Renewed Hope for Data Parallelism: Unintegrated Support for Task Parallelism in ZPL

Steven J. Deitz

Technical Report UW-CSE-03-12-04

Abstract

This paper suggests that data parallelism is more general than previously thought and that integrating support for task parallelism into a data parallel programming language is a mistake. With several proposed improvements, the data parallel programming language ZPL is surprisingly versatile. The language and its power are illustrated by the solution to several traditionally task parallel problems. In addition, limitations are discussed.

1 Introduction

For parallel programming, the coupling of machine and programming models has proven unwise: most scientists today write parallel codes with a sequential language like Fortran and a message passing library like MPI. While the Fortran + MPI combination provides great power, it also has serious drawbacks. By requiring programmers to keep track of what each processor must do as well as what is to be accomplished overall, by introducing the possibilities of deadlock and race conditions, by relying on an architectural concept like message passing rather than a programming abstraction, Fortran + MPI unnecessarily burdens its users. The machine model of messages passing between processors lacks the abstractions of a decent programming model, and this has hampered success.

Instead, success in high-performance scientific computing has stemmed from the development of faster machines and their improved machine models. Flynn [13] classified such machine models by their number of instruction and data streams. The Single Instruction Single Data (SISD) and Multiple Instruction Single Data (MISD) models apply to the uniprocessors found in most workstations and the vector processors found in some supercomputers. From a parallel programming standpoint, the Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) models are more interesting. Both inspired rough parallel programming models: SIMD's *data parallelism* and MIMD's *task parallelism*.

Task parallelism (also seen as control or process parallelism) refers to the parallel composition of sequential tasks which interact through explicit synchronization and communication. It is highly general, but requires a heroic effort from the programmer who must deal with deadlock and race conditions as a matter of course. Communicating Sequential Processes (CSP) [18] is an early notation for parallel programming that epitomizes task parallelism. In CSP, individual sequential processes communicate with each other through input and output mechanisms. CSP is closely

tied to the MIMD machine model of individual processors connected via a network accessed by input and output ports. Notably, communication through updates to global variables is specifically disallowed, though shared memory machine models have inspired this mechanism for task parallelism from time to time. Fortran + MPI is task parallel as well; in this case, the mechanism for communication is a set of library routines for sending and receiving data.

In contrast, data parallelism refers to the sequential composition of parallel operations typically applied to part or all of a large data structure. Notice the crucial distinction between the composition orders: data parallelism is "sequential of parallel" whereas task parallelism is "parallel of sequential." Data parallelism is simpler for the programmer, but lacks the generality of task parallelism. The same is true, of course, with respect to the SIMD and MIMD machine models. However, whereas task parallelism remains tied to the MIMD model, no longer is data parallelism even related to the SIMD model.

The virtual disappearance of SIMD architectures and the widespread acceptance of MIMD clusters of workstations does not necessitate a corresponding decline of data parallelism and rise of task parallelism. In demonstration of data parallelism's versatility, the focus for several recent data parallel languages is the MIMD machine model. For example, the data parallel language ZPL is significantly more advanced than early languages that supported SIMD data parallelism. In ZPL, parallel operations may be complex tasks more easily implemented in the MIMD model. Indeed, data parallelism's "operations" can no longer be interpreted literally; they are more like tasks than operations. Bougé [3] suggests this shift from data parallelism as an offshoot of the SIMD machine model to a programming model for the MIMD machine model is revolutionary:

> Data parallelism can thus be thought as a "Paradigm Revolution" with respect to control [task] parallelism. The essential aspect of this revolution is that the "**PAR** of **SEQ**" execution [machine] model is expressed by a "**SEQ** of **PAR**" programming model! This clear distinction between programming and execution models is the source of an enriched vision of programs.

Playfully referring to constructs in the decidedly task parallel transputer language OCCAM, Bougé abbreviates "sequential" and "parallel" when discussing the composition orders inherent to task and data parallelism.

Despite this revolution, data parallel languages still lack expressive power when compared to task parallel languages. Many powerful task parallel techniques are inexpressible in today's data parallel programming languages. The complimentary qualities, task parallelism's generality and data parallelism's simplicity, have naturally led to efforts to integrate the two models in a single programming language. However, this paper suggests data parallelism is more general than previously thought, and thus this integration is a mistake. With several proposed improvements, the data parallel programming language ZPL is surprisingly versatile. Even though there remains a set of problems that lack data parallel solutions, the set of problems for which data parallelism applies is growing. It is the view of this paper that this trend will continue.

This paper is organized as follows. In the next section, we provide a brief overview of ZPL. In Section 3, we

propose minor extensions to ZPL and demonstrate the resulting ZPL's applicability to several traditionally task parallel problems. In Section 4, we discuss related work. In Section 5, we conclude with a discussion on the limitations of ZPL. Not every task parallel technique can be effectively exploited in ZPL, even with the extensions. However, we argue that further development of ZPL's data parallel capabilities is possible, and that integrating task parallel constructs into ZPL, though easy, would add undesirable complexity.

2 ZPL Overview

ZPL is a portable, high-performance, data parallel array programming language for science and engineering computations. The current implementation is based on a compiler that translates a ZPL program to a C program with calls to a user-specified communication library such as MPI. In this section, we provide only a brief introduction to ZPL. The interested reader is referred to the literature [4, 20].

2.1 Replicated Data Structures

Replicated data structures, which include integers, records, and *indexed arrays*, are, as the name implies, replicated on every processor. (Indexed arrays are similar to the standard arrays provided by many programming languages, but are distinct from ZPL's *distributed data structure*, *parallel arrays*.) By default, replicated data structures are kept consistent on every processor. When different processors reach the same dynamic point in the program, the values contained within replicated data structures are constrained to be the same. Thus, for example, in the code segment

```
config var
  n : integer = 10;
var
  i : integer;
  a : array[1..n] of integer;
...
for i := 1 to n do
  a[i] := i;
end;
```

`a[4]` contains the value 4 on every processor when `i` ≥ 4 on every processor, but on any individual processor, `a[4]` may not yet have been assigned the value 4 even if `i` ≥ 4 on another processor. Note the special declaration of *configuration variable* `n`. Though constant during the execution of a program, configuration variables are not statically known. The default value, 10 in the case of `n`, may be overridden at the start of execution. This is an important feature of ZPL. Less information is required by the ZPL compiler than by compilers for other data parallel languages.

Unconstrained data structures are replicated data structures that may become inconsistent. The compiler ensures constrained data remain consistent in the presence of unconstrained data. If `u` is an unconstrained integer, then the following statements would result in a compile error:

```
(1) i := u;        (2) a[u] := i;        (3) for i := 1 to u do
                                             a[i] := i;
                                           end;
```

In the first statement, the potentially different values on different processors in unconstrained integer `u` could corrupt the consistency of constrained integer `i`. In the second statement, `u` could index into different locations of constrained array `a` on different processors, corrupting the array's consistency. The third statement is a loop which exhibits behavior called *shattered control flow*. Shattered control flow is control potentially executed differently by different processors. In this case, each processor executes the assignment statement `u` times, and a different number of elements in constrained array `a` could be assigned values on different processors. In general, constrained data may not be written to within shattered control flow. Arguably, shattered control flow is similar to task parallelism because of the parallel of sequential composition. It is not task parallelism because communication within shattered control flow is disallowed.

The constrained vs. unconstrained distinction is unique to ZPL, and is worth comparing and contrasting against the more familiar shared vs. private distinction associated with shared memory programming models as well as the recently emerging Unified Parallel C (UPC) language [11, 12]. Both shared and constrained refer to singular values; both private and unconstrained refer to per-processor values. Although shared data is often taken to exist on only one processor whereas constrained data is taken to be replicated across the processors, this view of shared data does not impede its replication for better performance [19, 21]. Rather, the major difference is that the programmer is not restricted by the shared vs. private distinction. The programmer may assign private data to shared data. This results in communication on distributed and distributed shared memory architectures, and synchronization becomes a major factor. In the constrained programming model of ZPL, communication only results from a programmer's use of several communication-inducing operators. Unconstrained data may not be assigned to constrained data.

2.2 Distributed Data Structures

Central to ZPL's parallelism is the *region* [6]. Regions are index sets with no associated data. They serve two fundamental purposes: to declare parallel arrays and to control parallel computations. In contrast to the indexed arrays of Section 2.1, a parallel array is distributed across, rather than replicated on, the processors. In keeping with ZPL's strict rule that there be no communication without the use of several parallel operators which induce it, parallel arrays may only be accessed through regions, i.e., they may not be indexed. Indexing allows for arbitrary access patterns which necessitate communication. In contrast, regions, which must be factored out of expressions, only allow for regular access patterns. The following code illustrates the two purposes of the region:

```
region
  BigR = [0..n+1, 0..n+1];
  R = [1..n, 1..n];
var
  A, B, C : [BigR] integer;
...
[R] C := A + B;
```

We declare a named region `BigR` to contain the indices (i, j) for all i and j such that $0 \leq i, j \leq n + 1$ and a named region `R` to contain only the interior indices of `BigR`. Regions do not require names, and may be inlined directly in the code. We use region `BigR` to declare the arrays `A`, `B`, and `C` and region `R` to assign the interior elements in `C` the sum of the corresponding elements in `A` and `B`.

Four important properties of the region are illustrated by the above code. First, arrays declared over the same region must be distributed across the processors in the same way. Since each of the arrays are declared over region `BigR`, no communication is required. Had `A`, `B`, or `C` been declared over differently distributed regions, the computation would result in an error. The statement would need to be rewritten using a communication-inducing parallel operator. Second, only one region may apply to a statement of expression without a communication-inducing parallel operator. Thus the arbitary access patterns defined by indexing are avoided. Third, the region is passive. It is activated by a statement containing a parallel array, but otherwise does not apply. In the code

```
[R1] begin
  [R2] begin
    S2;
  end;
  S1;
end;
```

region `R1` applies to statement `S1`; region `R2` applies to statement `S2`. The region scope is dynamic, inherited through procedure calls. Lastly, unlike its counterparts in many other data parallel languages, the region need not be statically known by the compiler, though the semantics of mutable regions are as yet undefined. A nascent type theory clearly differentiates the region type from the region.

2.3 Directions and the At Operator

The *at operator*, `@`, is the most basic parallel array operator having the potential to induce communication. It is based on the concept of the *direction*. Directions are offset vectors used to refer to a shifted region of a parallel array. The four cardinal directions, declared as

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

allow a computation in which every non-border element in an array is assigned the average of its four neighbors:

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

If the arrays are block distributed, this common idiom requires communication along the edges of the blocks. With MPI, programmers must treat these edges as special cases, and the code to do so is non-trivial and burdensome. This is a key advantage to ZPL's high-level global view of the computation.

2.4 The Reduction and Flood Operators

The *reduction operator*, `op<<`, reduces the values in a slice of an array (possibly the entire array) to either a lower-rank slice of an array or a scalar. The associative and commutative operator, `op`, resolves collisions in the many-to-one mapping, and may be one of several built-in operators, including `min`, `max`, and `+`, or may be user-defined [8]. If `val` is a constrained integer, then

```
[BigR] val := min<< A;
```

computes the minimum value in `A`. The result is accessible to every processor. Partial reductions provide a convenient way to reduce to a single processor. If `uval` is an unconstrained integer, then

```
[0, 0] uval := max<<[BigR] A;
```

results in only the processor that owns index $(0, 0)$ obtaining the maximum value. Partial reductions require the specification of two regions. They allow the programmer to reduce from and to slices of arrays. To calculate the sums from the rows of array `A` and store the results in the first column of array `B`, a programmer writes

```
[0..n+1, 0] B := +<<[BigR] A;
```

The *flood operator*, `>>`, replicates the values in a slice of an array (possibly a single element) to a higher-rank slice of an array (possibly the entire array). The semantics are nearly the reverse of the reduction operator. Since the mapping is one-to-many, there is no operator to resolve collisions. As an example, the following line of code replicates the value in `A` at index $(0, 0)$ everywhere in array `B`:

```
[BigR] B := >>[0, 0] A;
```

2.5 Flooded and Grid Dimensions

Using the flood operator to replicate data within a parallel array is a powerful programming tool. However, the replicated storage is wasteful. *Flooded dimensions*, `*`, let the programmer specify a minimal storage implementation

at a high level. The elements in a flooded dimension of an array are constrained to have the same value. Each processor owning a piece of that dimension need only store a single copy of the values in that dimension. As an example of using flooded dimensions, consider code in which we multiply an `n` $\times$ `1` column matrix by a `1` $\times$ `n` row matrix. For simplicity, we take the vectors from the first column and row of the `n` $\times$ `n` result, `M`. By declaring two temporary arrays, each flooded in a different dimension, we may write the computation as

```
var
  M : [R] double;
  Col : [1..n, *] double;
  Row : [*, 1..n] double;
...
[1..n, *] Col := >>[1..n, 1] M;
[*, 1..n] Row := >>[1, 1..n] M;
[R] M := Col * Row;
```

Equivalently, we may write this computation without declaring the temporary arrays since the result of the flood operator is a flooded array. The single line of code replacing the three above follows:

```
[R] M := >>[1..n, 1] M * >>[1, 1..n] M;
```

Related to the flooded dimension is the *grid dimension*, `::`. In a grid dimension of an array, each processor stores a different value. The storage requirements are identical to the flooded dimension, but the values are not required to be consistent across the processors. An analogy between these dimension types and the sequential data types is as follows: flooded is to grid as constrained is to unconstrained. There are important differences though. Regions are naturally passive. Recall that if we open a region scope with `begin` and `end`, the region only applies to statements that access parallel arrays. An array flooded in every dimension thus acts differently from a constrained scalar. Further, such arrays do not necessarily exist on every processor, but rather just on those they are distributed across.

The grid dimension provides a powerful mechanism for local view computing. As an example, the Fast Fourier Transform (FFT), due to its complex access patterns, is often computed locally. The common 2D FFT problem involves calculating the FFT on the rows and the columns of an array in sequence. If we do not distribute the rows of an array across the processors, then it is natural to transpose the array between FFT computations on the rows and columns-turned-rows. Given a sequential procedure defined by

```
procedure fft(inout a : array[] of double; in n : integer);
```

we compute the FFT on each row of an array as follows:

```
[1..n, ::] fft(M, n);
```

Viewed over the grid dimension, a parallel array is cast to an indexed array. In a blocked distribution, the indexed array corresponds to the local portion of the parallel array. For blocked-cyclic distributions, the semantics are as yet

undefined. Note that in the above example, each second dimension in its entirety is local to some processor because the second dimension is not distributed.

2.6 Grids and Domains

In ZPL, the distributions of arrays are specified with *grids* and *domains*. A grid specifies the processor layout. Its rank must be statically known, but the number of processors may be specified at runtime. Thus

```
grid
  G = [cp, rp];
```

declares a grid `G` with `cp` column processors and `rp` row processors where `cp` and `rp` are integers which may or may not be statically known. As with the region, the semantics of a mutable grid are as yet undefined.

A domain, which must be associated with a grid, specifies the decomposition (blocked, cyclic, ...) and its bounding box. The only decomposition currently supported is blocked. A domain's rank must be statically known and, as with the grid and region, the semantics of a mutable domain are as yet undefined. The following code creates a domain `D` associated with grid `G` and having a blocked decomposition over the `n` $\times$ `n` index set:

```
domain
  D : G = [blocked, blocked; 1..n, 1..n];
```

As domains are associated with grids, regions are associated with domains. For backward compatibility, but mostly for simplicity, domains and grids do not have to be defined; in this case, per-rank defaults are substituted.

Grids and domains allow the programmer to solve a serious load balancing problem. In a simple program that computes over an $m \times n$ and an $n \times m$ region, it would be a waste of processors to block distribute each array in the same way if m and n differ substantially.

2.7 The Remap Operator

The *remap operator* is a powerful array operator providing mechanisms to support both gather and scatter. As an illustration, consider transposing an $n \times n$ matrix `M`:

```
[R] M := M#[Index2, Index1];
```

The built-in arrays, `Index`i, contain the values of the indices in the ith dimension of the currently active region. In principle, they may be regarded as arrays in which every dimension other than the ith is flooded. In the implementation, no storage is required.

The above code illustrates the gather operation, though the transpose may be written with the scatter operation as well:

```
[R] M#[Index2, Index1] := M;
```

For a more involved discussion of the remap operator, the reader is referred to the literature [9].

The remap operator is typically the best method for copying data between parallel arrays that are associated with different grids or domains. The strict communication rules of ZPL do not allow a programmer to assign the data directly. So if A and B are 2D parallel arrays associated with different domains, the programmer needs to write the slightly obtuse

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

in order to transfer data directly from B to A. This is a reasonable technique because the required communication can be significant.

2.8 Indexed Arrays of Parallel Entities

Parallel arrays are associated with regions which in turn are associated with domains which in turn are associated with grids. These four constructs are the foundations of ZPL's data parallelism. However, they are not particularly special, though some restrictions, e.g. they must be constrained, are necessary.

All four constructs may be declared like normal variables. Thus we allow for indexed arrays of grids, domains, and regions, as well as parallel arrays, as in the following example:

```
var
  Gs : array[1..k] of grid = [,];
  Ds : array[1..k] of domain(Gs[]) = [1..index1*n, 1..index1*n];
  Rs : array[1..k] of region(Ds[]) = [1..index1*n, 1..index1*n];
  As : array[1..k] of [Rs[]] double;
```

This syntax replaces that of multi-regions referred to in the literature [5], though the functionality is similar. In the above code, we declare k 2D grids which distribute the processors evenly in both dimensions. We declare k 2D domains associated with the grids which block decompose index sets uniformly increasing in size. We declare k 2D regions over the domains each the size of the domain's bounding box. Finally, we declare k parallel arrays over the regions.

3 Task Parallelism in ZPL

In this section, we introduce several data parallel constructs to the ZPL language that allow programmers to employ techniques traditionally limited to task parallel languages. A simple data parallel approach often results in an inefficient solution to a problem especially if the size of the data is limited. In this case a better approach may involve the

application of data parallel tasks. In Section 3.1 we illustrate the concept of data parallel tasks with a simple problem whose solution requires a minor extension to ZPL. In Section 3.2 we present a more complicated problem whose solution requires a second extension. In Section 3.3 we present a third problem to illustrate the common idiom in which data parallel tasks are pipelined. Finally, in Section 3.4, we demonstrate the power of ZPL by offering a solution to a problem typically requiring an involved task parallel solution.

3.1 FFT Sum Example

This section introduces the contrived problem *FFT Sum* which illustrates the limitations of basic data parallel techniques. We introduce a simple extension to ZPL's support for grids to overcome these limitations.

3.1.1 Problem

The FFT Sum problem states the following: Given an $n \times n$ array of values, compute the Fast Fourier Transform (FFT) on the rows and on the columns of the array, and take the element-wise sum of these results. Data parallelism naturally applies to this problem. Figure 1(a) illustrates the ZPL solution. The grid and domain ensure the region is distributed only in the first dimension. The FFT's, with their complicated access patterns, may thus be efficiently computed on the rows of the array. Transposing a copy of the array lets us compute the FFT on the columns turned rows. Transposing the copy back, the solution may be obtained. The only communication involved in this algorithm is in the two transposes.

This data parallel solution illustrates the limitations of many data parallel languages. If the number of processors is greater than n, the excess processors are wasted. Increasing n is not always a feasible solution; the problem may require the fixed size of n. Often these issues are ignored in the data parallel world, but problems sizes do not always scale. In the case of this FFT example, the solution of distributing both dimensions is also poor. It is likely that the complicated access pattern of the FFT would increase communication and total running time even if more processors could be applied to the problem.

A better solution to increase parallelism is possible. The column and row FFT's can be run concurrently. This is equivalent to simultaneously executing lines 12 and 13 of the data parallel solution of Figure 1(a). Without extending ZPL, this is difficult.

3.1.2 ZPL Solution

By specifying which processors are associated with a grid, as opposed to just how many processors, we can define grids associated with disjoint sets of processors. In this way, we can achieve the desireable concurrency. This solution

(a) Basic Solution

```
1     grid
2       G = [ ,1];
3     domain
4       D : G = [1..n, 1..n];
5     region
6       R : D = [1..n, 1..n];
7     var
8       A1, A2 : [R] double;


9     procedure fftsum();
10    begin
11      [R] A2 := A1#[Index2, Index1];
12      [1..n, ::] fft(A1, n);
13      [1..n, ::] fft(A2, n);
14      [R] A1 += A2#[Index2, Index1];
15    end;
```

(b) Extended Solution

```
1     grid
2       G1 : [1..P/2; ,1];
3       G2 : [P/2+1..P; ,1];
4     domain
5       D1 : G1 = [1..n, 1..n];
6       D2 : G2 = [1..n, 1..n];
7     region
8       R1 : D1 = [1..n, 1..n];
9       R2 : D2 = [1..n, 1..n];
10    var
11      A1 : [R1] double;
12      A2 : [R2] double;

13    procedure fftsum();
14    begin
15      [R2] A2 := A1#[Index2, Index1];
16      [1..n, ::] fft(A1, n);
17      [1..n, ::] fft(A2, n);
18      [R1] A1 += A2#[Index2, Index1];
19    end;
```

Figure 1: Two ZPL solutions to the FFT Sum problem: (a) basic solution and (b) solution using the grid extension. Assigning each grid a disjoint set of processors in the extended solution enables the concurrent execution of lines 16 and 17. P refers to the number of processors.

is illustrated in ZPL in Figure 1(b). In this solution, we use pairs of grids, domains, and regions. The first grid uses the first half of the processors; the second grid, the second half.

In lines 15 and 18 of the code, all processors are involved. But in line 16, only the first half of the processors compute; in line 17, only the second half of the processors. Lines 16 and 17 are thus run in parallel. The processors not involved in the computation quickly fall through. We thus call this method of achieving more parallelism *fall-through data parallelism*. It is a more general form of data parallelism than would be possible to achieve with many data parallel languages. By calling the code segments in lines 16 and 17 "tasks," this may be called a form of task parallelism. It is limited, i.e. no communication is possible between the tasks.

3.1.3 Discussion

ZPL, therefore, supports what is traditionally considered task parallelism. This prompts the question: Is ZPL task parallel? The answer is resoundingly negative. ZPL is data parallel through and through. Admittedly, it is short-sighted to say that this computation is data parallel because we iterate over the two data arrays in parallel. Iterating over data in parallel is not data parallelism. We must recall the definition of "sequential of parallel." The execution of line 16 is followed by the execution of line 17. The parallelism is achieved by half the processors quickly executing line 16, and the other half quickly executing line 17. Data parallelism is sufficient for this problem; so is task parallelism.

It is interesting, and worthy of mention, to try to determine which solution should be preferred if n is very large.

Unfortunately, this is not easy to do. Lines 12 and 13 of the basic solution may be slower than lines 16 and 17 of the extended solution; the scaling is probably not linear especially if a more involved blocking scheme is used as in the NAS FT benchmark [1]. The main difference, however, stems from the two transposes. In the data parallel solution, it is easy to tell that the all-to-all communication pattern is identical in both transposes. The implementation uses this information to optimize the second transpose resulting in significant time savings [9]. In the task parallel solution, the transposes are not identical. However, in this latter solution the communication pattern is half-to-half rather than all-to-all which is potentially faster. Also, if the functions are to be called multiple times, the optimization of the same transpose executing again, would apply. Determining the better solution requires additional information about the problem.

3.2 Jacobi Sum Example

The preceding problem is often referred to as an example of data parallel tasks which is itself seen as a limited, but common, case of task parallelism. It is an especially simple case of data parallel tasks because there is no data parallel communication within these tasks. In this section, we present a more complicated problem whose efficient solution requires communication within the tasks. We introduce a second extension to ZPL which allows this solution.

3.2.1 Problem

The *Jacobi computation* is a simple data parallel computation. We are given an `n` $\times$ `n` array of values and a small constant `epsilon`. We repeatedly replace each element in the array with the average of its four neighbors until the maximum change seen by any element is less than `epsilon`. The admittedly contrived *Jacobi Sum* problem requires the use of data parallel tasks. In this problem, we run two Jacobi computations on different arrays and calculate the element-wise sum.

3.2.2 ZPL Solution

Figure 2 illustrates a solution to this problem in ZPL with a new construct. Before introducing the new construct, it is worth discussing the problem that results without its use. Ignoring lines 29, 31, 32, and 34, Figure 2 is a typical ZPL with the grid extension. Lines 30 and 33 call the Jacobi routine, but not in parallel. In the Jacobi function, half the processors execute line 22; the other half quickly determine they can continue. However, every processor is involved in the computation of line 23. The variable `delta` must be kept consistent on all the processors.

There are a number of ways to attempt to skirt this issue, but none work. Half the processors cannot skip the entire loop. The problem, in essence, is the prohibition of communication within shattered control flow. Were `delta` declared over a grid region, the `repeat` control structure would be shattered. A limited form of communication in

```
1     grid
2       G1 : [1..P/2; , ];
3       G2 : [P/2+1..P; , ];
4     domain
5       D1 : G1 = [0..n+1, 0..n+1];
6       D2 : G2 = [0..n+1, 0..n+1];
7     region
8       BigR1 : D1 = [0..n+1, 0..n+1]; R1 : D1 = [1..n, 1..n];
10      BigR2 : D2 = [0..n+1, 0..n+1]; R2 : D2 = [1..n, 1..n];
12    var
13      A1 : [BigR1] double; T1 : [R1] double;
15      A2 : [BigR2] double; T2 : [R2] double;

17    procedure jacobi(inout A, T : [,] double);
18    var
19      delta : double;
20    begin
21      repeat
22        T := (A@north + A@east + A@south + A@west) / 4;
23        delta := max<< fabs(T - A);
24        A := T;
25      until delta < epsilon;
26    end;

27    procedure jacobisum();
28    begin
29      on G1 begin
30        [R1] jacobi(A1, T1);
31      end;
32      on G2 begin
33        [R2] jacobi(A2, T2);
34      end;
35      [R1] A1 += A2#[Index1, Index2];
36    end;
```

Figure 2: A ZPL solution to the Jacobi Sum problem. The on construct allows the concurrent execution of lines 30 and 33.

shattered control flow is desireable and would solve this problem. In general, however, it is undesireable, and would result in a complicated, potentially slow implementation in which the performance of a code is difficult to foretell. The on construct balances these needs. Its purpose is to open up a grid dependent scope. Its general form is given by

```
on (grid) begin
  ...
end;
```

Within the construct's dynamic scope, data distributed over a grid other than the grid specified may not be accessed. In addition, constrained variables may not be written, unless it too is declared in the dynamic scope.

3.2.3 Discussion

The goal of the on construct is to allow processors not owning a portion of the specified grid to skip past code, thereby permitting more concurrency. Note that on constructs may not be nested; this would break the rule of only allowing

access to the specified grid within the construct.

The `on` construct is useful for communicating not only with the compiler and machine, but also with the programmer. It signals the possible concurrent execution of distinct code segments. For example, we might use the construct in our solution to the FFT Sum problem in Figure 1(b) even though it is not necessary. It could be used around lines 16 and 17 to indicate the concurrency.

3.3 2D FFT Example

The example described in this section is adapted from the CMU task parallel program suite [10] and based on the NAS FT benchmark [1].

3.3.1 Problem

The *2D FFT* problem states the following: Given a number of iterations, `iters`, and a series of `n` $\times$ `n` arrays, we compute the FFT first on the rows, and then on the columns, of the arrays which are generated over a period of time. As with the previous examples, the problem size, `n`, may be fixed; thus parallelizing only with respect to the data is insufficient. The difference between this example and the FFT sum example from Section 3.1 is non-trivial. In the FFT sum case, two tasks were conceptually spawned and the program continued when each finished. In this example, the tasks must be pipelined. As will be seen, ZPL can handle both of these cases easily. Again, we do not use the `on` construct, though it may be used by the programmer as an informative signal of the increased concurrency.

3.3.2 ZPL Solution 1

Figure 3 illustrates a pipelined solution in ZPL. Note the similarity to the declarations in the task parallel solution to the FFT sum example. The grids, domains, and regions are identical. In addition to the `A1` and `A2` arrays, the `A0` array is declared over region `R1`. As in the NAS FT benchmark, the `A0` array is modified each iteration, and the new modification is used as the next array. In the CMU benchmark suite, the arrays are streamed in from an outside source.

The main computation takes place between lines 17 and 23. Lines 18 to 20 only require the first half of the processors; the other half quickly skip over this computation. In line 21, the first half of the processors must communicate the current values in the array, after the row FFT's, to the second half of the processors. Then, in line 22, the second half of the processors compute the column FFT's. Not shown, the result of the column FFT's is analyzed or passed off to other processors to analyze. When the second half of processors start to compute the column FFT's, the first set quickly move ahead to work on the next iteration. Thus full concurrency is achieved after an initial warmup. If `n` is only 64, then we can effectively use up to 128 processors. However, even more concurrency is exploitable.

```
1     grid
2       G1 : [1..P/2; ,1];
3       G2 : [P/2+1..P; ,1];
4     domain
5       D1 : G1 = [1..n, 1..n];
6       D2 : G2 = [1..n, 1..n];
7     region
8       R1 : D1 = [1..n, 1..n];
9       R2 : D2 = [1..n, 1..n];
10    var
11      A0, A1 : [R1] double;
12      A2     : [R2] double;

13    procedure fft2d();
14    var
15      i :  integer;
16    [R1] begin
17      for i := 1 to iters do
18        evolve(A0);
19        A1 := A0;
20        [1..n, ::] fft(A1, n);
21        [R2] A2 := A1#[Index2, Index1];
22        [1..n, ::] fft(A2, n);
23      end;
24    end;
```

Figure 3: A ZPL solution to the 2D FFT problem in which the iterations are pipelined. After the first iteration, lines 20 and 22 execute concurrently.

3.3.3 ZPL Solution 2

Taking more advantage of the independence between iterations, we implement this example with a group of pipelines, rather than just one. Figure 4 illustrates this approach. In this code, we revert to the generalized form of declaring grids, domains, and regions since we want to declare indexed arrays of such entities. The variable `pipes` refers to the number of pipelines, and need not be statically known. In lines 2 and 3, the processors are carefully divided between the pipelines and between the stages of each pipeline. In lines 4-10, the domains, regions, and arrays are allocated with respect to these grids similarly to lines 4-12 of Figure 3.

Lines 15-22 contain the main computation. Each iteration is assigned to the pipelines in a round-robin manner. The pipelines behave in the same way as described in our discussion of the pipelined implementation. Note that we now must use the remap operator to assign `A0` to the proper element in array `A1` because `A1` may not be on the same grid. In a different implementation, `A0` may be allocated to its own grid or replicated on every processor. Depending on the size of `n`, the time taken by `evolve`, and other problem-dependent factors, either approach may be preferred.

```
1     var
2       G1 : array[1..pipes] of grid = [(index1-1)*(P/pipes)+1..index1*P/pipes/2; ,1];
3       G2 : array[1..pipes] of grid = [index1*P/pipes/2+1..index1*P/pipes; ,1];
4       D1 : array[1..pipes] of domain(G1[]) = [1..n, 1..n];
5       D2 : array[1..pipes] of domain(G2[]) = [1..n, 1..n];
6       R1 : array[1..pipes] of region(D1[]) = [1..n, 1..n];
7       R2 : array[1..pipes] of region(D2[]) = [1..n, 1..n];
8       A0 : [R1[1]] double;
9       A1 : array[1..pipes] of [R1[]] double;
10      A2 : array[1..pipes] of [R2[]] double;

11    procedure fft2d();
12    var
13      i, p : integer;
14    begin
15      for i := 1 to iters do
16        p := (i-1) % pipes + 1;
17        [R1[1]] evolve(A0);
18        [R1[p]] A1[p] := A0#[Index1, Index2];
19        [1..n, ::] fft(A1[p], n);
20        [R[p]] A2[p] := A1[p]#[Index2, Index1];
21        [1..n, ::] fft(A2[p], n);
22      end;
23    end;
```

Figure 4: A ZPL solution to the 2D FFT problem in which the iterations are executed in multiple independent pipelines. Substantial concurrency is achieved with the round-robin (line 16) application of tasks to pipelines.

3.3.4 Discussion

Coarse-grained pipelining is a fundamental technique for achieving greater concurrency in parallel programs. ZPL's support for pipelining is notable for its versatility, but also because no additional constructs are necessary. The remap operator serves the technique well.

3.4 Fock Matrix Construction

The Fock matrix is essential to *ab initio quantum chemistry*: the use of computer programs to compute the fundamental properties of atoms and molecules. The complicated access patterns make the Fock matrix construction algorithm difficult for many data parallel languages to handle [14]. In this section, we discuss a preliminary examination of this algorithm.

3.4.1 Problem

A simplified algorithm similar to those used to construct the Fock matrix is given by the following quadruply nested loop:

```
for i := 1 to n do
  for j := 1 to n do
    for k := 1 to n do
      for l := 1 to n do
        v := compute_integral(i,j,k,l);
        f[i,j] += v*d[k,l];
        f[k,l] += v*d[i,j];
        f[i,k] -= .5*v*d[j,l];
        f[i,l] -= .5*v*d[j,k];
        f[j,k] -= .5*v*d[i,l];
        f[j,l] -= .5*v*d[i,k];
      end;
    end;
  end;
end;
```

The write-only array `f` is the Fock matrix; the read-only array `d` is the density matrix. One major simplification is seen in the loop bounds. Typically the `j`-loop's upper bound is `i`. Likewise, the `k` and `l` loops may also be less than `n`. We ignore such important optimization to simplify our discussion.

The problem encountered by many data parallel languages stems from the array access patterns. As a result, implementations have relied on task parallel constructs. Processors asynchronously make requests to and answer requests from one another for data. In many implementations, the Fock and density matrices are restricted to be sufficiently small so as to allow for full replication. This severely limits the size of the problem that can be handled, but simplifies programming.

3.4.2 ZPL Solution

Figure 5(a) illustrates a ZPL implementation of the quadrupally nested loop in which the Fock matrix, `F`, and the density matrix, `D`, are distributed. In this code, the `k` and `l` loops are parallelized, though this is an arbitrary choice. Lines 15 to 20 correspond exactly to the 6 lines of code in the sequential solution. Lines 15 and 16 are straightforward, the remaining four lines are more involved. For these latter statements, we use the following technique: we transpose the source array thus allowing us to flood and reduce the same dimension. This powerful technique lets us block the computation at runtime.

3.4.3 Discussion

This example illustrates that, given a good set of constructs and techniques, data parallelism is highly-general. A data parallel programmer, armed with the operators used here and the knowledge of general techniques like "transpose to flood and reduce," stands a good chance of quickly writing a high-performance solution to a complex problem. The task parallel programmer may run into problems with deadlocks and race conditions or, due to the generality of the methods, may end up with less efficient solution. The many small messages required by a task parallel solution involv-

```
1     region
2       R = [1..n, 1..n];
3     var
4       F, D : [R] double;

5     procedure fock();
6     var
7       i, j : integer;
8       DT, V, VT : [R] double;
9     [R] begin
10      DT := D#[Index2, Index1];
11      for i := 1 to n do
12        for j := 1 to n do
13          V := compute_integral(i, j, Index1, Index2);
14          VT := V#[Index2, Index1];
15          [i, j] F += +<<[R] (V * D);
16          F += V * >>[i, j] D;
17          [i, 1..n] F -= .5 * +<<[R] (VT * >>[1..n, j] DT);
18          [j, 1..n] F -= .5 * +<<[R] (V * >>[1..n, i] DT);
19          [i, 1..n] F -= .5 * +<<[R] (V * >>[1..n, j] DT);
20          [j, 1..n] F -= .5 * +<<[R] (VT * >>[1..n, i] DT);
21        end;
22      end;
23    end;
```

Figure 5: A ZPL implementation of the simplified Fock matrix construction algorithm. A good communication to computation ratio suggests this solution is efficient.

ing server and worker threads will likely lead to performance bottlenecks on today's machines where the optimized case involves fewer large messages.

4 Related Work

Typically data parallel languages rely on sophisticated compilers while task parallel languages use only standard compilers. The opposite is more often the case for the runtime: task parallel languages employ sophisticated runtimes while data parallel languages use more standard runtimes. In addition, data parallel languages usually provide a global view of the computation; task parallel languages, a local view. Such differences are more than coincidence but less than necessary. They stem from task parallelism's generality and data parallelism's simplicity.

When the differences apply, integrating task and data parallelism into a single language is not only worthwhile, but also difficult. Adding more advanced runtime support to HPF systems would allow for more dynamic behaviors, but this could cost the compiler and result in worse performance. That is why many HPF systems require static knowledge of the problem size and processor grid.

The differences do not apply to ZPL. ZPL uses a relatively small compiler which, for example, is not responsible for parallelizing loops with or without programmer directives. ZPL has a sophisticated runtime allowing problem sizes and grids to change after compilation. In addition, ZPL provides a global view of the computation, but also lets

programmers take a local view of the problem.

In this case, it is useful to make the distinction between integrating task parallelism and extending data parallelism. The approach of extending data parallelism offers the advantage of retaining simplicity, by not permitting the parallel composition of sequential tasks, while still providing substantial generality. Despite this distinction, the work discussed in this paper is similar to the body of work done on integrating task and data parallelism. As Bal and Haines [2] point out in their overview of the topic, the integration efforts have always been incomplete. The end results favor the form of parallelism supported by their base language.

Like the work in this paper, Gross et. al. [16] shy away from fully integrating task and data parallelism. Starting with an implementation of HPF called the Fx compiler, they propose adding task parallel directives to compliment the data parallel directives that form traditional HPF. As a result, task parallelism is not fully supported and is static. The compiler is fully responsible for assigning different subroutines to different groups of processors. Subhlok and Yang [21] continued work on the Fx compiler, removing static constraints and loosening the stranglehold of the compiler.

A notable similarity between the work on Fx and our work is that neither language attempts to provide mechanisms for communication between tasks. The "sequential of parallel" composition order is maintained. Chapman et. al. [7] present the Opus language which achieves task parallelism similarly, though limits it to subroutines. They provide a mechanism, called *shared data abstraction*, for exchanging data between tasks. It allows for full task parallelism, but at the cost of program complexity.

An alternative approach to integrating task and data parallelism is through a coordination language. Foster et. al. [15] define a coordination language for HPF using a subset of MPI. This approach benefits from its simplicity; HPF can remain unchanged. However, the disadvantages are substantial. There is no clear programming model, and the compiler is limited.

The advantage of data parallelism's simplicity is not lost on developers of task parallel languages, and some work has been done to integrate data parallelism into a task parallel language. Hassen and Bal [17] added data parallel constructs to Orca. The data parallelism, though, is very limited. It is typically restricted to a single array, unless multiple arrays are stored in the same object. In addition, the "owner computes" rule is used exclusively.

5 Conclusions

This paper proposed extensions to the ZPL language which enable the expression of techniques traditionally limited to task parallel languages. The resulting language is significantly more general, but some limitations remain. For example, asynchronous producer-consumer relationships for enhanced load balancing are still difficult to express.

Recall the 2D FFT solution of Figure 4 in which we map a series of iterations of a computation to multiple independent pipelines in a round-robin manner. Now suppose that the time needed for the computation to proceed through a pipeline was dependent on the data. Then the round-robin application of tasks to pipelines would result in a possibly inefficient use of the resources. Instead, it would be worthwhile to deal the computations only to idle pipelines. A task parallel language can handle this situation well. ZPL, even with the proposed extensions, cannot.

The set of parallel programming techniques expressible with data parallel languages is growing. The question that needs to be addressed in the future is as follows: Does the sequential of parallel composition order pose fundamental limitations to data parallel programming or can data parallelism rival the expressibility of task parallelism while retaining a simpler programming model? The versatility of ZPL's simple constructs suggests that the latter may be the case. It just may be possible to reword the well-known phrase oft-touted by researchers of artificial intelligence:

> It's *task parallelism* until it works!

References

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical report, NASA Ames Research Center (NAS-95-020), December 1995.

[2] H. E. Bal and M. Haines. Approaches for integrating task and data parallelism. *IEEE Concurrency*, 6(3):74–84, 1998.

[3] L. Bouge. The data parallel programming model: A semantic perspective. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*. Springer-Verlag, 1996.

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

[5] B. L. Chamberlain, S. J. Deitz, and L. Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Proceedings of the ACM Conference on Supercomputing*, 2000.

[6] 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.

[7] B. Chapman, H. Zima, M. Haines, P. Mehrotra, and J. V. Rosendale. Opus: A coordination language for multidisciplinary applications. *Scientific Programming*, 6(4):345–362, 1997.

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

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

[10] P. Dinda, T. Gross, D. O'Hallaron, E. Segall, J. Stichnoth, J. Subhlok, J. Webb, and B. Yang. The CMU task parallel program suite. Technical report, Carnegie Mellon University (CMU-CS-94-131), March 1994.

[11] T. A. El-Ghazawi and F. Cantonnet. UPC performance and potential: A NPB experimental study. In *Proceedings of the ACM Conference on Supercomputing*, 2002.

[12] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC language specification, v1.0. Technical report, February 1999.

[13] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972.

[14] I. Foster. Task parallelism and high performance languages. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*. Springer-Verlag, 1996.

[15] I. Foster, D. Kohr, R. Krishaiyer, and A. Choudhary. Double standards: Bringing task parallelism to HPF via the Message Passing Interface. In *Proceedings of the ACM Conference on Supercomputing*, 1996.

[16] T. Gross, D. R. O'Hallaron, and J. Subhlok. Task parallelism in a high performance fortran framework. *IEEE Parallel and Distributed Technology*, 2(3):16–26, 1994.

[17] S. B. Hassen and H. E. Bal. Integrating task and data parallelism using shared objects. In *Proceedings of the ACM International Conference on Supercomputing*, 1996.

[18] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), August 1978.

[19] H. Lu, A. L. Cox, and W. Zwaenepoel. Contention elimination by replication of sequential sections in distributed shared memory programs. In *Proceedings of the ACM Conference on Principles and Practice of Parallel Programming*, 2001.

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

[21] J. Subhlok and B. Yang. A new model for integrated nested task and data parallel programming. In *Proceedings of the ACM Conference on Principles and Practice of Parallel Programming*, 1997.