Appears in "Proceedings of the Sixth International Workshop on Languages and Compilers for Parallel Computing," pages 96-114, 1993.

ZPL: An Array Sublanguage*

Calvin Lin
Lawrence Snyder
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

October 10, 1993

Abstract

The notion of isolating the "common case" is a well known computer science principle. This paper describes ZPL, a language that treats data parallelism as a common case of MIMD parallelism. This separation of concerns has many benefits. It allows us to define a clean and concise language for describing data parallel computations, and this in turn leads to efficient parallel execution. Our particular language also provides mechanisms for handling boundary conditions. We introduce the concepts, constructs and semantics of our new language, and give a simple example that contrasts ZPL with other data parallel languages.

1 Introduction

A variety of languages have been proposed that generally provide data parallel or array semantics, including C* [15], Fortran 90 [1], NESL [4], and HPF [8]. One characteristic of these languages is that data parallelism is the only model provided. This fact introduces a certain pressure to support a wide range of general purpose facilities. Though this has resulted in rich and interesting languages, it has not always served the goals of simplicity, cleanliness or efficiency. ZPL is a different kind of language.

ZPL is an array sublanguage of the Orca family of parallel programming languages [11, 12]. The Orca languages provide a general MIMD programming model [2, 6, 17], allowing programmers to write efficient, portable and scalable parallel programs. For many tasks, such as initializing an array A to all 0's, a full MIMD programming model is overly general. For other tasks, such as summing the elements of the array A, the best parallel solution is generally well understood, but somewhat machine sensitive. So, to simplify the programmer's job without limiting expressiveness, the Orca languages provide a set

*This research was supported in part by ONR Grant N00014-89-J-1368, ARPA Grant N00014-92-J-1824 and NSF Grant CDA-9211095 of facilities to perform data parallel operations, such as A := 0; and standard parallel abstractions, such as sum reduction (+\A). These convenience facilities, together with some control-flow operations, make up ZPL. Thus, ZPL is the "non-MIMD" component of the Orca languages. (Further discussion of the relationship between full Orca and ZPL is given in Section 4.)

ZPL can be thought of as a stand-alone parallel programming language. Though the Orca languages are explicitly parallel, the ZPL subset is implicitly parallel. The fact that ZPL is a sublanguage of a larger, more general language has been a critical advantage in its design because it has relieved the pressure to solve everything using the data parallel paradigm. That is, if the inclusion of some complex capability seemed to do violence to ZPL's clean semantics, there was always the option of leaving it out since that capability could be realized by the MIMD part of the language, albeit less conveniently.

Without the need to be fully general it has been possible to design a language that is simple and expressive, with seemingly few complicating characteristics. The simplicity will be evident in the description presented below. The expressiveness is justified by the fact that ZPL is sufficient to program the SIMPLE benchmark [11] and other standard data parallel applications, including the Ising model, the Floyd-Steinberg dithering model, Cypher *et al.*'s recursive connected component labeling algorithm [5], a median threshold filtering, and the game of Life. Examples of computations not sensibly programmed in ZPL would be the FFT (due to the butterfly-based data motion), LU Decomposition (due to the manipulation of the pivot), as well as more typically MIMD computations such as multipole methods for N-body simulation. Perhaps the most important aspect of ZPL, however, is that it provides a clean and uncomplicated context in which to study compilation techniques for data parallel and array languages. In this paper, we present the language, outline the semantics and explain the guiding principles behind this language.

The structure of the remainder of the paper is as follows. Section 2 uses the Jacobi example to highlight some differences between ZPL and other data parallel languages. Section 3 then presents ZPL's design goals and describes the major language features. We then discuss the relation of ZPL to the more general Orca languages. Finally, we briefly compare ZPL to other data parallel languages before concluding.

2 A Brief Comparison

Before presenting the features of ZPL it is useful to consider how different languages express data parallel computation. The goal is to point out the distinctive features of the languages in general terms rather than to give a tedious comparison of individual constructs.

As an illustrative example, consider the 4-point Jacobi computation on an array A[1..N, 1..N] in which each value in the array is to be replaced by the average of its four nearest neighbors. The boundary values are taken to be 0 except at the southern edge, where their value is 50.

In ZPL the initial values and main body of the Jacobi computation can be defined as shown below, where [R] is a user declared index set called a region; and `north`, `east`, `west` and south are programmer-defined vector constants called *directions*. For now we assume that the values of the above directions are [-1,0], [0,1], [0,-1], and [1,0], respectively [1].

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

        [north of R] A := 0;      /* Set boundary conditions */
        [east  of R] A := 0;
        [west  of R] A := 0;
        [south of R] A := 50;
        [R]          A := 100;
```

The kernel of the computation would be specified as follows:

```
        [R]   A := (A@north + A@east + A@west + A@south)/4;
```

The body of the above statement computes, for a given array element, the average of its four neighbors. Each neighbor is specified by the @ operator as an offset from a given element. For example, A@north refers to the element whose index is [-1,0] relative to the given element. The scope of this statement is defined by the region, [R]. That is, the body of the statement is applied (in parallel) to each element whose index is in [R].

If it is possible to distinguish the semantic approaches to data parallel computations by the terms "single point-based" and "array-based," then ZPL is clearly an array-based approach. The array is the basic unit of reference and the "at" operation replaces the more general array indexing operator with a disciplined method of accessing neighboring array values. Region descriptors are used to designate the scope over which data parallel operations will be applied. These regions can be specified where they are used and thus be different for every statement, but the assumption (borne out so far by our limited experience) is that scientific codes will generally sweep over a modest set of regions. This implies that there is not enormous diversity and that these regions have logical meaning worthy of names, i.e., naming provides useful mnemonic information. Accordingly, ZPL anticipates a mode of use that emphasizes setting up basic regions ahead of time as a means of avoiding tedious and error prone specification of a program's invariant information.

Jacobi in C*. C* employs a point-based approach. The Jacobi initialization and kernel statement are given below.

```
    shape [N][N] cell;

    with (cell) {
        /* Set boundary conditions */
        where ((!pcoord(0)) ||  (!pcoord(1)) || (pcoord(1) == (N-1))) {
```

[1]While the directions north, east, west and south can be defined by the programmer, they have the default values that are those used in this example.

```
            active = 0;
            oldA = newA = 0.0;
        } else where (pcoord(0) == (N-1)) {
            active = 0;
            oldA = newA = 50.0;
        } else {
            active = 1;
            newA = 100.0;
        }

        /* The Jacobi kernel */
        where (active) {
            oldA = newA;
            newA = ([.-1][.]oldA + [.+1][.]oldA + [.][.+1]oldA + [.][.-1]oldA)/4.0;
        }
    }
```

The above kernel is specified using indices relative to each data element. For example, `[.-1][.]oldA` is equivalent to `A@[-1,0]` in ZPL. Because the above code applies to every point in the data space, there is the need to define "active" and "inactive" points. The boundary values never change and are therefore marked as "inactive." Notice that the nested logic obfuscates the specification of the boundary conditions, yet it is desirable for efficiency reasons. By contrast, the ZPL region descriptors allow different code to be applied to different portions of the data space. For this reason, boundary condition definitions are cleaner and more concise in the ZPL program. In addition, the point-based view of C* forces the programmer to explicitly define "new" and "old" copies of the data; this is not necessary in ZPL. Finally, the access of neighbor values is similar to that in the ZPL program except that ZPL allows neighbor values to be represented as named vectors.

Jacobi in HPF. There are several ways to implement Jacobi in HPF. We present two variants below.

```
! Variant 1: use FORALL

REAL a(0:n+1,0:n+1)
!HPF$ DISTRIBUTE a(BLOCK,BLOCK)

! Initialize boundary conditions
FORALL (i=0,   j=1:n) a(i,j) =   0.0
FORALL (i=n+1, j=1:n) a(i,j) =  50.0
FORALL (i=1:n, j=0)   a(i,j) =   0.0
FORALL (i=1:n, j=n+1) a(i,j) =   0.0
FORALL (i=1:n, j=1:n) a(i,j) = 100.0
```

```
! The Jacobi kernel
FORALL (i=1:n, j=1:n)   a(i,j) = (a(i-1,j)+a(i+1,j)+a(i,j-1)+a(i,j+1))/4


! Variant 2: use array assignment

REAL a(0:n+1,0:n+1)
!HPF$ DISTRIBUTE a(BLOCK,BLOCK)
   ...
! Set boundary conditions
a(0,   1:n) =   0.0
a(n+1, 1:n) =  50.0
a(1:n, 0)   =   0.0
a(1:n, n+1) =   0.0
a(1:n, 1:n) = 100.0

! The Jacobi kernel
a(1:n, 1:n) = (a(0:n-1,1:n) + a(2:n+1,1:n) + a(1:n,0:n-1) + a(1:n,2:n+1))/4
```

The HPF programs are closer in spirit to ZPL than the C* program. The first variant uses explicit iterators and thus takes the point-based approach, while the latter uses an array-based approach. In both cases, the indices must be specified in both the boundary condition initialization and the kernel computation. This is an error prone situation for two reasons. First, the heavy use of explicit indices provides many opportunities to make a mistake, while ZPL's use of named directions is self-documenting. Second, the HPF approach duplicates the specification of indices in the boundary condition code and in the kernel code, which further increases the potential for error. The ZPL approach reduces repetition by defining direction vectors once and using these for both the boundary conditions and the kernel specification.

Jacobi in Fortran 90.

```
REAL a(n,n)
   ...
a = (EOSHIFT(a,-1,DIM=1,BOUNDARY= 0.0) +  &
     EOSHIFT(a, 1,DIM=1,BOUNDARY=50.0) +  &
     EOSHIFT(a,-1,DIM=2,BOUNDARY= 0.0) +  &
     EOSHIFT(a, 1,DIM=2,BOUNDARY= 0.0)) / 4
```

The Fortran 90 example[2] is closest in style to the ZPL program. Rather than deal with explicit indices, the EOSHIFT function is a higher level operation that shifts the

[2]Since HPF is a superset of Fortran 90, this Fortran 90 example can also be considered an HPF program.

array reference in a particular direction. Besides being more verbose than the ZPL "at" operator, EOSHIFT separates the dimension from the magnitude and direction of the shift, while ZPL's direction vectors convey all of this information in a single entity. Thus, for example, when using the EOSHIFT function it's easy to imagine confusing a "-1" with a "1" or "DIM=1" with "DIM=2." Such errors are still possible in ZPL but the use of named vectors reduces their likelihood since "east" and "west" are more syntactically distinct than "1" and "-1." As a performance issue, notice that because each call to EOSHIFT can specify a different boundary value as a parameter, each invocation of EOSHIFT must include a test to determine whether the data points reside on a process boundary.

Jacobi in NESL. NESL [4] is an applicative language that is based on vector operations such as scan and reduce. NESL supports nested parallelism and distinguishes itself from most other data parallel languages by its functional nature. Below is NESL code that applies to both regular and irregular meshes. Each Jacobi iteration is a matrix-vector product, where a sparse matrix is represented as a nested sequence. Most of the code is used to set up the initial conditions.

```
function sparse_MxV(mat,vect) =
   {sum({val * vect[i] : (i,val) in row}): row in mat} $

% Each Jabobi iteration is just a matrix vector product. %
% This will repeat it n times. %

function Jacobi_Iterate(Mat,vect,n) =
  if (n == 0) then vect
  else Jacobi_Iterate(Mat,sparse_MxV(Mat,vect),n-1) $

% The following two functions are mesh-specific.        %
% Each is called once to initialize the mesh and vector %

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

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

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

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

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

% invoke it %
function test(n, steps) =
let matrix = make_2d_n_by_n_mesh(n);
    vector = make_initial_vector(n);
in Jacobi_Iterate(matrix,vector,steps) $
```

The main contributor to the NESL code's length is the manipulation of 1D vectors to implement 2D abstractions. For example, it is cumbersome to modify the interior of the 2D problem state separately from the borders. In fact, for convenience, the above code initializes the interior points to the value 0.0 instead of 100.0.

As a test of ZPL's expressiveness, we have written the SIMPLE computational fluid dynamics benchmark in our new language. We have no comparison against other data parallel implementations, but we can compare our ZPL version against those written in other paradigms. A parallel implementation written in a MIMD message passing style was about 5000 lines of C code [10]. A sequential implementation from Cornell was about 2400 lines of Fortran code. Meanwhile, our ZPL version was less than 500 lines. While comparing lines of code is admittedly a poor metric of clarity, the large disparity, along with our own experience with the C version, point out the superiority of the ZPL formulation with respect to readability.

We conclude this discussion of data parallel languages by briefly considering compilation issues. Optimizing compilers typically stumble because they must be conservative in the face of uncertainty. This uncertainty can have many sources, such as pointer references or functions in external modules with unknown side effects. Our language makes it difficult for the compiler to stumble because so much information is known at compile time. In ZPL there are no arbitrary array indices, no pointers, and no goto's. Moreover, ZPL's high level abstractions provide semantic information that the compiler can exploit. For example, the use of regions provides a way to recognize subportions of arrays and to perform finer grained data dependency analysis on arrays than is typically feasible. Thus, we are confident that ZPL can be very efficiently compiled.

3 Language Definition

By limiting the scope of ZPL to purely data parallel aspects we are following the creed of giving special treatment to the "common case." Given this premise, our design goals are as follows.

- *Allow users to program at a high level*, namely, by using arrays.
- *Provide an extremely efficient language.* The use of high level abstractions can supply information to optimizing compilers that lower level languages cannot. For example, communication is only induced by operators such as scan and "at." It is not possible to generate irregular communication.
- *Provide a clean language with only a few central concepts.* This, too, is intended to help both the ZPL compiler writer and the applications programmer by reducing feature interaction. There are no explicitly parallel constructs.
- *Provide support for boundary conditions* since they are the most tedious aspect of data parallel computing.
- *Provide freedom to the MIMD aspects of Orca.* Although ZPL can be viewed as a stand-alone language, ZPL must also fit in the framework of the Orca languages where programmers will write their own MIMD phases. This integration is possible because ZPL makes few assumptions regarding parallelism.

The ZPL design can be divided into a sequential component and a data parallel component. In essence, any sequential language can form the basis for ZPL – we choose one that is similar to Modula-2. The remainder of this section focuses on the data parallel aspects of ZPL.

Array Operations. ZPL has two classes of variables – arrays and scalars – that can serve as basic units of computation. All of the standard scalar operators that are typically found in sequential procedural languages exist in ZPL and can be applied to either class of variables. In the case of array operands, the operator is applied to each element of the array to produce an array result. For this reason, array operators can only be applied to conformable arrays – arrays with the same size, shape and base type. For example, the binary addition operator (`+`), when applied to arrays, sums the corresponding elements of two arrays.

When scalars and arrays are combined in an expression, scalars are promoted to conformable arrays. Similarly, functions can be promoted, which allows scalar code to be re-used without making source code changes. For example, a `sqrt()` library routine can be promoted by passing it an array argument, as shown below. The result of this function call is an array of the same size and shape as `A` containing the square roots of each element of `A`.

```
                            /* A and B are arrays. */
    B := sqrt(A);           /* Promote the scalar sqrt() function. */
```

Regions. Conformability of arrays is specified at the statement level through the use of regions. A Region is an index set that is prepended to statements and qualifies all arrays in the statement that have the same rank as the prepended region. Regions can have an arbitrary number of dimensions and are defined in a manner similar to the way arrays are declared in most languages:

```
    region      R = [x1..x2, y1..y2];
```

The above declaration defines a two dimensional region whose points are the cross product of (x1..x2) and (y1..y2). The values of all regions are fixed at load time. They typically will depend on some runtime parameter that describes the problem size. Once defined, regions can be applied to statements as shown below, where we assume that `A`, `B` and `C` are 2D arrays.

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

The effect of the above statement is to compute `A[i][j] = B[i][j] + C[i][j]` for $x1 \leq i \leq x2$ and $y1 \leq j \leq y2$. In other words, the region [R] defines an index set for all 2D arrays to which the region is applied. This example applies a region to a single statement, but in general, regions provide scope in a block structured manner. They can be applied to blocks of statements, and regions can be nested within other regions. The following code fragment illustrates this point.

```
  [R1] A := B + C;          /* use Region R1 for this statement */

  [R2] begin
         A := B + 5*C;      /* use Region R2 for this statement */
  [R1]   B := A;            /* use Region R1 for this statement */
         foo(A);            /* use Region R2 for this function call */
       end
```

The above uses of regions assume that all array operands have been declared to include at least the points in their respective regions. Arrays can be declared using regions as shown below:

```
    var         A: real [R];
                B: real [R];
                C: real [R];
```

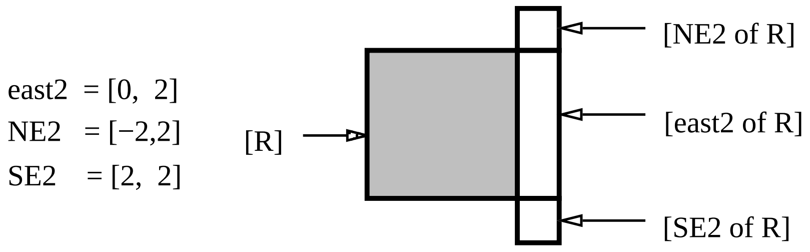

Figure 1: Illustration of "Of" Regions.

By applying regions to entire statements, ZPL programs are syntactically clean since they are not muddled by complicated indices. Significantly, region scopes are the *only* way to reference arrays. In particular, explicit array indexing is not allowed, since this would allow or even encourage programmers to use ZPL for applications that are not strictly data parallel. [3]

Directions. A direction is a vector that can be used to access neighboring array values with the At operator (`@`) and to define new regions from existing regions. Like regions, directions have an associated rank, and their values are initialized once at load time and do not change during the execution of a program. Examples of direction declarations are shown below.

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

The At operator (@) is used to access neighboring array elements. For example, `A@east` refers to the array that has the same shape as `A` – as defined by the region scope that applies to this expression – but is displaced from `A` by the vector `east`, which in this case is `[0,1]`.

All regions can be defined using explicit upper and lower bounds for each dimension, as shown above for region `R`. For example, the region consisting of the two columns to the right of `R` could be defined in the following manner:

```
region   R     = [1..N, 1..N];
         East2 = [1..N, N+1..N+2];
```

However, ZPL provides a less tedious solution that uses the Of operator and the `east2` `vector`, shown below. (See also Figure 1.)

[3] It is, however, possible to initialize arrays through I/O statements.

```
direction east2 = [0,2];
          SE2   = [2,2];

region    East2 = [east2 of R];
```

The Of operator takes two operands – a vector v and a region R – and describes a new region that is adjacent to R. The size of this new region is defined by the direction and magnitude of v and its location relative to R is defined by the direction of v. For example, `[east2 of R]` refers to the region that is offset from R by the vector [0,2], whose height is the same as R's height (as specified by the special value 0), and whose width is 2. As another example, `[SE2 of R]` refers to the 2×2 square that is adjacent to the southeast corner of region R.

Mathematically, if vector v $= (v_1, v_2, ...v_n)$ and
region R $= (l_1, u_1) \times (l_2, u_2) \times ... \times (l_n, u_n)$,

[v of R] defines an array of size $v'_1 \times v'_2 \times ... \times v'_n$,
where $v'_i = | v_i |$ if $v_i \neq 0$,
or
$v'_i = u_i - l_i + 1$ if $v_i = 0$.

This array is located such that it is adjacent to R at $(r_1, r_2, ...r_n)$,
where $r_i = l_i$ if $v_i < 0$
or
$r_i = u_i$ if $v_i > 0$.

In other words, if $v_i < 0$, the upper bound of the "Of" array is the lower bound of the i^{th} dimension. If $v_i > 0$, the lower bound of the "Of" array is the upper bound of the i^{th} dimension.

Reductions and Scans. Reductions and scans are common operations that are known to have efficient parallel implementations [3, 9]. ZPL supplies reductions and scans as operators because this allows them to be optimized by the compiler. For example, if reductions are not provided by the hardware, the messages needed to implement consecutive reductions can be combined to reduce communication latency.

Reductions are operations applied to the elements of an array to produce a scalar result. For example, the sum reduction of an array is the sum of the array's elements. The following reductions are defined as operators in ZPL:

`+\`	– Sum reduction
`*\`	– Product reduction
`max\`	– Maximum reduction
`min\`	– Minimum reduction

`and\`	– And reduction
`or\`	– Or reduction

Each of the above reduction operators has an analogous scan operator. The syntax for scans uses `\\` instead of `\`. Scan operators take an array argument and return an array containing the parallel prefix of the given array: Each element of the returned array is the reduction of all elements whose indices are less than its own. For multidimensional arrays each element is assigned an index in row-major order. That is, the last (rightmost) dimension's indices vary fastest.

Partial reductions and scans are also allowed. For example, row reductions can be performed across each row of a 2D array, returning a vector result. Partial reductions and scans are expressed by using an optional parameter that specifies the dimensions across which the reduction or scan should take place. Dimensions are specified by placing numbers in brackets, with `[1]` indicating the first dimension and `[d]` indicating the d^{th} dimension. The code fragment below, where `v` is a vector and `s` is a scalar, shows examples of partial reductions.

```
[R]   v := +\[1] A;      -- reduce A along columns and assign to vector v
[R]   s := +\[1][2] A;   -- full reduction of A
[R]   s := +\ A;         -- full reduction of A
```

The Where Statement. The Where statement is the array analog of an If statement. Its form is given below, where `cond` is a conditional expression involving array values, and `s1` and `s2` are statements.

```
where (cond)
then  s1;
else  s2;
```

The statement `s1` is executed for those index values that evaluate to true in the Where's conditional expression. Statement `s2` is executed for all other index values. Semantically, `s1` and `s2` are allowed to execute concurrently. A Where statement can only be qualified by a single region scope. That is, the statements in the body of the Where cannot be modified by additional region scopes. Given these constraints, Where statements are allowed to nest.

Reflect and Wrap. ZPL provides specific support to two common types of boundary conditions: the case where a set of values is reflected across some boundary of an array, and the case where a set of values is wrapped around from the other side of an array as in a torus. These operations are supported by the Reflect and Wrap operators, respectively.

Reflect and Wrap are syntactic sugar for initializing two common types of boundary conditions. They are particularly useful for boundary conditions that are wider than a single row or column. The Reflect operator assigns values by mirroring them across some

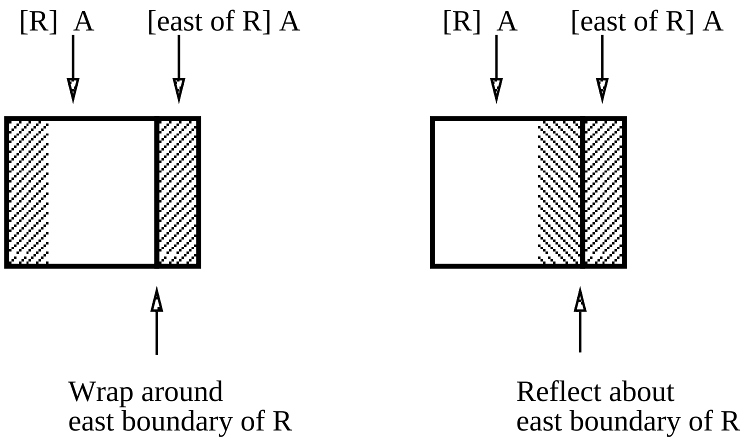
[R] A
[east of R] A
[R] A
[east of R] A
Wrap around east boundary of R
Reflect about east boundary of R

Figure 2: The Reflect and Wrap Operators.

point of reference, while the Wrap operator is used to set boundary conditions that wrap around as if the array were a torus.

The Reflect operator takes a region, [v of R], and a variable (or list of variables), X, and sets the value of X in the region [v of R] to the values of X that are reflected across the boundary between [R] and [v of R]. The region is a no-op if it does not contain an Of operator. The Reflect operator can be applied to multiple variables at once. For example, the following code sets, for variables `A`, `B`, and `C`, the column to the east of `R` to have the value of the eastern-most column of `R`. This is illustrated in Figure 2, which corresponds to the program text below.

```
[east of R]   reflect A, B, C;   -- reflect across the eastern border
[R]           reflect A;         -- no-op, nothing to reflect across
```

Imagine that the boundary between [R] and [v of R] is connected to the opposite boundary of [R] as in a torus. The Wrap operator then sets the values of X in the region [v of R] such that this torus is emulated. The Wrap operator is illustrated in Figure 2. The semantics of Wrap are defined mathematically as follows.

if vector v $= (v_1, v_2, ...v_n)$ and
region R $= (l_1, u_1) \times (l_2, u_2) \times ... \times (l_n, u_n)$,

[v of R] Wrap X; sets the values of X in the region [v of R] to the value of the conformable array [v′ of R], where $v'_i = (u_i - l_i) + v_i$.

Execution Order. Semantically, each region scope is executed sequentially. Within each region scope, statements are logically executed sequentially. Within each assignment statement, the right hand side is first evaluated and then assigned to the left hand side. While such assignments logically imply the use of temporary array variables, compiler analysis can often remove the need for temporaries. Recursion is allowed.

Lists of Regions. As syntactic sugar, lists of regions are allowed. For example, a region named `border` can be defined as shown below.

```
region     R      = [x1..x2, y1..y2];
           border = [north of R][east of R][west of R][south of R];

[border]
   b1;     /* b1 represents a block of code. */
```

When this region is then used to qualify a block of statements, `b1`, the block of statements is logically executed once for each of the regions in the list.

Syntax and Other Details. The current implementation of ZPL uses modified Modula-2 syntax. All Modula-2 constructs are preserved except there are no Case statements, no goto's, and no pointers. Recursion is allowed. In addition, ZPL uses two-level scoping, i.e. procedures are not allowed to be defined inside the scope of other procedures. This detail stems from the fact that our implementation is based on the Parafrase-2 [13] source to source translator for the C language. However, as in Modula-2, a block of statements (bracketed by `begin` and `end`) is itself considered a statement.

4 Discussion

4.1 Support for Boundary Conditions

ZPL gives support for boundary conditions in the form of the Reflect and Wrap statements described above. In addition to these two special forms, ZPL provides general support for boundary conditions through the use of regions.

Regions describe geometric areas of the data space. Because the computations on the various regions can be specified independently, the initialization of boundary conditions can be separated from the rest of the code. This modularization simplifies both the boundary condition code and the "main" body of code. From the Jacobi example we see that other languages also allow for the isolation of boundary condition code, but in these languages the separation is contingent upon the programmer's good programming style. For example, in HPF the individual indices of each array (or the individual indices of each forall statement) must be inspected to determine what portions of the data space are being manipulated.

An advantage of applying regions to entire blocks of statements is that the separation among different portions of the data space is clearly manifested. For example, the typical structure of a ZPL program is shown below, and it's immediately apparent which parts of the computation apply to which portions of the data space.

```
[border]
  begin
  ...
  end

[R]
  begin
  ...
  end
```

4.2 Relationship to Orca C

As a first approximation, a program written using the full capabilities of the Orca C language [12] will utilize one of four types of statements:

```
    [R]   A := sqrt(A);           -- data parallel assignment

    err := max\ Delta;            -- "standard" parallel abstraction

    if err < 10**-6 then. . .     -- control construct
        else  . . .

    PivotExchange();              -- phase invocation
```

The first three types, data parallel operations, "standard" parallel abstractions and control operations, form the ZPL sublanguage. The fourth statement type, phase invocation, provides access to the full MIMD capabilities of Orca C. By limitation, these can also be regarded as SPMD capabilities.

Collectively, the four statement types are said to define programming at the "problem level" of Orca C. This name emphasizes the fact that these statement types express the high level solution of the programmer's problem. For historical reasons, the problem level is also known as the *Z level* [17]. The problem level text describes the main logic of the problem solution. If the solution is sufficiently simple, ZPL constructs alone may suffice to express the program. But for complex algorithms or certain performance-critical situations, the full MIMD capabilities of the language may be needed.

A phase is a user-defined parallel computation composed of concurrently executing processes. (Phases can also be library defined.) An example might be an FFT computation. Phases have a characteristic communication structure induced by the data dependencies of the problem, e.g. the butterfly for the FFT. The language constructs provided in Orca C for defining phases are referred to, again for historical reasons, as the *Y level* of programming. And the constructs provided for defining the processes, out of which phases are defined, are known as the *X level* of programming. Further description of the phase abstractions programming model can be found in the literature [18].

Though a full introduction of the Y and X programming facilities of Orca C, and a careful introduction to the phase abstractions concepts would be beyond the scope of this paper, a brief description of the phase definition approach is appropriate. Phases are defined using an abstraction called *ensembles*. Three kinds of ensembles are required to define a phase. *Data ensembles* are partitioned data structures, *code ensembles* are partitioned sets of process instances, and *port ensembles* are partitioned graphs, the edges of which define the channels for interprocessor communication. To create a phase, the user defines the necessary ensembles so that they have the same partitioning. (See Figure 3.) This allows the partitions of the three types of ensembles to be put into one-to-one correspondence. In each partition there will be the following: a process instance, the portions of the partitioned data structures the process is to operate on, and a vertex of the partitioned graph that defines the neighbors with which the process communicates. Collectively, the ensembles define a parallel computation whose logical concurrency is given by the number of partitions.

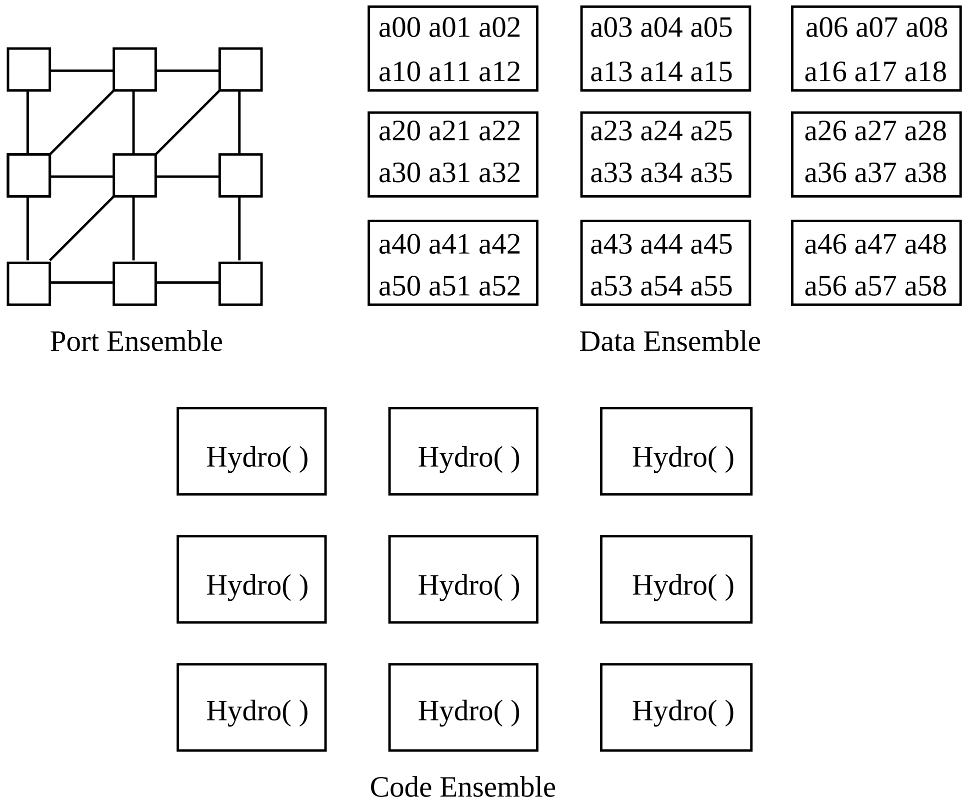

Figure 3: Illustrations of Ensembles for the Hydro Phase of SIMPLE.

The ZPL language has been presented (and can be used) as if it were a stand-alone language, but it is simply a language convenience: The phase abstractions programming model dictates that Z level statements are either control statements, such as an `if` statement, or they are phase invocations. A literal implementation of this requirement would force users to program all of the concurrent activity of the computation explicitly, a potentially tedious task given how simple certain aspects of parallel computation can be.

Accordingly, ZPL's data parallel constructs and "standard" parallel abstractions are small compiler generated phases, or *phaselets.* That is, logically, each data parallel statement or "standard" operation is converted by the compiler into a phase composed of generated processes that execute the operation on the portion of the array stored locally in the processor. The data layout is not specified in the ZPL portion of the language, but is inherited from whatever data ensemble is declared as part of the user defined phases. If no data layout has been specified by the user, e.g. if ZPL is simply treated as a stand alone language, then a default block layout is used for ZPL arrays.

Logically, there is a barrier synchronization following each statement in the Z level language. Since in most cases the barrier is unnecessary, i.e. the computation is correct without actually synchronizing, the compiler eliminates barriers when possible. Also, phase invocation overhead can be eliminated by combining the phaselets into larger, more completely optimized phases. Indeed, when ZPL is used as a stand alone language, without true phase invocations, then only one phase per processor is generated. This gives ZPL a greater efficiency than might be apparent if the literal interpretation of phase abstractions – every line is a control structure or a phase invocation – were strictly enforced.

5 Other Related Work

ZPL inherits the notion of regions from Spot [19], a language designed to support stencil-based computations on distributed memory computers. The two languages are syntactically similar, but Spot presents a point-based view which leads to a certain awkwardness. For example, *history values* are introduced to maintain multiple values of variables across iterations.

Numerous other data parallel languages exist, including C*, Fortran 90, HPF, and NESL, which were briefly discussed in Section 2. Here we point out a few others. Broadly speaking, these languages are more general than ZPL since ZPL is an attempt to trade off expressiveness for improved clarity and efficiency. On the other hand, these other languages do not provide the same support for boundary conditions that ZPL does. C* and Data-parallel C [7] have SIMD execution semantics and provide pointers. Dino [16] programs can specify arbitrary point-to-point communication through tagged accesses to variables. This more general communication adds power but complicates optimizations.

6 Conclusion

In this paper we have introduced ZPL, a data parallel programming language whose goals are clarity and efficiency. Because this sublanguage exists in the context of the more general programming model, ZPL is freed from having to support complicating operations that would be needed for pivots or irregular communication. By allowing region scopes to be applied to entire statements (and blocks of statements), ZPL provides support for boundary conditions by syntactically separating boundary condition code from common case code. This same mechanism removes explicit indexing from array references, which improves readability and allows the re-use of existing sequential code for data parallel applications.

The implementation of a ZPL compiler began in April, 1993 and is expected to be completed in the summer of 1993. We are modifying the Parafrase-2 source to source translator [13, 14] to compile ZPL code down to C for a variety of parallel computers. We believe that this system will provide an ideal testbed to experiment with novel compiler optimizations for parallel computers.

Acknowledgments. We thank Alex Klaiber, Chuck Koelbel, and Guy Blelloch for providing C*, HPF, and NESL implementations of Jacobi. We are grateful to Brad Chamberlain, George Forman, and Derrick Weathersby for writing ZPL programs for the Floyd-Steinberg algorithm, the game of Life, and the Ising model, respectively, and to Bill Griswold and Sungeun Choi for comments on an early draft. Finally, we are indebted to George Forman for many fruitful conversations regarding the ZPL language implementation.

References

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

[2] Gail Alverson, William Griswold, David Notkin, and Lawrence Snyder. A flexible communication abstraction for nonshared memory parallel computing. In *Proceedings of Supercomputing '90*, November 1990.

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

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

[5] R. E. Cypher, J. L. C. Sanz, and L. Snyder. Algorithms for image component labeling on simd mesh connected computers. *IEEE Transactions on Computers*, 39(2):276–281, 1990.

[6] William Griswold, Gail Harrison, David Notkin, and Lawrence Snyder. Scalable abstractions for parallel programming. In *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990. Charleston, South Carolina.

[7] Philip J. Hatcher, Michael J. Quinn, Ray J. Anderson, Anthony J. Lapadula, Bradley K. Seevers, and Andrew F. Bennett. Architecture-independent scientific programming in Dataparallel C: Three case studies. In *Proceedings of Supercomputing '91*, pages 208–217, 1991.

[8] High Performance Fortran Forum. *High Performance Fortran Specification.* January 1993.

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

[10] Jinling Lee. Extending the SIMPLE program in Poker. Technical Report 89–11–07, Department of Computer Science and Engineering, University of Washington, 1989.

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

[12] Calvin Lin and Lawrence Snyder. Data ensembles in Orca C. In *5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[13] Constantine Polychronopolous, Milind Girkar, Mohammad Reza Haghighat, Chia Ling Lee, Bruce Leung, and Dale Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 39–48, August 1989.

[14] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. The structure of parafrase-2: an advanced parallelizing compiler for c and fortran. In *Workshop on Languages and Compilers for Parallel Computing*, pages 423–453.

[15] J.R. Rose and Guy L. Steele Jr. C*: An extended C language for data parallel programming. Technical Report PL 87-5, Thinking Machines Corporation, 1987.

[16] M. Rosing, R. Schnabel, and R. Weaver. The Dino parallel programming language. Technical Report CU-CS-457-90, Dept. of Computer Science, University of Colorado, April 1990.

[17] Lawrence Snyder. The XYZ abstraction levels of Poker-like languages. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 470–489. MIT Press, 1990.

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

[19] David Grimes Socha. *Supporting Fine-Grain Computation on Distributed Memory Parallel Computers.* PhD thesis, University of Washington, Department of Computer Science and Engineering, 1991.

A Fragments of the SIMPLE Code

There is not enough space to show the entire SIMPLE code, so this appendix contains fragments of the SIMPLE program.

```
direction ne    = [-1,1];                         /* declare directions */
          . . .

region    R     = [minX..maxX, minY..maxY];       /* declare regions */

var       Rho, P, Q    : real [R];                /* declare arrays */
          Delta_tau    : real [R];
          . . .
                                                  /* define functions */
function Jacobian();
var   Tmp_J1, Tmp_J2 : real[R];
begin
  Tmp_J1 := (X.r * (X.z@south - X.z@west)
            + X.r@west * (X.z - X.z@south)
            + X.r@south * (X.z@west - X.z)) / 2.0;

  Tmp_J2 := (X.r@west * (X.z@south - X.z@sw)
            + X.r@sw * (X.z@west - X.z@south)
            + X.r@south * (X.z@sw - X.z@west)) / 2.0;

  J := Tmp_J1 + Tmp_J2;
  Old_S := New_S;
  New_S := ((X.r + X.r@west + X.r@south) * Tmp_J1 +
           (X.r@west + X.r@sw + X.r@south) * Tmp_J2) / 3;
end

function main()                              /* Main body */
[R]
begin
  /* Hydro Phase */
  [west of R]  P := west_bound_p;
  [north of R]
  [NE of R]
  [east of R]  reflect Rho, P, Q, J;

  Jacobian();
  . . .

  En_error := Int_en + Kin_en - Work + Heat;
  En_error := +\ En_error;                /* implicit temporary variable */
end;
```