The Design and Implementation of a Parallel Array Operator for the Arbitrary Remapping of Data[*]

Steven J. Deitz[‡] Bradford L. Chamberlain[⋆‡] Sung-Eun Choi[§] Lawrence Snyder[‡]

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

[⋆]Cray Inc.
Seattle, WA 98104
bradc@cray.com

[§]Los Alamos National Laboratory[†]
Los Alamos, NM 87545
sungeun@lanl.gov

ABSTRACT

Gather and scatter are data redistribution functions of long-standing importance to high performance computing. In this paper, we present a highly-general array operator with powerful gather and scatter capabilities unmatched by other array languages. We discuss an efficient parallel implementation, introducing three new optimizations—schedule compression, dead array reuse, and direct communication—that reduce the costs associated with the operator's wide applicability. In our implementation of this operator in ZPL, we demonstrate performance comparable to the hand-coded Fortran + MPI versions of the NAS FT and CG benchmarks.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed and parallel languages*

General Terms

Languages

Keywords

parallel programming, gather, scatter, array languages, ZPL

1. INTRODUCTION

Highly-general data remapping operations, like gather and scatter, are noticeably absent from most parallel programming languages. Instead, scientists must rely on low-level mechanisms to redistribute data across processors. In this paper, we introduce the remap array operator, #, that provides power unmatched by the array operators of Fortran 90 and APL. We demonstrate its use and an efficient implementation in the context of ZPL, a parallel array language.

Gather and scatter are of long-standing importance to high performance computing, having been included in Cray Fortran for decades. Being data transfer operations, gather and scatter require a source array, `S`, a destination array, `D`, and a specification of how the elements are to be rearranged. As the names imply, gather describes where a sequence of elements comes from and scatter describes where a sequence of elements goes to. Accordingly, gather can be thought of logically as operating on the right hand side of an assignment statement and so is written with the remap operator as

```
D := S#[<specification of index positions>];
```

Symmetrically, scatter can be thought of logically as operating on the left-hand side of an assignment statement and so is written as

```
D#[<specification of index positions>] := S;
```

The specification of index positions is defined by a sequence of arrays called *map arrays*. For gather, the map arrays must have the same shape as the destination array; for scatter, the source array. In addition, for gather, the number of map arrays required for the remap is the rank of the source array; for scatter, the destination array. In general, if D and S are the destination and source arrays, d and s are the ranks of these arrays, and $M1$, $M2$, ... are the map arrays, then gather remap implements

$$D_{i_1,i_2,\cdots,i_d} := S_{M1_{i_1,i_2,\cdots,i_d},M2_{i_1,i_2,\cdots,i_d},\cdots,Ms_{i_1,i_2,\cdots,i_d}}$$

while scatter remap implements

$$D_{M1_{i_1,i_2,\cdots,i_s},M2_{i_1,i_2,\cdots,i_s},\cdots,Md_{i_1,i_2,\cdots,i_s}} := S_{i_1,i_2,\cdots,i_s}$$

For example, ZPL's built-in constant arrays, `Index1` and `Index2`, may be thought of, for the 3×3 case, as given by

```
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 in ZPL is expressed with either of the following lines:

```
D := S#[Index2, Index1];

D#[Index2, Index1] := S;
```


[*]This work was supported in part by a grant of HPC resources from the Arctic Region Supercomputing Center and Los Alamos National Laboratory.

[†]Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. LANL LA-UR-03-2362.

PPoPP'03, June 11–13, 2003, San Diego, California, USA.

That is, the arrays of index values for the two dimensions are simply interchanged.

The remap operator is clearly powerful, but implementing such a communication operator in a high-level language such as ZPL is a concern because of its potential expense. Specifically, to implement a gather of the form

```
D := S#[M1, M2, ..., Md];
```

implies potential for considerable data motion (the problems are identical for scatter). Even presuming that all $d + 2$ arrays are allocated to processors identically, an all-to-all communication is typically required to specify where the elements are to be moved to. A second all-to-all communication is potentially required to transfer the elements. Further, because the data is coming from or going to arbitrary positions in the memory, considerable memory management is necessary to marshal and distribute the data. Such generality is required in the most complex cases, but in many common cases much less communication and memory management are possible. The technical problem considered in this paper is: *How can the remap operator for gather and scatter be implemented efficiently in a high-level language?* The research goals are first to understand where the costs are for remapping, and second to discover ways to optimize those portions of the implementation so that they approximate the performance of hand-coded gather and scatter.

This paper's contributions are as follows:

- We present an operator for arbitrary gathers and scatters that has unique semantics and provides power unmatched by other array languages including APL and F90, resulting in cleaner, more understandable code. Moreover, the operator is general enough to apply to most array languages.

- We discuss a parallel implementation for the operator and introduce optimizations for schedule compression, dead array reuse, and direct communication that lessen the costs of the operator's generality.

- We demonstrate comparable performance to efficient, hand-coded Fortran + MPI benchmarks.

This paper is organized as follows. In the next section, we discuss related work. In Section 3, we introduce the ZPL language. In Section 4, we describe the remap operator through a series of examples that demonstrate its power, and we discuss our implementation of the operator in ZPL. In Section 5, we evaluate the performance of the remap operator in the context of the NAS FT and CG benchmarks, and, in Section 6, we conclude.

2. BACKGROUND AND RELATED WORK

2.1 Traditional Array Remappings

Fortran remains the most widely-used language for scientific computing. The Fortran 90 and 95 revisions [1, 12] include several extensions for array-based programming, including the ability to refer to *array sections* using a slicing notation. While Fortran allows programmers to use index vectors to specify irregular array accesses, it supports a "cross-product" interpretation of the map vectors, rather than ZPL's "elementwise" style. For example, the Fortran expression `A(I(:), J(:))` results in a 2-dimensional array

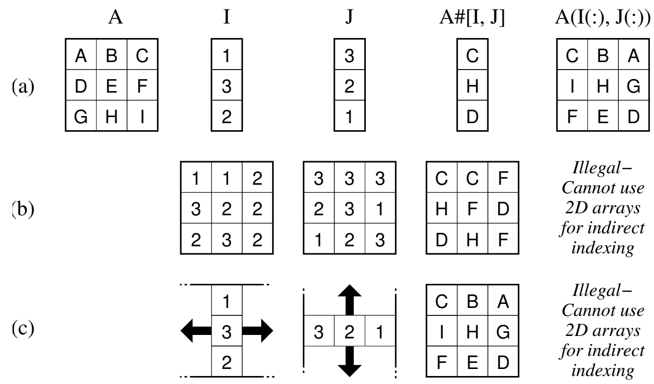

Figure 1: An example contrasting ZPL's elementwise remap semantics with Fortran's cross-product semantics. The first column shows array A, the source array for all examples. The next two columns show three pairs of map arrays, I and J. The final two columns show the result of applying these map arrays to A in ZPL and Fortran. (a) When using 1-dimensional map arrays, ZPL generates a 1-dimensional result while Fortran generates a 2-dimensional result using the cross-product of I and J thereby forcing the rows and columns to stay aligned as in a matrix pivot. (b) ZPL can use multidimensional maps to generate a multidimensional result, arbitrarily scrambling, replicating, and deleting values. (c) Multidimensional maps with replicated values (arrows) can be used to express efficient cross-product semantics as in Fortran.

whereas in ZPL the equivalent expression would result in a 1-dimensional array (Figure 1a). ZPL's remap operator creates higher-dimensional expressions by using higher-dimensional map arrays; these are illegal in Fortran (Figure 1b). When cross-product semantics are desired, ZPL programmers would use higher-dimensional map arrays with replicated values, as described in Section 3.3 (Figure 1c).

One way to express ZPL-style elementwise remaps in Fortran is to flatten the multidimensional source array down to one dimension and then use a single index vector to access its elements arbitrarily. The result would then have to be reinterpreted as a higher-dimensional array to return to the original problem space. Alternatively, loops and scalar indexing can be used to access the array's values one at a time as in traditional scalar languages like C. These idioms weaken the multidimensional array abstraction by failing to cleanly express atomic random access to its elements.

APL [15] is a well-known array language which was first introduced in the 1960's and remains in use today. It provides many built-in operators, including special operators for transpose and rotate (special instances of remapping). Unlike Fortran, APL supports indexing into arrays using higher-dimensional maps, but this still has cross-product semantics. For example, consider the APL statement:

$$D \leftarrow S[M1; M2]$$

If S, $M1$, and $M2$ are all 2-dimensional arrays, D would be a 4-dimensional array where $D_{i,j,k,l} = S_{M1_{i,j},M2_{k,l}}$. APL programmers can flatten the source array to perform an elementwise remap, as in Fortran, but they can use multidimensional map arrays to produce a higher-dimensional result. For example, in the statement:

$$D \leftarrow S[M]$$

if S is 1-dimensional and M is 2-dimensional, D would be a 2-dimensional array with the same size and shape as M.

2.2 Parallel Array Remappings

The vast majority of today's parallel programs are written using a sequential language like Fortran or C in a *single program, multiple data* (SPMD) style with a communication library to transfer data between the program instances. MPI [18] is the most widely-used example of such a library. Its core functionality provides the programmer with send and receive constructs for passing messages between program instances. Higher-level functionality is also provided in the form of collective communication calls, including routines that support scatter, gather, and all-to-all communication patterns. While these routines are quite general, their argument lists are nontrivial, requiring the user to specify data and buffer layouts in memory, the amount of data being sent, and the amount expected to be received from cooperating processors. While such parameters are not unreasonable for a communication library like MPI, they force programmers to take a processor- and memory-centric view of their programs. In contrast, ZPL's remap operator is data-centric, factoring details of data distribution and communication away from the specification of an array expression's global access pattern.

It should be noted that the overhead of MPI's collective communication routines is sufficient that in degenerate cases, programmers can often achieve a performance benefit by replacing the collective call with individual sends and receives. Examples of this can be found in the replicated vector transpose operation for the NAS CG benchmark. For arbitrary problem sizes and processor grids, all-to-all communication would be required for this operation, and an `MPI_Alltoallv()` call would be appropriate. However, by restricting the size and shape of the processor grid, as well as the problem size, the NAS implementation uses a single send and receive per processor to achieve the minimal amount of communication required. Ideally, such optimizations could be encapsulated in higher-level abstractions, allowing the user to focus on the algorithm at hand rather than details of tuning the implementation and restricting its generality.

SHMEM [3] is an alternate communication library. It strives to support a shared-memory model on distributed-memory architectures via one-sided communication. Like MPI, SHMEM supports gather and scatter data transfers in the form of `shmem_ixget()` and `shmem_ixput()`. The one-sidedness of the SHMEM routines allows data to be read or written to a processor's memory without explicitly coordinating with that processor. This provides a nicer abstraction for scattering and gathering data, yet still imposes a processor- and memory-centric view of the algorithm on the user.

Many of the current wave of parallel languages also utilize the SPMD model, but provide higher-level abstractions to express data transfer between the program instances. Examples include Co-Array Fortran [17] and Unified Parallel C (UPC) [6]. Both of these languages are a dialect of a traditional language, which eases the learning curve for experienced Fortran and C users. However, each language also inherits its array access idioms from the base language, implying no additional power for expressing array remaps. Co-Array Fortran supports a new type of array dimension—the *co-dimension*—which spans the executing instances of the program. Indexing into this dimension allows programmers to refer to data on remote processors. This results in a much more abstract view of communication than either MPI or SHMEM, yet still requires the programmer to express their algorithm with a localized per-processor view. UPC supports similar features for indexing into program instances and adds additional concepts for referring to shared data.

Titanium [21] is an SPMD-style language based on Java, yet it introduces its own multidimensional array type for reasons of performance and flexibility. Titanium's arrays support scatter and gather methods which use a list of indices to scatter/gather a 1D vector of values to/from a multidimensional array. Titanium users can therefore remap a multidimensional array by gathering the source array's values into a vector temporarily and then scattering them to the destination array. However, programmers typically declare one Titanium array per processor in the SPMD style and move data between processors manually, typically using array copy methods. The implication is that the scatter and gather methods can only be applied to these local per-processor arrays and not to the complete array that they represent. As with the other SPMD languages, programmers must be aware of processors when expressing parallel remaps.

2.3 Scheduling Communication

Traditionally, in parallel computation, unknown access patterns caused by indirect indexing are implemented using the *inspector-executor techniques* of Saltz, et al. [16, 20]. In such techniques, an inspector loop nest first observes the array indices, tracking data dependences. Then the executor implements the actual computation based on the inspector's findings. The base implementation of ZPL's remap operator also uses inspector-executor techniques to determine the communication required by the map arrays and then implement the array statement. It is worth noting that while some inspector-executor schemes seek to parallelize and preserve data dependences in sequential user code, ZPL's statements are explicitly parallel and have clear data dependences. Thus, our inspector is concerned with maximizing performance rather than preserving correctness.

High Performance Fortran (HPF) [14] is similar to ZPL in that it provides a global view of the program and syntactically separates computation from the specification of processor grids and data distributions. Unlike ZPL, HPF does not restrict operations between arrays with different data distributions. Thus, a statement like `C = A + B` may or may not require communication depending on the distributions of `A`, `B`, and `C`. A great deal of research in the HPF community has focused on generating efficient and minimal communication for HPF programs [13, 5]. Some of these efforts apply strength reduction to communication, replacing general all-to-all communications with reductions or nearest neighbor exchanges. ZPL handles this issue by providing distinct array operators for different styles of communication combined with a performance model that lets users reason about the communication induced by each [8].

In the absence of successful optimizations, HPF statements require similar inspector-executor constructs as ZPL's remap operator. A recent work of interest strives to minimize the overhead of these inspectors by giving users a means of naming, specifying, and capturing communication schedules [4]. Thus, users may assert that a communication schedule is identical between multiple invocations of a loop or from one loop to another. This allows the compiler to amortize the inspector overhead across multiple statements in cases where it otherwise would have to take a more conservative approach. In our work, we rely on the compiler to determine when remap communication schedules can be reused. As ZPL programs grow in size and complexity, it is possible that we will want to provide the user with similar cues to indicate when a remap's schedule can be reused.

3. ZPL

ZPL is a data-parallel array programming language developed at the University of Washington. It provides the programmer with a global view of the computation and transparent control of communication. The current ZPL implementation is based on a compiler that translates the ZPL code to a C program with calls to a chosen communication library, including MPI and SHMEM. In this section we introduce aspects of ZPL relevant to this paper. The interested reader is referred to the literature for more information [7, 19].

3.1 Regions and Parallel Arrays

Central to ZPL is the concept of the *region*. A region is an index set with no associated data. The region serves two fundamental purposes in ZPL: declaring parallel arrays and specifying parallel computation. To declare a parallel array, the programmer specifies its shape and size using a region; alternatively, in the case of dynamic parallel arrays, the programmer specifies the region in the program. In the following example, we (1) declare a region `R` to be the index set containing (i, j) for all i and j such that $1 \leq i, j \leq n$, (2) declare a region `IntR` to contain the interior indices of `R`, $1 < i, j, < n$, (3) declare arrays `A`, `B`, and `C` over region `R`, and (4) assign the interior elements in `C` the sum of the corresponding elements in `A` and `B`:

```
1    region R = [1..n, 1..n];
2           IntR = [2..n-1, 2..n-1];
3    var A, B, C : [R] double;
     ...
4    [IntR] C := A + B;
```

Since `A`, `B`, and `C` are defined over the same region, they are distributed in the same way over the processors, and no communication is required to compute the statement in line 4. Had `A`, `B`, and `C` been declared with differing distributions, the code in line 4 would result in a compile or runtime error. Instead the statement would need to be rewritten using an array operator to manage the data movement.

3.2 Array Operators

In ZPL, all communication results directly from the use of array operators. Programmers are thus provided with a syntactic cue as to the type and amount of communication occurring in parallel executions of their codes. This syntactic cue provides a simple, yet powerful, performance model [8] that further distinguishes ZPL from parallel programming languages like HPF and UPC in which the programmer may not always see from the syntax that a statement requires communication. In this section, we provide a brief introduction to ZPL's reduction and flood operators in preparation for the in-depth introduction to the remap operator's usage in Section 4.

3.2.1 The Reduction Operator

The *reduction* operator, `op<<`, reduces the values in an array to a lower-rank slice of the array or a single scalar value. A common use of the reduction operator is to compute the minimum of all the elements in an array. We might also use a reduction to find the sums of the elements in every row of an array and store these sums in the first column of another array. These examples follow:

```
1    [R] val := min<< A;
2    [1..n, 1] B := +<< [R] C;
```

We assume in these examples that `A`, `B`, and `C` are as declared before while `val` is declared as a scalar double. Line 1 computes the minimum of every element in `A` with indices in `R` and stores the result in `val`. Line 2 uses two regions to control the computation. The column region controls where the result is stored in `B`. The first dimensions of the two regions match, so we only reduce over the second (collapsed) dimension. As a rule, we reduce over each dimension that is collapsed. We use `+` to find the sum of the elements in every row. In general, reductions may use several built-in or user-defined operators [11].

3.2.2 The Flood Operator

The *flood* operator, `>>`, provides nearly the opposite behavior of the reduction operator. With this operator, the programmer is able to replicate a value throughout an array or values in a slice of the array to a larger slice. For example, suppose the programmer wants to multiply the value in the $(1, 1)$ position of array `A` with every value in array `B` and store the result in array `C`. One way to accomplish this is to replicate that value in `A` throughout `A` as with the following statements:

```
[R] A := >>[1, 1] A;
[R] C := A * B;
```

As written, the above code is inefficient: in total, the p processors store the same value n^2 times rather than p times. In the next section, we discuss a type of region dimension that allows for the efficient storage and computation of the result of the flood operator.

3.3 Flooded Dimensions

The flood operator results in potentially redundant storage on any given processor. The flooded dimension solves this problem. A *flooded dimension*, `*`, is one in which every value in that dimension is constrained to have the same value. Each processor owning a piece of that dimension stores only a single copy of that value. For example, the code below multiplies an $n \times 1$ column matrix by a $1 \times n$ row matrix to form an $n \times n$ square matrix. For simplicity, we take the first column of array `A` to be the column matrix and the first row of array `B` to be the row matrix. The resulting product is stored in array `C`.

```
1    var Col :  [1..n, *] double;
2        Row :  [*, 1..n] double;
     ...
3    [1..n, *] Col := >>[1..n, 1] A;
4    [*, 1..n] Row := >>[1, 1..n] B;
5    [R]       C := Col * Row;
```

Since we need access to the row and column matrices over the entire square region, it makes sense to replicate the arrays with the flood operator. All communication occurs in lines 3 and 4. The storage needed for the partial values, `Col` and `Row`, is minimized. We could also write the same computation without explicitly declaring the flooded arrays. There is no change in the computation since the result of the flood operator is an array with the appropriate flooded dimensions. This code is as follows:

```
[R] C := (>>[1..n, 1] A) * (>>[1, 1..n] B);
```

As an aside, flooded dimension are important for defining the arrays `Index1` and `Index2` that were informally mentioned in the introduction. These built-in constant arrays belong to a series of arrays, `Index`i, where each contains the values of the indices in the ith dimension of the region scope and all dimensions other than the ith are flooded.

4. THE REMAP OPERATOR

ZPL's remap operator, `#`, performs either gather or scatter operations on arrays. The general form of the gather is

```
[R] D := S#[M1, M2, ..., Ms];
```

where the region, `R`, the destination array, `D`, and the map arrays, `M1`, `M2`, ..., `M`s, are of the same rank and the source array, `S`, is of rank s. In addition, `D` must be writable over `R` and `M1`, `M2`, ..., `M`s must contain valid indices for `S`. The general form of the scatter is

```
[R] D#[M1, M2, ..., Md] := S;
```

where the region, `R`, the source array, `S`, and the map arrays, `M1`, `M2`, ..., `M`d, are of the same rank and the destination array, `D`, is of rank d. In addition, `S` must be readable over `R` and `M1`, `M2`, ..., `M`d must contain valid indices for `D`.

In this section we demonstrate the power of the remap operator with a number of examples, examine the use of the remap operator in ZPL versions of the NAS FT and CG benchmarks, and discuss the implementation of this operator in ZPL.

4.1 Some Basic Examples

For the following examples, let `R` be a region containing the indices (i, j) for all i and j such that $1 \leq i, j \leq n$ and let `A` and `B` be arrays of double-precision floating-point numbers declared over the region `R`.

4.1.1 Row Permute

Problem: If `M` is a parallel array of integers declared over `R` and the rows of `M` contain permutations of the integers between 1 and n, then permute the elements in the rows of `A` according to `M`.

Solution: The following two statements suffice:

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

In the gather, the map array `M` specifies where the elements are to be mapped from whereas, in the scatter, `M` specifies where the elements are to be mapped to.

4.1.2 Skew

Problem: A common use of the remap operator is to permute the array elements in a structured way. The skew permutation shows up in certain numerical algorithms. Permute the data in array `A` so that the elements in row i are cyclically shifted to the right $i-1$ times.

Solution:

```
[R] A := A#[Index1, ((Index2+Index1-2)%n)+1];
```

Note the use of the modulus operator, `%`. Using the same maps, we can write a similar computation with the scatter:

```
[R] A#[Index1, ((Index2+Index1-2)%n)+1] := A;
```

In this case the direction of the shift is reversed.

4.1.3 Redistribute

Problem: Let `Z` be an array of the same shape and size as `A`, but assume its distribution is different. Copy the data in `Z` to `A`.

Solution: The following results in a compile or runtime error:

```
[R] A := Z;
```

Communication may be necessary, but there is no hint to the compiler or programmer of this eventuality. Since no logical remapping is taking place, only a physical redistribution, the *identity gather* suffices:

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

The *identity scatter* results in the same movement, though A's region cannot specify it:

```
[1..n, 1..n] A#[Index1, Index2] := Z;
```

4.1.4 Diagonal Replicate

Problem: Copy the main diagonal of array `A` to a replicated row array. This problem illustrates gather's one-to-many mapping.

Solution:

```
var Row :  [*, 1..n] double;
    ...
[*, 1..n] Row := A#[Index2, Index2];
```

This solution cannot be implemented with the scatter operator which does not support one-to-many mappings.

4.1.5 Diagonal Reduce

Problem: Compute the sum of the elements in each column of array `A` and leave the results in the main diagonal of array `B`. This problem illustrates scatter's many-to-one mapping.

Solution:

```
[R] B := 0;
[R] B#[Index2,Index2] += A;
```

The `+=` assignment operator resolves collisions. Symmetric to the previous example, a gather would be insufficient. It is interesting to note that the reduction operator may be a better choice. By reducing to a flooded row array a basic assignment could move the results to the main diagonal of `B`, and we could take advantage of the parallelism inherent to addition's associativity.

4.1.6 Rank Change

Problem: Compute the matrix multiplication of A × B using bulk communication.

Solution:

```
1    region IJ  = [1..n, 1..n, *];
2           JK  = [*, 1..n, 1..n];
3           IK  = [1..n, 1, 1..n];
4           IJK = [1..n, 1..n, 1..n];
5    var C  : [IK] double;
6        A3 : [IJ] double;
7        B3 : [JK] double;
     ...
8    [IJ] A3 := A#[Index1, Index2];
9    [JK] B3 := B#[Index2, Index3];
10   [IK] C := +<< [IJK] (A3 * B3);
11   [R]  A := C#[Index1, 1, Index2];
```

Since arrays of different rank in ZPL are distributed across the processors differently, programmers must use the remap operator to change ranks. In the code above, each 2D array is promoted into 3D space by replicating it in a single dimension (lines 8 and 9). These flooded arrays are multiplied and accumulated in the final dimension to form the product (line 10). The product is then remapped to a 2D array (line 11). This algorithm is described in the literature [9].

4.2 NAS FT 3D Transpose

The NAS FT benchmark [2] numerically solves a 3D partial differential equation using forward and backward Fast Fourier Transforms (FFTs). The computation involves solving 1D FFTs on each dimension of a 3D array. The basic idea is to always leave at least one dimension of the array local to a processor in order to keep the complicated access patterns required by a 1D FFT from inducing communication. After computing an FFT on the local dimension, the array is transposed, if necessary, so that a different dimension is local. If two dimensions are distributed (2D layout), the array is transposed twice to compute the three FFTs; if only one dimension is distributed (1D layout), the array is transposed only once to compute the three FFTs.

In the 1D layout, we distribute only the first dimension. Note that in the Fortran code the opposite is done because of the column-major layout choice. Given the region declarations

```
region RXYZ = [1..nx, 1..ny, 1..nz];
       RYZX = [1..ny, 1..nz, 1..nx];
```

and knowing that X1 is first declared over RXYZ and then reallocated over RYZX while X2 is first declared over RYZX and then reallocated over RXYZ, the backward and forward transposes in ZPL are given by

```
[RYZX] X2 := X1#[Index3, Index1, Index2];
...
[RXYZ] X2 := X1#[Index2, Index3, Index1];
```

These two lines of code are equivalent to 88 lines in the Fortran + MPI implementation. In Fortran + MPI, instead of regions, loops guide the computation, and an array is used to store the different dimension lengths for the transposed arrays. Communication is specified with MPI function calls that require specifying processor dependent information.

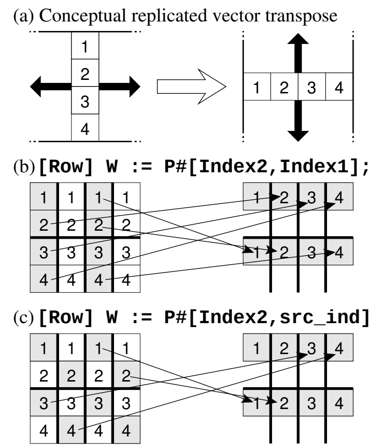

Figure 2: Three illustrations of the replicated vector transpose of the NAS CG benchmark: (a) a conceptual interpretation of the data movement, (b) an implementation on a 2×4 processor grid using the standard transpose expression, and (c) an implementation on a 2×4 processor grid where the second map is specified so as to induce a one-to-one communication pattern. The shaded values are those that are copied and arrows indicate interprocessor data movement.

4.3 NAS CG Replicated Vector Transpose

The NAS CG benchmark [2] estimates the largest eigenvalue of a symmetric positive definite sparse matrix by the inverse power method. The main loop contains a sparse matrix vector multiplication, several reductions, and a replicated vector transpose.

In the replicated vector transpose, the values in a column vector are transposed to a row vector. Figure 2(a) illustrates the data movement without regard to implementation. In ZPL, the row and column vectors are defined as follows:

```
region Row = [*, 1..n];
       Col = [1..n, *];
var W : [Row] dcomplex;
    P : [Col] dcomplex;
```

We can transpose the values in P to W with the following statement:

```
[Row] W := P#[Index2, Index1];
```

Figure 2(b) illustrates the parallel implementation of this transpose on a 2×4 processor grid. In this case, the communication pattern is not one-to-one. Some processors are sending data to processors that do not need the data (because it is replicated there), other processors are sending data to two processors, and still other processors are not sending data at all.

If each processor specifies an index that points it to a unique processor, then we can achieve a one-to-one communication pattern on both $2^k \times 2^k$ and $2^k \times 2^{k+1}$ processor grids. In ZPL, we use a scalar variable, src_ind, to achieve this pattern. If *rows* and *cols* are the number of row and column processors and *row* ($0 \leq row < rows$) and *col* ($0 \leq col < cols$) identify the computing processor, then we set `src_ind` with the formula

$$row \times \left\lfloor \frac{n}{rows} \right\rfloor + \left(col \times \left\lfloor \frac{n}{cols} \right\rfloor \right) \bmod \left\lceil \frac{n}{rows} \right\rceil + 1$$

The statement

```
[Row] W := P#[Index2, src_ind];
```

implements the improved communication pattern, illustrated in Figure 2(c).

The NAS Fortran + MPI implementation achieves this same communication pattern, but with less generality. As with ZPL, each processor determines a unique processor with whom to communicate. Then, unlike in ZPL, single matching MPI send and receive calls are used to move the data. Since there are only single calls, the implementation can only run on $2^k \times 2^k$ and $2^k \times 2^{k+1}$ processor grids. To make it more general would require significant work. The ZPL version runs successfully on different processor grids, though the communication pattern is not necessarily one-to-one on the different grids.

4.4 Implementation

The two-sided message-passing implementation of the general remap operator is non-trivial. There is the potential for all-to-all communication, and before the actual data can be transmitted between processors, the pattern of communication must first be established. In the case of the gather, the processors do not initially know where they must send data, and in the case of the scatter, the processors do not initially know from where they must receive data.

Figure 3 illustrates the basic two-sided message-passing implementation of both gather and scatter versions of the remap operator. These implementations are identical up to line 12. In the initial loop, lines 2 to 7, we compute the processor map, per-processor buckets of local indices, and local counts. The processor map contains the processor number that owns the value pointed to by the map arrays. The buckets of local indices are filled with the indices specified by the map arrays such that the bucket for the processor owning a given index contains that index. The local counts are set to the number of indices in each bucket of local indices.

We communicate between the processors in lines 8 to 12. The local counts are sent to the other processors' remote counts so the remote count of processor q on processor r equals the local count of processor r on processor q. Similarly, the buckets of local indices are sent to corresponding buckets of remote indices. The counts are sent before the indices so that the buckets for the remote indices may be allocated to the proper size.

The gather and scatter differ in lines 13 to 22. We discuss the gather first. In the loop of lines 13 to 14, we fill per-processor buckets of local data from the source array. We use the buckets of remote indices to read from the source array in an arbitrary order. The buckets of local data are filled in order. Then, in lines 15 to 17, the local data is sent to remote data buckets. The last step, lines 18 to 22, is to copy the remote data into the destination array. Here we read from the remote data buckets in order and, by traversing the region, write to the destination array also in order. We use the processor map to select which remote data bucket to read from. Since the indices used by the remote processor were in the order of the region traversal, we obtain the correct result.

The scatter is symmetric to the gather, differing in the following way. We fill the local data buckets, reading from the source array in order. We then write to the destination array in an arbitrary order. Note the fundamental differences between the scatter and the gather. In the gather, we read from an array in a cache-unfriendly way whereas, in the scatter, we write to an array in a cache-unfriendly way. More distinctions extend to the parallel implementation. In the scatter, we read from the source array before requiring the remote counts and indices; in the gather, we need the remote counts and indices before reading from the source array.

These distinctions lead us to believe that we should be able to tell whether to prefer the scatter or the gather based on certain rules of thumb if we are in a situation where either applies. For example, we could use either the scatter or the gather to write the 2D transpose of Section 1, the redistribution of Section 4.1.3, and the 3D transpose of Section 4.2. However, it is unclear which is preferable in these situations. Nonetheless, the importance of an optimization discussed in Section 4.5.4 suggests we favor the gather since this optimization is less readily applicable to the scatter.

4.5 Optimizations

The generality of the remap operator and its wide applicability make it slower than the other array operators in ZPL. Indeed, it is the communication operator of last resort. Even so, there are a number of optimizations that greatly improve its efficiency. In this section, we discuss a number of general optimizations. We have not focused on specific idiomatic optimizations in our implementation, though it is easy to imagine several that could further improve our results.

4.5.1 Map Saving/Sharing

The remap operator is commonly used to perform stylized collective communication. Examples include transposing arrays or slices of arrays, rotating arrays or slices of arrays, translating arrays or slices of arrays, etc. Moreover, such uses might occur within the main loop of a program. Great benefit may be reaped by caching copies of the counts, indices, and processor map so that they do not need to be recalculated every iteration. We call this optimization *map saving* since we save the maps used to remap the data.

If the region and map arrays remain unchanged between two instances of the same remap operator, we can skip lines 1 to 12 of Figure 3 for both the scatter and gather. There are two ways to implement this optimization; either we may use static analysis or we may use a more dynamic approach. The static approach is more conservative but may result in cleaner and faster code. We opt for the dynamic approach due to the optimization's importance and because the additional runtime support is not substantial.

The optimization is as follows. If the map information exists when we come to the start of the gather or scatter, we use it. Otherwise, we recompute the map. Additionally, wherever the region or map arrays change in the program, we destroy the map information. Care is taken to assure that if the map arrays are changed on any processor, the map information is destroyed on all processors. ZPL's programming model lets us do this without the need for communication.

Another benefit of the dynamic scheme is that it aids another optimization called *map sharing*. In this optimization, the map information is shared between remap operators that

Gather Implementation

```
[R] D := S#[M1, M2, ..., Mk];

1   Lcnt[1..PROCS] := 0
2   forall i = (i1, i2, ..., ik) in R
3      M := (M1[i], M2[i], ..., Mk[i])
4      p := proc_owns(M)
5      Pmap[i] := p
6      Lind[p][Lcnt[p]] := M
7      Lcnt[p] := Lcnt[p] + 1
8   forall p in 1..PROCS
9      send Lcnt[p] to p
10     receive Rcnt[p] from p
11     send Lind[p][1..Lcnt[p]] to p
12     receive Rind[p][1..Rcnt[p]] from p
13  forall p in 1..PROCS and e in 1..Rcnt[p]
14     Ldata[p][e] = S[Rind[p][e]]
15  forall p in 1..PROCS
16     send Ldata[p][1..Rcnt[p]] to p
17     receive Rdata[p][1..Lcnt[p]] from p
18  Lcnt[1..PROCS] := 0
19  forall i = (i1, i2, ..., ik) in R
20     p := Pmap[i]
21     D[i] := Rdata[p][Lcnt[p]]
22     Lcnt[p] := Lcnt[p] + 1
```

Scatter Implementation

```
[R] D#[M1, M2, ..., Mk] := S;

1   Lcnt[1..PROCS] := 0
2   forall i = (i1, i2, ..., ik) in R
3      M := (M1[i], M2[i], ..., Mk[i])
4      p := proc_owns(M)
5      Pmap[i] := p
6      Lind[p][Lcnt[p]] := M
7      Lcnt[p] := Lcnt[p] + 1
8   forall p in 1..PROCS
9      send Lcnt[p] to p
10     receive Rcnt[p] from p
11     send Lind[p][1..Lcnt[p]] to p
12     receive Rind[p][1..Rcnt[p]] from p
13  Lcnt[1..PROCS] := 0
14  forall i = (i1, i2, ..., ik) in R
15     p := Pmap[i]
16     Ldata[p][Lcnt[p]] := S[i]
17     Lcnt[p] := Lcnt[p] + 1
18  forall p in 1..PROCS
19     send Ldata[p][1..Lcnt[p]] to p
20     receive Rdata[p][1..Rcnt[p]] from p
21  forall p in 1..PROCS and e in 1..Rcnt[p]
22     D[Rind[p][e]] := Rdata[p][e]
```

Figure 3: Pseudo-code for the implementation of the Gather and Scatter operators.

access the same region and set of map arrays at different static points in the program. In the NAS CG benchmark, for example, the same remap occurs twice within the main loop.

4.5.2 Computation/Communication Overlap

A common optimization parallel programmers often employ is to *overlap communication with computation* in order to hide latency. This optimization applies to the remap operator in ZPL. The compiler will automatically push independent computations between lines 16 and 17 of the gather implementation and between lines 19 and 20 of the scatter implementation as shown in Figure 3. In addition, the compiler will push independent computations between lines 11 and 12 of both remap forms. This additional push is done with a lower priority because the map saving optimization may eliminate this communication altogether. This optimization cannot be applied by the MPI programmer to the monolithic collective communication routines.

4.5.3 Schedule Compression

Stylized collective communication patterns like those mentioned in Section 4.5.1 benefit from encoding the processor map and buckets of indices in such a way as to decrease the storage and communication requirements and improve the performance of indexing into the arrays when the potentially arbitrary access pattern is actually a strided sequence. We use *strided run length encoding* to store the processor map and buckets of indices. Through a careful implementation, we never need to use the full amount of memory necessary to store unencoded representations. We use exactly the memory required to store the encoding plus a small constant amount of space for the work of actually encoding the sequences. Moreover, our implementation is designed so that if the encoding does not appear to have a benefit, we will stop the encoding process early and use unencoded representations.

We use a recursive scheme so we can encode the encoding if this is beneficial. In our implementation, by default, we base the number of recursive encodings on the rank of the remap operator. This choice is based on the optimal number of encodings we would need for the common redistribution example of Section 4.1.3.

As a basic example of the strided run length encoding, consider the sequence: 1, 2, 3, 4, 5, 6. Our run length encoder would stream in this sequence and output: 1, 1, 6. The initial value is 1, the stride is 1, and the length is 6.

The 2D transpose implemented with the gather demonstrates the power of run length encoding the indices. As we traverse the array in row major order, the map arrays, `Index2` and `Index1`, provide pairs of integers used to index the source array. The stream of pairs

(1 1) (2 1) (3 1) (4 1) (1 2) (2 2) (3 2) (4 2) (1 3) (2 3) (3 3) (4 3) (1 4) (2 4) (3 4) (4 4)

is easily compressed. One level of encoding produces

(1 1) (1 0) 4   (1 2) (1 0) 4   (1 3) (1 0) 4   (1 4) (1 0) 4

There are four sequences to decode. In the first sequence, the initial pair is (1, 1), the stride is (1, 0), and the length is 4. Since we are working on a 2D array, we use two levels of encoding, and produce

(1 1) (1 0) 4 (0 1) 4

There is one sequence to decode which starts with the pair (1, 1), the inner stride is (1, 0), the outer stride is (0, 1), and the inner and outer lengths are both 4. In producing this recursive encoding, the level one encoding is never produced, not even as an intermediate result. The total memory used to produce this encoding from the stream of indices is never more than the memory to store the final result, 8 integers in this case, and some constant amount of additional memory for the computations.

4.5.4 Dead Source/Destination Reuse

The buckets of data used in the implementation of the remap operator may consume significant memory. To avoid this, we employ an optimization called *dead source reuse* and *dead destination reuse*. If the destination array is dead before the remap, we use its memory for the local data buckets. Note that in the case of the gather, it is relatively easy to determine what data in the destination array will be overwritten. This is not the case for the scatter. If the source array is dead after the remap, we may use its memory for the remote data buckets. Then, in essence, we copy the source array to the destination array, locally with possible rearrangements of the data, send the data in the destination array to the remote source array, and, lastly, copy the source array to the destination array, again locally with possible rearrangements of the data.

This optimization is done by hand in the Fortran + MPI implementation of the NAS FT 3D transpose. It is easy for the ZPL compiler to determine that both the source and destination arrays are dead, thus it is able to duplicate the work of the Fortran programmer.

4.5.5 Direct Sending/Receiving

Both dead source reuse and dead destination reuse decrease the storage required to implement the remap operator, but an interesting case arises if, during either of the local copies to the destination array or a data bucket, no rearrangement of the data takes place. If the data is copied in order from one array to another, a *straight copy*, there is no reason to buffer the data. It may just be sent or received directly. The difficult task, then, is to detect whether a straight copy will take place. For this detection, the run length encoding of Section 4.5.3 comes to the rescue.

A small, well-structured, easily-detectable encoding of the indices is both necessary and sufficient to prove that the copy from the source array to the data buckets in the case of the gather or from the data buckets to the destination array in the case of the scatter is a straight copy when coupled with information about where the first and last elements are placed in memory, the size of each element, and the number of elements. It is even easier to tell if the other copy is straight; we just need the information about the first and last elements, the size of each element, and the number of elements. If the data is dense in memory, we know it is straight because, in these latter copies, we are copying the data in order.

This optimization, performed dynamically by the ZPL runtime, is equivalent to the straight-forward approach of the Fortran + MPI version of CG. Due to the dynamic nature, the ZPL implementation is more general, but also suffers some overhead. More interestingly, this optimization benefited the ZPL version of FT on some processor configurations though we did not anticipate this.

5. EVALUATION

In this section, we evaluate our implementation of the remap operator in the context of the NAS CG and FT benchmarks. The NAS parallel benchmarks are a suite of scientific applications and kernels representative of codes scientists write for parallel computers [2]. The Fortran + MPI provided implementations are highly-tuned. We compare the NAS codes qualitatively first, then examine differences in memory usage and execution time. We evaluate across three platforms: a Cray T3E, an IBM SP2, and a LinuxBIOS/BProc cluster. These machines are further discussed in Figure 6.

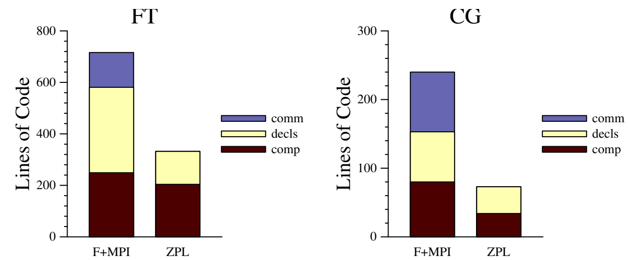

Figure 4: Charts showing line counts of the Fortran + MPI and ZPL implementations of the NAS FT and CG benchmarks. The counts are subdivided into lines used for communication, declarations, and computation.

5.1 Clarity

The remap operator and ZPL's high-level constructs make the programmer's job easier. Figure 4 counts the lines of code in the timed portions of the ZPL and Fortran + MPI implementations of the NAS FT and CG benchmarks. Lines of code is an imperfect metric for clarity, but yields important information nonetheless. Each ZPL implementation requires less than half as many lines of code as the corresponding Fortran + MPI implementations. The figures show a breakdown of the lines of code into those used for computation, declarations, and communication. The high-level approach of ZPL eliminates the need for the programmer to specify communication. The computation was written with significantly fewer lines because of ZPL's powerful array syntax based on the region. For example, the savings in implementing the 3D transpose remapping of the FT benchmark are enormous. Contrast the 88 Fortran + MPI lines of code (not shown) to the 2 ZPL lines of code (shown in Section 4.2).

5.2 Memory Usage

Memory usage is often as important a metric as execution time. Frequently, scientists would prefer to run their applications using the largest possible data sets. Thus the implementation of their code should use as little memory as possible. Figure 5 shows the effect of the remap optimizations discussed in this paper on the total memory usage for class C of the NAS FT and CG benchmarks running on 256 processors of a Cray T3E. The effects are similar on the SP2 and the cluster, and so are omitted.

For FT, the memory is subdivided into that needed for several tables and for the three main arrays. For CG, the memory is subdivided into that needed for the sparse array and for the vectors. In addition, the memory used for the remap is subdivided into that needed for the processor map, the indices, and the data buckets. There is significant memory overhead in the ZPL implementation of CG not

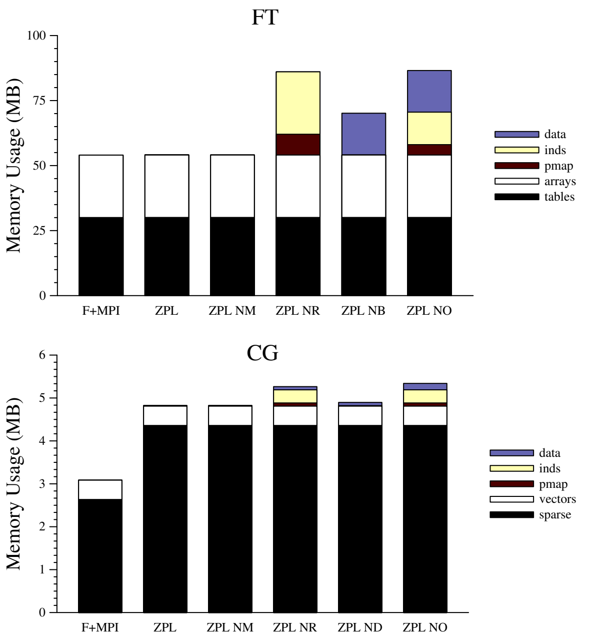

Figure 5: Charts showing the effect of the remap optimizations on the per-processor memory usage during the execution of a remap for class C of the NAS FT and CG benchmarks run on 256 processors of a Cray T3E. In FT, the memory is subdivided between general program data and the three large arrays; in CG, the memory is subdivided between the sparse array and the vectors. For both benchmarks, the memory used for the remap is subdivided between the processor map, the indices, and the data buckets. From left to right, we compare different implementations. The first two bars show the Fortran + MPI and optimized ZPL implementations. The remaining four bars show the ZPL implementation when certain optimizations are disabled: NM for no map saving, NR for no run length encoding, NB for no destination/source reuse, ND for no direct sending/receiving, and NO for no optimizations.

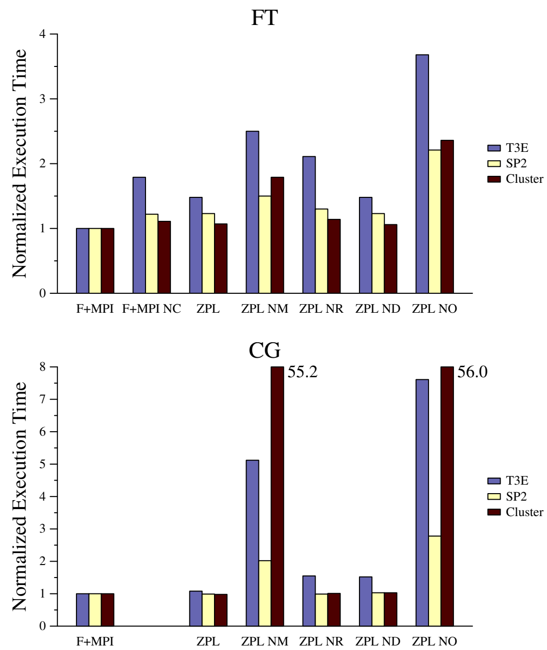

Figure 7: Charts showing the effect of the remap optimizations on the execution time of the transpose portion of the NAS FT and CG benchmarks across three different platforms: 256 processors of a T3E, 128 processors of an SP2, and 1024 processors of a cluster. The NC bar refers to disabling the hand-coded cache-blocking optimization in the Fortran + MPI implementation of the NAS FT benchmark. The other labels are as described in Figure 5.

connected to the remap operator. This stems from ZPL's general sparse array format described in the literature [10].

The optimized ZPL implementation of the remap operator uses nearly the same amount of memory as the Fortran + MPI implementation. Disabling the map saving optimization has little effect on the memory usage alone. The run length encoding, destination/source reuse, and direct sending/receiving optimizations all have significant effects on memory usage. Notice also that the map saving optimization is detrimental if the maps are big. Thus, for the FT benchmark, in which maps for different remaps are simultaneously saved, disabling run length encoding without also disabling map saving results in significantly more required memory.

5.3 Performance

Figure 6 shows results for the NAS FT and CG benchmarks for the class C problem size on increasing numbers of processors on our three platforms. We show speedup graphs for total time where the speedups are calculated over the best implementation's time on the fewest number of processors for which any implementation could complete without exhausting the memory. We show graphs of execution time when examining the time consumed by the remap operator. For all three platforms, we compiled ZPL to C and MPI. On the T3E, ZPL could exhibit improved performance using a SHMEM implementation.

The key observation is that the ZPL codes compete with the lower-level, highly-tuned Fortran + MPI codes. The speedup graphs show the total execution time to be comparable on all three platforms. The transpose time graphs show that the optimized remap implementation performs as well as the equivalent code in Fortran + MPI on both benchmarks, despite the very different communication patterns in each benchmark. In FT, the communication pattern is all-to-all; in CG, it is one-to-one.

Note that on the SP2, the Fortran + MPI implementations cannot take advantage of all 176 user processors because their communication is written to work only with 2^k processors. While general implementations are possible with MPI, they would require additional programmer effort that would complicate the code. In contrast, due to the generality of the remap operator and its implementation, the ZPL versions can run on 176 processors (or any other number) without modifying or recompiling the code.

Figure 7 shows the effect of individual remap optimizations on the transpose time. We disabled each of the map saving, run length encoding, and direct sending optimiza-

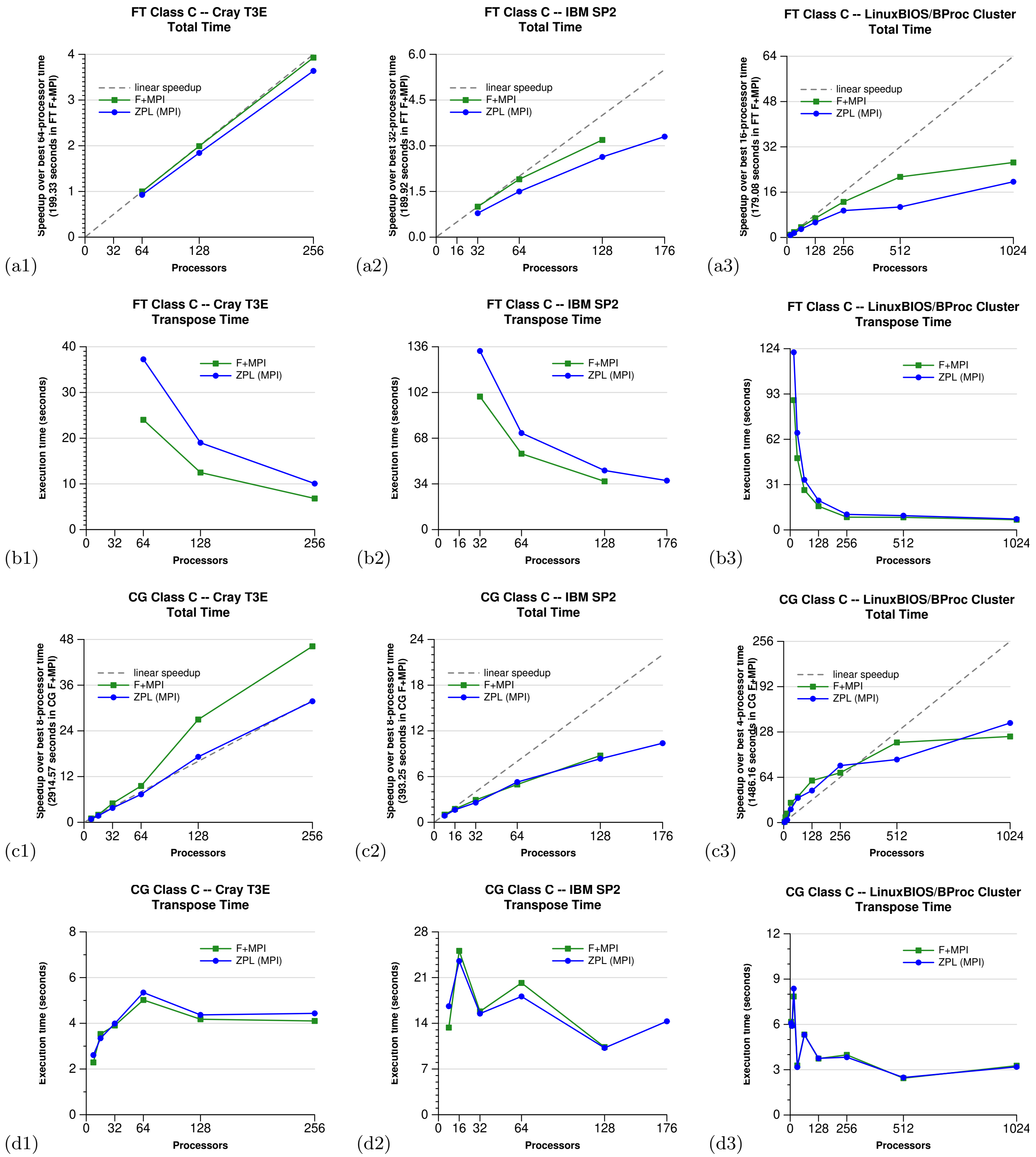

Figure 6: Graphs showing the total speedups and transpose times of class C of the NAS FT and CG benchmarks across three platforms. The *a* and *b* rows show the FT benchmark; the *c* and *d* rows show the CG benchmark. The *a* and *c* rows show the total speedup of the benchmark, while the *b* and *d* rows focus only on the transpose portion of the benchmark and show execution time. The *1* column shows results on Yukon, a 272 processor Cray T3E with 260 user processors. Each processor is a 450 MHz Alpha processor with 256 MB of memory. The *2* column shows results on Icehawk, a 200 processor IBM SP with 176 user processors. The SP2 is composed of 44 nodes with 2 GB of memory per node. Each node contains four 375 MHz power3 processors. The *3* column shows results on up to 1024 processors of Pink, a 2048 processor cluster built with the LinuxBIOS/BProc technology. Pink is composed of 1024 nodes with 2 GB of memory per node. Each node contains two 2.4 GHz Intel Xeon processors.

tions as well as all of the optimizations to see how each affected performance on each platform. In addition, we disabled a hand-coded, cache-blocking optimization of the local copy in the Fortran + MPI transpose code of the FT benchmark. This cache-blocking optimization, specific to the actual indexing pattern used in NAS FT, accounts for the ZPL overhead on all three platforms.

The effect of the remap optimizations differs significantly across the different platforms. For example, map saving optimization is crucial for the cluster, but run length encoding has little effect. On the T3E, run length encoding is almost as crucial as map saving.

6. CONCLUSIONS

This paper describes an array remap operator that provides power for gathering and scattering arrays unmatched by other languages. Unlike other parallel languages which require attention to memory layout, processor boundaries, and communication schedules, ZPL's remap operator permits the programmer to describe data movement only in terms of global logical indices. Through optimizations such as map saving, communication/computation overlap, schedule compression, dead array reuse, and direct communication, the operator allows for an efficient parallel implementation which is comparable in performance to hand-tuned Fortran and MPI.

7. ACKNOWLEDGMENTS

While completing this work, Steven Deitz was supported by the DOE High-Performance Computer Science Graduate Fellowship (HPCSGF) program and Sung-Eun Choi was funded by the Mathematical Information and Computer Sciences (MICS) program of the DOE Office of Science. We would like to thank past members of the ZPL research group for their contributions which made this work possible.

8. REFERENCES

[1] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook*. McGraw-Hill, New York, NY, USA, 1992.

[2] 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.

[3] R. Barriuso and A. Knies. SHMEM user's guide. Technical report, Cray Research Inc., May 1994.

[4] S. Benkner, P. Mehrotra, J. V. Rosendale, and H. Zima. High-level management of communication schedules in HPF-like languages. In *Proceedings of the ACM International Conference on Supercomputing*, pages 109–116, 1998.

[5] S. Benkner and H. Zima. Compiling High Performance Fortran for distributed memory architectures. *Parallel Computing*, 25(13–14):1785–1825, 1999.

[6] 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.

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

[8] 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.

[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] B. L. Chamberlain and L. Snyder. Array language support for parallel sparse computation. In *Proceedings of the ACM International Conference on Supercomputing*, 2001.

[11] 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.

[12] W. Gehrke. *Fortran 95 Language Guide*. Springer Verlag, October 1996.

[13] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proceedings of the ACM Conference on Supercomputing*, December 1995.

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

[15] K. E. Iverson. *A Programming Language*. Wiley, New York, NY, USA, 1968.

[16] R. Mirchandany, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the ACM International Conference on Supercomputing*, pages 140–152, July 1988.

[17] 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.

[18] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: the complete reference*. MIT Press, Cambridge, MA, USA, 1996.

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

[20] M. Ujaldon, S. D. Sharma, J. Saltz, and E. L. Zapata. Run-time techniques for parallelizing sparse matrix problems. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 43–57, 1995.

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