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
`^`	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]`).

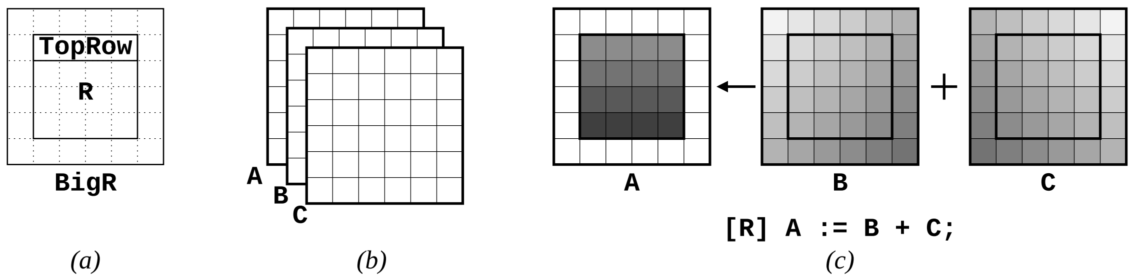


Figure 2.1: Using Regions and Arrays. (a) An illustration of the three regions declared in Section 2.3: R, TopRow, and BigR. (b) Three parallel integer arrays of size BigR—A, B, and C. (c) An example of how a statement's enclosing region scope restricts the range of its operators. Only indices within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

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

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

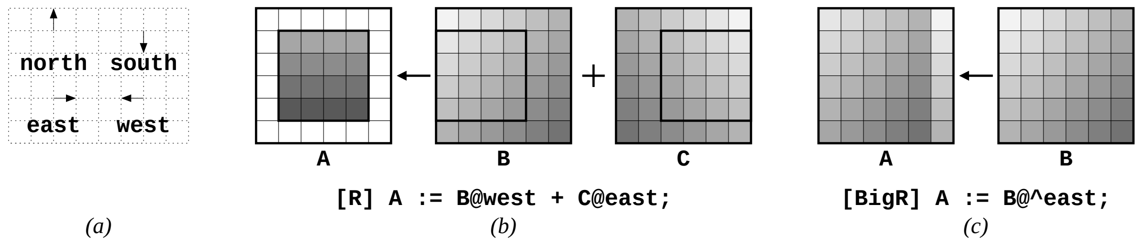


Figure 2.2: The @ Operator. (a) An illustration of the directions declared in Section 2.4.1. (b) A use of the @ operator to add shifted references of B and C, storing the result in region R of A. (c) A diagram illustrating the application of the wrap-@ operator to assign a cyclically-shifted version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

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

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.

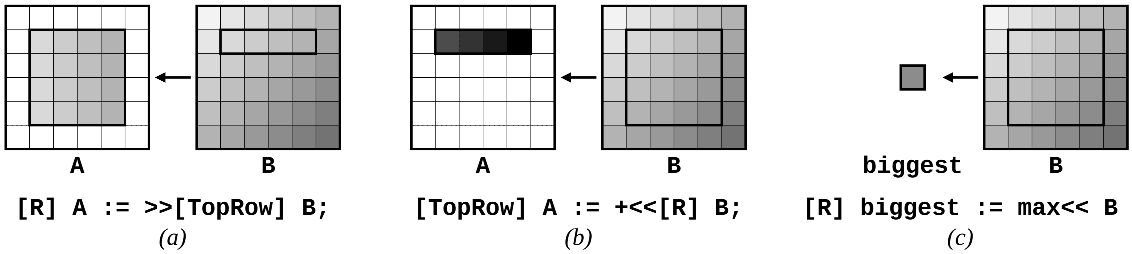


Figure 2.3: The Flood and Reduction Operators. (a) An illustration of the flood operator, causing the top row of B within R to be replicated across all rows of A within R. (b) An application of the sum reduction operator, which totals the values of B within each column of R and assigns the sum to the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

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

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

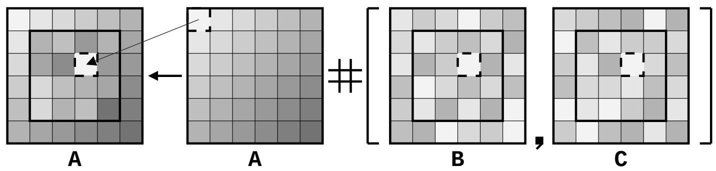


Figure 2.4: The Remap Operator. The B and C arrays serve as the map arrays for the remap of A in this assignment, thus they must contain values from 0 to 5 within region R (displayed here using varying levels of grey). As a specific example, consider the assignment to row 2, column 3, outlined with a 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:

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

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:

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

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:

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

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:

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

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, $\ldots$, l_d..h_d]

Such declarations correspond to the following formal region definition:

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

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:

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

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

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

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

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:

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

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.

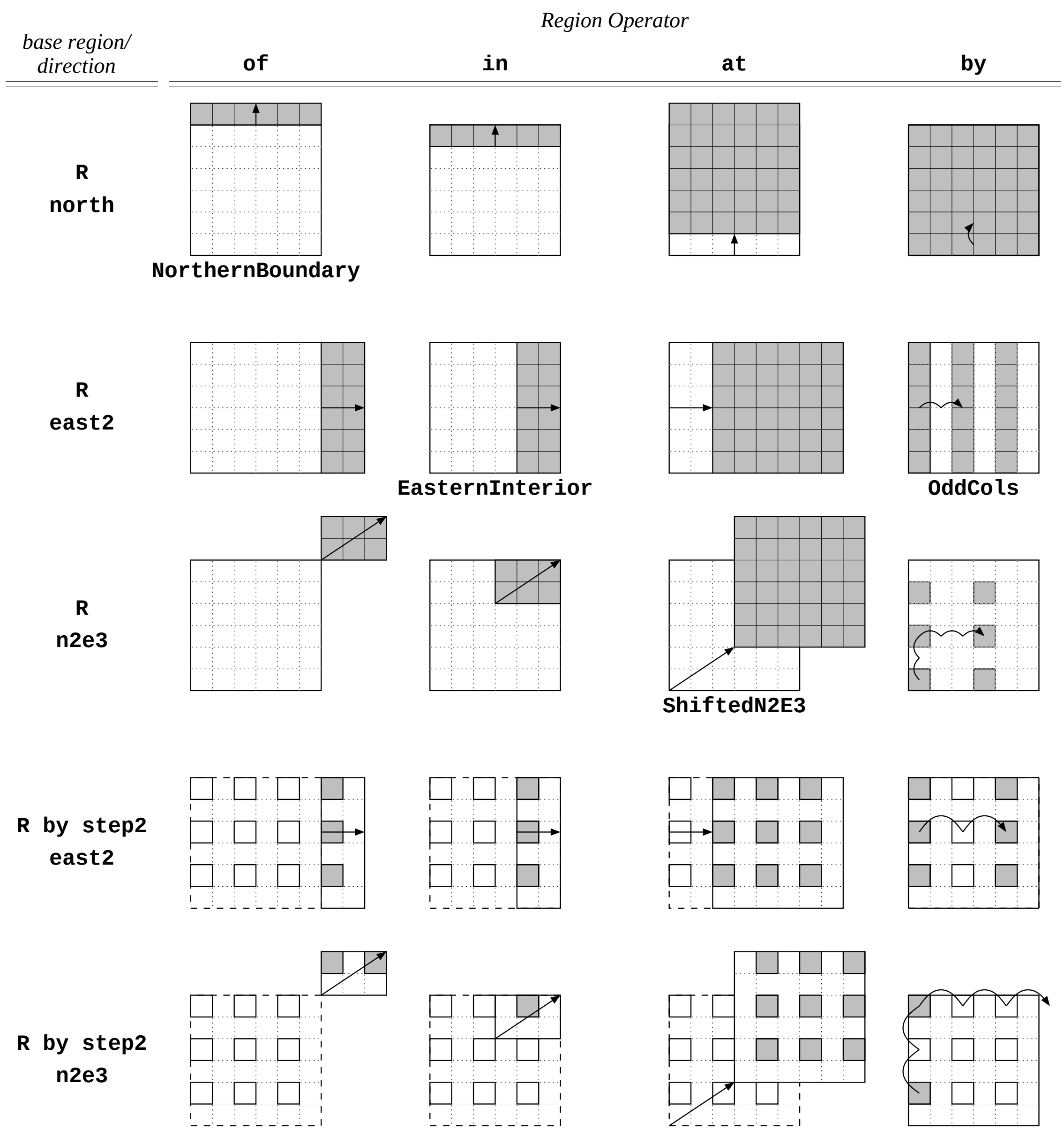


Figure 2.5: The Region Operators. This diagram illustrates the region operators applied using the declarations of Listing 2.6. Each column of pictures represents a single region operator (**of**, **in**, **at**, and **by**), while each row illustrates a different base region/direction pair. The indices 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.

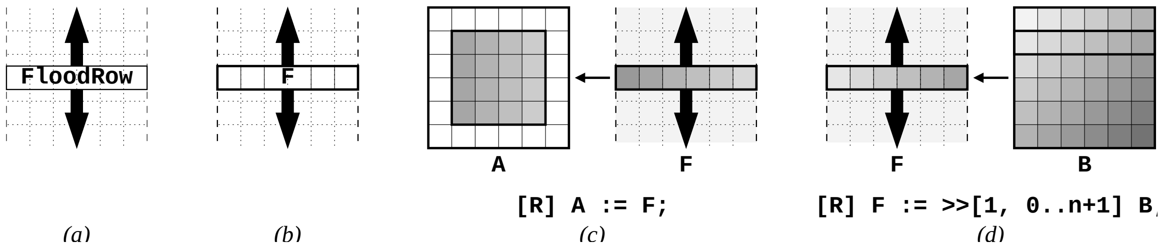


Figure 2.6: Flood Dimensions and Flood Arrays. (a) An illustration of a region whose first dimension is flooded. It represents a single row of values that are conformable to any row. (b) An array `F` declared using region `FloodRow`. (c) An assignment from `F` to `A` within region `R`. (d) An assignment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

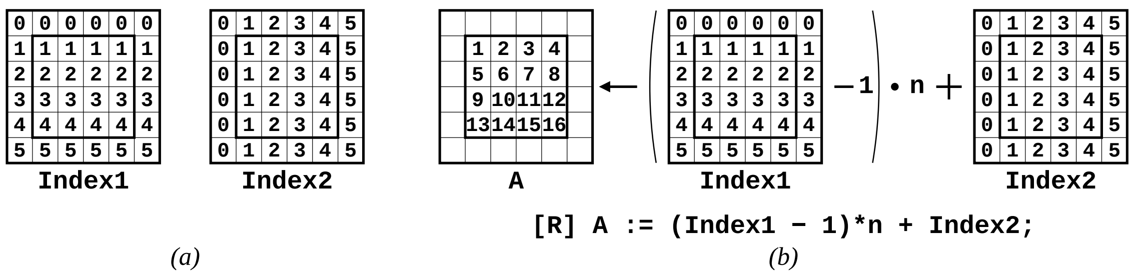


Figure 2.7: The Index Constants. (a) Pictures depicting **Index1** and **Index2**. BigR and R are indicated by the outlines. (b) A diagram showing the unique numbering of elements in R using **Index1** and **Index2**.

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];
```

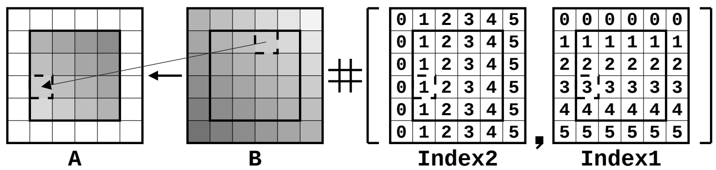


Figure 2.8: An Array Transpose. This diagram illustrates how the Index constants can be used to transpose arrays when used as the map arrays in a remap operation. By using column indices as the map array for B's rows and row indices for its columns, the elements of B are transposed during their assignment 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.

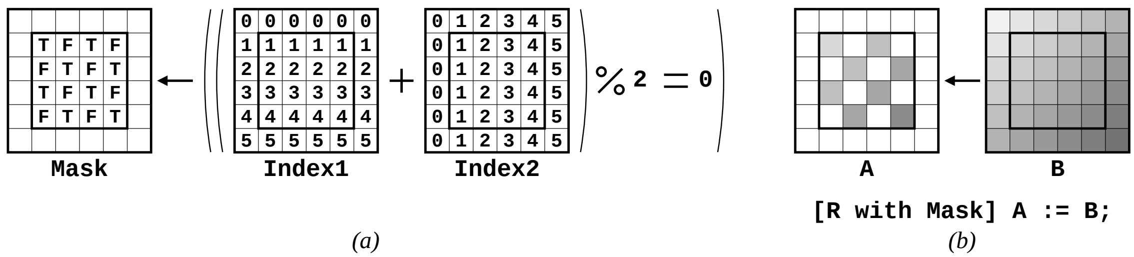


Figure 2.9: An Example of Using Masks. (a) The mask is assigned **`true`** for all locations in `R` where the sum of the row and column indices is even. (b) The mask is used to restrict the indices of `R` when assigning from `B` to `A`.

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:

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

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).

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].

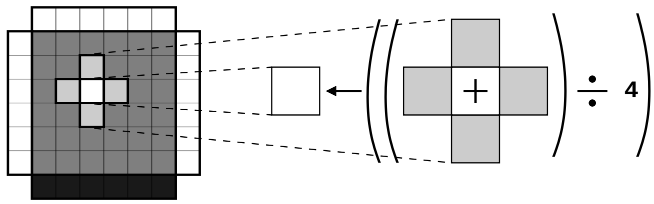


Figure 2.11: The Jacobi Iteration

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.

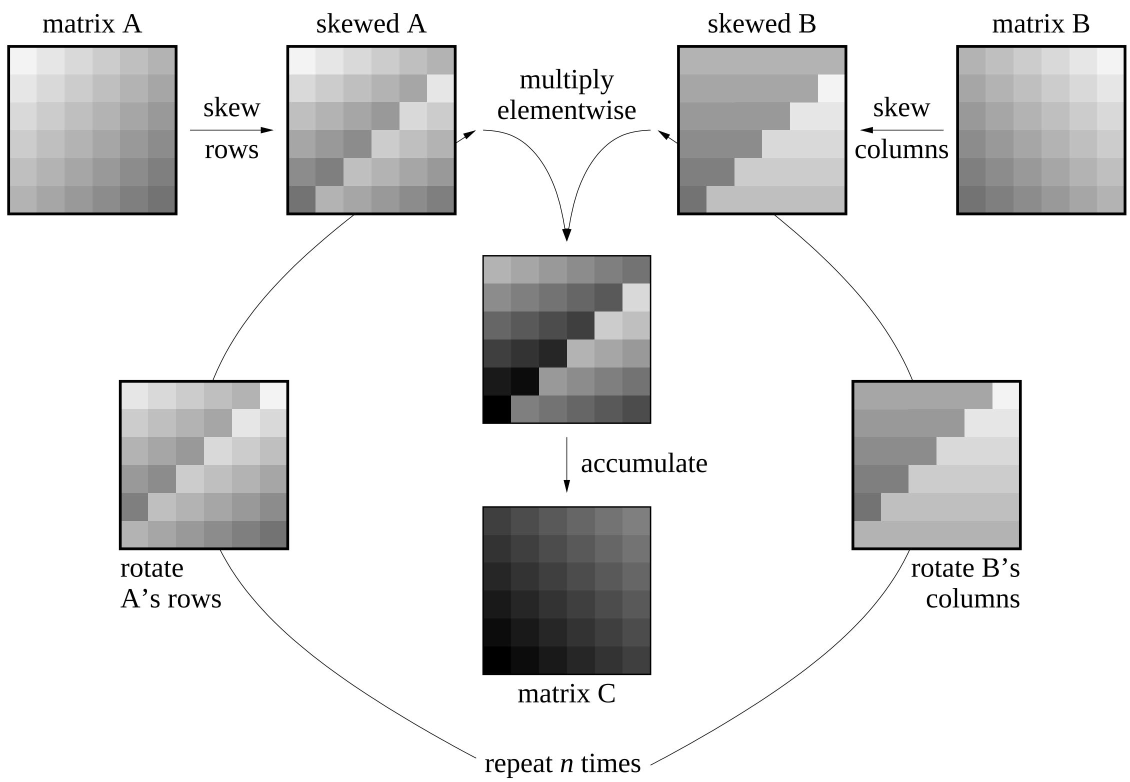


Figure 2.12: Cannon's Algorithm For Matrix Multiplication

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;
```

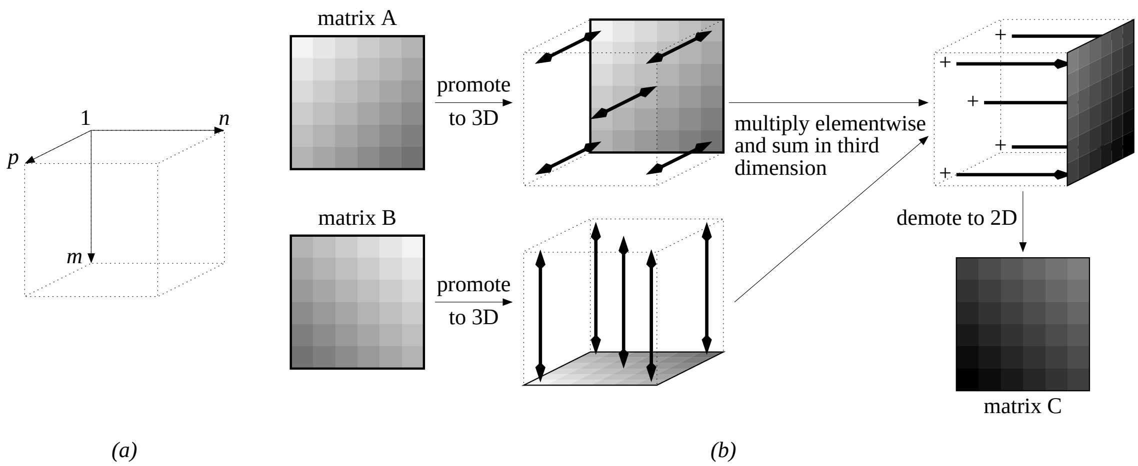


Figure 2.13: The PSP Algorithm For Matrix Multiplication

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

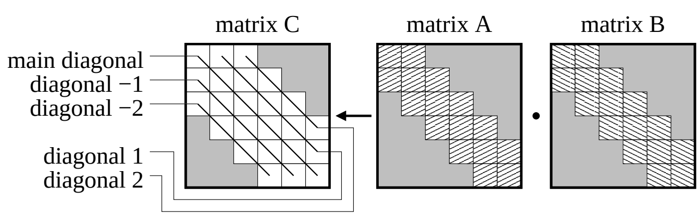
As a final application area, consider the multiplication of two tridiagonal matrices. Though any of the algorithms from the previous section can be used for this problem, the presence of so many zeroes allows more specialized techniques to be used. In particular, the resulting product will be a pentadiagonal 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

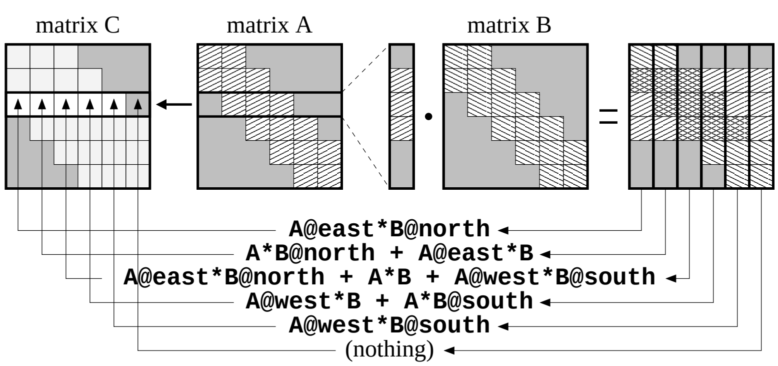
One approach for implementing tridiagonal matrix multiplication in ZPL is to use a mask to restrict computation to one of the five resulting diagonals at a time. An implementation of this approach is given in Listing 2.21.

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



(a)



(b)

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.

2.18 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.