On the Sensitivity of Parallel Programming Languages to the Number and Arrangement of Processors

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
{deitz,bradc}@cray.com

Abstract

Call a parallel program *p-independent* if it always produces the same output on the same input regardless of the number or arrangement of processors on which it is run. This paper introduces and explores the principle of p-independent parallel programming. ZPL is discussed as a parallel programming language with a largely p-independent framework and four *p-dependent* abstractions, two of them new in this paper: free variables, grid dimensions, scatter remaps, and user-defined reductions and scans. These features have the advantage of limiting where and how p-dependent values can be introduced into a program's execution without unnecessarily preventing a programmer from expressing p-dependent algorithms and optimizations. We explore the concept of p-dependence in current programming languages and demonstrate how ZPL's p-dependent features support low-level parallel programming, including calls to MPI.

1 Introduction

Parallel programs are notoriously difficult to write. They require a tremendous effort from the programmer and, when the inevitable errors show up, debugging can easily compromise the productivity of even the most seasoned programmer. When a program produces the correct results on one processor or a few processors, but fails to work correctly on many processors—perhaps hundreds or thousands—the reason can often be difficult to ascertain. Moreover, the bug may not show up until years after the program was written. An interesting case of this is the NAS MPI reference implementation of the CG benchmark, which contained a bug that only showed up when run on 1024+ processors and was not found until April 2003, more than five years after it was released [5]. In order to maximize the productivity of the parallel programmer, it seems prudent for our community to better understand where such processor-set-dependent bugs can occur and to minimize the number of such occurrences as much as possible.

Call a parallel program *p-independent* if and only if it always produces the same output on the same input regardless of the number or arrangement of processors on which it is run; otherwise call it *p-dependent*. For example, the canonical "hello world" program is trivially a p-independent program assuming it prints out the "hello world" message exactly once regardless of how many processors are executing it. Likewise, a program that prints out the number of processors on which it is being run is trivially a p-dependent program.

In this paper, we call a programming language in which only p-independent programs can be expressed a *p-independent programming language*. Similarly, we'll refer to language abstractions that cannot introduce p-dependent values into a program's execution *p-independent programming abstractions*. In our discussion, we choose to ignore three important cases in which p-dependent values can be introduced in practice: round-off errors caused by reordering floating-point arithmetic operations, non-deterministic routines involving timing or external machine state, and resource exhaustion issues such as out-of-memory errors stemming from finite memories.

P-independent programming languages are easier to use for developing and debugging because the difficulties associated with fragmenting a problem over a set of processors do not exist. For example, race conditions and deadlocks are absent from p-independent programming languages. Moreover, if a program written in a p-independent programming language produces the correct answer on one processor, it will produce the correct answer on any number of

processors. This means that as the number of processors grows, programmers need only concern themselves with the scalability of their algorithm, not its correctness. This allows programmers to spend more effort on parallel algorithm development—load balancing and exposing parallelism—and less on managing the p-dependent details that clutter most parallel codes.

Despite the advantages of p-independent parallel programming languages, they are currently not in vogue and are often considered insufficient. This is not without reason—high performance sometimes demands p-dependent programming. Low-level optimizations that introduce temporary p-dependent values into a program's execution (*e.g.*, array contraction) are sometimes critical. Moreover, some problems (*e.g.*, search) allow for multiple correct solutions and a p-dependent program that computes a different correct solution on different numbers of processors can be significantly faster than a p-independent program that computes the same solution on all numbers of processors.

We believe that an ideal language should provide a largely p-independent programming framework and a small set of p-dependent abstractions. Such a language would support a 90/10 principle in which 90% of a program could be easily expressed with p-independent features while the most performance-critical 10% might require more detailed control from the programmer via p-dependent abstractions. If the language were to support a clear performance model, enabling the programmer to reason about the compiler's implementation (including identification of communication) and a highly-optimizing compiler, the fraction of the program that relied on p-dependent features could potentially be made smaller or eliminated altogether. Debugging programs in such a language would be easier since there would be a limited number of places in the code where the results could vary with the processor set—namely, in those places where the p-dependent features were used.

The productivity advantage of languages that are largely p-independent is enormous. A large fraction of parallel problems being solved by scientists are p-independent by nature. That said, the vast majority of today's parallel programming languages allow programmers to write p-dependent code easily and unwittingly. This disconnect between the restricted class of algorithms that parallel programmers want to write and the unrestricted set of codes that modern programming languages allow them to write results in a great deal of the difficulty associated with parallel programming.

This paper discusses p-dependent extensions that have been implemented in ZPL [22], a high-level parallel programming language that is largely p-independent and has syntactically identifiable communication. The novel contributions of this paper are as follows:

- It introduces the concepts of p-independence and p-dependence and classifies contemporary parallel programming facilities according to these criteria.
- It describes two new programming abstractions that have been implemented in ZPL to support the expression of lower-level, p-dependent computation: the free qualifier and grid dimensions. This is the first time that the concept of the free qualifier has been published, apart from the PhD thesis that introduced it [11].
- It identifies four p-dependent abstractions in ZPL—the free qualifier, grid dimensions, the scatter operator, and user-defined reductions and scans—and argues that the rest of ZPL (the majority of it) is p-independent and retains the associated advantages.
- It demonstrates how ZPL's p-dependent features allow the programmer to write lower-level code, including calls to the MPI library.

It is important to note that the term "processor" in this paper refers to a virtual processor rather than a physical processor. Most parallel programming languages virtualize their processors, often referring to the virtual processors using a different term to make the distinction clear. For example, Co-Array Fortran's images, Titanium's demesnes, ZPL and Chapel's locales, UPC's threads, X10's places, as well as MPI's processors, all refer to the virtual processors on which the program is running, be they threads, processes, or some other implementation. The chief point is that they let the programmer reason about locality within the parallel program. They do not make it any easier to write p-independent programs, however.

This paper is organized as follows. The next section compares and contrasts several modern parallel programming facilities from a p-independent programming perspective. Section 3 presents ZPL's p-independent framework. Section 4 introduces ZPL's four p-dependent abstractions. Section 5 discusses the impact of these abstractions on ZPL and describes possible future work. Section 6 concludes.

2 Related Work

2.1 MPI

MPI [19], today's most widely used parallel programming facility, is based on two-sided message-passing primitives that let programmers *send* and *receive* data between processors. The MPI programmer's fragmented view of computation is inherently p-dependent. The programmer must choreograph the actions of each processor. This is not to say that one cannot write p-independent programs with MPI, only that it is difficult to do so.

MPI has had many successes, allowing programmers to write portable, scalable, high-performance applications. It is generally regarded as being machine-independent, meaning that the same programs produce the same results on different parallel computers, but it is not p-independent. Indeed, MPI is often criticized as being difficult to use, and this is in part because there is little support for writing p-independent programs.

2.2 SHMEM

SHMEM [1] is a communication library that provide support for one-sided messages. Instead of writing the matching sends and receives of MPI, SHMEM programmers only need to write unmatched *puts* or *gets* in order to move data between processors. Though this may ease the burden of the parallel programmer, there is no change to the p-dependent nature of the facility. These abstractions are arguably even more likely to result in p-dependent code since the decoupled semantics require synchronization to be considered separately.

2.3 PGAS Languages

Co-Array Fortran (CAF) [21], Titanium [23], and Unified Parallel C (UPC) [3] are commonly referred to as *Partitioned Global Address Space* (PGAS) languages. Although they employ global address spaces to make it easier to write complex codes, they have fragmented memory models so programmers still must write code on a per-processor basis, much like with MPI and SHMEM. For this reason, these languages do little to ease the difficulty of writing p-independent programs.

UPC arguably does more than the others because it provides a slightly more global view. The ability to declare "shared" distributed arrays is p-independent because the bounds are based on the problem size rather than a quantity such as the problem size divided by the number of processors.

Titanium provides a qualifier for variables called *single*. This qualifier statically enforces that a variable must have the same value on all processors at any given program point. It is similar to the default behavior of regular scalar variables in global-view parallel languages like HPF and ZPL. ZPL's free qualifier (introduced in Section 4.1) is the opposite of single.

2.4 OpenMP

OpenMP [10] is a parallel programming facility for shared memory. It provides directives for parallelizing sequential code written in C, C++ or Fortran. The programmer must understand the sequential code well enough to write correct synchronization and parallelization directives. If the directives are too few or too conservative, the code will not perform as well as it can. If the directives are too aggressive, the results are p-dependent in that the observed incorrectness depends on the underlying parallel execution [17].

2.5 High Performance Fortran

HPF [15] provides data-parallel extensions to Fortran 90. Like OpenMP, it provides directives for parallelizing the code, but unlike OpenMP, it also provides directives for controlling the parallel distribution. It runs into the same p-dependent bugs as OpenMP if the directives are incorrect.

Although OpenMP and HPF are not p-independent, they do not provide p-dependent abstractions for cases when a programmer needs to write per-processor code. The only kind of p-dependent code that can be written in OpenMP and HPF is incorrect code.

2.6 SISAL, NESL, and Id

SISAL [18], NESL [2], and Id [20] are implicitly parallel, functional languages. They are p-independent parallel programming languages. Their semantics are not based on the underlying processors and, unlike in HPF and OpenMP, there can be no p-dependent errors.

A tension in p-independent programming is that an algorithm's communication requirements are often hidden from the programmer and the compiler. Since communication is a major bottleneck in parallel execution, lack of knowledge about communication makes it difficult for a programmer to design a good algorithm or even to choose between two alternatives. It also makes it difficult for the compiler to optimize a code. Note that most of the other facilities we have discussed have this same problem. HPF and OpenMP are more p-independent, but communication is hidden. In HPF, this has historically been a barrier to creating good compilers for the language. Communication is hidden in UPC as well. Indeed, other than MPI and SHMEM, the only facility we have discussed with a transparent view of communication is Co-Array Fortran, a decidedly p-dependent language.

3 ZPL's P-Independent Framework

ZPL provides a largely p-independent framework with a syntactic performance model. Because the framework is p-independent, the programmer cannot make errors that only show up on large numbers of processors. Because the communication is syntactically identifiable, both the programmer and the compiler are keenly aware of the communication induced by any given section of code even though the details of this communication are managed by the compiler and runtime. This section briefly describes this framework; the reader interested in learning more about ZPL is referred to the literature [22, 4, 11].

3.1 Regions and Parallel Arrays

The key abstraction in ZPL is the *region* [7]. A region is an index set with no associated data. Regions are used to declare parallel (distributed) arrays and to control the indices over which a computation is applied. The following code illustrates a simple use of regions:

```
region   R = [1..n  , 1..n  ];
       InR = [2..n-1, 2..n-1];
var A, B, C : [R] float;
[InR] C := A + B;
```

Region `R` is declared to be an $n \times n$ index set, and region `InR` is declared to contain only the interior indices of `R`. The arrays `A`, `B`, and `C` are declared over region `R`. The last line stores the element-wise sums of the interior elements of `A` and `B` in the corresponding positions of `C`.

The region's parallelism is p-independent. The programmer can control how the region is distributed over the processors [14], and because these abstractions are orthogonal to parallel arrays and regions, the semantics of ZPL remain p-independent.

3.2 Parallel Array Operators

One of ZPL's strengths is that there is no communication except where it can be seen in the syntax of the code. Thus the above code is guaranteed to both the programmer and the compiler to be completely parallel. ZPL introduces several array operators that have the distinction of being the only constructs capable of inducing communication. Taken together, they constitute a syntax-based performance model that lets the programmer and compiler reason about where communication occurs in a program and what kind of communication it is [6].

The `@` operator shifts the elements of an array by a *direction*, or offset vector. For example, the following code replaces each of the elements in the interior of `A` with the sum of the four neighboring elements:

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

This operator induces point-to-point communication as the boundary elements are moved between processors. ZPL has seven special parallel operators that induce some form of communication. In this section, we will discuss three more: reduction, prefix reduction or scan, and gather remap.

The *reduce* operator, *op*<<, computes a reduction using one of several built-in associative and commutative operators. For example, the following reductions compute the minimum element in array A and the sum of the rows of array A, respectively:

```
var small : float;
[R] small := min<< A;
[1..n, 1] A := +<< [1..n, 1..n] A;
```

In the partial reduction, the sum of each row is stored in the first column. This operator induces communication that can be implemented using a fan-in tree.

The *scan* operator, *op*||, computes the prefix reductions over one or more dimensions of an array. For example, the following scan computes, at each position in A, the sum of the preceding elements in a row-major order traversal of array A:

```
[R] A := +|| [2, 1] A;
```

This operator induces communication that can be implemented with a parallel-prefix algorithm.

The *gather remap* operator, #, computes a general gather. It remaps the elements in an array based on the indices stored in one or more parallel arrays of integers. For example, ZPL's built-in constant arrays, Index1 and Index2, may be thought of, for the 3×3 case, as

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

implying that the 2D transpose of A in ZPL can be expressed with a gather as

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

The values in the array are simply transposed such that, for all indices i and j, the value in position (i, j) is swapped with the value in position (j, i). This operator induces potentially all-to-all communication. It is the most expensive of ZPL's operators [12].

All of these operators are p-independent. They are orthogonal to both the number and arrangement of processors.

3.3 Scalars and Indexed Arrays

In ZPL, variables other than parallel arrays are replicated and kept consistent on every processor. As processors reach the same program points, the same variables are guaranteed to hold the same values. The compiler ensures this consistency by disallowing codes that would assign potentially different values to the same variables on different processors. For example, assigning values from parallel arrays to scalars is illegal.

ZPL also provides a second kind of array, an *indexed array*, that is more like arrays found in other languages. Indexed arrays are replicated and kept consistent on every processor, and the standard access practice of indexing is allowed. A powerful programming technique lets us assign values from an indexed array to a parallel array by indexing into it with another parallel array. For example, given a parallel array I of index values between 1 and 8, the following code would assign to a parallel array A the values in an indexed array w of weights:

```
var I : [R] integer;
    A : [R] float;
    w : array [1..8] of float;
[R] A := w[I];
```

However, one could not add up the weights in A for the different indices in I and store the results in w using the line [R] w[I] += A; because w would become inconsistent on different processors. Instead, this could be computed using a sum reduction as in the following code:

```
var Temp : [R] array [1..8] of float;
[R] Temp[I] := 0;
```

```
[R] Temp[I] += A;
[R] w[] := +<< Temp[];
```

Note that the use of the blank dimensions on indexed arrays is syntactic sugar for a loop over the entire index range. Also note that a more optimal code, barring compiler optimizations, can be written using ideas introduced in Section 4.1.

3.4 Shattered Control Flow and Interleave

When a control structure is used on a parallel array, the control of the processors shatters, giving rise to *shattered control flow*. For example, in the code

```
[R] if A > B then
      A := A - B;
    else
      A := B - A;
    end;
```

different processors must execute different code depending on their local data. The results are p-independent, however, because the choice is made independent of the processors.

In an interleaved statement block, all the statements' elemental operations are executed in an interleaved manner. For example, in the code

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

the two statements are executed together, *i.e.*, the two sentences are executed for each position in `R` before either is executed for the next position. By using an interleave-block, the programmer is essentially fusing the scalarized implementation. This language-level abstraction thus makes fusion a source-to-source compiler optimization. This makes for an easier compiler implementation and gives programmers control over it should they need or desire such control. The interleave keyword can also be used to implement wavefront computations [8].

4 ZPL's P-Dependent Abstractions

Though ZPL is based on a largely p-independent framework, it provides four abstractions and several intrinsic procedures that may produce p-dependent behavior. This section enumerates the four abstractions.

4.1 Free Variables

As discussed in Section 3.3, variables not declared over a region are replicated and kept consistent on every processor. Alternatively a variable can be declared with the *free* qualifier. Such variables are also replicated on every processor, but their values on different processors are not constrained to be the same at the same program points.

Analogously, free variables can be compared to private variables in shared memory programming systems, being to private variables as ZPL's regular variables are to shared variables. The difference between private and free variables are in the usage rules. Whereas private values can be assigned to shared variables and the programmer must explicitly synchronize, free variables cannot simply be assigned to ZPL's regular variables without the use of one of ZPL's communication-inducing array operators. The advantages to free variables are (1) the programming model is simpler since race conditions and deadlocks are impossible and (2) communication remains visible in the syntax of the code.

There is a subtle difference between free variables and ZPL's regular variables that contain potentially p-dependent values. ZPL provides two intrinsic p-dependent procedures:

```
procedure numLocales() : integer;
free procedure localeID() : integer;
```

The procedure `numLocales()` returns the number of processors a program is using while the procedure `localeID()` returns a unique integer from `0` to `numLocales()-1` to identify each processor. Both procedures return p-dependent values, but only the second procedure returns an free value. Note that it is a compile-time error to assign the result of `localeID` to a regular variable.

The following two examples illustrate the power of free variables.

4.1.1 Example: Array Contraction

Free variables let array programmers implement low-level optimizations such as array contraction. While compilers can reliably contract arrays [16], free variables enable it as a source-to-source transformation. For example, in the following code, programmers need to use a temporary array in order to capture the contents of the expression `A+B` before changing `A` and `B`:

```
var A, B, Temp : [R] integer;
[R] begin
  Temp := A + B;
  A := Temp / A;
  B := Temp / B;
end;
```

In a scalar language, the insertion of a temporary would be trivial, but in array languages, where the temporaries are entire arrays, the amount of introduced storage is substantial. In the following code, the array `Temp` is contracted and replaced by an free variable `temp`:

```
free var temp : integer;
[R] interleave
  temp := A + B;
  A := temp / A;
  B := temp / B;
end;
```

The variable `temp` must be free because during and after the execution of the `interleave` block, the values in `temp` are different on different processors. Note also that the values left in each processor's copy of `temp` are p-dependent because the values that are assigned to a given processor's instance of `temp` depend on the distribution of `A` and `B`. Nonetheless, this transformation preserves the p-independent nature of `A` and `B`.

4.1.2 Example: Full Reduction Decomposition

Free variables let the compiler factor the local computation out of a full reduction. For example, the following reduction computes the sum of an array:

```
var A   : [R] integer;
    sum :     integer;
[R] sum := +<< A;
```

With the additional declaration of free variable `lsum` to store the per-processor sums, the following code computes the same reduction:

```
free var lsum : integer = 0;
[R] lsum += A;
sum := +<< lsum;
```

This transformation is performed by the compiler for all full reductions as a source-to-source optimization. It has the benefit of potentially enabling other optimizations, *e.g.*, fusion and contraction. Because this transformation is now a source-level optimization, the programmer can apply it as well. This may be desirable if the reduction is a more complicated function, but not one that requires a user-defined reduction, or if the compiler cannot make the optimization due to the use of external functions or timing routines.

4.2 Grid Dimensions

Flood dimensions [7, 9] are a p-independent abstraction for array programming. In parallel programming, it is common for lower-rank arrays to be replicated across some processors and distributed across others. For example, when multiplying a 1D vector by a 2D matrix, the 1D vector may be distributed across one dimension and replicated across the other. Flood dimensions, indicated by an asterisk, specify a replicated dimension. For example, a vector `V` can be replicated across the rows and distributed across the columns of a corresponding matrix `A` by declaring the first dimension of `V` to be a flood dimension as in the following code:

```
var V : [*, 1..n] float;
```

There is one vector of values. It is replicated and kept consistent across the processors. The language ensures this consistency by restricting how values can be assigned to arrays with flood dimensions.

Grid dimensions can analogously be called free flood dimensions, being to flood dimensions as free variables are to regular variables. Denoted by `::` rather than `*`, a *grid dimension* associates a single value with *each* processor that the grid dimension is distributed over rather than a single value for *all* processors that a flood dimension is distributed over. Whereas the flood dimension values are replicated and kept consistent on the processors over which it is distributed, grid dimensions values are only replicated on the processors over which it is distributed.

Grid dimensions can be used to exact a per-processor view into a parallel array. When a parallel array with a normal dimension is read or written over a grid dimension, the parallel array can be treated as if it was a parallel array of an indexed array declared over the grid dimension. For example, given an array

```
var A : [1..n, 1..n] integer;
```

that is read over the region `[1..n, ::]`, it is treated as if it was declared as

```
var A : [1..n, ::] array [low..high] of integer;
```

where `low` and `high` depend on the processor and the array's distribution. As we will see in the second example, this is a powerful mechanism for writing per-processor code even while the overall program is p-independent.

4.2.1 Example: Partial Reduction Decomposition

Grid dimensions let programmers factor the local computation out of a partial reduction in the same way that free variables let programmers factor the local computation out of a full reduction. For example, the following partial reduction stores the sum of each row of array `B` in the first column of array `A`:

```
var A, B : [1..n, 1..n] integer;
[1..n, 1] A := +<< [1..n, 1..n] B;
```

With the additional declaration of `T`, which is used to store the per-processor sums of each row, the following code computes the same partial reduction:

```
var T : [1..n,  :: ] integer = 0;
[1..n, 1..n] T += B;
[1..n, 1   ] A := +<< [1..n, ::] T;
```

This transformation is similar to what the compiler does for all partial reductions and has the same benefits as the transformation on full reductions that was discussed earlier. Note that a similar transformation applies to scans. For example, the following codes are equivalent:

```
[1..n, 1..n] A := +|| [2, 1] B;
```

and

```
[1..n,  :: ] T := 0;
[1..n, 1..n] T += B;
[1..n,  :: ] T := +|| [2, 1] T;
[1..n, 1..n] interleave
               A := T;
               T += B;
             end;
```

4.2.2 Example: Parallel Text I/O

Grid dimensions let programmers write their own implementations of parallel text I/O routines. Because of the difficulty of determining how many bytes of data will be printed before a given position in the array is reached, this computation is non-trivial. For example, the following code uses the sprintf, fseek, and fprintf functions of C to implement efficient parallel text output to a file:

```
var f         :              file;
    A         : [1..n, 1..n] float;
    RowBytes,
    Offset    : [1..n,  :: ] integer;
free var str  :              string;
[1..n,  :: ] RowBytes := 0;
[1..n, 1..n] RowBytes += sprintf(str, "%f ", A);
[1..n,  :: ] Offset := +|| [2, 1] RowBytes;
[1..n,  :: ] interleave
               fseek(f, Offset, SEEK_SET);
               fprintf(f, "%f ", A[]);
             end;
```

In the first two lines of code, `RowBytes` is used to calculate the number of bytes that each processor will write for any given row. The scan operator is then used to determine `Offset`, the number of bytes that will be written prior to the start of each row on each processor. In the final interleaved statement block, the data is written to a file starting at the position in the file specified in `Offset`. The scan operator is ideal for the parallel computation and the grid dimensions make it efficient in memory $O(n * p)$.

Note that line feeds could be printed at the end of each row if this were accounted for in the computation of `RowBytes` before the scan using the following line of code:

```
[1..n, n] RowBytes += 1;
```

4.3 Scatter

In Section 3.2, ZPL was shown to support a general gather operation by applying the remap operator to a parallel expression on the right-hand side of a statement. By applying the remap operator to a parallel array on the left hand side of a statement, ZPL supports a general scatter operation. In sharp contrast to the other operators, however, this usage can produce p-dependent values.

In the ZPL statement

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

every value in `A2` is assigned to position (1, 1) of `A1`, and the last value assigned is the one that ends up there. Since there is no order associated with the traversal of the region (as specified), the last assignment is unknown to the programmer and depends on the number of processors and their arrangement. In general, for any many-to-one mapping, the results may be p-dependent. In fact, it is worse since it could produce different results on different executions on the same number and arrangement of processors.

In ZPL, scatters are p-dependent but, by placing certain restrictions on their use, their behavior could be made p-independent. The question posed to the language designer is whether these restrictions unduly hamper performance.

For example, one way to make scatters p-independent is to force the programmer to use an associative and commutative combining assignment operator, such as += or $\star$=, instead of simple assignment. This restriction could be loosened if the compiler can identify a scatter as not resulting in any many-to-one mappings. Another way to make scatters p-independent would be to impose an order on the assignment of the elements. We believe that both of these ideas would substantially lower performance.

4.4 User-Defined Scans and Reductions

ZPL provides a sophisticated mechanism for programmers to define their own scan and reduction operators [13]. These *user-defined* scans and reductions often lead to cleaner, more efficient solutions than can be achieved using only the built-in operators. For example, if a programmer had to find the second smallest integer in an array of integers, a user-defined reduction would be more efficient than a program that relied on the application of two `min` reductions because, in this latter program, the minimum integer would need to be removed from the array before the second `min` reduction could be applied (along with other rigmarole).

While this mechanism is powerful, letting programmers define arbitrary operators for user-defined scans and reductions is also dangerous. User-defined reductions that exhibit p-dependent behavior can be difficult to debug. P-dependent values can emerge for several reasons. If the identity function is defined incorrectly, the programmer could see abnormal behavior. For example, in a user-defined sum reduction, if the identity were defined to be one instead of zero, the result would be the sum plus the number of processors. Other p-dependent values could be produced if the reduction is incorrectly said to be associative or commutative or if the accumulator and combiner are not computing the same thing (*e.g.*, the accumulator for a sum is specified with the combiner for a product).

In some cases, a p-dependent implementation of a user-defined reduction may improve performance. For example, in a reduction that determines the position of the minimum element in an array, the programmer may not be concerned with which position is returned if there are multiple instances of the minimum value. In general, the code to resolve such arbitrary choices may induce excessive overhead.

5 Discussion

In the previous section, we described the four features of the ZPL language that could result in p-dependent behavior. Two of these have been supported in ZPL for some time: the scatter assignments which can lead to unpredictable (yet occasionally desirable) results in the presence of many-to-one mappings; and the user-defined scans and reductions which can be written to produce results dependent on the number and arrangement of the processors. The other two features—free variables and grid dimensions—are new concepts that have been added to ZPL in order to support lower-level programming than the language has traditionally been able to support. While many compelling algorithms have been expressed in ZPL using its traditional p-independent features [4, 8, 9], the new p-dependent features have enabled many new parallel codes to be written, due to the additional power supported by the features as well as their ability to support lower-level coding for performance-critical kernels [11].

As an example of the low level of control that these new features support, the following program demonstrates the use of free variables to call into the MPI library directly from a ZPL program:

```
program MPI_CircularShift;

extern type MPI_Datatype = (MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_COMPLEX, MPI_CHAR);
extern type MPI_Comm = opaque;
extern type MPI_Status = opaque;

extern free prototype MPI_Send(free inout buf : opaque; free count : integer;
                               free datatype : MPI_Datatype; free dest : integer;
                               free tag : integer; free comm : MPI_Comm) : integer;

extern free prototype MPI_Recv(free inout buf : opaque; free count : integer;
                               free datatype : MPI_Datatype; free source : integer;
                               free tag : integer; free comm : MPI_Comm;
                               free status : MPI_Status) : integer;

extern var MPI_COMM_WORLD : MPI_Comm;
extern var MPI_STATUS_IGNORE : MPI_Status;
```

```
procedure MPI_CircularShift();
free var i : integer;
var PrintArray : [::] integer;
[::] begin
  if numLocales() < 2 then
    halt("This program requires at least 2 processors.");
  end;
  i := 100 * localeID();
  MPI_Send(i, 1, MPI_INT, (localeID() + 1) % numLocales(), 0, MPI_COMM_WORLD);
  MPI_Recv(i, 1, MPI_INT, (localeID() - 1) % numLocales(), 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  PrintArray := i;
  writeln(PrintArray);
end;
```

The program begins by providing external declarations of MPI datatypes and routines using ZPL's standard features for interfacing with C code. Note the use of the free qualifier to indicate that each parameter to the MPI routines may vary from processor to processor. Next, a free variable is initialized based on each processor's unique ID. Blocking MPI send and receive calls are used to circularly shift the free values between the processors. The values are then assigned to a grid array for the purposes of printing. While this program is admittedly trivial and would be more easily (and efficiently) implemented using the `@` operator, its correct execution serves as a proof-of-concept that larger MPI computations could be embedded into a ZPL program in a similar manner.

It is important to note that without the free variable qualifier, it would be impossible to interface with MPI in this way since any interesting use of MPI requires different processors to supply different values for the actual parameters. A second note is that by interfacing to MPI, the programmer has tapped into a p-dependent behavior that would not be available to them within ZPL, even using its p-dependent features: namely, the MPI code allows the user to specify communication that is outside of ZPL's performance model and therefore to embed communication within shattered control flow. As with the p-dependent languages described in Section 2, the impact is both liberating and a potential source of bugs.

5.1 Future Work

For languages like ZPL in which the p-dependence of a value is statically well-defined, a useful tool would be one that utilized dataflow analysis to compute the flow of p-dependent values through a computation. Recall the manual contraction example of Section 4.1. The free variable `temp` was assigned a p-dependent value, yet the p-dependence of this value does not propagate to `A` or `B` due to the elemental nature of the promoted assignments. If the code erroneously used `begin` rather than `interleave`, the p-dependent values *would* flow to `A` and `B` since the scalar value would be promoted and assigned to the whole array. Our hypothesis is that a tool that detects such presumed mistakes would be useful in keeping p-dependent bugs out of codes by detecting areas where p-dependent values could emerge and identifying them to the programmer.

6 Conclusion

Language abstractions and compiler analysis that support the clear and distinct use of p-independent and p-dependent features have the potential to radically improve the current state of the art in parallel productivity. By guaranteeing to the programmer that a code working on one processor will work on any number of processors or, at least, by limiting the places in a program where this class of parallel bugs can occur, development becomes substantially simpler. This paper has argued that ZPL is a mostly p-independent parallel programming language and has introduced several p-dependent abstractions that provide powerful flexibility.

Acknowledgements During the time of this work, the first author was supported under a DOE High Performance Computer Science Graduate Fellowship (HPCSGF).

References

[1] R. Barriuso and A. Knies. SHMEM user's guide. Technical Report SN-2516, Cray Research Inc., May 1994.

[2] G. E. Blelloch. NESL: A nested data-parallel language (Version 3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, Pittsburgh, PA, September 1995.

[3] 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.

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

[5] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and L. Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[6] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. ZPL's WYSIWYG performance model. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998.

[7] 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.

[8] B. L. Chamberlain, E. C. Lewis, and L. Snyder. Array language support for wavefront and pipelined computations. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1999.

[9] B. L. Chamberlain, E. C. Lewis, and L. Snyder. Problem space promotion and its evaluation as a technique for efficient parallel computation. In *Proceedings of the ACM International Conference on Supercomputing*, 1999.

[10] 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.

[11] S. J. Deitz. *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*. PhD thesis, University of Washington, December 2004.

[12] S. J. Deitz, B. L. Chamberlain, S.-E. Choi, 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.

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

[14] S. J. Deitz, B. L. Chamberlain, and L. Snyder. Abstractions for dynamic data distribution. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2004.

[15] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*. 1997.

[16] E. C. Lewis, C. Lin, and L. Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1998.

[17] G. Matthews, R. Hood, H. Jin, S. Johnson, and C. Ierotheou. Automatic relative debugging of OpenMP programs. In *Proceedings on the European Workshop on OpenMP*, 2003.

[18] J. R. McGraw, S. K. Skedzielawski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2. Technical Report Manual M-146, Lawrence Livermore National Laboratory, March 1985.

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

[20] R. S. Nikhil. Id language reference manual (version 90.1). Technical Report 284-2, Massachusetts Institute of Technology, July 1991.

[21] 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.

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

[23] 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 Workshop on Java for High-Performance Network Computing*, 1998.