Abstractions for Dynamic Data Distribution

Steven J. Deitz* Bradford L. Chamberlain*† Lawrence Snyder*

*University of Washington
Seattle, WA 98195
{deitz,brad,snyder}@cs.washington.edu

†Cray Inc.
Seattle, WA 98104
bradc@cray.com

Abstract

Processor layout and data distribution are important to performance-oriented parallel computation, yet high-level language support that helps programmers address these issues is often inadequate. This paper presents a trio of abstract high-level language constructs—grids, distributions, and regions—that let programmers manipulate processor layout and data distribution. Grids abstract processor sets, regions abstract index sets, and distributions abstract mappings from index sets to processor sets. Each of these is a first-class concept, supporting dynamic data reallocation and redistribution as well as dynamic manipulation of the processor set. This paper illustrates uses of these constructs in the solutions to several motivating parallel programming problems.

1. Introduction

Languages and libraries for parallel programming can roughly be divided into two categories: *local-view* facilities and *global-view* facilities. Local-view facilities, typified by message passing libraries (*e.g.*, MPI), Co-Array Fortran, Titanium, and UPC, require programmers to express their computation from the point of view of a single processor. On the other hand, global-view facilities, typified by HPF and ZPL, allow programmers to express their computation as a whole (without excessive regard to the multiple processors that will execute it).

Both schemes must handle dynamic data distributions because important applications such as Adaptive Mesh Refinement (AMR) require data to be redistributed on the fly. Local-view languages handle dynamic distributions because with a minimum of abstraction the programmer must implement everything; changing the organization of data is complicated, but it is possible with the same abstractions necessary for managing all the normal details of data reallocation, inter-processor communication, and intra-processor copying.

On the other hand, global-view languages, which rely heavily on high-level abstractions to insulate programmers from low-level details, present much greater challenges for the language designer. Programmers need to describe, abstractly, their preferences for initial data distribution, and then must be provided with mechanisms to move data around in response to dynamically changing conditions.

In this paper we present a hierarchy of constructs—grids, distributions, regions, and arrays—created for ZPL for dynamically distributing data, and we illustrate that they are convenient for solving commonly arising parallel programming problems. Figure 1 illustrates these constructs with a simple example. Grids abstract processor sets. In our example, grid `G` is a 2×2 processor set. Distributions abstract mappings from index sets to processor sets. Distribution `D` divides the 6×4 index set into equally sized blocks and distributes these blocks across `G`. Regions abstract index sets. Region `R` is the 6×4 index set that is distributed by `D` across `G`. Parallel arrays allocate data over regions. Array `A` contains 24 values corresponding to the positions in `R`; `D` distributes these values across `G`.

A key property of our solution is that it allows programmers to assess and to control the communication demands of dynamically distributing data. Global-view language abstractions for supporting dynamic data distribution have not previously done so. For example, HPF includes directives for specifying data distributions, but the communication requirements for implementing the specified data motion are invisible to programmers. Our solution avoids this problem because, in part, the hierarchy of abstractions provides a layer of insulation around the processors.

In the next section, we introduce a subset of ZPL. In Section 3, we present the grid and distribution abstractions, which allow programmers to specify dynamic data distributions. In Section 4, we outline solutions to several motivating parallel programming problems. We describe related work in Section 5 and conclude in Section 6.

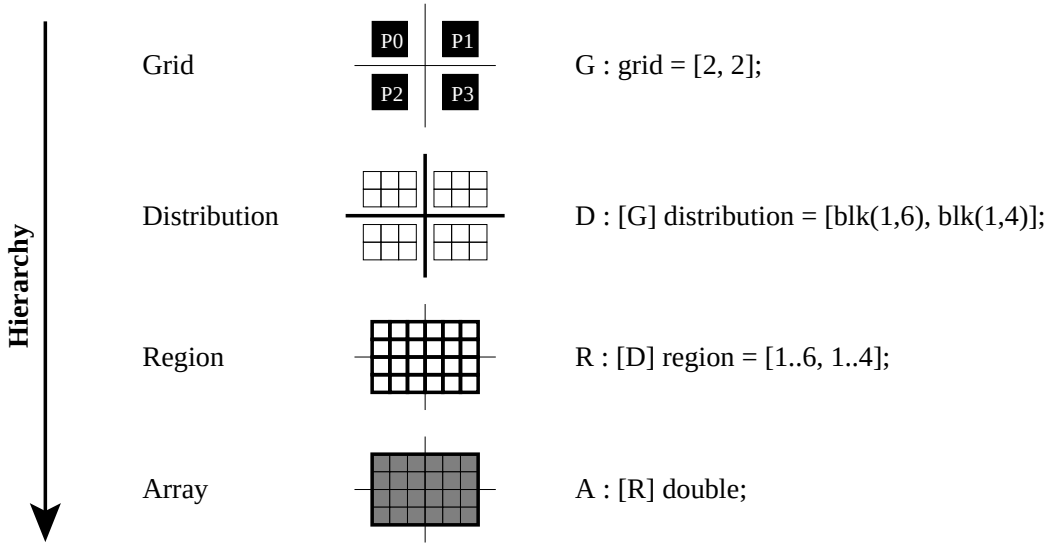

Figure 1. An example of a grid, distribution, region, and array organized in a simple hierarchy. The array A allocates data over region R. The data values are distributed according to distribution D over the processors in grid G.

2. Introduction to ZPL

ZPL is a global-view parallel programming language developed at the University of Washington. The current implementation is a compiler that translates ZPL to C with calls to MPI, PVM, or SHMEM, as the user chooses. In this section, we present a brief introduction to ZPL. For a more detailed introduction to ZPL, the interested reader is referred to the literature [2, 12].

2.1. Basic Concepts

The central abstraction underlying ZPL is the *region* [3]. Regions are index sets with no associated data. To declare the `n` $\times$ `n` region `R` and the `(n+2)` $\times$ `(n+2)` region `BigR`, which expands `R` at its border, a programmer writes

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

Regions are used to specify indices for computation and to declare *parallel arrays*. To declare parallel arrays using regions `R` and `BigR`, a programmer writes

```
var
  A, B : [BigR] double;
  C : [R] double;
```

A value is allocated for each index in the array's region, resulting in an `n` $\times$ `n` array for `C` and `(n+2)` $\times$ `(n+2)` arrays for `A` and `B`.

To specify indices for computation so that the element-wise sums of the interiors of `A` and `B` are stored in the corresponding positions in `C`, a programmer writes

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

In ZPL, each region's indices are distributed over a set of processors, implying a distribution for any arrays declared using that region. (Later we will see how modifying a region's distribution will result in the reallocation of an array.) Scalars, on the other hand, are replicated and kept consistent on every processor by the language's semantics. For example, to declare the scalar integer `n` used in the region specifications above, a programmer writes

```
var n : integer;
```

In addition to the parallel array, ZPL supports a second type of array called an *indexed array*. Like scalars, indexed arrays are replicated and kept consistent. As the name implies, values in indexed arrays are accessed through *indexing*, much like in C and Fortran. Indexing is disallowed for parallel arrays whose values can only be accessed using regions.

Indexed arrays can be composed with parallel arrays. To declare an indexed array of parallel arrays and a parallel array of indexed arrays, a programmer writes

```
var
  IofP : array[1..n] of [R] double;
  PofI : [R] array[1..n] of double;
```

Indexed arrays can be composed arbitrarily unlike parallel arrays. It is illegal to declare a parallel array of parallel arrays.

2.2. Parallel Array Operators

For computations that are not strictly element-wise, ZPL supports several array operators, each providing a different access pattern and communication style. The *at operator* (@) shifts an array's values by an offset vector called a *direction*. As an example, the statement

```
[R] A := A@[0, -1] + A@[0, 1];
```

replaces each element in the interior of A with the sum of its left and right neighbors.

The *reduce operator* (op<<) computes reductions using either built-in or user-defined combining operators [6]. It can be used to reduce values in a parallel array to a scalar or parallel subarray. As an example, the following statements compute the minimum value in the interior of A and store the sums of the rows of A in its first column:

```
var minval : double;

[R] minval := min<< A;
[0..n+1, 0] A := +<<[BigR] A;
```

The *remap operator* (#) generalizes gather and scatter; it is used to move data within or between arrays in more complicated patterns than the shifts supported by the at operator. Using the built-in constant Index*i* arrays, values in an array can be transposed. The Index*i* arrays contain the *i*-dimension index values. For example, the first two Index*i* arrays over the region given by [1..3,1..3] are

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

To transpose the values of A, a programmer can write

```
[BigR] A := A#[Index2, Index1];
```

The implementation of the remap operator is necessarily complex, though its use can often be optimized [7].

2.3. WYSIWYG Performance

A key aspect of ZPL (and one that distinguishes it from all other global-view parallel languages) is its performance model. Known as the *what-you-see-is-what-you-get* (WYSIWYG) performance model, it specifies when the program may require communication and how that communication will be implemented. The indication of when communication must be generated is indicated by the use of certain operators. All arrays that interact in a given statement must have the same distribution as all other arrays unless the remap operator is applied to it. So, for example, when programmers write

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

they know there will be no communication because ZPL's performance model guarantees that basic element-wise array computations never require communication.

However, when programmers write a similar expression

```
[R] A := A@[0, -1] + A@[0, 1];
```

they know there probably will be communication (to refer to A's westerly and easterly neighbors) because @s induce communication. Furthermore, they know that the communication will be a simple point-to-point communication. Similarly, the reduction operator op<< typically induces log-depth communication to combine values across processors. The remap operator # induces potentially all-to-all communication and is considered expensive.

Thus, although ZPL is a global-view language, the communication is explicit: Whenever programmers use an operator inducing communication, they know it. The WYSIWYG model contributes to faster programs by giving programmers knowledge about communication, allowing them to choose solutions that minimize communication overheads without requiring them to actually program it.

3. Grids, Distributions, Regions, Arrays

In this section, we introduce *grids* and *distributions*. These two constructs give programmers control over data decomposition. We then reintroduce regions as first-class constants or variables with associated types and values. We show that it is possible to dynamically redistribute and reallocate data by merely reassigning grids, distributions, or regions

3.1. Grids

ZPL's grid construct lets programmers abstract the processor set. For example, the following three grids organize the processors:

```
const
  p : integer = numLocales();
grid
  G1 = [1, 1..p];
  G2 = [1..p, 1];
  G3 = [1..p/2, 1..2];
```

Grid G1 organizes the processors into a $1 \times p$ grid, grid G2 into a $p \times 1$ grid, and grid G3 into a $p/2 \times 2$ grid. The built-in procedure numLocales returns the number of processors executing the program.

Although it is easy to determine from the above declarations that each grid contains every processor, it is impossible to control where the processors are positioned. Allocating specific processors to specific positions in a grid is possible with indexed arrays of non-repeating processor IDs. The size and shape of the grid is inherited from the size and

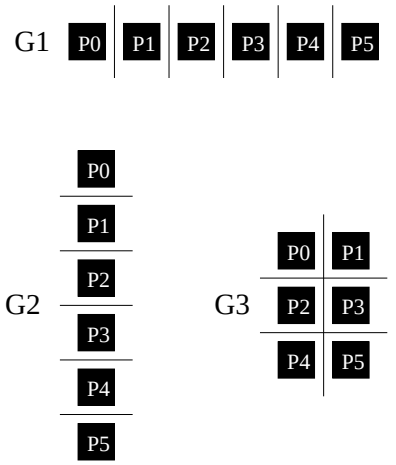

Figure 2. An example showing grids G1, G2, and G3 from Section 3.1. For illustrative purposes, we assume there are six processors.

shape of the indexed array. For example, the following declarations create a pair of 2×2 grids that split the first eight processors into two disjoint sets based on whether the processor ID is even or odd:

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

The ability to create disjoint grids extends the applicability of ZPL into problems traditionally characterized as *task parallel* [5].

Programmers are often unconcerned with which processors map to which positions of a grid, and in this case the first syntax is sufficient. Much of the time, even the number of processors allocated to each grid dimension is more than programmers wish to consider. The keyword **auto** asks the compiler and runtime to heuristically allocate processors to a grid dimension. For example, the following grid declaration uses the **auto** keyword to divide the processors between the first two dimensions of grid G6, leaving the last dimension degenerate:

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

With the **auto** keyword, a grid dimension is indexed starting with 1 and ending with the number of processors allocated to that dimension.

The syntax we have been using to declare grids is syntactic sugar for constant grid declarations. Grids are first-class concepts. They have types and values associated with them and can be declared as variables. Without changing the semantics of the grids we have introduced in this section, we can redeclare all of the above grids as follows:

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

Notice that the type of a grid is not simply **grid**, but also includes the rank of the grid. In addition, whether the dimension *may* be allocated to more than one processor (`..`) or not (`.`) is part of the type. While the casual programmer can use "`..`" everywhere, using "`.`" could result in a more optimized program. The type of a grid can be inferred from its initializer; this inference is always done with the syntactic sugar. In this case, the keyword **grid** is a sufficient type specification. The BNF for grid types and values can be found in Figure 4. By changing the keyword **const** to **var**, we can create mutable grids that do not need initializers.

3.2. Distributions

ZPL's distribution construct lets programmers map logical indices to grids. For example, if G is a 2D grid where neither dimension is degenerate, n and m are integers, and a1 and a2 are 1D indexed arrays of integers, then the following distributions map indices to grid G using two built-in distributions:

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

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

The syntax encountered so far for declaring a distribution is syntactic sugar for a constant distribution declaration. Distributions are first-class concepts with associated types and values. Without changing the semantics, we can redeclare distributions D1 and D2 as follows:

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

Distributions are statically bound to grids; in this example, D1 and D2 are bound to G. The distribution's type

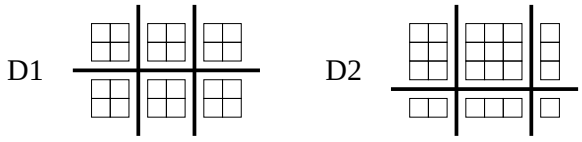

Figure 3. An example showing distributions D1 and D2 from Section 3.2. For illustrative purposes, we take G to be a 2 by 3 grid of processors. We assume values of 4 and 6 for n and m respectively and illustrate the 4 by 6 index set over these two distributions. The array a1 is taken to contain the single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

is composed of the grid's type, the rank of the distribution (which must match the rank of the grid), and the per-dimension distribution types. The grid is not part of the distribution's type. For example, we can gather the following information from the type of `D1`: It is a 2D distribution built on a non-degenerate 2D grid and both of it's dimensions use block distribution functions. Both the `blk` and `cut` distributions are built-in distribution functions and have a **`block`** distribution function type.

There are five distribution function types. We have already discussed the **`block`** type. The **`cyclic`** type implements the standard cyclic distribution instantiated by the built-in distribution `cyc`. The **`multiblock`** distribution function type implements distributions like block-cyclic with `blkcyc`. It is also easy to image a cut-cyclic distribution implemented with `cutcyc` of the **`multiblock`** type. The **`nondist`** distribution function type with the `null` distribution optimizes the degenerate case where the grid dimension is allocated to only one processor. Lastly, the **`irregular`** distribution type is a catch-all for distributions that do not fall into the other categories. (In this paper, we focus only on dimension-orthogonal distributions omitting any discussion of multidimensional distributions like recursive coordinate bisection.)

By separating a distribution's type from its value, our implementation strategy for differing distributions is apparent. The broad definitions of the types limits the complexity of the compiler and runtime. For example, they can treat the cut and block distribution identically because the loop structure and communication patterns are similar. At the same time, new distributions are easy to add to the language since the amount of code required is small and modular. We are currently finalizing the mechanisms which would let ZPL programmers define their own distributions.

The BNF for distribution types and values can be found in Figure 4. As with grids, by changing the keyword **`const`** to **`var`**, we can create distributions that may be changed and need not be initialized.

3.3. Regions

In Section 2, we encountered ZPL's regions. A region's distribution is controlled through the distribution it is declared over. Regions are bound to distributions in the same way that distributions are bound to grids. For example, the following declarations define two $n \times n$ regions, one bound to `D1` and one bound to `D2`:

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

Recall that regions are used to declare parallel arrays. To declare parallel arrays over `R1` and `R2`, a programmer writes

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

This creates a static hierarchy from grids to distributions to regions and finally to arrays. In particular, the elements of an array are associated with the indices of a region which are mapped to a processor in a grid via a distribution. This hierarchy is used to control array distributions. Changing any piece of this hierarchy potentially changes the distribution of data and/or computation without requiring the user to manage the details or modify the expression of the computation that acts on the arrays.

Since `A` and `B` are distributed using different distributions, they cannot interact without explicit use of the remap operator. This maintains the WYSIWYG performance model in the presence of multiple grids and distributions. To copy the values from `B` to `A`, programmers write

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

Programmers can also elide the default *i*th map arrays by writing

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

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

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

the anonymous region inherits its distribution from array `A`. When distributionless regions are used to declare parallel arrays, the built-in per-rank distributions and grids called *implicit distributions* and *implicit grids* are applied. This lets programmers who do not wish to think about grids and distributions use ZPL in a simpler form.

The syntax for a region declaration is syntactic sugar for a constant region declaration. Regions are first-class with

associated types and values. Without changing the semantics, we can redeclare R1 and R2 as follows:

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

The type of a region includes its rank and its distribution's type. In addition, it includes per-dimension information specifying the dimension's type (degenerate, range, replicated, *etc.*) unimportant to this paper [3]. The BNF for region types and values can be found in Figure 4. As with grids and distributions, by changing the keyword **const** to **var**, we can create mutable regions that do not need initializers.

3.4. Dynamic Grids, Distributions, and Regions

The following code illustrates a simple hierarchy of grid, distribution, region, and array:

```
var
  G : grid = [auto, auto];
  D : [G] distribution = [cut(a1), cut(a2)];
  R : [D] region = [1..nx, 1..ny];
  A : [R] double;
```

Notice that the dimensional type information for the grid, distribution, and region are inferred from the initializers. Since G, D, and R are variables, they can be changed. Such changes result in further changes that ripple down the hierarchy. For example, changing a region effects the data in arrays defined by that region. Changing a distribution effects all of its regions and, consequently, all of the arrays that are declared over these regions. These rippling effects can either preserve the array's contents or not. Which semantics are preferable depends on the application. In destructive assignment (<==) of grids, distributions, or regions, array data values are not preserved.

For example, we can use destructive assignment to transpose the size and shape of R as in the following statement:

```
R <== [1..ny, 1..nx];
```

The data values in A are lost; its new data values are uninitialized. The number of data values on each processor may have changed. Since the newly allocated A is shaped to store the transpose of the old data, we may want to transpose the data from the old A to the new A. In this case, we would need a temporary array to store the data while we reallocated the array:

```
var
  RT : [D] region = [1..ny, 1..nx];
  AT : [RT] double;

AT := A#[Index2, Index1];
R <== RT;
A := AT;
```

A second style of assignment, preserving assignment (<=#), differs from destructive assignment in that the data in the array is moved. Communication can be expensive in preserving assignment, and this is indicated in the operator's resemblance to the remap operator. The data is preserved for every index that is preserved. Data is lost for indices that are lost and new indices are uninitialized. In the above example, destructive assignment was appropriate since we wanted to transpose the data and presumably reuse the space of A.

Suppose instead of transposing the data, we want to redistribute the data in A so that instead of using the cut distribution, we used the blk distribution. The following code will do the trick:

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

Region R's indices are redistributed according to the new distribution value. Then the data in array A is redistributed; this typically requires communication.

In the next section, these assignments are illustrated in various motivating examples.

4. Examples

This section illustrates, in four examples, the use of dynamic grids, distributions, and regions. For each example, we pose a problem, outline a solution in ZPL, and then interpret and comment on the code.

4.1. Computational Zoom

Problem. This problem approximates aspects of a simplification of the Adaptive Mesh Refinement (AMR) technique. Assume an array computation with the property that the more time spent computing on some portion of the array, the more precise the solution is on that portion. This problem asks the programmer to compute an approximate solution over the entire array, determine a portion of the array where more precision is important, and compute an improved solution using *all* the processors.

Solution.

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

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

grid-type ::= **grid** { '<' grid-dimension-type { ',' grid-dimension-type } '>' }
grid-dimension-type ::= '.' | '..'
grid-value ::= '[' grid-range { ',' grid-range } ']' | integer-indexed-array-expression
grid-range ::= **auto** | integer-expression | integer-expression '..' integer-expression

distribution-type ::= '[' grid-expression ']' **distribution** { '<' distribution-dimension-type { ',' distribution-dimension-type } '>' }
distribution-dimension-type ::= **nondist** | **block** | **cyclic** | **multiblock** | **irregular**
distribution-value ::= '[' distribution-function { ',' distribution-function } ']'
distribution-function ::= distribution-identifier ['(' expression { ',' expression } ')']

region-type ::= ['[' distribution-expression ']'] **region** { '<' region-dimension-type { ',' region-dimension-type } '>' }
region-dimension-type ::= '.' | '..' | '*' | '::'
region-value ::= '[' region-range { ',' region-range } ']'
region-range ::= integer-expression | integer-expression '..' integer-expression | '*' | '::'

Figure 4. Collected BNF for the types and values of grids, distributions, and regions. In this extended BNF, curly braces indicate 0 or more instances of the enclosed syntax and square brackets indicate 0 or 1 instances of the enclosed syntax.

Discussion. In this program, we start by declaring grid `G`, distribution `D`, region `R`, and array `A` over which we will compute an approximate solution. We then declare distribution `DZoom`, region `RZoom`, and array `Zoom` over which we will compute an improved solution. Then we pass `A` to the `computeApprox` procedure which computes an approximate solution over `A` and assigns values to the integers which define the indices where more precision is required. In the next two lines, we assign values to `DZoom` and `RZoom`. Because the changes ripple down the hierarchy, array `Zoom` is allocated and distributed across the processors. The data is moved into `Zoom` using the remap operator in the next line. Since the area in `R` that corresponds to `RZoom` may only exist on a subset of the processors and/or be unevenly distributed, a potentially large amount of data may need to be transferred between processors. After procedure `computeMore` is called to compute an improved solution, the remap operator is used again to move the data back from `Zoom` to `A`. Notice that we cannot use region `RZoom` here since its distribution is `DZoom`. Instead, the anonymous region gets its distribution, `D`, from `A`.

Note that an AMR code could be constructed by executing multiple instances of this example in a recursive context and interpolating the zoomed areas. By using indexed arrays of grids, distributions, and regions, the contents could be saved between iterations.

4.2. FFT Corner Turn

Problem. A common way of computing the Fast Fourier Transform (FFT) of a multi-dimensional array in parallel is to leave at least one dimension of the array undistributed. Then the FFT can be computed by executing 1D FFTs alternated with corner turns. This problem asks the programmer to compute the FFT of an `nx` $\times$ `ny` $\times$ `nz` array. Assume only one dimension of the array is distributed, and thus we only need to do a single corner turn. To conserve space, the solution may use only two arrays. Because the lengths of the dimensions are different, load balancing must be taken into account; the arrays before and after the corner turn should be distributed differently.

Solution.

```
const
  p : integer = numLocales();
  G : grid = [p, 1, 1];
  D : [G] distribution
    = [blk(1,nx), null, null];
  R : [D] region = [1..nx, 1..ny, 1..nz];
  DT : [G] distribution
     = [blk(1,ny), null, null];
  RT : [DT] region = [1..ny, 1..nz, 1..nx];
var
  DX1 : [G] distribution = D;
  DX2 : [G] distribution = D;
  RX1 : [DX1] region = R;
  RX2 : [DX2] region = R;
  X1 : [RX1] complex;
  X2 : [RX2] complex;

[R] computeFFT2(X1, X2);
[R] computeFFT3(X2, X1);
DX2 <== DT;
RX2 <== RT;
[RT] X2 := X1#[Index3, Index1, Index2];
DX1 <== DT;
RX1 <== RT;
[RT] computeFFT3(X2, X1);
```

Discussion. To focus on the corner turn, we omit definitions for the functions `computeFFT`*i*`(A1, A2)` which compute 1D FFTs on the *i*th dimension of `A1` (assuming that dimension is not distributed) and leave the result in `A2`, which must have the same distribution as `A1`.

The constant regions `R` and `RT` define the problem space

before and after the corner turn. Because they are bound to different distributions, they are distributed in different ways. The arrays X1 and X2 are bound to variable distributions and regions thus allowing their distributions to be changed in the program.

In the first two lines of the computation, we compute 1D FFTs on the two undistributed dimensions of X1. We then change DX2 and RX2 so that X2 is allocated to hold the corner turn of X1 in a load-balanced way. The remap operator is used to compute the corner turn. To compute the final 1D FFT, we first reallocate X1 for use as the scratch array.

4.3. Sample Sort

Problem. In this problem we want to sort an array of values by partially sorting them into per-processor sets, transferring the different sets to different processors, and locally sorting them. After the local sort is completed, we want to load balance the array so that each processor has a more balanced number of data values.

Solution.

```
const
  p : integer = numLocales();
  G : grid = [auto];
  D : [G] distribution = [blk(1, n)];
  R : [D] region = [1..n];
var
  DA : [G] distribution = D;
  RA : [DA] region = R;
  A : [RA] double;
  T : [R] double;
  keys, cuts :
    array[1..p-1] of integer;

[R] determineKeys(A, keys, cuts, p);
[R] T := A;
DA <== [cut(cuts)];
globalSort(T, A);
localSort(A);
DA <=# D;
```

Discussion. The constant region R is the balanced problem space. Array A is initially allocated over R. We first determine p-1 keys which we will use to split the contents of A. Then we can copy the contents of A to temporary array T. By changing DA to use a cut distribution based on the number of values between the keys, array A is reallocated, each processor containing a potentially different number of values. Next, we need to move the data from T to A, in the globalSort procedure, and then locally sort the values in A, in the localSort procedure. These procedures are omitted. Finally, by changing the distribution of A back to D, we can rebalance the load.

4.4. Processor Homogeneity Test

Problem. Given a new machine, we want to see how greatly the processor layout affects an application's performance.

Solution.

```
const
  p : integer = numLocales();
var
  procs : array[1..p] of integer;
  G : grid<..>;

repeat
  permute(procs, p);
  G <== procs;
  -- Run and time program
until /* condition */;
```

Discussion. The solution in ZPL is easy to write and largely independent of the application. The code above indicates what changes would need to be made. Grids need to be arrays of processor IDs so that they can be permuted. The rank of the grids depends on the application. In the code above, we use a 1D array of integers to store processor IDs. Inside a loop, we assign the grid. Since the application will presumably initialize all the data, we can use destructive assignment. If, for some reason (*e.g.*, time-consuming initialization process), data did need to be preserved and shuffled, using preserving assignment would be appropriate.

5. Related Work

The most prevalent "language" currently used for parallel programming is sequential C or Fortran with calls to MPI [9]. Its popularity is primarily due to its generality and portability. In particular, since MPI programmers write in an SPMD style calling low-level send and receive routines, they can express any conceivable data distribution and be confident that their code will run on any parallel platform. The downside to MPI is that it is fundamentally a local-view approach requiring the programmer to manage details of data distribution, parallel computation, and interprocessor communication. This is tedious and error-prone, and it often obscures the logical computation. In contrast, ZPL's grids and distributions provide the programmer with abstractions to express data distribution at a global view, leaving the numerous implementation details to the compiler and runtime.

Global shared address space languages such as Co-Array Fortran [11], Unified Parallel C [1], and Titanium [14] require the programmer to think in an SPMD style as with MPI, but provide abstractions that ease the burden of data distribution and communication somewhat. The specific concepts provided by each language differ and are worth surveying briefly.

Co-Array Fortran (CAF) is a superset of Fortran 90 that supports a new kind of array dimension, the *co-dimension*. The co-dimension provides a logical view of processor space. A variable declared with a co-dimension results in a value with the specified type to be allocated for each image of the executable as described by the co-dimension's range. One common usage is to implement a blocked array decom-

position by declaring an array variable with a co-dimension to represent each processor's sub-array block. Each SPMD image of the program can refer to remote copies of a variable simply by indexing into the variable's co-dimension. This syntax provides a performance model of sorts since all remote references are easily identified by co-dimension indexing. Co-dimension specifications are very similar to ZPL's grid concept since they provide an abstract, indexed view of processor space. Unlike ZPL's grids, co-dimensions are not a first-class concept and do not support the ability to map processors to specific locations within the logical processor set. Moreover, the language's SPMD nature requires the programmer to go to greater effort to implement distributions that are more complex than a simple blocked decomposition—typically this would involve manipulating indices and/or mapping multiple program instances to a single physical processor.

Unified Parallel C (UPC) supports a slightly more global view of parallel programming than CAF. Its array declarations are interpreted as being in a shared memory space that is partitioned across processors, allowing distributed arrays to be declared wholesale rather than replaced into local per-image segments. Array elements are distributed cyclically by default and may also be distributed in a block-cyclic manner by specifying a block size. UPC's distributions take a 1-dimensional view of the array's elements, making it difficult to achieve distributions that are logically multidimensional. UPC currently only supports a single, flat view of the processor set, although users can presumably build their own abstractions using C constructs and explicit mapping. It also bears mentioning that while UPC supports global array declarations and a forall-style loop that partitions iterations among processors, the code is still SPMD and requires a certain degree of explicit communication and synchronization between program images to work correctly and achieve good performance.

Titanium is an SPMD variant of Java that supports a shared address space similar to UPC's. In Titanium, blocked array distributions are typically achieved by allocating an array object in the code, which results in an array per processor as in CAF. A replicated "directory" array is then used to map each processor's index to its local portion of the array. As with CAF and UPC, such idioms support blocked decompositions well but require additional work for users who require more complex distributions. Titanium's object-oriented nature is likely to ease this burden somewhat, since one can imagine users creating processor view classes and data distribution interfaces to implement abstractions like those in ZPL. In doing so, the primary disadvantage is that code must be written in the SPMD style rather than taking a global view of the computation.

OpenMP [4] is an example of a parallel language that supports a global view of computation by assuming a flat memory that is shared by its computational threads. In such a model, processor views, data distributions, and locality have no real consequence, and thus OpenMP does not support concepts that mirror those described in this paper. However it does support a "schedule" clause as part of its for-loop annotation that indicates how loop iterations should be assigned to the implementing threads. The schedules tend to distribute iterations in a blocked style, though the assignment of blocks to threads can be done dynamically to help tolerate dynamic load imbalances. A drawback to OpenMP is that the lack of actual, scalable architectures that support a flat shared memory requires OpenMP programmers to be concerned with locality in order to maximize performance. This has resulted in a common usage pattern today where MPI is utilized for the coarse-grained decomposition of a problem across a cluster's nodes while OpenMP is used to manage the parallelism at each node because of its simplicity and reduced overheads in such a context.

High Performance Fortran (HPF) [8] is another global view language, and one that supports data distributions and a logical processor view. HPF programmers express these concepts in the form of directives that can be used to annotate existing Fortran programs. HPF's data distributions include dimensional block, cyclic, and degenerate distributions similar to those described in Section 3.2. The main difference between these annotations and ZPL's approach is that HPF's directives are not first-class or named. This limits a programmer's ability to modify distributions or processor views in an abstract way and makes it difficult to modularize routines by allowing distributions and processor grids to be passed as parameters. HPF's directives are also not imperative, giving the user an ill-defined implementation model to code against [10]. In particular, whereas particular operators identify communication in ZPL, it is invisible in HPF, making it difficult to reason about. It is worthwhile to mention that although the official HPF support for distributions was fairly modest, research-based versions of the language sought to support more aggressive distributions such as recursive coordinate bisection [13].

6. Conclusions and Future Work

The grid, distribution, and region are powerful abstractions for specifying parallel computation at a high level. They allow the compiler and runtime to manage details of data distribution, communication, redistribution, and reallocation while ensuring that the costs associated with these operations are manifest in the code. The hierarchical and modular nature of the abstractions maintains the global view of the computation by insulating the program from data distribution and keeping separable programmer concerns separate.

Future work will address an evaluation of the perfor-

mance of these constructs. We are also looking into the possibility of user-defined distributions that would provide extensibility.

Acknowledgments

During the time of this work, the first author was supported by a DOE High Performance Computer Science Graduate Fellowship (HPCSGF). The authors would like to thank the many who have contributed to ZPL in the past, and thus made this work possible.

References

[1] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, Center for Computing Sciences, Bowie, MD, May 1999.

[2] B. L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.

[3] B. L. Chamberlain, E. C. Lewis, C. Lin, and L. Snyder. Regions: An abstraction for expressing array computation. In *Proceedings of the ACM International Conference on Array Programming Languages*, 1999.

[4] L. Dagum and R. Menon. OpenMP: an industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January–March 1998.

[5] S. J. Deitz. Renewed hope for data parallelism: Unintegrated support for task parallelism in ZPL. Technical report, University of Washington (2003-12-04), December 2003.

[6] S. J. Deitz, B. L. Chamberlain, and L. Snyder. High-level language support for user-defined reductions. *Journal of Supercomputing*, 23(1):23–37, August 2002.

[7] S. J. Deitz, B. L. Chamberlain, and L. Snyder. The design and implementation of a parallel array operator for the arbitrary remapping of data. In *Proceedings of the ACM Conference on Principles and Practice of Parallel Programming*, 2003.

[8] High Performance Fortran Forum. *High Performance Fortran Specification Version 1.1*, November 1994.

[9] Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4):169–416, 1994.

[10] T. A. Ngo, L. Snyder, and B. L. Chamberlain. Portable performance of data parallel languages. In *Proceedings of the ACM Conference on Supercomputing*, 1997.

[11] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, August 1998.

[12] L. Snyder. *Programming Guide to ZPL*. MIT Press, Cambridge, MA, USA, 1999.

[13] M. Ujaldon, E. L. Zapata, B. M. Chapman, and H. P. Zima. Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(10):1068–1083, October 1997.

[14] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.