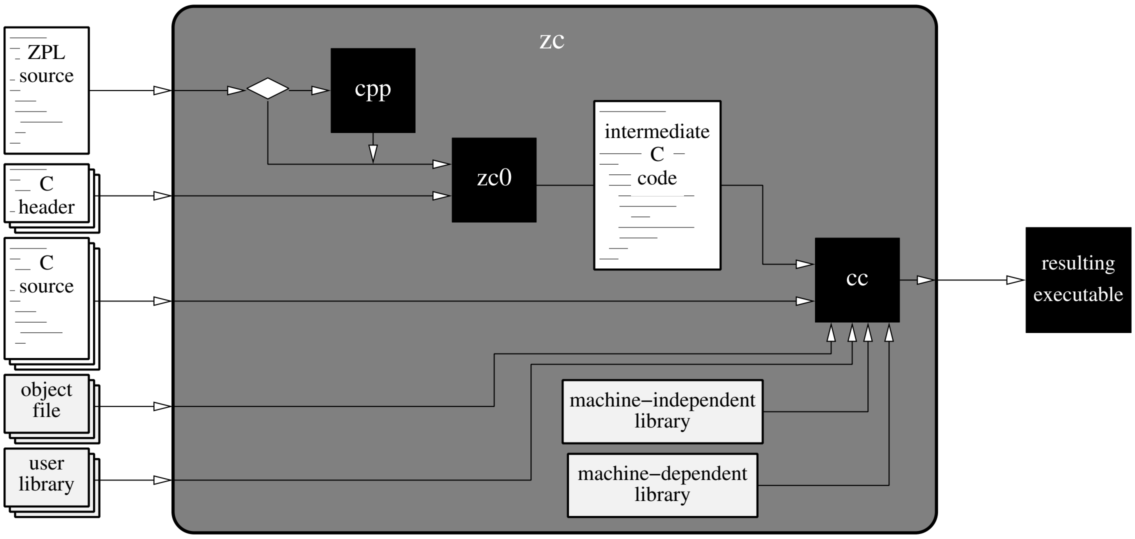


Figure 4.1: ZPL's Compilation Model. This diagram shows a loose architecture for the ZPL compiler. Users can supply a ZPL source file, C header files, C source files, object files, and libraries to the zc compiler. The ZPL source file is optionally run through cpp and then fed to zc0, which translates it to an intermediate representation implemented using ANSI C with calls to the ZPL runtime libraries. Directives to include the user-supplied header files are also inserted into the intermediate code to provide prototypes for any external routines implemented in the user's code. The intermediate C, user C files, object files, and libraries are then fed to cc, along with ZPL's machine-independent and machine-dependent runtime libraries. The result is an executable that implements the user's ZPL program.

4.1.1 Compilation Model

ZPL's compilation model is heavily influenced by the language's goal of portability. Portability is achieved in ZPL by compiling to ANSI C and linking in library routines to implement machine-specific features. This model is illustrated in Figure 4.1.

The ZPL compiler is named *zc*. It produces an executable from source files written in ZPL and C, object files, and libraries. The current implementation assumes that only a single ZPL source file will be specified, though it may be defined in terms of other ZPL files using preprocessor `#include` directives. A program called *zc0* forms the heart of zc by performing the ZPL to C translation step. The resulting code is then fed into a machine's native C compiler, along with ZPL's runtime libraries and any other C files, object files, or libraries specified by the programmer on the command line. This link step results in an executable that implements the ZPL program.

ZPL's runtime libraries are divided into two layers. The first is a machine-independent layer which contains routines that are large enough, complicated enough, or used frequently enough to warrant creating a library routine rather than having the compiler generate the code for each program. Such routines include the parsing of the executable's command line, the creation of dynamic regions, file I/O routines, memory allocation, and high-level communication routines. This layer is written as generically as possible to ensure that it will execute correctly on all platforms.

The second layer of the libraries is a machine-dependent layer which implements low-level routines that may vary from machine to machine. This layer can also override routines or data structures from the machine-independent layer to meet a particular platform's requirements. Most of the routines in the machine-dependent layer implement data transfer routines, since details of communication vary so much from platform to platform. Versions of the machine-dependent libraries have been developed for MPI, PVM, SHMEM, and sequential platforms[1]. Future versions are planned for threaded shared memory platforms.

[1] An NX implementation also existed at one time, but has fallen out-of-date along with the library itself.

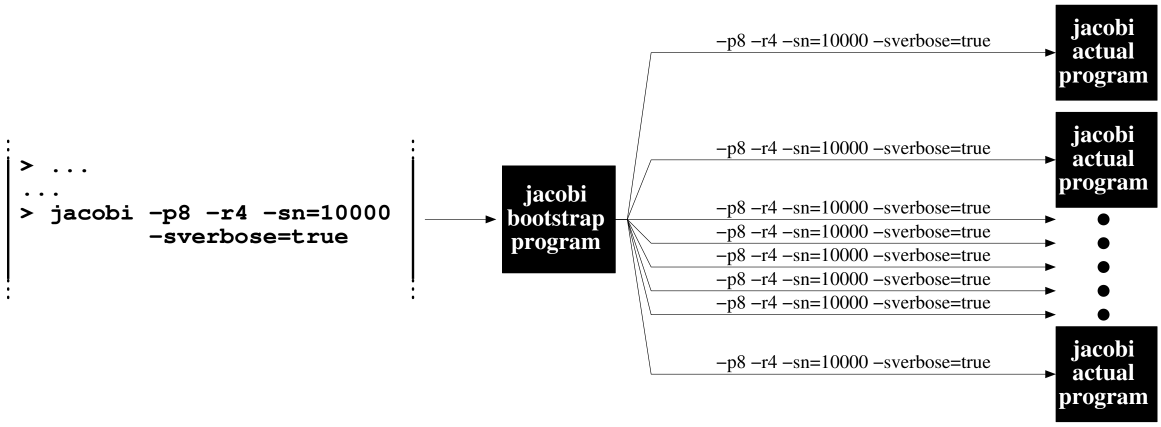


Figure 4.2: ZPL's Execution Model. The user executes a ZPL program, specifying the processor grid(s) and overriding configuration variables on the command line. This invokes the bootstrap program which determines the number of processors requested and spawns a corresponding number of copies of the actual program, passing the command line to each.

4.1.2 Execution Model

As mentioned previously, ZPL users can use command-line options to specify processor grids and to override the default values of their configuration variables. For example, a command line for the Jacobi implementation in Section 2.15.1 might appear as follows:

```
jacobi -p8 -r4 -sn=10000 -sverbose=true
```

The effect of this command line is to run on a 4×2 processor grid using a problem size of 10,000 and verbose output.

Compiling a ZPL program for a parallel platform generates two executables, known as the *bootstrap program* and the *implementing program*. Different platforms typically have diverse methods of executing multiple cooperating versions of an executable. The bootstrap program is designed to hide these details from the user by parsing the executable's command line, determining the number of processors required, and then spawning that many copies of the implementing program using the machine-specific mechanism. The original command line is passed to each of the copies so that they can set up their processor grid views and configuration variables correctly. This results in an execution environment that is consistent from machine to machine, sheltering users from much of the overhead of learning how to spawn processes on each platform. See Figure 4.2 for an illustration.

Conceptually, ZPL uses a single thread of control to implement each processor specified by the user. In reality, these threads of control may be implemented using cooperating threads or processes, and the number of physical processors may be smaller than the logical set. This discussion will reinforce the conceptual view by using the term “processor” to refer to a single thread of control and logical processor, regardless of its actual implementation.

4.2 Runtime Descriptors

This section describes the runtime data structures that implement ZPL, referred to here as *descriptors*. Each descriptor corresponds to one of the major concepts in the language and contains the data and functions used to implement that concept. Descriptors are implemented using C structures to store their defining values and a pointer to the structure to support efficient parameter passing. Every processor stores a private copy of each descriptor, allowing its values to differ from those of other processors.

4.2.1 Descriptor Implementation Notes

This section covers some basics related to the implementation of ZPL’s runtime descriptors.

Helper Data Structures

The descriptors in the following sections make use of some helper data structures in order to store related fields close together in memory. These structures store dimension bounds and sequence descriptors and are defined in Listing 4.1. Additional structures are used to store fields that are less obviously related adjacent in memory for purposes of cache line utilization. For clarity, this discussion will ignore such structures.

Listing 4.1: Descriptor Helper Structures

```
/* a low/high bound pair */
typedef struct _lohi {
  int lo;
  int hi;
} lohi;

/* a low/high bound pair with stride */
typedef struct _lohistr {
  int lo;
  int hi;
  int str;
} lohistr;

/* a sequence descriptor */
typedef struct _seq {
  int lo;
  int hi;
  int str;
  int align;
} seq;
```

Array Fields of Non-Constant Size

Many of the descriptors in this chapter contain arrays whose sizes are defined by a region's rank. Since C structures cannot contain non-constant sized arrays, this is done for clarity and does not represent the types used in the actual implementation. In reality, the implementor has three obvious choices for declaring such fields, described here.

The first is to store a pointer in the structure to an array that will be dynamically allocated at runtime. This is the most flexible solution, but involves dynamic memory and an extra pointer dereference to access such fields.

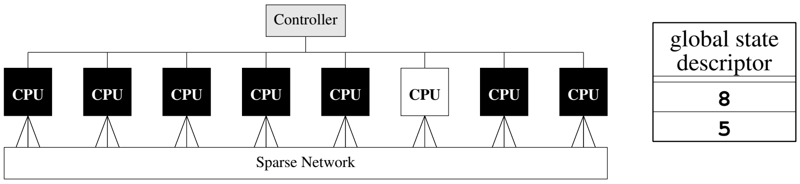
A second option is to use fixed-sized arrays by imposing an arbitrary maximum on the rank of the descriptors. This is the least flexible approach and it obviously breaks the *zero-one-infinity principle* [Mac87]. In practice, however, even a modest bound is sufficient to handle the vast majority of programs. Furthermore, fixing such a bound simplifies the implementation. These reasons make this implementation tempting in spite of its inflexibility.

The third option is to create a helper structure that contains all of the array element types as its fields. A descriptor structure is then declared for each rank used by a program, storing an array of corresponding size as its last field. Since the prefixes of the structures are all identical, library code can treat these structures uniformly regardless of rank (provided that the C compiler does not reorder the fields in any way). Such structures can be declared using C preprocessor macros for simplicity. The chief disadvantage of this approach is that it is somewhat nonintuitive and may spread related values out in memory further than ideal in terms of cache utilization.

Listing 4.2 illustrates each of these approaches as well as the conceptual array that they are being used to implement. Our implementation takes the second approach using a maximum rank of 6. Although this approach initially seemed painfully inflexible, its simplicity quickly became seductive. In the long run, no users have ever been limited by this decision, and ZPL's implementation and research have benefitted greatly from the choice.

Listing 4.2: Implementing Structures with Non-Constant Array Fields

```
/* Conceptual structure */
typedef struct _concept {
  int numdims;
  ...
  int a[numdims];
  double b[numdims];
  lohi c[numdims];
} concept;

/* Approach one: dynamic arrays */
typedef struct _dynarr_struct {
  int numdims;
  ...
  int* a;
  double* b;
  lohi* c;
} dynarr_struct;

/* Approach two: fixed-size arrays */
typedef struct _constarr_struct {
  int numdims;
  ...
  int a[MAX_DIMS];
  double b[MAX_DIMS];
  lohi c[MAX_DIMS];
} constarr_struct;

/* Approach three: struct-per-dim */
typedef struct _abc_struct {
  int a;
  double b;
  lohi c;
} abc_struct;

#define FLEXARR_STRUCT_OF_RANK(rank) \
  typedef struct _flexarr_struct##rank { \
    int numdims; ...\
    abc_struct abc[rank]; \
  } flexarr_struct##rank;

FLEXARR_STRUCT_OF_RANK(2) my2Dstruct;  /* define a 2d version */
```




```
typedef struct _glob {
  int numprocs;
  int index;
} *glob;
```

Figure 4.3: The Global State Descriptor. This figure shows the CTA machine model and the global state descriptor that describes the processor shown in white. The `numprocs` field describes the number of processors in use by the program, and the `index` field describes the unique ID of the processor storing the descriptor.

4.2.2 Global State Descriptor

The *global state descriptor* stores information relating to a program's global properties. In its simplest form, this descriptor has two data values: the total number of processors, p (stored as `numprocs`), and the unique index of the local processor ($\texttt{index} = 0 \ldots p-1$). See Figure 4.3 for an illustration.

Each copy of the implementing program determines the number of processors by reading the processor grid specifications from the command line. If a single grid is specified, `numprocs` is simply set to the number of processors in that grid. If multiple grids are specified, the one that uses the largest number of processors determines the value of `numprocs`.

Most parallel platforms provide cooperating processes with a means of querying a unique index value from 0 to $p - 1$. If a platform does not provide such a mechanism, it can be supplied to the implementing programs by the bootstrap program on their command lines. In either case, this value serves as the processor's `index`.

Some platforms require additional global bookkeeping information that can be implemented in the state vector or using global variables. For example, in PVM, each processor has a unique *task identifier* (TID) that describes the process within the PVM environment, using a unique value other than 0 to $p - 1$. The TID of each processor must be determined and stored in an array for use throughout the libraries when communicating with other processors.

4.2.3 The Processor Grid Descriptor

The *processor grid descriptor* is used to represent a single view of the processor set as a regular grid. It contains fields that describe the number of dimensions in the processor grid (`numdims`), the number of processors contained by the processor grid (`numgridprocs`), and the unique index of the current processor (`gridindex`). ZPL typically uses the first `numgridprocs` processors from the global set to implement a given processor grid, causing a processor's `gridindex` to be set equal to its `index` in the global state descriptor. Processors that do not participate in a particular grid are assigned a `gridindex` of -1 in their descriptors.

In addition, the number of processors in each grid dimension is stored using an integer array, `procsperdim[]`. These values are identical to the values $p_1 \times p_2 \times \ldots \times p_d$ used to describe processor grids in Section 3.2. Each processor also stores its logical coordinates in the processor grid using a second integer array, `myloc[]`. For a given dimension i, `myloc[i]` $= 0 \ldots$`procsperdim[i]`-1. Processors are ordered in row-major order for each grid. Thus, for a 3-dimensional grid:

$$\begin{aligned}\texttt{gridindex} = \; & \texttt{myloc[0]}\cdot\texttt{procsperdim[1]}\cdot\texttt{procsperdim[2]} \\ & + \texttt{myloc[1]}\cdot\texttt{procsperdim[2]} \\ & + \texttt{myloc[2]}\end{aligned}$$

Processor Grid Slices

Processor grids are also organized into *processor grid slices* (or *slices* for short). Each grid slice describes a subset of processors that either have the same index in a given dimension of the grid or do not. For example, in a 2-dimensional grid, processors would be organized into row slices and column slices. Since regions are distributed in a grid-aligned manner, processor grid slices are useful for referring to the subset of processors participating in

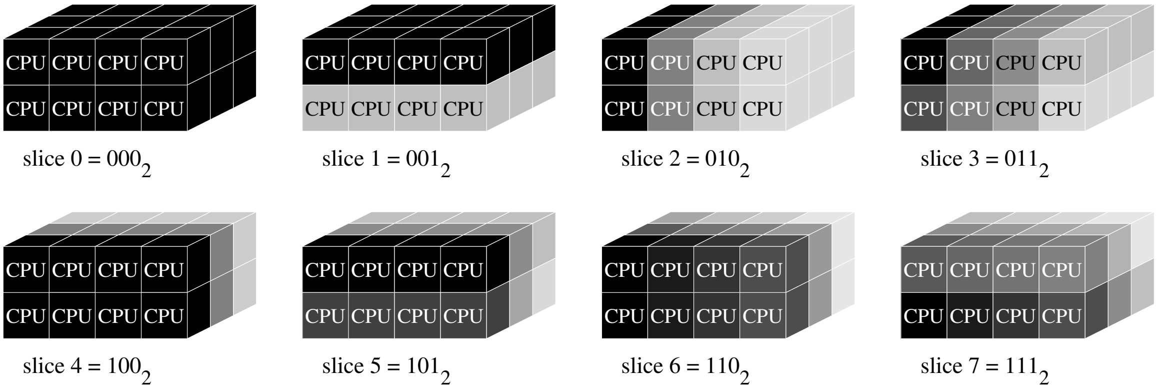


Figure 4.4: Processor Grid Slices in 3D. This figure illustrates the 8 ways of slicing a 3D processor grid and their corresponding slice numbers.

the broadcast and reduction routines used to implement ZPL's flood and partial reduction operators.

A d-dimensional processor grid has 2^d slices, numbered $0 \ldots 2^d - 1$. Each slice is interpreted by converting its number to a binary representation and associating a bit with each processor grid dimension. In particular, the lowest order bit represents the first dimension of the processor grid, the next bit represents the second dimension, and so on. If a given dimension holds the value "0," it implies that all processors in that dimension are involved in the slice. The value "1" indicates that the processor grid is sliced in that dimension. This means that in any grid, slice number 0 refers to the complete set of processors in the grid, whereas slice number $2^d - 1$ describes a set of p slices, each of which contains a single processor.

As a more interesting example, in a 2-dimensional grid, slice 1 (01_2) refers to the processor grid's rows, while slice 2 (10_2) refers to its columns. Thus, slices 1 and 2 might be used to implement the following statements, respectively:

```
[1..n, *] A := >>[ , k] B;   -- broadcast within slice 1
[1, 1..n] C := +<<[R] D;     -- reduce within slice 2
```

See Figure 4.4 for an illustration of the eight ways to slice a 3D processor grid.

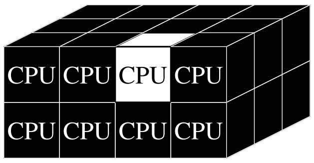
Listing 4.3: A C Routine to Determine if a Grid Slice is Distributed

```
int slice_distributed(procgrid grid, int slicenum) {
  int dim;
  int dimbit = 0x1;

  /* loop over the processor grid's dimensions... */
  for (dim=0; dim<grid->numdims; dim++) {
    /* if the slice spans this dim and the grid isn't flat... */
    if ((slicenum & dimbit == 0) &&
        (grid->numprocs[dim] > 1)) {
      return 1;       /* ...then the dimension is distributed */
    }
    dimbit = dimbit << 1;  /* ...else, continue with next dim */
  }
  return 0;         /* if we get here, no dim was distributed */
}
```

Any given processor slice has a number of properties that might be of interest to the ZPL runtime libraries. These include:

- Is slice s distributed across multiple processors?
- What is my/processor p's logical number within slice s?
- What is the lowest-/highest-numbered processor in slice s?
- In my instance of slice s, which processor stores index **i**?

All of these properties are fairly trivial to compute using a loop that checks the bits of the binary slice value by iterating over the processor grid's dimensions. For instance, the code in Listing 4.3 computes whether or not a particular processor grid slice is distributed across multiple processors. This information can be used by library calls to determine whether communication is necessary. For example, when using a $1 \times p$ grid, this routine would say that slice 1 is distributed, but that slice 2 is not. Thus, the partial reduction shown previously would be completely local, requiring no call to the reduction routine in the runtime libraries.



processor grid descriptor
3
24
2
{2, 4, 3}
{0, 2, 0}
8
{1,1,1,1, 1,1,1,1}

```
typedef struct _grid {
  int numdims;
  int gridprocs;
  int gridindex;
  int procsperdim[numdims];
  int myloc[numdims];
  int numslices;
  int slice_distributed[numslices];
} *grid;
```

Figure 4.5: The Processor Grid Descriptor. A processor grid and the grid descriptor for the processor shown in white. The fields are as follows: `numdims`—the rank of the processor grid; `gridprocs`—the number of processors in the grid; `gridindex`—the ID of the processor storing the descriptor; `procsperdim[]`—the grid's dimensions; `myloc[]`—the logical location of the processor storing the descriptor; `numslices`—the number of slices in the grid; `slice_distributed[]`—an array storing whether or not each slice is distributed across multiple processors.

The processor grid descriptor stores the number of slices in the grid, as well as the answers to any of the questions above that are considered worthwhile to precompute and store. For example, the runtime libraries check whether slices are distributed fairly often, so this information is precomputed and stored in an array indexed by slice number to avoid recomputing it every time.

Certain platforms support user-specified processor groups to describe a subset of the processors. For example, MPI's *communicators* allow processors to be organized into disjoint subsets [Mes94]. Such mechanisms can be used to arrange processors in their respective slices. As a result, the machine-independent libraries often play a role in the initialization of a processor grid by defining slices for use by the communication routines.

This concludes the description of the processor grid descriptor. Figure 4.5 shows an illustration summarizing its structure.

4.2.4 The Distribution Descriptor

Distribution descriptors are used to represent the mapping of indices to a processor grid. They correspond to the domain declarations described in Section 3.12.2. As such, they are used to distribute the indices from a group of interacting regions to their common processor grid view. Distribution descriptors are also useful for finding the location of a particular index as required by the remap operator or a dynamic region.

Distribution descriptors contain a reference to the processor grid descriptor that represents the target of the distribution (`procgrid`). They also store the bounding indices of the regions that are being distributed (`bound[].lo/hi`), since these bounds are typically used to compute a distribution. Although the distribution's rank (`numdims`) could be looked up in its processor grid descriptor, in practice this value is replicated in the distribution descriptor because of its small size and to avoid the lookup. The final component of the descriptor is a description of the distribution itself.

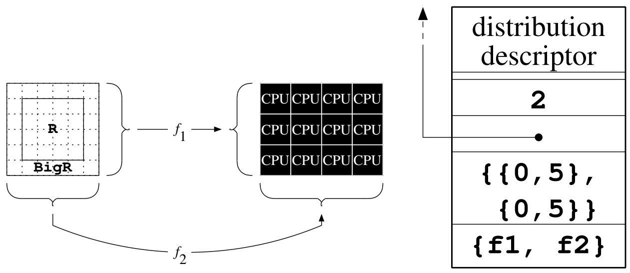
Since the current implementation of ZPL only supports blocked decompositions, it does not utilize distribution descriptors that support arbitrary distributions. Rather, the values that define the blocked distribution are stored for each dimension of the index space.

Continuing with the arbitrary distribution proposal of Section 3.12.2, it is easy to conceive of storing an array of function pointers within the distribution descriptor that describe the distribution of each dimension (`map[]`). The runtime routines could then query the location of index ind in dimension i using a call as follows:

```
proc = (dist->map[i])(ind, dist->bound[i].lo,
                           dist->bound[i].hi,
                           dist->procgrid->procsperdim[i]);
```

Figure 4.6 shows an illustration of the distribution descriptor.

It seems likely that more advanced data distribution schemes will want to store some amount of state information to describe the distribution and provide fast lookups for a particular index. For example, consider an irregularly blocked pattern whose distribution is not defined using simple math. In such a case, the implementor might want to store a simple




```
typedef struct _dist {
  int numdims;
  grid procgrid;
  lohi bound[numdims];
  int (* map[numdims])(...);
} *dist;
```

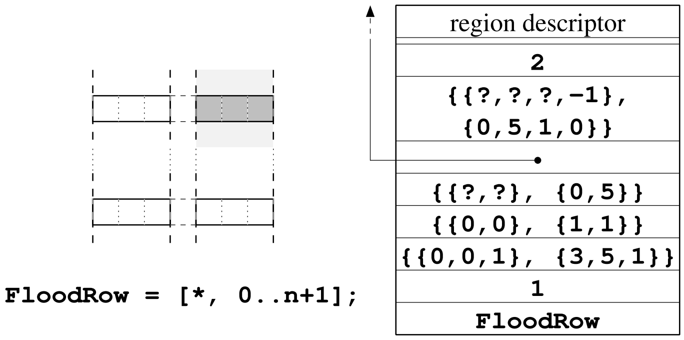
Figure 4.6: The Distribution Descriptor. The distribution on the left is encoded in the distribution descriptor in the center. The fields are as follows: `numdims`—the rank of the distribution; `procgrid`—a pointer to the processor grid descriptor to which this distribution maps; `bound[]`—an array of low/hi bounds that describe the bounding box of the index space being distributed; `map[]`—an array of pointers to mapping functions (or possibly sentinel functions representing built-in distributions).

array of values to indicate the first index owned by each processor. This would support the use of binary searches to locate a particular index quickly. Such data would need to be associated with a distribution descriptor's dimensions rather than the distribution function itself in order to allow a single function to be reused with multiple index sets. The question is therefore how to store and initialize such arbitrary data structures so that they can be passed into the distribution function. This topic requires further study.

The remainder of this chapter's technical description will assume that regions are distributed using a blocked distribution (regular or irregular). The topic of supporting alternative distributions will be reconsidered in Section 4.7.

4.2.5 The Region Descriptor

The *region descriptor* is used to store information for each region in a ZPL program. As such, it is one of the most fundamental descriptors in the implementation, describing the index sets used for array definitions and the language's implementing loop nests. Figure 4.7 shows an illustration of a region descriptor in ZPL.




```
typedef struct _region {
  int numdims;
  seq seqdesc[numdims];
  dist distribution;
  lohi globbbound[numdims];
  lohi procloc[numdims];
  lohistr locbound[numdims];
  bool iown;
  char* name;
} *region;
```

Figure 4.7: The Region Descriptor. This figure shows a flooded row region and the region descriptor for the processor owning the shaded portion of its indices. The fields are as follows: `numdims`—the rank of the region; `seqdesc`—the region's defining sequence descriptors; `distribution`—a reference to the region's distribution descriptor; `globbound[]`—the lowest and highest indices represented by the region in each dimension; `procloc[]`—the processor coordinates owning these lowest and highest indices; `locbound[]`—the bounds and stride that define the processor's local indices; `iown`—a boolean flag indicating whether or not the processor owns a piece of the region; `name`—the name of the region, for debugging purposes.

Global-view Fields

To describe a region's global view, its descriptor stores the region's rank (`numdims`), and an array of sequence descriptors (`seqdesc[]`). By convention, the alignment value of each sequence descriptor is stored as a value from 0 to $s - 1$, where s is its corresponding stride value. Flood and grid dimensions are indicated using sentinel alignments of -1 and -2, respectively, since these are illegal alignments for a traditional dimension. The other sequence descriptor fields are never referenced for a grid or flood dimension, and therefore may contain arbitrary values.

Distribution Fields

The global distribution of the region is captured in the region descriptor by storing a reference to its distribution descriptor (`distribution`) as well as the *actual* low and high global indices for each dimension (`globbound[]`). Note that these bounds describe the lowest and highest indices that are actually defined by a sequence descriptor rather than the descriptor's low and high bounds. For example, the global bounds of descriptor $(1, 6, 2, 1)$ would be 1 and 5 since it only describes odd elements. These bounding indices are used at runtime to dynamically compare the dimensions in the source and destination regions of a flood or reduction operator. They are also used to determine the bounds of an array I/O operation, to compute the alignment of strided regions, *etc*.

In addition, the region descriptor stores the processor grid coordinates where the region's lowest and highest indices are mapped (`procloc[]`). Although these values could be computed from the region's sequence descriptors and distribution descriptor, the current implementation caches them in the region descriptors for convenience and to avoid excessive use of the modulus operator.

Local-view Fields

Each processor's local view of a region is represented in its descriptor by storing low, high, and stride values for each dimension (`locbound[]`). These bounds represent the lowest and highest indices in the dimension that map to the processor. They can therefore be used as loop bounds when iterating over a processor's local portion of the dimension in either ascending or descending order. If a processor does not own any indices in a certain dimension of a region, it stores a low bound that exceeds its high bound so that any loops over the dimension will be degenerate. In addition, a boolean value is stored to indicate whether or not the processor owns a block of the region's indices (`iown`). Although this could be computed by looking at the bounds of each dimension, it is stored explicitly as a quick means of checking a processor's involvement in a computation over the region.

Note that a processor's local stride values are identical to the strides stored in its sequence descriptors. They are stored again with its local bounds for improved cache utilization when looping over the processor's indices (the most common use of a region descriptor). In fact, the current implementation does not contain a stride field in the sequence descriptors to avoid replicating the value. Since sequence descriptors are only used when defining regions, the separation in memory is less important in this case.

A flood or grid dimension is represented using a stride of 1 and an arbitrary index owned by the processor as its low and high bounds (typically the processor's lowest local index). This allows the processor to loop over the single implementing index of the dimension as it would any other dimension. Since ZPL's semantics prevent the user from directly referring to this index, its implementing value is inconsequential.

Miscellaneous Fields

The current implementation of regions also uses a string to store the name of the region for debugging purposes (`name`). This is never required to implement a ZPL program, but has made development much easier at times.

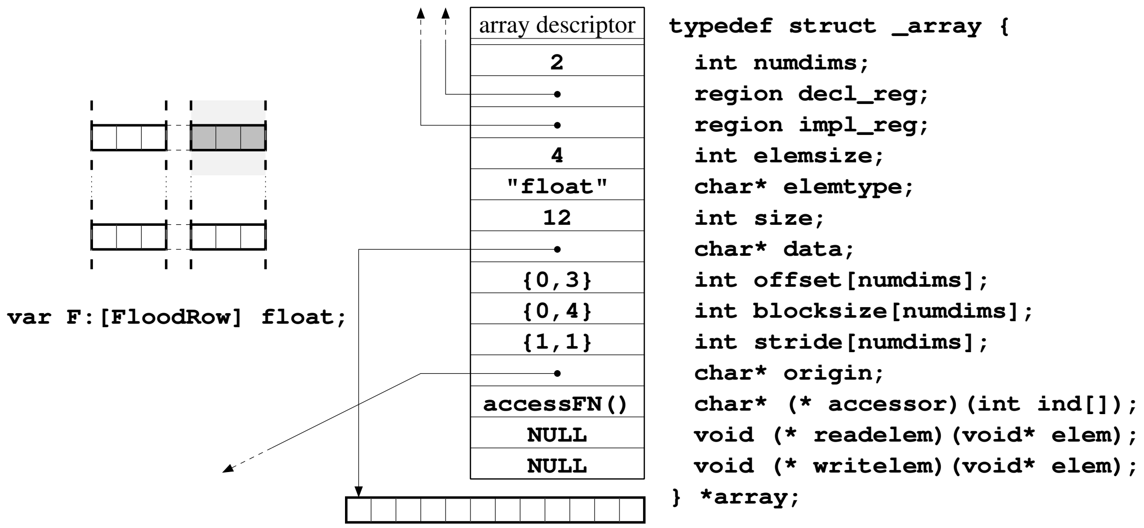


Figure 4.8: The Parallel Array Descriptor. This figure shows the parallel implementation of a flooded row array, and the array descriptor for the processor owning the shaded portion of the array. The fields are as follows: `numdims`—the rank of the array; `decl_reg`—the region used to declare the array; `impl_reg`—the region describing the array's size once implicit storage is added to it; `elemsize`—the size of the array's element type in bytes; `elemtype`—a string describing the array's element type; `size`—the size of the data buffer allocated for the array's data values; `data`—a pointer to the array's data buffer; `offset[]`—offsets for making global indices zero-based when accessing the array; `blocksize[]`—multiplicative factors used to scale each dimension of an array's access; `stride[]`—divisors used to stride an array's dimensions; `origin`—a special pointer used to optimize array accesses; `accessor()`—a function used to access the array element at a given index; `readelem()`/`writelem()`—functions to read or write the elements of an array, used primary for arrays with non-scalar element types.

4.2.6 The Parallel Array Descriptor

ZPL's *parallel array descriptor* (*array descriptor* for short) is used to represent a processor's view of a parallel array. Since parallel arrays are defined using regions, much of the high-level information about the array is stored in its defining region's descriptor. This leaves the array descriptor with the task of describing the memory layout used to implement an array. Figure 4.8 shows an array descriptor in ZPL.

High-level Fields

In addition to storing a reference to its declaring region, (`decl_reg`), each array descriptor stores its rank (`numdims`), its element size in bytes (`elemsize`), and a string holding the name of the element type for use by debuggers (`elemtype`) [WA00, Wat00]. Because ZPL currently supports the implicit extension of an array's storage (Section 2.18.5), the array descriptor also stores a reference to a region that defines its actual size, including any implicit storage (`impl_reg`). Arrays that require no implicit storage store a second reference to their defining region in this field.

Data Layout Fields

The array's data is stored in row-major order using a single block of memory whose size is nominally equal to the number of local indices in the defining region times the element size. This size is explicitly stored in the descriptor for accounting purposes and as a debugging tool (`size`). The data block itself is dynamically allocated and pointed to using a character pointer to support pointer arithmetic (`data`).

The rest of the array descriptor's data layout fields provide a means of translating logical indices to memory addresses. Since an array's indices are not necessarily 0-based, each dimension must store an offset to shift indices to 0 (`offset[]`). In addition, each dimension stores a multiplicative factor used to describe the distance between consecutive elements (`blocksize[]`) and a stride used to represent strided dimensions (`stride[]`).

While it is tempting to combine the `blocksize[]` and `stride[]` fields into a single multiplicative factor, the fact that their ratio may be less than 1 requires the value to be fractional. For example, an array of characters strided by 8 would require a blocksize of `1/8`. Rather than incur the overhead and imprecision involved in using a floating point multiplicative factor, the current implementation uses two integer values, optimizing them away when possible. This optimization uses an additional character pointer field (`origin`) whose use is described further in Section 4.3.2.

Arrays with flooded or grid dimensions are implemented using a multiplicative factor of 0 to collapse any index in that dimension down to its single set of implementing values. Note that this is also the justification for explicitly storing the base element size rather than simply referring to `blocksize[numdims-1]`. In cases where the outermost dimension is a flood or grid dimension, this multiplier will be set to 0 though the element size is obviously non-zero.

Array Operation Fields

The remaining fields of the array descriptor are function pointers used to store other operations relating to the array. One of these function pointers represents the array's *accessor function*, a routine that takes an array index as input and returns a pointer to the address of the array element (`accessor()`). Accessor routines are a means of getting at an array's data in a generic but efficient manner, and are discussed further in Section 4.3.2.

The other two function pointers are used to read or write an element of the array using the console or a file and can be set by the user in a ZPL program (`readelem()`, `writelem()`). These fields are provided to give the user high-level control for specifying I/O on non-scalar array elements. When performing I/O on an parallel array, the appropriate function is applied to each element one at a time, if it is defined. If it is not, scalar array values are handled using default scalar formatting and non-scalar values result in an error message at runtime.

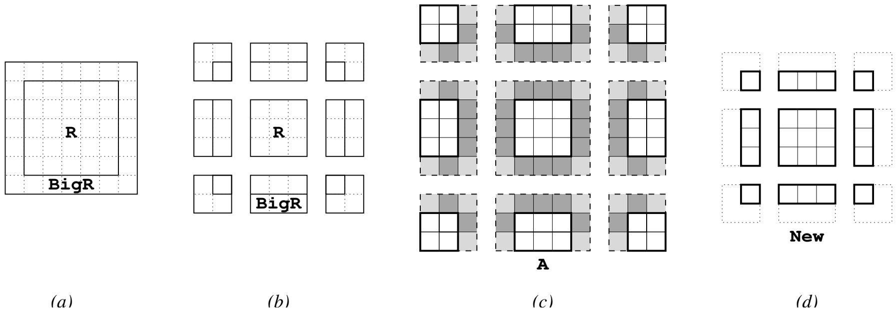


Figure 4.9: Fluff Required by the Jacobi Iteration. (a) The global-view of regions `R` and `BigR` for the Jacobi iteration. (b) The blocked distribution of the index space for a 3×3 processor grid. (c) The memory allocated on each processor for array `A`. White values are normal array values. Dark grey values indicate fluff values required by the processor to store elements transferred for the @-references. Light grey values indicate additional values that are allocated to keep each processor's data rectangular in shape. (d) The memory allocated on each processor for array `New`. No fluff values are required since it has no @-references within the program.

Fluff

One issue that has been ignored throughout this dissertation is the following: Where do the runtime libraries store values that are communicated between processors as the result of an @-, flood, reduction, or remap operator? For the last three operators, the communicated data can always be represented using an array—either the left-hand side of the assignment, or a temporary array introduced by the compiler. However, the answer for the @ operator is somewhat more complicated.

Since the @ operator is typically used to access an array's nearby neighboring elements, the communicated values are stored in memory adjacent to the processor's local block of array data, known as *fluff* (other work refers to these locations as *ghost cells*, *overlap regions*, or one of several other colorful names). The idea behind fluff is to store communicated values in close proximity with the original values, both to support a uniform means of accessing them, and in hopes that they will share a cache line with the processor's local array values.

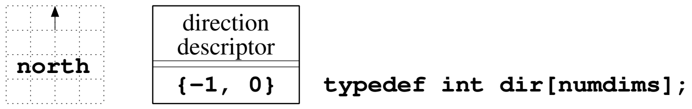


Figure 4.10: The Direction Descriptor. A direction is shown with its descriptor, a simple integer array.

The fluff required by each array is computed at compile time by searching for @-references and storing the maximal extent that every array is accessed in each direction. These values are then used by the array descriptor's initialization code to allocate extra rows or columns of memory along with the original chunk of data. For example, in the Jacobi implementation of Listing 2.15, the compiler would determine that the `A` array requires an extra row and column of data in each direction due to the 5-point stencil applied to it, while the `New` array requires no additional memory. See Figure 4.9 for an illustration.

4.2.7 The Direction Descriptor

Direction descriptors are the simplest of all. They are implemented using a simple integer array to store the direction's defining values. Direction descriptors are mentioned here for completeness rather than because of any intriguing aspects in their design. Figure 4.10 illustrates this fascinating data structure.

4.2.8 Initializing Runtime Descriptors

When a ZPL program is executed, the first thing that happens in the actual program is the initialization of the runtime libraries. Then, the global state descriptor is set up by parsing the command line and querying the processor's index from the runtime libraries. Next, the user's global variables are initialized in the order that they were declared in the program text. Since ZPL does not support forward declarations of grids, domains, regions, or variables, this ensures that the program's descriptors are set up in a legal order.

Each configuration variable is initialized by checking the command line to see if its default value was overridden. If it was not, the initializing expression is evaluated and stored in the configuration variable (implemented as a traditional C variable).

Grid descriptors are initialized by reading their global specification from the command line and storing it in the descriptor. Each processor then computes its position within the grid using simple computations based on a row-major layout of processors by global index. They also calculate the number of slices in the grid, compute whether or not each slice is distributed, and store this information in the descriptor.

Regions are initialized in two phases. When a region's declaration is first encountered, the processor computes its global bounds and stores them in its descriptor, along with the number of dimensions, sequence descriptors, and name of the region. The bounding indices of the region's distribution descriptor are also expanded to reflect the new region's contribution to the bounding index space.

The second phase of region initialization occurs after all of a distribution descriptor's regions have contributed their bounds. At that point, the indices of each dimension are blocked up to define each processor's local portion of the bounding index space. These indices are then intersected with each region's global indices to determine the processor's local portion, and the bounds and ownership fields in its descriptor are updated.

Array descriptors are similarly initialized in two phases. When the array's declaration is first encountered, its rank, defining regions, element information, and accessor are all stored in the descriptor. None of the array's data or layout can be set up until its defining region has been distributed. This marks the second phase of an array's initialization. Once the region's local indices have been determined, each array declared over that region reads the indices, expands them by the appropriate amount of fluff, allocates its block of data, and stores all the values in its descriptor.

Dynamic regions and local arrays are initialized in much the same way. Their descriptors are declared as variables within a function or C scope. These descriptors are initialized as soon as the scope is entered, and deallocated when it is left.

Listing 4.4: A General 2D M-Loop over Region R

```
if (R->iown) {
  const int i0Lo = R->locbound[0].lo;
  const int i0Hi = R->locbound[0].hi;
  const int i0Str = R->locbound[0].str;
  int i0;
  const int i1Lo = R->locbound[1].lo;
  const int i1Hi = R->locbound[1].hi;
  const int i1Str = R->locbound[1].str;
  int i1;

  for (i0 = i0Lo; i0 <= i0Hi; i0 += i0Str) {
    for (i1 = i1Lo; i1 <= i1Hi; i1 += i1Str) {
      /* loop body goes here */
    }
  }
}
```

4.3 Code Idioms

This section describes the most important code idioms used in generating ZPL code. Of particular importance are the nested loops generated by the compiler to iterate over a region's index set and the means by which an array's data is accessed.

4.3.1 M-Loops

Any time a region's indices need to be enumerated, ZPL uses a nested loop called an *m-loop* (multi-loop) to implement the traversal. Since regions are central to ZPL's parallel execution, m-loops abound in the generated C code of ZPL programs. M-loops come in many different varieties to optimize the traversal of the index set as much as possible. This section describes the most important m-loop varieties. This discussion assumes that each region's indices should be traversed in row-major order, though arbitrary orders are possible simply by reversing or reordering the implementing loops.

Listing 4.5: A Non-Strided M-Loop

```
if (R->iown) {
  const int i0Lo = R->locbound[0].lo;
  const int i0Hi = R->locbound[0].hi;
  int i0;
  const int i1Lo = R->locbound[1].lo;
  const int i1Hi = R->locbound[1].hi;
  int i1;

  for (i0 = i0Lo; i0 <= i0Hi; i0++) {
    for (i1 = i1Lo; i1 <= i1Hi; i1++) {
      /* loop body goes here */
    }
  }
}
```

General M-Loops

The most general m-loop implementation traverses all indices of a region R using code as shown in Listing 4.4. The entire loop is guarded by a check to see whether the processor executing the code actually owns a piece of the region. This is used to prevent unnecessary initialization and to avoid the possibility of performing useless iterations in outer loops whose inner loops are degenerate.

If the processor owns a piece of the region, the processor's local loop bounds and stride are read from the region descriptor and stored into constant integer variables to assure the C compiler that they are fixed regardless of the loop's contents. The loop itself is implemented using a straightforward nested loop whose depth corresponds to the region's rank. Note that this single loop idiom works for normal, strided, singleton, flood, and grid dimensions given region descriptors as defined in Section 4.2.5.

Listing 4.6: An M-Loop whose Second Dimension is Flat

```
if (R->iown) {
  const int i0Lo = R->locbound[0].lo;
  const int i0Hi = R->locbound[0].hi;
  int i0;
  const int i1 = R->locbound[0].lo;

  for (i0 = i0Lo; i0 <= i0Hi; i0++) {
    /* loop body goes here */
  }
}
```

Non-strided M-Loops

For region dimensions that can statically be determined to be non-strided, the stride variables in the general m-loop are eliminated to provide more precise information to the C compiler. As an example, a completely non-strided m-loop would be implemented as shown in Listing 4.5.

Flat M-Loops

Iterating over singleton, flood, and grid dimensions using a loop is overkill, since at most one index is represented per processor. When the compiler can statically classify a region dimension as one of these three types, the loop is replaced by a constant declaration of that dimension's index counter. For example, Listing 4.6 shows an m-loop whose second dimension is either a singleton, grid, or flood dimension. Note that this loop works for singleton dimensions even on processors that do not own its single index due to the conditional that guards the loop as a whole.

Macroized M-Loops

To keep ZPL's generated code more compact and readable, macros are defined for each of the m-loop idioms. For example, the macroized version of Listing 4.6 is shown in List-

Listing 4.7: A Macroized M-Loop

```
if (I_OWN(R)) {
  INIT_2D_LOOP(N, F, R);

  MLOOP_UP(R, 0) {
    MLOOP_UP_FLAT(R, 1) {
      /* loop body goes here */
    }
  }
}
```

ing 4.7. The `INIT_2D_LOOP()` macro declares the loop bounds, strides, and iteration variables as necessary. Its arguments indicate whether the region's dimensions are non-strided (`N`), strided (`S`), or flat (`F`), respectively. It also takes the region descriptor itself as an additional argument.

The loops themselves are defined using macros whose names reflect whether they are going up or down, and whether they are strided or flat (non-strided is the default). The arguments to these macros include the region descriptor (for documentation purposes only), and the dimension being traversed. Note that although the `MLOOP_UP_FLAT()` macro expands to the empty string, it is generated for the purposes of readability and completeness.

Masked M-Loops

When looping over region scopes that are masked, the m-loop's body is guarded by a conditional that checks the value of the mask at the current loop indices. Listing 4.8 shows such an m-loop. M-loops for which it is statically unknown whether or not a mask is used either add this check to the conditional or create two copies of the loop—one that uses masks and a second that does not.

Listing 4.8: A Masked M-Loop

```
if (I_OWN(R)) {
  INIT_2D_LOOP(N, F, R);

  MLOOP_UP(R, 0) {
    MLOOP_UP_FLAT(R, 1) {
      if (READ_2D_MASK(M)) {
        /* loop body goes here */
      }
    }
  }
}
```

Rank-Independent M-Loops

In many library routines, it is necessary to iterate over an m-loop of arbitrary rank. Rather than create m-loops for each possible rank, an *odometer-style m-loop* can be used, which stores its loop bounds in an integer array. Listing 4.9 shows such an m-loop. The bulk of the work in this code is spent adjusting the odometer by updating the inner loop's index and then propagating the carry as necessary.

4.3.2 Accessing Arrays

Accessing parallel arrays is the second crucial idiom in ZPL's generated C code since most m-loop bodies refer to array variables. As with m-loops, array accesses have many different variations based on the properties of the array's dimensions. This section gives an overview of the major array access styles.

General Array Access

The most general array access for a 2D array of doubles, `A`, appears as follows in ZPL:

```
*(double*)(A->data +
  (((i0 - A->offset[0])*A->blocksize[0])/A->stride[0]) +
  (((i1 - A->offset[1])*A->blocksize[1])/A->stride[1]));
```

Listing 4.9: A Rank-Independent M-Loop

```
if (R->iown) {
  int i[R->numdims];  /* the index */
  int dim;            /* a dim counter */
  int done = 0;       /* are we done yet? */

  /* set the index to the region's low bounds */
  for (dim=0; dim<numdims; dim++) {
    i[dim] = R->locbounds[dim].lo;
  }

  while (!done) {
    /* loop body goes here */

    /* increment the odometer */
    for (dim=numdims-1; dim>=0; dim--) {
      i[dim] += R->locbounds[dim].stride;
      if (i[dim] <= R->locbounds[dim].hi) {
        break;            /* done incrementing */
      } else {
        if (dim == 0) {
          done = 1;       /* done looping */
        } else {
          i[dim] = R->locbounds[dim].lo;
        }
      }
    }
  }
}
```

The index for each dimension is adjusted by the dimension's offset, scaled by its multiplicative factor, and then rescaled by its stride. As mentioned in Section 4.2.6, the two scaling fields cannot be collapsed into a single integer in the general case, requiring an explicit multiplication and division per dimension. In spite of its complexity, this access style has the benefit of working for all arrays, whether their dimensions are strided, non-strided, singleton, flooded, or grid.

Non-strided Array Access

If an array is statically known to be non-strided, the divisions in its access expression can be removed, simplifying it as follows:

```
*(double*)(A->data +
  ((i0 - A->offset[0]) * A->blocksize[0]) +
  ((i1 - A->offset[1]) * A->blocksize[1]));
```

This saves an integer division per dimension, which represents a fair amount of computational overhead in the general case.

Optimized Non-strided Array Access

Non-strided array accesses can be further optimized by folding the offset values into the data pointer itself. This is done using the `origin` pointer mentioned in Section 4.2.6. When a non-strided array descriptor is initialized, its `origin` field is set as follows:

```
A->origin = A->data - (A->offset[0]*A->blocksize[0])
                    - (A->offset[1]*A->blocksize[1]);
```

This allows the following optimized access expression, since the contribution of the offset factors has been precomputed:

```
*(double*)(A->origin + (i0 * A->blocksize[0])
                     + (i1 * A->blocksize[1]));
```

Note that this optimization is not permissible for strided dimensions due to the fact that the stride is not guaranteed to evenly divide the product of its offset and its blocksize.

Optimizing Flat Dimensions

As described in Section 4.2.6, flood and grid array dimensions have a blocksize of 0 so that all indices in that dimension will access the same values. In these cases, the array access component corresponding to the flat dimension can be eliminated altogether, saving the explicit multiply-by-zero. For example, the following array access would be legal for a 2D array whose first dimension was a singleton, flood, or grid dimension:

```
*(double*)(A->origin + (i1 * A->blocksize[1]));
```

As a related optimization, the multiplier for a singleton array dimension can also be set to 0 as long as all accesses to the array are legal.

Macroized Array Accesses

As with m-loops, the ZPL compiler typically generates array accesses using macros to improve readability and conciseness. For example, the array access just above could be generated as follows:

```
*(double*)ACCESS_2D(N, F, A, i0, i1)
```

As with the m-loop macros, each access macro is defined in terms of lower-dimensional access macros. The first two arguments indicate that the array's dimensions are non-strided and flat, respectively. The third argument is the array descriptor itself. The final arguments are the indices, which are specified explicitly in order to support arbitrary indexing. Note that strided arrays require a separate set of macros to support accesses using their `data` pointers rather than their `origin` pointers.

This concludes the description of the most important array access expressions. At this point, it seems useful to illustrate the implementation of an actual ZPL array statement. Listing 4.10 shows the main loop in the PSP matrix multiplication algorithm (line 27 of Listing 2.20), which iterates over `R3D` to reduce the elementwise products of `A3D` and `B3D` into a temporary grid array, `RedTemp` of size `[1..m, ::, 1..o]`.

Listing 4.10: The Main Loop in PSP Matrix Multiplication

```
if (I_OWN(R3D)) {
  INIT_3D_LOOP(N, N, N, R3D);

  MLOOP_UP(R3D, 0) {
    MLOOP_UP(R3D, 1) {
      MLOOP_UP(R3D, 2) {
        *(double*)ACCESS_3D(N, F, N, RedTemp, i0, i1, i2) +=
          *(double*)ACCESS_3D(N, N, F, A3D, i0, i1, i2) *
          *(double*)ACCESS_3D(F, N, N, B3D, i0, i1, i2);
      }
    }
  }
}
```

Accessor Functions

Within ZPL's runtime library routines, it is often necessary to access an array's elements without knowing its rank or anything about its dimensions' characteristics. This presents a challenge to accessing the array efficiently. One solution is to use a loop over the array's dimensions that incrementally calculates the most general array access as given at the beginning of this section. This has the disadvantages of using control flow and of potentially involving more math operations than the array actually requires.

The approach taken by the current implementation is to associate an accessor function with each array descriptor that returns its array elements' addresses as efficiently as possible. This has the disadvantage of requiring a function call, but supports a clean interface and the most efficient access possible for that array. Accessor functions are automatically created by the compiler and are associated with an array's descriptor during its initialization.

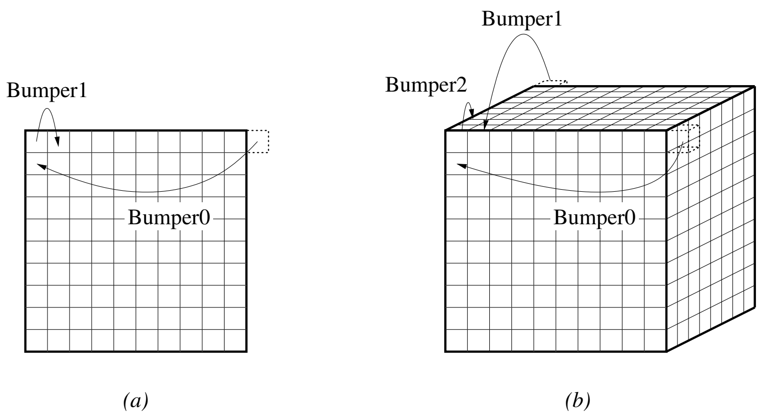


Figure 4.11: Walker/Bumper Accesses. (a) The two bumper values required to walk a pointer along a 2D array. (b) The three bumper values required to walk a pointer along a 3D array.

Walker/Bumper Accesses

When an array is being read or written within an m-loop, the addresses of the accessed data values have a very regular pattern due to the rectangular layout of the array's data. For this reason, access expressions as described in this section are rarely used within a compiler-generated m-loop. Although a perfect C compiler might use induction variable elimination to automatically recognize the regularity of the accesses, this is rarely achieved in practice due to ZPL's heavy use of pointers to refer to array descriptors. As a result, the current implementation performs this optimization manually for each array reference using a technique called *walker/bumper accesses*. It should be noted that this does not negate the usefulness of the access idioms above, since they are still required to initialize the walker/bumper approach. In addition, randomly accessing an array, as required by the remap operator, utilizes the full gamut of access techniques.

The idea behind walker/bumper accesses is to store a pointer (the walker) into the data array and to increment it by an appropriate amount (the bumpers) at the end of each implementing loop. This is shown in Figure 4.11. One bumper value is used per dimension, to advance the walker from its current position to the next index in that dimension. Note that bumper values depend heavily on the enclosing region as well as the order in which its indices are traversed.

Listing 4.11 shows the inner loop of the PSP matrix multiplication using walkers and bumpers rather than access macros. Although the resulting code looks much more complicated, the m-loop's body is now computationally much simpler and will result in faster overall execution. Note that the fact that certain dimensions of `A3D`, `B3D`, and `RedTemp` are flat cause the bumpers referring to those dimensions to be simplified or eliminated altogether. As expected, macros are used to make this code a bit more readable (not shown here).

Walkers and bumpers are also used within the odometer-style m-loops of library routines to eliminate the overhead of calling an array's accessor function for every iteration.

4.3.3 Advanced M-Loop Idioms

In addition to changing the order and direction of m-loop iterations, the ZPL back-end provides ZPL implementors with several additional hooks for generating m-loops. For example, optimizations may specify arbitrary statements that should be generated just inside or outside of each dimension's loop. In addition, an optimization my unroll or tile each m-loop dimension by a constant amount. These hooks provide implementors with a rich set of mechanisms that simplify the optimization of an m-loop and its array accesses [DCS01, Gul00].

Unrolling M-Loops

Unrolling an m-loop is a fairly simple matter of stamping out two sibling loops. The upper bound of the first loop is adjusted by its unrolling factor. The body of the loop, including walker and iteration variable updates, is stamped out the specified number of times. The second loop cleans up by iterating over any remaining iterations in the traditional manner.

Listing 4.11: The PSP Matrix Multiplication Loop Using Walkers and Bumpers

```
if (I_OWN(R3D)) {
  INIT_3D_LOOP(N, N, N, R3D);

  /* set up A's walkers and bumpers */
  char* Walker_A3D = ACCESS_3D(N, N, F, A3D, i0, i1, i2);
  const int Bumper_A3D_1 = A3D->blocksize[1];
  const int Bumper_A3D_0 = A3D->blocksize[0] +
                           (i1Lo - i1Hi - 1)*A3D->blocksize[1];

  /* set up B's walkers and bumpers */
  char* Walker_B3D = ACCESS_3D(F, N, N, B3D, i0, i1, i2);
  const int Bumper_B3D_2 = B3D->blocksize[2];
  const int Bumper_B3D_1 = B3D->blocksize[1] +
                           (i2Lo - i2Hi - 1)*B3D->blocksize[2];
  const int Bumper_B3D_0 = (i1Lo - i1Hi - 1)*B3D->blocksize[1];

  /* set up RedTemp's walkers and bumpers */
  char* Walker_RedTemp = ACCESS_3D(N, F, N, RedTemp, i0, i1, i2);
  const int Bumper_RedTemp_2 = RedTemp->blocksize[2];
  const int Bumper_RedTemp_1 =
             (i2Lo - i2Hi - 1)*RedTemp->blocksize[2];
  const int Bumper_RedTemp_0 = RedTemp->blocksize[0];

  MLOOP_UP(R3D, 0) {
    MLOOP_UP(R3D, 1) {
      MLOOP_UP(R3D, 2) {
        *(double *)Walker_RedTemp += *(double *)Walker_A3D *
                                     *(double *)Walker_B3D;

        Walker_B3D += Bumper_B3D_2;
        Walker_RedTemp += Bumper_RedTemp_2;
      }
      Walker_B3D += Bumper_B3D_1;
      Walker_A3D += Bumper_A3D_1;
      Walker_RedTemp += Bumper_RedTemp_1;
    }
    Walker_B3D += Bumper_B3D_0;
    Walker_A3D += Bumper_A3D_0;
    Walker_RedTemp += Bumper_RedTemp_0;
  }
}
```

Listing 4.12: An M-Loop Unrolled k Times

```
if (R->iown) {
  INIT_1D_LOOP(N, R);
  i0HiUnr = i0Hi-(k-1);

  /* unrolled loop */
  for (i0=i0Lo; i0<=i0HiUnr; i0++) {
    /* copy 1 of the loop body */
    /* increment walkers and i0 */

    /* copy 2 of the loop body */
    /* increment walkers and i0 */
    ...
    /* copy k of the loop body */
    /* increment walkers */
  }

  /* clean-up loop */
  for ( ; i0<i0Hi; i0++) {
    /* loop body */
    /* increment walkers */
  }
}
```

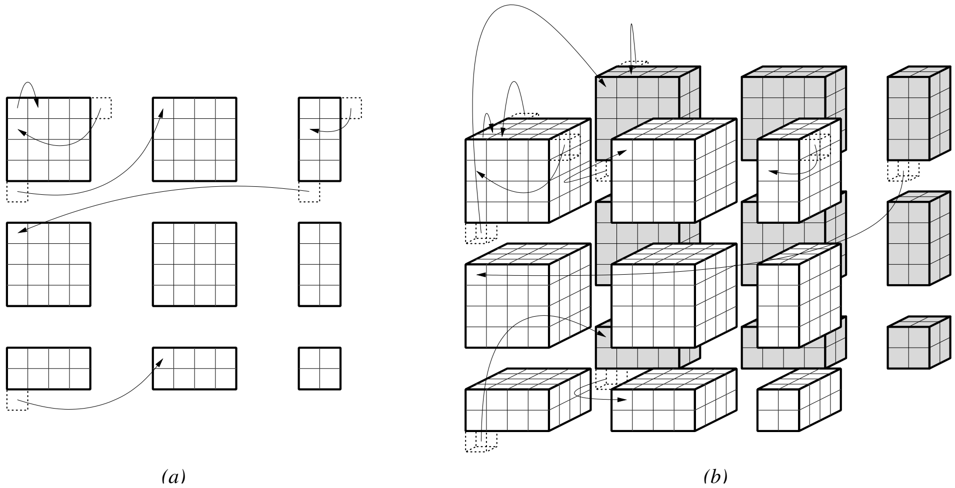


Figure 4.12: Tiled Walker/Bumper Accesses. (a) The six bumper values required to walk a pointer along a tiled 2D array. (b) The ten bumper values required to walk a pointer along a tiled 3D array.

Listing 4.12 shows a simple 1-dimensional m-loop that is unrolled k times. M-loops are generated recursively, making it simple to unroll their outer or inner loops.

Tiling M-Loops

Tiling an m-loop is somewhat trickier. Each tiled dimension is implemented using an outer and inner loop. Loops for non-tiled dimensions are represented using a traditional (inner) loop. Loops for dimensions with the degenerate tile size of 1 are implemented using a traditional outer loop and no inner loop. The m-loop's outer loops are generated first, followed by its inner loops, using a recursive strategy. Note that an m-loop's tiles can also be fully unrolled.

The real challenge is that the loop now potentially requires four bumpers per dimension: one to advance the walker within a normal-sized tile; one to advance it within a cleanup tile; one to advance to the next tile in that dimension; and one to reset to the first tile in the dimension. This tends to get complicated quickly, as shown in Figure 4.12.

Listing 4.13: Declarations for the Top of the Region/Mask Stack

```
struct _rmstackframe {
  region reg;
  array mask;
  int with;
} rmstackframe;

rmstackframe rmstack[maxrank+1];
```

The loops generated for these tiled m-loops tend to be sufficiently complicated that they are not worth reproducing here. The nice thing is that, having been implemented, client optimizations can take advantage of the mechanism simply by adjusting an m-loop's tiling factor within the compiler's internal representation.

4.3.4 The Region/Mask Stack

To implement ZPL's dynamic region scopes, the compiler uses a conceptual *region/mask stack* at runtime to keep track of the nested scopes that are open in each dimension. The region/mask stack has a fairly simple implementation using traditional C scopes. The top of the stack is implemented using a global array of structures, each of which stores a reference to a region descriptor (the current region), an array descriptor (the current mask), and a boolean flag (whether the mask was specified using the **`with`** or **`without`** keyword). Each dimension of this array corresponds to a single rank. Listing 4.13 shows the declaration of the region/mask stack. For example, the expression `rmstack[2].reg` would refer to the current 2D region.

New region scopes are "pushed" onto the top of the stack by opening a new scope in the C code which declares local references to region and array descriptors. The previous values at the top of the stack are saved into these variables, and the top of the stack is re-assigned. When the end of the region scope is reached, the local variables are restored to the top of the stack and the C scope is closed. Listing 4.14 illustrates this idiom.

Listing 4.14: Pushing and Popping the Region/Mask Stack

```
{  /* Open region scope for [R with Mask] */
  region prev2Dreg = rmstack[2].reg;
  array  prev2Darr = rmstack[2].arr;
  int prev2Dwith   = rmstack[2].with;
  rmstack[2].reg  = R;
  rmstack[2].arr  = Mask;
  rmstack[2].with = 1;

  ...region scope code ...

  rmstack[2].reg  = prev2Dreg;
  rmstack[2].arr  = prev2Darr;
  rmstack[2].with = prev2Dwith;
} /* Close region scope for [R with Mask] */
```

Listing 4.15: Fixing up the Region/Mask Stack on Procedure Returns

```
void proc_returns_early() {
  region covering2Dreg = rmstack[2].reg;
  array  covering2Darr = rmstack[2].arr;

  { /* Push new region scope */
    ...
    { /* Push new region scope */
      ...

      if (quit_now) {
        /* prepare to return from procedure */
        rmstack[2].reg = covering2Dreg;
        rmstack[2].arr = covering2Darr;

        return;
      }
    }
  }
}
```

The only tricky case comes when returning from a procedure call in the middle of its body. This case is handled by generating such procedures so that they take a snapshot of their covering scope immediately after being called. Then, all return statements prior to the procedure's end are generated to restore the original state before returning. Listing 4.15 illustrates this idiom.

4.4 Runtime Libraries

As mentioned previously, ZPL's runtime libraries contain routines to implement memory management, I/O, and region operators. Most of these routines are sufficiently straightforward and uninteresting that they are not discussed in this dissertation.

The primary role of the runtime libraries is to support the interprocessor communication required by ZPL's array operators. This section describes the philosophy of these routines, examines one piece of the interface in detail, and then provides an overview of the rest of the communication interface.

4.4.1 The Ironman Philosophy

ZPL's communication routines are guided by a set of ideas known collectively as the *Ironman philosophy* [CCS97]. This philosophy contains general principles that we believe should guide the design of any communication library used to implement the back-end of a portable parallel language. The philosophy contains the following points:

- A communication interface should remain paradigm-neutral to avoid favoring one particular communication style over another.

- A communication interface should be flexible enough that different implementations can map it to diverse architectures in a natural manner.

- Specifying the arguments to a communication interface should require little or no data movement to avoid unnecessary copying and overheads.

- A communication interface should provide optimization opportunities to its clients and to its implementors.

The Ironman philosophy was developed in response to the challenge of providing a single communication interface that works equally well on architectures with shared or distributed memory. In particular, recall that distributed memory machines tend to be most amenable to message-passing libraries such as MPI and PVM, which usually require copying data into messages and buffering them at the sending and/or receiving ends. In contrast, shared memory machines allow processors to access data values directly in the local memories of other processors, making one-sided puts and gets or direct memory accesses more attractive due to their reduced synchronization and copying requirements. The goal of the Ironman philosophy is to allow each architecture to use its preferred mechanism with a single interface.

In addition to the Ironman philosophy, ZPL's communication libraries are guided by an additional principle, motivated by its software engineering benefits:

- Communication interfaces for high-level languages should be divided into two layers: (1) a machine-independent, language-dependent layer and (2) a machine-dependent, language-independent layer.

For ZPL, this means that the machine-independent interface for a particular communication style will take region, array, and direction descriptors as arguments to describe the communication in a concise, high-level manner. In contrast, the machine-dependent version of the interface will take parameters whose types are traditional C types. This is done in order to make the low-level interface useful for other languages and to permit implementors who are unfamiliar with ZPL's data structures to port the interface to new architectures.

4.4.2 The Point-to-point Interface: An Ironman Case Study

This section illustrates the application of the Ironman philosophy in the context of the point-to-point communication routines used to implement ZPL's @ operator. As described by the previous section, this interface has both machine-dependent and machine-independent layers. The complete point-to-point interface is referred to as the *Ironman interface* due to the fact that it was the first practical application of the Ironman philosophy.

The key to understanding ZPL's point-to-point communication interface is to realize that there are two spans of time that characterize any point-to-point data transfer: the span during which it is legal to access the values that need to be transferred, and the span during which it is legal to access the locations into which they are being transferred. While this observation does not seem particularly deep, it forms the definition of the Ironman interface and allows data transfers to be implemented on any architecture with a minimal amount of overhead.

This discussion will refer to the values being transferred as the *source* of the communication and their target locations as the *destination*. First the abstract Ironman interface will be introduced, followed by descriptions of its machine-dependent and machine-independent implementations.

Abstract Ironman Interface

The core of the Ironman interface consists of a set of four calls, two for each of the time spans described above. The first two calls are named `SR()` (*source ready*) and `SV()` (*source volatile*). These calls are used to mark the span of time during which the source values can be accessed for transfer. The call to `SR()` indicates that the source values can be read by the communication library, whereas the call to `SV()` indicates that the source values are in danger of being changed, and may not be referenced by the library again once the call returns.

The corresponding set of calls on the destination side are `DR()` (*destination ready*) and `DN()` (*destination needed*). These calls indicate the span of time during which the destination locations can be written. The call to `DR()` indicates that the destination locations may be overwritten by the interface while `DN()` indicates that the destination values will be expected to be in place as soon as the call returns.

Table 4.1: Sample Point-to-Point Ironman Bindings

Routine	Blocking Messages	Non-Blocking Messages	One-Sided Comm. (Push-based)	One-Sided Comm. (Pull-based)	Threaded Communication
`DR()`	—	post receive	set flag	—	spawn thread to receive
`SR()`	send	post send	wait for flag; put data; clear flag	set flag	spawn thread to send
`DN()`	receive	wait for receive	wait for flag to clear	wait for flag; get data; clear flag	wait for receiving thread
`SV()`	—	wait for send	—	wait for flag to clear	wait for sending thread

It should be noted that these calls do not give any indication of how the communication should actually be implemented; their semantics do not reflect aspects of either message-passing or a one-sided communication style. In this sense, they satisfy the Ironman philosophy's paradigm-neutral requirement.

Table 4.1 shows that the interface also meets the flexibility requirements of the Ironman philosophy by showing how the calls might be implemented using message-passing (blocking or non-blocking) and one-sided/shared-memory communication (push- or pull-based). The final column indicates how the interface might be implemented on platforms with support for lightweight threads.

The Machine-Dependent Ironman Interface

This section describes the machine-dependent Ironman interface, as well as the data structures that it uses for its parameters. The prototypes and data structures that define the interface are given in Listing 4.16.

The machine-dependent Ironman interface consists of the four routines described in the abstract interface as well as two new routines to specify the lifetime of a particular communication. The first new routine, `PP_New()` defines a batch of point-to-point communications by providing the relevant source and destination addresses. This function returns a handle to an implementation-specific data structure (`PP_info`) that uniquely defines the communications. This handle serves as the single argument to the other five calls. The second new routine, `PP_Old()`, allows the client to signify that it has finished with this batch of communications, and that the `PP_info` structure can be deallocated or recycled.

The `PP_New()` and `PP_Old()` routines were added to the abstract interface in order to amortize the overheads associated with setting up new communications. Many programs perform the same communication several times within a loop, such as the communications required by the 5-point stencil in the Jacobi iteration. In such cases, it makes sense to describe the communication a single time outside of the main loop using `PP_New()`, then perform the communication within the loop using the four main calls as many times as necessary. After the loop has exited, a call is made to `PP_Old()` to let the library know that the runtime is done with the batch of communications for the time being.

Memory locations are described to the machine-dependent interface using a C structure known as a `memblock`. This data structure is used to describe a regularly-strided, multidimensional block of data, and is defined using a base pointer, an element size, and a vector that contains the number of elements and stride per dimension. The idea behind the `memblock` is that data locations in a multidimensional array can be described passively without marshaling data values, using only $\Theta(d)$ space. This is an example of the third Ironman principle in practice.

Listing 4.16: The Machine-Dependent Ironman Interface and Structures

```
/* a single block of memory locations */
typedef struct {
  void* baseptr;
  int elemsize;
  int numdims;
  struct {
    int numelems;
    int stride;
  } diminfo[numdims];
} memblock;

/* a single communication with another processor */
typedef struct {
  int procnum;
  int numblocks;
  memblock* mbvect;
} comminfo;

/* a pair of communications */
typedef struct {
  comminfo* recvinfo;
  comminfo* sendinfo;
} commpair;

/* the machine-dependent communication structure */
typedef ... PP_info;

/* The six point-to-point calls */
PP_info* PP_New(int numcomms, commpair* commlist);
void PP_DR(PP_info* commhandle);
void PP_SR(PP_info* commhandle);
void PP_DN(PP_info* commhandle);
void PP_SV(PP_info* commhandle);
void PP_Old(PP_info* commhandle);
```

Each point-to-point communication is described using a comminfo structure that indicates the other processor involved in the communication along with the memblocks that need to be sent or received. Note that although specifying the cooperating processor may seem antithetical to the notion of one-sided communication, the fact that all of ZPL's Ironman calls are being generated for a system in which all processes are cooperating makes this information both computable and appropriate.

The last data structure is the commpair which stores references to two comminfo structures: one to be sent, the other to be received. The idea behind the commpair is that for most point-to-point communications, many processors will act both as a sender and a receiver of data. Simple shifts of data as shown in Figure 3.11c are a common example of this phenomena.

The PP_New() call takes as its arguments a vector of commpairs and the length of that vector. Depending on the implementation and the data to be transferred, it may set up memory for marshaling data, exchange addresses with other processors, or simply tuck the parameters away for future reference. In any case, it uses a structure of its choosing to store all the state it needs to describe the communication and returns a handle to that structure to the caller.

The PP_DR(), PP_SR(), PP_DN(), and PP_SV() calls may then be implemented for a given platform as summarized in Table 4.1, or using any other technique that is consistent with the way the routines define the legal data transfer ranges. The PP_Old() routine simply cleans up the implementation's internal data structures and returns.

The Machine-Independent Ironman Interface

The machine-independent Ironman interface implements point-to-point communication using six routines, each of which corresponds to one of the six routines in the machine-dependent interface (Listing 4.17). Each @-reference and wrap-@ reference in a ZPL program is implemented using a set of calls to these six routines to implement its point-to-point communication.

Listing 4.17: The Machine-Independent Ironman Interface and Structures

```
/* a structure for defining an array's @ references */
typedef struct {
  array arr;                   /* an array */
  direction* dirlist;          /* a list of directions */
  int numdirs;                 /* # of @-reference directions */
  int numwrapdirs;             /* # of wrap-@ directions */
} atinfo;


/* The machine-independent Ironman interface */
void At_New(region reg,        /* the enclosing region */
            atinfo comminfo[], /* the info for each array */
            int numarrs,       /* the number of arrays */
            int commid);       /* a unique comm ID */

void At_DR(int commid);
void At_SR(int commid);
void At_DN(int commid);
void At_SV(int commid);

void At_Old(int commid);
```

As in the machine-dependent Ironman interface, the machine-independent interface's parameters support the ability to specify multiple communications/@-references at once. This fulfills the fourth Ironman principle by giving the implementation the opportunity to batch multiple messages to a single processor, thereby amortizing the cost of their communication overheads.

As in the low-level interface, the arguments to the initialization call, `At_New()` define the required communications, simply using a higher level of abstraction. The communications are described using the `atinfo` data structure which contains an array reference, a list of all the directions being applied to it, and a count of how many are @-references and how many are wrap-@ references. The `At_New()` routine takes as its arguments a reference to the covering region descriptor, a vector of `atinfo` structures, and a communication ID used to uniquely identify the communication.

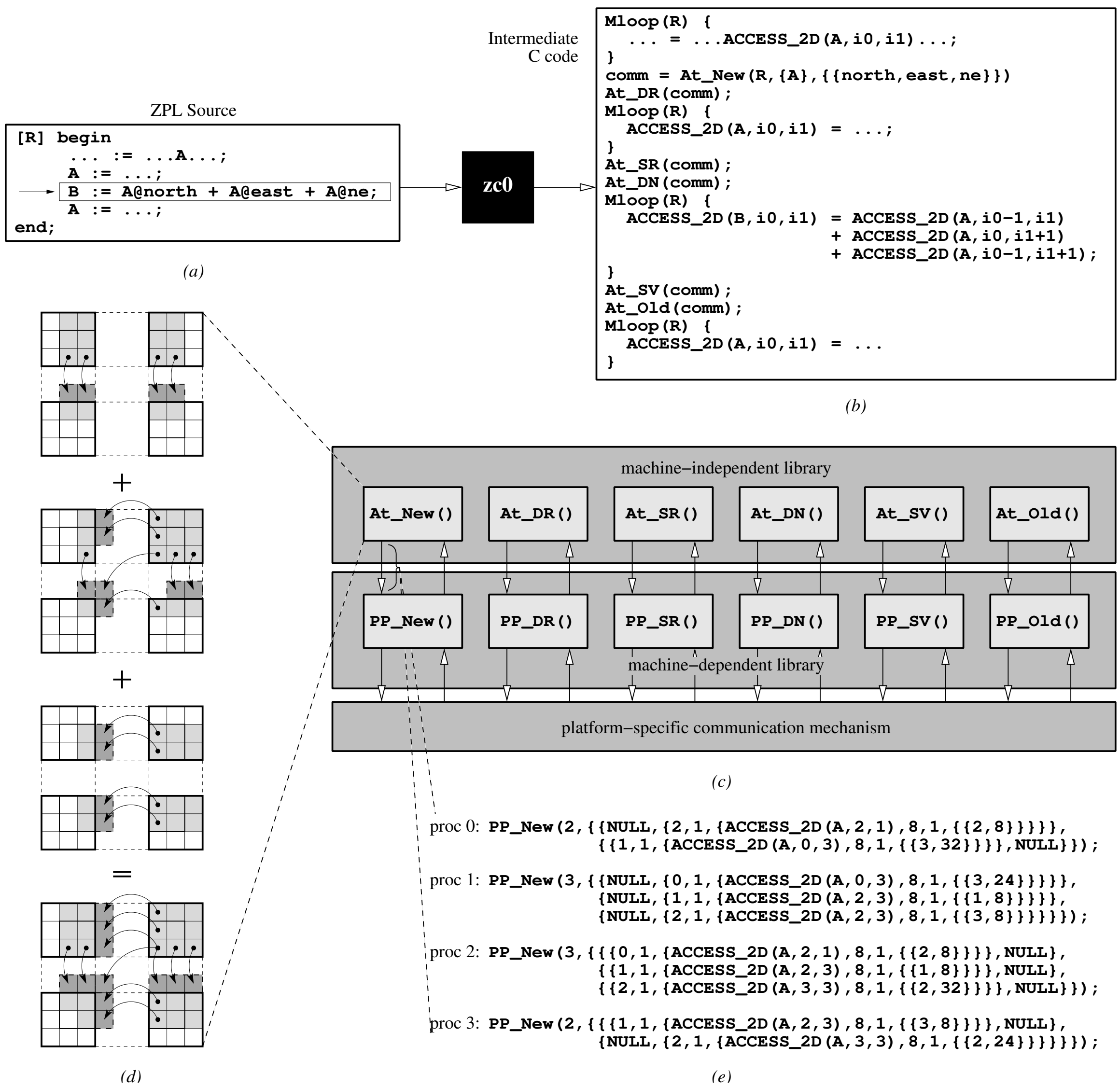


Figure 4.13: Ironman Communication. (a) ZPL source code that contains @-references. This example shows the communication induced by the boxed assignment statement. (b) The intermediate C code created by zc0. The compiler has combined the three @-references into a single set of calls and moved the calls apart in the code to hide latency. (c) The Ironman interface, shown as its six calls in two layers. The low-level calls are implemented in terms of a platform-specific mechanism such as MPI, SHMEM, or shared memory. (d) The communication required by each of the three @-references, and their union as computed by `At_New()`. Note that some redundant communication is eliminated in this manner and that multiple communications to the same processor are collapsed to one. (e) The calls to `PP_New()` that will be made by each processor.

The implementation of `At_New()` begins by computing which @-references refer to remote values. It buckets those references by processor number, and sets up the `memblock` structures needed to describe the source and destination locations. This information is then passed on to `PP_New()` and the handle that it returns is stored in an array based on the unique communication ID.

Each of the other machine-independent calls simply takes the communication ID as its parameter, looks up the communication handle, and passes the work off to its corresponding low-level routine. In addition, any communications that were classified as being local to a processor (*e.g.*, a wrap-@ for a region dimension that lies on a single processor) are implemented in these high-level routines using traditional memory copies.

The ZPL compiler inserts calls for a given @-reference based on the nearby reads and writes of the array's values. Not surprisingly, the decisions about how to group @-references into a single set of calls and how far apart to push these calls are a complicated matter. These amount to policy decisions for using the Ironman mechanism, and fortunately have been studied in great depth elsewhere [Cho99, CS97]. Figure 4.13 shows a complete overview of the Ironman interface and its use.

4.4.3 Other Communication Interfaces

This section gives a brief overview of the other communication interfaces required to implement the ZPL operators described in this dissertation. It should be noted that these routines have not always been as faithful to the Ironman philosophy as the point-to-point communication routines, primarily due to time constraints and a lack of compiler optimizations that have utilized them. Nevertheless, most reflect at least some of Ironman's guiding principles, and all have the potential for Ironman-motivated improvements.

Listing 4.18: The Flood Library Interface

```
/* Machine-independent flood routine */
void Flood(region dstreg,      /* destination region */
           region srcreg,      /* source region */
           array dstarr,       /* destination array */
           array srcarr);      /* source array */

/* Machine-dependent broadcast Routine */
void Bcast(void* data,         /* pointer to values */
           int size,           /* total data size */
           int slice,          /* processor grid slice */
           int srcproc);       /* source processor */
```

Floods

The flood operator is implemented using a machine-independent flood routine and a lower-level machine-dependent broadcast routine (Listing 4.18). The high-level call takes the source and destination regions and arrays that define the flood operation as its arguments.

The low-level routine is a generic broadcast that takes a data pointer and the data size to describe the source or destination of the broadcast. It also takes a processor grid slice number and a source processor index to indicate which processors are involved in the broadcast. It should be noted that this routine is used not only for flood operations, but also for file input of scalar values and the broadcast step of reduction routines that require them.

The current flood and broadcast interfaces limit compiler optimizations by failing to permit multiple arrays to be passed in, as with the Ironman interface. Moreover, the low-level routine requires its data to be consecutive in memory rather than referenced using a `memblock`-style structure.

It is natural to wonder whether dividing the flood and broadcast routines into component parts like the Ironman interface would be beneficial to an implementation or not. It should be noted that the log-based broadcast trees often used to implement broadcasts require tighter synchronization between the processors, since data is propagated through

Listing 4.19: The Reduction Library Interface

```
/* Machine-independent reduction routine */
void Reduce_int(region dstreg, /* destination region */
                region srcreg, /* source region */
                array locdata, /* proc's local contribution */
                int op);       /* reduction operator */

/* Machine-dependent reduction routine */
void _Reduce_int(int* data,     /* pointer to values */
                 int numelems, /* number of elements */
                 int op,       /* reduction operator */
                 int slice,    /* processor grid slice */
                 int dstproc,  /* destination processor */
                 int bcast);   /* broadcast result to slice? */
```

the network in multiple steps. While this would seem to limit the benefits of a split-phase broadcast interface, the topic deserves more study. For now, the implementation has chosen to implement broadcasts using a single atomic operation.

Reductions

As might be expected, the reduction routines are quite similar to those used to implement floods (Listing 4.19). The chief difference is that the type of data being reduced is now important since the routines must apply the reduction operation to combine values. As a result, each reduction routine is instantiated for each of the scalar types. The array used to store a processor's local contribution is re-used to store the result of the reduction, minimizing the number of arrays passed to the high-level call. Each routine takes an argument specifying the reduction operator. User-specified operations are handled by a separate call. The low-level routine takes an additional parameter indicating whether or not the result should be re-broadcast back to the processor grid slice. This is used by full reductions as well as reductions to flood dimensions. Such broadcasts may be implemented in the reduction routine itself, or using the broadcast routine from the previous section.

Listing 4.20: The Remap Library Interface

```
/* Machine-dependent remap routines */
void SetMaps(region reg,      /* destination region */
             array map[],     /* vector of map arrays */
             int srcrank);    /* rank of source array */

void Remap(region reg,        /* destination region */
           array dst,         /* destination array */
           array src);        /* source array */
```

The current implementation of reductions has similar failings with regard to the Ironman philosophy as the flood routines. An interface which strives to achieve the Ironman principles more completely has been studied in previous work, but has never been added to the actual runtime interface [Wea99].

Remaps

The remap operator fails to meet the Ironman philosophy even more severely, as it does not even have a high-level interface. The current interface consists of two low-level calls, one to define the operator's map arrays and the second to perform the remap itself (Listing 4.20). This division allows the communication schedule dictated by the map arrays to be used for multiple instances of the remap operator, if optimized appropriately by the compiler.

Each of these routines takes region and array descriptors as arguments and performs iterations over them within their implementations. The result is that loops and array accesses cannot be optimized as described in Sections 4.3.1 and 4.3.2, resulting in some of the performance problems seen in Chapter 3's experiments. This poor organization is a symptom of the fact that this is a first-generation implementation of the remap operator, written with correctness in mind, but not performance. Work is currently being done to break the loops and accesses out of these routines and into the compiler-generated code. It is expected that this will also result in a set of machine-independent and -dependent routines to implement the all-to-all communication.

Tokens

One final communication mechanism is the use of *tokens* to pass small control messages from one processor to another. Tokens are used primarily to implement file I/O so that processors take turns reading or writing a data stream in a coordinated manner. They are mentioned here for completeness, but are otherwise not important to this dissertation.

4.5 Compiling ZPL

This section provides a high-level overview of how zc0 compiles ZPL to C. The zc0 compiler can be thought of as having a number of distinct phases, illustrated in Figure 4.14. These phases are defined as follows:

- The *Parsing phase* reads the ZPL source code and stores it as a high-level *abstract syntax tree* (AST) that directly mirrors its structure. This high-level representation is maintained up until the final phase, which preserves ZPL's clean specification of parallel computation throughout the compilation process.

- The *Analysis I phase* performs simple analyses such as callgraph analysis, classification of parallel procedures, and type inference.

- The *Legalization phase* breaks complex expressions out of array statements to ensure that each one can be implemented using a single m-loop. Such expressions include floods, reductions, and remaps that are not part of a simple assignment, as well as procedure calls that return parallel arrays. After legalization has completed, each statement in the AST can be classified as having a single rank.

- The *Typecheck phase* ensures that the ZPL program is legal. This is performed later than one might expect in order to simplify the typechecker's job by legalizing the AST first. As a result, earlier phases are written to be tolerant of illegal code.

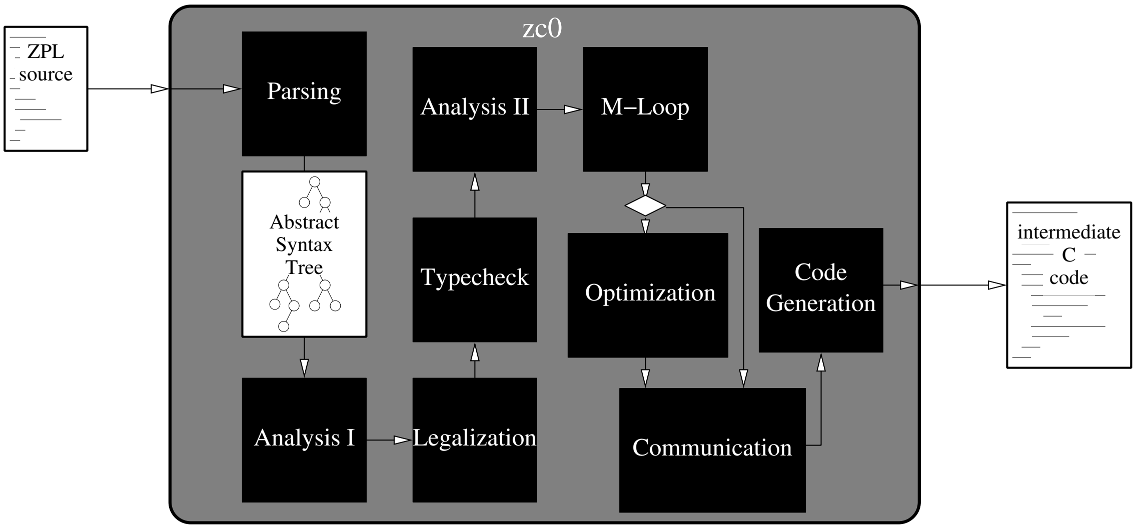


Figure 4.14: The Phases of ZPL Compilation. The zc0 compiler starts by parsing the ZPL source code into an abstract syntax tree representation. The AST then passes through various analysis, transformation, and optimization stages until it reaches code generation, at which point the C implementation is emitted.

- The *Analysis II phase* conducts more analysis, primarily relating to parallel issues. These analyses include determination of each statement's covering region scope, alias analysis, and dependence analysis. In addition, each array's implicit storage and fluff requirements are determined in this phase.

- The *m-loop phase* inserts an m-loop node into the AST for each parallel statement. Each node represents the m-loop that will eventually be generated for the statement, and contains fields to indicate the iteration order, unrolling and tiling factors, and other relevant details.

- The *Optimization phase* performs a number of optional optimizations, including stencil optimizations [DCS01], m-loop fusion and array contraction [LLS98, Lew00], and optimizations on flood arrays and regions [Gul00].

- The *Communication phase* inserts communication as required by array operators and optimizes it to reduce overheads [CS97, Cho99, Wea99]. These optimizations, as well as the m-loop fusion in the optimization phase, use a *factor-join* compilation model in which array expressions are broken into atomic *factors*, moved around as constrained by their dependences, and *joined* with compatible factors in order to increase locality and reduce the overheads associated with m-loops and communication [CCL$^+$96]. For example, a full reduction operator would be broken into a local computation factor, a global reduction factor, a global broadcast factor, and an assignment factor to store the result. These factors can then be joined with compatible factors stemming from other reductions or array statements.

- The *Code generation phase* produces the C code that implements a ZPL program. This consists largely of generating traditional scalar code, the idioms of Section 4.3, and calls to the runtime libraries as appropriate.

Although the process of compiling ZPL contains many nitpicky details, most of the interesting and challenging ones have been described either in this thesis or in the references provided in this discussion. Rather than delve into the more tedious details, this dissertation elects to postpone such discussion to a future document.

4.6 Related Work

4.6.1 Implementation of Parallel Languages

Most of the parallel languages in the previous chapter have articles detailing their implementations. This section gives a brief overview of some of the most relevant examples.

High Performance Fortran

A vast number of documents have been written detailing implementations of High Performance Fortran and its predecessors, Vienna Fortran and Fortran-D [Tse93, vRDSP96, Dut97, BCF+93, GMS+95]. This plethora of publications is partially due to the concerted effort that the parallel programming community put into supporting parallel Fortran dialects, and partially due to the challenges and obstacles involved in efficiently implementing HPF programs. In particular, the language's support for general array indexing and directive-based parallelism requires HPF compilers to serve as parallelizing compilers in the limit. A good HPF compiler must be able to interpret an arbitrary loop nest, determine how its referenced arrays should be aligned, and then insert communication to efficiently bring disparate array elements together as required by the computation. Communication issues such as vectorization and identification of reductions which are taken for granted in a language like ZPL must explicitly be located and implemented by the HPF compiler. This has proven to be challenging to do correctly and efficiently, especially given the lack of an HPF performance model [Ngo97].

Single Assignment C

Single-Assignment C's functional semantics both ease and complicate the task of implementing it efficiently in parallel. To its benefit, SAC's functional semantics eliminate many issues of aliasing and data dependences found in imperative languages, resulting in simpler semantics. However, the temporary arrays that would be used by a naive implementation of SAC can result in excessive amounts of memory being used for each statement. To combat this problem, the SAC compiler uses a form of loop fusion known as with-loop folding [GKS99]. This optimization is similar to the m-loop fusion and array contraction found in the ZPL compiler. The SAC compiler implements with-loops using a number of threads that are created during program startup and used to execute different loop iterations [Gre98]. Other optimizations in SAC attempt to reduce synchronization of these threads between with-loops.

KeLP

Since KeLP targets the parallel execution of irregular block-structured algorithms, one of its biggest challenges is determining an efficient communication schedule to transfer values between the Regions owned by different processors [FKB98]. KeLP's Mover class analyzes a particular MotionPlan at runtime to construct an efficient communication pattern using an inspector/executor model [BF99, ASS95]. This data movement is then implemented using non-blocking MPI calls. The result is a communication schedule that tends to be competitive with hand-coded MPI.

NESL

NESL is compiled using an intermediate language called VCODE [BCH+94]. VCODE is a stack-based intermediate language that operates on vectors and segmented vectors. The compilation challenge is therefore to convert nested operations on nested NESL sequences into segmented VCODE operations on sets of flat vectors. The VCODE is then compiled to shared memory MIMD machines in a manner that strives to reduce synchronization and intermediate storage [Cha91].

4.6.2 Communication Interfaces

As mentioned in the previous chapter, the communication interfaces that are most widely used today are the MPI, PVM, and SHMEM libraries [Mes94, BDG+91, BK94]. These libraries deviate from the Ironman philosophy in that their routines describe particular communication paradigms such as message-passing or one-sided communication. This decision makes sense since the interfaces are designed for human users rather than the back-ends of compilers. However, it limits their efficiency. Attempts to implement these styles of communication on platforms whose architectures do not support the interface's paradigm have typically resulted in communication overheads that are sub-optimal [SSO+95, CCS95, BB00].

One communication system that shares the goals of Ironman is the *Cooperative Data Sharing* (CDS) system developed by DiNucci [DiN97, DiN96]. Unlike other interfaces, CDS supports interprocessor communication without making assumptions about the architecture's communication paradigm and without bloating the interface. This is achieved using a model that provides each processor with a local memory and a local *communication heap*. In addition, a set of public *communication cells* are made available, which allow processors to publicize references to regions within their communication heaps. Using the supported operations on communication cells and heaps, CDS supports a wide range of communication styles including one-sided communication, two-sided communication, client-server models, broadcasts, and reductions. As such, CDS offers a flexible and elegant abstract model for interprocessor communication. The big question for CDS is whether or not the system can be implemented efficiently enough on a variety of architectures, and whether it can be programmed such that a single use of it performs well on all architectures.

4.7 Discussion

The biggest weakness of ZPL's current implementation is its lack of support for non-blocked distributions. This assumption restricts the generality of its region and array descriptors. In particular, note that the region descriptor's `locbound[]` fields assume a blocked distribution, as do the array descriptor's memory layout fields (`offset[]`, `blocksize[]`, and `stride[]`).

Regular and irregular blocked distributions provide good data locality and surface-to-volume ratios, making them an acceptable choice for a majority of array-based parallel algorithms. However, support for cyclic, block-cyclic, and more arbitrary grid-aligned distributions seems crucial in order for ZPL to be a general, robust language. This section considers potential implementations for each of these distributions.

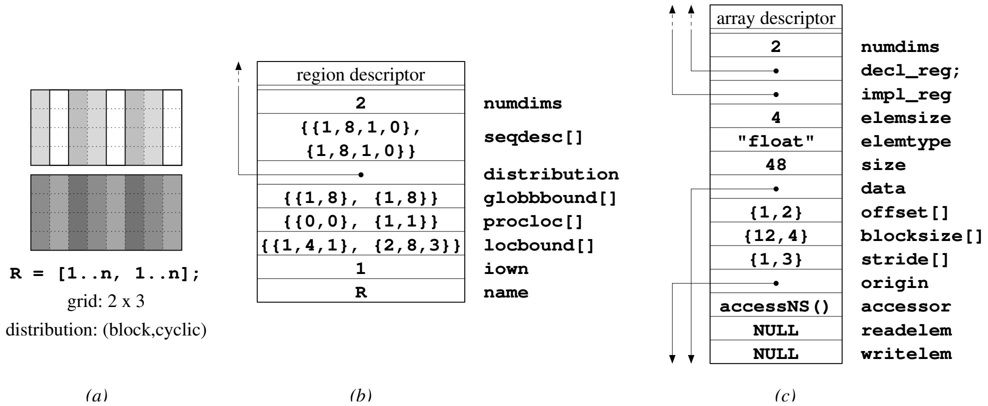


Figure 4.15: Proposed Cyclic Distribution Implementation. (a) A square region that is distributed to a 2×3 processor grid using a blocked distribution for the first dimension and a cyclic distribution for the second. The unshaded indices indicate those that would be owned by the second processor in the grid. (b) The region descriptor that describes the unshaded indices. Note that the local bounds in the second dimension are strided by 3 to capture the cyclic striding. Furthermore, the region's upper processor location is processor 1 rather than processor 2 since it holds the highest index. (c) The array descriptor for an array of floats declared using this region. As usual, the offsets are set so that the lowest local index—(1,2)—maps to the origin. The blocksize is also set as though this was a normal 3×4 block of data. However, the stride is set to 3 in the second dimension to compress the strided indices down to consecutive data values. A traditional accessor function can be used to access these values, provided that it uses the stride in the second dimension.

4.7.1 Cyclic Distributions

Cyclic distributions seem like a fairly simple modification to the current blocked distribution scheme. In particular, the stride field of the region descriptor's local bounds can be set to the number of processors in that dimension, causing m-loops to skip through its indices as required. Furthermore, the array values for a cyclic distribution can be stored using a dense block of memory by setting the stride fields of the array descriptor to values that counteract the large increment between indices, thereby accessing adjacent values. See Figure 4.15 for an illustration.

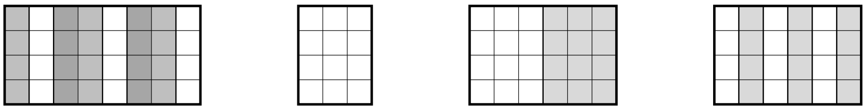


Figure 4.16: Fluff Allocation for Cyclic Distributions. (a) An array whose columns are distributed cyclically. (b) A processor's dense allocation of the unshaded values from part a, assuming that no fluff is required. (c) The fluff allocation for an `@east` reference in which the fluff values are placed to the side of the processor's data. Note that the amount of fluff required is equal to the amount of data that the processor owns, due to the cyclic distribution. This allocation has the disadvantage that calculating the address for a fluff value does not match the normal array access idioms since the alignment of an index affects its resulting address. (d) An alternative scheme in which the fluff is interleaved with the actual data. This allows traditional access idioms to be used, but has the disadvantage of spacing a processor's actual values out farther in memory.

The biggest challenge to cyclic distributions seems to be the issue of fluff. When applying a simple @-reference like `A@east` to a cyclically-distributed array, every index requires a data value to be communicated from its neighboring processor, since adjacent indices are no longer located on the same processor. If the fluff is stored in an adjacent block of memory, as in the blocked implementation, it breaks the array's indexing scheme, since a non-affine accessor function would be required to access normal and fluff values uniformly (Figure 4.16c). Yet, if the fluff is interleaved with the normal array values, it has the effect of spreading values out in memory, possibly decreasing cache utilization for array accesses that do not refer to fluff (Figure 4.16d).

In deciding between these implementations, the thing to keep in mind is that cyclic distributions are a poor choice for applications that use @-references (and therefore fluff) since each one requires the communication of an amount of data equal to the global array size to be transferred. For this reason, it can be assumed that wise users will tend not to use cyclic distributions in conjunction with the @ operator, and interleaved storage can be used to implement any programs that do.

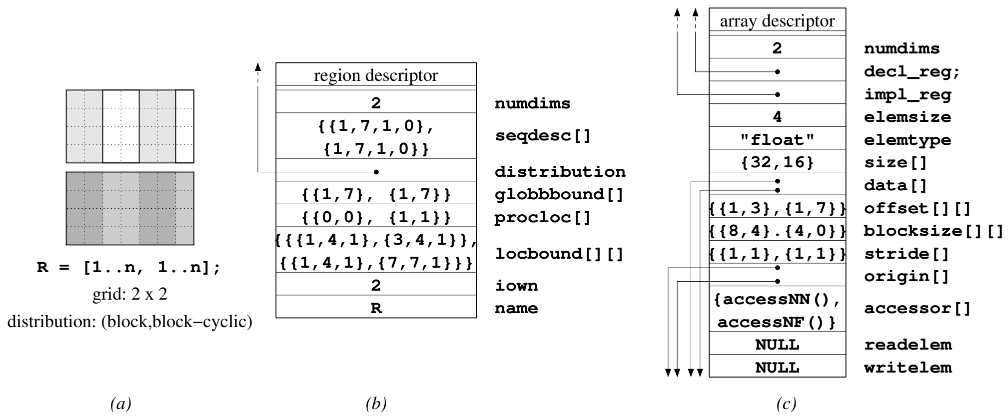


Figure 4.17: Proposed Block-Cyclic Distribution Implementation. (a) A square region that is distributed to a 2×2 processor grid using a blocked distribution for the first dimension and a block-cyclic distribution for the second. The unshaded indices indicate those that would be owned by the second processor in the grid. (b) The region descriptor that describes the unshaded indices. Note that an extra dimension has been added to the traditional local bounds field so that the bounds of each block owned by the processor can be described. The `iown` field has been changed from a boolean to an indicator of the number of distinct blocks owned by this processor. (c) The array descriptor for an array of floats declared using this region. Several of the fields (the data size, data and origin pointers, offsets, blocksizes, and strides) have had an extra dimension added so that the memory layout of each block owned by the processor can be described.

4.7.2 Block-Cyclic Distributions

Block-cyclic distributions cannot be described by the current region and array descriptors due to the fact that they essentially have two strides: one to move between elements within a block and a second to move from one block to the next. The most straightforward modification to the descriptors would be to store a sequence of local bounds and memory layout information, one for each block that the processor owns. Figure 4.17 gives an illustration. This approach is similar in concept to assigning multiple virtual processors to a single physical processor, but reduces some of the overhead since only the descriptors' local fields need to be replicated. This approach would also increase the depth of the m-loop nest, since an additional loop would be required to iterate over blocks for each block-cyclic dimension.

An advantage to this scheme is that all existing code idioms and runtime libraries would require only minor changes to iterate over these blocks of indices and data.

4.7.3 Arbitrary Distributions

One of the biggest challenges for implementing arbitrary region distributions is generating m-loops that can iterate over a processor's local indices, whether in a compiler-generated loop or a library routine. It seems unlikely that compiler analysis of a user-supplied distribution function would result in the production of efficient loop nests. A related challenge is the memory layout of data values for arrays distributed using arbitrary distributions.

One solution would be to require the user to supply iteration and memory layout descriptions for each distribution function that they specify. The problem with this of course is that it requires more work on the user's part and would tend to thwart the compiler's ability to perform optimizations on m-loops or array accesses.

My proposal is rather to perform a probing of the distribution function during the configuration portion of a ZPL executable, to find regular patterns in the indices that are mapped to each processor. For a given domain dimension whose distribution is user-defined, each processor would iterate over the domain's global bounds, keeping track of the indices that it owns. As it does so, the processor would keep track of consecutive indices or indices strided by a regular amount. As soon as one of these patterns was broken, the processor would store that pattern in a single region/array block, similar to those used in the block-cyclic distribution. In the worst case, every pair of indices would require its own block, since any two indices are strided by some constant amount. Figure 4.18 shows an example of this scheme.

This approach is admittedly naive. It cannot hope to efficiently summarize a completely arbitrary set of region indices. But then again, it is difficult to imagine an automated scheme that could. In its favor, if there is any amount of regularity in a user-defined distribution, it seems likely that this scheme will detect the pattern and represent it succinctly. In particular, this proposal would do a great job with user-supplied distributions that are essentially

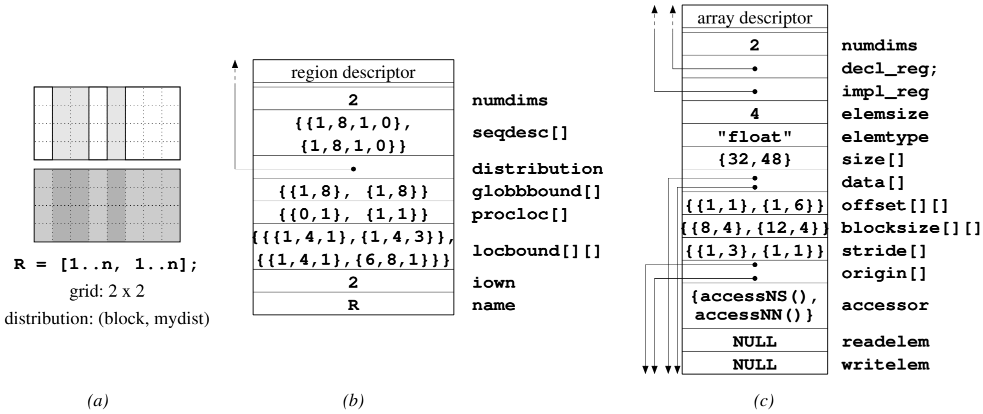


Figure 4.18: Proposed User-Defined Distribution Implementation. (a) A square region that is distributed to a 2×2 processor grid using a blocked distribution for the first dimension and a user-defined distribution, `mydist()` for the second. Calling `mydist()` with the global indices $1 \ldots 8$, processor 1 determines that it owns indices 1, 4, 6, 7, and 8 (shown unshaded). Indices 1 and 4 are considered to be strided by 3. However, index 6 breaks this pattern, so a new block is started. Indices 7 and 8 are each strided by 1 relative to 6, so they both belong to the second block. Thus, the processor's indices can be summarized as a block from 1 to 4 by 3 and a second block from 6 to 8 by 1. (b) The region descriptor that describes the processor's indices. As in the block-cyclic implementation, the local bound field is vectorized to describe the separate blocks of indices. (c) The array descriptor for an array of floats declared using this region. Once again, several fields are vectorized to describe each block of data owned by the processor.

blocked, cyclic, or block-cyclic in nature. For example, imagine that users specify nonstandard block sizes, such as in the `logdist()` distribution of Section 3.12.2. Or, that they want a variation of a cyclic or block-cyclic distribution that the compiler does not support. In these instances, the proposed approach would detect the blocks or cycle and use an internal representation that would be almost as efficient as a built-in implementation. This should reduce the number of built-in distributions that the ZPL language and compilers feel compelled to support.

Finally, note that the time spent classifying a distribution should be small given that grid-aligned distributions only support shuffling of a dimension of indices rather than the global index set. Thus, for any multidimensional problem, each processor is only probing the edges of the index space rather than the space as a whole.

Obviously, this scheme will have some challenges that I have not anticipated, but it seems like a reasonable first step toward supporting arbitrary distributions once ZPL can support cyclic and block-cyclic distributions.

4.8 Summary

This chapter has provided detailed descriptions of the three main components used to implement ZPL: (1) the runtime descriptors used to represent ZPL's parallel concepts at runtime, (2) code idioms such as the m-loops and array accesses used to implement the language, and (3) the runtime libraries used to implement interprocessor communication. Regions play an important role in all three of these areas due to their concise runtime representation of an index set and their use in both compiler-generated and library codes.

This chapter also gave an introduction to the Ironman philosophy, an overview of ZPL compilation, and a proposal for the implementation of region distributions other than blocked. The next two chapters will consider some modifications that have been added to the base ZPL region to help with the expression of more complex algorithmic paradigms: namely, hierarchical computation and sparse computation.

Chapter 5

REGION SUPPORT FOR HIERARCHICAL COMPUTATIONS

This chapter examines the use of regions to implement hierarchical array-based algorithms. Such computations require region support beyond that described in previous chapters in order to create a variable-sized collection of regions, each with its own characteristics. The approach described here parameterizes a region's characteristics to express hierarchical index sets. Though this approach results in a reasonable solution, it is a prime example of a poorly-designed language construct, as it lacks generality and orthogonality with other ZPL concepts. The strengths and weaknesses of the approach are evaluated at the end of the chapter, and a proposal for eliminating its defects is presented.

The rest of this chapter is organized as follows. Section 5.1 describes hierarchical algorithms, concentrating on *multigrid* algorithms and the NAS MG benchmark. Sections 5.2 and 5.3 explain how parameterized regions can be used to express multigrid-style computations in ZPL and demonstrate their use in a ZPL implementation of MG. The implementation of ZPL's parameterized regions is explained briefly in Section 5.4. Section 5.5 compares versions of MG written in several different languages in terms of clarity, performance, and portability. Related work is described in Section 5.6, and the strengths and weaknesses of ZPL's support for hierarchical programming are discussed in Section 5.7. This chapter's contents are an expanded presentation of work that was published previously [CDS00, CDS99].

5.1 Multigrid Algorithms

Hierarchical algorithms are characterized by their use of hierarchical data structures to exponentially reduce the complexity of a traditional algorithm. An important subclass of hierarchical algorithms are *multigrid methods* [Wes92, Bra77], so named due to their technique of computing using multiple arrays, or "grids." Multigrid methods enjoy widespread use due to their simplicity and regularity, as well as their applicability to a number of important problem domains.

Multigrid algorithms tend to work as follows. A user has a problem to solve in a finely discretized space. However, the complexity of the computation and the size of the problem space are so great that a straightforward solution to the problem would be prohibitively expensive. Rather than attack the original problem directly, the multigrid method gradually simplifies the discretization space to create smaller and smaller approximations of the original problem. This phase of the algorithm is known as *coarsening*, and it proceeds by collapsing a group of discretized elements down to a single element. Typically, a pair of elements per dimension is projected down to a single element of the coarser grid.

Once this coarsening process produces a discretization that can be solved in a reasonable amount of time, the simplified problem is solved using traditional methods. Then, its solution is used to construct an approximate solution to the original problem. This is done using a series of *interpolation steps* that map the solution back down the hierarchy, expanding it along the way until it forms an appropriate solution for the original discretization of the problem. These interpolation steps are typically performed by comparing the solutions at each level with the approximations to the problem that were computed during coarsening. The entire process of coarsening and refining the solution is often performed repeatedly to gradually refine the solution. Figure 5.1 provides a simple schematic of the multigrid method's execution.

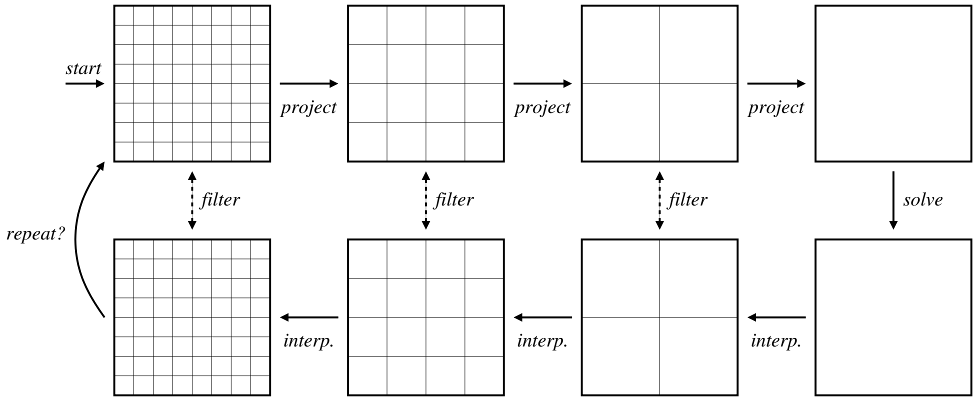


Figure 5.1: The Multigrid Method. This diagram shows a simple overview of the multigrid method. The original discrete problem is projected repeatedly to create coarser and coarser approximations to the original problem. That problem is then solved and the result is interpolated back down the hierarchy, filtering along the way against the original approximations. The entire cycle may then be repeated to improve the solution.

5.1.1 NAS MG

A well-known benchmark that makes use of the multigrid method is the MG benchmark from the NAS Parallel Benchmark Suite (NPB) [BBB$^+$94]. NPB contains a number of benchmarks developed with the intent of capturing algorithms and communication patterns used in important parallel scientific computations. One goal of this effort is to provide a consistent means of measuring performance from one parallel platform to the next.

The original NAS benchmarks were distributed as English-text descriptions of the algorithms, with sample inputs and outputs provided for verification purposes. Version 2 of the benchmark suite (NPB2) is distributed as a collection of hand-coded implementations using Fortran and MPI, representing a reasonable and portable implementation of the version 1 specifications [BHS$^+$95, SvdWWY97].

The NAS MG benchmark uses a multigrid computation to obtain an approximate solution to a scalar Poisson problem on a discrete 3D grid with periodic boundary conditions. As such, it represents a crisp algorithm for use in evaluating ZPL's support for hierarchical

Table 5.1: Parameters for Production Grade Classes of MG

Class	*Problem Size*	*Iterations*
A	$256 \times 256 \times 256$	4
B	$256 \times 256 \times 256$	20
C	$512 \times 512 \times 512$	20

array-based programming. Five different classes of MG exist, each with its own problem size and number of iterations. Two of these classes—S and W—are small versions designed for development purposes only. The other three—A, B, and C—are production grade classes. The defining parameters for these three classes are summarized in Table 5.1. Note that classes A and B use the same problem size, but for differing numbers of iterations.

Though NAS MG contains simplifications that may not reflect the assumptions of more complex multigrid solvers, its authors explain that they "chose it for its portability and simplicity, and expect that a supercomputer which can run it effectively will also be able to run more complex multigrid problems" [BF95]. This study of MG adopts a similar philosophy, assuming that in order to support more complex multigrid methods and hierarchical algorithms, a parallel language must first demonstrate success with simpler, more regular examples.

The bulk of the computation in MG is implemented using four 27-point stencils (Figure 5.2). Two of these stencils—*interp* and *rprj3*—interpolate and project between representations of the problem at adjacent levels of the hierarchy. The other two—*resid* and *psinv*—implement refinement operations at a single level of the hierarchy. A good parallel implementation of these stencils would use point-to-point communication to transfer boundary values between processors. MG's communications and stencils are very similar to those found in the Jacobi iteration, simply involving more directions. MG also requires support for periodic boundary conditions and for full reductions over the finest grid.

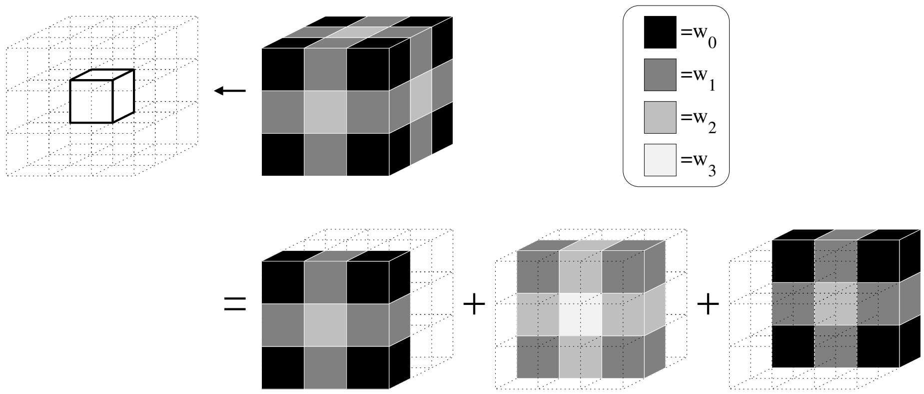


Figure 5.2: A 27-point Stencil. The center element in a $3 \times 3 \times 3$ block of elements is replaced by a weighted sum of its neighboring values. Typically, the weights are based on an element's distance from the center element, as shown in this diagram.

5.1.2 Hierarchical Arrays

The main data structure used by multigrid algorithms like MG is the *hierarchical array*. In its most basic form, the hierarchical array is simply a data structure with multiple *levels*, each of which is implemented using an array with half as many elements per dimension. In an array language like ZPL, a hierarchical array could be represented by a set of regions, each of which has either an upper bound half as big as the previous, or a stride that is twice as big (Figure 5.3).

In a good multigrid implementation, the size of the finest grid and the number of levels should ideally be specifiable by the programmer at execution time, to allow a reasonable amount of flexibility. In ZPL, the obstacle to doing so is the representation of an arbitrary number of regions, each of which has a unique upper bound or stride. As the language has been described so far, this cannot be done, since it provides no means for statically-sized groups of regions, let alone dynamically-sized. While it would be possible to use recursion to dynamically create the levels of the hierarchical array at execution time, this solution is not general and prevents the allocation of a persistent hierarchical array of data.

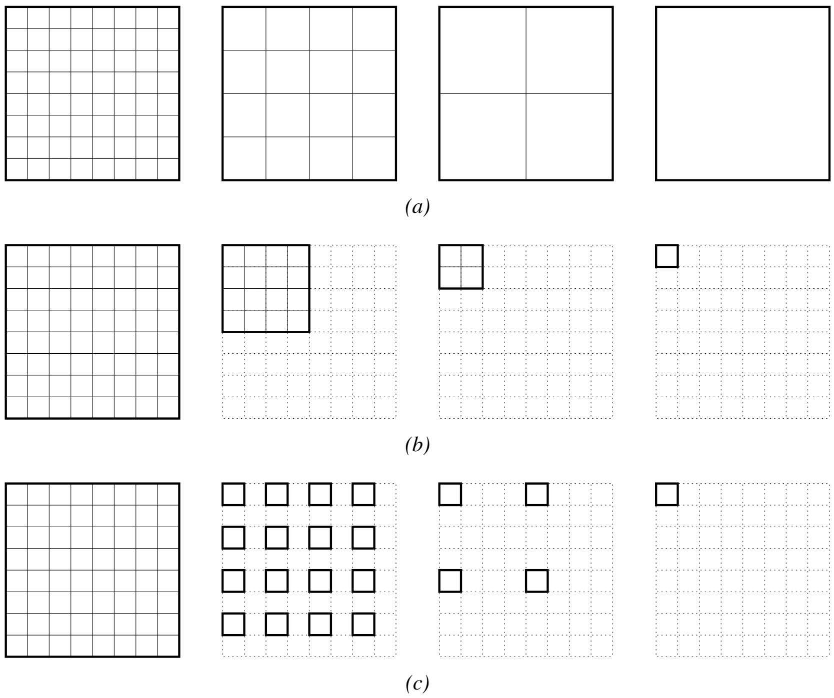


Figure 5.3: Hierarchical Array Implementations. (a) The conceptual view of a hierarchical array. (b) An implementation in which the bounds of the array are halved at each level in the hierarchy. (c) An alternative implementation in which the bounds of the array are kept constant, but the stride is doubled for each level.

A very poor hierarchical array implementation could be done by adding an additional dimension to describe the levels of the hierarchy. For example, the following 4D region coarsely describes the three-dimensional hierarchical array used by MG:

```
region RGrid = [1..nx, 1..ny, 1..nz, 1..num_levels];
```

Though this describes too many indices at the coarser levels of the hierarchy, shattered control flow or masks can be used to limit the indices to the required subset.

In spite of the fact that it could work, this implementation should be avoided at all costs due to the number of extraneous indices at the coarser levels of the hierarchy. Declaring an array over this region would result in the allocation of $n^3 \cdot \log n$ data values, assuming that $nx = ny = nz = n = 2^{num\_levels}$. In contrast, an efficient hierarchical array implementation should only require $\Theta(n^3)$ elements to be allocated. This solution is therefore unacceptable and serves as motivation for the solution described in the following section.

5.2 ZPL's Multi-Concepts

To solve the problem of creating a set of related regions or arrays, ZPL programmers are given the means to declare a parameterized group of ZPL concepts such as regions, arrays, and directions. For example, the *multi-region* allows programmers to declare a parameterized collection of regions whose attributes can vary with the parameter's value. Similarly, multi-arrays and multi-directions can be created, whose properties vary within the group. Before continuing, it should be stated that these *multi-concepts* have a number of problems that make them less-than-ideal as a general computing concept. These issues will be addressed at the end of this chapter, but will be ignored until that point.

5.2.1 Multi-Regions

As an example of a multi-region declaration, the following line of code creates a set of increasingly small 1D regions:

```
region MR{0..k} = [1+{} .. 1000-{}];
```

This declaration creates $k + 1$ regions, numbered $0 \ldots k$. Each pair of empty braces ({ }) in the region's definition is replaced by its parameter index when the definition is evaluated. Thus, each instance of region `MR{ }` has its upper and lower bounds adjusted by its index. For example, `MR{0}` is equivalent to the region `[0..1000]`, `MR{1}` is equivalent to `[1..999]`, and `MR{i}` represents `[i..1000-i]`.

Declarations of arrays, directions, and other variables may also be parameterized in this way. When multi-concepts are defined in terms of other multi-concepts, they may inherit the parameter range from that object rather than specifying it explicitly. For example, the following declaration creates a set of $k + 1$ arrays corresponding to the regions above:

```
var MA{}: [MR{}] double;
```

5.2.2 *Hierarchical Arrays and Multi-Regions*

ZPL programmers can use multi-regions to create hierarchical arrays simply by appropriately parameterizing a region's upper bound or stride. For example, the declarations in Listing 5.1 create two sets of 1D hierarchical regions and arrays. The first region declaration uses an upper bound that shrinks by half at each level, whereas the second strides each level by twice the amount of the previous. Each approach is legal and each describes a hierarchical array with the appropriate number of elements. However, the approaches have somewhat different implications when it comes to communication and load balancing.

Communication Considerations

In terms of communication, note that to project two elements from a finer level to a coarser level in ZPL, the first approach would require the remap operator, since the indices of interacting elements are not separated by a constant:

```
[MRShr{i}] MAShr{i} := proj(MAShr{i-1}#[Index1*2-1],
                            MAShr{i-1}#[Index1*2]);
```

Listing 5.1: Two Implementations of Hierarchical Arrays in ZPL

```
region MRShr{0..num_levels} = [1..(n/2^{})];
       MRStr{0..num_levels} = [1..n] by [2^{}];

var MAShr{}: [MRShr{}] double;
    MAStr{}: [MRStr{}] double;
```

In contrast, the strided approach keeps interacting elements near each other in the index space, admitting a solution that only requires the @ operator:

```
[MRStr{i}] MAStr{i} := proj(MAStr{i-1},
                            MAStr{i-1}@step{i-1});
```

Since the stencils of multigrid methods like MG ought to be implemented using point-to-point communication, the second expression of the projection is preferable.

Load-Balancing Considerations

The second consideration is the load balancing of the hierarchical array's levels. Note that if a single blocked distribution is used for every level of `MRShr`, approximately $3/4$ of the processors would own no indices at each successive level of the hierarchy (Figure 5.4a). To keep the elements distributed across all processors, a different distribution would be required for each level (Figure 5.4b). In contrast, the fact that `MRStr` has the same lower and upper bounds at each level means that a single distribution can distribute all of its levels as evenly as possible (Figure 5.4c).

Due to both the communication and load balancing issues, the strided approach is preferred when implementing hierarchical arrays in ZPL.

5.2.3 Semantic Sugar for Multi-Concepts

As even the short 1D examples above show, stencil computations involving multi-regions and multi-arrays tend to use simple variations of the parameter indices over and over again

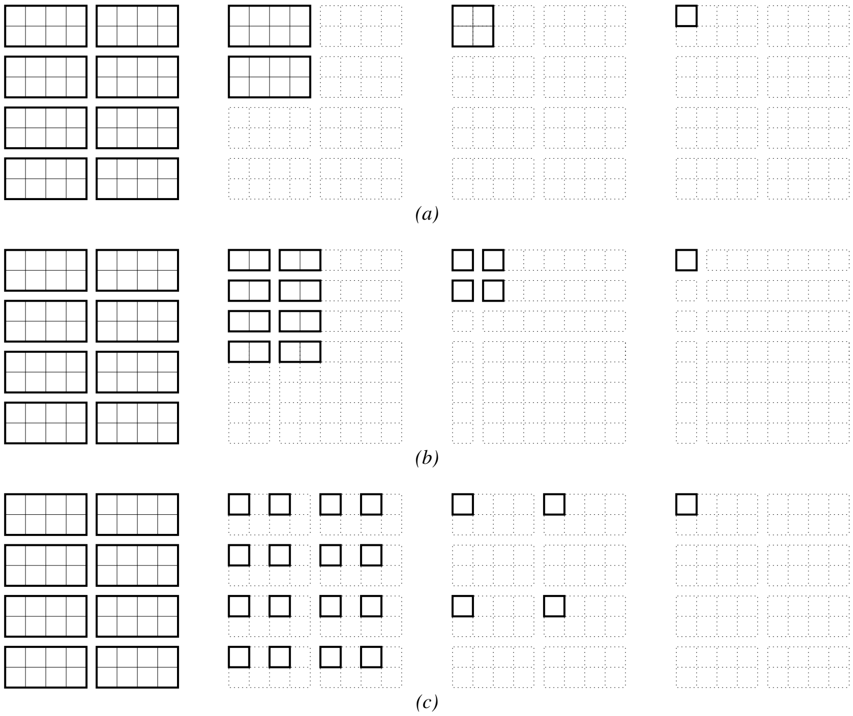


Figure 5.4: Load Balancing Hierarchical Arrays. Consider the different distribution possibilities for each hierarchical array implementation on a 4×2 processor grid. (a) For implementations that shrink the global bounds of each array, using a single distribution for all levels will cause a load imbalance at all levels of the hierarchy other than the finest. (b) Using a unique distribution per level in the hierarchy can fix this problem, but complicates the distribution somewhat. (c) With a strided implementation, a good distribution for the finest level will also give the best load balance possible at all coarser levels.

in a single statement. To help with this problem, ZPL allows the index for a multi-array to be elided and inherited from the enclosing region scope of appropriate rank (this causes an error if the enclosing region scope is not a multi-region). Similarly, multi-directions can inherit their indices from the region or array to which they are applied when using the region operators or an @ operator. Using these conventions, the simple stencil on `MRStr` could be rewritten as follows:

```
[MRStr{i}] MAStr{} := proj(MAStr{i-1},
                            MAStr{i-1}@step{});
```

To further aid with the problem, relative indices can be used to adjust the inherited index by a constant offset. The syntax for this type of reference has never been formalized, but the current syntax is the unsightly use of empty curly-braces to refer to the inherited index. For example, the previous line could be rewritten as follows:

```
[MRStr{i}] MAStr{} := proj(MAStr{{}-1},
                            MAStr{{}-1}@step{});
```

Although this does not save any keystrokes in this example, indexing expressions with longer identifiers typically benefit.

5.3 ZPL MG Implementation

Given ZPL's multi-concepts, implementing MG in ZPL is fairly straightforward. The implementation described here (referred to as ZPL MG) follows the NAS implementation as closely as possible.

5.3.1 Declarations

ZPL MG Directions

The directions declared in ZPL MG are simply the 26 non-degenerate vectors described by $(-1\ldots1)\times(-1\ldots1)\times(-1\ldots1)$, for use in the 27-point stencils at all levels of the hierarchy. These are declared using ZPL's `scaledby` direction operator which scales a direction by an unsigned integer value. In addition, for clarity of naming, direction $(-1,-1,-1)$ is redeclared as the step for the strided multi-region.

Listing 5.2 shows the complete set of direction declarations. It should be noted that many of the traditional benefits of ZPL are absent in these declarations. In particular, the use of so many directions negates the benefit of naming them since the names are now as descriptive as the directions themselves. Furthermore, the similarity in the directions' names and definitions is as tedious and error-prone as traditional array indexing. To some extent, this is a symptom of the benchmark itself, but it also represents a failing in the language that will be addressed in this chapter's discussion section.

ZPL MG Regions

The only region required by ZPL MG is a hierarchical region that describes the problem space using striding:

```
region Level{} = [1..nx, 1..ny, 1..nz] by step{};
```

Typically the benchmark is run using equal values for `nx`, `ny`, and `nz`, though this declaration allows different values to be specified at runtime.

ZPL MG Arrays

The NAS implementation of MG uses an input array at the finest level of the hierarchy (`V`) and two hierarchical arrays (`U` and `R`). These are declared in ZPL MG trivially as follows:

```
var V: [Level{0}] double;
    U{}: [Level{}] double;
    R{}: [Level{}] double;
```

5.3.2 A Single Iteration

A single iteration of NAS MG consists of a round of projections, the computation of an approximate solution, and then a series of interpolations using filtering. This is expressed

Listing 5.2: Directions used by ZPL MG

```
direction
        dir100{0..num_levels} = [ 1, 0, 0] scaledby 2^{};
        dirN00{0..num_levels} = [-1, 0, 0] scaledby 2^{};
        dir010{0..num_levels} = [ 0, 1, 0] scaledby 2^{};
        dir0N0{0..num_levels} = [ 0,-1, 0] scaledby 2^{};
        dir001{0..num_levels} = [ 0, 0, 1] scaledby 2^{};
        dir00N{0..num_levels} = [ 0, 0,-1] scaledby 2^{};

        dir110{0..num_levels} = [ 1, 1, 0] scaledby 2^{};
        dir1N0{0..num_levels} = [ 1,-1, 0] scaledby 2^{};
        dirN10{0..num_levels} = [-1, 1, 0] scaledby 2^{};
        dirNN0{0..num_levels} = [-1,-1, 0] scaledby 2^{};
        dir101{0..num_levels} = [ 1, 0, 1] scaledby 2^{};
        dir10N{0..num_levels} = [ 1, 0,-1] scaledby 2^{};
        dirN01{0..num_levels} = [-1, 0, 1] scaledby 2^{};
        dirN0N{0..num_levels} = [-1, 0,-1] scaledby 2^{};
        dir011{0..num_levels} = [ 0, 1, 1] scaledby 2^{};
        dir01N{0..num_levels} = [ 0, 1,-1] scaledby 2^{};
        dir0N1{0..num_levels} = [ 0,-1, 1] scaledby 2^{};
        dir0NN{0..num_levels} = [ 0,-1,-1] scaledby 2^{};

        dir111{0..num_levels} = [ 1, 1, 1] scaledby 2^{};
        dir11N{0..num_levels} = [ 1, 1,-1] scaledby 2^{};
        dir1N1{0..num_levels} = [ 1,-1, 1] scaledby 2^{};
        dir1NN{0..num_levels} = [ 1,-1,-1] scaledby 2^{};
        dirN11{0..num_levels} = [-1, 1, 1] scaledby 2^{};
        dirN1N{0..num_levels} = [-1, 1,-1] scaledby 2^{};
        dirNN1{0..num_levels} = [-1,-1, 1] scaledby 2^{};
        dirNNN{0..num_levels} = [-1,-1,-1] scaledby 2^{};

        step{0..num_levels} = [-1,-1,-1] scaledby 2^{};
```

in ZPL as shown in Listing 5.3. Note that blank curly-brace references are used heavily throughout this code to inherit multi-concept indices. As in NAS MG, each of the 27-point stencils is implemented using a procedure that takes individual levels of the array as parameters. The following section describes these stencil routines.

5.3.3 The Four Stencils

All of the ZPL MG stencils are written using wrap-@ references to describe the 27 array references using periodic boundary conditions. As in the direction declarations, it should be noted that while these stencils are very clear, unambiguous representations of the operations, they are also very tedious and error-prone simply due to the size of the stencil. For 27-point stencils, these statements are manageable, but when larger stencils such as those used by the fast multipole method [ED95] are used, any typos on the programmer's part would be very difficult to debug. We return to this problem at the end of the chapter, concentrating for now on the implementation of each stencil.

The rprj3 *Stencil*

The projection stencil for ZPL MG is shown in Listing 5.4. The 27 wrap-@ references are grouped by weight, summed, and then multiplied by the appropriate value. The covering region and sizes of `S` and `R` are all inherited from the callsite to make the code work at any level of the hierarchy. Note that the uses of blank curly-braces cause the directions in `R`'s wrap-@ references to inherit its scale and refer to its adjacent elements.

The psinv *and* resid *Stencils*

The *psinv* and *resid* stencils are almost identical to *rprj3*, except that some terms are weighted by 0 and therefore elided to avoid unnecessary communication and computation. The fact that these stencils are used to compute within a level of the hierarchy is evident only when viewing the stencils' callsites in Listing 5.3. In particular, the calls to

Listing 5.3: A Single Iteration of ZPL MG

```
procedure iterate();
var lvl: integer;
begin
  ----------------------------------------------------------
  -- Project to coarsest level
  ----------------------------------------------------------
  for lvl := 1 to num_levels do
    [Level{lvl}] rprj3(R{}, R{{}-1});
  end;

  ----------------------------------------------------------
  -- Compute an approximate solution on the coarsest grid
  ----------------------------------------------------------
  [Level{num_levels}] begin
                        U{} := 0.0;
                        psinv(U{}, R{});
                      end;

  ----------------------------------------------------------
  -- Interpolate solution
  ----------------------------------------------------------
  for lvl := num_levels-1 downto 1 do
    [Level{lvl}] begin
                   U{} := 0.0;
                  [Level{{}+1}] interp(U{{}-1}, U{});
                   resid(R{}, R{}, U{});
                   psinv(U{}, R{});
                 end;
  end;

  [Level{0}] begin
              [Level{1}] interp(U{{}-1}, U{});
               resid(R{}, V, U{});
               psinv(U{}, R{});
             end;

  ----------------------------------------------------------
  -- Compute residual against input
  ----------------------------------------------------------
  [Level{0}] resid(R{}, V, U{});
end;
```

Listing 5.4: The *rprj3* Stencil in ZPL

```
procedure rprj3(var S, R: [ , , ] double);
begin
  S := 0.5000 * R
     + 0.2500 * (R@^dir100{} + R@^dir010{} + R@^dir001{} +
                 R@^dirN00{} + R@^dir0N0{} + R@^dir00N{})
     + 0.1250 * (R@^dir110{} + R@^dir1N0{} + R@^dirN10{} + R@^dirNN0{} +
                 R@^dir101{} + R@^dir10N{} + R@^dirN01{} + R@^dirN0N{} +
                 R@^dir011{} + R@^dir01N{} + R@^dir0N1{} + R@^dir0NN{})
     + 0.0625 * (R@^dir111{} + R@^dir11N{} + R@^dir1N1{} + R@^dir1NN{} +
                 R@^dirN11{} + R@^dirN1N{} + R@^dirNN1{} + R@^dirNNN{});
end;
```

Listing 5.5: The *psinv* Stencil in ZPL

```
procedure psinv(var U, R: [ , , ] double);
begin
  U += c[0] *  R
     + c[1] * (R@^dir100{} + R@^dir010{} + R@^dir001{} +
               R@^dirN00{} + R@^dir0N0{} + R@^dir00N{})
     + c[2] * (R@^dir110{} + R@^dir1N0{} + R@^dirN10{} + R@^dirNN0{} +
               R@^dir101{} + R@^dir10N{} + R@^dirN01{} + R@^dirN0N{} +
               R@^dir011{} + R@^dir01N{} + R@^dir0N1{} + R@^dir0NN{});
end;
```

Listing 5.6: The *resid* Stencil in ZPL

```
procedure resid(var R, V, U: [ , , ] double);
begin
  R := V - a[0] *  U
         - a[2]*(U@^dir110{} + U@^dir1N0{} + U@^dirN10{} + U@^dirNN0{} +
                 U@^dir101{} + U@^dir10N{} + U@^dirN01{} + U@^dirN0N{} +
                 U@^dir011{} + U@^dir01N{} + U@^dir0N1{} + U@^dir0NN{})
         - a[3]*(U@^dir111{} + U@^dir11N{} + U@^dir1N1{} + U@^dir1NN{} +
                 U@^dirN11{} + U@^dirN1N{} + U@^dirNN1{} + U@^dirNNN{});
end;
```

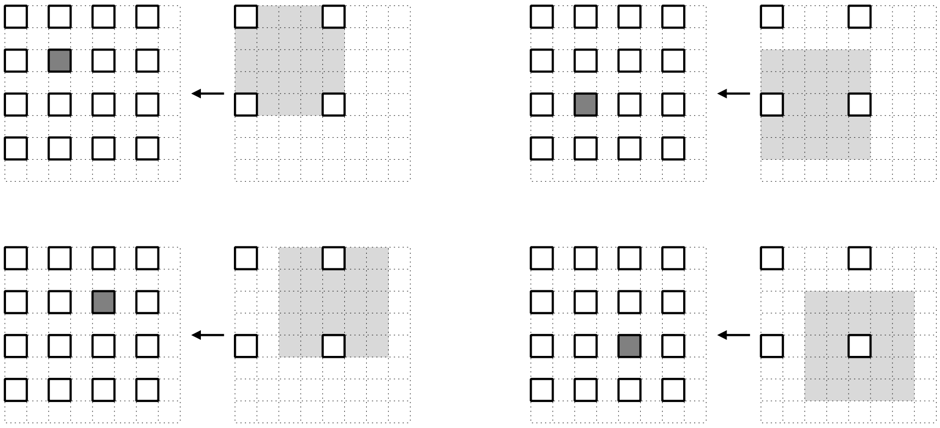


Figure 5.5: MG's *interp* Stencil. This figure shows a 2D illustration of MG's *interp* stencil. For each of the four applications of the stencil shown, the grey element on the right-hand side of the assignment indicates the value being written, and the grey box on the right-hand side shows the extent of each 3×3 stencil. Note that a different set of neighboring values are referenced for each application, based on the relative alignment between the fine and coarse grid values. These values are averaged in MG's *interp* stencil.

rprj3 pass in arrays from adjacent levels of the hierarchy whereas the calls to *psinv* and *resid* pass arrays from the same level of the hierarchy. The fact that the procedures look so similar implies that ZPL does a good job of abstracting the 27-point stencil. In fact, a single procedure could be used for all three stencils if the weights were sent in as an argument and the zero-weight optimizations were not considered important enough to perform by hand.

The interp *Stencil*

The *interp* stencil is slightly different due to the fact that interpolation has different characteristics. Conceptually, MG's interpolation stencil can be thought of as a 27-point averaging stencil whose scale matches the finer level of the hierarchy. This means that as the 3×3 stencil is used to read the coarse level of the hierarchy, a number of elements between one and eight will be read, depending on the fine element being assigned (Figure 5.5). Since ZPL's array references are only legal when referring to indices for which an array

Listing 5.7: The *interp* Stencil in ZPL

```
procedure interp(var R, S: [ , , ] double);
begin
  R += S;
  R@dirN00{{}-1} += .500*(S + S@^dirN00{});
  R@dir0N0{{}-1} += .500*(S + S@^dir0N0{});
  R@dir00N{{}-1} += .500*(S + S@^dir00N{});
  R@dirNN0{{}-1} += .250*(S + S@^dirN00{} + S@^dir0N0{} + S@^dirNN0{});
  R@dirN0N{{}-1} += .250*(S + S@^dirN00{} + S@^dir00N{} + S@^dirN0N{});
  R@dir0NN{{}-1} += .250*(S + S@^dir0N0{} + S@^dir00N{} + S@^dir0NN{});
  R@dirNNN{{}-1} += .125*(S + S@^dirN00{} + S@^dir0N0{} + S@^dir00N{} +
                          S@^dirNN0{} + S@^dir0NN{} + S@^dirN0N{} +
                          S@^dirNNN{});
end;
```

is explicitly allocated, eight statements have to be written to describe each of the possible alignments. Listing 5.7 shows an implementation of this stencil.

Looking at the callsite, note that the procedure's covering region is always one level coarser than R's actual parameter. This causes each assignment to write to every 8^{th} element. The procedure then uses a left-hand side wrap-@ reference to shift each assignment and describe one of the possible alignments. Note that directions that are one level coarser than the grid must be used to achieve this. The references to the fine grid are then averaged based on the number of elements being interpolated.

The *interp* stencil could also be written in a few other ways. For example, a shattered conditional could be used to sort out the 8 cases based on the values of the `Index`*i* variables, eliminating the need for using the coarser region and directions. The approach shown here was chosen primarily due to its lack of control flow, though a comparison of the two approaches would be interesting.

5.3.4 Comparing the ZPL and NAS Implementations

For the most part, the ZPL implementation of MG follows the NAS version very closely. The data structures and procedural decomposition are identical, and ZPL's performance

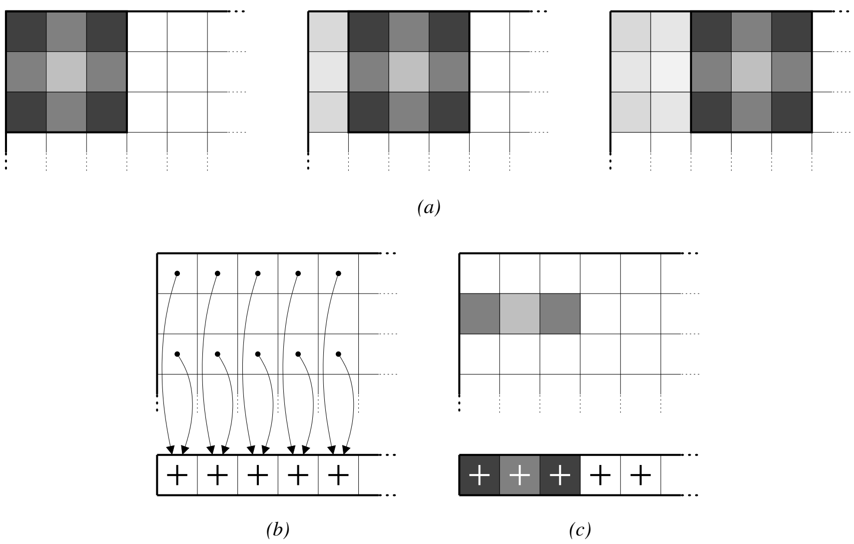


Figure 5.6: The NAS MG Stencil Optimization. This figure demonstrates the style of stencil optimization used by the NAS MG benchmark using a 2D example. (a) Adjacent applications of the stencil. Note that the sum of the rightmost top and bottom values in the first application are re-used in the second and third. (b) To take advantage of this, a vector of sums is pre-computed. (c) To compute an application of the stencil, the sum values are weighted and added to the weighted values from the middle row. Note that the benefits for 3D stencils are even greater.

model implies that the communication required by the code matches that specified using MPI calls in the NAS version.

The chief difference between the two codes is that the NAS implementation utilizes clever hand-optimizations in each of the 27-point stencils. These optimizations compute arithmetic subexpressions that are reused in adjacent applications of a stencil, caching the values to minimize redundant FLOPs (Figure 5.6). This significantly improves performance at the cost of obscuring the source code's intent. In terms of performance, it turns out to be a wise decision: current Fortran compilers do not perform this transformation when the stencils are written in a naive manner, resulting in performance degradations of 17% or more.

It is difficult to write the same optimizations in ZPL due to the fact that the programmer is not writing scalar code and cannot specify code that executes between the implementing loops of an m-loop. However, it turns out that performing the optimization within the ZPL compiler is straightforward, given the relative strides of the covering region, arrays, and directions. This information is readily computable given ZPL's high-level semantics, and allows not only for the optimization of stencils as simple as those found in MG, but also a variety of more complicated stencil patterns [DCS01]. The result is that programmers can express their algorithms in a straightforward manner without sacrificing the performance that can be achieved using a scalar local-view language.

Section 5.5 evaluates ZPL MG against NAS MG and three implementations written in other languages. Before describing that study, however, the next section provides a brief description of the implementation of ZPL's multi-concepts.

5.4 Implementation of Multi-Concepts

5.4.1 Basic Implementation

Implementing ZPL's multi-concepts is fairly trivial. For each multi-direction, multi-region, or multi-array declared by the user, a vector of that object's descriptors is allocated, indexed by the identifier's parameter range. Setting up this vector of descriptors is simply a matter of wrapping a loop around the traditional descriptor initialization code and substituting the loop's index for any blank curly-braces found in the definition. References to multi-concepts are simply generated as accesses to the appropriate descriptor within the vector.

In order to support inheritance of multi-region and multi-array indices, an extra integer field must be added to the region and array descriptors to store their parameter index for cases in which it cannot be determined at compile time. For example, in the ZPL implementation of *resid*, the parameter index of array `U` is checked dynamically to use the correct version of the directions in its twenty-six @-references.

5.4.2 Challenges to Fluff Analysis

The primary challenge in the implementation of multi-concepts is that they complicate fluff analysis. For example, imagine that the MG benchmark contained the following @-reference:

```
...A{i}@dir100{j} ...
```

If the compiler cannot determine the relationship between i and j in this expression, it will not be able to determine the number of additional rows of data to allocate for `A{i}`'s fluff. The conservative approach would be to assume that $j = num\_levels$, but this can result in an exponential amount of space being wasted due to the doubling of the direction vectors at each level of the hierarchy.

A second approach would be to use runtime checks to determine if an array's fluff is insufficient and expand it as necessary. Aside from the runtime overheads of performing the checks and reallocating the arrays, this approach is sound.

A third option is to insert a temporary array for this expression whose size is determined by the enclosing region scope. The temporary array would then serve as the storage to cache non-local values. The primary disadvantages to this scheme are the memory that it requires and the fact that the rest of the array would have to be filled with shifted, local values of `A`. Alternatively, the m-loop for the original expression could be split into two loops, one that accesses the local shifted values of `A` and the second to read the remote values from the temporary array.

In hierarchical codes, such @-references are rare due to the fact that the directions applied to a level of the hierarchical array will either match its level or correspond to an adjacent level. In these cases, the minimal amount of required fluff can be determined statically without difficulty. Currently, more general @-references have not been implemented, causing a compiler warning to be generated for expressions like the one above. Future work should consider each of the implementation options to determine which has the smallest impact on the common case.

5.5 Evaluation

To evaluate the ZPL implementation of hierarchical arrays, this section compares ZPL MG against versions of the benchmark written in other languages in terms of clarity, performance, and portability.

5.5.1 Methodology

One of the advantages of using MG to evaluate a language's support for hierarchical array computation is that it is such a popular benchmark that many other languages have performed similar studies. This provides a number of implementations for comparison with ZPL MG. In searching for candidate implementations of MG, this study had two requirements. The first was motivated by the emphasis in NPB2 on portability rather than algorithmic cleverness [BHS$^+$95]. In particular, this study required each implementation to follow the spirit of the NAS implementation as closely as the source language and compiler would allow. For example, an implementation could not use a sparse array to store the twenty nonzero values in the $n \times n \times n$ input array, as this would constitute a radical departure from the original algorithm. However, it could certainly change loop nest iteration orders, strive to minimize communication, *etc.* All of the implementations used in this study met this requirement without modification.

The second requirement was that each benchmark's programmer had to be familiar with the language and compiler in question. This goal was designed to ensure that our own ignorance of a language or compiler would not adversely affect that language's evaluation. This requirement was easy to meet due to MG's popularity as a benchmark: all of the implementations in this study were coded by the language's development team with the exception of the HPF version which was written at NASA Ames.

As in any programming context, there is a potential tension between clarity and performance. In general, the implementors of each benchmark strove to optimize for performance without unduly compromising clarity. For example, the original NAS implementation of MG relies on F77 loops and indexing rather than the equivalent (and arguably more expressive) F90 slice notation. This decision was motivated by the desire to ensure that the crucial stencil code would be implemented efficiently rather than relying on each platform's compiler to get it right [vdW00, BHS+95]. Our efforts to make the code more expressive using array statements proved to adversely affect performance, as anticipated by the code's authors. Moreover, slice notation did not improve the code's clarity tremendously, given the number of characters required to express each 3D array slice.

Since the results in this section evaluate specific implementations of MG using a specific set of compilers, they may not necessarily reflect the maximum potential of each compiler or language. In truth, one would be hard-pressed to design an experiment that would. However, the requirements above strive to ensure that while one might conceive of a faster or clearer implementation, the benchmarks used here represent reasonable examples of how knowledgeable, performance-minded programmers might implement a specific multigrid algorithm in each language using current compiler technology.

5.5.2 MG Implementations

In addition to the NAS MG and ZPL MG implementations, reasonable versions of the MG benchmark were found for HPF, CAF, and SAC. This section gives a brief description of each implementation, emphasizing ways in which they differ from the original NAS version.

F77+MPI

The F77+MPI code used in this study is the NAS implementation, version 2.3. It serves as the baseline for the study.

ZPL

The ZPL implementation of MG is the version described in Section 5.3. As described there, it follows the NAS version as closely as possible, using multi-regions to implement the hierarchical arrays, but omitting the hand-coded stencil optimization.

HPF

The HPF implementation is obtained from NASA Ames and stems from an effort to implement all of the NAS benchmarks in HPF [FJY98]. PGI identifies this implementation as the best-known version of MG for their compiler [Por99], which serves as the HPF compiler in our experiments. This implementation follows the NAS implementation very closely with one major exception: HPF has no specific support for hierarchical arrays apart from F90's traditional array language concepts. Thus, in order to specify a hierarchical array at the global view such that one may iterate over its levels and parallelize them effectively, an array of pointers to dynamically allocated arrays should be used [FJY98]. Although F90 supports such an approach, this feature is currently unsupported by most HPF compilers. Thus, the benchmark's authors chose to represent their hierarchical arrays using a 4D implementation as described in Section 5.1.2 [FJY98, Fru00].

The 4D implementation has the effect of wasting memory and of walking through it in large strides for coarser levels of the hierarchy. These factors will impact performance due to the memory hierarchy. The 4D hierarchical array also requires extensive use of HPF's `HOME` directive in order to align the arrays in a distributed and load-balanced manner. Other than this issue, the implementation is extremely true to the NAS version and includes its hand-coded stencil optimizations. Future work might consider how these 4D arrays could be eliminated in other HPF compilers [GMS$^+$95, BCG$^+$95] or by using HPF extensions for sparse or irregular problems [Hig97].

CAF

The CAF implementation was written using the NAS implementation as a starting point. Since both of these approaches use a local per-processor view and Fortran as their base language, the CAF implementation simply involved removing the NAS MPI calls and replacing them with the equivalent co-array syntax. Although solutions more tailored to CAF could be imagined, this implementation is as true to the original NAS version as one could imagine and was a fairly trivial port for its authors.

SAC

The SAC implementation of MG comes as part of the SAC distribution [SAC01] and forms the most radical departure from the NAS implementation. SAC's functional nature makes it natural to express hierarchical applications using a recursive approach. This allows programmers to implement each level of a hierarchal array using local arrays whose dimensions are each half as big as the incoming arrays. While this approach is natural for multigrid solvers like MG, it would be insufficient for techniques like adaptive mesh refinement [LM90] in which the coarse levels of the hierarchy often need to be preserved from one iteration to the next.

SAC's recursive array specifications differ from the iterative approach in the NAS implementation in which the hierarchical arrays are allocated at the outset of the program and reused thereafter. In spite of this difference, we used the official SAC implementation of MG rather than implementing an iterative solution for fear that a shift in paradigm would neither be in the spirit of SAC nor in its best interests performance-wise.

The NAS stencil optimization could not be cleanly hand-coded in SAC without disabling other essential optimizations, so all stencils are expressed in their simplest but less optimal form.

Table 5.2: Summary of the MG Implementations

Language	*Author*	*# Processors unbound?*	*Problem Size unbound?*	*Data Distribution*	*Contains Stencil Opt?*
F77+MPI	NAS	no (2^x)	no (2^x)	3D blocked	hand-coded
HPF	NAS	yes	no	1D blocked[1]	hand-coded
CAF	CAF group	no (2^x)	no (2^x)	3D blocked	hand-coded
SAC	SAC group	yes	yes[2] (2^x)	1D blocked	no
ZPL	ZPL group	yes	yes	3D blocked	compiler-generated

Summary

Table 5.2 summarizes the versions of the benchmark used in these experiments. The *Author* column indicates the origin of each implementation. The next two columns indicate whether the number of processors and/or problem size can be specified at execution time rather than fixed at compile-time. If the implementation also constrains the number of processors or problem size to be a power of two, this is indicated in parenthesis. The *Data distribution* column indicates the way in which arrays are distributed across the processor set. The last column indicates whether or not the implementation includes the hand-coded stencil optimizations as in the NAS version.

Note that the ZPL implementation is the only one which allows the problem size and number of processors to be specified at execution time. While a completely dynamically-specified version of MG *could* be written in each language, doing so could adversely affect clarity and/or performance by making the code more general and providing less informa-

[1]The implementors found that distributing more than one dimension hurt performance in HPF, so chose to distribute just one [FJY98].

[2]Though the problem size may be specified dynamically, the code is written such that only a few problem sizes are possible.

tion to the compiler. For example, the NAS and CAF implementations use their knowledge of the number of processors and problem size to allocate hierarchical arrays statically. Similarly, by constraining the problem sizes in SAC, inlining and loop unrolling optimizations are enabled. The point here is not to argue whether or not such constraints are reasonable, but rather to highlight that the more dynamic implementations start with a certain disadvantage.

5.5.3 Evaluation of Clarity

The clarity of the MG implementations is determined both by performing quantitative line counts, and by qualitatively evaluating the code itself.

Quantitative Analysis

Each line of each benchmark is classified as being one of four types: *declarations*, *communication*, *computation*, and *non-essentials*. As in the experiments of Chapter 2, declarations include all lines of code that are used to declare variables, constants, functions, and other identifiers. Communication lines are those that are used for interprocessor data transfer or synchronization. Code related to comments, initialization, timings, and I/O is considered non-essential and excluded from the analysis. The remaining lines of code form the timed, computational kernel of the benchmark and are considered computation. Because line counts will vary somewhat due to a programmer's style, these figures constitute only a coarse indication of clarity.

Figure 5.7 gives a summary of the quantitative evaluation, showing the number of essential lines of code and how they break down into the different categories. The most immediate observation is that languages with a local view of computation require 2–8 times as many lines of code as those providing a global view, and that the majority of these lines implement communication. Inspection of this communication reveals that it is not only lengthy, but also quite intricate in order to handle the exceptional cases that are required

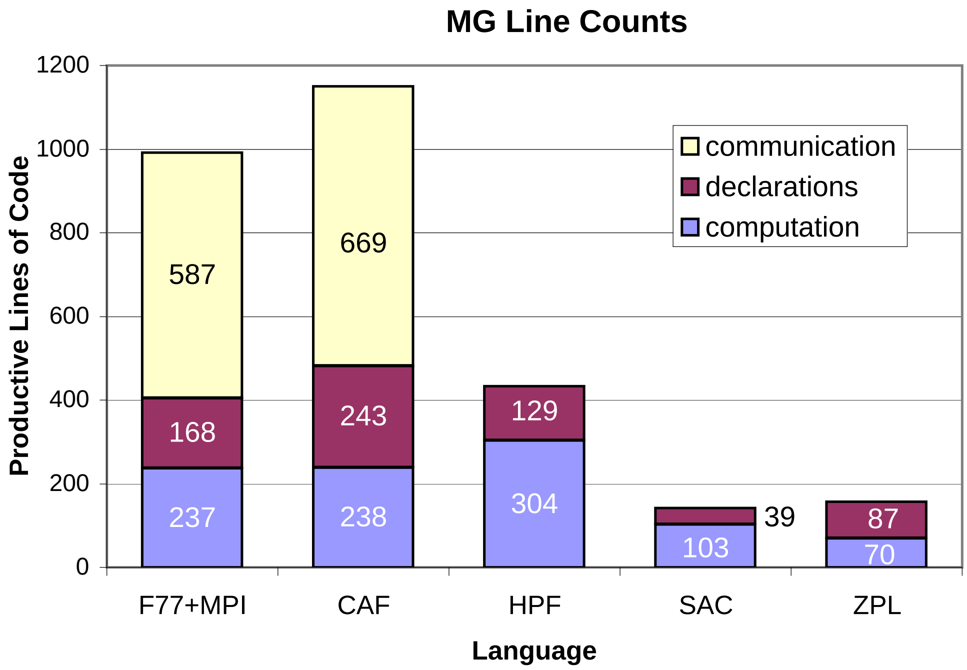


Figure 5.7: MG Implementation Linecounts. This graph gives an indication of the number of useful lines of code in each implementation of MG. *Communication* indicates the number of lines devoted to communication and synchronization. *Declarations* indicates code used to declare variables, constants, and identifiers. *Computation* indicates the number of lines used to express the computation itself. Note that the local-view languages require 2–8 times as many lines of code, due primarily to the fact that programmers must explicitly manage communication.

to maintain a processor's local boundary values in three dimensions at all levels of the hierarchy. The difference in communication counts between F77+MPI and CAF stems from MPI's built-in support for collective communication routines such as broadcasts and reductions. In the CAF version, such operations are manually implemented as functions, adding to both its communication and declaration counts.

The next thing to note is that of the global-view languages, HPF is considerably lengthier than either SAC or ZPL. This is an unfortunate consequence of the extra code and directives required to implement its 4D hierarchical arrays as described in the previous section. Examples of such code can be seen in lines 14–21 and 31–44 of Listing 5.9. Though the SAC and ZPL line counts are similar to one another, SAC takes a 30-line computation hit by explicitly replicating top-level function calls for each problem size to enable inlining and unrolling. On the other hand, it requires 48 fewer lines for declarations due to its support for automatic variable declarations.

In terms of computation, one observes that the F77+MPI and CAF implementations have 2–3 times the number of lines as the SAC and ZPL benchmarks. This can largely be attributed to the looping and indexing overheads associated with hand-coding the stencil optimization. Although converting the stencils in these implementations to an unoptimized slice notation brings the computation line count closer to that of SAC and ZPL, this change causes the performance to degrade significantly, and the overall line counts and complexity of the implementations are still dominated by communication code.

Qualitative Analysis

The qualitative evaluation consists of reading through the code by hand and looking to see if the line counts seem inversely related to the clarity with which each implementation expresses the algorithm. Our conclusion is that they are. Listings 5.8–5.10 show the *rprj3* stencil in F77+MPI/CAF, HPF, and SAC respectively. Note that recurring idioms in the HPF code are eliminated for brevity. Furthermore, the communication for the F77+MPI and CAF versions is omitted since it constitutes several hundred additional lines of code.

Listing 5.8: The F77 and CAF Implementation of *rprj3*

```
      subroutine rprj3( r,m1k,m2k,m3k,s,m1j,m2j,m3j,k )
      implicit none
      include 'mpinpb.h'
      include 'globals.h'

      integer m1k, m2k, m3k, m1j, m2j, m3j,k
      double precision r(m1k,m2k,m3k), s(m1j,m2j,m3j)
      integer j3, j2, j1, i3, i2, i1, d1, d2, d3, j
      double precision x1(m), y1(m), x2,y2

      if(m1k.eq.3)then
        d1 = 2
      else
        d1 = 1
      endif

      if(m2k.eq.3)then
        d2 = 2
      else
        d2 = 1
      endif

      if(m3k.eq.3)then
        d3 = 2
      else
        d3 = 1
      endif

      do  j3=2,m3j-1
         i3 = 2*j3-d3
         do  j2=2,m2j-1
            i2 = 2*j2-d2
            do j1=2,m1j
              i1 = 2*j1-d1
              x1(i1-1) = r(i1-1,i2-1,i3  ) + r(i1-1,i2+1,i3  )
     >                 + r(i1-1,i2,  i3-1) + r(i1-1,i2,  i3+1)
              y1(i1-1) = r(i1-1,i2-1,i3-1) + r(i1-1,i2-1,i3+1)
     >                 + r(i1-1,i2+1,i3-1) + r(i1-1,i2+1,i3+1)
            enddo
            do  j1=2,m1j-1
              i1 = 2*j1-d1
              y2 = r(i1,  i2-1,i3-1) + r(i1,  i2-1,i3+1)
     >           + r(i1,  i2+1,i3-1) + r(i1,  i2+1,i3+1)
              x2 = r(i1,  i2-1,i3  ) + r(i1,  i2+1,i3  )
     >           + r(i1,  i2,  i3-1) + r(i1,  i2,  i3+1)
              s(j1,j2,j3) =
     >            0.5D0 * r(i1,i2,i3)
     >          + 0.25D0 * ( r(i1-1,i2,i3) + r(i1+1,i2,i3) + x2)
     >          + 0.125D0 * ( x1(i1-1) + x1(i1+1) + y2)
     >          + 0.0625D0 * ( y1(i1-1) + y1(i1+1) )
            enddo
         enddo
      enddo

      j = k-1
      call comm3(s,m1j,m2j,m3j,j)

      return
      end
```

Listing 5.9: The HPF Implementation of *rprj3*

```
      extrinsic (HPF) subroutine rprj3(r,m1k,m2k,m3k,s,
     >                                 m1j,m2j,m3j,k)

      implicit none
      include 'globals.h'
      include 'rprj3.h'
      integer j3, j2, j1, i3, i2, i1, d1, d2, d3, j
      double precision x1(m), y1(m), x2,y2
      double precision w000(m1j,m2j,m3j), w100(m1j,m2j,m3j),
     >                 w010(m1j,m2j,m3j), w110(m1j,m2j,m3j),
     >                 w001(m1j,m2j,m3j), w101(m1j,m2j,m3j),
     >                 w011(m1j,m2j,m3j), w111(m1j,m2j,m3j)

!hpf$   align w000(i1,i2,i3) with r(2*i1,2*i2,2*i3)
!hpf$   align w100(i1,i2,i3) with r(2*i1-1,2*i2,2*i3)
!hpf$   align w010(i1,i2,i3) with r(2*i1,2*i2-1,2*i3)
!hpf$   align w110(i1,i2,i3) with r(2*i1-1,2*i2-1,2*i3)
!hpf$   align w001(i1,i2,i3) with r(2*i1,2*i2,2*i3-1)
!hpf$   align w101(i1,i2,i3) with r(2*i1-1,2*i2,2*i3-1)
!hpf$   align w011(i1,i2,i3) with r(2*i1,2*i2-1,2*i3-1)
!hpf$   align w111(i1,i2,i3) with r(2*i1-1,2*i2-1,2*i3-1)

      if(m1k.eq.3)then
        d1 = 2
      else
        d1 = 1
      endif

! TWO CONDITIONALS OF A SIMILAR FORM DELETED DUE TO SPACE CONSTRAINTS

!hpf$ independent, on home(w000(i1,i2,i3))
         do  i3=1,m3j-1
!hpf$ independent
            do  i2=1,m2j-1
               do i1=1,m1j-1
                 w000(i1,i2,i3) = r(2*i1,2*i2,2*i3)
     >                          + r(2*i1,2*i2+2,2*i3)
     >                          + r(2*i1+2,2*i2,2*i3)
     >                          + r(2*i1+2,2*i2+2,2*i3)
               end do
            end do
         end do

! 7 LOOP NESTS OF A SIMILAR FORM DELETED DUE TO SPACE CONSTRAINTS

         s(2:m1j-1,2:m2j-1,2:m3j-1) =
     >             0.5D0* w111(2:m1j-1,2:m2j-1,2:m3j-1)
     >      + 0.25D0 * (  w011(1:m1j-2,2:m2j-1,2:m3j-1)
     >               +    w011(2:m1j-1,2:m2j-1,2:m3j-1)
     >               +  w101(2:m1j-1,1:m2j-2,2:m3j-1)
     >               +  w101(2:m1j-1,2:m2j-1,2:m3j-1)
     >               +      w110(2:m1j-1,2:m2j-1,1:m3j-2)
     >               +      w110(2:m1j-1,2:m2j-1,2:m3j-1) )
     >      + 0.125D0 * ( w001(1:m1j-2,1:m2j-2,2:m3j-1)
     >               +    w001(2:m1j-1,1:m2j-2,2:m3j-1)
     >               +  w010(1:m1j-2,2:m2j-1,1:m3j-2)
     >               +  w010(1:m1j-2,2:m2j-1,2:m3j-1)
     >               +    w100(2:m1j-1,1:m2j-2,1:m3j-2)
     >               +    w100(2:m1j-1,1:m2j-2,2:m3j-1) )
     >     + 0.0625D0 * ( w000(1:m1j-2,1:m2j-2,1:m3j-2)
     >               +  w000(1:m1j-2,1:m2j-2,2:m3j-1)  )

       s(m1j,:,:) = s(2,:,:)
       s(1,:,:) = s(m1j-1,:,:)
       s(:,m2j,:) = s(:,2,:)
       s(:,1,:) = s(:,m2j-1,:)
       s(:,:,m3j) = s(:,:,2)
       s(:,:,1) = s(:,:,m3j-1)

      return
      end
```

Listing 5.10: The SAC Implementation of *rprj3*

```
#define P gen_weights( [ 1d/2d , 1d/4d , 1d/8d  , 1d/16d])

inline double[] gen_weights( double[] wp) {
  res = with( . <= iv <= . ) {
          off = with( 0*shape(iv) <= ix < shape(iv)) {
                  if( iv[ix] != 1)
                    dist = 1;
                  else
                    dist = 0;
                } fold( +, dist);
        } genarray( SHP, wp[[off]]);
  return( res);
}


inline double weighted_sum( double[] u, int[] x, double[] w) {
  res = with( 0*shape(w) <= dx < shape(w) )
          fold( +, u[x+dx-1] * w[dx]);
  return(res);
}


double[] setup_periodic_border( double[] u)
{
  for( x=1; x<shape(u)[0]-1; x++) {
    for( y=1; y<shape(u)[1]-1; y++) {
      z = shape(u)[2];
      u = modarray( u, [x,y,0], u[[x,y,z-2]]);
      u = modarray( u, [x,y,z-1], u[[x,y,1]]);
    }
  }

  for( x=1; x<shape(u)[0]-1; x++) {
    y = shape(u)[1];
    for( z=0; z<shape(u)[2]; z++) {
      u = modarray( u, [x,0,z], u[[x,y-2,z]]);
      u = modarray( u, [x,y-1,z], u[[x,1,z]]);
    }
  }

  x = shape(u)[0];
  for( y=0; y<shape(u)[0]; y++) {
    for( z=0; z<shape(u)[2]; z++) {
      u = modarray( u, [0,y,z], u[[x-2,y,z]]);
      u = modarray( u, [x-1,y,z], u[[1,y,z]]);
    }
  }

  return(u);
}


double[] fine2coarse( double[] r) {
  rn = with( 0*shape(r)+1 <= x<= shape(r) / 2 -1)
         genarray( shape(r) / 2 + 1, weighted_sum( r, 2*x, P));
  rn = setup_periodic_border(rn);
  return(rn);
}
```

Comparing these excerpts to the ZPL implementation of *rprj3* in Listing 5.4, one finds that the ZPL code is the clearest representation of the stencil. Its clarity can be attributed to its global-view, array-based nature as supported by regions. The lack of a hand-coded stencil optimization also aids in ZPL's clarity.

In contrast, the Fortran-based codes are obscured by looping structures, computations related to management of data distribution, hand-coded optimizations, and communication specification. The performance evaluation in the next section will indicate whether this additional coding effort pays off in terms of execution speed.

The core of the SAC code represents the *rprj3* stencil very clearly and concisely, but is complicated somewhat by the explicit support for periodic boundary conditions implemented in its `setup_periodic_border()` function. While it is possible to write the *rprj3* stencil more succinctly by wrapping references around the array as with ZPL's wrap-@ operator, this feature was not yet working at the time of this study.

Comparing the other stencils in a similar manner, one draws similar conclusions. The ZPL version tends to be more concise and clear than the other languages due to its use of regions to specify the computation.

5.5.4 Evaluation of Portability and Performance

Experimental Setup and Portability Results

To evaluate performance, the MG implementations were run on five hardware platforms: a high-performance Linux cluster, a Cray T3E, a Sun Enterprise 5500, an IBM SP, and an SGI Origin. The machines represent the diversity of parallel architectures in use today. Relevant details about each platform are summarized in Appendix B.

Table 5.3 summarizes the hardware/language combinations represented in this study, and gives an idea of each implementation's portability. Currently, no single platform supports all of the languages since CAF only runs on the Cray T3E and SAC only runs on the Sun Enterprise.

Table 5.3: Summary of Language/Hardware Combinations in the MG Study

	F77+MPI	CAF	HPF	SAC	ZPL
Linux cluster	•	×	•	×	•
IBM SP	•	×	•	×	•
Cray T3E	•	•	•	×	•
Sun Enterprise 5500	•	×	⊖	•	•
SGI Origin	•	×	•	×	•

× = language not currently supported on machine

• = language supported and included in our study

⊖ = language supported, but we could not obtain a compiler

Appendix C summarizes information about the compilers and command-line flags that were used in this study. Typically, the highest safe level of single-flag optimizations was used with each compiler. For the HPF implementation, the flags suggested by its implementors were used. Both the SAC and ZPL compilers use ANSI C as an intermediate language, so details for the C compilers used by their back ends are also provided.

Performance Results

The graphs in Figures 5.8–5.10 give speedup results for each machine. Timings were obtained by running each configuration (*language* × *machine* × *number of processors*) a handful of times over the course of several days and recording the best time for each. Appendix D contains the raw timings used to construct these graphs. Note that most of the machines did not have sufficient memory to run the production grade classes on small numbers of processors. Thus, speedup numbers for each graph are computed relative to the fastest implementation on the smallest number of processors. Classes B and C are shown for each platform except the Sun Enterprise which had insufficient memory to hold the class C problem size. The discussion of the performance results begins by focusing on the languages that are supported on multiple platforms.

HPF The most striking observation is that the HPF implementation performs poorly as compared to all other languages. This is due to the overheads associated with its 4D hierarchical array implementation, as described in Section 5.5.2. The amount of storage required by these arrays prevents many of the HPF configurations from running due to memory limitations. Notable exceptions are the Linux cluster, which has the largest amount of memory per node, and the SGI Origin, whose shared memory architecture allows small processor configurations to utilize memory from other processors on the machine that are not involved in the computation. The only class C HPF results are obtained on the SGI Origin using 8–16 processors. Running on fewer nodes causes each processor to exceed the system-specified limit for how much memory a single process can use. Running on more processors exhausts the global amount of memory available to the group of 128. Note that while the Linux cluster has a greater amount of memory per processor, its distributed memory architecture prevents it from running a class C problem using any number of processors.

When the HPF implementation does have sufficient memory to run, the 4D arrays impact overall performance due to their large memory footprints and the extra work required to correctly distribute and align them. Note that the HPF implementations *are* in fact scaling, just not at a rate that makes them competitive with the other implementations. It is difficult to judge whether these results are a symptom of the youth of HPF compilers or whether the language itself poses significant obstacles to efficient compilation of multigrid applications. The fact that so few HPF compilers support arrays of pointers to dynamic memory suggests that the latter might be true.

F77+MPI & ZPL In contrast, the F77+MPI and ZPL implementations perform quite well. As the processor set size increases, F77+MPI tends to outpace ZPL. This gap is

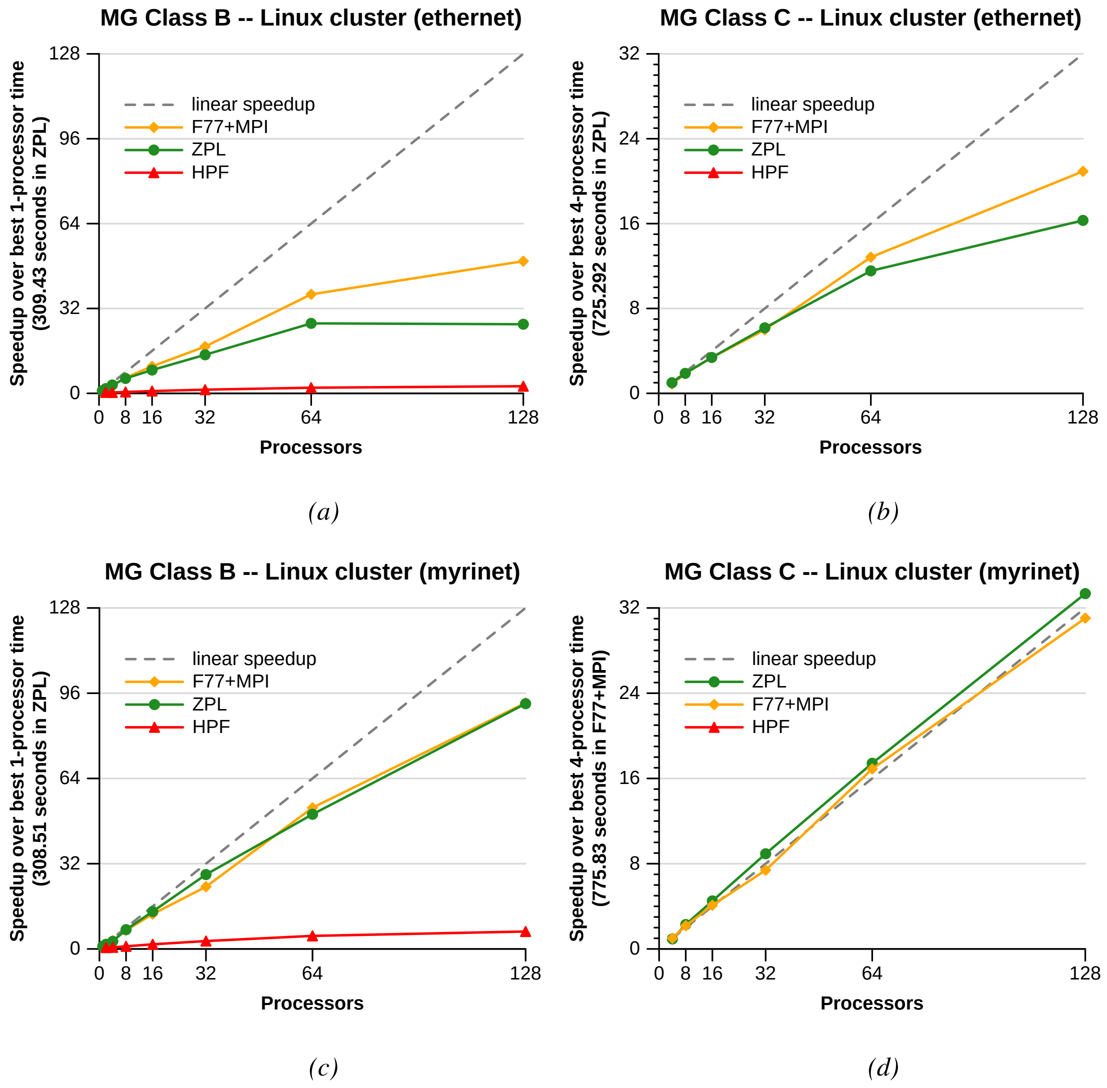
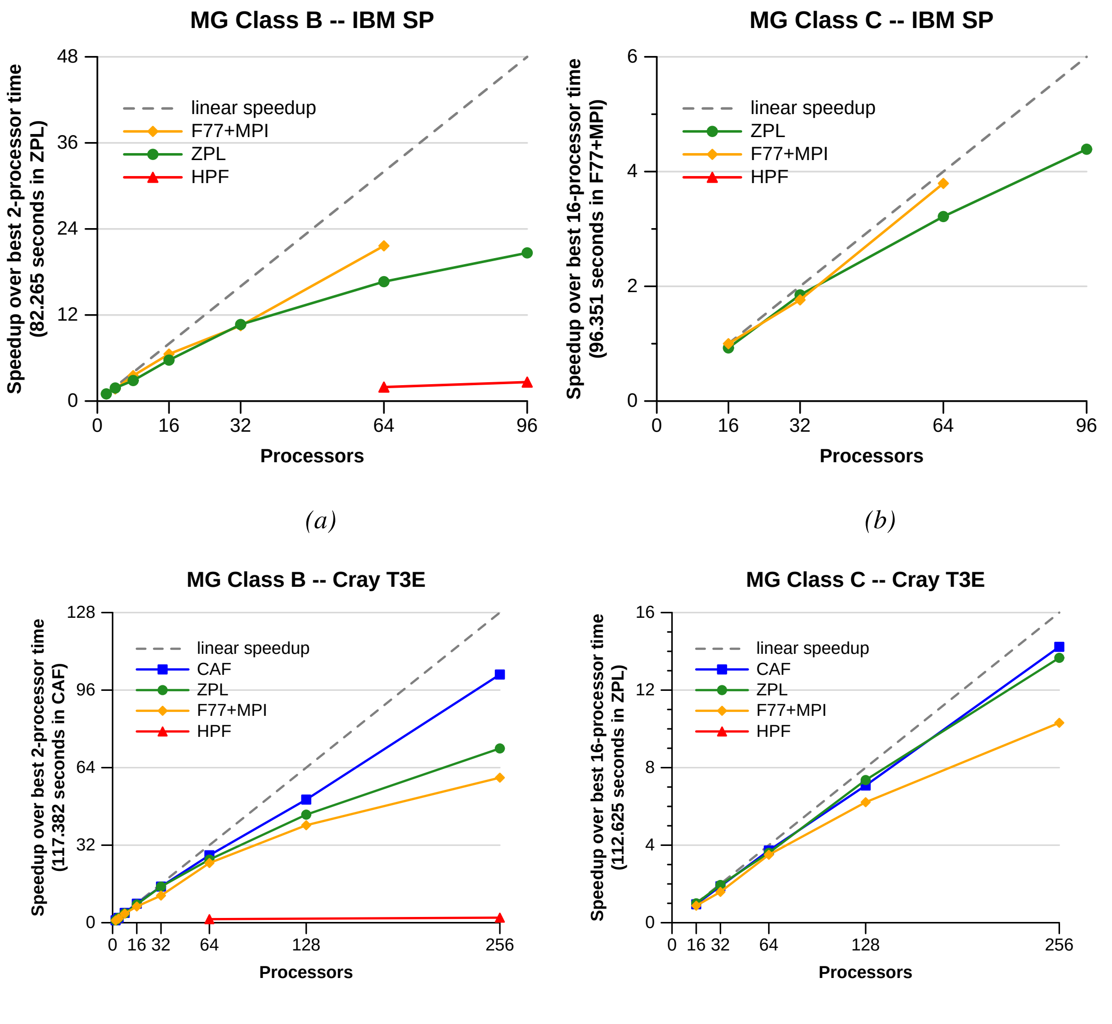


Figure 5.8: MG Performance on the Linux Cluster. These speedup graphs show MG classes B and C on the Linux cluster using both ethernet and myrinet. Note that there is not enough memory to run class C on small processor sets. Thus, its speedups are computed using the fastest execution time on the smallest number of processors (indicated in the *y*-axis label). Due to its excessive memory allocation, the HPF version is unable to run for class C problems.



(a) *(b)* *(c)* *(d)*

Figure 5.9: MG Performance on the IBM SP and Cray T3E. These speedup graphs show MG classes B and C on the IBM SP and Cray T3E. As in the previous figure, speedups for each graph are computed using the fastest execution time on the smallest number of processors. Once again, the memory requirements of the HPF version prevent it from being able to run on most configurations. Note that the F77+MPI version cannot run on 96 processors since it is written to work only with processor sets that are powers of two.

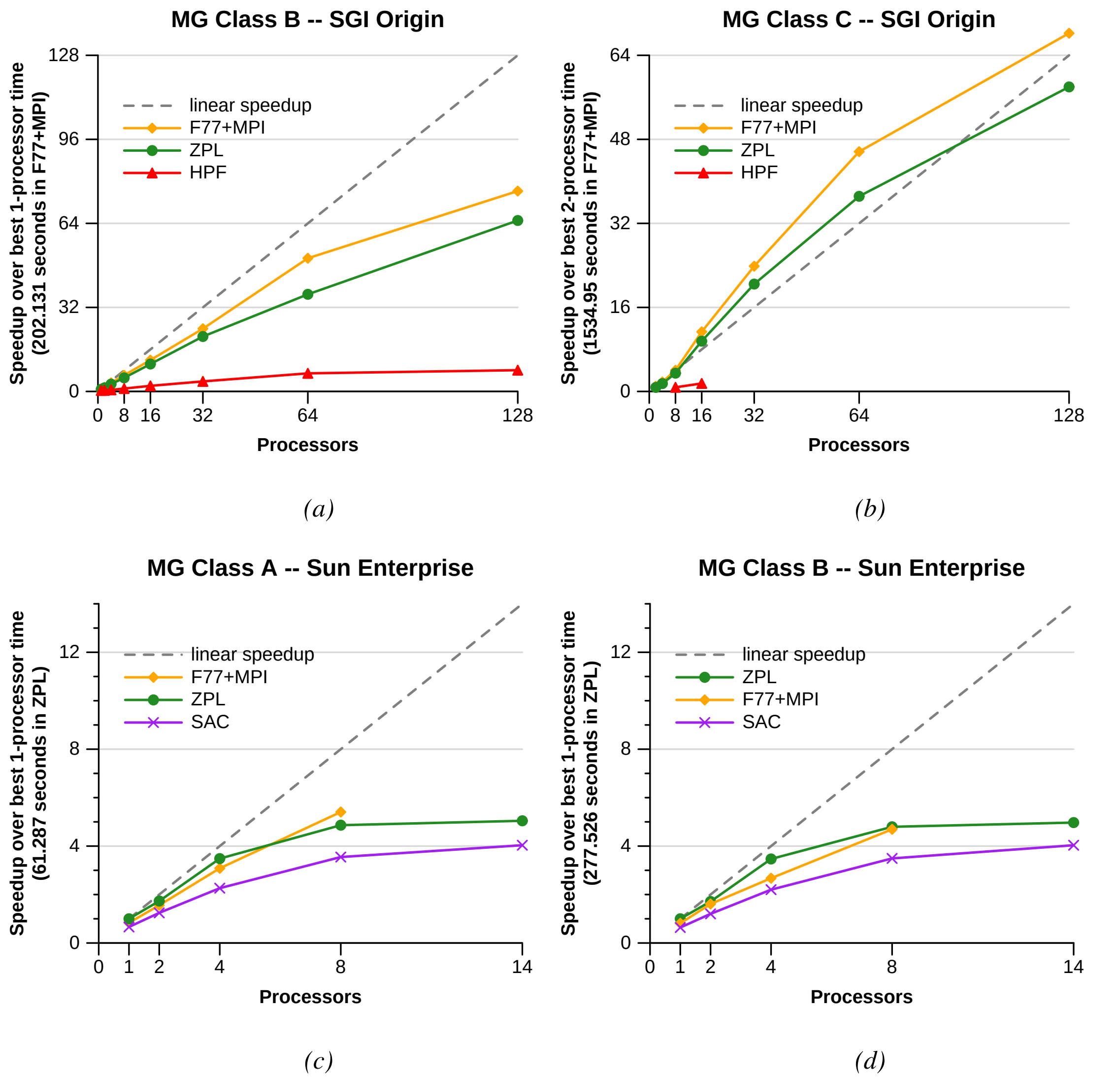


Figure 5.10: MG Performance on the SGI Origin and Sun Enterprise 5500. These speedup graphs show MG classes B and C on the SGI Origin and A and B on the Sun Enterprise 5500. Once again, speedups for each graph are computed using the fastest execution time on the smallest number of processors. The superlinear speedup on the Origin is due to the memory traffic required to run a class C problem on 2 processors. We were unable to obtain reasonable timings on more than 128 Origin processors due to the network traffic involved in crossing between machines. See Appendix D for details.

due to the fact that the F77+MPI implementation's local view allows the programmer to explicitly and precisely indicate where communication is required. Furthermore, the implementation's constraint that the problem size and processor set must be powers of two causes the calculation of a processor's involvement in communication to be simple and highly regular. In contrast, ZPL's data transfers are implemented using the Ironman library and are designed to work with any problem size or processor set. This results in a higher per-communication overhead. In addition, although the ZPL compiler performs several communication optimizations, hand inspection shows that it fails to insert the minimal amount of communication required by the benchmark, particularly for the *interp* stencil. The fact that the gap between F77+MPI and ZPL tends to be greater for class B problems than for class C is due to the fact that communication forms a smaller portion of the execution time for the larger problem size.

ZPL's support for arbitrary processor sets pays off on machines such as the IBM SP and Sun Enterprise whose processors are not a power of two. On these machines, the ZPL implementation can run using all of the processors while the F77+MPI implementation is limited to the next smallest power of two. When the amount of computation is great enough (as for class C on the IBM SP), this can result in ZPL performance that is unmatched by the F77+MPI version.

On smaller numbers of processors, the two implementations are fairly comparable. This testifies to the fact that the ZPL compiler's stencil optimizations are at least as good as the hand-coded optimizations in the F77+MPI implementation. In particular, the ZPL compiler implements the optimization by computing its subexpressions on an as-needed basis and storing the resulting sums using a small set of scalar temporaries. This results in better temporal and spatial locality as compared with the vector of values used in the NAS version [DCS01]. The ZPL compiler also unrolls the inner loop by the stencil's width to minimize shifts between these temporaries. These tweaks represent optimizations that a programmer may not be willing to do by hand, but which can have a measurable positive impact on performance.

Note that the Cray T3E results are exceptional due to the fact that the ZPL implementation significantly outperforms F77+MPI. This is a result of the fact that ZPL's Ironman communication is taking advantage of the SHMEM interface rather than MPI. It should be noted that while the HPF implementation was also compiled to the SHMEM interface, it failed to hoist synchronization out of the stencils' inner loops, negating its benefits.

CAF The CAF implementation also benefits from the Cray T3E's one-sided communication. The CAF compiler directly generates assembly instructions to perform the remote puts and gets required by the benchmark. This results in performance that is unmatched by any other implementation on large processor sets. For configurations that are not computation bound, ZPL falls away from CAF much as it did from F77+MPI on other platforms. Once again, this is due to the overhead of ZPL's flexible compiler-generated communication as compared to CAF's precise programmer-specified communication.

SAC Looking at the Sun Enterprise, it can be seen that SAC scales similarly to ZPL and F77+MPI, but lags slightly in performance. The reason for this is the lack of any stencil optimization, either hand-coded or by the compiler. Detection and optimization of stencils in SAC should be no more difficult than in ZPL, and would likely be a worthwhile addition to the SAC compiler. In combination with its implicit support for shared memory, this optimization should allow it to easily outperform both the F77+MPI implementation and the MPI-based ZPL implementation on shared memory platforms.

5.5.5 Summary

In summarizing this evaluation, note that the results in Table 5.3, Figure 5.7, and Figures 5.8–5.10 are naturally bimodal. In terms of clarity, local-view languages are far less clear and concise than global-view languages. Presently, F77+MPI, HPF, and ZPL are far more portable than CAF or SAC. And in terms of performance, all languages other than HPF currently obtain reasonable performance for MG. These observations are sum-

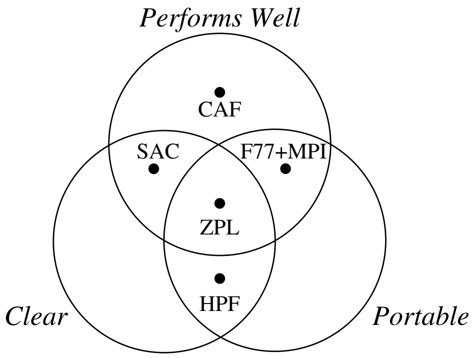


Figure 5.11: Summary of the MG Experiments. This Venn diagram summarizes the MG implementations used in this study. When comparing the implementations in terms of clarity, portability, and performance, they naturally fall into distinct groups. These groups are mapped to the Venn diagram directly.

marized in the Venn diagram of Figure 5.11. It should be emphasized that this diagram applies only to the implementations used in this study and not for the languages as a whole. Furthermore, as the compiler technology for each language matures, the placement of languages in this diagram will change. Nevertheless, we are pleased to note that ZPL is currently the only language in the study that does well for all three metrics.

5.6 Related Work

Each of the languages included in this chapter's study of MG have had published work describing how they express and implement hierarchical computation [FJY98, NRK98, Sch98a, BHS$^+$95, CDS99]. However, each of these publications studies one language in isolation, and typically only on a single platform. This chapter's study is unique due to its cross-language and cross-platform comparison.

Future work would do well to extend this chapter's study to compare language-based implementations of MG to an implementation that uses libraries like KeLP [FKB98].

5.7 Discussion

5.7.1 Analysis of ZPL's Multi-Concepts

Evaluating ZPL's multi-concepts from a language design perspective, they are seen to have a number of drawbacks, summarized as follows:

- They introduce a completely new concept, "parameterization", into the language. While parameterization might be useful as a general concept, the fact that it has not been required by other applications, or proven useful to them, makes it seem like a special case.

- Only a single parameter index is allowed for each multi-concept. In general, programmers may want to use multiple parameters to control different aspects of their regions, directions, and arrays. For example, a programmer might want to stride two of a region's dimensions according to one parameter and the third dimension by a second parameter.

- Syntactically, the parameterization of a concept is linked with its identifier rather than its type. This is an unfortunate consequence of the fact that regions and directions do not have explicit types in ZPL.

- While the concept of relative parameter indexing is useful, its current syntax is rather unique and awkward.

- Moreover, the curly-braces used by parameterization generally become cumbersome rather quickly.

In spite of these drawbacks, this chapter demonstrates that multi-concepts also provide a number of benefits to the programmer and compiler. These include:

- They establish a semantic relationship between a parameter value and an identifier's definition. This can embed meaning for the user and provide helpful information to the compiler.

- They support code re-use across levels of a hierarchy, as demonstrated by MG's stencil codes.

- They allow for a concise representation of multigrid computations, as compared to implementations in other global-view or local-view approaches.

- They support the efficient implementation of multigrid algorithms.

- The ability to use relative parameter indexing makes codes like MG easier for the programmer to write and for the compiler to analyze.

It would be ideal to replace ZPL's current multi-concepts with a similar mechanism that supports all of their benefits, yet eliminates their drawbacks. The following section proposes such a mechanism.

5.7.2 A Proposal for Improving Multi-Concepts

My proposed solution for fixing multi-concepts returns to the idea of making regions and directions into values in ZPL, as described in Section 2.18.3. It also requires two simple language concepts that would be natural extensions to ZPL's current syntax. Before presenting the proposed solution, these extensions are described briefly:

Index Constants for Indexed Arrays

The first requirement is that ZPL's indexed arrays should have a set of array constants equivalent to the `Index`*i* variables supported for parallel arrays. Such constants would support a richer set of computations to be written without using for-loops. For simple

Listing 5.11: Initialization of Indexed Arrays Using Proposed Indexing Scheme

```
var a: array [1..n, 1..n] of double;
    b: array [1..n] of array [1..n] of double;

-- Number the elements of a[] and b[][] in row-major order
a[] := (index1 - 1)*n - index2;
a[] := (index1[1] - 1)*n - index2[1];  -- equivalent to previous
b[][] := (index1[1] - 1)*n - index1[2];
```

array types, the identifiers could simply be the non-capitalized equivalents of the `Index`*i* constants: **`index1`**, **`index2`**, *etc.* However, nested arrays require some means of referring to the index's nesting depth in addition to its dimension. One way to support this would be to attach a dimension to the constant to indicate the desired depth. This dimension could be elided for arrays that are not nested. As an example, Listing 5.11 shows how two $n \times n$ indexed arrays could be initialized such that each element contains its unique position in row-major order.

Such compiler-supported constants would allow indexed array statements to express relative indexing without loops. For example, a 5-point stencil for indexed arrays could be written as follows:

```
a[] := (b[index1 + 1][index2] + b[index1][index2 + 1]
      + b[index1 - 1][index2] + b[index1][index2 - 1]);
```

Although the proposed identifiers may not be ideal, the concept is sound. Supporting such constants for indexed arrays are a natural extension to ZPL given its support for `Index`*i* constants and blank array indexing.

Promotion of Initializing Expressions

The second requirement is to support the initialization of array variables using a promoted expression. This is a natural extension to ZPL's current support for promotion within array assignments, but it is not currently supported by ZPL. As an example, the declarations in

Listing 5.12: Demonstration of Promoted Array Initializers

```
constant n: integer = 3;
         a: array [1..n, 1..n] of integer = {{2, 4, 6},
                                             {3, 5, 7},
                                             {5, 7, 9}};
         b: array [1..n, 1..n] of integer = index2*2 + index1-1;

         Mask: [R] integer = (Index1 + Index2)%2;
```

Listing 5.12 show a traditional constant array declaration and an equivalent specification that uses a promoted initializer. Note that the second declaration is not only cleaner, but it also does not require editing if the value of n is modified, as the first one would. The final declaration shows that promoted initializers could also be used to support parallel array constants, something that ZPL does not support due to the challenges involved in implementing traditional array initialization for distributed arrays.

Eliminating Multi-Concepts

My proposal for eliminating multi-concepts is to implement parameterization using indexed arrays of regions, directions, and arrays. This is possible only if directions and regions are made into true values in ZPL, as proposed in Section 2.18.3. Given that change, the definitions of multi-concepts can be expressed as constant indexed arrays with promoted initializers. By using index constants in their initializing expressions, parameterized relationships can be defined similar to those supported by ZPL's multi-concepts. References to a multi-concept would simply become traditional array references. As an example, Listing 5.13 shows excerpts from a revised implementation of MG using the proposed syntax.

This proposal eliminates most of the existing problems with multi-concepts: It uses traditional and well-understood concepts such as indexed arrays and promoted expressions to represent a parameterized concept. It supports an arbitrary number of parameters, since indexed arrays can be multi-dimensional. The use of an indexed array to represent the

Listing 5.13: MG Excerpts Using Proposed Syntax

```
constant dir100: array [0..num_levels] of direction =
                 [ 1, 0, 0] scaledby 2^index1;
         ...
         step:   array [0..num_levels] of direction =
                 [-1,-1,-1] scaledby 2^index1;

         Level: array [0..num_levels] of region =
                [1..nx, 1..ny, 1..nz] by step[index1];

var V: [Level[0]] double;
    U: array [0..num_levels] of [Level[index1]] double;
    R: array [0..num_levels] of [Level[index1]] double;


procedure rprj3(var S, R: [ , , ] double; lvl: integer);
begin
  S := 0.5000 * R
     + 0.2500 * (R@^dir100[lvl] + R@^dir010[lvl] + R@^dir001[lvl]
              + R@^dirN00[lvl] + R@^dir0N0[lvl] + R@^dir00N[lvl])
     + 0.1250 * (R@^dir110[lvl] + R@^dir1N0[lvl] + R@^dirN10[lvl]
              + R@^dirNN0[lvl] + R@^dir101[lvl] + R@^dir10N[lvl]
              + R@^dirN01[lvl] + R@^dirN0N[lvl] + R@^dir011[lvl]
              + R@^dir01N[lvl] + R@^dir0N1[lvl] + R@^dir0NN[lvl])
     + 0.0625 * (R@^dir111[lvl] + R@^dir11N[lvl] + R@^dir1N1[lvl]
              + R@^dir1NN[lvl] + R@^dirN11[lvl] + R@^dirN1N[lvl]
              + R@^dirNN1[lvl] + R@^dirNNN[lvl]);
end;

procedure iterate();
var lvl: integer;
begin
  -------------------------------------------------------
  -- Project to coarsest level
  -------------------------------------------------------
  for lvl := 1 to num_levels do
    [Level[lvl]] rprj3(R[lvl], R[lvl-1], lvl-1);
  end;

  ...
end;
```

parameter space links parameterization with the identifier's type rather than with its name. And, the proposed syntax improves the support for relative indexing via traditional array accesses and index constants. The primary remaining problem is that the syntax is still fairly cumbersome. The following section returns to this issue.

As desired, the proposed syntax also retains most of the benefits of ZPL's original multi-concepts: The use of constant initializers maintains the relationship between a parameter value and an object's definition. Code re-use is still supported. Expression of MG remains as concise and clear as before. Furthermore, the implementation should be almost identical to the current one, given that multi-concepts are implemented using indexed arrays. The main disadvantage is that inheriting a region or array's parameter index would no longer be supported without additional modifications. This is not a serious problem. For example, in the *rprj3* stencil, the entire `S` and `R` arrays could be passed in as parameters, allowing the user to index into `S` explicitly and use `index`*`i`* references for `R` and the directions.

5.7.3 Making MG Less Cumbersome

As noted in Section 5.3.1, ZPL's traditional elimination of tedious, error-prone indexing is somewhat lost in the 27-point stencils of the MG benchmark. Though indexing is no longer required, the 26 directions used by the stencil are difficult to name intelligently and thus their identifiers end up resembling indices. This causes their definitions and uses to be almost as tedious and error-prone as traditional array indexing.

Given the proposed extensions of the previous section, MG's direction declarations can be tidied up quite a bit. In particular, once an array of directions can be created, all of the directions required by MG's stencils can be expressed using a single direction declaration, shown in Listing 5.14. This declaration represents the entire hierarchy of directions by using one indexed array to describe the levels of the hierarchy and a second to define the actual directions themselves. Using this new scheme, a direction that was previously written `dirN10[i]` would now be written `dir[i][-1,1,0]` (or `dir[i][N,1,0]`, given the appropriate definition of `N`).

Listing 5.14: Declaring MG's Directions With a Single Identifier

```
constant dir: array [0..num_levels] of
              array [-1..1, -1..1, -1..1] of direction =
              [index1[2], index2[2], index3[2]]
              scaledby 2^index1[1];
```

Listing 5.15: The *rprj3* Stencil Using a Single Array of Directions

```
procedure rprj3(var S, R: [ , , ] double;
                var dir: array [ , , ] of direction);
begin
  S := 0.5000 * R
     + 0.2500 * (R@^dir[ 1, 0, 0] + R@^dir[ 0, 1, 0] + R@^dir[ 0, 0, 1]
               + R@^dir[-1, 0, 0] + R@^dir[ 0,-1, 0] + R@^dir[ 0, 0,-1])
     + 0.1250 * (R@^dir[ 1, 1, 0] + R@^dir[ 1,-1, 0] + R@^dir[-1, 1, 0]
               + R@^dir[-1,-1, 0] + R@^dir[ 1, 0, 1] + R@^dir[ 1, 0,-1]
               + R@^dir[-1, 0, 1] + R@^dir[-1, 0,-1] + R@^dir[ 0, 1, 1]
               + R@^dir[ 0, 1,-1] + R@^dir[ 0,-1, 1] + R@^dir[ 0,-1,-1])
     + 0.0625 * (R@^dir[ 1, 1, 1] + R@^dir[ 1, 1,-1] + R@^dir[ 1,-1, 1]
               + R@^dir[ 1,-1,-1] + R@^dir[-1, 1, 1] + R@^dir[-1, 1,-1]
               + R@^dir[-1,-1, 1] + R@^dir[-1,-1,-1]);
end;
...
[Level[lvl]] rprj3(R[lvl], R[lvl-1], dir[lvl-1]);
```

Listing 5.16: The *rprj3* Stencil Using a 3D Array of Weights

```
constant w_rprj3: array [-1..1, -1..1, -1..1] of double = 1.0 /
                  (1 + (index1!=0) + (index2!=0) + (index3!=0));

procedure rprj3(var S, R: [ , , ] double;
                var dir: array [ , , ] of direction);
begin
  S := 0;
  S += w_rprj3[] * R@^dir[];
end;
...
[Level[lvl]] rprj3(R[lvl], R[lvl-1], dir[lvl-1]);
```

In addition to reducing the tedious direction declarations, this approach also allows the set of directions for a single level of the hierarchy to be passed into a stencil routine as a parameter. For example, Listing 5.15 shows the *rprj3* stencil written to take a set of directions as a parameter.

The net result of these changes seems to be the replacement of traditional array indexing with indexing into an array of directions. However, a redeclaration of the stencil's weights combined with a clever use of blank array references can reduce this code to an even simpler form, as shown in Listing 5.16. This is arguably the most reasonable expression of a stencil computation since it allows the programmer to specify the weights and directions using arrays that match the stencil's logical 3 dimensions. Moreover, it allows weights to be applied to their corresponding @-references using a concise syntax that scales with the size of the stencil.

The main drawback to this method of expressing stencils is that a naive implementation would perform too many multiplications by not factoring common weights out of the expression as in the previous explicit stencil expressions. However, the current implementation of the stencil optimization would actually perform this factoring without any trouble. The optimization starts by constructing a *tablet* data structure that summarizes the weights applied to each array element, explicitly replicating any weights that the user had factored into a single multiplication [DCS01]. In this compact stencil expression, the array of weights, `w_rprj3[]`, *is* the tablet that the stencil optimization would compute. Thus, it should be expected that the stencil optimization would be able to implement a compact stencil expression identically to the previous ones.

5.7.4 Supporting Other Hierarchical Algorithms

Although providing reasonable support for the MG benchmark may indicate ZPL's ability to support other multigrid algorithms cleanly and efficiently, many other classes of hierarchical algorithms exist that differ significantly enough to present additional challenges. Some of these benchmarks are described here.

Adaptive mesh refinement algorithms (AMR) [LM90] are similar to the multigrid algorithms described here, except that the initial grid is the coarsest in the hierarchy, rather than the finest. AMR algorithms determine which areas of a problem space require additional resolution, and then compute on those regions using finer grids. To an extent, this is simply the multigrid hierarchy turned upside-down, except that only portions of the finer levels are referenced. This forms one of the motivations for supporting collections of dynamic regions as originally discussed in Section 2.18.2. These dynamic regions could be established to describe the indices of interest, strided by twice as much in each dimension. Without such support, AMR algorithms cannot be written in ZPL in a manner that is clean and uses a minimal amount of memory.

Another hierarchical array algorithm with similarities to AMR is the Barnes-Hutt algorithm for n-body simulations [BH86]. It uses an *octree* structure to store approximate representations of bodies that are distant in space. Rendering algorithms often use similar octree-based techniques to recursively subdivide scenes for rendering [CDL$^+$96, FvDFH92]. These algorithms share the AMR property of starting at the coarsest grid and moving to finer levels of the hierarchy as needed. Parallel Barnes-Hutt algorithms are typically written in a task-parallel manner so that different branches of the octree are traversed by different processors. Since task parallelism is one of the features that is in the works for A-ZPL, this represents another class of hierarchical algorithms that currently cannot be expressed cleanly in ZPL.

Other hierarchical algorithms are graph-based, collapsing groups of vertices and their interconnecting edges to a single vertex [KK98, KK96]. Although graphs can be represented in ZPL using 1D arrays of nodes and edges, such graph-based algorithms would be better served by A-ZPL's intended support for irregular data structures, to better emphasize locality and the graph's structure.

A final hierarchical algorithm is the fast multipole method (FMM) [ED95]. Like the multigrid method, it uses coarse approximations of a fine grid to accelerate calculations in a discrete space. However, unlike the multigrid techniques described in this chapter, each level of the hierarchy is not a complete cube of grid points, but rather an arbitrary subset of indices. While this could be represented by overlaying a set of masks on the hierarchical regions of this chapter, the space and time overheads would be prohibitive for most FMM calculations. Thus, the programmer really needs a means of specifying a hierarchy of sparse arrays. This possibility is addressed in the following chapter.

5.8 Summary

When considering the complete space of hierarchical algorithms, it can be seen that ZPL's support for hierarchical programming is still rather primitive. Without support for sparse arrays, task parallelism, and the naming of dynamic regions, many hierarchical algorithms simply cannot be expressed cleanly or efficiently in ZPL. Fortunately, many of these features will be added in A-ZPL, at which point these algorithms can be reconsidered.

In spite of these shortcomings, this chapter's use of parameterized regions to express hierarchical index sets supports a fundamental concept required by all array-based hierarchical algorithms. These hierarchical regions will serve as a solid foundation for more advanced algorithms as the other missing features are filled in. Furthermore, multigrid methods constitute an important class of algorithms in their own right, and should not be dismissed due to their regularity. This chapter indicates that ZPL's use of regions allows these algorithms to be expressed with an unrivaled combination of clarity, portability, and performance.

Chapter 6

SPARSE REGIONS

As pointed out in Section 2.9, one of the drawbacks of regions is that they must be rectangular and regular. Although masks allow users to select arbitrary indices from a region, they also require space and time proportional to the region's indices rather than the number of "true" values in the mask. For some algorithms, such as red-black SOR, this may be completely reasonable. However, for *sparse* algorithms in which the number of true values is asymptotically less than the region's indices, the excessive memory and execution time required by masks makes them inappropriate.

This chapter provides a solution to this problem in the form of *sparse regions*. Sparse regions are similar to regular regions, except that they can describe an arbitrary set of indices. Sparse regions are defined in A-ZPL as part of its goal to support more general forms of parallel computation.

This chapter is organized as follows: The next section gives some motivation for providing high-level support for parallel sparse computation. Section 6.2 introduces sparse regions and describes their benefits in terms of clarity and code reuse. The benchmarks used to evaluate sparse regions are introduced in Section 6.3. Sections 6.4–6.6 detail the implementation of sparse regions and arrays in A-ZPL, concentrating on the unique runtime representation of sparse index sets. Section 6.7 evaluates the sparse benchmarks in terms of clarity, memory usage, and performance. Other efforts for supporting high-level sparse computation are described in Section 6.8. Finally, section 6.9 discusses future directions for using sparse regions. The contents of this chapter are an expanded presentation of work that was published previously [CS01, CLS98].

0	0	0	0	0	0
0	3	0	4	0	9
0	0	0	0	0	0
0	0	5	0	2	0
0	1	0	0	0	0
0	0	0	7	0	0

Figure 6.1: A Sample Sparse Array. In this array, $d = 2$, $n = 6$, $nnz = 7$, and the IRV is 0.

6.1 Motivation

The *sparse array* is a fundamental data structure in scientific computing due to the pervasiveness of sparsity in the universe and the models that scientists use to represent aspects of it. Sparse arrays are commonly used to represent *sparse matrices*, which have innumerable applications in mathematics, engineering, and the sciences. Sparse arrays can also represent *structural sparsity*, such as the placement of particles in space or of coastlines on the earth's surface. In addition, *sparse iteration* can be used with traditional dense arrays in order to focus on particular values of interest.

This discussion considers a sparse array to be an array of arbitrary dimension d, in which a single value appears sufficiently frequently that it benefits the user to store just a single copy of it (along with any values that differ). This *implicitly replicated value* (IRV) is often 0, but can take on any value in practice. Typically, sparse arrays are most useful when the number of explicitly stored values, nnz, is asymptotically less than the number of values represented by the array (*i.e.*, $nnz = o(n^d)$, where n is the number of elements per dimension[1]). Figure 6.1 illustrates a sample sparse array and gives its defining parameters.

This chapter makes a sharp distinction between sparse matrices and sparse arrays. A sparse array should be considered a general data structure of arbitrary dimension that stores

[1]In practice each dimension can have a different number of elements, n_i, but for simplicity this discussion assumes that they are all equal.

Listing 6.1: Dense and Sparse Matrix-Vector Multiplications in Fortran

```
!             (a)                                  (b)
 do j = 1,numrows                     do j = 1,numrows
   sum = 0.d0                           sum = 0.d0
   do k = 1,numcols                     do k = rowstr(j),rowstr(j+1)-1
     sum = sum + a(j,k)*p(k)              sum = sum + a(k)*p(colidx(k))
   enddo                                enddo
   w(j) = sum                           w(j) = sum
 enddo                                enddo
```

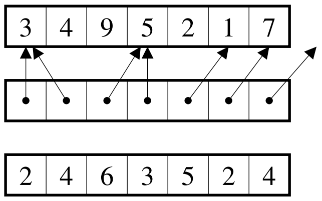
a sparse set of values within a dense index space. Sparse arrays support structural operations such as iteration and array accesses. In contrast, sparse matrices have a specific mathematical interpretation, are usually two-dimensional, and support high-level operations such as matrix multiplication, factoring, and the solving of linear equations. Just as ZPL's arrays can be used to implement dense matrices, so too should sparse arrays be useful for representing sparse matrices.

6.1.1 Challenges to Parallel Sparse Computation

Two of the primary challenges in providing reasonable facilities for sparse computation are clarity and performance. Performing sparse computation in parallel presents additional obstacles. The following paragraphs consider these challenges in turn.

Clarity

Conceptually, sparse arrays are no different than normal arrays; they represent a dense set of values, but simply use a nontraditional implementation to do so. Most programming languages have no built-in support for sparse array representations, forcing users to implement their own sparse data structures by hand. The presence of these data structures causes familiar operations to be represented in a complex manner, making the programmer's intent less clear.



data structure	*space*
dense data vector	*O(nnz)*
row starting points	*O(n)*
column indices	*O(nnz)*

Figure 6.2: The CSR Sparse Array Format. This figure shows the sparse array of Figure 6.1 stored in CSR format. The array's interesting values are stored in a dense vector format. A second vector of integer pointers indicates the starting position of each row of the array. The third vector stores the column index of each value.

A prime example of this confusion can be found in the NAS CG benchmark which implements the conjugate gradient method [BHS$^+$95]. The main computation in CG is a series of sparse matrix-vector products. Conceptually, matrix-vector multiplication is a simple operation that can be written in Fortran using dense arrays as shown in Listing 6.1a. NAS CG represents the sparse array using a *compressed sparse row* (CSR) storage format [Saa90]. Figure 6.2 shows the sample array from Figure 6.1 as it would be implemented using CSR. The dense Fortran matrix-vector multiplication code must change considerably to work with CSR storage, as shown in Listing 6.1b. The use of the sparse representation obfuscates the logical operation being performed to the point that it is difficult to recognize as matrix-vector multiplication for anyone not familiar with the CSR format (and even for those who are).

Over time, dozens of different sparse representations have been developed [AMPS00], most of which are designed to capitalize on the structural properties of a class of sparse arrays. This proliferation of formats further hurts the clarity of sparse array codes, since each format requires unique loop structures and array access idioms. Ideally, programmers should be able to write sparse computations using traditional syntax that reflects the dense computation being expressed. This reinforces the orthogonality of the dense computation and its sparse implementation.

Performance

Performance represents a second challenge to effective sparse computation. The use of the CSR format in the example above disguises the code's intent not only for human readers, but also for the compiler. In particular, the code uses *index arrays*—arrays that act as indices for other arrays. Index arrays obscure data dependences and are a notorious obstacle for compiler analysis and automatic parallelization [SLY89, BBCR98]. Although many compiler techniques have been developed to combat this problem, a breakdown in communication has occurred: the language has failed to communicate the programmer's intent to the compiler as clearly as it could. For example, CSR guarantees certain useful properties about the `rowstr` and `colidx` arrays used in the matrix-vector code above, but the compiler has no way of detecting these properties when analyzing the code.

Parallel Implementation Details

Performing sparse computations in parallel presents a number of additional challenges. The distribution of computation and data across the processor set may not prove terribly difficult (the Fortran example above only requires changing the upper bound of the outer loop to `lastrow-firstrow+1`), but the code required to implement data transfer and synchronization between processors tends to be lengthy, tedious, and complex. Even simple dense parallel array operations such as boundary value updates become significantly more complicated in the sparse context.

Flexibility

Writing portable parallel sparse codes that perform well *can* be done—the NAS CG benchmark is one such example. However, this is achieved only through great programmer care, and the result is often an extremely brittle piece of code. For example, if NAS CG was extended in such a way that it required iteration over the sparse matrix columns as well as rows, a vast amount of code would have to be reformulated and rewritten.

Summary

All of these issues—clarity, performance, parallel implementation, and flexibility—call out for a language-based solution. In particular, they motivate a language that provides a high-level, global means of specifying sparse computation using syntax which resembles the equivalent dense computation. This syntax should be clear to the programmer as well as the compiler and should support an efficient parallel implementation.

Since regions serve to separate an array's indices from its values, they represent an ideal means for meeting these goals. The key is simply to support "sparse" regions by relaxing the constraint that a region's index set must be regular and rectangular.

6.2 Sparse Regions

The goal of this work is to extend regions to efficiently support general sparse index sets without sacrificing the benefits of traditional regions. In particular, the syntax should remain crisp, the parallel overheads should remain apparent to the user, and the region's time and space overheads should be proportional to the number of indices it represents. The rest of this section describes the syntax used to declare and use sparse regions.

6.2.1 Sparse Region Declarations

A-ZPL programmers declare sparse regions by modifying a traditional region specification with a boolean expression that defines the region's *sparsity pattern*. For example:

```
region Tri = [1..n,1..n] where (abs(Index1-Index2) < 2);
```

This declaration bounds `Tri` to an $n \times n$ index set, but restricts it to those locations whose row and column indices differ by no more than 1. Thus, this declaration creates a tridiagonal region.

Sparse region declarations appear similar to the masking of regions, but have entirely different semantics. In particular, the declaration of `Tri` above creates a brand new region

Listing 6.2: Various Sparse Region Declarations

```
region Diag = Tri where (Index1 = Index2);    -- limit Tri to its main diagonal
       Rs   =  R  where Pattern;              -- Pattern is a boolean array
       Rs2  =  R  where foo(Index1,Index2);   -- foo(i,j) returns true/false
       Rs3  =  R  where read_pat(infile);     -- read pattern from a file
       Rs4  =  R  where ?;                    -- dynamic sparsity pattern that
                                              -- will be specified in code body
```

that only represents those indices where the sparsity pattern is true. Storing or iterating over `Tri`'s indices requires space and time proportional to the number of "true" values in the sparsity pattern, rather than its bounding box.

Sparsity patterns need not be statically defined as in this first example. Moreover, the base region itself may be sparse. Listing 6.2 shows a variety of different sparse region declarations. Each declaration is evaluated at runtime by iterating over its base region and using a sparse data structure to store those indices where its pattern evaluates to "true." Of particular interest is the final declaration which uses a question mark to indicate that the sparsity pattern cannot be specified statically. Such regions must have their sparsity pattern set within the program's body before being used. This topic is discussed in more detail in Section 6.9.

A-ZPL programmers declare sparse arrays in the traditional manner, using sparse regions to define their size:

```
var As, Bs: [Rs] integer;
```

This declaration causes each processor to allocate a dense vector of memory for `As` and `Bs` whose size is equal to the number of local indices in `Rs` (plus one additional value to represent the IRV). Since sparsity patterns are associated with regions rather than arrays, the space and time required to operate on sparse arrays can be amortized across groups of arrays that share the same sparsity pattern.

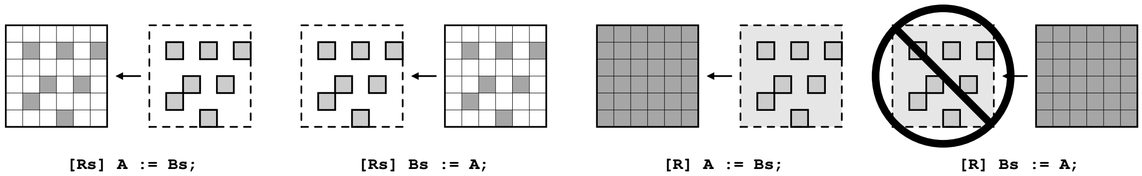


Figure 6.3: Sample Sparse/Dense Assignments. Assume that R and A are dense, and that Rs and Bs are sparse. Note that sparse reads and writes of both arrays are legal, referring only to the elements in the sparsity pattern. Dense reads of sparse arrays are also legal—the IRV is used for values that are not in the sparsity pattern. However, a dense write to a sparse array is not legal due to the fact that a unique memory location is not allocated for each index. These examples extend in the obvious way to statements which refer to multiple sparsity patterns.

6.2.2 Using Sparse Regions

Sparse regions concisely support a rich variety of semantics. As a simple example, consider the assignments in Figure 6.3. In the first two assignments, the controlling region is sparse. The effect is that references to dense array A only read or write those values indicated by Rs. When sparse array Bs is referenced, all of its values are accessed since its sparsity pattern is defined by Rs.

The third assignment illustrates a dense read of a sparse array. In this statement, all values of A will be written in spite of the fact that Bs is sparse. This is because even though Bs is sparse, it logically represents a full $n \times n$ array of values. Thus, when reading an index in $\texttt{R} - \texttt{Rs}$, the IRV of Bs will be referenced, resulting in a virtual $n \times n$ array. Note that this statement will take $\Theta(n^2)$ time as compared to the first statement which, though similar, requires only $\Theta(|\texttt{Rs}|)$ time.

The last assignment is illegal because it attempts to store values in As for which no storage has been allocated.

Although these assignments are quite simple, they illustrate the basic rules of reading and writing sparse and dense arrays in the context of a sparse or dense region. The same principles naturally extend to statements that refer to a variety of sparsity patterns. Furthermore, the traditional ZPL array operators extend to sparse regions and arrays in the natural manner. The next section gives further examples of sparse programming in A-ZPL by demonstrating the use of sparse regions in a number of benchmarks.

6.3 Sparse Benchmarks

This section describes the implementation of a number of sparse benchmarks in A-ZPL designed to evaluate sparse regions.

6.3.1 Sparse Matrix-Vector Multiplication and NAS CG

Returning to the motivating sparse matrix-vector multiplication example from Section 6.1, compare the dense ZPL representation of matrix-vector multiplication from Chapter 2 (Listing 6.3a) with its sparse equivalent (Listing 6.3b). The only change required is to specify a sparsity pattern for region `R` (elided here for brevity and generality). Other than this change, the code remains exactly the same. Though not shown here, the same would hold true for the 1D ZPL implementation of matrix-vector multiplication. These codes demonstrate that regions achieve the goal of allowing sparse computations to resemble their dense equivalents. In general, a sparse A-ZPL code will appear identical to its dense representation plus any additional code needed to specify its regions' sparsity patterns.

As mentioned in Section 6.1, the core of the NAS CG benchmark is a series of sparse matrix-vector multiplications. The rest of CG consists of simple vector-vector operations and reductions over vectors. The CG benchmark has a straightforward implementation in A-ZPL, excerpts of which are shown in Listing 6.4. A single sparse region, `RS`, is required to describe the structure of the sparse matrix. Its sparsity pattern is specified dynamically in the program text, either by reading it from disk or by computing it on the fly.

The bulk of CG's computation takes place in the `conj_grad()` routine which contains two sparse matrix-vector multiplications (one of which is executed within a loop).

Listing 6.3: Dense and Sparse Matrix-Vector Multiplications in ZPL

```
--            (a)                                 (b)
program matvectmult;              program sparse_matvectmult;

config var m: integer = 100;      config var m: integer = 100;
           n: integer = 100;                 n: integer = 100;

region R = [1..m, 1..n];          region R = [1..m, 1..n] where ...;
       TopRow =  [1, 1..n];              TopRow = [1, 1..n];
       RowVect = [*, 1..n];              RowVect = [*, 1..n];
       ColVect = [1..m, n];              ColVect = [1..m, n];

var M: [R] double;                var M: [R] double;
    I: [TopRow]  double;              I: [TopRow] double;
    V: [RowVect] double;              V: [RowVect] double;
    S: [ColVect] double;              S: [ColVect] double;

procedure matvectmult();          procedure sparse_matvectmult();
begin                             begin
 [RowVect] V := >>[1, ] I;         [RowVect] V := >>[1, ] I;
 [ColVect] S := +<<[R] (M * V);    [ColVect] S := +<<[R] (M * V);
end;                              end;
```

Listing 6.4: Excerpts from the CG benchmark written in A-ZPL

```
region Row = [  *   , 1..na];          -- row indices
       Col = [1..na,  *   ];           -- col indices
       RS  = [1..na, 1..na] where ?;   -- sparse pattern

var X, Z, P, Q, R: [Row] double;       -- row vectors
    W: [Col] double;                   -- col vector
    A: [RS] double;                    -- sparse array

procedure conj_grad(): double;
var
   rho, rho0, alpha, beta, d, rnorm: double;
   cgit: integer;
[Row] begin
  Z := 0;
  R := X;
  P := R;

  rho := +<< (R*R);

  for cgit := 1 to cgitmax do
    [Col] W := +<<[RS] (A*P);          -- mat-vect mult.
    Q := W#[Index2,Index2];            -- transpose result

    d := +<< (P*Q);                    -- adjust Z and R
    alpha := rho / d;
    rho0 := rho;
    Z += alpha*P;
    R -= alpha*Q;

    rho := +<< (R*R);                  -- adjust P
    beta := rho / rho0;
    P := R + beta * P;
  end;

  [Col] W := +<<[RS] (A*Z);            -- mat-vect mult.
  R := W#[Index2,Index2];              -- transpose result

  d := +<< ((X-R)^2);                  -- calculate the norm
  rnorm := sqrt(d);
  return rnorm;
end;
```

The remap operator is used after each multiplication to transpose the resulting column vector back into a row. The row vectors are then scaled accordingly and the process repeats. Section 6.7 compares this implementation against the hand-coded NAS version.

6.3.2 Tridiagonal Matrix Multiplication

The tridiagonal matrix multiplication algorithms of Section 2.15.4 demonstrated a tradeoff between implementations that require $\Theta(n^2)$ time/space to represent the matrix and those that used a more concise $\Theta(n)$ format, yet failed to reflect the logical 2D index space being used. Sparse regions provide a compromise between these two solutions by supporting the conceptual representation of a tridiagonal in $n \times n$ space using only $\Theta(n)$ space and time.

Once again, the sparse implementation is identical to the dense case, except for the declarations of the regions. For example, the shattered control-flow implementation of Listing 2.22 could be made into a sparse implementation by modifying its region declarations as follows:

```
region R    = [1..n, 1..n]     where (abs(Index1-Index2)<3);
       BigR = [0..n+1,0..n+1] where (abs(Index1-Index2)<2);
```

The first declaration is a pentadiagonal region, while the second is a tridiagonal region with additional boundary indices. The computation itself remains unchanged. Wise programmers would probably choose to change the region names to something more descriptive like `Pent` and `Tri`.

A second implementation possibility would be to represent all three arrays using a single sparse pentadiagonal region. While this would result in extraneous data elements being allocated for the two tridiagonal arrays, it also would eliminate the tridiagonal sparsity pattern, allowing a single pattern and its overheads to be shared by all three arrays.

Listing 6.5: Declarations for a Sparse Implementation of MG

```
region RBase = [1..nx, 1..ny, 1..nz];
       RS = RBase where ?;
       Level{} = RBase by step{};

var V: [RS] double;
    U{}: [Level{}] double;
    R{}: [Level{}] double;
```

6.3.3 NAS MG

By most standards, the NAS MG benchmark is considered a dense computation, since its hierarchical arrays must represent a full set of indices at each level. However, the input to the NAS MG benchmark is extremely sparse. It consists of ten positive and ten negative charges stored at the finest discretization of the problem space. For the class C version of the benchmark, this implies that only 20 of the 512^3 elements in the input array are nonzero—15 millionths of a percent!

If MG's input array was used only at the outset of the program, the drawbacks to using a dense representation would be minimal since its storage could be reclaimed to avoid wasting memory. However, recall that MG computes two residual stencils against the input array every iteration (Listing 5.3). This requires walking across the array's entire memory footprint to find the 20 values that are of interest.

For this reason, it seems worthwhile to use a sparse array to store MG's input. As in the previous benchmarks, the change is minimal, simply requiring new declarations for the sparse region and input array. This application demonstrates the use of a sparse array to represent structural sparsity rather than a sparse matrix.

Listing 6.5 contains the modified declarations to convert ZPL MG's input from sparse to dense. As in CG, the sparsity pattern for `RS` is specified dynamically, either by reading it from a file or by computing it explicitly. This region is then used to declare the input array `V`.

Listing 6.6: A Sparse Implementation of the *resid* Stencil

```
procedure resid(var R, U: [ , , ] double; var V: [RS] double);
begin
  R := - a[0] *  U
       - a[2]*(U@^dir110{} + U@^dir1N0{} + U@^dirN10{} + U@^dirNN0{} +
               U@^dir101{} + U@^dir10N{} + U@^dirN01{} + U@^dirN0N{} +
               U@^dir011{} + U@^dir01N{} + U@^dir0N1{} + U@^dir0NN{})
       - a[3]*(U@^dir111{} + U@^dir11N{} + U@^dir1N1{} + U@^dir1NN{} +
               U@^dirN11{} + U@^dirN1N{} + U@^dirNN1{} + U@^dirNNN{});
  [RS] R += V;
end;
```

The implementation of the *resid* stencil can be left as-is since it is legal to read the sparse input array within the context of a dense region. To implement it correctly, the A-ZPL compiler must automatically create two versions of the stencil—the traditional dense version for use within the hierarchy and a second sparse version for use with the input array.

A savvy programmer would realize that this trivial reuse of *resid* could be improved. In particular, its reference to `V` within the context of the hierarchy's finest region will result in a number of useless additions of zero, as well as pointless references to `V`'s IRV. To eliminate these overheads, the programmer can explicitly rewrite the stencil for the sparse case as shown in Listing 6.6. In this version, the 27-point stencil is computed using the traditional dense region, and then the 20 sparse elements are accumulated using the sparse region `RS`.

Although this modification requires the programmer to create two copies of `resid()`, the change is quite trivial and ought to pay off in terms of performance. Future work should consider the possibility of implementing such optimizations automatically in the compiler.

6.4 Implementation of Sparse Regions and Arrays

This section describes the implementation of sparse regions and arrays in A-ZPL. It begins by giving a broad overview of the implementation and then focuses on the runtime representation of sparsity patterns.

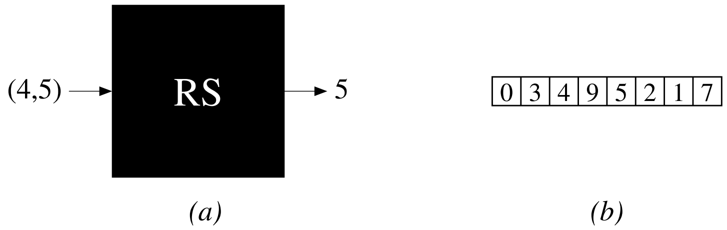


Figure 6.4: A-ZPL's Sparse Array and Region Scheme. This diagram shows the sparse array of Figure 6.1 as it would be implemented in A-ZPL. (a) A black box representing the sparse region implementation. It must be able to translate logical indices into physical ones as shown here. In addition, it must support arbitrary iteration and array slicing to implement ZPL's array and region operations. (b) The dense vector of values for the array. The initial element is the IRV; all other elements correspond to a single index in the sparse region.

6.4.1 Overview of Sparse Regions and Arrays

As mentioned in Section 6.2.1, sparse regions support the idea of associating sparsity patterns with regions rather than arrays. This gives the compiler the ability to amortize overheads related to storing sparse index sets and iterating over them when multiple arrays share the same sparsity pattern.

Sparse regions are implemented using a traditional ZPL region descriptor with additional fields tacked on to describe the set of sparse indices. The traditional descriptor fields are used to describe the region's bounding indices as always. The sparsity pattern itself is represented with a unique data structure described in the next section. Each index stored by the data structure has a unique ID that is used to access its value in all sparse arrays declared using the region.

Sparse arrays are implemented using a traditional ZPL array descriptor to represent the dense vector of defining values. The length of the vector is equal to the number of sparse indices owned by the processor, plus one to store the IRV. The IRV is stored at position 0 in the array. All other elements are accessed using the unique IDs stored in the region descriptor as their indices (Figure 6.4).

6.4.2 The Sparse Representation

In order to understand A-ZPL's choice of sparse representations, it is useful to review the functionality that its regions and array operators require. The following list summarizes the most time-crucial operations and the situations that require them:

region operations:

- row-major iteration (the common case for most array statements)
- iteration in arbitrary directions, dimensions (certain uses of the `@`, `@^` operators)
- random access to a slice of indices (singleton dimensions, dynamic regions)

array operations:

- ordered array access (general array references)
- random array access (the remap operator)

The generality of these requirements poses an obstacle to using most of the standard sparse representations, since they are typically optimized for specific access patterns. For example, the need for efficient iteration in arbitrary dimensions eliminates the possibility of using formats that only support particular iteration orders such as CSR. A-ZPL's representation must be extremely general-purpose in order to support the combination of arbitrary iteration and fast random access required by the language. The A-ZPL strategy is therefore to design a format that is general, but which can be automatically optimized to efficiently meet a program's requirements.

In A-ZPL's sparse representation, every index is logically represented by a node that stores: (1) the unique ID used to access its values in sparse arrays, (2) the logical index that it represents, and (3) pointers to the next and previous nodes in each dimension (Figure 6.5a). This *lattice* of nodes is a generalization of a multilist structure [Wei99], supporting the ability to move quickly from any node to its neighbors in any dimension. The space required by the lattice is $\Theta(nnz)$.

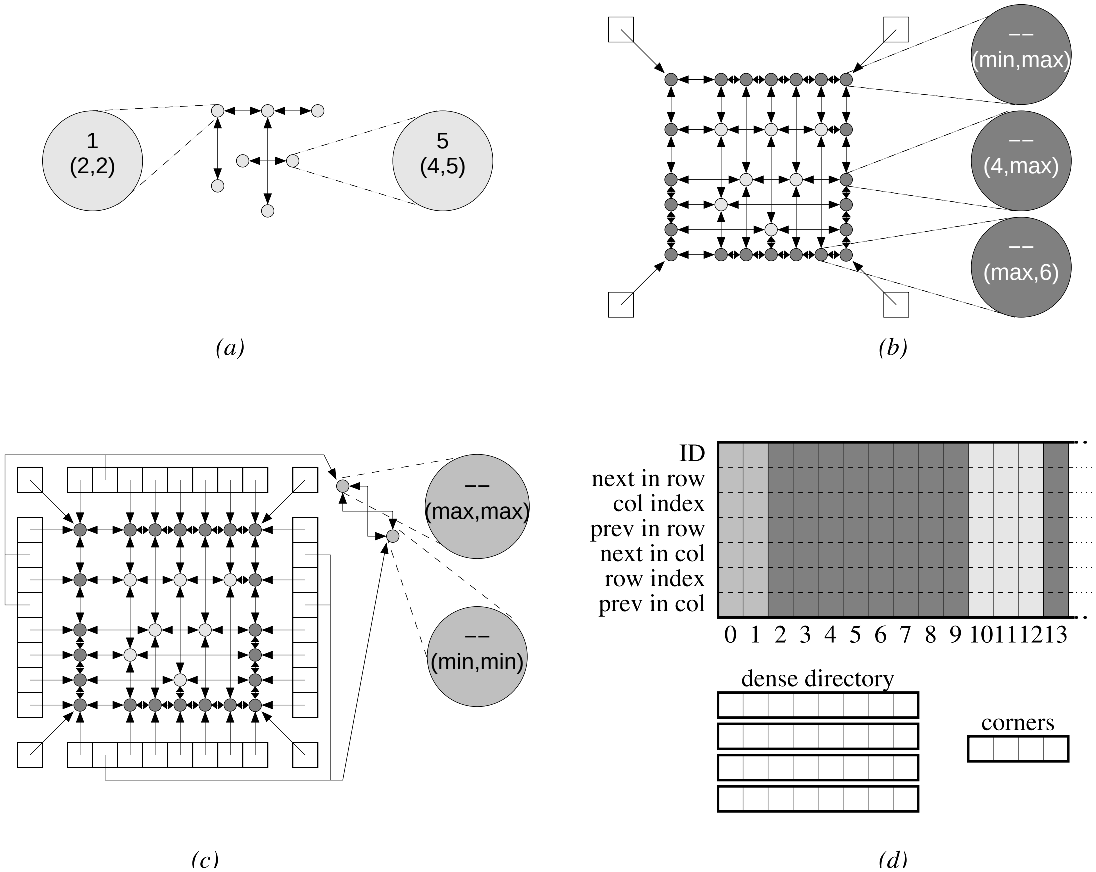


Figure 6.5: A-ZPL's Sparse Representation. (a) The lattice nodes describing the sparsity pattern for the array of Figure 6.1. In addition to a unique ID, each node stores its logical index and pointers to its neighboring nodes in each dimension. The unique ID is used to index into the vector of values that implements a sparse array. (b) The lattice's sparse directory, composed of header nodes. Header nodes have no unique ID and store sentinel values in their logical indices to terminate lattice traversals. An array of pointers to the corners of the directory is used to anchor the directory, and to initiate sparse iteration along any dimension of the region. (c) The dense directory structure used to randomly access a particular row or column. Each directory is implemented using an array or hash table, depending on the sparsity of its dimension. Empty entries point to a shared set of dummy nodes with sentinel values. (d) A naive implementation of this data structure which uses an array of records for the nodes. Each column in the main array represents a single lattice, header, or dummy node, as indicated by its shading. Additional arrays are used to store the corner and dense directories. Note that a single copy of this data structure can be shared between multiple arrays declared using the same region.

Since the lattice may not be strongly connected, some sort of *sparse directory* structure is required to support iteration over all of the nodes. This is achieved by recursively adding *header nodes* to the front and back of each list in the multilist (Figure 6.5b). Since header nodes do not represent region indices, they do not require unique IDs. They do store logical indices, but use minimal or maximal sentinel values for the dimensions in which they act as headers to provide a simple termination condition during iteration. Like traditional nodes, header nodes store pointers to their neighbors in each dimension. The sparse directory of header nodes supports the ability to iterate over the entire region in $\Theta(nnz)$ time. A very loose bound on the required space is $O((3^d - 1) \cdot nnz)$ which is $O(nnz)$ when d is considered a constant.

To provide the random access required by region slicing and random array accesses, a *dense directory* structure is added for each dimension to provide fast access to its lists (Figure 6.5c). This structure is represented using a hash table or array, depending on the density of the indices in each dimension. Empty locations within the directory point to a pair of shared *dummy nodes* that serve as sentinels. Since each list contains $o(n)$ elements, the dense directory provides a means for accessing any index within the region in $\Theta(1) + o(n)$ time—not quite constant, but reasonably close. An upper bound on the space required by the dense directory structure is $O(2d \cdot \min(2nnz, n^{d-1}))$ or $O(nnz)$ when d is considered a constant.

The union of these parts is a very flexible data structure that supports all of the required operations using space proportional to the number of indices represented by the region. While the generality of this structure has a fair amount of overhead (shown in Figure 6.5c), a clever implementation of these components admits an implementation whose memory requirements can be optimized to rival CSR.

6.4.3 Optimizing the Sparse Representation

The actual implementation differs from its logical description in several respects. This section describes these differences incrementally.

In a naive implementation, the collection of nodes used to implement the sparsity structure might be represented using a vector of records, with additional arrays of pointers to represent the corners and dense directory structure (Figure 6.5d). The A-ZPL compiler chooses instead to represent each node field as a separate vector (Figure 6.6a). This refactoring requires that the nodes' pointers be represented using integer indices into the arrays rather than true pointers. Representing the nodes in this manner improves spatial locality due to the fact that most loops do not refer to all of a node's fields. For example, if a region's nodes are traversed in row-major order, the "next in row" fields would be referenced frequently, whereas the "previous in row" and "previous in column" fields would never be used. Implementing each field using a vector allows a cache line's memory to be packed with potentially useful values, rather than being diluted by unused fields. This organization also supports selective allocation, as described below.

The second transformation is to order the nodes in each field such that all header nodes come first, followed by the dummy nodes, the IRV's node, and then the actual lattice nodes, arranged in row-major order. The vectors are then shifted in memory so that index 0 describes the IRV's fields (Figure 6.6b). This transformation has two effects, shown in Figure 6.6c. First, an explicit "ID" field is no longer is required, since each lattice node's ID is now encoded by its index within the vectors. Second, the "next in row" and "previous in row" fields are no longer required for lattice nodes, due to their adjacency in memory—simple index increments and decrements can be used to move along a row instead. Since each field is stored as a separate vector, it can be allocated for only a subset of nodes rather than for all of them. In the current implementation, each field is allocated either for the header nodes, for the lattice nodes, for all nodes, or for none of them.

The final optimization is based on the observation that the complete region structure is not required by every program. For example, a program that only iterates over a sparse region in row-major order has no need for the dense directory, nor for many of the fields within the header and lattice nodes. As the A-ZPL compiler generates loop nests and calls to the runtime libraries, it keeps track of whether each component of the sparse represen-

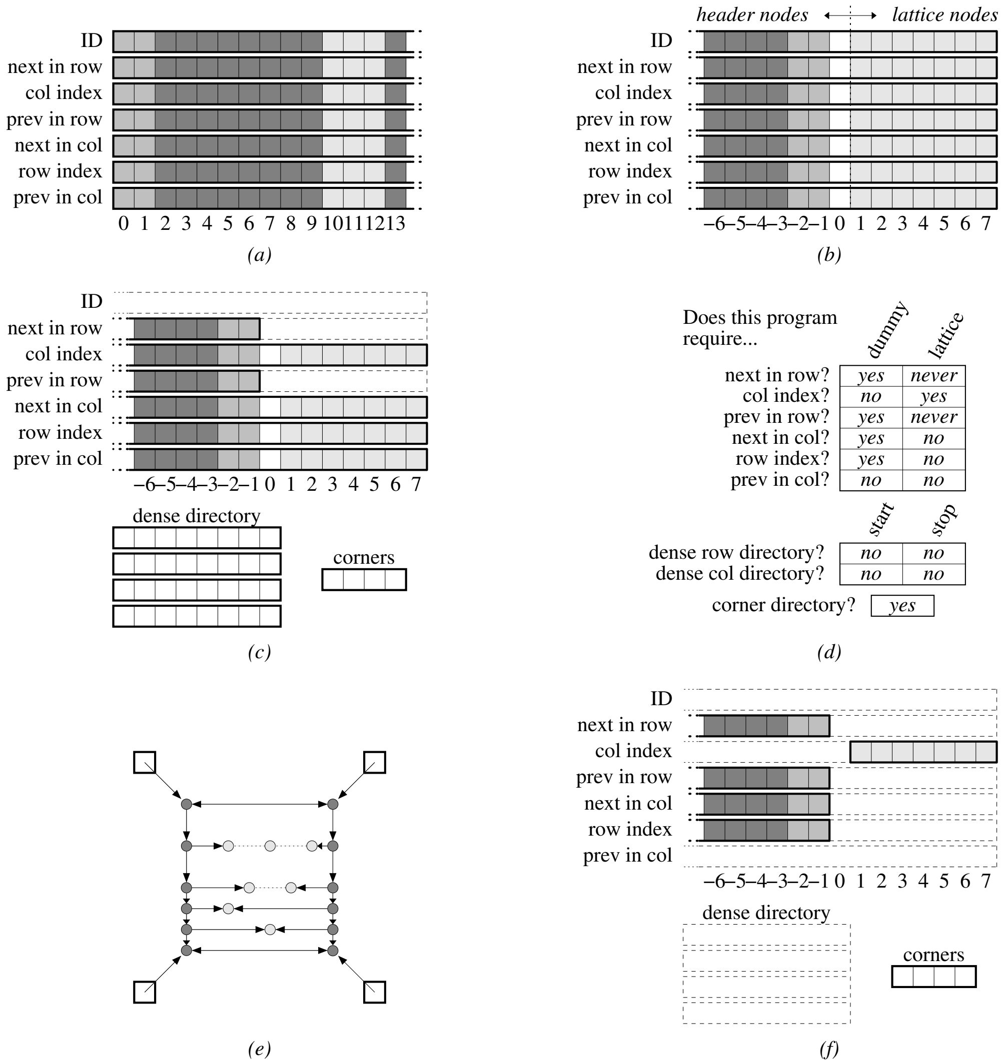


Figure 6.6: Optimizing the Sparse Representation. (a) The nodes of Figure 6.5d, re-implemented using a vector per node field. (b) The same nodes reordered such that all header nodes come first, followed by the dummy nodes, the IRV's node, and then the lattice nodes stored in row-major order. Each vector is shifted so that the IRV lies at index 0. (c) Optimizations supported by this node reordering. Explicit node IDs are no longer necessary because each element's position can now serve as its ID. Furthermore, pointers to a lattice node's neighbors within a row are no longer required as those nodes will be adjacent in memory. (d) Summary information generated by the compiler on a per-region basis, indicating which components of the data structure are required by a particular program. (e) The simplified data structure required by programs that only use row-major order iteration, such as CG and the sparse implementation of MG. (f) The resulting optimized implementation.

tation is used by the generated code. For each region, the compiler emits a summary of this information as shown in Figure 6.6d. This summary is used by the sparse region's runtime initializer to allocate only the structures that are required. For example, in the CG and MG benchmarks, the compiler detects that the sparse regions are only traversed in row-major order. Therefore, the initialization code eliminates the sparse region's dense directory, much of its sparse directory, and most of the fields for the lattice and header nodes (Figure 6.6e–f). The result is a representation that is quite similar to CSR, both in size and functionality.

6.4.4 Sparse Descriptor Summary

To summarize the implementation of sparse regions and arrays, each uses the traditional dense descriptor as a starting point. Sparse regions add to this a number of dynamically allocated arrays to store (1) the nodes' logical indices and next and previous pointers, (2) the corners of the sparse directory, and (3) the dense directory. Each of these arrays is allocated on an as-needed basis. In addition, the sparse region descriptor stores the number of sparse indices that are local to the processor and the total number of sparse nodes allocated by the processor.

Sparse array descriptors are identical to dense array descriptors, except that they are set up to represent a 1-dimensional vector of values rather than the logical d-dimensional array. The number of values is determined by reading the appropriate field in the sparse region descriptor.

6.4.5 Initializing Sparse Region Descriptors

The sparsity structure for a sparse region is set up as follows: Within the region's local initializer, a traditional m-loop is generated to iterate over the base region's indices. Within the loop, the boolean sparsity pattern expression is evaluated for each index. Any index that evaluates as "true" is stored at the end of a dynamically-grown array of indices. Once the m-loop has completed, the indices are sorted into row-major order using a linear-time radix/bucket sort [CLR92] that short-circuits in the presence of pre-sorted sequences.

Next, the dense directory structure is allocated to serve as a frame for "knitting" the lattice of nodes together. If the dense directory is not required by the user's program, it will be deallocated once the region's initialization is finished. All dense directory locations are initialized to refer to the dummy nodes. The sorted index list is then traversed. For each index, its projection to each of the dense directories is calculated, and if that position contains a reference to the dummy nodes, new header nodes are allocated and their indices are added to the region's dynamically grown logical index vectors. Once all of the indices have been processed, the projection is repeated for the header nodes themselves, to recursively add header nodes to the multidimensional structure. When this completes, the processor knows how many lattice and header nodes it requires, and stores these values in the region descriptor. It also allocates the nodes' field vectors as specified by the A-ZPL compiler.

Next, all of the header nodes are linked together to form the complete sparse directory structure. Then, the lattice nodes are linked into the structure one at a time using traditional sorted doubly-linked list insertions for each dimension.

Though this entire process is somewhat complex, it can be done in $\Theta(nnz)$ time and space, and a clean implementation can keep the constant factors fairly reasonable.

6.5 Advanced Sparse Implementation Notes

6.5.1 Sparse Region Operators, Dynamic Regions

Given the amount of space required by even an optimized sparse region descriptor, it makes sense to share sparse representation components between regions whenever possible. Examples of this include region operators applied to sparse arrays and dynamic regions that inherit from sparse regions. These cases are considered here.

Dynamic Regions with Blank Dimensions

Dynamic regions that inherit from sparse regions using blank dimensions can be implemented by computing their bounding indices as usual and storing these in the traditional region descriptor fields. All fields related to the sparsity pattern can be implemented by referring to the same vectors as the base region. Although the complete set of nodes will represent a superset of the indices described by the new dynamic region, its m-loops will be generated such that only those indices within its bounding box are referenced. Thus, these regions require no additional overhead to store a sparsity structure.

***`in`** and **`of`** Regions*

As in the case of blank dimensions, using the **`in`** and **`of`** region operators on a sparse region simply refers to a subset of that region's sparsity pattern. Thus, the implementation is the same: the sparse region's bounds are computed as normal and the sparse fields are shared with the base region.

***`at`** Regions*

Regions created by applying the **`at`** operator to a sparse region have the ability to inherit much of their structure from the base region, but not all of it. Most importantly, the logical indices of the lattice and header nodes must be shifted by the direction's values. This implies that for each dimension in which the direction is non-zero, the base region's logical index vectors must be explicitly copied and shifted. The logical index vectors for dimensions in which the direction is zero may simply be reused. This implementation demonstrates yet another benefit of A-ZPL's use of vectors to implement node fields: only fields that require changes have to be explicitly copied. The use of array indices for node pointers makes such changes transparent to the rest of the data structure.

The dense directory arrays must also be shifted to point to the right values. This can be done by copying and modifying their array descriptors, while inheriting the actual buffer of pointers. The base region's dense directory structure may also need to be extended to support references to indices that lie outside of its bounding box. This modification is similar to the implicit region storage already performed by the compiler, so it does not require any special support.

***by* Regions**

Applying the **by** operator to a sparse region is slightly more complicated due to the fact that an arbitrary subset of the lattice nodes may be described by the new region. This makes it difficult to simply refer to the existing sparsity structure. One option is to do so and then create m-loops that skip over any nodes that do not have the proper alignment. The second option is to make an explicit copy of the nodes contained in the new region. This represents a time versus space tradeoff that has not yet been explored in this work.

6.5.2 Implementing Fluff

Fluff is implemented somewhat differently for sparse regions than for dense. In particular, recall that fluff is associated with an array in the dense context. In the sparse context, fluff is associated with the array's region. The reason for this is that it does not suffice to know that a row of fluff values must be stored on an array's north boundary—the actual indices in that row must be stored as well. In order to avoid encoding sparsity in the array descriptor, the region's sparse directory is extended by the appropriate amount, and the remote indices are either computed locally or communicated, depending on the region's sparsity pattern. For example, the remote indices of a tridiagonal region can be computed on each processor in isolation, whereas more complex sparsity patterns will require communication to determine a processor's fluff indices.

This implementation of fluff has the disadvantage that one array's fluff requirements will cause the same fluff values to be allocated for other arrays declared using the sparse region, even if they are not required. Since the number of values communicated by any sparse @-reference is assumed to be quite small, this is a reasonable investment of memory when compared to the amount that would be required to replicate the sparsity pattern for each array's distinct fluff requirements.

6.5.3 Adapting Runtime Libraries

A final implementation issue is the adaptation of the runtime libraries to support sparse regions. For example, the machine-independent interface for the @ operator requires modification so that in the presence of sparse regions and arrays it can describe the memory layout of a sparse set of data values to the Ironman interface. The current approach used for the libraries is to specialize each routine based on whether its argument regions and arrays are sparse or dense. For example, when the machine-independent Ironman interface finds that it has been passed a sparse array, it allocates a temporary buffer in which to store sparse values during communication.

Although this approach works, it also requires the specialization of each runtime library routine. An alternative approach would be to add function pointers to all region descriptors to support a consistent implementation of operations like iteration. Section 6.9.1 discusses this approach in a bit more detail.

6.6 Sparse Code Idioms

The key to generating code for sparse regions and arrays is to be able to iterate over a sparse region's structure in a variety of contexts. This section describes some of the idioms used to implement sparse array statements.

6.6.1 Sparse M-loops and Accesses

For Sparse Region Scopes

As an example of a simple sparse m-loop, consider the second assignment in Figure 6.3. This statement requires its m-loop to iterate over region `Rs` which also describes the spar-

Listing 6.7: A Simple Sparse M-loop

```
if (I_OWN(Rs)) {            /* traditional m-loop guard */
  INIT_2D_LOOP(N, N, Rs);  /* traditional 2D loop initialization */
  int Rs_Walker0;          /* this "pointer" walks over Rs' first dim */
  int Rs_Walker1;          /* this one walks over its second */
  int Rs_Stop;             /* the stopping point for the inner loop */
  int* Bs_Ptr;             /* pointers used to refer to values of Bs */
  int* const Bs_IRV = (int*)(Bs->origin);

  /* start at the top left corner */
  Rs_Walker0 = Rs->corner[0x0|0x0];
  /* advance past the row of header nodes */
  Rs_Walker0 = Rs->next[0][Rs_Walker0];

  /* while the walker is within the region's bounds, advance it */
  for ( ; (i0 = Rs->index[0][Rs_Walker0]) <= i0Hi;
       Rs_Walker0 = Rs->next[0][Rs_Walker0]) {

    /* start the second walker at the current position */
    Rs_Walker1 = Rs_Walker0;
    /* save stopping point (opposite headers are always adjacent) */
    Rs_Stop = Rs->prev[1][Rs_Walker0 + 1];
    /* advance past the column of header nodes */
    Rs_Walker1 = Rs->next[1][Rs_Walker0];

    /* until the stopping point is reached, increment the pointer */
    for ( ; (i1 = (Rs->index[1][Rs_Walker1]), Rs_Walker1 <= Rs_Stop);
         Rs_Walker1++) {
      /* grab the current location of B */
      Bs_Ptr = Bs_IRV + Rs1_Walker1;

      /* use a traditional array access to read from the dense array */
      *Bs_Ptr = *(int*)ACCESS_2D(N, N, A, i0, i1);
    }
  }
}
```

Listing 6.8: The Simple Sparse M-loop Using Macros

```
if (I_OWN(Rs)) {
  INIT_2D_LOOP(N, N, Rs);
  INIT_SPS_REG_MAIN(Rs, 2);    /* declare sparse walkers */
  INIT_SPS_ARR_WRIT(int, Bs); /* declare value pointers for Bs */

  SPS_WALK_SETUP_MAIN(Rs, 0, UP0 | UP1);
  SPS_MAIN_MLOOP_UP(Rs, 0) {
    SPS_COPY_WALKER_MAIN_INNER_UP(Rs, 0, 1);
    SPS_MAIN_MLOOP_UP_INNER(Rs,1) {
      SPS_PTR(Bs) = (SPS_ORIG(Bs) + SPSID_MAIN(Rs, 1));
      *SPS_PTR(Bs) = *(int*)ACCESS_2D(N, N, A, i0, i1);
    }
  }
}
```

sity pattern of `Bs`. Listing 6.7 shows the code that implements this assignment. The loop traverses the sparse region's nodes using two integer values (`Rs_Walker`i), one for each dimension. Since each lattice node's ID is the same as its index, these values are used both as node pointers and to access values of `Bs`. As usual, the compiler generates sparse m-loops using macros to simplify many of the repetitive details. Listing 6.8 shows the same loop in its sanitized form.

For Sparse Array References

The other type of sparse m-loop iteration iterates over a sparse region's nodes within the context of a dense m-loop or a sparse m-loop with a different structure. In these cases, the sparse structure is traversed as before, but in a passive way to keep the array's references synchronized with the controlling m-loop. When possible, these traversals are shared between all array references with the same sparse region. Listing 6.9 shows a simple example that implements the third assignment of Figure 6.3.

This code is somewhat more complex than the previous case due to the fact that the sparse region is no longer controlling the m-loop's iteration. Rather, it has to keep pace

Listing 6.9: Iterating over a Sparse Region Within a Dense M-loop

```
if (I_OWN(R)) {            /* traditional m-loop guard */
  INIT_2D_LOOP(N, N, R);   /* traditional 2D loop initialization */

  int Rs_Walker0;  /* Values for iterating over the sparse region Rs */
  int Rs_Walker1;
  int Rs_Stop;     /* Stopping point for the inner dimension of Rs */
  int Rs_i0;       /* Indices of the current Rs Walkers */
  int Rs_i1;

  /* Values for accessing the sparse array Bs */
  int Bs_Val;
  const int* const Bs_Orig = (int*)(Bs->origin);

  /* Start in the corner of Rs and set index to its sentinel value */
  Rs_Walker0 = Rs->corner[0x0|0x0];
  Rs_i0 = Rs->index[0][Rs_Walker0];

  /* Traditional dense m-loop */
  MLOOP_UP(R, 0) {
    /* if the Rs iteration is lagging behind... */
    if (Rs_i0 <= i0) {
      /* advance it until it catches up */
      while (Rs_i0 <= i0) {
        Rs_Walker0 = Rs->next[0][Rs_Walker0];
        Rs_i0 = Rs->index[0][Rs_Walker0];
      }
      /* if it matches the current index */
      if (Rs_i0 == i0) {
        /* set up the next dimension for iteration */
        Rs_Walker1 = RsWalker0;
        Rs_Stop = Rs->prev[1][Rs_Walker1 + 1];
        Rs_Walker1 = Rs->next[1][Rs_Walker0]-1;
        Rs_i1 = INT_MIN;
      } else {
        /* otherwise, set the indices to prevent iteration */
        Rs_i1 = INT_MAX;
        Bs_Val = *(Bs_Orig);
      }
    }
    /* the inner loop of the dense m-loop */
    MLOOP_UP(R, 1) {
      /* if the inner dimension needs catching up */
      if (Rs_i1 <= i1) {
        /* advance it, making sure not to fall off the row */
        while ((Rs_i1 < i1) && (Rs_Walker1 < Rs_Stop)) {
          Rs_Walker1++;
          Rs_i1 = Rs->index[1][Rs_Walker1];
        }

        /* set B's value based on whether or not this is a hit */
        Bs_Val = *(Bs_Orig + ((Rs_i1 == i1) ? (Rs_Walker1) : 0));
      }

      /* assign to A (typically walkers and bumpers would be used) */
      *(int*)ACCESS_2D(N, N, A, i0, i1) = Bs_Val;
    }
  }
}
```

Listing 6.10: Iterating over a Sparse Region Within a Dense M-loop Using Macros

```
if (I_OWN(R)) {
  INIT_2D_LOOP(N, N, R);

  INIT_SPS_REG(Rs,2);          /* declare walkers, indices */
  INIT_SPS_ARR_READ(int, Bs); /* declare values for Bs */

  SPS_WALK_SETUP(Rs, 0, UP0 | UP1);
  MLOOP_UP(R, 0) {
    if (SPS_NEEDS_CATCHING_UP(Rs, 0)) {
      SPS_CATCH_UP(Rs, 0);
      if (SPS_HIT(Rs, 0)) {
        SPS_COPY_WALKER_INNER_UP(Rs, 0, 1);
      } else {
        SPS_MAXIFY_DIM(Rs,1);
        SPS_VAL(Bs) = *(SPS_ORIG(Bs));
      }
    }
    MLOOP_UP(R, 1) {
      if (SPS_NEEDS_CATCHING_UP(Rs, 1)) {
        SPS_CATCH_UP_INNER(Rs, 1);
        SPS_VAL(Bs) = *(SPS_ORIG(Bs) + SPSID_VAL(Rs,1));
      }

      *(int*)ACCESS_2D(N, N, A, i0, i1) = SPS_VAL(Bs);
    }
  }
}
```

with it, tracking whether each index is a “hit” or a “miss.” As always, macros make this code much more readable (Listing 6.10).

6.6.2 Rank-Independent Sparse M-loops

As with dense m-loops, it is possible to create a rank-independent sparse m-loop using a vector of integers to represent the node traversals in each dimension. This implementation is quite similar to the dense case and therefore omitted from this discussion for brevity. Such loops are useful for traversing a sparse region within the runtime libraries. For example, they would be used to marshall a sparse set of values for communication in the machine-independent Ironman routines.

6.6.3 Optimized Sparse M-loops

Sparse m-loops present a number of opportunities for optimization. This section describes a few examples.

Specializing Dense Loops

One optimization is motivated by the amount of control flow introduced by the traversal of a sparse region within a dense m-loop, as demonstrated by Listing 6.10. The presence of this control flow greatly increases the m-loop’s complexity and can hurt its performance due to the potential for branching in each iteration. This overhead is especially regrettable given that the branches will have a strong tendency to take the “miss” path due to the fact that there are asymptotically fewer sparse values than dense indices.

For this reason, dense m-loops can be specialized to maximize the amount of branch-free code between sparse values. Listing 6.11 gives an example by re-implementing the loop of Listing 6.10. The loops in lines 16–20 and 32–34 are the key to the code, implementing the m-loop for a range of indices that contains no sparse indices. The first loop iterates over a set of rows with no sparse indices. The second iterates over a segment of

Listing 6.11: An M-loop Specialized for Dense Index Ranges

```
if (I_OWN(R)) {
  INIT_2D_LOOP(N, N, R);

  INIT_SPS_REG(Rs,2);          /* declare walkers, indices */
  INIT_SPS_ARR_READ(int, Bs); /* declare values for Bs */

  SPS_WALK_SETUP(Rs, 0, UP0 | UP1);
  i0 = i0Lo;
  while (i0 <= i0Hi) {               /* main loop over dim 0 */
    if (SPS_NEEDS_CATCHING_UP(Rs, 0)) {
      SPS_CATCH_UP(Rs, 0);
    }
    if (!SPS_HIT(Rs, 0)) {           /* if Rs misses in dim 0 */
      i0Stop = min(i0Hi+1, Rs_i0);  /* loop to next sparse index */
      SPS_VAL(Bs) = *SPS_ORIG(Bs);  /* using Bs' IRV as its value */
      for ( ; i0<i0Stop; i0++) {
        for (i1=i1Lo; i1<=i1Hi; i1++) {
          *(int*)ACCESS_2D(N, N, A, i0, i1) = SPS_VAL(Bs);
        }
      }
    }
    if (SPS_HIT(Rs, 0)) {
      SPS_COPY_WALKER_INNER_UP(Rs, 0, 1);
      i1 = i1Lo;
      while (i1 <= i1Hi) {         /* iterate over dim 1 */
        if (SPS_NEEDS_CATCHING_UP(Rs, 1)) {
          SPS_CATCH_UP_INNER(Rs, 1);
        }
        if (!SPS_HIT(Rs, 1)) {     /* if Rs misses in dim 1 */
          i1Stop = min(i1Hi+1, Rs_i1);
          SPS_VAL(Bs) = *SPS_ORIG(Bs);
          for ( ; i1<i1Stop; i1++) { /* loop to next sparse index */
            *(int*)ACCESS_2D(N, N, A, i0, i1) = SPS_VAL(Bs);
          }
        }
        if (SPS_HIT(Rs, 1)) {
          SPS_VAL(Bs) = *(SPS_ORIG(Bs) + SPSID_VAL(Rs,1));
          *(int*)ACCESS_2D(N, N, A, i0, i1) = SPS_VAL(Bs);
          i1++;
        }
      }
      i0++;
    }
  }
}
```

a row with no sparse indices. The bounds of these loops are determined using the indices of the sparse node pointers, which indicate the next sparse index that will be encountered. Since most of the dense indices will not have a corresponding sparse element, the majority of the time will be spent in the simplified loops.

This optimization has not currently been implemented in the A-ZPL compiler. However, initial studies have shown that it benefits codes that read extremely sparse arrays within dense regions. One such example is found in the naive sparse implementation of MG that refers to the input array within the context of the hierarchy's finest grid (Section 6.3.3).

Iterating over a Single Sparsity Pattern

A second optimization occurs when all array references within a sparse m-loop are declared using that same sparse region. In this case, the logical m-loop can be abandoned altogether and the computation can be performed on the sparse arrays' dense representations. For example, consider the following code:

```
var As, Bs: Cs: [Rs] integer;

[Rs] Cs := As + Bs;
```

For such a statement, the compiler can detect that `As`, `Bs`, and `Cs` share the same sparsity pattern and implement the assignment using a loop over their dense vector of values. This would appear as shown in Listing 6.12.

The only tricky case for this optimization is that some sparse arrays may have fluff values mixed in with their normal values. For operations that cannot cause exceptions, computing over the fluff values represents wasted computation. However, the performance saved by ignoring `Rs`' sparsity structure more than makes up for it. For example, the loop above may sum fluff values of `As` and `Bs`, storing the result in the corresponding fluff value of `Cs`. This addition cannot cause an arithmetic exception and the fluff values will be refreshed before they are referenced again, so the optimization can be performed without concern.

Listing 6.12: A Sparse M-loop Optimized due to its use of a Single Sparsity Pattern

```
int* As = (int*)(As->origin) + 1;
int* Bs = (int*)(Bs->origin) + 1;
int* Cs = (int*)(Cs->origin) + 1;

/* loop over the number of lattice nodes in the region */
for (i = 0; i < Rs->num_lattice_nodes; i++) {
  *Cs := *As + *Bs;  /* perform the operation */

  Cs++;              /* increment the data pointers */
  As++;
  Bs++;
}
```

Operations that can cause exceptions have to be treated more carefully. For example, if the addition in the code above was changed to division and `Bs` contained stale fluff values of 0, the optimization would result in an exception that the original code would not have thrown.

Note that this optimization is only possible due to the fact that sparse fluff values are associated with regions rather than individual arrays. This guarantees that all values of `As`, `Bs`, or `Cs` will be aligned, regardless of each array's actual fluff requirements.

This optimization is currently unimplemented in the A-ZPL compiler. However, previous work shows that when the optimization is applied to simple codes by hand, performance can improve by an order of magnitude [CLS98].

6.7 Evaluation

This section evaluates the design and implementation of sparse regions by considering their use in the sample codes of Section 6.3—sparse matrix-vector multiplication, tridiagonal matrix multiplication, and the CG and MG benchmarks. Each benchmark is evaluated in terms of clarity, memory usage, and performance.

6.7.1 Implementations

All four benchmarks are written in A-ZPL using sparse regions. Other versions of sparse matrix-vector multiplication include the sequential C implementation from Chapter 3, a sequential C implementation using the CSR format (Appendix A), and the 2D ZPL implementation. The sparse matrices are generated to have $\log n$ evenly distributed non-zeroes per row. The matrix values in the dense C and ZPL codes are different than Chapter 3 to reflect the sparsity of the matrix.

Two A-ZPL implementations of tridiagonal matrix multiplication are used. The first uses two sparse regions, one to describe the tridiagonal index set and the second to describe the pentadiagonal. The second implementation uses a single pentadiagonal region to represent all three matrices. These implementations are compared to the compact ZPL and C implementations, as well as the shard-based ZPL implementation.

Both CG and MG are compared to the original hand-coded F77+MPI NAS implementations. In addition, the sparse implementation of MG is compared to the traditional dense ZPL implementation.

6.7.2 Clarity

Figure 6.7 indicates the number of lines required to implement each version of each benchmark. As in previous chapters, each productive line of code is classified as being a declaration, communication, or computation. These graphs indicate that the introduction of sparsity tends not to complicate the dense ZPL implementations.

For example, the A-ZPL implementation of sparse matrix-vector multiplication uses the same number of lines as the 2D ZPL version since the only thing that needs to change is the region declaration. Similarly, in the tridiagonal codes, the number of declaration lines vary only slightly between the A-ZPL versions and the shard-based ZPL version on which it was based. These fluctuations are indicative of the number of regions being declared in each code. The A-ZPL versions use slightly less computation due to the fact that the array's

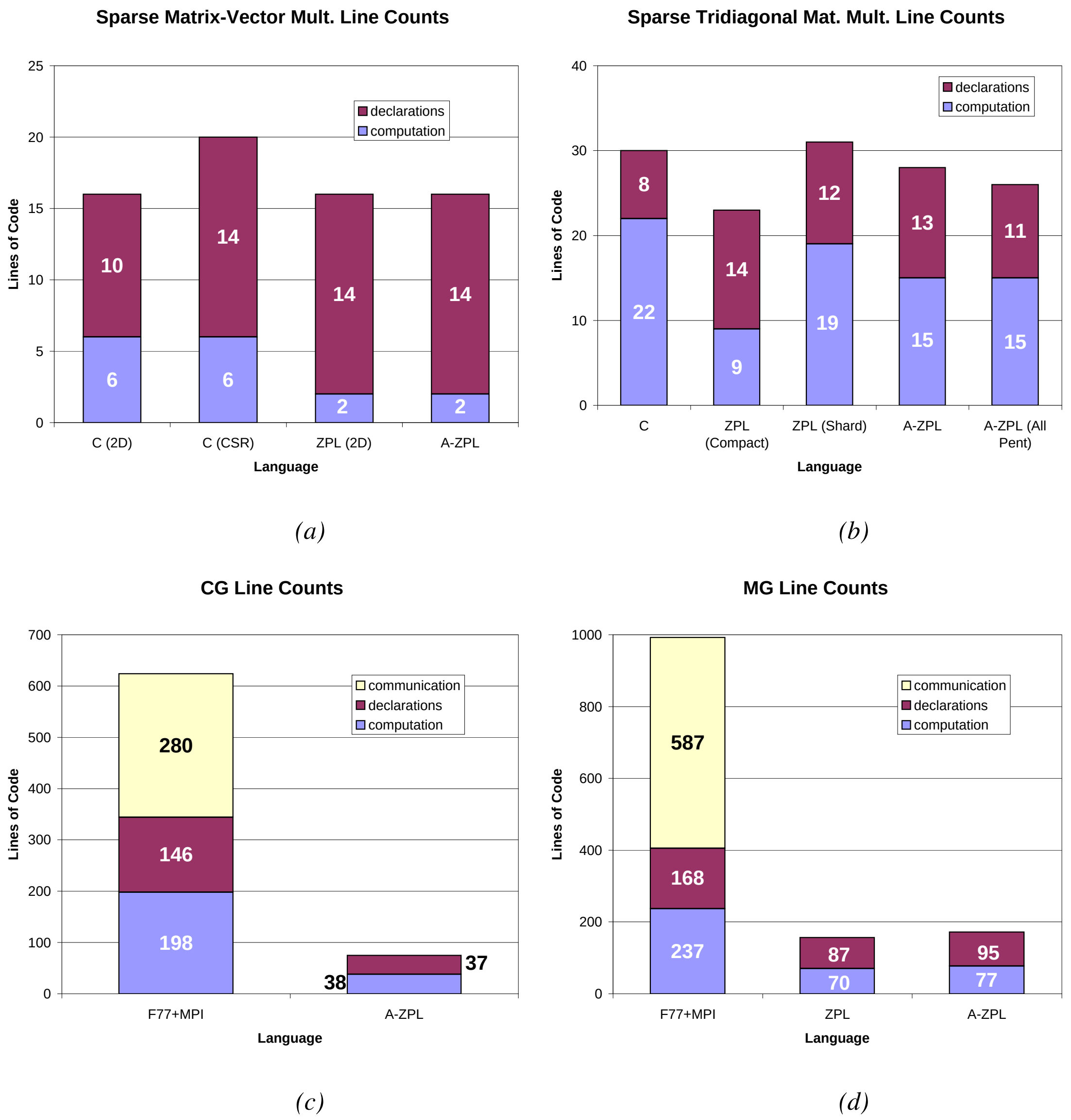


Figure 6.7: Linecounts for Sparse Benchmarks. Each graph indicates the number of productive lines of code used to implement each benchmark, classified as communication, declarations, or computation. (a) Implementations of matrix-vector multiplication written in C using traditional dense arrays and the CSR format, in ZPL using a dense region, and in A-ZPL using a sparse region. (b) Implementations of tridiagonal matrix multiplication in C and ZPL using a compact format, in ZPL using a dense implementation, and in A-ZPL using a sparse region. (c) The CG benchmark as implemented by NAS in F77+MPI and in A-ZPL using sparse regions. (d) The MG benchmark as implemented by NAS in F77+MPI, in ZPL using a dense input array, and in A-ZPL using a sparse input array.

boundaries no longer need to be zeroed out—they are simply excluded from the index set and are therefore implicitly represented by the IRV of zero.

The A-ZPL implementation of MG requires a few more lines than the shard-based ZPL version in order to declare the specialized version of the *resid* stencil as described in Section 6.3.3. Even with these additional lines, the A-ZPL implementation remains far more concise than the F77+MPI version due to its lack of loops and explicit communication.

The A-ZPL implementation of CG shows a similar result. More than half of the lines in the NAS version are devoted to handling parallel communication and data distribution issues by hand. In contrast, A-ZPL's use of sparse regions results in a succinct, clear representation of the algorithm that is one-eighth the size of the F77+MPI code.

These graphs indicate that sparse regions succeed in their ability to express sparse computation succinctly. Examination of the codes by hand confirms that they are expressed using computation identical to that of the equivalent dense codes, modulo the specifications of the sparse regions' indices.

6.7.3 Memory Usage

The graphs in Figure 6.8 indicate the amount of memory required to execute each benchmark for the indicated problem size on a single processor. Memory is expressed using megabytes or gigabytes, as indicated by each graph's y-axis label.

The sparse matrix-vector multiplication results indicate that the optimized A-ZPL representation requires a half megabyte more memory to represent the sparse indices than a hand-coded CSR implementation. This excess memory is due to the fact that A-ZPL's fully optimized representation is not quite as compact as a strict CSR implementation since it uses a sparse directory of rows rather than a dense row directory. This sparse directory can benefit arrays that do not contain indices in every row like MG's input array. Programs that do contain indices in each row pay a bit of overhead compared to CSR due to the row indices and "next in column" pointers. Nevertheless, it should be noted that this memory represents less than 2% of the total memory required by the benchmark, and is therefore

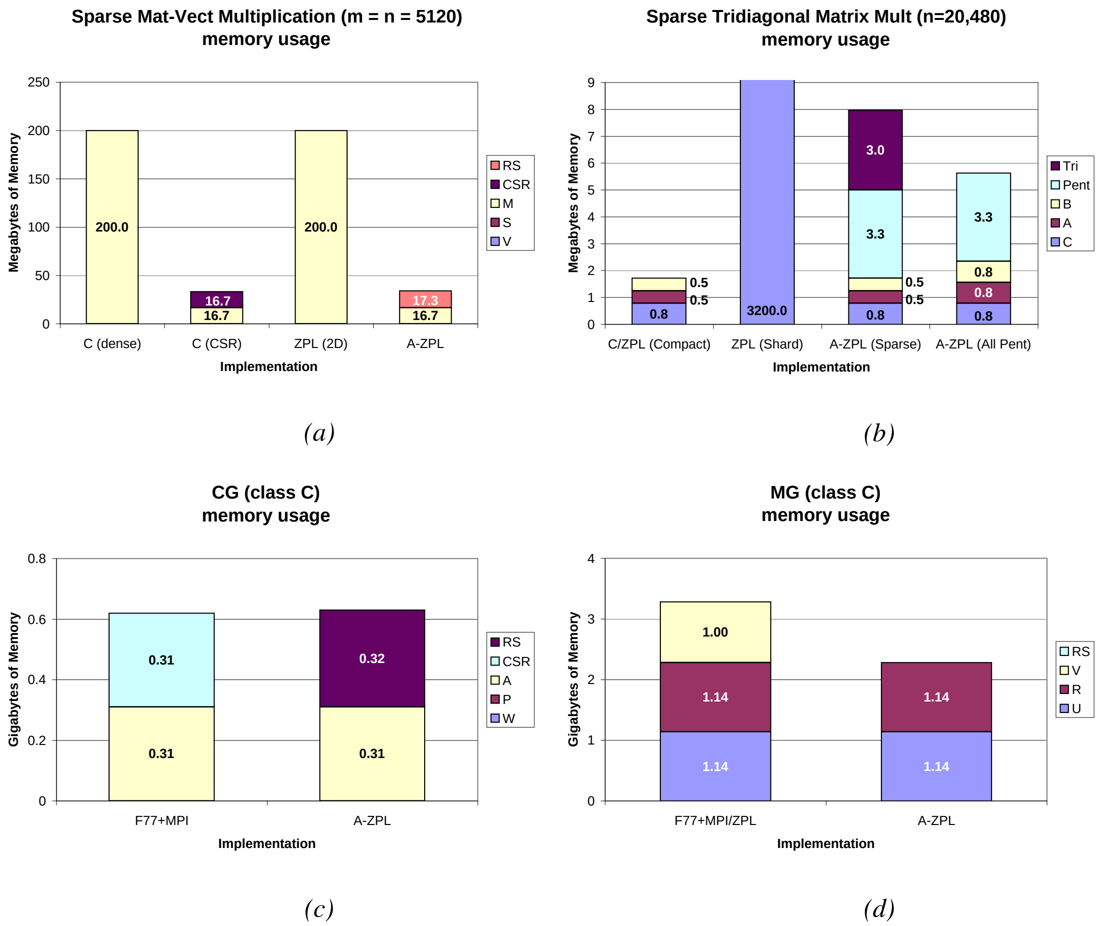


Figure 6.8: Memory Requirements for Sparse Benchmarks. These graphs indicate the memory used by each benchmark's main data structures when run on a single processor. In general, "RS" indicates the memory required by a sparse region whereas "CSR" indicates the memory used by a hand-coded CSR format. (a) Sparse matrix-vector multiplication as written in C using dense arrays and the CSR format, in ZPL using a dense region, and in A-ZPL using a sparse region. Recall that "M" is the sparse matrix, while "V" and "S" are the input and solution vectors. (b) Sparse tridiagonal matrix multiplication as written in C and ZPL using a compact format, in ZPL using a dense format, and in A-ZPL using two or one sparse regions. "A," "B," and "C" are the three matrices. "Tri" and "Pent" are the sparse tridiagonal and pentadiagonal regions as declared in A-ZPL. (c) The CG benchmark as written by NAS in F77+MPI and in A-ZPL using a sparse region. "P" and "W" are vectors, whereas "A" represents the sparse array values. (d) The MG benchmark as written in F77+MPI or ZPL using a dense input array and in A-ZPL using a sparse input array. Recall that "U" and "R" are the two hierarchical arrays and that "V" is the input array.

negligible. More importantly, both of the sparse implementations use less than $1/5$ of the memory used by their dense equivalents, making the use of sparsity worthwhile.

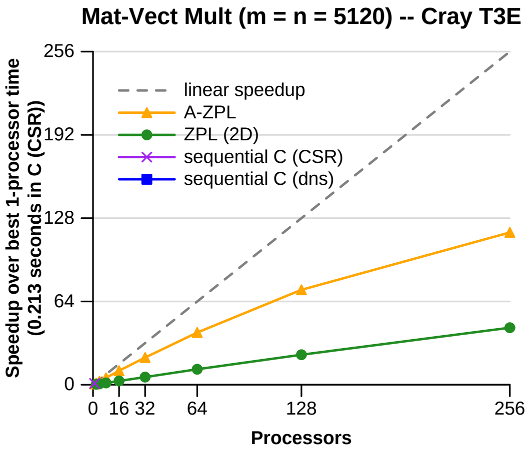
The tridiagonal matrix multiplication codes demonstrate this point even more dramatically. Each of the three arrays requires 3200 megabytes of memory when using dense $n \times n$ storage as in ZPL's shard-based implementation. In contrast, the sparse A-ZPL implementations use around 8 and 6 megabytes each. While this is still 2–3 times the amount of memory required by the compact C and ZPL implementations, it represents a reasonable amount of memory given its preservation of the logical $n \times n$ computation space without explicitly allocating any arrays of that size. As anticipated, the use of a single pentadiagonal region to represent all three arrays requires slightly more memory for each tridiagonal array to store the extra pair of diagonals. However, it also saves quite a bit of memory by eliminating the tridiagonal region's sparse representation, making it a worthwhile tradeoff.

Class C of the CG benchmark uses a $150,000 \times 150,000$ matrix with approximately 31 million non-zeroes. The memory counts for CG give a very similar result to the sparse matrix-vector multiplication benchmark: A-ZPL's sparse region requires slightly more memory than the equivalent CSR format, but the difference is negligible when compared to the total amount of memory used.

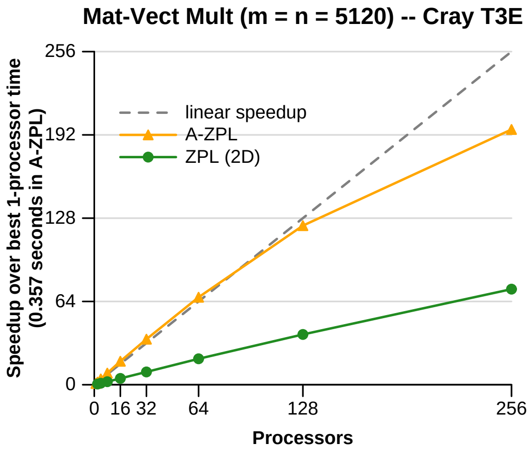
The MG benchmark demonstrates the benefit of taking advantage of sparsity in a code traditionally implemented using dense arrays. The elimination of the input array reduces the benchmark's memory requirements by almost a third. Moreover, the extreme sparsity of the input array results in a very compact sparse region implementation. Its storage requires just a few kilobytes of memory and is completely negligible when compared to the benchmark's hierarchical arrays. This savings in memory allows programmers to run larger problem sizes on smaller numbers of processors.

6.7.4 Performance

Each implementation of the benchmark was run on the Cray T3E to evaluate the performance obtainable using sparse regions. Technical details for the T3E and its compilers are



(a)



(b)

Figure 6.9: Performance for Sparse Matrix-Vector Multiplication. (a) A speedup graph for matrix-vector multiplication comparing the sparse A-ZPL implementation against the dense ZPL implementation and sequential versions in C written using dense arrays and the CSR format. (b) The same comparison, but without the sequential C versions.

given in Appendices B and C. Figures 6.9–6.11 show the resulting speedup graphs. The raw timings used to construct these graphs are given in Appendix D.

Figure 6.9 shows the speedup results for matrix-vector multiplication with and without comparison to the sequential C codes. As expected, reducing the computational complexity and memory footprint of the algorithm from $\Theta(mn)$ to $\Theta(nnz)$ ($\Theta(nlogn)$ for these experiments) results in a significant performance benefit. The overhead associated with iterating over A-ZPL's sparse region representation is more than made up for by the asymptotic improvement in computation and memory references. As the number of sparse indices per processor decreases, the overhead of communication becomes a bigger and bigger factor in the A-ZPL implementation, causing its speedup to drop off slightly.

Figure 6.10 shows that the A-ZPL implementations of sparse tridiagonal matrix multiplication perform significantly better than the dense implementation, but still fail to match the performance of the compact C and ZPL implementations. This is not surprising con-

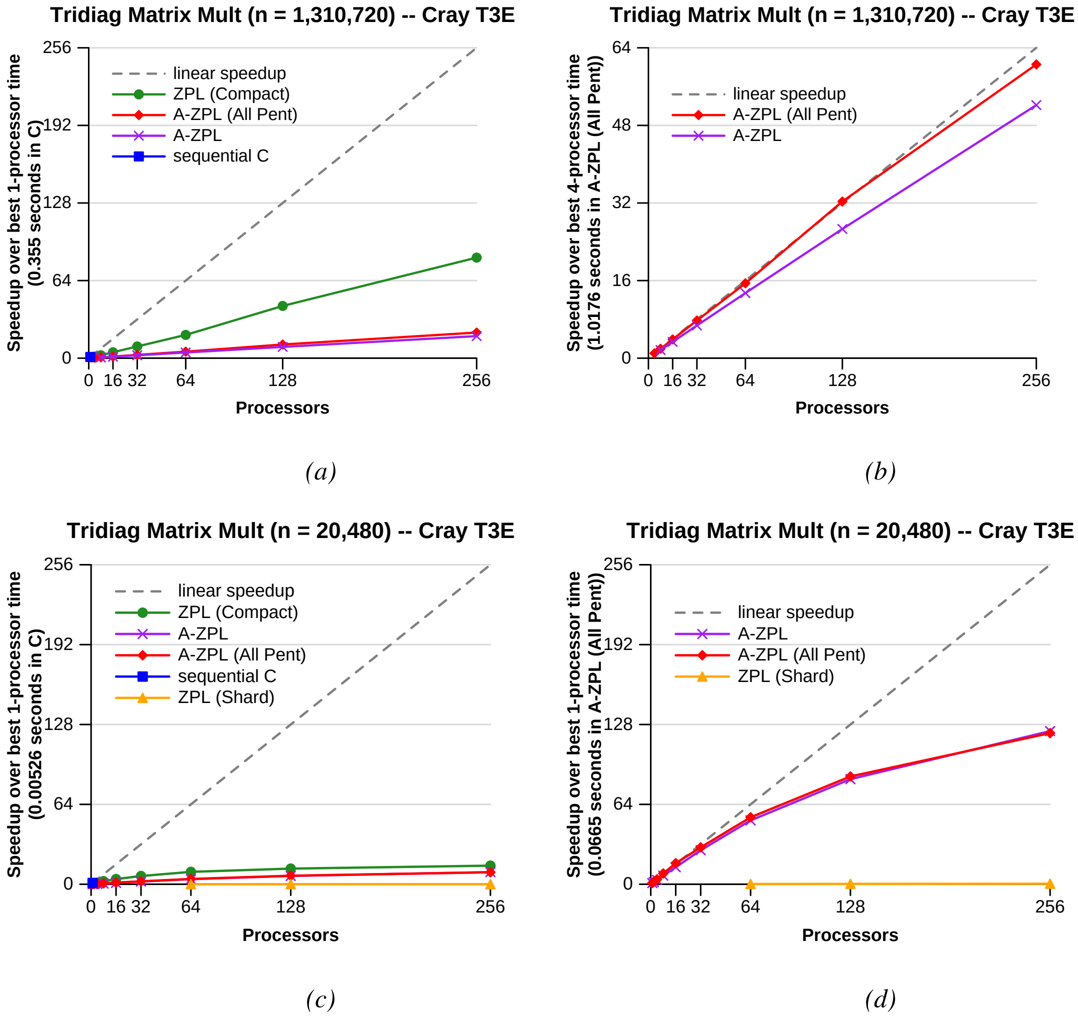


Figure 6.10: Performance for Sparse Tridiagonal Matrix Multiplication. (a) A speedup graph for tridiagonal matrix multiplication comparing the sparse A-ZPL implementations against compact implementations in ZPL and C. (b) The same comparison, but without the compact C and ZPL versions. (c, d) The same experiments, but using a smaller problem size to compare against the 2D ZPL shard-based implementation.

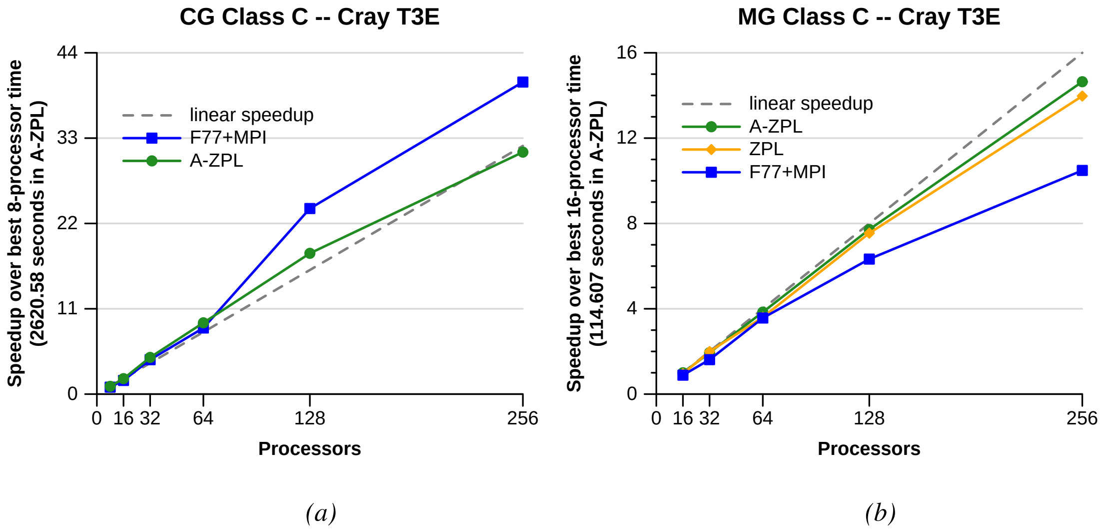


Figure 6.11: Performance for Sparse CG and MG Benchmarks. These graphs show speedup curves for class C of the NAS CG and MG benchmarks on the Cray T3E. Note that there is not enough memory to run these problem sizes on small processor sets. Thus, the speedups for each graph are computed using the fastest execution time on the smallest number of processors (indicated in the y-axis label).

sidering the overheads of iterating over A-ZPL's general sparse region structure as compared to iteration over a compact array representing the diagonal values. Once the compact implementations are removed from consideration (Figures 6.10b and d), the sparse implementations prove to scale nicely, tapering off once the number of values per processor drops below 1,000. On the larger problem size, the version that uses a pentadiagonal region to represent all arrays can be seen to outperform the version with two sparse regions due to the fact that its m-loops only need to iterate over a single sparse region rather than two of them.

For the CG benchmark (Figure 6.11a), the A-ZPL implementation outperforms the F77+MPI version for the smaller processor sets ($p \leq 64$), but then loses ground as the number of processors increases. Its speed advantage on smaller numbers of processors indicates that its implementation and loops are competitive with the NAS benchmark's hand-coded

CSR format. In addition, its use of the SHMEM library to implement communication results in lower overhead than the MPI calls of the NAS version.

The A-ZPL implementation loses ground on the larger processor sets once again due to the overheads associated with the remap operator's current implementation. Since remap is used in this benchmark to perform a simple vector transpose using the `Index`*i* arrays, it is expected that an implementation that breaks its m-loops out of the libraries will help close the gap for the larger processor sets. For the time being, we are pleased to note that A-ZPL's speedup is near-linear for all processor sets.

For the MG benchmark (Figure 6.11b), the sparse A-ZPL implementation results in slight but measurable improvements over the dense ZPL implementation. This may be somewhat surprising, given the drastic reduction in memory accesses for the sparse A-ZPL *resid* stencil. The reason that the improvement is not greater is due to the T3E's hardware support for aggressive data prefetching in the presence of regular memory accesses. The references to the input array in the dense implementation benefit greatly from this prefetching, minimizing the advantages of a sparse representation. For this platform, the greater benefit to users comes from the benchmark's 30% reduction in memory, allowing larger problem sizes to be run on smaller numbers of processors.

6.7.5 Summary

To summarize, the experiments in this section validate the sparse region concept. Sparse regions were shown to support the representation of sparse algorithms in A-ZPL and to preserve its dense syntax. Each benchmark tends to be as succinct as equivalent dense ZPL codes and hand-coded sparse sequential C codes. Sparse regions also represent a great improvement in clarity and conciseness as compared to parallel codes written in F77+MPI. All of the benchmarks indicate that programming using sparse regions results in memory requirements that are far smaller than an equivalent dense representation, and which compete with a hand-coded sparse format like CSR. Performance results indicate that A-ZPL's implementation of sparse m-loops and array accesses results in programs that outperform the equivalent dense codes, and perform competitively with hand-coded CSR formats. For all of these reasons, sparse regions constitute a useful concept in the array programmer's toolbox.

6.8 Related Work

The past few decades have seen a considerable amount of research and development in the area of sparse computation. The vast majority of this work has involved the development of libraries that support sparse matrix computations [SM95, THHS99, MS96, Kro98, JP95]. Each of these systems exports a set of highly-tuned sparse matrix operations to the user via a custom library interface. These interfaces are subject to the standard tradeoffs: they can be small and extremely special-purpose or they can be general but "wide." In contrast, sparse regions are a language-based solution. A-ZPL provides users with a small set of general operators that can express sparse array codes, of which sparse matrices are but a subset. In doing so, the standard tradeoffs are made: A-ZPL is unlikely to outperform a specific isolated library operation, but its generality allows users to solve a larger set of problems. Furthermore, the compiler's ability to comprehend the program's specification permits it to perform optimizations that span consecutive operations.

Although other parallel languages such as NESL and HPF allow sparse matrix computations [BCH$^+$94, FJY98], they typically require users to implement their own sparsity structures by hand as in the motivating Fortran matrix-vector multiplication example. This results in the same barriers to clarity and performance seen there. To aid with the performance problem, Ujaldon *et al.* have proposed extensions to HPF that provide language support for declaring sparse matrices using a variety of storage schemes [UZCZ97]. This approach solves the problem of communicating to the compiler that a sparse matrix is being used. However, it still requires users to refer to the underlying sparse matrix representation explicitly rather than allowing them to use a traditional dense array syntax as sparse regions do.

One approach that addresses this lack is a compiler-based technique by Bik and Wijshoff [BW93, Bik96] in which dense matrix programs are automatically transformed into an equivalent sparse program. Their compiler automatically selects an appropriate sparse format depending on the placement of non-zeroes in an array. More recently, the Bernoulli compiler group has developed a technique that allows users to specify sparse matrix operations using traditional dense syntax. In their approach, the compiler uses a form of *generic programming* to implement the code using a programmer-specified sparse matrix implementation [AMPS00, MPSK00]. Sparse regions share the goals of both of these projects, but take a different approach by using an array-based syntax and having the compiler automatically generate a sparse array representation optimized according to the program's operations rather than its structure. In doing so, sparse regions are not expected to compete with matrix representations that are highly specialized to a particular application, but they should support a broader class of sparse codes at a high level. Sparse regions also support the representation of higher-dimensional sparse arrays (rather than simply 2D matrices).

One other language-based approach that deserves mention is Matlab, which supports seamless interactions between sparse and dense matrices [GMS92]. A-ZPL's sparse region philosophy is very much like Matlab's, since it seeks to express sparse computation using a high-level array-based syntax. However, the application contexts are quite different in that Matlab is interpreted, sequential, and matrix-oriented whereas A-ZPL is compiled, parallel, and array-based.

6.9 Discussion

6.9.1 Alternate Sparse Formats

One of the advantages of using regions to encode sparsity is that the sparse implementation used at runtime is not exposed to the user in any way. This gives the compiler the ability to use any sparse format to implement sparse regions. The format described in this chapter was developed for its generality and flexibility, but for specific applications, different formats are likely to work better. It is therefore attractive to imagine that the A-ZPL compiler could use a variety of different sparse formats to implement sparse regions based on characteristics detected by the compiler or specified by the user.

The principal challenge to such an approach would be to make the runtime libraries understand all of the possible region formats. To avoid specializing the routines for each format, region descriptors could be amended to include function pointers that implement the operations required by the runtime such as iteration over the region's indices or the values required by an @-reference's communication. Such a scheme could even be designed so that users could plug in their own sparse representations as in the Bernoulli compiler [AMPS00]. This path has not been pursued at this point, but it represents an interesting possibility that should be considered in the future.

6.9.2 Support for Dynamic Sparsity

While the current region support for sparse computations forms an excellent starting point, there are still features missing from A-ZPL that any robust system for sparse computation should contain. One of the most important of these is the dynamic specification of sparsity patterns.

The benchmarks studied in this chapter use sparsity patterns that are fixed and unchanging for the program's duration. This makes them rather simplistic. A great number of real-world sparse applications use index sets that change over time as the data values migrate or fill in. Such applications require the ability to dynamically specify and modify sparsity patterns.

Advanced applications are not the only programs that require dynamic sparsity patterns. Although the sparsity patterns in this chapter's benchmarks are static, some of them cannot be efficiently expressed using A-ZPL's static region definitions. For example, although CG's sparsity patterns can be stored in a file and read during the region's declaration, reading the class C sparsity pattern using a large processor set results in a huge I/O bottleneck. The severity of this bottleneck makes reading the file untenable for benchmarking, due to

Listing 6.13: Proposed Support for Dynamic Sparsity Patterns Using Region Types

```
var RS: region = [1..m, 1..n] where ?;

...

RS := [ , ] where (A+B) < epsilon;

-- or

RS.spspat := (A+B) < epsilon;
```

the waste of computational resources. As a second example, the sparse tridiagonal regions given in this chapter require iteration over the full $n \times n$ base region to define the tridiagonal elements that evaluate as "true." This iteration is prohibitively expensive for the larger problem sizes used in this chapter's experiments. Both of these applications had to be rewritten to define their sparsity patterns dynamically in order to initialize large problem sizes in a reasonable amount of time.

The issue of specifying dynamic sparsity patterns is related to ZPL's current lack of support for dynamically naming and altering regions as addressed in Section 2.18.2. In particular, users need some way to assign a region's sparsity pattern within the context of their program's execution. Supporting assignments to regions as for any other type would help with the dynamic specification of sparsity patterns. Listing 6.13 shows an example.

Such support would still be insufficient, however, since it continues to require iteration over the base index space to evaluate the new sparsity pattern. For example, specifying a tridiagonal region using this syntax would yield no improvement over a static declaration. An ideal solution would permit a sparse index set to be specified imperatively, allowing the computational overhead to be proportional to the sparsity pattern itself.

One such mechanism would allow the user to add indices into a region's index set one at a time as they are computed. For example, Listing 6.14 shows some pseudo-code to specify the sparse tridiagonal region dynamically. Although the syntax is somewhat ugly, Listing 6.14: Proposed Support for Dynamic Index Specification

```
var Tri: region = [1..n, 1..n] where ?;
...
start_modifying_pattern(Tri);

add_index(Tri,1,1);         -- add first row
add_index(Tri,1,2);

for i := 2 to n-1 do        -- add interior rows
  add_index(Tri,i,i-1);
  add_index(Tri,i,i);
  add_index(Tri,i,i+1);
end;

add_index(Tri,n,n-1);       -- add last row
add_index(Tri,n,n);

stop_modifying_pattern(Tri);
```

the idea is sound in that it allows the user to specify the tridiagonal indices in $\Theta(n)$ time. It also makes the period in which changes are being made clear to the compiler, to prevent the runtime libraries from modifying the sparse representation with each new index (which could result in $O(nnz^2)$ insertion sort-like behavior). Once the modification to the sparsity pattern has ended, the sparse representation would be updated and all arrays declared over the region would automatically be reallocated.

The A-ZPL CG and tridiagonal benchmarks studied in this chapter use precisely this type of scheme to specify their sparsity patterns. Since the appropriate syntax does not yet exist, they simply use ZPL's **`extern`** keyword to call into the runtime routines that are used to define static sparsity patterns for traditional region initializers. These benchmarks demonstrate that such an approach is feasible, even though an appropriate syntax has not yet been adopted.

Challenges to implementing dynamic sparsity effectively still abound. It is likely that as support for specifying sparsity patterns becomes richer, additional semantics for specifying

Listing 6.15: Supporting Union and Intersection on Sparse Regions

```
region R = [1..n, 1..n];
       RS1 = R where ...;
       RS2 = R where ...;
       RSInter = R where ?;
       RSUnion = R where ?;
...
RSInter := [ , ] where (RS1 & RS2);
RSUnion := [ , ] where (RS1 | RS2);
```

indices will become increasingly desirable. As with any language feature, the key will be to develop mechanisms that are useful, general, and able to be implemented efficiently.

6.9.3 Sparse Regions as Boolean Arrays

Any sparse region can be interpreted as a boolean array by treating its defining indices as "true" values and all other indices within its bounding box as "false." This admits another way of specifying sparsity patterns, by supporting computation on sparse regions using traditional ZPL array operators. For example, Listing 6.15 shows the specification of sparsity patterns that are the union and intersection of other sparse regions.

More advanced computations would support the specification of more complex sparsity patterns such as the ones resulting from sparse matrix multiplication. Listing 6.16 shows a sparse implementation of SUMMA matrix multiplication which first runs the algorithm on the arguments' sparsity patterns to determine the pattern of their product, and then on the matrices themselves.

This ability to operate on sparse regions as though they were boolean arrays seems crucial in order to support completely general sparse index set specifications. The implementation of these operations comes virtually for free, given that the operators are already supported for sparse arrays—in general, the only change required is to replace the boolean cases which choose between the IRV and an actual value with a boolean value to indicate whether or not the index exists.

Listing 6.16: A Proposed Sparse SUMMA Implementation

```
region RA = [1..m, 1..n] where ...;
       RB = [1..n, 1..o] where ...;
       RC = [1..m, 1..o] where ?;

var A: [RA] double;
    B: [RB] double;
    C: [RC] double;

...

for i := 1 to n do
  RC |= (>>[ , i] RA) & (>>[i, ] RB);
end;

[RC] for i := 1 to n do
       C += (>>[ , i] A) * (>>[i, ] B);
     end;
```

6.9.4 Why Explicit Sparsity?

Two fundamental and related questions emerge from these issues of specifying sparse index sets. Namely: "Should the user have to specify a region's sparsity pattern explicitly?" and "Should the user even have to specify whether a region is sparse or dense?"

It is easiest to answer the second question first. The main problem with having the compiler or runtime automatically determine whether a region or array should be sparse is the cost of making the wrong decision. Representing a dense array using a sparse format or a sparse array using a dense format would be entirely detrimental to a program's performance in terms of both memory and time. Furthermore, automatic detection of a region or array's sparsity would require runtime overheads that performance-minded programmers would prefer to avoid. For most applications, it seems reasonable to assume that the programmer will know whether their problems contain sufficient sparsity to warrant special treatment. For all of these reasons, the approach of giving the programmer a high-level means of specifying a region's sparsity seems appropriate.

The other question is motivated somewhat by the hassle of manipulating sparsity patterns as well as the existence of systems like Matlab that support automatic sparsity management [GMS92]. Couldn't the user simply tag an array as being sparse and then let the compiler manage its sparsity pattern? The primary reason that this model works for Matlab and not ZPL is due to the difference between sparse matrices and arrays. As mathematical objects, operations on sparse matrices have a well-defined outcome that can be managed and implemented by a runtime system. In contrast, arrays do not have such semantics. If two sparse arrays are added together, is the user's intention to calculate sums for the union of the sparsity patterns, or the intersection? Different applications will expect different behaviors from such operations, and sparse regions provide a succinct method for describing the intended result.

A second reason not to support automatic sparsity patterns in A-ZPL is motivated by the decoupling of arrays and regions. An implementation of automatic sparsity would probe an array's values for the frequency of the IRV—the region itself would offer no clues. But what should be done if multiple arrays that are declared over a generic sparse region are found to have different sparsity patterns? Use a different sparse format for each? Use the union of the patterns? Again, A-ZPL's guiding principle is that users will know best, and that by providing them with an appropriate set of high-level operators, sparse regions can remain a clean enough abstraction that specifying their indices is not crippling to the programmer (especially when compared to hand-coded sparse parallel programs like NAS CG).

6.9.5 Specifying the IRV

One of the minor concepts missing from the current implementation of sparse regions is a syntax for specifying a sparse array's IRV. Though the concept of an arbitrary IRV is completely supported in the compiler-generated code idioms and runtime libraries, A-ZPL currently has no syntax for a user-specified IRV. The current implementation zeroes out the IRV's memory for each sparse array at initialization time.

Listing 6.17: One Proposal for Setting a Sparse Array's IRV

```
region RS = [1..n, 1..n] where ...;
var A: [RS] integer;

[RS] A := (Index1-1)*n + Index2;   -- assign A's elements
[!RS] A := -1;                     -- set A's IRV to -1
```

One solution would be to have some means of specifying the complement of the sparse region to refer to the IRV, as shown in Listing 6.17. The main drawback to this approach is that regions tend to signify the computational complexity of a statement. Modifying the IRV costs $\Theta(1)$ time rather than $\Theta(n^2 - nnz)$, as implied by this region. The second problem is that the assignment seems to write to indices for which `A` was not declared, which is typically illegal.

Another approach would be to specify the IRV as part of `A`'s declaration. The chief drawback with this is that some programs may require an array's IRV to change over time.

A third approach would be to give the user a peek into the array's implementation by allowing reference to a "virtual field" within the array descriptor `A` as follows:

```
A.IRV := -1;
```

This would be somewhat strange given that no other qualities of an array can currently be changed in this manner. However, this syntax could also be a useful mechanism for changing a region's bounds, stride, sparsity pattern, *etc.* It could also be used to specify an array's I/O functions, which are currently defined using function calls.

To summarize, though this is a conceptually simple operation and trivial to implement, a satisfactory syntax has not yet been developed.

6.9.6 Hierarchical Sparse Applications

As alluded to in the previous chapter, one logical extension of this work would be to combine sparse regions with hierarchical regions to support more interesting hierarchical algorithms like the fast multipole method or adaptive mesh refinement [ED95, LM90]. The concepts of hierarchical index sets and sparse index sets are sufficiently orthogonal that once some of the missing features of each construct are filled in, such applications should be fairly straightforward to write in A-ZPL. Extrapolating from the contributions of this dissertation, this forms the most obvious goal to work towards in the immediate future. Assuming that the benefits of regions described so far continue to hold, such an approach should result in an extremely clean, efficient, performance-aware implementation of some fairly complex and important algorithms.

6.10 Summary

This chapter has described a means of providing sparse computation using regions. The fact that regions decouple indices from array references makes them an ideal indexing mechanism for sparse computations. In particular, region-based array computations look the same whether they are implemented using sparse arrays or dense arrays. The result is that any dense code can be converted into a sparse equivalent simply by specifying the desired sparsity patterns. The fact that the implementation of sparse regions is not exposed to the user allows the compiler to use unique data structures such as the general but optimizable format described in this chapter. Other representations could also be used by the compiler given sufficient runtime support.

This chapter serves as a nice stopping point due to the fact that it takes the region concept, which already had numerous benefits in the dense context, and finds a few additional and unexpected benefits in it.

Chapter 7

CONCLUSIONS

As desribed in the first chapter, this dissertation is motivated by the software problem of parallel computing. This problem seeks to find a means of expressing computation such that parallel computers can be used effectively. A good solution to the software problem will express the computation in a clear manner, result in good parallel performance, and will be portable across the diverse parallel architectures available today. ZPL is a programming language that meets all of these requirements. As demonstrated by the experiments of Chapters 2, 5, and 6, ZPL expresses parallel computations more clearly and concisely than other parallel programming languages, typically matching the size and complexity of a sequential C implementation. These experiments also demonstrated that ZPL exhibits excellent performance and portability when compared with other parallel and sequential programming langues.

The region is the primary reason for much of ZPL's success. Regions serve to factor the size, shape, and base indices of an array computation into a compact, nameable, user-specified concept, eliminating much of the redundancy and tedium of traditional array indexing and slicing. Regions promote code reuse and support a global view of parallel programming that is crucial for the clear expression of parallel algorithms. Regions have a descriptor-based runtime implementation that supports the ability to generate efficient parallel loops and array accesses. Region descriptors also aid in the effort to design portable runtime libraries like the Ironman interface, by providing a compact means of describing a distributed index set to a library routine.

A good solution to the parallel computing software problem should also support the ability to reason about a program's concurrency, data distribution, communication requirements, and load balance. ZPL supports these features due to the parallel interpretation of regions as a distributed index set. The distribution of regions across processor grids defines the distribution of array data, as well as the concurrency and load balance of array-based computation. Distributing interacting regions in a grid-aligned manner makes communication requirements and paradigms apparent in ZPL programs due to the use of array operators to specify nontrivial array interactions. These parallel attributes of regions result in ZPL's unique ability to make parallel overheads apparent in a program's syntax without sacrificing its global view of the computation.

Finally, a good solution to the parallel computing software problem should support general styles of computation. Currently, ZPL falls short in this regard. Though any computation can be expressed in ZPL, many parallel computations are impossible to write without sacrificing performance or clarity. The support for parameterized regions and sparse regions described in chapters 5 and 6 represent significant improvements to the basic region's generality. However, in spite of the success of these extensions, many computations remain out of ZPL's reach. It is expected that A-ZPL's support for task parallelism, pipelined parallelism, abstract data structures, improved data placement, and load balancing will help address most of these lacks. It is also expected that concepts similar in spirit to the region will allow such codes to be written in a clean, concise manner that continues to perform well. The following section presents some ideas for these improvements and also summarizes other region-related work that should be done in the future.

7.1 Future Work

7.1.1 Irregular Data Structures

For this discussion, assume that an "irregular data structure" is a pointer-based data structure such as a linked list, binary tree, octree, or general graph. Though some of these structures have array-based implementations, thinking of them in their graph-based form is often more intuitive. Given ZPL's success with array-based stencil computations like the

Jacobi iteration and MG, it is logical to extend these victories to the graph-based realm to support similar algorithms like finite differencing.

A key insight for supporting graph-based structures using a high-level concept like the region is to realize that traditional arrays also have a graph-based interpretation. In particular, any array can be thought of as a graph in which each index is represented by a node linked to its two nearest neighbors in each dimension. Using this interpretation, ZPL's operators can be rephrased in terms of the graph's structure. For example, an expression like `A@[2,1]` would refer to taking two steps along A's south link, followed by one step along the resulting node's east link. Similarly, a partial reduction of an array dimension might imply that all values should be combined along the links in that dimension, subject to some bounding nodes.

Extending this analogy, it is attractive to consider support for graph-based programming using a region-like concept to describe subsets of nodes in more general graphs. Programmers would specify a node structure containing data values and a multidimensional array of links. The language would support variations on ZPL's region and array operators to specify computations on subsets of the graph's nodes and traverse their links.

Such an approach would undoubtedly raise many challenges in defining its semantics. For instance, the graph-based interpretation of ZPL's @ operator is not sufficient since link traversals in graphs are rarely commutative. For example, accessing the left child of a binary tree node's right child will result in a different node than accessing the right child of its left child. Furthermore, the parallel implementation of such a graph-based language will contain challenges such as distributing graph nodes between processors using graph partitioning algorithms [Cha98] and implementing communication efficiently to avoid using all-to-all communications for every remote link traversal.

In spite of these challenges, the use of a region-like concept to represent a graph's vertices seems like an attractive one for parallel computing, given the size and complexity of these problems.

Listing 7.1: Proposed Task-Parallel Implementation of Quicksort

```
procedure Quicksort(tlo, thi: integer;        /* task bounds */
                    lo, hi: integer;          /* array bounds */
                    var X: [] double);        /* the array */
var pvtloc: integer;      /* the location of the pivot */
    tsplit: integer;      /* where to split the tasks */
begin
  /* don't sort if there's only one element */
  if (lo != hi) then

    if (tlo = thi) then
      /* use a local sort if there's only one task */
      Localsort(lo, hi, X);

    else

      /* otherwise, use all the tasks to partition the array */
      {tlo..thi} [lo..hi] pvtloc := Partition(lo, hi, X);

      /* compute the location of the pivot in task space */
      tsplit := p*(pvtloc-lo)/(hi-lo)

      /* make the recursive calls using a subset of the tasks */
      {tlo..tsplit}   Quicksort(tlo, tsplit, lo, pvtloc, X);
      {tsplit+1..thi} Quicksort(tsplit+1, thi, pvtloc+1, hi, X);
  end;
end;
```

7.1.2 Task Parallelism

One idea for supporting task parallelism in A-ZPL is to provide the user with a region-like concept for describing processor grid space rather than index space. In such an approach, the user could specify that a subset of processors should work on one task while other processor groups work on other tasks.

As a simple example, consider a divide-and-conquer algorithm like Quicksort running on a 1-dimensional processor grid. Listing 7.1 gives an extremely rough expression of such an algorithm using a region-like expression in curly braces to refer to task space. The routine takes the bounds of both the task space and array space involved in the sorting as parameters. It uses all tasks to partition the array and then computes the approximate placement of the pivot in task space, `tsplit`. Next, it specifies that tasks `tlo`...`tsplit` should implement the first recursive call while processors (`tsplit+1`)...`thi` implement the second recursive call. Though this example is crude, it illustrates how task space might be described using a region-like structure.

Ideally, programmers should be able to map task spaces to the processor grid either directly, assigning a task per processor, or conceptually, allowing each processor to implement multiple tasks. Furthermore, high-level concepts equivalent to the `Index`*`i`* arrays should be supported to allow tasks to take actions based on their global location, or to query the size of the task space without explicitly using variables as in this example.

This is just a germ of an idea. Many other challenges exist, including specifying communication channels between tasks, or organizing tasks using a more graph-based structure. However, it seems that some of the benefits gained by using regions to specify array computations might be applied to task space to implement task-based parallelism.

7.1.3 Grid Dimensions

Grid dimensions are a relatively new concept in A-ZPL, and they are relatively unexplored as a result. Their ability to expose a program's local view without completely abandoning the global-view framework makes them a powerful feature for advanced programmers.

As one example, the dynamic initialization of the CG benchmark in Chapter 6 uses grid dimensions to store each processor's local sparse array values while the sparse array's indices were being generated. This required a bucketing algorithm much like the one in Section 3.10.5. Without grid dimensions, the program would either have had to store the entire sparse array explicitly on each processor, or to recompute it from scratch after the region had been specified. Although this initialization step is a fairly artificial computation that NAS CG uses to generate a random data set, it represents an algorithm that is difficult to write without grid dimensions in ZPL without sacrificing efficiency or scalability.

Future work should continue to explore the applications of grid dimensions and their use in exposing aspects of the ZPL runtime to the user in a controlled fashion. For example, their use might form a means of providing the user with a back door for hand-implementing optimizations like array contraction as desired in Section 3.12.4.

7.1.4 Regions as Values

As indicated in the discussion sections of Chapters 2, 5, and 6, it is high time for regions and directions to be made true values in ZPL. Without doing so, it becomes very difficult to create regions whose properties might change over time. Furthermore, it makes parameterization of regions an embarrassing wart in the language rather than a natural synthesis of pre-existing ZPL concepts.

There will be two educational challenges to supporting regions as values. The first is convincing users to declare regions as constants or configuration variables rather than normal variables whenever possible to continue to support the optimization of compiler-generated loops and array accesses. The second will be to make sure that users understand the overheads involved in changing a region's bounds or sparsity pattern, and to make these overheads clear in the code.

In terms of the implementation, making regions into mutable concepts will potentially introduce more uncertainty into the compiler which will have to be combatted using semantic restrictions and compile-time analysis. For example, will users be able to change normal dimensions into singleton dimensions? Singleton dimensions into flood dimensions? Sparse regions into dense regions? The more flexibility the user is given, the harder it will be to generate optimal code. It seems likely that a good balance can be reached to give users the power they need without making the compiler's analysis tasks prohibitively challenging.

7.1.5 Alternative Data Distributions

While simple blocked distributions have served ZPL well, support for more interesting data distributions will become increasingly crucial as more advanced applications and sparse applications are written in the language. ZPL's global view does an excellent job of insulating user code from such features, and the language ought to support a high-level means of specifying data distributions as proposed in Section 3.12.2. The primary challenges will come in implementing other distributions, though Section 4.7 indicates that they can fit into the current implementation without any vast changes.

7.1.6 Advanced Sparse Computation

Chapter 6 showed that sparse regions are an extremely powerful and clean means of specifying sparse computation. Continued evolution of A-ZPL's sparse regions will be necessary to support applications with dynamic or hierarchical sparsity as discussed in Section 6.9.

7.1.7 The Remap Operator

As demonstrated by this dissertation's benchmarks that use the remap operator, its current implementation leaves much to be desired. The first improvement should be to pull its implementing m-loops out of the runtime libraries and into the compiler-generated code to ensure that efficient m-loops and array accesses are generated when accessing its map and source arrays. In addition, the common case of using `Index`*i* expressions as the map arrays should be optimized to avoid any unnecessary communication of the map array values. In such cases, each processor should be able to compute both its source and destination values, eliminating an all-to-all communication and potentially optimizing the actual transfer of data values. Moreover, communication optimizations such as those used for the @ operator should be implemented to reuse map array communication schedules whenever possible, and to hide the latency of the all-to-all communications. While it is unlikely that the remap operator will ever be as cheap as ZPL's other array operators, all of these optimizations will help it cease to be the bottleneck for larger processor sets that it currently is.

7.1.8 Shared Memory Ironman Implementations

A shared memory implementation of the Ironman interface is long overdue. The ZPL compiler generates code that has the ability to run very well on shared memory machines, but then uses MPI to transfer data between their processors, resulting in unnecessary overheads. A direct shared memory implementation should be written to maximize performance on these platforms, using the SHMEM implementation as a template. Furthermore, a hybrid version of Ironman should be written for clusters of SMPs such that each processor uses shared memory to communicate with other processors on its node and MPI to communicate with remote processors. The Ironman libraries provide a perfect interface for implementing such hybrid communication mechanisms with low overhead, but without a shared memory implementation, they cannot be realized.

7.2 Summary

As in any project that is large and worthwhile, it is inevitable that this discussion of regions leaves many possibile paths unexplored. In spite of these unknown possibilities, it is clear that regions represent a fundamental strength to parallel array-based languages like ZPL. Regions impact every aspect of the language from its syntax to its semantics to its parallelism to its implementation. And all of these facets have made regions enjoyable for me to work with throughout my years as a graduate student. However, the most important property of regions for me is their elegance. In most instances that inelegance has crept into the design and implementation of regions, time has proven that choice to be wrong. It is the fact that the clean choice typically proves to be the right choice for regions—the beauty of their elegance—that has made them continually rewarding for me.

BIBLIOGRAPHY

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

[AMPS00] Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghill. A framework for sparse matrix code synthesis from high-level specifications. In *Proceedings of the 2000 ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC2000)*, November 2000.

[ASS95] G. Agrawal, A. Sussman, and J. Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):747–754, July 1995.

[Bac98] John Backus. The history of Fortran I, II, and III. *IEEE Annals of the History of Computing*, 20(4):68–78, Oct–Dec 1998.

[BB00] S. Booth and E. Bourao. Single sided MPI implementations for SUN MPI. In *Proceedings of the 2000 ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC2000)*, November 2000.

[BBB$^+$94] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Fredrickson, T. Lasinski, R. Schreiber, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[BBC$^+$94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, second edition, 1994.

[BBCR98] T. Brandes, F. Bregier, M. C. Counilh, and J. Roman. Contribution to better handling of irregular problems in HPF2. In *Proceedings of Europar '98*, volume 1470 of *LNCS*. Springer-Verlag, 1998.

[BCC$^+$97] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, July 1997.

[BCF$^+$93] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, and Sanjay Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results. In *Proceedings of Supercomputing '93*, November 1993.

[BCG+95] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[BCH+94] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[BCSvS01] S. B. Baden, P. Colella, D. Shalit, and B. van Straalen. Abstract KeLP. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*, March 2001.

[BDG+91] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A user's guide to PVM (parallel virtual machine). Technical Report ORNL/TM 11826, Oak Ridge National Laboratories, July 1991.

[BF95] Eric Barszcz and Paul Fredrickson. NAS MG 2.3 release notes, December 1995.

[BF99] Scott B. Baden and Stephen J. Fink. The data mover: A machine-independent abstraction for managing customized data motion. In *Proceedings of Languages and Compilers for Parallel Computing 1999*, pages 333–349, 1999.

[BFS01] Scott B. Baden, Richard Frost, and Daniel Shalit. KeLP user guide version 1.4. Technical report, Department of Computer Science and Engineering, University of California, San Diego, February 2001.

[BH86] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, December 1986.

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

[Bik96] A. J. C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, Leiden University, 1996.

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

[Ble95] Guy E. Blelloch. NESL: A nested data-parallel language (3.1). Technical Report CMU-CS-95-170, Carnegie Mellon School of Computer Science, September 1995.

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

[Bra77] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31:333–390, 1977.

[Brä95] Thomas Bräunl. Parallaxis-III: A language for structured data-parallel programming. In *Proceedings of the IEEE First International Conference on Algorithms and Architectures for Parallel Processing*, pages 43–52. IEEE, April 1995.

[BW93] A. J. C. Bik and H. A. G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the International Conference on Supercomputing*, pages 416–424, 1993.

[CCL+96] Bradford 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 David Sehr, Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 481–500. Springer-Verlag, 1996.

[CCL+98] 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 Third International Workshop on High-Level Programming Models and Supportive Environments*, pages 50–61. IEEE Computer Society Press, March 1998.

[CCS95] Kenneth Cameron, Lyndon J. Clarke, and A. Gordon Smith. CRI/EPCC MPI for CRAY T3D. In *1st European Cray T3D Workshop*, September 1995.

[CCS97] Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. A compiler abstraction for machine independent communication generation. In *Languages and Compilers for Parallel Computing*, pages 261–76. Springer-Verlag, August 1997.

[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.

[CDG+93] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[CDG+95] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Steve Luna, Thorsten von Eicken, and Katherine Yelick. *Introduction to Split-C (version 1.0)*. Computer Science Division — EECS, University of California, Berkeley, Berkeley, CA 94720, April 1995.

[CDL+96] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder. Fast rendering of complex environments using a spatial hierarchy. In *Proceedings of the 22nd Annual Graphics Interface Conference*, pages 132–141, May 1996.

[CDS99] Bradford L. Chamberlain, Steven Deitz, and Lawrence Snyder. Parallel language support for multigrid algorithms. Technical Report UW-CSE 99-11-03, University of Washington, November 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 2000 ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC2000)*, November 2000.

[Cha91] Siddhartha Chatterjee. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors*. PhD thesis, Carnegie Mellon University, School of Computer Science, October 1991.

[Cha98] Bradford L. Chamberlain. Graph partitioning algorithms for distributing workloads of parallel computations. Technical Report UW-CSE-98-10-03, University of Washington, October 1998.

[Cho99] Sung-Eun Choi. *Machine Independent Communication Optimization*. PhD thesis, University of Washington, Department of Computer Science and Engineering, March 1999.

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

[CLLS99] Bradford L. Chamberlain, E Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: An abstraction for expressing array computation. In *ACM/SIGAPL International Conference on Array Programming Languages*, pages 41–49, August 1999.

[CLR92] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 1992.

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

[CLS99] 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 13th ACM International Conference on Supercomputing*, pages 311–318, June 1999.

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

[CS01] Bradford L. Chamberlain and Lawrence Snyder. Array language support for parallel sparse computation. In *Proceedings of the 2001 International Conference on Supercomputing*, pages 133–145. ACM SIGARCH, June 2001.

[DCS01] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the 2001 International Conference on Supercomputing*, pages 65–77. ACM SIGARCH, June 2001.

[DiN96] David C. DiNucci. Cooperative data sharing: A layered approach to an architecture-independent message-passing interface. In *Proceedings of the Second MPI Developer's Conference*, pages 58–65, July 1996.

[DiN97] David C. DiNucci. A simple and efficient process and communication abstraction for network operating systems. In *Proceedings of the Workshop on Communication and Architectural Support for Network-Based Computing*, volume 1199 of *LNCS*, pages 31–45. Springer-Verlag, February 1997.

[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 9^{th} International Conference on Supercomputing*, pages 365–374, 1995.

[Dut97] Swaroop Vasudeva Dutta. Compilation and run-time techniques for data-parallel programs. Master's thesis, Louisiana State University and Agricultural and Mechanical College, Department of Electrical and Computer Engineering, December 1997.

[ED95] M. A. Epton and B. Dembart. Multipole translation theory for the three-dimensional Laplace and Helmholtz equations. *SIAM Journal on Scientific Computing*, 16(4):865–97, July 1995.

[FHK$^+$90] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR 90079, Rice University, Center for Research on Parallel Computation, December 1990.

[FJY98] Michael Frumkin, Haoqiang Jin, and Jerry Yan. Implementation of NAS parallel benchmarks in High Performance Fortran. Technical Report NAS-98-009, Nasa Ames Research Center, Moffet Field, CA, September 1998.

[FKB98] S. J. Fink, S. R. Kohn, and S. B. Baden. Efficient run-time support for irregular block-structured applications. *Journal of Parallel and Distributed Computing*, 50(1–2):61–82, April–May 1998.

[Fru00] Michael A. Frumkin. *Personal communication*. NASA Ames Research Center, April 2000.

[FvDFH92] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practices*. Addison-Wesley Publishing Company, second edition, November 1992.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[Geh96] Wilhelm Gehrke. *Fortran 95 Language Guide*. Springer Verlag, October 1996.

[GKS99] Clemens Grelck, Dietmar Kreye, and Sven-Bodo Scholz. On code generation for multi-generator with-loops in SAC. In *Proceedings of the 11th International Workshop on Implementation of Functional Languages*, volume 1868 of *LNCS*, pages 77–94. Springer-Verlag, 1999.

[GL00] Samuel Z. Guyer and Calvin Lin. Broadway: A software architecture for scientific computing. In *IFIPS Working Group 2.5: Working Conference on Software Architectures for Scientific Computing Applications*, October 2000.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[GMS92] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIMAX*, 13(1):333–356, January 1992.

[GMS$^+$95] Manish Gupta, Sam Midkiff, Edith Schonberg, Ven Seshadri, David Shields, Ko-Yang Wang, Wai-Mee Ching, and Ton Ngo. An HPF compiler for the IBM SP2. In *Proceedings of Supercomputing '95*, December 1995.

[Gre98] Clemens Grelck. Shared memory multiprocessor support for SAC. In *Proceedings of the 10th International Workshop on Implementation of Functional Languages*, volume 1595 of *LNCS*, pages 38–54. Springer-Verlag, 1998.

[Gro01] William Gropp. MPI web page. http://www-unix.mcs.anl.gov/mpi/, (current October 24, 2001).

[Gul00] Maria L. Gullickson. Optimizing loops in an array-based programming language. Qualifying Exam, University of Washington, Department of Computer Science and Engineering, June 2000.

[HAA+96] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S. W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, December 1996.

[HC93] Paul N. Hilfinger and Phillip Colella. FIDIL reference manual. Technical Report UCB/CSD 93-759, University of California Berkeley, May 1993.

[Hig94] High Performance Fortran Forum. *High Performance Fortran Specification Version 1.1*, November 1994.

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

[Ive62] Kenneth E. Iverson. *A Programming Language*. Wiley, New York, 1962.

[Joi01] Joint Institute for Computational Science. Lecture notes in parallel computing for undergraduates. http://www-jics.cs.utk.edu/PCUE/, (current October 24, 2001).

[JP95] Mark T. Jones and Paul E. Plassmann. *BlockSolve95 Users Manual: Scalable Library Software for the Parallel Solution of Sparse Linear Systems*. Argonne National Laboratory, December 1995. (revised June 1997).

[KAI01] KAI Software. KAI web page. http://www.kai.com/, (current October 24, 2001).

[KeL01] KeLP Research Group. KeLP web page. http://www-cse.ucsd.edu/groups/hpcl/scg/kelp/, (current October 24, 2001).

[KK96] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing '96 Conference Proceedings*. ACM/IEEE, November 1996. (a more complete version is available at http://www-users.cs.umn.edu/˜karypis/metis/publications/main.html).

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[KLS94] Robert H. Kuhn, Bruce Leasure, and Sanjiv M. Shah. The KAP parallelizer for DEC Fortran and DEC C programs. *Digital Technical Journal*, 6(3), 1994.

[Koh95] S. R. Kohn. *A Parallel Software Infrastructure for Dynamic Block-Irregular Scientific Calculations*. PhD thesis, University of California at San Diego, 1995.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, June 1988.

[Kro98] A. R. Krommer. Parallel sparse matrix computations in the industrial strength PINEAPL library. In *Applied Parallel Computing: Proceedings of PARA'98*, volume 1541 of *LNCS*, pages 281–285. Springer-Verlag, 1998.

[Lew00] E Christopher Lewis. *Achieving Robust Performance in Parallel Programming Languages*. PhD thesis, University of Washington, Department of Computer Science and Engineering, March 2000.

[Lin92] Calvin Lin. *The Portability of Parallel Programs Across MIMD Computers*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1992.

[Lit95] Vassily Litvinov. Design of graph ZPL: Extensions to ZPL to handle irregular and dynamic data structures. Qualifying Exam, University of Washington, Department of Computer Science and Engineering, October 1995.

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

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

[LM90] R. Leveque and M. Merger. Adaptive mesh refinement for hyperbolic partial differential equations. In *Proceedings of the 3rd International Conference on Hyperbolic Problems*, Uppsala, Sweden, 1990.

[Mac87] Bruce J. MacLennan. *Principles of Programming Languages*. Holt, Rinehart and Winston, second edition, 1987.

[Mat93] Mathworks. *MATLAB User's Guide*, 1993.

[Mes94] Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4):169–416, 1994.

[Mes97] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[MPSK00] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *Proceedings of the International Conference on Supercomputing*, May 2000.

[MS96] K. J. Maschhoff and D. C. Sorensen. P_ARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures. In *Applied Parallel Computing: Proceedings of PARA'96*, pages 478–486. Springer-Verlag, 1996.

[NAG00] Numerical Algorithms Group. *Essential Introduction to the NAG Parallel Library (release 3)*, 2000.

[NES01] NESL Research Group. NESL web page. http://www.cs.cmu.edu/ scandal/nesl.html, (current October 24, 2001).

[Ngo97] Ton A. Ngo. *The Role of Performance Models in Parallel Programming and Languages*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1997.

[NR98] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, August 1998.

[NRK98] Robert W. Numrich, John Reid, and Kieun Kim. Writing a multigrid solver using Co-Array Fortran. In *Proceedings of the Fourth International Workshop on Applied Parallel Computing*, Umea, Sweden, June 1998.

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

[Ohi96] Ohio Supercomputer Center, The Ohio State University. *MPI Primer / Developing with LAM*, November 1996.

[Ope01] OpenMP Forum. OpenMP: A proposed standard API for shared memory programming. http://www.openmp.org/, (current October 24, 2001).

[Pie93] Paul Pierce. The NX message passing interface. *Parallel Computing*, 1993.

[Por99] Portland Group Technical Reporting Service. Personal communication, October 1999.

[PVM01] PVM development group. PVM web page. http://www.epm.ornl.gov/pvm/, (current October 24, 2001).

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

[Saa90] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[SAC01] SAC Research Group. SAC web page. http://www.informatik.uni-kiel.de/ sacbase/, (current October 24, 2001).

[Sch94] S. B. Scholz. Single assignment C — functional programming using imperative style. In *Proceedings of IFL '94*, Norwich, UK, 1994.

[Sch98a] S. B. Scholz. A case study: Effects of WITH-loop-folding on the NAS benchmark MG in SAC. In *Proceedings of IFL '98*, London, 1998. Springer-Verlag.

[Sch98b] Sven-Bodo Scholz. On defining application-specific high-level array operations by means of shape-invariant programming. In *Proceedings of the ACM-SIGAPL International Conference on Array Processing Languages*, pages 40–45, 1998.

[Sec78] Secretariat, Computer and Business Equipment Manufacturers Association, editor. *American National Standard Programming Language FORTRAN*. American National Standards Institute, Inc., April 1978.

[SH89] Luigi Semenzato and Paul Hilfinger. Arrays in FIDIL. In Robert Grossman, editor, *Symbolic Computation: Applications to Scientific Computing*, pages 155–169. SIAM, 1989.

[SLY89] Z. Shen, Z. Li, and P. C. Yew. An empirical study on array subscripts and data dependences. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, 1989.

[SM95] Y. Saad and A. Malevsky. PSPARSLIB: A portable library of distributed memory sparse iterative solvers. In V. E. Malyshkin et al., editor, *Proceedings of Parallel Computing Technologies (PaCT-95), 3rd international conference*, LNCS. Springer-Verlag, Sept. 1995.

[Sny86] Lawrence Snyder. Type architecture, shared memory and the corollary of modest potential. *Annual Review of Computer Science*, pages 289–317, 1986.

[Sny95] Lawrence Snyder. Experimental validation of models of parallel computation. In A. Hofmann and J. van Leeuwen, editors, *Lecture Notes in Computer Science, Special Volume 1000*, pages 78–100. Springer-Verlag, 1995.

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

[SSO$^+$95] T. Stricker, J. Subhlok, D. O'Hallaron, S. Hinrichsand, and T. Gross. Decoupling synchronization and data transfer in message passsing systems of parallel computers. In *9th International Conference on Supercomputing*, July 1995.

[SvdWWY97] William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. New implementations and results for the NAS parallel benchmarks 2. In *8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[TD97] S. Tucker Taft and Robert A. Duff, editors. *Ada 95 Reference Manual: Language and Standard Libraries*, volume 1246 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Ter99] Tera Computer Company. *Tera Programming Guide*, 1999. http://www.npaci.edu/MTA/tera-doc/pg/html/Preface.html (current October 24, 2001.

[THHS99] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid. *Official Aztec User's Guide: Version 2.1*. Sandia National Laboratories, December 1999.

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

[Tse93] Chau-Wen Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.

[UZCZ97] Manuel Ujaldon, Emilio L. Zapata, Barbara M. Chapman, and Hans P. Zima. Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(10), October 1997.

[vdG97] Robert van de Geijn. *Using PLAPACK — Parallel Linear Algebra Package*. The MIT Press, 1997.

[vdGW95] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. Technical Report TR-95-13, University of Texas, Austin, Texas, April 1995.

[vdW00] Rob van der Wijngaart. *Personal communication*. MRJ Technology Solutions, April 2000.

[vECGS96] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 1996 IEEE/ACM Supercomputing Conference*, 1996.

[vRDSP96] C. van Reeuwijk, W. Denissen, H. J. Sips, and E. M. Paalvast. An implementation framework for HPF distributed arrays on message passing computers. *IEEE Transactions on Parallel and Distributed Systems*, 7(9):897–914, September 1996.

[WA00] Greg Watson and David Abramson. Relative debugging for data-parallel programs: A ZPL case study. *IEEE Concurrency*, 8(4):42–52, October–December 2000.

[Wal01] Alan Wallcraft. CAF web page. http://www.co-array.org/, (current October 24, 2001).

[Wat00] Gregory R. Watson. *The Design and Implementation of a Parallel Relative Debugger*. PhD thesis, Monash University, School of Computer Science and Software Engineering, October 2000.

[Wea99] W. Derrick Weathersby. *Machine-Independent Compiler Optimizations for Collective Communication*. PhD thesis, University of Washington, Department of Computer Science and Engineering, August 1999.

[Wei99] Mark Allen Weiss. *Data Structures & Algorithm Analysis in C++*. Addison-Wesley, second edition, 1999.

[Wes92] Pieter Wesseling. *An Introduction to Multigrid Methods*. John Wiley and Sons, 1992.

[WGS00] A. J. Wagner, L. Giraud, and C. E. Scott. Simulation of a cusped bubble rising in a viscoelastic fluid with a new numerical method. *Computer Physics Communications*, 129(1–3):227–232, July 2000.

[Wir83] Nicholas Wirth. *Programming in Modula-2*. Springer-Verlag, NY, second edition, 1983.

[ZBC+92] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna fortran - a language specification version 1.1. Technical Report ACPC/TR 92-4, Austrian Center for Parallel Computation, March 1992.

[ZPL01] ZPL Research Group. ZPL web page. http://www.cs.washington.edu/research/zpl/, (current October 24, 2001).

Appendix A

C VERSIONS OF BENCHMARKS

This appendix contains the C implementations of the simple benchmarks introduced in Chapters 2 and 6. Each program uses preprocessor macros to implement the configuration variables from the ZPL versions. Arrays are implemented using static multidimensional arrays whenever possible.

Listing A.1: The Jacobi Iteration in C

```
#include <stdio.h>
#include <values.h>
#include "timer.h"

static double A[n+2][n+2];
static double B[n+2][n+2];

int main() {
  double delta;
  int iterations;
  int i;
  int j;
  double diff;
  double runtime;

  ResetTimer();

  iterations = 0;

  for (i=1; i<=n; i++) {
    A[i][n+1] = 0.5;
  }
  for (i=1; i<=n; i++) {
    A[i][0] = 0.75;
  }
  for (i=1; i<=n; i++) {
    A[0][i] = 0.0;
  }
  for (i=1; i<=n; i++) {
    A[n+1][i] = 0.5;
  }
  for (i=1; i<=n; i++) {
    for (j=1; j<=n; j++) {
      A[i][j] = 1.0;
    }
  }

  do {
    iterations++;
    for (i=1; i<=n; i++) {
      for (j=1; j<=n; j++) {
        B[i][j] = (A[i][j+1] + A[i][j-1] + A[i-1][j] + A[i+1][j])/4;
      }
    }
    delta = -DBL_MAX;
    for (i=1; i<=n; i++) {
      for (j=1; j<=n; j++) {
        diff = A[i][j] - B[i][j];
        diff = (diff >= 0) ? diff : -diff;
        delta = (diff > delta) ? diff : delta;
        A[i][j] = B[i][j];
      }
    }
  } while (delta >= epsilon);
  runtime = CheckTimer();
}
```

Listing A.2: Matrix-Vector Multiplication in C

```
#include <stdio.h>
#include <values.h>
#include "timer.h"

static double M[m][n];
static double V[n];
static double S[m];

void setM(double M[m][n]) {
  double step;
  int i;
  int j;

  step = 1.0/(m*n);
  for (i=0; i<m; i++) {
    for (j=0; j<n; j++) {
      M[i][j] = (i*n + (j+1))*step + 1.0;
    }
  }
}

void setV(double V[n]) {
  double step;
  int i;

  step = 1.0/n;
  for (i=0; i<n; i++) {
    V[i] = (i+1)*step + 1.0;
  }
}

int main() {
  int i;
  int j;
  int k;
  double runtime;

  setM(M);
  setV(V);

  ResetTimer();

  for (i=0; i<m; i++) {
    S[i] = 0;
    for (j=0; j<n; j++) {
      S[i] += M[i][j] * V[j];
    }
  }
  runtime = CheckTimer();
}
```

Listing A.3: Matrix Multiplication in C

```
#include <stdio.h>
#include "timer.h"

static double A[m][n];
static double B[n][p];
static double C[m][p];

void setA(double A[m][n]) {
  double step;
  int i;
  int j;

  step = 1.0/(m*n);
  for (i=0; i<m; i++) {
    for (j=0; j<n; j++) {
      A[i][j] = (i*n + (j+1))*step + 1.0;
    }
  }
}

void setB(double B[n][p]) {
  double step;
  int i;
  int j;

  step = 1.0/(n*p);
  for (i=0; i<n; i++) {
    for (j=0; j<p; j++) {
      B[i][j] = (i*p + (j+1))*step + 1.0;
    }
  }
}

int main() {
  int i;
  int j;
  int k;
  double runtime;

  setA(A);
  setB(B);

  ResetTimer();
  for (i=0; i<m; i++) {
    for (j=0; j<p; j++) {
      C[i][j] = 0;
    }
  }
  for (i=0; i<m; i++) {
    for (j=0; j<n; j++) {
      for (k=0; k<p; k++) {
        C[i][k] += A[i][j] * B[j][k];
      }
    }
  }
  runtime = CheckTimer();
}
```

Listing A.4: Tridiagonal Matrix Multiplication in C Using a Compact Representation

```
#include <stdio.h>
#include "timer.h"

static double A[n+2][3];
static double B[n+2][3];
static double C[n+2][5];

void setMat(double X[n+2][3]) {
  double step;
  int i, j;

  step = 1.0/((double)n*n);
  for (i=0; i<n+2; i++) {
    for (j=0; j<3; j++) {
      if (i >= 1 && (j+2) <= n) {
        if ((i + (j+2) - 3) >= 1 && (i + (j+2) - 3) <= n) {
          X[i][j] = ((i-1)*n + (i + (j+2) - 3))*step + 1.0;
        } else {
          X[i][j] = 0.0;
        }
      } else {
        X[i][j] = 0.0;
      }
    }
  }
}

int main() {
  int i, j;
  double runtime;

  setMat(A);
  setMat(B);

  ResetTimer();
  for (i=1; i<=n; i++) {
    for (j=0; j<5; j++) {
      switch (j) {
      case 0:
        C[i][j] = A[i][j] * B[i-1][j];
        break;
      case 1:
        C[i][j] = (A[i][j-1] * B[i-1][j]) + (A[i][j] * B[i][j-1]);
        break;
      case 2:
        C[i][j] = (A[i][j-2] * B[i-1][j]) + (A[i][j-1] * B[i][j-1]) +
                  (A[i][j] * B[i+1][j-2]);
        break;
      case 3:
        C[i][j] = (A[i][j-2] * B[i][j-1]) + (A[i][j-1] * B[i+1][j-2]);
        break;
      case 4:
        C[i][j] = (A[i][j-2] * B[i+1][j-2]);
        break;
      }
    }
  }
  runtime = CheckTimer();
}
```

Listing A.5: Sparse Matrix-Vector Multiplication in C Using a CSR Format

```
#include <stdio.h>
#include "timer.h"

static double* M = NULL;
static int* col = NULL;
static int row[m+1];
static double V[n], S[m];

/* routines for array allocation omitted for brevity */

void setM(void) {
  double step;
  int i, j, firstinrow, arrcap=0, nnz=0, lgn = lg(n);

  step = 1.0/(m*n);
  for (i=0; i<m; i++) {
    firstinrow = 1;
    for (j=0; j<n; j++) {
      if (((i+1) + (j+1))%lgn == 0) {
        if (nnz+1 > arrcap) {
          growarrays();
        }
        M[nnz] = (i*n + (j+1))*step + 1.0;
        col[nnz] = j;
        if (firstinrow) {
          row[i] = nnz;
          firstinrow = 0;
        }
        nnz++;
      }
    }
  }
  row[m] = nnz;
}

void setV(double V[n]) {
  double step;
  int i;

  step = 1.0/n;
  for (i=0; i<n; i++)
    V[i] = (i+1)*step + 1.0;
}

int main() {
  int i, j;
  double runtime;

  setM();  setV(V);

  ResetTimer();
  for (i=0; i<m; i++) {
    S[i] = 0;
    for (j=row[i]; j<row[i+1]; j++) {
      S[i] += M[j]*V[col[j]];
    }
  }
  runtime = CheckTimer();
}
```

Appendix B

EXPERIMENTAL PLATFORMS

Table B.1 summarizes the experimental platforms used in this dissertation. For each machine type, the location of the machine, number of processors, speed of the processors, and memory per processor are given. In addition, the memory model used by the machine is indicated for reference.

Table B.1: Experimental Platforms

Machine	*Location*	*Processors*	*Speed*	*Memory*	*Memory Model*
Linux cluster[1]	LANL	128 dual P-IIIs	500 MHz	0.938 GB	Distributed Memory
IBM SP	MHPCC	96 (6×16)	222 MHz	0.5 GB	Cluster of Shared Memory MPs
Cray T3E	ARSC	256	450 MHz	0.256 GB	Shared Address Space
Sun Enterprise 5500	UT Austin	14	400 MHz	0.143 GB	Shared Memory Multiprocessor
SGI Origin	LANL	2048 (16×128)	250 MHz	0.25 GB	Cluster of Shared Memory MPs

[1]The Linux cluster is linked by Myrinet as well as 100Mb ethernet. Both networks are represented in the experiments of Chapter 5.

Appendix C

COMPILER SPECIFICATIONS

Table C.1 summarizes the compilation process used in this dissertation's experiments. The compiler, version number, and command-line arguments used are given for each language and platform. In addition, the communication mechanism used at runtime is noted.

Table C.1: Compilers Used in Experiments

Language	*Compiler*	*Version*	*Command-line arguments*	*Comm. Mechanism*
Linux cluster compilers				
F77+MPI	GNU g77	0.5.25	–O3	MPI (MPICH 1.2)
HPF	PGI pghpf	3.1-2	–O3 –Mautopar –Moverlap=size:1 –Mmpi	MPI (MPICH 1.2)
ZPL	U. Wash. zc	1.16a		MPI (MPICH 1.2)
	GNU gcc	2.95.2	–O3	
IBM SP compilers				
F77+MPI	IBM mpxlf	2.4	–O3	MPI (IBM)
HPF	PGI pghpf	2.4-4	–O3 –Mautopar –Moverlap=size:1 –Mmpi	MPI (IBM)
ZPL	U. Wash. zc	1.16a		MPI (IBM)
	IBM mpcc	2.4	–O3	
Cray T3E compilers				
F77+MPI	Cray f90	3.3.0.0	–O3	MPI (Cray)
C	Cray cc	6.4.0.0	–O3	N/A
CAF	Cray f90	3.3.0.0	–O3 –X 1 –Z *nprocs*	E-registers
HPF	PGI pghpf	3.0-1	–O3 –Mautopar –Moverlap=size:1 –Msmp	SHMEM
ZPL	U. Wash. zc	1.17a		SHMEM, MPI (Cray)
(Ch. 3, 6)	Cray cc	6.4.0.0	–O3	
ZPL	U. Wash. zc	1.16a		SHMEM
(Ch. 5)	Cray cc	6.3.0.0	–O3	
Sun Enterprise 5500 compilers				
F77+MPI	WorkShop f90	5.0	–fast	MPI (Sun)
SAC	U. Kiel sac2c	0.90alpha	–O3 –v1 –noLIR –noTSI –maxlur 3 –mt	shared memory
	SUNW cc	5.0	–fast	
ZPL	U. Wash. zc	1.16a		MPI (Sun)
	SUNW cc	5.0	–fast	
SGI Origin compilers				
F77+MPI	MIPSpro 7 f90	7.3.1.1m	–O3	MPI (SGI)
HPF	PGI pghpf	2.4-4	–O3 –Mautopar –Moverlap=size:1 –Mmpi	MPI (SGI)
ZPL	U. Wash. zc	1.16a		MPI (SGI)
	MIPSpro cc	7.3.1.1m	–O3	

Appendix D

EXPERIMENTAL TIMINGS

This appendix contains the timings used to construct the speedup graphs in this dissertation's experiments. Each time represents the best observed time across a number of trials. All timings are given in seconds unless noted otherwise in the table's heading.

Table D.1 contains timings for the performance model experiments reported in Chapter 3. Tables D.2 and D.3 contain the best observed times for each configuration of the MG experiments of Chapter 5. Table D.4 indicates the best observed times for each configuration of the sparse experiments in Chapter 6.

Table D.1: Raw Timings for Performance Model Experiments

Cray T3E — Jacobi (n = 2560)

processors	1	2	4	8	16	32	64	128	256
C	34.16	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—
ZPL	36.57	18.24	8.22	3.100	2.03	1.03	0.52	0.27	0.14

Cray T3E — Matrix-Vector Multiplication (m = n = 2560) — in milliseconds

processors	1	2	4	8	16	32	64	128	256
C	131.00	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—
ZPL (1D)	—.—	4736.00	3346.00	1714.00	1328.00	691.00	585.00	335.00	389.00
ZPL (2D)	408.00	205.00	98.10	51.10	27.50	15.30	8.30	3.46	2.02

Cray T3E — Matrix Multiplication (m = n = o = 1280)

processors	1	2	4	8	16	32	64	128	256
C	7.18	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—
ZPL (Cannon)	—.—	162.19	82.29	41.46	21.29	11.03	5.95	3.36	2.14
ZPL (SUMMA)	126.86	63.66	31.65	16.03	8.21	4.28	2.37	1.32	0.69
ZPL (PSP)	—.—	76.93	38.97	20.36	10.80	5.55	3.28	1.94	1.16

Cray T3E — Tridiagonal Matrix Multiplication (n = 1,310,720) — in milliseconds

processors	1	2	4	8	16	32	64	128	256
C	355.00	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—
ZPL (Mask)	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—
ZPL (Shard)	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—
ZPL (Compact)	1255.00	771.00	386.00	196.00	98.60	50.10	25.80	11.20	6.06

Cray T3E — Tridiagonal Matrix Multiplication (n = 20,480) — in milliseconds

processors	1	2	4	8	16	32	64	128	256
C	5.26	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—
ZPL (Mask)	—.—	—.—	—.—	—.—	—.—	—.—	3993.00	2017.00	1049.00
ZPL (Shard)	—.—	—.—	—.—	—.—	—.—	—.—	406.00	228.00	160.00
ZPL (Compact)	19.40	10.40	5.66	3.22	1.91	1.08	0.73	0.62	0.58

Table D.2: Raw Timings for MG Experiments

Linux cluster (ethernet) — MG (Class B)

processors	1	2	4	8	16	32	64	128
F77+MPI	362.60	194.51	110.41	52.51	30.39	17.55	8.29	6.21
ZPL	309.43	175.12	97.83	55.14	35.24	21.30	11.73	11.88
HPF	—.—	896.02	964.65	618.68	362.18	223.62	146.13	115.47

Linux cluster (ethernet) — MG (Class C)

processors	1	2	4	8	16	32	64	128
F77+MPI	—.—	—.—	828.18	390.26	215.02	121.04	56.51	34.66
ZPL	—.—	—.—	725.29	384.99	214.52	117.45	62.82	44.49
HPF	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—

Linux cluster (myrinet) — MG (Class B)

processors	1	2	4	8	16	32	64	128
F77+MPI	359.33	181.84	99.13	44.02	23.59	13.21	5.82	3.34
ZPL	308.51	176.92	108.82	42.57	21.96	11.05	6.10	3.35
HPF	—.—	616.38	516.15	314.54	177.57	104.10	62.95	47.15

Linux cluster (myrinet) — MG (Class C)

processors	1	2	4	8	16	32	64	128
F77+MPI	—.—	—.—	775.83	356.45	189.27	104.95	45.93	24.99
ZPL	—.—	—.—	827.56	338.99	172.15	86.87	44.53	23.27
HPF	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—

IBM SP — MG (Class B)

processors	1	2	4	8	16	32	64	96
F77+MPI	—.—	83.48	48.86	23.19	12.55	7.83	3.80	—.—
ZPL	—.—	82.27	45.12	28.75	14.41	7.71	4.94	3.98
HPF	—.—	—.—	—.—	—.—	—.—	—.—	42.17	31.10

IBM SP — MG (Class C)

processors	1	2	4	8	16	32	64	96
F77+MPI	—.—	—.—	—.—	—.—	96.35	54.65	25.40	—.—
ZPL	—.—	—.—	—.—	—.—	104.01	52.08	29.96	21.95
HPF	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—

Table D.3: Raw Timings for MG Experiments (continued)

Cray T3E — MG (Class B)

processors	1	2	4	8	16	32	64	128	256
CAF	—.—	117.38	58.86	28.98	14.98	7.87	4.22	2.31	1.15
F77+MPI	—.—	127.22	68.94	31.47	17.52	10.49	4.77	2.92	1.96
ZPL	—.—	127.79	59.08	30.39	15.55	7.88	4.51	2.63	1.63
HPF	—.—	—.—	—.—	—.—	—.—	—.—	80.16	—.—	56.35

Cray T3E — MG (Class C)

processors	1	2	4	8	16	32	64	128	256
CAF	—.—	—.—	—.—	—.—	118.42	59.89	30.23	15.91	7.91
F77+MPI	—.—	—.—	—.—	—.—	128.59	70.89	32.14	18.12	10.92
ZPL	—.—	—.—	—.—	—.—	112.63	57.85	31.24	15.30	8.24
HPF	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—

SGI Origin — MG (Class B)

processors	1	2	4	8	16	32	64	128	256
F77+MPI	202.13	130.04	61.88	32.84	16.93	8.46	3.98	2.65	264.17
ZPL	244.88	147.27	72.37	39.06	19.37	9.66	5.46	3.10	310.15
HPF	556.13	488.93	341.50	183.94	94.67	52.80	29.33	24.96	—.—

SGI Origin — MG (Class C)

processors	1	2	4	8	16	32	64	128	256
F77+MPI	—.—	1534.95	853.55	380.58	135.18	64.36	33.62	22.51	340.35
ZPL	—.—	2004.99	1026.91	441.74	160.35	75.09	41.31	26.47	445.16
HPF	—.—	—.—	—.—	1949.62	1006.18	—.—	—.—	—.—	—.—

Sun Enterprise — MG (Class A)

processors	1	2	4	8	14
F77+MPI	73.49	39.36	19.87	11.34	—.—
ZPL	61.29	35.37	17.59	12.60	12.15
SAC	92.56	49.27	27.07	17.27	15.18

Sun Enterprise — MG (Class B)

processors	1	2	4	8	14
F77+MPI	341.77	172.33	103.78	59.13	—.—
ZPL	277.53	162.00	80.02	57.87	55.83
SAC	434.02	230.80	126.08	79.49	68.76

Table D.4: Raw Timings for Sparse Experiments

Cray T3E — Sparse Matrix-Vector Multiplication (m = n = 5120) — in milliseconds

processors	1	2	4	8	16	32	64	128	256
C	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—
C (CSR)	213.00	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—
ZPL (2D)	—.—	658.00	304.00	153.00	73.60	36.30	17.90	9.24	4.86
A-ZPL	357.00	179.00	80.00	40.00	19.80	10.20	5.31	2.92	1.82

Cray T3E — Sparse Tridiagonal Matrix Multiplication (n = 1,310,720) — in milliseconds

processors	1	2	4	8	16	32	64	128	256
C	355.00	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—
ZPL (Shard)	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—
ZPL (Compact)	—.—	585.00	292.00	147.00	72.80	36.70	18.50	8.24	4.28
A-ZPL	—.—	—.—	—.—	604.00	303.00	151.00	75.80	38.20	19.50
A-ZPL (All Pent)	—.—	—.—	1017.60	525.00	263.00	131.00	65.90	31.50	16.80

Cray T3E — Sparse Tridiagonal Matrix Multiplication (n = 20,480) — in milliseconds

processors	1	2	4	8	16	32	64	128	256
C	5.26	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—
ZPL (Shard)	—.—	—.—	—.—	—.—	—.—	—.—	406.00	228.00	160.00
ZPL (Compact)	14.90	8.14	4.30	2.17	1.27	0.80	0.53	0.42	0.35
A-ZPL	70.40	37.70	19.20	9.66	4.87	2.44	1.30	0.79	0.54
A-ZPL (All Pent)	66.50	30.30	16.90	7.71	3.95	2.25	1.24	0.77	0.55

Cray T3E — CG (Class C)

processors	1	2	4	8	16	32	64	128	256
F77+MPI	—.—	—.—	—.—	2922.03	1489.15	590.65	307.45	109.53	65.14
A-ZPL	—.—	—.—	—.—	2620.58	1314.66	553.95	285.05	144.40	84.01

Cray T3E — MG (Class C)

processors	1	2	4	8	16	32	64	128	256
F77+MPI	—.—	—.—	—.—	—.—	129.08	70.95	32.12	18.11	10.93
ZPL	—.—	—.—	—.—	—.—	120.09	57.46	31.80	15.19	8.20
A-ZPL	—.—	—.—	—.—	—.—	114.61	58.99	29.81	14.85	7.83

Appendix E

FORMAL PARALLEL ZPL DEFINITIONS

The contents of this appendix mirror the formal definitions of Chapter 3, but consider each array operator in a parallel context. Each definition is given in its 1-dimensional form for a processor q. Function $f()$ refers to the distribution function used to distribute the dimension.

These tables also include the definitions for grid dimensions. Index :: refers to the grid dimension index that conforms to all of a processor's indices just as $*$ refers to all indices for a flood dimension. The notations $X_{::_i}$ and Y_{*_i} are used to refer to processor i's implementing values of grid array X or flood array Y, respectively.

Table E.1: Parallel Definition of Array Writing

`[r]` A `:=` ... (where A is defined over $\mathbf{a}$)

	$\mathbf{a}$ is normal	$\mathbf{a}$ is singleton	$\mathbf{a}$ is flooded	$\mathbf{a}$ is grid
$\mathbf{r}$ is normal	Legal if $I(\mathbf{r}) \subseteq I(\mathbf{a})$. Writes $A_i, \forall i \in I(\mathbf{r})$ s.t. $f(i) = q$.	Illegal, since $\mathbf{r}$ describes multiple indices but $\mathbf{a}$ only describes one.	Illegal, to ensure that all implementing copies of A remain consistent.	Legal. Writes $A_{::_q}, \forall i \in I(\mathbf{r})$ s.t. $f(i) = q$.
$\mathbf{r}$ is singleton	Legal if $i \in I(\mathbf{a})$, where $I(\mathbf{r}) = \{i\}$. Writes A_i, if $f(i) = q$.	Legal if $I(\mathbf{r}) = I(\mathbf{a}) = \{i\}$. Writes A_i, if $f(i) = q$.	Illegal, since it would require a broadcast of the singleton value.	Legal. Writes $A_{::_q}$ if $f(i) = q$, where $I(\mathbf{r}) = \{i\}$.
$\mathbf{r}$ is flooded	Illegal, since $\mathbf{r}$ has only a single implementing index per processor.	Illegal, since $\mathbf{r}$ resides on multiple processors and $\mathbf{a}$ does not.	Legal. Writes A_{*_q}.	Legal. Writes $A_{::_q}$.
$\mathbf{r}$ is grid	Illegal, since $\mathbf{r}$ has only a single implementing index per processor.	Illegal, since $\mathbf{r}$ resides on multiple processors and $\mathbf{a}$ does not.	Illegal, to ensure $A_{*_i} = A_{*_j}$, $\forall i, j \in 0 \ldots (p-1)$.	Legal. Writes $A_{::_q}$.

Table E.2: Parallel Definition of Array Reading

`[r]` ... `:=` $\ldots A \ldots$ (where A is defined over $\mathbf{a}$)

	$\mathbf{a}$ is normal	$\mathbf{a}$ is singleton	$\mathbf{a}$ is flooded	$\mathbf{a}$ is grid
$\mathbf{r}$ is normal	Legal if $I(\mathbf{r}) \subseteq I(\mathbf{a})$. Reads $A_i, \forall i \in I(\mathbf{r})$ s.t. $f(i) = q$.	Illegal, since $\mathbf{r}$ describes multiple indices but $\mathbf{a}$ only describes one.	Legal. Reads $A_{*_q}, \forall i \in I(\mathbf{r})$ s.t. $f(i) = q$.	Legal. Reads $A_{::_q}, \forall i \in I(\mathbf{r})$ s.t. $f(i) = q$.
$\mathbf{r}$ is singleton	Legal if $i \in I(\mathbf{a})$, where $I(\mathbf{r}) = \{i\}$. Reads A_i, if $f(i) = q$.	Legal if $I(\mathbf{r}) = I(\mathbf{a}) = \{i\}$. Reads A_i, if $f(i) = q$.	Legal. Reads A_{*_q} if $f(i) = q$, where $I(\mathbf{r}) = \{i\}$.	Legal. Reads $A_{::_q}$ if $f(i) = q$, where $I(\mathbf{r}) = \{i\}$.
$\mathbf{r}$ is flooded	Illegal, since $\mathbf{r}$ has only a single implementing index per processor.	Illegal, since it would require a broadcast of the singleton value.	Legal. Reads A_{*_q}.	Illegal since it could read different values on different processors.
$\mathbf{r}$ is grid	Illegal, since $\mathbf{r}$ has only a single implementing index per processor.	Illegal, since $\mathbf{r}$ resides on multiple processors and $\mathbf{A}$ does not.	Legal. Reads A_{*_q}.	Legal. Reads $A_{::_q}$.

Table E.3: Parallel Definition of @ Operator

`[r]` $\ldots A$`@[`δ`]`$\ldots$ (where A is defined over $\mathbf{a}$)

	$\mathbf{a}$ is normal	$\mathbf{a}$ is singleton	$\mathbf{a}$ is flooded	$\mathbf{a}$ is grid
r is normal	Legal if $I(\mathbf{r}\ \mathtt{at}\ \delta) \subseteq I(\mathbf{a})$. Uses point-to-point comm. to get $A_{i+\delta}, \forall i \in I(\mathbf{r})$ s.t. $f(i) = q$, $f(i+\delta) \neq q$. Returns $A_{i+\delta}$ at i.	Illegal, since **r** describes multiple indices but **a** only describes one.	Legal. Returns A_{*_q} at $i, \forall i \in I(\mathbf{r})$ s.t. $f(i) = q$.	Currently undefined.
r is singleton	Legal if $i + \delta \in I(\mathbf{a})$, where $I(\mathbf{r}) = \{i\}$. Uses point-to-point comm. to get $A_{i+\delta}$ if $f(i) = q$, $f(i+\delta) \neq q$. Returns $A_{i+\delta}$.	Legal if $I(\mathbf{a}) = \{i + \delta\}$, where $I(\mathbf{r}) = \{i\}$. Uses point-to-point comm. to get $A_{i+\delta}$ if $f(i) = q$, $f(i+\delta) \neq q$. Returns $A_{i+\delta}$.	Legal. Returns A_{*_q} at i, if $f(i) = q$, where $I(\mathbf{r}) = \{i\}$	Currently undefined.
r is flooded	Illegal, since **r** has only a single implementing index per processor.	Illegal, since it would require a broadcast of the singleton value.	Legal. Returns A_{*_q} at $*$.	Illegal, since it could read different values on different processors.
r is grid	Illegal, since **r** has only a single implementing index per processor.	Illegal, since **r** resides on multiple processors and A does not.	Legal. Returns A_{*_q} at ::.	Currently undefined.

Table E.4: Parallel Definition of the Flood Operator

`[d] ...>>[s] A...`

	$\mathbf{s}$ is normal	$\mathbf{s}$ is singleton	$\mathbf{s}$ is flooded	$\mathbf{s}$ is grid
$\mathbf{d}$ is normal	Legal if $I(\mathbf{s}) = I(\mathbf{d})$. Returns A_i at $i, \forall i \in I(\mathbf{d})$ s.t. $f(i) = q$.	Legal. Broadcasts A_k to all other processors if $f(k) = q$, where $I(\mathbf{s}) = \{k\}$. Returns A_k at $i, \forall i \in I(\mathbf{d})$, s.t. $f(i) = q$.	Legal. Returns A_{*_q} at $i, \forall i \in I(\mathbf{d})$ s.t. $f(i) = q$.	Legal. Returns $A_{::_q}$ at $i, \forall i \in I(\mathbf{d})$ s.t. $f(i) = q$.
$\mathbf{d}$ is singleton	Illegal, since multiple values cannot be replicated to a single value.	Legal. Sends A_k to proc. r, if $f(k) = q$, where $I(\mathbf{s}) = \{k\}$, $I(\mathbf{d}) = \{i\}$, and $f(i) = r$. Returns A_k at i if $r = q$.	Legal. Returns A_{*_q} at i, if $f(i) = q$, where $I(\mathbf{d}) = \{i\}$.	Legal. Returns $A_{::_q}$ at i, if $f(i) = q$, where $I(\mathbf{d}) = \{i\}$.
$\mathbf{d}$ is flooded	Illegal, since multiple values cannot be replicated to a single conceptual value.	Legal. Broadcasts A_k to all other processors if $f(k) = q$, where $I(\mathbf{s}) = \{k\}$. Returns A_k at $*$.	Legal. Returns A_{*_q} at $*$.	Illegal, since it could read different values on different processors.
$\mathbf{d}$ is grid	Illegal, since multiple values cannot be replicated to a single value per processor.	Legal. Broadcasts A_k to all other processors if $f(k) = q$, where $I(\mathbf{s}) = \{k\}$. Returns A_k at ::.	Legal. Returns A_{*_q} at ::.	Legal. Returns $A_{::_q}$ at ::.

Table E.5: Parallel Definition of the (plus) Reduce Operator

`[d] ...+<<[s] A...`

	$\mathbf{s}$ is normal	$\mathbf{s}$ is singleton	$\mathbf{s}$ is flooded	$\mathbf{s}$ is grid
$\mathbf{d}$ is normal	Legal if $I(\mathbf{s}) = I(\mathbf{d})$. Returns A_i at i, $\forall i \in I(\mathbf{d})$ s.t. $f(i) = q$.	Illegal, since a single value cannot be reduced to multiple values.	Legal. Returns A_{*_q} at i, $\forall i \in I(\mathbf{d})$ s.t. $f(i) = q$.	Illegal, since multiple values cannot be reduced to multiple values.
$\mathbf{d}$ is singleton	Legal. Reduces $\sum_i A_i$ with all other processors, $\forall i \in I(\mathbf{s})$ s.t. $f(i) = q$, leaving result on proc. r, where $I(\mathbf{d}) = \{k\}$, $f(k) = r$. Returns sum of all contributions at k if $r = q$.	Legal. Sends A_i to proc. r, if $f(i) = q$, where $I(\mathbf{s}) = \{i\}$, $I(\mathbf{d}) = \{k\}$, and $f(k) = r$. Returns A_i at k if $r = q$.	Legal. Returns A_{*_q} at k, if $f(k) = q$, where $I(\mathbf{d}) = \{k\}$.	Legal. Reduces $A_{::_q}$ with all other processors, leaving result on proc. r, where $I(\mathbf{d}) = \{k\}$, $f(k) = r$. Returns sum of all contributions at k if $r = q$.
$\mathbf{d}$ is flooded	Legal. Reduces $\sum_i A_i$ with all other processors, $\forall i \in I(\mathbf{s})$ s.t. $f(i) = q$, broadcasting the result back. Returns sum of all contributions at $*$.	Legal. Broadcasts A_i to all processors, if $f(i) = q$. Returns A_i at $*$.	Legal. Returns A_{*_q} at $*$.	Legal. Reduces $A_{::_q}$ with all other processors, broadcasting the result back. Returns sum of all contributions at $*$.
$\mathbf{d}$ is grid	Legal. Reduces $\sum_i A_i$ with all other processors, $\forall i \in I(\mathbf{s})$ s.t. $f(i) = q$, broadcasting the result back. Returns sum of all contributions at $::$.	Legal. Broadcasts A_i to all processors, if $f(i) = q$. Returns A_i at $::$.	Legal. Returns A_{*_q} at $::$.	Legal. Returns $A_{::_q}$ at $::$.

Table E.6: Parallel Definition of the Remap Operator

`[r]` $\ldots A$`#[`M`]`$\ldots$ (where A is defined over $\mathbf{a}$)

	$\mathbf{a}$ is normal or singleton	$\mathbf{a}$ is flooded	$\mathbf{a}$ is grid
$\mathbf{r}$ is normal or singleton	Legal if $M_i \in I(\mathbf{a}), \forall i \in I(\mathbf{r})$. Uses all-to-all comm. to get $A_{M_i}, \forall i \in I(\mathbf{r})$ s.t. $f(i) = q$, $f(M_i) \neq q$. Returns A_{M_i} at i.	Legal. Returns A_{*_q} at $i, \forall i \in I(\mathbf{r})$.	Legal. Uses all-to-all comm. to get $A_{::r_i}, \forall i \in I(\mathbf{r})$ s.t. $f(i) = q$, $f(M_i) = r_i \neq q$. Returns $A_{::r_i}$ at i.
$\mathbf{r}$ is flooded	Legal if $M_{*_q} \in I(\mathbf{a})$. Broadcasts $A_{M_{*_q}}$ to all processors if $f(M_{*_q}) = q$. Returns $A_{M_{*_q}}$ at $*$.	Legal. Returns A_{*_q} at $*$.	Legal. Broadcasts $A_{::M_{*_q}}$ to all processors if $f(M_{*_q}) = q$. Returns $A_{::M_{*_q}}$ at $*$.
$\mathbf{r}$ is grid	Legal if $M_{::_i} \in I(\mathbf{a})$, $\forall i \in 0 \ldots (p-1)$. Uses all-to-all comm. to get $A_{M_{::_q}}$. Returns $A_{M_{::_q}}$ at ::.	Legal. Returns A_{*_q} at ::.	Legal if $M_{::_q} \in I(\mathbf{a})$. Uses all-to-all comm. to get $A_{M_{::_q}}$. Returns $A_{M_{::_q}}$ at ::.

VITA

Bradford L. Chamberlain prefers to be called "Brad" by his friends and dislikes writing about himself in the third person. He was born in Stanford, California in 1970 but grew up in Annapolis, Maryland and Northern Idaho. He made his way through Annapolis' public school system at a reasonable rate, graduating from Rolling Knolls Elementary in 1982, Wiley H. Bates Jr. High in 1985, and Annapolis Senior High School in 1988. His final years in high school involved several defining transformations including: (i) the realization that few things meant as much to him as friends, music, and words; (ii) successfully earning the rank of Eagle Scout within days of his 18th birthday, foreshadowing a lifelong habit of deadline-driven work habits; and (iii) doing a summertime stint on the quarter-pounder grill at his local McDonald's.

In the fall of 1988, he returned to Stanford University, graduating in 1992 with a Bachelor's of Science degree in Computer Science, with honors. His education was crucially enriched by two years of living in the Italian House, a quarter's study in Florence, a stint DJ-ing at KZSU 90.1 FM, and an occasional appearance on trombone with the Leland Stanford Junior (pause) University Marching Band.

To continue his study of Computer Science, he went directly from college to the University of Washington where he earned his Master of Science degree in 1995 and his Doctor of Philosophy in 2001. He never hesitates to describe these years as the happiest of his life to anyone who will listen.