ZPL: A Machine Independent Programming Language for Parallel Computers[*]

Bradford L. Chamberlain Sung-Eun Choi[†] E Christopher Lewis
Calvin Lin[‡] Lawrence Snyder W. Derrick Weathersby

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350

[†] Los Alamos National Laboratory
Advanced Computing Laboratory
P.O. Box 1663, MS B287
Los Alamos, NM 87545

[‡]Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Abstract

The goal of producing architecture-independent parallel programs is complicated by the competing need for high performance. The ZPL programming language achieves both goals by building upon an abstract parallel machine and by providing programming constructs that allow the programmer to "see" this underlying machine. This paper describes ZPL and provides a comprehensive evaluation of the language with respect to its goals of performance, portability, and programming convenience. In particular, we describe ZPL's machine-independent performance model, describe the programming benefits of ZPL's region-based constructs, summarize the compilation benefits of the language's high-level semantics, and summarize empirical evidence that ZPL has achieved both high performance and portability on diverse machines such as the IBM SP-2, Cray T3E, and SGI Power Challenge.

Index Terms: portable, efficient, parallel programming language.

[*]This research was supported by DARPA Grant F30602-97-1-0152, a grant of HPC time from the Arctic Region Supercomputing Center, NSF Grant CCR–9707056, and ONR grant N00014-99-1-0402.

1 Introduction

Tools and languages for parallel programming are faced with three goals:

- High performance, because the need for parallelism implies that performance is critical.
- Machine independence across all parallel and sequential platforms, because complex scientific computations must survive many generations of computers.
- Programming convenience, because scientists and engineers have problems that are complex enough without having to overcome language inadequacies.

These goals create a problem because they often conflict. For example, it would be convenient to ignore low level machine specifics, but attention to such details is typically required to achieve good performance. Moreover, high performance and machine-independence are also often at odds. High performance can be achieved by programming close to the hardware to exploit specific machine features, but this sacrifices portability as other machines may provide a different set of features. On the other hand, portability can be achieved by programming far from the hardware at a high-level of abstraction that ignores the specifics of any particular machine, but this sacrifices performance on all machines.

This paper describes one approach to resolving these seemingly conflicting goals. The approach is based on the following conjecture: an effective (high performance, machine independent) parallel programming language cannot be founded on the standard von Neumann model of sequential computation, but rather must be founded on an abstract parallel machine that reflects characteristics of real parallel architectures [43]. Thus, the ZPL programming language was built on the CTA abstract parallel machine model [43], and a scientific goal of ZPL, beyond the three language goals mentioned above, was to test the second half of this conjecture [34, 45].

Several core concepts of ZPL were developed and announced in 1993 [35]. The first portability and performance measurements for the base language were reported in 1994 [36], and these were followed by additional comparisons in 1995 [33]. The language was substantially enhanced over the next two years, and in July 1997, ZPL was packaged with appropriate support software and documentation and released to the public.[1] This paper presents for the first time a comprehensive evaluation of the ZPL language, its compiler, and its scientific approach to achieving high performance, machine-independence, and programming convenience.

[1]Available at `http://www.cs.washington.edu/research/zpl`.

This paper is structured as follows. Section 2 connects the ZPL design with its scientific foundations, and Section 3 describes the primary features and concepts of the language. Section 4 discusses the language's unique WYSIWYG performance model and evaluates ZPL with respect to its design goals. Generalizations and advanced features of ZPL are then covered in Section 5. Section 6 covers related work, and Section 7 presents conclusions.

2 Foundations of ZPL

The problem of designing a successful parallel programming language can be divided into three cases: (1) Use an existing sequential language, (2) augment an existing sequential language, or (3) create a new language. The first approach preserves the value of legacy codes, but forces all concurrency to be discovered by the compiler, a very difficult task for which discouragingly little progress has been made in a decade and a half. Both the second and third approaches devalue the existing software legacy, either requiring that existing programs be revised to incorporate the parallel constructs, perhaps at considerable intellectual effort, or requiring that programs be completely rewritten. There is no effortless solution, and since the second approach may introduce constructs that conflict with sequential semantics, the third approach emerges as the alternative that is most likely to be fully successful. Given the goal of creating a practical portable[2] language, the greatest challenge is in enabling programmers to write machine independent programs despite the wide variability of parallel platforms. Clearly, abstractions are needed to raise the level of expression away from the hardware, but abstractions that prevent compilers from exploiting the best hardware features and avoiding the worst, will greatly degrade performance. Thus, it's important to choose the *right* abstractions, those that map well to the hardware.

A further observation is that to write fast programs, programmers must be able to judge accurately which of various competing solutions expressed in the language is the fastest. The estimate need not be exact, but it must be good enough to select solutions that are dramatically better than the alternatives. In sequential languages such as Fortran and C, programmers estimate performance by imagining how a von Neumann machine would execute the compiled code. The von Neumann machine is not completely accurate, of course, nor does every programmer have a perfect understanding of how the compiler translates the source code into object code, but this crude mechanism is good enough to produce quality machine independent programs.

[2]"Portable" as used here is often expressed as "portable with performance," but this should be considered redundant: All languages seriously proposed for parallel programming are "universal," making their programs portable in a Turing sense, *i.e.*, it is possible for them to run on any platform ignoring performance. Thus, if "portable" is to mean anything nontrivial, it must mean that programs that run well on one platform can run well on any platform.

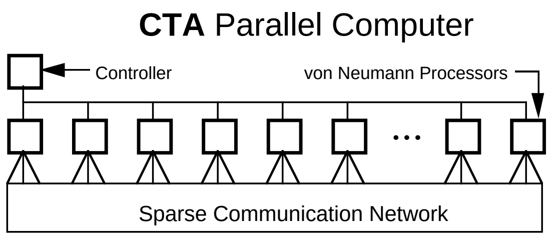


Figure 1: The CTA abstract parallel machine model.

These considerations guided the development of ZPL. Specifically, we adopted analogs to the sequential programming concepts [45]:

	Sequential	Parallel
Machine model	von Neumann	CTA
Programming model	Imperative Procedural	Phase Abstractions
Language	C, Fortran, Pascal, ...	ZPL

The analog to the von Neumann model of computation is the CTA[3] [43] shown in Figure 1. This idealized machine provides the foundation of the ZPL design and implementation, and is visible within the language. Programmers use the CTA to make performance estimates of alternative programming solutions, allowing them to write high quality portable programs [34, 43, 6, 44, 45].

Though the machine model is perhaps the most critical foundational element, the programming model is also crucial. In sequential programming the programming model provides useful abstractions not directly available in the machine. The imperative-procedural model, typified by Fortran, C, Pascal, *etc*., has symbolic naming, data structures, procedures, parameters, recursion, control structures, *etc*. For ZPL, the parallel analog to the imperative-procedural model is the Phase Abstractions programming model [4, 34]. Phase Abstractions provide equivalent features to the imperative procedural model as well as data allocation and processor assignment information [4]. In a nutshell, the programming model defines a scalable unit of parallelism that encapsulates three aspects of parallel computations—code, data and communication—so that performance-critical characteristics of a parallel program can be adjusted for different target machines. For example, the model allows the granularity of parallelism to be easily adjusted. By expressing the code, data and communication topology of parallel programs as independent units, the model encourages component reuse and simplifies the tuning process. Most significantly, the model does not obscure the underlying abstract parallel machine [44].

[3]Acronym for Candidate Type Architecture.

The language binds to the abstractions of the programming model a specific syntax and semantics. Though the remainder of the paper presents ZPL in detail, some of its notable properties are worth highlighting in the context of this foundational discussion:

- ZPL is implicitly parallel, though this is not stipulated by the programming model.
- ZPL has a global view of data, but this should not be confused with a shared memory view, as the Phase Abstractions are a non-shared memory programming model.
- The WYSIWYG feature of ZPL, which indicates to the programmer where and when communication is necessary, is included to ensure the transparency required of the programming model.

Thus, ZPL has interpreted the requirements of the Phase Abstractions programming model in a specific way. Other language designs could interpret it differently. Finally, it should be noted that ZPL is the array subset of a larger language, dubbed Advanced ZPL, which will provide support for general data structures and more general forms of parallelism, such as task parallelism [35].

3 A Slice of ZPL

This section introduces some basic concepts of the ZPL language and explains how these are executed on an abstract parallel machine.

As evident in the simple ZPL program shown in Figure 2, ZPL has many of the same data types and control structures found in standard imperative languages such as C and Pascal. These include boolean, integer, floating point, record, and array data types, and control flow constructs such as `if`, `for`, `while` and `repeat` statements, as well as a typical assortment of arithmetic and logical operators.

Regions

The fundamental concept of the language is the region, which encapsulates parallelism, describes data distribution, and provides concise and readable syntax [16]. A *region* is an index set. For example, the following declaration

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

defines a region `R` that includes the indices in the set $\{$(1,1), (1,2), $\ldots$ (1,`n`)$\ldots$ (`n`,1), (`n`,2), $\ldots$ (`n`,`n`) $\}$. As shown on line 4 of Figure 2, the region bounds can be more involved. Regions may

```
program thinner;

config var m: integer = 10;             -- runtime constants
           n: integer = 20;

region R = [-m/2..+m/2,-n/2..+n/2];     -- declarations

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

procedure skeletonize(var S: [R] integer);
var Obj, T, Temp: [R] integer;
    err: integer;
[R] begin
[east of  R] S := 0;                     -- initialize boundary conditions
[west of  R] S := 0;
[north of R] S := 0;
[south of R] S := 0;

      Obj := S;                          -- copy input image
      repeat                             -- iterate over the thinning algorithm
        Temp := min(min(S@north,S@east),min(S@south,S@west));
        T := Obj + Temp;
        err := max<< abs(S-T);
        S := T;
      until err = 0;

      S := (S>=S@north) & (S>=S@east) & (S>=S@south) & (S>=S@west) & (S!=0);
end;

procedure thinner();                     -- entry procedure
var S: [R] integer;
    objfile: file;
[R] begin
      objfile := open("object.dat", "r");
      read(objfile, S);
      close(objfile);
      skeletonize(S);
      writeln("%d ":S);
end;
```

Figure 2: Sample ZPL program. This program computes the shape skeleton (medial axis) of a two dimensional object that is represented as non-zero pixels. The algorithm takes as input a discretized representation of an object and iteratively performs *volume thinning* [7] by nearest neighbor minimization until only the skeleton remains.

have any static rank, and the upper and lower bounds of each dimension must be integral values. Mechanisms for specifying strided and hierarchical regions are given in Section 5. Once defined, regions can be used to declare arrays, specifying the size and shape of these arrays. Line 8 of Figure 2

```
var Obj, T, Temp:  [R] integer;
```

declares `Obj`, `T`, and `Temp` to be 2-dimensional integer-valued arrays with memory allocated for each index in region `R`.

Regions are also used to specify indices for array operations. Line 15 of the figure

```
[R] Obj := S;
```

shows how the assignment statement extends scalar assignment: Elements of the right-hand side, whose indices are given by the region `R`, are assigned to their corresponding elements on the left-hand side, whose indices are again given by the region `R`. Here, we say that Line 15 is in the *scope* of region `R` because `R` is attached directly to the statement, and hence both the `Obj` and `S` arrays use the same region. In general, an *n*-dimensional region defines indices for all *n*-dimensional array expressions in its scope. Regions also apply to compound statements, so Line 16 defines a scope that applies to the entire body of the `repeat` statement. Finally, regions are dynamically scoped, so procedures can inherit regions from the scope of their call site. As described below, other constructs, such as the reduction operator, require two regions, while expressions involving only scalar variables require none.

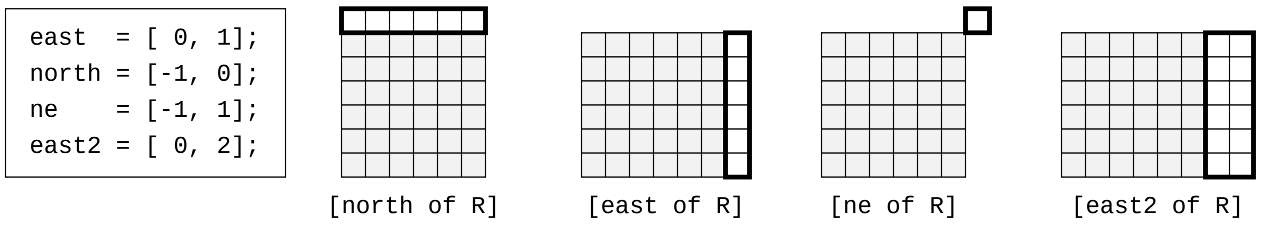
Regions are often named for convenience and clarity, but this is not required. For example, the following lines might be used to assign zeroes to the upper triangular portions of an $n \times n$ array:

```
              for i := 1 to n do
[i,i..n]          A := 0;
```

A slightly more complicated array statement is given on Line 18,

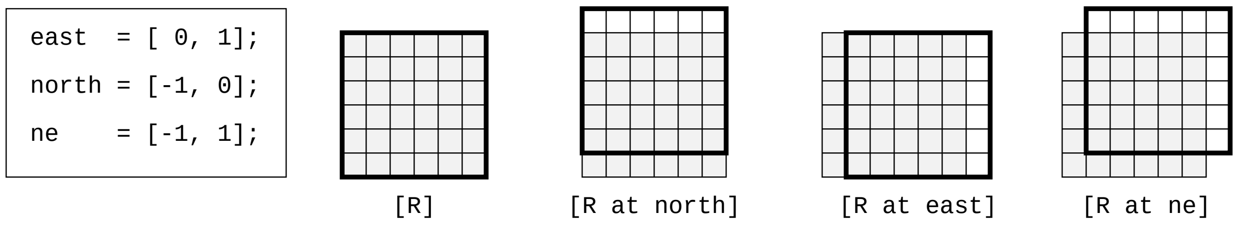
```
T := Obj + Temp;
```

which performs an array addition and then assigns the corresponding sums to elements of `T`. Array addition extends the scalar `+` operator by summing corresponding elements of its array operands, so the result of evaluating `Obj + Temp` is an array of sums. When scalars and arrays both appear as operands in an element-wise operation, the scalars are said to be *promoted* to the rank of the arrays. Other scalar arithmetic and logical operators are similarly extended to operate on arrays. In addition, programmer defined scalar functions may be promoted to perform element-wise computation over arrays, as seen in Line 17's application of the scalar `min` function to array operands (explained below).



```
east  = [ 0, 1];
north = [-1, 0];
ne    = [-1, 1];
east2 = [ 0, 2];
```

Figure 3: Examples of the `of` region operator. Shading represents region `R`, and the outlined area represents the region beneath each example.



```
east  = [ 0, 1];

north = [-1, 0];

ne    = [-1, 1];
```

Figure 4: Examples of the `at` region operator. Shading represents region `R`, and the outlined area represents the region beneath each example.

Region Operators

To simplify the specification of regions, ZPL provides a number of operators that describe new regions relative to existing ones. To use these operators we first define *directions*, which are user-defined vectors that are used with the various region operators. Line 5 shows the definition of two directions,

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

which are used to describe the geometric relationships used in this program.

Once defined, directions can be used with the various region operators such as the `of`, `in`, and `at` operators, which we now describe informally. The `of` operator applies a vector to a region of the same rank and produces a new region that is adjacent to the original region in the direction of the vector. For example in Figure 2, `[north of R]` refers to the row of indices above `R`. Figure 3 shows examples of how the magnitude and signs of the direction vector determine the size and placement of the resulting region.

Lines 11-14 of the example show how boundary conditions are defined. Two points are noteworthy. First, ZPL simplifies memory allocation by implicitly defining storage for boundary values, which are defined to be any portion of an array that is initialized in the scope of a region that uses an `of` operator. Thus, array `S` is allocated sufficient memory to store elements in `[east of R]`, `[west of R]`, `[north of R]`, and `[south of R]`. The second point is that boundary conditions are typically difficult to deal with in scientific computations because they make it impossible to treat the entire computational space uniformly. ZPL provides support for boundary conditions through the use of the `wrap` and `reflect` statements, which can be used to initialize periodic and mirrored boundary conditions. (`Wrap` and `reflect` are are described elsewhere [46].) Furthermore, the use of regions allows boundary condition code to be clearly separated and identified, as evident in Figure 2.

The `in` operator is the dual of the `of` operator, producing a region that is *inside* the original region and adjacent to the border in the direction of the vector. For example, `[north in R]` refers to the first row of indices inside of region `R`.

Finally, the `at` operator translates an index set by some specified vector without changing the size or shape of the original region. For example, the region

```
[R at north]
```

refers to the translation of the `R` region by one in the `north` direction. In general, the `at` operator produces a new region by adding the specified direction to each element of the specified region. Examples are given in Figure 4.

Because computations must often refer to different portions of an array, the `at` operator has an alternate form that can be applied to individual arrays—the only region operator for which this is true. For example, the statement on line 17

```
[R] Temp := min(min(S@north,S@east),min(S@south,S@west));
```

computes, for each point, the minimum of its four neighboring elements. The right-hand side of this statement first uses the built-in `min` function to compare corresponding elements of `S`'s north and east neighbors. Here, correspondence is natural; for example, the upper left-hand elements of arrays `S@north` and `S@east` correspond. The result of this function evaluation is an array whose values are the minimum between `S`'s `north` and `east` neighbors. The remainder of the right-hand side operates analogously, minimizing over the `south` and `west` arrays, and then combining these results into a single array of minimum values.[4] As mentioned earlier, the `min` function is a scalar C function that is promoted to operate on each element of an array expression.

Array Operators

In addition to providing mechanisms for simplifying the specification of array indices, ZPL provides a number of operators, such as reduce and scan operators, that simplify the manipulation of

[4]Lines 17 and 18 would more naturally be combined into a single statement, removing the need to explicitly declare the `Temp` array, but the statement was split for pedagogical reasons. Nevertheless the compiler eliminates such temporaries as described in Section 4.3.1.

arrays. The reduce operators perform an associative, commutative operation, such as + or *max*, on all elements of an array, producing a single scalar. Line 19 shows an example of the max-reduce operator

```
      err := max<< abs(S-T);
```

which takes the largest element of the absolute values of `S-T`, and assigns the result to the scalar variable `err`.

The scan operators apply an associative, commutative operator to an array of values. The resulting value is an array in which each element is the reduction of all preceding elements of the array. Scan operations are also known in the literature as the parallel prefix [29, 9]. The syntax of a scan operator is shown below,

```
      A := max|| B;
```

where we assume that `A` and `B` have been declared to be of the same rank. The syntax of the reduce, scan and flood operators (defined in Section 5) are chosen to remind users that reduce (`<<`) produces a smaller result, scan (`||`) produces a result of the same size and shape, and floods (`>>`) produce a larger result.

Reductions collapse all dimensions of an array to a single scalar, while scans perform the parallel prefix operation across all dimensions of an array. ZPL provides variations of these operations, known as partial reductions and partial scans, which operate over a subset of the dimensions. For example, the following declarations and statement

```
     region IK  = [1..n,1,   1..n];
            IJK = [1..n,1..n,1..n];

     [IK] A := +<<[IJK] B;
```

perform a partial reduction of the `B` array and assign the result to the array `A`. The source region, `IJK`, applies to the `B` array, and the destination region `IK` applies to the `A` array. Here, the region names are chosen to indicate that the region `IJK` includes all indices in a 3-dimensional region, while the region `IK` includes a single plane of the same region. Thus, the source and destination regions combine to indicate that the second dimension is the collapsed dimension.

The most general of ZPL's array operators is the remap operator (#), which permits arbitrary array-level indexing of array data. For example, consider the following statement, where `A`, `B`, `I`, and `J` are 2-dimensional arrays.

```
      [R] A := B#[I,J];
```

The right operand (`[i,J]` in this case) is a sequence of integer arrays that are used to index into the left operand (`B`). In this case, arrays `I` and `J` define the source indices for the first and second dimensions, respectively, *gathering* elements from array `B` to array `A`. When a remap operator appears on the left-hand side of an assignment, a *scatter* is performed. Assignment operators such as `+=` or `*=` are used to accumulate multiple values that scatter to the same index. Otherwise, an arbitrary value persists when there is a conflict.

The remap operator can be used to perform a matrix transpose as follows:

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

Here, `Index1` and `Index2` are built-in ZPL arrays whose elements contain their row index and column index, respectively. These arrays are defined to be conformable with any array, and are generalized to all dimensions that appear in a program.

Overall Program Structure

The overall structure of a ZPL program can be seen in Figure 2. The name of the program is given on the first line. This name is used to define the main procedure, where program execution will begin. The top of the program contains various declarations, and this is followed by a series of procedure definitions.

The first set of declarations specify configuration variables, which are variables whose value is bound once at load time and remain constant thereafter. Configuration variables are useful for specifying values of parameters that are likely to be specific to a given instantiation of a program. These variables can be assigned default values as shown on Lines 2-3, can be assigned values on the command line, or can be initialized through configuration files that are specified on the command line. The next declarations define regions in terms of the configuration variables, and these are followed by the declaration of directions. This program has no global variables, but globals can be defined if necessary.

The main procedure, `thinner()`, declares some local variables, performs I/O to initialize data, and then performs the actual thinning by invoking the `skeletonize()` procedure. The `skeletonize()` procedure illustrates how array parameters may be declared with specific regions defining their size and shape. By contrast, the `read()` and `writeln()` routines can operate on arrays of arbitrary size and shape because they were defined without providing specific regions for their array parameters. Thus, line 27 attaches regions to the call sites of lines 29 and 32, and the dynamic scoping of regions propagates these regions to the bodies of the I/O routines.

Runtime Execution Model

To complete this description of ZPL, we describe its array language semantics and parallel runtime execution model.

As with most array languages, the semantics of array assignment are that the right-hand side is evaluated before it is assigned to the left-hand side, and one array statement is logically completed before the subsequent statement is executed. Each array statement specifies a collection of operations on the elements of the statement's arrays. This collection can logically be performed in any order, which allows the implementation to execute the operations in parallel. Thus, the amount of parallelism in a ZPL program is described by the region attached to each statement. At the same time, these array language semantics allow programmers to reason about ZPL programs as if they were sequential programs.

To achieve parallelism, arrays are distributed across processors based on the distribution of their defining regions. The distribution of regions is restricted to obey the invariant that *interacting regions* are distributed in a *grid-aligned* fashion.

Two regions are considered to be interacting when they are both explicitly or implicitly referenced in a single statement. Explicit region references are those encoded in array operators (*e.g.*, partial scans and reductions) and those that specify the indices of an array statement. Implicit region references are those that are used to declare the arrays appearing in the statement. For example, the following statement (Figure 2 line 14) implicitly references region `R` because array `S` is declared over `R` and explicitly references region `south of R`, so they are interacting regions.

```
[south of R] S := 0;
```

Grid-aligned means that if two *n*-dimensional regions are partitioned across a logical *n*-dimensional processor grid, both regions' slices with index *i* in dimension *d* will be mapped to the same processor grid slice *p* in dimension *d*. For example, since `R` and `north of R` are interacting, they must be grid-aligned, and therefore column *i* of `north of R` must be distributed across the same processor column as column *i* of `R`. Moreover, grid-alignment implies that element (i, j) of two interacting regions will be located on the same processor. This is a key property of our distribution scheme. Note that using a blocked, cyclic, or block-cyclic partitioning scheme for the indices of a set of interacting regions causes the regions to be grid-aligned. Our ZPL compiler uses a blocked partitioning scheme by default, and for simplicity we will assume this scheme for the remainder of this paper.

Once regions are partitioned among the processors, each array is allocated using the same distribution as its defining region. Array operations are computed on the processors containing the elements in the relevant region scopes.

Grid-alignment allows ZPL to provide syntactic cues to indicate where the compiler will generate various forms of communication. In particular, @'s reductions and scans may involve indices that are distributed to different processors. For example, the statement

```
    T := Obj + min(min(S@north,S@east),min(S@south,S@west));
```

refers to both `T` and `S@north`, and the indices that are not common to both regions may belong to different processors.

One final characteristic of ZPL's data distribution scheme is that sequential variables are replicated across processors. Coherency is maintained through redundant computation when possible, or interprocessor communication when not. This coherency-related communication can only be induced by a small number of operations, such as reductions and scalar `read` operations, so this type of communication, as with all communication in ZPL, is visible to the programmer.

4 Evaluation

Section 2 explained ZPL's approach to language design. This section argues that this approach is successful for several reasons. First, it allows programmers to "see" the underlying machine: We will illustrate this by describing ZPL's WYSIWYG performance model and explaining how it allows programmers to reason about performance. Second, it conveniently describes parallelism: We explain the programming benefits of ZPL's region construct. Third, the language's high-level array constructs facilitate the compilation process: We describe various compiler optimizations, including a machine-independent abstraction for specifying communication that can be easily tailored to individual machines. This section also summarizes earlier performance results that indicate that ZPL has achieved its goals of performance and portability, and we explain how the language, compiler and their underlying models combine to achieve this success. We conclude this section by discussing issues of usability, including debugging support for ZPL.

4.1 WYSIWYG Performance Model

The performance of ZPL programs depends both on the mapping of ZPL constructs to the CTA parallel machine model and on the CTA's ability to model real parallel computers. If both of these mappings are straightforward, programmers will be able to reason accurately about the performance of ZPL programs. In this regard, the two significant features of the CTA are that it emphasizes data locality and that it neither specifies nor places importance on the processor interconnection topology. ZPL reflects the CTA's emphasis on locality by its syntactic identification of operators that induce communication. By not including any specification of interconnection topology, ZPL programs can execute using a virtual hypergrid of processors, which is perfectly suited for array computations.

ZPL's data distribution invariant and syntactically-evident communication are also crucial. Previous work [14] showed how they can be used to estimate the relative performance of the various ZPL operators. We now explain how ZPL's syntax and data distribution invariant allow programmers to reason about parallel performance by exposing rather than hiding communication costs. For example, the statement

```
    A := B;
```

is guaranteed never to incur communication because the region that defines `A`'s distribution interacts with the region that defines `B`'s distribution. Thus, `A` and `B` are aligned so that corresponding elements will be located on the same processor, and no communication is needed. In contrast, the statement

```
    A := B@east;
```

may require communication at some point in the program, because the left-hand side and right-hand side refer to different sets of indices. In a blocked decomposition, grid alignment implies that point-to-point communication along the processor rows will be needed.

Moreover, the language's semantics maintain a clean distinction between parallel and sequential constructs. In contrast with most array languages, elements of parallel arrays cannot be manipulated as scalars, which is significant because it disallows statements such as the following

```
    X[i] = Y[j]; /* Not ZPL syntax */
```

which might require communication even though such communication is not syntactically evident. The equivalent statement in ZPL would either use the permutation operator or apply a single region to the entire statement and translate the right-hand side (or the left-hand side) using an @ operator with a direction that represents the difference of `i` and `j`.

It might appear that ZPL's data distribution scheme is too restrictive, forcing programmers to formulate solutions that are amenable to the grid-alignment property. Alternatively, ZPL could allow arbitrary array alignment and indexing, as many languages do [26], but in such a scenario the communication cost of a statement would be a function of both its data access pattern and the alignment of its arrays. This model would be complicated by the fact that a single source-level array (*e.g.*, a formal parameter) might refer to multiple arrays during execution, each with its own alignment scheme. Estimating performance in such a scheme is complex because communication is not manifest in the source code, and the analysis required to locate and evaluate communication requires a global analysis of the source code. This situation has given rise to a number of analysis and interactive parallelization tools [3]. In contrast, ZPL's communication costs are dependent only on the operations within a statement and can therefore be trivially identified.

4.2 Programming Benefits of Regions

Regions in ZPL replace the array subscripting notation found in most other languages [16]. Scalar languages use subscripts to identify individual elements of an array, while array languages such as Fortran 90 [2] and MATLAB [25] extend subscripting to include *slices*, which represent regular subsets of array indices. For example, the following ZPL statement (line 15, Figure 2)

```
[R] Obj := S;
```

could be expressed in Fortran 90 as follows[5]

```
Obj(1:m+1,1:n+1) := S(1:m+1,1:n+1)
```

Regions might seem to be equivalent to slices, but in fact regions provide a powerful abstraction with many programming benefits. We now describe how regions provide notational advantages and support code reuse.

Regions eliminate redundancy. Factoring the common portions of a statement's references (or a compound statement's references) into a single region eliminates the redundancy of subscript specification. As a result, a region-based representation is more concise, easier to read, and less error prone.

Regions can be named. Naming regions becomes practical because they have a broader scope (potentially including multiple statements) than a subscript on a single array reference. By naming regions, programmers can give meaning to index sets. For example, the name `TopFace` is far more illustrative than `(0,0:n-1,0:n-1)`. This same benefit cannot be achieved by providing the ability to name slices (as in APL), because a programmer would potentially have to name a great many slices.

Regions accentuate commonalities and differences. Because the common portions of references are described by the region, all that is left on the array references is an indication of how they differ. This is consistent with the well-known language design principle that similar things should look similar and different things should look different [37]. For example, the following ZPL statement uses four references to array `A`, each shifted in one of the cardinal directions. It is clear exactly how array `A` is being referenced in each operand.

[5]We have adjusted the array bounds to reflect the fact that the indices of Fortran arrays begin at 1.

```
    [1..n,1..n] B := A@north + A@south + A@east + A@west;
```

A subscripted equivalent of this code requires closer scrutiny to discover the same relationship among the operands, let alone verify correctness:

```
  B(2:n+1,2:n+1) = A(1:n,2:n+1)+A(3:n+2,2:n+1)+A(2:n+1,3:n+2)+A(2:n+1,1:n);
```

Regions encode high-level information that can be manipulated by operators. While most languages allow arithmetic operators to be applied to individual dimensions of a subscript, ZPL's region operators are applied to the index set as a whole. The operators encapsulate common forms of reference (shifting, striding, *etc.*), resulting in clearer code. These operators allow regions to be defined in terms of other regions, which is conceptually simpler than repeatedly constructing similar but different index sets. For example, the `of` operator assists in defining and understanding the definition of `TopFace` as `top of cube`. Furthermore, a change to one region is reflected in all regions that are defined in terms of it, thus localizing the changes in the code.

Regions support code reuse. By separating the specification of computation from the specification of array indices, regions produce code that is more general and reusable. For example, regions make it trivial to write statements or procedures that operate on arrays of arbitrary size, while subscripted languages require the programmer to pass around and manipulate array bound information in order to achieve the same generality. Moreover, changing a region-based program to operate on higher dimensional arrays can be a simple matter of changing the region declarations. The array computations themselves may not need to change, or they may need to change in minor and obvious ways, depending on the properties of the computation. In contrast, the use of subscripts or slices would require modifications to every array reference.

4.3 Compilation Benefits

ZPL provides high level semantics that explicitly represent parallel operations. There are many benefits to performing analyses and transformations at this array language level [42, 12]. Some optimizations, such as message vectorization, become trivial [19], and others, such as array contraction, become more effective [31]. Another advantage is the easy identification of communication operations, which facilitates the Factor-Join compilation strategy [12]. This strategy normalizes a program by decomposing it into *factors*, where each factor represents a class of operations that share a common communication pattern. For ZPL the various factors represent pure computation, point-to-point communication, broadcast, global reduce, and global scan. Once factored, a program can be rescheduled, or *joined*, to improve the overall performance of the program. The remainder of this section describes various optimizations in more detail.

4.3.1 Array Contraction

Array temporaries can be a large source of inefficiency because they pollute the cache and increase memory requirements. These temporaries can often be removed through the combination of *statement fusion*—an analog of loop fusion [51] that is performed at the array statement level—and array contraction. The ZPL compiler uses an algorithm that performs such fusion while array statement semantics are still explicit in the internal representation [31]. This approach is advantageous because it is less constrained than traditional loop fusion, and it supports the integration of fusion and communication optimization.

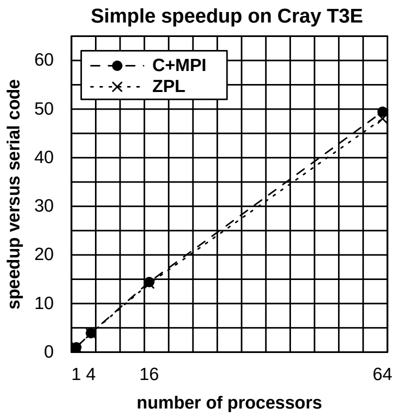
ZPL also facilitates other types of locality-improving transformations such as tiling [50] and padding [41], although such transformations are not currently implemented by our ZPL compiler. Tiling reduces cache misses by iterating over blocks of data that are small enough to fit in the cache, and array padding reduces cache conflict misses by adjusting the layout of arrays so that tiles do not map to the same lines of the cache. As with array contraction, ZPL's region construct helps by providing an unconstrained context in which to perform tiling and padding. Furthermore, ZPL does not allow programmers to access arrays directly through pointers, so such transformations are safe in ZPL, which is not true for languages such as C and C++.

4.3.2 Machine-Independent Communication Interface

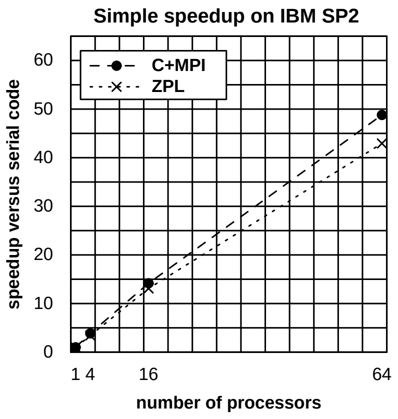
The performance-portability tradeoff is particularly troublesome when performing communication optimizations, as different machines have different low-level mechanisms for achieving the best performance. The ZPL compiler uses the Ironman machine-independent communication interface to provide a separation of concerns [15]. The compiler determines what data to send and when it can legally be sent. Machine-specific libraries then specify *how* to send the data, which allows each machine to use the low-level mechanism that is most suitable. For example, this approach allows the use of shared memory *put* operations on the Cray T3E and non-blocking sends and receives on the IBM SP-2.

4.3.3 Reduce and Scan Optimizations

Reduce and scan are convenient but expensive operators, requiring $O(log(P))$ communication costs and $\frac{N}{P}$ computation costs for P processors operating on an array of size N. Because reduce and



(a)



(b)

Figure 5: SIMPLE performance results on the Cray T3E and the IBM SP-2.

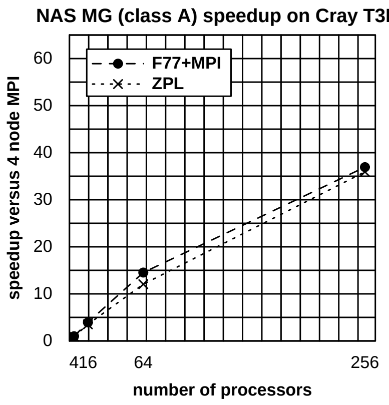
scan are explicit operators in ZPL, compilers can easily identify and optimize them in machine-independent ways. In keeping with the Factor-Join compilation strategy, the ZPL compiler separates the communication and computation components of reduce and scan operations, allowing the communication components of nearby reduce operations to be combined as long as data dependences are obeyed. For example, consider a program that performs a min-reduce and max-reduce on an array A over region R:

```
    [R] lo := min$<<$ A;
    [R] hi := max$<<$ A;
```

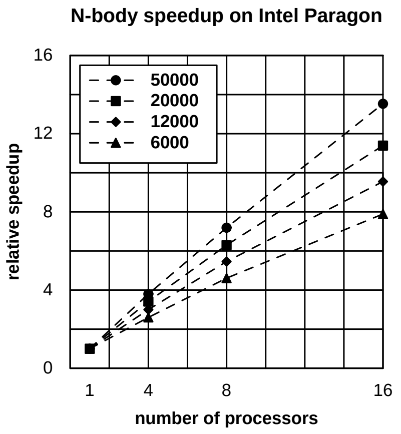
The only difference in the two reductions is the binary operator applied to the elements. The compiler can combine the communication for the two statements, appropriately applying the min and max operators for the respective statements. This essentially halves the communication cost but leaves the computational cost and data volume unchanged. The compiler also performs other optimizations such as overlapping, to hide the latency of the communication, and hoisting, to move invariant components outside of loop nests.

4.4 Empirical Evidence of Success

The performance and portability of the ZPL language have been carefully documented. The first step towards achieving good parallel performance is to achieve good performance on a single node. Experiments on sequential computers have shown that ZPL is competitive—typically within a few percent—with languages such as C and Fortran [36, 32, 20]. Comparisons against hand-coded message passing programs [13, 36] show similar success. For example, Figure 5 shows that for 64



(a)



(b)

Figure 6: (b) NAS Class A MG (Multigrid) performance on the Cray T3E, (a) Hierarchical N-body performance on the Intel Paragon for various numbers of particles.

processors on the IBM SP-2, a ZPL implementation of the SIMPLE fluid dynamics benchmark [21] is about 16% slower than the same program written using C and MPI. On the Cray T3E the ZPL program's performance is almost identical to the MPI version. Similarly, Figure 6(a) show that on the T3E, ZPL performs nearly the same as a Fortran 77 MPI implementation of the NAS MG (Multigrid) parallel benchmark.

Comparisons against other languages are more difficult to characterize. Ngo's thesis [38] uses a subset of the NAS benchmarks to perform an in-depth study of the ZPL and HPF languages and their compilers, in this case our ZPL compiler and three commercial HPF compilers. Ngo draws two conclusions. First, ZPL is more consistent in giving the best performance, while HPF's performance fluctuates from one compiler to another and from one benchmark to another. Second, ZPL's absolute performance and scaling are generally good. These results are consistent with experiments from 1994 where an early version of ZPL was compared against an early HPF compiler on a set of eight small benchmark programs [33], indicating an overall performance advantage to ZPL.

ZPL has also been used by researchers to produce parallel programs for which parallel counterparts in other languages do not exist. In these cases, performance is studied in two steps. First, the ZPL programs are compared on a single processor against sequential implementations, written in either C or Fortran. Then, speedups are computed to show that the programs' performance scaled well as the number of processors grew. For example, Figure 6(b) shows the relative speedup of a ZPL implementation of a hierarchical N-body program that uses Anderson's fast multipole method [5]. This ZPL program [32], written by a civil engineering student as part of a wind engineering simulation, scales well, relative to a ZPL program running on one processor, as the number of particles increases. As a measure of absolute performance, the ZPL program on one processor is about 25% slower than a sequential C implementation for 6000 particles, with the overhead decreasing as the number of particles increased.

Similar studies have been performed for two mathematical biology codes [20]. Both applications scale well, with one, a fibroblast simulation, exhibiting an overhead of roughly 38% over sequential Fortran and the other, a bacteria model, exhibiting a speedup of 6.5% over sequential Fortran. In addition, scientists have used ZPL to implement a parallel version of the CLAWPACK library for solving hyperbolic systems of conservation laws [30], to parallelize a large (10,000 lines of ZPL) synchronous circuit simulator [40], and to perform Lattice Boltzmann simulations to study multiphase flow problems for polymer processing applications [49]. In all of these cases, the ZPL programs were written by scientists and engineers outside of the ZPL group.

In all of these studies, portability has been demonstrated by showing good performance across different machines. The machines have exhibited generational differences as newer machines have replaced older ones, and have exhibited architectural differences in their communication structure and balance of computation and communication speeds. For example, we have shown results for the now-extinct Kendall Square KSR-2, Intel Paragon, the SGI Power Challenge, and Cray T3D, as well as current machines such as the Cray T3E and IBM SP-2. The KSR-2 was a ring-of-rings machine with a cache-only memory structure that provided a single coherent address space. The Intel Paragon was a non-shared memory machine consisting of a 2-dimensional mesh of processors. The Cray T3D and T3E are 3-dimensional tori that provide high bandwidth, low latency and relatively slow processors, while the IBM SP-2 provides much lower bandwidth, higher latency, and fast processors. Finally, the SGI Power Challenge is a cache-coherent non-uniform memory access (CC-NUMA) machine that provides hardware support for accesses to remote data at a fine granularity.

The Role of the Underlying Models. We have explained how various aspects of ZPL provide benefits to the programmer and compiler. We now explain how the CTA abstract machine and the Phase Abstractions programming model contribute to ZPL's overall success. The CTA provides guidance in three ways. First, constructs are only included if they map well to the abstract machine. This is effective because all current and proposed parallel machines can be described by the CTA. Second, the language borrows the CTA's principle of exposing costs. Thus, it is critical that ZPL's communication constructs be syntactically exposed, much as the CTA exposes costs in ways that the PRAM [22] does not. Third, the implementation of the language is phrased in terms of the Phase Abstractions model, so as language features are considered, their benefits can be weighed against their implementation costs and execution costs. The Phase Abstraction programming model is most evident in the design of the compiler, where the model's notion of a distributed data structure, or *ensemble*, has a direct correlation to ZPL's region construct. Together, this integrated approach to language and compiler design leads to constructs that can be efficiently implemented across diverse parallel computers, leads to a language and execution model that supports the WYSIWYG performance model, and, most importantly, leads to the exclusion of numerous constructs that would hinder the goal of portable performance.

4.5 Tools and Debuggers

ZPL's sequential semantics allow programmers to develop and debug their programs on familiar workstation environments before compiling them for production runs on parallel computers. Debugging on sequential platforms is supported by Monash University's zgdb, an extension of the gdb debugger that supports ZPL.[6] Debugging on parallel platforms is supported by GUARD [1, 47], a tool that allows the behavior of a parallel program to be compared against that of a known correct program (which may be a sequential C or Fortran program).

ZPL is not directly supported by any performance analysis tools (the resulting C code can, of course, be analyzed). However, ZPL's transparent performance model reduces the need for such tools; this contrasts with other languages, such as HPF, which *increase* the need for performance analysis tools, as discussed in Section 6.

5 Other Features of ZPL

To paint a more complete picture of ZPL, this section describes generalizations of regions, additional region operators, and a generalization of sequential control flow.

Generalizations of Regions

The notion of regions described in Section 3 have been generalized in several ways. Regions can be regularly sparse (strided regions), have parameterized bounds and strides (multi-regions), and even have replicated dimensions (flood regions).

[6]For more information on zgdb, or to download zgdb, see `http://www.dgs.monash.edu.au/research/guard/gdb/`

Strided Regions. The regions presented in the Section 3 are dense; every element in the index set is present. Regularly sparse regions can be declared by using *striding*. Striding is applied to one or more dimensions in a region declaration using the `by` operator. For example, the statement

```
region sR = [1..n,1..n] by [2,2];
```

declares a 2-dimensional region with indices {`(1,1), (1,3), ... (1,n), (3,1), (3,3), ... (n,n-2), (n,n)`} (for `n` odd). Arrays declared using strided regions only have values and memory allocated for the defined indices. Standard operations (arithmetic, reductions, *etc*.) can be applied to strided arrays. As with dense arrays, an appropriate region must be applied to strided arrays, *i.e.*, elements present in the region must be present in the array.

Multi-Regions. Regions with similar structure can be parameterized and grouped together to form a multi-region, which is a collection of indexable regions with bounds and/or strides that are expressions of the index value. For example, the statement

```
region mR{0..3} = [{}..n+{}];
```

declares a multi-region that is a collection of four regions, `[0..n]`, `[1..n+1]`, `[2..n+2]`, and `[3..n+3]`. When using `mR`, an index is enclosed in curly braces to indicate which region in the collection is to be applied. For example, the following statement uses the region `[1..n+1]` to assign `A`.

```
[mR{1}] A := 1.0;
```

Multi-regions are often coupled with strided regions to create *hierarchical regions*. Hierarchical regions are extremely useful for *multigrid* and *multi-resolution* codes where the granularity of the grid is selectively varied to focus computational effort where it is most needed. The following statements combine to declare a multi-region that becomes more sparse as the index increases.

```
region R = [1..8,1..8];
       hR{0..3} = R by [2^{},2^{}];
```

`hR{0}` is dense because it is strided by 1, `hR{1}` is strided by 2, *etc*. `hR{3}` includes only one index, (1,1). Arrays declared using multi-regions are called multi-arrays. Similarly, *multi-directions* are parameterized directions designed to be used with multi-regions and multi-arrays. The following statement performs the restriction step to move between *levels* in a multigrid computation using multi-regions, multi-arrays, and multi-directions.

```
[hR{i+1}] A{i+1} :=  A{i}/2 + (A{i}@north{i}+A{i}@east{i}+
                                A{i}@south{i}+A{i}@west{i})/8;
```

Multi-regions increase code reuse in multigrid computations. The statement above can be used for restriction at any level of the grid, so no level-specific code is necessary.

Flood Regions. Flood regions omit the indices of certain dimensions to indicate that arrays defined on these regions have replicated data. For example, the following declaration

```
        region fR = [1..n,*];
```

defines a flood region, `fR`, in which the second dimension is omitted. Subsequent arrays declared with such regions are referred to as *flood arrays.* The following declaration

```
        var fA: [fR] integer;
```

defines a 2-dimensional flood array, `fA`, that has `n` rows and an infinite number of columns. The data in the omitted dimensions of the flood region are logically replicated across all indices. In the implementation, only the defining values of each element in the flood dimension exist on any processor. The associated flood operator (`>>`) is used to spread data across the unspecified dimension. The following statement *floods* column `n` of `A`, a dense array, into `fA`, a flood array.

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

Flood arrays can be used to compute the outer product of two vectors. The following example computes the outer product of the first row and the first column of the matrix `A`.

```
        region R   = [1..n,1..n];
               Row = [*,1..n];
               Col = [1..n,*];
        var    A, OP: [R] integer;         -- n x n matrices
               Vr: [Row] integer;          -- flooded row
               Vc: [Col] integer;          -- flooded column

        [Row] Vr := >>[1,1..n] A;          -- flood the first row into Vr
        [Col] Vc := >>[1..n,1] A;          -- flood the first column into Vc

        [R] OP := Vr*Vc;                   -- compute the outer product
```

Other Region Operators

In previous sections, we introduced the `of`, `at`, `in`, and `by` operators. We now describe the two remaining region operators: `with` and `without`.

The `with` and `without` operators are use to perform *masked* computations. A mask is a boolean array used to turn off computation for certain array elements. The following statements perform *red-black* successive over relaxation (SOR) of a 3-dimensional body (we assume that `Red` is a mask that has been initialized to describe a 3-dimensional checkerboard pattern).

```
    [R with Red]    U := f*(hsq*F+U@top+U@bot+U@left+U@right+U@front+U@back);
    [R without Red] U := f*(hsq*F+U@top+U@bot+U@left+U@right+U@front+U@back);
```

Masking restricts the elements over which computation takes place, but it does not change the indices represented by the region. As a result, masking is used only to specify the extent of computation, not to declare arrays.

Generalizations of Control Flow Constructs

Masking is convenient but requires extra memory for storing the mask itself. An alternative is to use generalized control flow constructs. Their use is more restrictive, but their performance is typically better. When a control expression contains an array, the computation is *shattered* such that, conceptually, an independent thread is spawned to perform the computation for each element in the array. All threads implicitly *join* at the end of the control structure. The following statements perform the factorial operation on every element in the array.

```
F := 1;               -- initialize factorial with identity
while A != 1 do
   F *= A;            -- accumulate product
   A -= 1;
end;
```

To prevent deadlock and non-deterministic behavior, the body of shattered control is constrained in various ways. For example, scalars cannot be assigned inside shattered control flow. Details are provided elsewhere [46].

6 Related Work

Apart from ZPL, many other parallel programming languages have been proposed and developed, with similar goals of providing architecture-independent programming. Here we consider some of the main languages and contrast their approaches with that taken by ZPL.

Perhaps the best known language effort for parallel computing is High Performance Fortran (HPF) [26]. HPF was designed by extending the sequential Fortran 90 language to support the distribution of arrays across multiple processors, resulting in parallel computation. Programmers may give suggestions for array alignment and distribution in the form of directives, though their use is optional and they may be ignored by the compiler. This flexibility in implementation has two drastic effects: (1) programmers have no direct means for determining the communication overheads associated with their programs since communication is dependent on data distribution and alignment, and (2) compilers are free to distribute data as they see fit, implying that a program which has been tuned to work well on one platform may perform terribly when compiled on another system. This lack of a performance model in the language is completely antithetical to the notion of portable performance.

Ngo *et al.* demonstrate that HPF's failure to specify a data distribution model results in erratic execution times when compiling HPF programs with different compilers on the IBM SP-2 [39]. To alleviate this problem, tools such as the dPablo toolkit [3] have been designed which give source-level feedback about compilation decisions and program execution. However, these tools are tightly coupled to a compiler's individual compilation model and therefore do not directly aid in the development of portable programs.

NESL [10] is a parallel functional programming language. Its designers recognized that in the parallel realm the ability to reason about a program's execution is crucial, so a work/depth-based performance model was designed to support this task [11]. Although this model matches NESL's functional style well and allows for coarse-grained implementation decisions, it uses a very abstract machine model that reveals little about the mapping of NESL constructs to actual architectures. For example, the cost of interprocessor communication is considered negligible in the NESL model and is therefore ignored entirely.

C$*$ [48] is an extension to the C programming language that was developed for programming the Connection Machine (CM). Several aspects of its design do an excellent job of making the mapping of C$*$ programs to the hardware transparent. For example, the CM architecture supports two general types of interprocessor communication with significantly different overheads—*grid communication* and the more costly *general communication*. This disparity is reflected in the language by its syntactic classification of array references as being either grid or general. Although this does an excellent service for the CM programmer, its benefits are diminished when C$*$ is implemented on different architectures since they may support additional forms of communication with intermediate costs, *e.g.*, broadcasts along subdimensions of a processor grid.

As an alternative to parallel languages, many runtime libraries have been developed to support the creation of portable parallel codes. As libraries, these approaches do not offer the same syntactic benefits as ZPL, and they cannot benefit from the same compiler optimizations that a language can.

The most notable libraries those that provide support for message passing, PVM [8] and MPI [23]. These libraries have been hailed as successes due to their widespread implementation on numerous parallel and sequential architectures, and for the relative ease with which codes written on one architecture can be run on another. However, the libraries are not without their drawbacks. First of all, they put the burden of parallel programming on the users, requiring them to code at a per-processor level and manage all memory and communication explicitly. This is tedious and error prone, and is considered by many to be equivalent to programming sequential computers in assembly language. In addition, the libraries restrict the user to a particular paradigm of communication, which may or may not be optimal for a given architecture [15]. Although extensions to the libraries [24] seek to alleviate this problem by supporting a richer set of communication styles, this does not solve the problem because to achieve optimal performance, a program would have to be rewritten for each machine to use the interface that is most appropriate.

LPARX [28] is a library that supports the parallel implementation of non-uniform problems. LPARX provides user-controlled index sets and a more general version of ZPL's regions that support set theoretic operations, such as union, intersection, and difference. LPARX programmers can specify the distribution of index sets to processors, relying on the runtime system to implement transparent interprocessor communication for non-local array references. LPARX does not provide a WYSIWYG performance model.

HPC++ [27] extends C++ by providing class libraries to support both task and data parallelism. HPC++ uses a parallel implementation of the C++ Standard Template Library to provide parallel container classes and parallel iterators, and HPC++ uses pragmas to identify parallel loops. HPC++ also provides support for multithreaded programming. In short, HPC++ supports task parallelism and a wider range of data structures via lower-level mechanisms than those in ZPL.

7 Conclusion

This paper has explained how the ZPL programming language provides architecture-independence, high performance, and programming convenience for data parallel applications. We have explained how this language was founded on an abstract parallel machine, and we have argued that the relationships between the language, its compiler, and their underlying programming model were central to this success. Finally, we have shown how the notion of regions plays an important role in the ZPL language, both in providing programming convenience and in developing the language's WYSIWYG performance model.

One enabling factor in ZPL's success is its focus on data parallelism. ZPL was designed as a sub-language of the more powerful Advanced-ZPL (A-ZPL) language [35], so ZPL could afford to provide support for arrays at the exclusion of other data structures. As a consequence, ZPL is not ideally suited for solving certain types of dynamic and irregular problems. (Of course, as a Turing complete language, ZPL is not restrictive in any fundamental sense.) We are using the lessons learned from ZPL to guide the design of A-ZPL. For example, we have already demonstrated ways by which A-ZPL can extend the notion of regions to support sparse computation [17] and pipelined wavefront codes [18]. Furthermore, we envision that A-ZPL will support richer data structures, as well as task parallelism and irregular parallelism.

Acknowledgments. We thank the anonymous referees for their helpful comments.

References

[1] D. A. Abramson, I. Foster, J. Michalakes, and R. Sosic. Relative debugging: A new paradigm for debugging scientific applications. *Communications of the ACM*, 39(11):67–77, November 1996.

[2] 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.

[3] Vikram S. Adve, Jhy-Chun Wang, John Mellor-Crummey, Daniel A. Reed, Mark Anderson, and Ken Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Supercomputing '95*, December 1995.

[4] Gail Alverson, William Griswold, Calvin Lin, David Notkin, and Lawrence Snyder. Abstractions for portable, scalable parallel programming. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):1–17, January 1998.

[5] Christopher R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM Journal of Sci. Stat. Computing*, 13(4):923–947, July 1992.

[6] Richard J. Anderson and Lawrence Snyder. A comparison of shared and nonshared memory models of parallel computation. In *Proceedings of the IEEE*, volume 79, 4, pages 480–487, 1991.

[7] D.H. Ballard and C.M. Brown. *Computer Vision*. Prentice-Hall, 1982.

[8] Adam Beguelin and Jack Dongarra. *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1994.

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

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

[11] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[12] Bradford Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. Factor-Join: A unique approach to compiling array languages for parallel machines. In David Sehr, Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 481–500. Springer-Verlag, 1996.

[13] Bradford Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. The case for high level parallel programming in zpl. *IEEE Computational Science and Engineering*, 5(3):76–86, July-September 1998.

[14] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL's WYSIWYG performance model. In *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 50–61. IEEE Computer Society Press, March 1998.

[15] Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. A compiler abstraction for machine independent communication generation. In *Languages and Compilers for Parallel Computing*, pages 261–76. Springer-Verlag, August 1997.

[16] Bradford L. Chamberlain, E Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: An abstraction for expressing array computation. In *ACM SIGAPL/SIGPLAN International Conference on Array Programming Languages*, pages 41–9, August 1999.

[17] Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. A region-based approach to sparse parallel computation. Technical Report UW-CSE-98-11-01, University of Washington Department of Computer Science and Engineering, November 1998.

[18] Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. Language support for pipelining wavefront computations. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1999.

[19] Sung-Eun Choi and Lawrence Snyder. Quantifying the effect of communication optimizations. In *Proceedings of the International Conference on Parallel Processing*, pages 218–222, August 1997.

[20] Marios D. Dikaiakos, Calvin Lin, Daphne Manoussaki, and Diana E. Woodward. The portable parallel implementation of two novel mathematical biology algorithms in ZPL. In *9^{th} International Conference on Supercomputing*, pages 365–374, 1995.

[21] Kattamuri Ekanadham and Arvind. SIMPLE: Part I, an exercise in future scientific programming. Technical Report 273, MIT CSG, 1987.

[22] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

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

[24] MPI Forum. MPI standard 2.0. Technical report, http://www.mcs.anl.gov/mpi/ (Current on October 13, 1997).

[25] Duane Hanselman and Bruce Littlefield. *Mastering MATLAB*. Prentice-Hall, 1996.

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

[27] E. Johnson, D. Gannon, and P. Beckman. HPC++: Experiments with the parallel standard template library. In *Proceedings of the 11th International Conference on Supercomputing (ICS-97)*, pages 124–131, New York, July 1997. ACM Press.

[28] Scott R. Kohn and Scott B. Baden. A robust parallel programming model for dynamic non-uniform scientific computations. Technical Report CS94-354, University of California, San Diego, Dept. of Computer Science and Engineering, March 1994.

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

[30] Randall J. LeVeque and Derek S. Bale. Wave propagation methods for conservation laws with source terms. In *Procedings of the 7th International Conference on Hyperbolic Problems*, February 1998.

[31] E Christopher Lewis, Calvin Lin, and Lawrence Snyder. The implementation and evaluation of fusion and contraction in array languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–59, June 1998.

[32] E Christopher Lewis, Calvin Lin, Lawrence Snyder, and George Turkiyyah. A portable parallel n-body solver. In D. Bailey, P. Bjorstad, J. Gilbert, M. Mascagni, R. Schreiber, H. Simon, V. Torczon, and L. Watson, editors, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 331–336. SIAM, 1995.

[33] C. Lin, L. Snyder, R. Anderson, B. Chamberlain, S. Choi, G. Forman, E. Lewis, and W. D. Weathersby. ZPL vs. HPF: A comparison of performance and programming style. Technical Report 95–11–05, Department of Computer Science and Engineering, University of Washington, 1994.

[34] Calvin Lin. *The Portability of Parallel Programs Across MIMD Computers*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1992.

[35] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 96–114. Springer-Verlag, 1993.

[36] Calvin Lin and Lawrence Snyder. SIMPLE performance results in ZPL. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 361–375. Springer-Verlag, 1994.

[37] Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation and Implementation*. Prentice Hall, Holt, Rinehart, and Winston, 1987.

[38] Ton A. Ngo. *The Role of Performance Models in Parallel Programming and Languages*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1997.

[39] Ton A. Ngo, Lawrence Snyder, and Bradford L. Chamberlain. Portable performance of data parallel languages. In *SC97: High Performance Networking and Computing*, November 1997.

[40] Wilkey Richardson, Mary Bailey, and William H. Sanders. Using ZPL to develop a parallel Chaos router simulator. In *1996 Winter Simulation Conference*, pages 806–16, December 1996.

[41] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 38–49, Montreal, Canada, 17–19 June 1998.

[42] Gerald Roth and Ken Kennedy. Dependence analysis of Fortran90 array syntax. In *Proceedings of the Int'l Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1225–35, August 1996.

[43] Lawrence Snyder. Type architecture, shared memory and the corollary of modest potential. In *Annual Review of Computer Science*, pages I:289–318, 1986.

[44] Lawrence Snyder. Foundations of practical parallel programming languages. In *Proceedings of the Second International Conference of the Austrian Center for Parallel Computation*, pages 115–34. Springer-Verlag, 1993.

[45] Lawrence Snyder. Experimental validation of models of parallel computation. In A. Hofmann and J. van Leeuwen, editors, *Lecture Notes in Computer Science, Special Volume 1000*, pages 78–100. Springer-Verlag, 1995.

[46] Lawrence Snyder. *A Programmer's Guide to ZPL*. MIT Press, Cambridge, Massachusetts, 1999.

[47] R. Sosic and D. A. Abramson. Guard: A relative debugger. *Software Practice and Experience*, 27(2):185–206, February 1997.

[48] *C* Programming Guide, Version 6.0.2*. Thinking Machines Corporation, Cambridge, Massachusetts, June 1991.

[49] Alexander J. Wagner and Chris E. Scott. Lattice boltzmann simulations as a tool to examine multiphase flow problems for polymer processing applications. In *Society of Plastics Engineers Annual Technical Conference (ANTEC'99)*, 1999.

[50] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, pages 30–44, Toronto, Ontario Canada, 26–28 June 1991.

[51] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.