99

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

Steven J. Deitz
University of Washington
Seattle, WA 98195
deitz@cs.washington.edu

Bradford L. Chamberlain†
University of Washington
Seattle, WA 98195
brad@cs.washington.edu

Sung-Eun Choi
Los Alamos National Laboratory
Los Alamos, NM 87545
sungeun@lanl.gov

Lawrence Snyder
University of Washington
Seattle, WA 98195
snyder@cs.washington.edu

ABSTRACT

The data redistribution or remapping functions, gather and scatter, are of long-standing in high-performance computing, having been included in Cray Fortran for decades. In this paper, we present a highly-general array operator with powerful gather and scatter capabilities unmatched in other array languages. We discuss an efficient parallel implementation, introducing several new optimizations–run length encoding, dead array reuse, and direct communication–that lessen the costs associated with the operator's wide applicability. In our implementation of this operator in ZPL, we demonstrate comparable performance to the highly-tuned, hand-coded Fortran plus MPI versions of the NAS FT and NAS CG benchmarks.

1. INTRODUCTION

Gather and scatter operations are noticeably absent from most parallel programming systems. Instead, inadequate mechanisms serve to mitigate the difficult task of the scientist who must arbitrarily redistribute data across processors. ZPL [13], a parallel array programming language for scientific and engineering computations, provides the functionality necessary to solve the scientist's problem.

Gather and scatter are data redistribution or remapping functions of long standing in 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–gather the items–and so is written in ZPL 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–scatter the items–and so is written in ZPL as

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

Almost all aspects of the gather and scatter operations are symmetric.

Specifying the remapped positions is particularly easy for linear arrays since another linear array defining the remapped index positions can be given. Accordingly, if `S` and `D` are five element arrays, and `Rev` is an array containing the integers 5, 4, 3, 2, 1 in that order, then both

```
D := S#[Rev]; and D#[Rev] := S;
```

result in assigning `D` the elements of `S` in reverse order.

For higher rank arrays, say rank k, k-element index vectors are required to specify the positions of the new arrangement. This can be cumbersome, and so it is common for gather and scatter to be implemented only for linear arrays, implying that higher dimensional arrays must first be flattened. ZPL takes the view that a gather or scatter between rank k arrays can be specified by a sequence of k rank k arrays, each giving the index values for a specific dimension. For example, the built-in constant arrays, `Index1` and `Index2`, may be thought of as 3×3 arrays 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 transpose in ZPL is expressed with either of the following lines:

*This work was supported in part by a grant of HPC resources from the Arctic Region Supercomputing Center. The first author is supported by a DOE High Performance Computer Science Graduate Fellowship.

†This author is currently in the employ of Cray Inc.

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

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

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

That is, the arrays of index values for the two dimensions are simply interchanged. Arbitrary remappings merely require that the position-specifying arrays be set up properly.

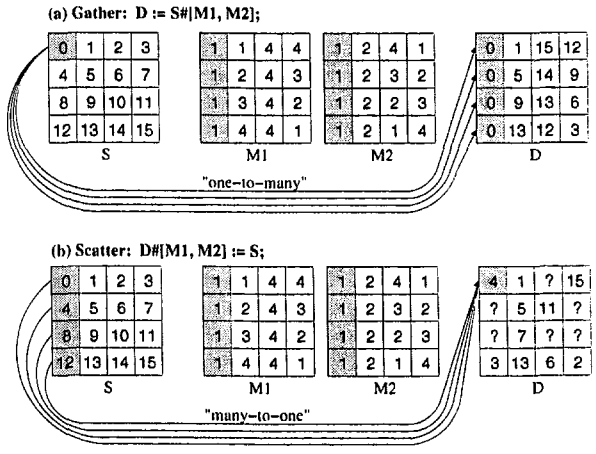


Figure 1: An illustration of an arbitrary (a) gather and (b) scatter between 2D arrays. In the gather, a 0 is replicated in the first column of the destination array as specified by the gather's one-to-many mapping. In the scatter, a 4 is placed in the first position of the destination array as an arbitrary resolution to the scatter's many-to-one mapping. Note, also, that not every element in the destination array is assigned a value based on the scatter.

A problem arises for gather and scatter if the index specification is not a permutation, i.e., not a one-to-one mapping. (Such cases are often forbidden.) If the indices are not unique, then a gather fetches multiple elements from a single position. On the other hand, a scatter maps multiple elements to the same position. Figure 1 illustrates a general two dimensional gather and scatter operation based on two map arrays showing that gathers may result in many-to-one mappings and scatters in one-to-many mappings. The exact behavior depends on how the remap is used in ZPL. If the scatter is given as a simple assignment as shown in all previous examples, the behavior is undefined, but still legal–the element assigned last "wins." The scatter may be written with operator assignment statements like += in which case the sum of the elements mapped to the same position would be stored in that position. Thus

```
D#[1] += S;
```

is an inefficient way to add up the items of S and store them in the first position of D.

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 (the problems are identical for scatter) of the form

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

implies potential for considerable data motion. Even presuming that all $k+2$ arrays are allocated to processors identically, an all-to-all communication is potentially 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 a unique semantics and provides power unmatched by other array languages including APL, 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 run length encoding, dead array reuse, and direct communication that lessen the costs of the operator's generality.
- We demonstrate comparable performance to highly-tuned, hand-coded Fortran + MPI benchmarks.

This paper is organized as follows. In the next section, we show how programmers typically write arbitrary gather and scatter operations using array languages like APL and Fortran 90 or communication libraries like MPI and SHMEM. In Section 3, we introduce the ZPL language. In Section 4, we describe the remap operator through a series of examples that illustrates 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 NAS CG benchmarks, and, in Section 6, we conclude.

2. BACKGROUND AND MOTIVATION

In this section we examine ways in which programmers write arbitrary gathers and scatters when using systems other than ZPL. These methods include communication libraries like MPI and SHMEM as well as array languages like APL and Fortran 90/95.

2.1 Array Languages

ZPL is a parallel array language, and, from the beginning, it was designed with the parallel implementation in mind. In many ways, this forced us to place more constraints on the programmer than other developers of array languages would have had to. However, in the case of the gather and scatter, the remap operator provides unmatched functionality.

2.1.1 APL

APL [10, 11] is a well-known array language first introduced in the 1960's and still in use today. It provides about 100 built-in operators, and in addition to having special operators for transpose and rotate (cases of gather and scatter), APL provides a relatively powerful form of arbitrary

gather and scatter based on indexing. For destination and source vectors, D and S, a vector map array, M, produces the standard gather and scatter operations:

$$D \leftarrow S[M]$$

$$D[M] \leftarrow S$$

A way to perform gather for higher dimensional arrays in APL is first to flatten the source array to produce a vector and then to construct an array of indices of the same shape as the destination array specifying for each position where the item is to be found in the flattened source. Thus, if M is the two dimensional array

$$M = \begin{matrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{matrix}$$

and S contains the flattened (row major order) elements of a 3x3 source array, $S2$, then the gather written above produces the transpose of $S2$, an alternative to using the built-in operator. Similarly, for the scatter we would produce a flattened destination array.

Interestingly, APL lets the programmer specify more than one multi-dimensional map array, but the semantics are very different from ZPL's remap operator. Considering only gather, though these concepts apply to the scatter as well, let $M1$ be an $m \times n$ array and let $M2$ be a $p \times q$ array. Then the statement

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

implies that the destination array, D, is an $m \times n \times p \times q$ array and the source array, S, is of rank two. Then for all permutations of i, j, k, and l, $D_{i,j,k,l}$, would be assigned $S_{M1_{i,j},M2_{k,l}}$. These ideas extend to arbitrary dimensions. In the 2D case, using two vectors of lengths m and n for the map arrays would result in $m \times n$ assignments from a 2D source array to a 2D destination array.

2.1.2 *Fortran 90*

Fortran 90 [1] can implement the standard gather and scatter operations using array subscripting. If the destination array, D, source array, S, and map array, M, are all one-dimensional, then gather and scatter are expressed as

```
D = S(M)
```

```
D(M) = S
```

Higher-dimensional arrays must be treated as 1D vectors in order to scatter or gather data between them. For multiple map arrays, the semantics are similar to APL, but the map arrays must be 1D vectors.

2.2 Communication Libraries

The de-facto standard for scientific parallel programming remains a sequential language like Fortran or C and a communication library. This combination provides performance currently unmatched by other approaches to parallel programming. Unfortunately, it is often difficult to write codes and is always a time-consuming endeavor.

2.2.1 *MPI*

MPI [12] is arguably the most used tool of parallel programming. At its core, the programmer is left to write matching "send" and "receive" function calls throughout the program. Building on top of this base, the MPI standard provides the programmer with several higher-level functions for collective communication. These functions may serve to implement an arbitrary gather and scatter, but not without significant work on the part of the programmer.

The `MPI_Gather` and `MPI_Scatter` functions are logical choices. However, these functions lack the generality of ZPL's remap operator and other language facilities of Section 2.1 which let the programmer take a global view of the computation. In a global view, programmers do not need to concentrate on the data movement between specific processors. These MPI functions force the programmer to be fully explicit. With the `MPI_Gather` function, a programmer specifies a root processor, a receiving buffer on the root processor, and a sending buffer on every processor (including the root). The execution entails that the sending buffers are concatenated and placed in the receiving buffer on the root processor. Thus the root processor gathers the smaller buffers on every processor and places the results in its larger receiving buffer. Similarly, in a scatter, the root processor scatters segments of its larger sending buffer to the smaller receiving buffers on the other processors.

We can implement the arbitrary scatter and gather operations by writing per-processor code to copy data for sending to other processors into buffers and then use up to p calls of either `MPI_Gather` or `MPI_Scatter`, where p is the number of processors. An alternative function that may be more appropriate is the `MPI_Alltoall` function. In this function, each processor specifies a receiving buffer and a sending buffer that are partitioned into sections for receiving data from each processor and sending data to each processor. The same per-processor copying code needs to be written to implement our arbitrary gather and scatter operations. Note there are numerous variations on these functions that may be used by the MPI programmer.

2.2.2 *SHMEM*

SHMEM [4] is a proprietary message-passing library implemented on various CRAY and SGI systems. Again, the programmer cannot take a global view of the computation and must specify what data is going to which processors. In SHMEM, rather than writing two-sided "send" and "receive" functions, the programmer specifies one-sided "get" or "put" functions. On top of these standard functions, the SHMEM standard supplies functions called `shmem_ixget` and `shmem_ixput`. In these functions, the programmer specifies the indices for where the data is to be gathered from or scattered to on the remote processor. To implement an arbitrary gather or scatter, the programmer must specify up to p calls to `shmem_ixget` or `shmem_ixput`, where p is the number of processors. Note that the arrays must be treated as one-dimensional for the indices.

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 as well as complete control over communication. The current ZPL implementation is based on a compiler that translates the ZPL code to a C program with calls to a chosen message-passing 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 [5, 13].

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 controlling 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 of the above example. Had A, B, and C been declared in such a way as to be distributed in different ways, the code in line 4 would result in either a compiler or runtime error. Instead the statement would need to be rewritten using one of ZPL's several array operators that induce communication.

3.2 Communication Operators

In ZPL, all communication directly results from the use of several array operators that induce it. Programmers are thus provided with a syntactic clue as to the type and amount of communication occurring in parallel executions of their codes. This syntactic clue provides a simple, yet powerful, performance model [6] that further distinguishes ZPL from parallel programming languages like HPF and UPC in which the programmer may not always see syntax indicating that a code segment requires communication. In this section, we provide a brief introduction to the reduction and flood operators. The remap operator, which also induces communication, is the subject of this paper; we provide an in-depth introduction to its 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 for line 1 of the above example that val is declared as a scalar double. In line 1, then, we take the minimum of every element in A that exists in R and store the result in val. Since A was declared over R, this is every element in A. In line 2, we use two regions to control the computation. The dynamically specified region controls where the result is stored in B. The first dimensions of the two regions match, so we only reduce over the second dimension. As a rule, we reduce over each dimension that is collapsed. We use + to find the sum of the elements in every row. Reductions may use a number of built-in operators or user-defined ones [9].

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. Its name implies the dramatic visualization of the replication taking place. 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 in the following lines of code:

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

Clearly there is an inefficiency in this code. While a smart compiler might perform optimizations, the programmer should not have to write this. In the next section, we discuss another type of region that allows for the efficient storage and computation of the result of the flood operator. To close this section, we mention that the previous code can be optimally rewritten as the following single line:

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

3.3 Flooded Dimensions

The flood operator results in potentially redundant storage on any given processor. In the example from the last section in which we replicate one value throughout A, we end up storing the same value $\frac{n^2}{p}$ times. 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. Consider the example of multiplying an $n \times 1$ column matrix by a $1 \times n$ row matrix to form an $n \times n$ square matrix. Take the first column of array A as our column matrix and the first row of array B as our row matrix. We want to store the product in C. Since the region factors out the indices in a computation so there is no communication without the use of communication operators and since there must be communication if A, B, and C are distributed in the same way, we need to use a communication operator. The flood operator is a perfect choice.

In the following code which performs the matrix multiplication, we declare (1) a column array and (2) a row array using flooded dimensions, use the flood operator to fill the (3) column array and (4) row array, and (5) compute the multiplication:

```
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;
```

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 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 the ith Index$i$ array contains the values of the indices in the ith dimension of any array and all dimensions other than the ith are flooded. Because all but one of the dimensions is flooded, one should assume that the memory required for the implementation of these arrays is minimal. In practice, we do even better: no memory is needed.

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, ..., Mk];
```

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

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

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

In this section we demonstrate the power of the remap operator with a number of telling examples, examine the use of the remap operator in ZPL versions of the NAS FT and NAS 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 Skew

A common use of the remap operator is for permuting data in an array. The skew permutation shows up frequently in numerical algorithms. The idea is to cyclically shift each successive row an increasing number of times. To permute the data in array A so that the elements in row i are cyclically shifted to the right $i-1$ times, we write (note the use of the modulus operator, %)

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

We may do the same computation with a scatter. Alternatively, we can keep the same maps and write

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

in which case the direction of the shift is reversed.

4.1.2 Redistribution

In this example we assume A and B are distributed across the processors in different ways. Then the following line of code would result in an error:

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

Communication is necessary so the programmer must use a communication operator. Since no logical remapping is taking place, only a physical redistribution, the *identity gather* suffices:

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

4.1.3 Diagonal Replication

In the following contrived example, we wish to replicate the values along the major diagonal of array A leaving the result in B such that $B_{i,j} = B_{j,i} = A_{i,i}$ for all i and j. The following single line of code does what we want:

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

This may not be done with a scatter since it requires the one-to-many mapping provided only by gathers.

4.1.4 Diagonal Reduction

Consider a nearly opposite problem from the previous example. Suppose we want to set the values in the major diagonal of array B such that $B_{i,i} = \sum_{j=1}^{i} A_{i,j} + \sum_{j=1}^{i-1} A_{j,i}$ for all i. The following line of code is sufficient:

```
[R] B#[min(Index1, Index2),
       min(Index1, Index2)] += A;
```

Notice the use of the += assignment operator to resolve collisions. Regular assignment, :=, is legal as well and, as discussed in the introduction, has the semantics of resolving collisions arbitrarily. Symmetrically to the example in Section 4.1.3, a gather would be insufficient.

4.1.5 Rank Change

One well-known technique for matrix multiplication in which synchronization is minimized may be written in ZPL using Problem Space Promotion (PSP) [7]. The basic idea behind PSP is to compute with arrays of rank higher than the initial arrays and use flooded dimensions to make the computation efficient. The PSP matrix multiplication algorithm is written in ZPL as

```
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#[Index3, Index2];
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 copy data between such arrays. In the above code, 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).

4.2 NAS FT 3D Transpose

The NAS FT benchmark [2, 3] numerically solves a 3D partial differential equation using forward and backward Fast Fourier Transforms (FFT's). The computation centers around 1D FFT's 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, transpose the array, if necessary, so that another dimension is local. In the 2D layout, there are four transposes, one between each of the three FFT's in both the forward and backward directions. In the 1D layout, we need only transpose the array twice since two of the dimensions are kept local.

In the NAS FT benchmark, the array on which we compute the FFT's is not a cube. Thus, to achieve a load-balanced program, we use arrays that are distributed in different ways. The remap operator is a perfect choice for transposing from one array to another especially given the different distributions.

In the case of a 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

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

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

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

These same two lines of code require well over fifty for the Fortran + MPI implementation (shown in Appendix A.2). In Fortran + MPI, instead of regions, loops guide the computation and the 3×3 array dims stores the different dimension lengths for the transposed arrays. Communication is not induced by operators, but is specified with MPI function calls.

The 3D transpose in the NAS FT benchmark is not an obvious piece of code. Even the ZPL version requires some thought! Consider the following reasonable attempt to write the first transpose:

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

At first glance, this code may appear correct. The region of the statement is the region over which X1 is allocated and specifies the new layout: 2, 3, 1. The index maps match this layout. However, since X1 is allocated over RXYZ we must index into its first dimension using indices ranging over one to nx. In the region that applies to the statement, these indices are in the third dimension. The same reasoning applies to the other two dimensions. Note that this line of reasoning must also be followed by the Fortran programmer although the result is more convoluted.

4.3 NAS CG Row Column Transpose

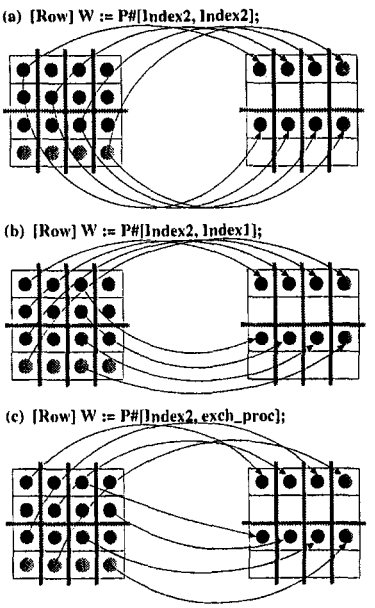


Figure 2: An illustration of three alternative communication patterns on a 2×4 processor grid, each induced by a different implementation of the transposition of a flooded column vector to a flooded row vector in the NAS CG benchmark: (a) an implementation where the map arrays are readable over the region of computation, (b) an implementation using the Index1 array that looks more like the standard transpose, and (c) an optimal implementation where the index is chosen so as to duplicate the clever trick in which each processor communicates with at most one other processor.

The NAS CG benchmark [2, 3] estimates the largest eigenvalue of a symmetric positive definite sparse matrix by the inverse power method. The main iteration loop contains a sparse matrix vector multiplication, several reductions, and a column to row transpose. It is in this transpose that we are primarily interested. A clever trick is used in the Fortran + MPI code (shown in Appendix B.2) in which each processor needs only to communicate with at most one other processor when using a $k \times k$ or $k \times 2k$ processor grid where k is a power of 2. In ZPL, we duplicate this trick, but it is worthwhile to consider more basic alternatives first.

We start with the following definitions:

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

We can transpose the values in P to W using the following line of code:

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

Since the second dimension of P is flooded, the second map array is irrelevant. The transpose stems from using the indices ranging over the second dimension of region Row to index into the first dimension of P. Using Index2 in the second dimension may appear to be a reasonable choice because

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: The basic implementation of the Gather and Scatter operators.

it takes on different values as we traverse the region. Figure 2(a) illustrates this transpose assuming a 2×4 processor grid. Note the inefficient communication pattern in which there is no one-to-one mapping between processors. As we increase the number of processors, this pattern becomes significantly worse. We can do better with the following code:

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

Figure 2(b) illustrates this second approach. Here is the distinction. Since W is flooded in the first dimension, every processor must write the same value to its representative element. Thus the map arrays must also, in general, be flooded in the first dimension. Otherwise different processors would potentially read different values in the map arrays and might ultimately get different values from the source array. However, since the source array is flooded in the second dimension, the value of the second map is irrelevant; it affects only performance. Using `Index1` results in a communication pattern that more closely achieves a one-to-one communication pattern, and does achieve just this with a $k \times k$ processor grid.

If each processor specifies an index in place of the second map that would point it to the processor with which it should communicate, then we can duplicate the clever one-to-one communication pattern implemented by the Fortran + MPI benchmark writers. The ZPL code is as follows:

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

The scalar variable, `exch_proc`, is set so that on each processor it contains an index specifying a position on a unique processor. If *rows* and *cols* equal the number of row and column processors and *row* and *col* identify the computing processor, then we set `exch_proc` with the formula

$$row \times \frac{n}{cols} + \; col \bmod \frac{rows}{cols} \; \times \frac{n}{rows} + 1.$$

The communication pattern induced by this approach is illustrated in Figure 2(c).

4.4 Implementation

The implementation of the general remap operator is nontrivial. 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 base-line implementation of both gather and scatter versions of the remap operator. These implementations are identical through 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 of Figure 3. 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 there are some 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.

These 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. In a clever implementation of the gather we could start to read from the source array as soon as we have the indices from any processor. Likewise in the scatter we could start to write to the destination array as soon as we have data from any processor. 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 4.1.2, and the 3D transpose of Section 4.2. However, it is unclear which is preferable in these situations. Nonetheless, the importance of the 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 communication 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 repeated computation 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. We call this optimization *map saving* since we save the map 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 with another optimization called *map sharing*. In this optimization, the map information is shared between remap operators that 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 inside and outside of 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 detailed 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 and there is typically less to communicate.

This optimization cannot be applied by the MPI programmer using the monolithic MPI_Alltoall, MPI_Scatter, and MPI_Gather intrinsics. Of course, the optimization would have no effect if the ZPL implementation were based on these MPI routines.

4.5.3 Run Length Encoding

Stylized collective communication patterns like those listed 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 a *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 such 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 strided run length encoding so we can encode the encoding if this will benefit us. In our implementation, by default, we base the number of recursive encodings on the rank of the remap operator. So if there are three map arrays, we encode an encoded encoding. This choice is based on the optimal number of encodings we would need for the basic redistribution pattern of Section 4.1.2.

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. The stride applies element-wise to the pair so the next pair is (2, 1). 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 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 enough memory to store the final result, 8 integers in this case, and some constant amount of additional memory for the computations. Because we encode multiple streams of indices at a time, one stream for each processor we need to communicate with, we cannot simply use variables for this computation, but rather need a buffer array.

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 may 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 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 shown in Appendix A.2. 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 copy is dense, we know it is straight because, in these latter copies, we are copying the data in order.

This optimization, performed in the ZPL runtime, is equivalent to the straight-forward approach taken by the Fortran + MPI programmer in the context of the NAS CG transpose. Due to the dynamic nature, the ZPL implementation necessarily suffers from some overhead. More interestingly, this optimization fired in certain configurations of the NAS FT benchmark that we did not expect. We discuss this further in Section 5 where we evaluate our implementation of the remap operator.

5. EVALUATION

In this section, we evaluate our implementation of the remap operator in the context of the NAS CG and NAS FT benchmarks. The NAS parallel benchmarks are a suite of scientific applications and kernels representative of codes scientists write for parallel computers [2, 3]. The Fortran and MPI provided implementations are highly-tuned. We compare the NAS codes qualitatively first, then examine differences in memory usage and execution time.

5.1 Expressiveness

Throughout this paper, we have argued that the remap operator and ZPL's high-level constructs make the programmer's job easier. Figure 4 contains a breakdown of the lines of code in the ZPL and Fortran + MPI implementations. While lines of code is not even close to being a perfect metric for expressive power, it does yield some useful information. The ZPL implementations of both the NAS FT and NAS CG benchmarks are written with less than half the number of lines uses to write the Fortran + MPI versions. The figures show a breakdown of the lines of code into those used for declarations, the actual computation, and communication. The high-level approach of ZPL eliminates the need for the programmer to specify details of communication. The computation was written with significantly fewer lines because of ZPL's powerful array syntax based on the region. The reduction in lines is especially great for the NAS CG benchmark because of ZPL's support for sparse arrays [8].

5.2 Memory Usage

Execution time is not the only important metric. It is frequently the case that 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 optimiza-

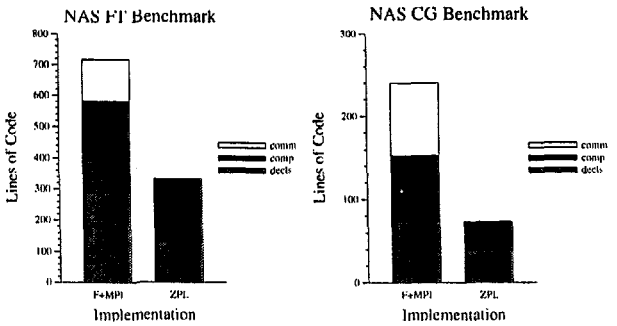


Figure 4: The number of lines of code in the Fortran + MPI and ZPL versions of the NAS FT and NAS CG benchmarks broken down into lines used for communication, computation, and declarations.

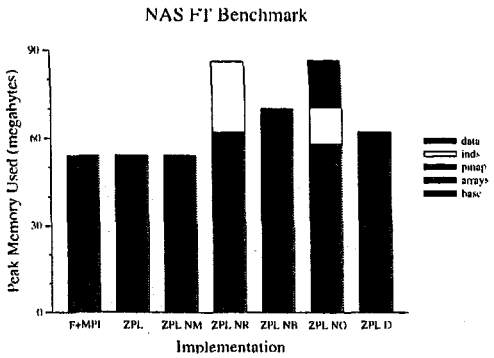


Figure 5: The effect of the various remap optimizations on the peak memory usage during execution of a remap for class C of the NAS FT benchmark run on 256 processors of a Cray T3E. The memory is subdivided into five uses: general program data, the three large arrays, the remap processor map, the remap indices, and the remap data buckets.

tions discussed in this paper on the total memory usage for class C ($512 \times 512 \times 512$ arrays, the largest size) of the NAS FT benchmark running on 256 processors of a Cray T3E. The memory needed to implement the remap in the NAS CG benchmark is insignificant regardless of the optimizations because the amount of data movement is relatively insignificant. The optimized ZPL implementation (ZPL) uses nearly the same amount of memory as the Fortran + MPI (F+MPI); the memory usage is broken down into the memory needed for the three major arrays and for the rest of the program including a massive lookup table. The optimized ZPL overhead, on the order of 84K, is small enough so as to not show up in the chart.

Disabling the map saving optimization (ZPL NM) saves the memory used for storing the map for the forward transpose. In the benchmark, we do a forward transpose followed by twenty backward transposes. The map is recalculated for the backward transposes, but is not saved for the forward transpose, a savings in memory on the order of 42K.

Run length encoding proves crucial for reducing the memory footprint. The three indices per position and the processor map use half the memory needed to store the three major arrays of the computation. Disabling run length encoding (ZPL NR) increases the memory needed to save the two maps from about 84K to 24M. Compared to disabling all the optimizations (ZPL NO), including map saving, we use significantly more memory for the map. We make up this loss with the dead source/destination reuse optimization which eliminates the data buckets. Disabling this optimization (ZPL NB) increases the memory usage by 16M.

As mentioned at the end of Section 4.5.5, the direct sending/receiving optimization works under certain conditions for the NAS FT benchmark. The processor grid used for this benchmark is a $p \times 1 \times 1$ grid if p is less than or equal to nx, after which a $p - nx \times nx \times 1$ grid is used, where p is a power of two. When $p \geq nx$, the copy from the source array is a straight copy, and the direct optimization applies. This optimization has no effect with less than 512 processors for class C, but if it did, the direct optimization could be used at the expense of the dead source/destination reuse optimization. The memory use would increase by 8M (ZPL D) for the one bucket. In our current implementation, we give the direct optimization priority, however, we are looking into whether a scheme based on dynamic profiling or an inspector/executor model would be worthwhile.

5.3 NAS FT Speedup

Figures 6-9 show results for the NAS FT and NAS CG benchmarks for the class C problem size on increasing numbers of processors of a Cray T3E. We use speedup graphs except where more information can be gleamed from a graph of execution times. We calculate the speedups over the best implementation time on the fewest number of processors for which any implementation could complete without exhausting the memory or our time allotment. (Complete results for classes A, B, and C are in Appendix C; raw times are listed in Appendix D.) The ZPL implementation is based on C and MPI. Though ZPL would likely exhibit improved performance with the SHMEM library, time constraints have limited our work.

Figure 6(a) compares the ZPL and Fortran + MPI implementations of the NAS FT benchmark. The ZPL implementation exhibits noticeable overhead stemming mostly from the transpose, shown in Figure 6(b). This overhead in the ZPL implementation can be attributed completely to the cache blocking done by hand in the Fortran + MPI code. In Figure 6(c), we undo the cache blocking (F+MPI NB), and the ZPL implementation becomes noticeably faster. Now the overhead lies in the Fortran + MPI implementation's local copying between source and destination arrays. In ZPL, these copies are especially fast because of prior work done in computing the maps.

In Figure 7(a) we look at the effect of the map saving optimization on ZPL. Disabling this optimization (ZPL NM) decreases the execution time by nearly a factor of two. In this graph, ZPL AMORTIZED refers to an implementation of ZPL in which the maps are pre-computed. On the first transpose, the maps are initially calculated adding some overhead to the ZPL implementation. As the graph indicates, in only twenty iterations, the optimization is almost fully amortized.

Disabling run length encoding (ZPL NR), Figure 7(b), slows execution time significantly. Further, if the indices are not encoded, the ZPL version cannot run on 64 processors. The importance of the full suite of optimizations is shown in Figure 7(c) where we compare the ZPL implemen-

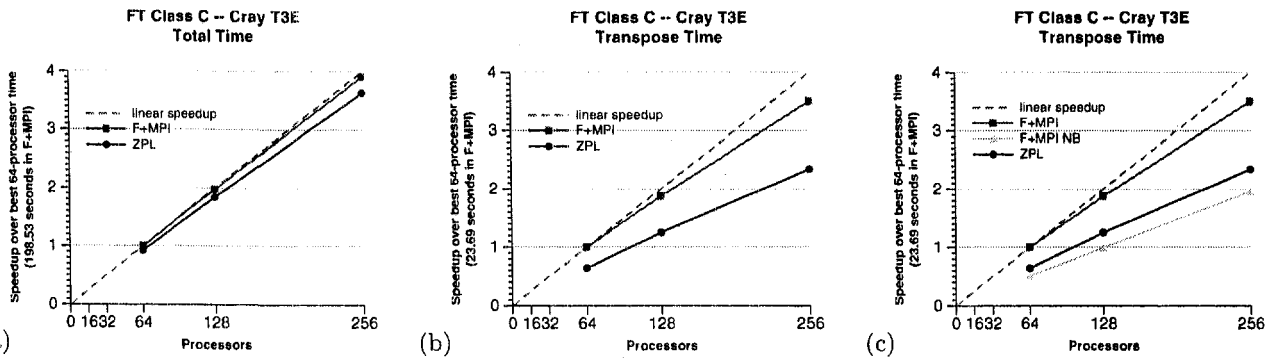


Figure 6: Speedup graphs for class C of the NAS FT benchmark showing the (a) total time, (b) transpose time, and (c) effect of the hand-coded cache-blocking optimization on the transpose time.

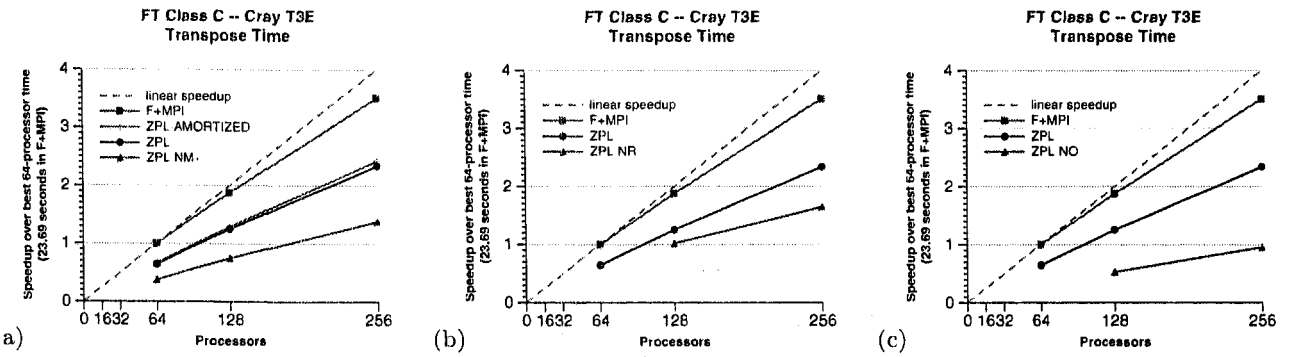


Figure 7: Speedup graphs for class C of the NAS FT benchmark showing the effect of the (a) map saving, (b) run length encoding, and (c) suite of optimizations on the transpose time.

tation without the remap optimizations (ZPL NO). The direct sending/receiving optimizations have no impact on class C until there are 512 processors, but do show up for class A at 128 processors and class B at 256 processors. The dead source/destination reuse optimization had a critical impact in decreasing the memory footprint, but did not affect execution time.

5.4 NAS CG Speedup

The transpose in NAS CG differs from the one in NAS FT in a number of ways, two of which are worth mentioning. First, the transpose in NAS CG results in a one-to-one communication pattern as opposed to the all-to-all communication of NAS FT. Second, there is little data movement in NAS CG, but much in NAS FT.

Figure 8(a) compares the ZPL and Fortran + MPI implementations. Nearly all the noticeable overhead in this graph stems from the sparse matrix-vector multiplication. In ZPL, the programmer's job is made easier with the built-in support for sparse arrays, but the final code is slightly less efficient. For a discussion of this support, the reader is referred to the literature [8]. In this paper we focus on the relatively insignificant transposition. The transpose time of the ZPL and Fortran + MPI implementations as well as that for the baseline ZPL (ZPL NO) is charted in Figure 8(c). More overhead is exhibited in the ZPL version than was seen for the NAS FT benchmark, a result of the critical nature of the direct sending/receiving optimization which has inherently more overhead from the detection process and the decreased data motion which exposes more overhead. As a whole, the optimizations are seen to be more critical resulting in about a factor of 6 to 8 slowdown.

Figure 8(b) shows the variations for the ZPL transpose discussed in Section 4.3. The simple method of using the `Index2/Index1` transpose (ZPL I2I1) shows comparable performance to the more complicated scheme that results in an exactly one-to-one communication pattern, and, of course, the performance is identical on a $k \times k$ processor grid. The transpose using `Index2` twice (ZPL I2I2) has noticeably worse performance because the communication pattern is inefficient.

Figure 9 shows the effect of disabling any one of the (a) map saving (ZPL NM), (b) run length encoding (ZPL NR), and (c) direct sending/receiving (ZPL D) optimizations. Each of these optimizations is critical to performance.

6. CONCLUSIONS

ZPL's remap operator possesses great power, yet retains an efficient implementation. The operator is more versatile than those provided even by APL, and is significantly more general than ZPL's other operators, including the reduction and flood operators discussed in this paper. Remap arguably subsumes these other operators because we can easily rewrite any reduction or flood with a remap. However, the purpose of ZPL is not only to communicate with the machine, but also with the programmer. The reduction and flood operators indicate, to both the programmer and the implementer, specific communication patterns with efficient implementations that may be based on the parallel

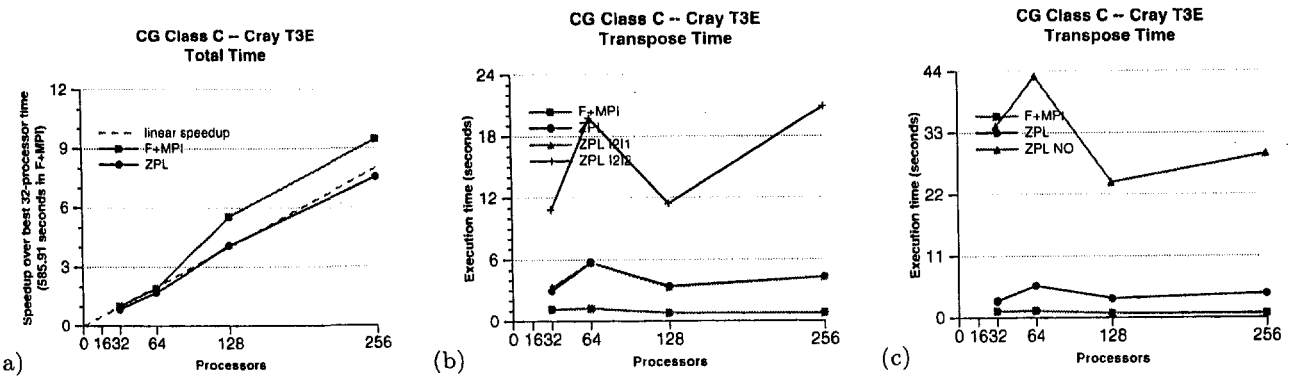


Figure 8: Speedup graphs for class C of the NAS CG benchmark showing the (a) total time, (b) transpose time for the variations of Section 4.3, and (c) effect of the suite of optimizations on the transpose time.

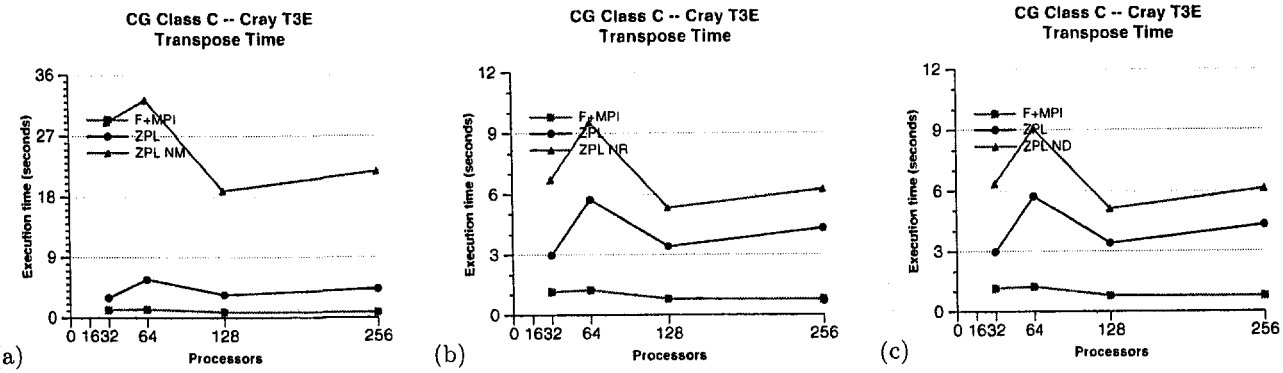


Figure 9: Speedup graphs for class C of the NAS CG benchmark showing the effect of the (a) map saving and sharing, (b) run length encoding, and (c) direct sending and receiving optimizations on transpose time.

prefix algorithm and/or special hardware.

Rather than replace the other operators, remap compliments them in creating a small suite of parallel array operators that one could use to write scalable, high-performance, parallel codes. In the end, though, remap remains a catch-all operator for ZPL, to be used when other operators do not suffice. The implementation described in this paper lessens the cost of remap's wide applicability. Through optimizations such as map saving, communication/computation overlap, run length encoding, dead array reuse, and direct communication, our implementation produces code comparable in performance to hand-tuned Fortran and MPI.

7. 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, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical report, NASA Ames Research Center (RNR-94-007), March 1994.

[3] 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.

[4] R. Barriuso and A. Knies. SHMEM user's guide. Technical report, May 1994.

[5] B. L. Chamberlain. The design and implementation of a region-based parallel language. Technical report, University of Washington (Ph.D. Thesis), November 2001.

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

[7] B. L. Chamberlain, E. C. Lewis, 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.

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

[9] S. J. Deitz, B. L. Chamberlain, and L. Snyder. High-level language support for user-defined reductions. In *Proceedings of the Los Alamos Computer Science Institute Symposium*, 2001.

[10] A. D. Falkott and K. E. Iverson. *APL/360 User's Manual.* IBM Corporation, 1968.

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

[12] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: the complete reference.*

MIT Press, Cambridge, MA, USA, 1996.

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

APPENDIX

A. NAS FT 3D TRANSPOSE

The ZPL and Fortran + MPI codes for implementing the 3D transpose in the NAS FT benchmark follow.

A.1 ZPL Version

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

A.2 Fortran Version

```
1   subroutine transpose_x_yz(l1, l2, xin, xout)
2   implicit none
3   include 'global.h'
4   integer l1, l2
5   double complex xin(ntotal/np), xout(ntotal/np)
6   call transpose2_local(dims(1,l1),
   >    dims(2, l1)*dims(3, l1), xin, xout)
7   call transpose2_global(xout, xin)
8   call transpose2_finish(dims(1,l1),
   >    dims(2, l1)*dims(3, l1), xin, xout)
9   return
10  end

11  subroutine transpose_xy_z(l1, l2, xin, xout)
12  implicit none
13  include 'global.h'
14  integer l1, l2
15  double complex xin(ntotal/np), xout(ntotal/np)
16  call transpose2_local(dims(1,l1)*dims(2, l1),
   >    dims(3, l1), xin, xout)
17  call transpose2_global(xout, xin)
18  call transpose2_finish(dims(1,l1)*dims(2, l1),
   >    dims(3, l1), xin, xout)
19  return
20  end

21  subroutine transpose2_local(n1, n2, xin, xout)
22  implicit none
23  include 'mpinpb.h'
24  include 'global.h'
25  integer n1, n2
26  double complex xin(n1, n2), xout(n2, n1)
27  double complex z(transblockpad, transblock)
28  integer i, j, ii, jj
29  if (n1 .lt. transblock .or. n2 .lt. transblock) then
30     if (n1 .ge. n2) then
31        do j = 1, n2
32           do i = 1, n1
33              xout(j, i) = xin(i, j)
34           end do
35        end do
36     else
37        do i = 1, n1
38           do j = 1, n2
39              xout(j, i) = xin(i, j)
40           end do
41        end do
42     endif
43  else
44     do j = 0, n2-1, transblock
45        do i = 0, n1-1, transblock
46           do jj = 1, transblock
47              do ii = 1, transblock
48                 z(jj,ii) = xin(i+ii, j+jj)
49              end do
50           end do
51           do ii = 1, transblock
52              do jj = 1, transblock
53                 xout(j+jj, i+ii) = z(jj,ii)
54              end do
55           end do
56        end do
57     end do
58  endif
59  return
60  end

61  subroutine transpose2_global(xin, xout)
62  implicit none
63  include 'global.h'
64  include 'mpinpb.h'
65  double complex xin(ntotal/np)
66  double complex xout(ntotal/np)
67  integer ierr
68  call mpi_alltoall(xin, ntotal/(np*np), dc_type,
   >    xout, ntotal/(np*np), dc_type,
   >    commslice1, ierr)
69  return
70  end

71  subroutine transpose2_finish(n1, n2, xin, xout)
72  implicit none
73  include 'global.h'
74  integer n1, n2, ioff
75  double complex xin(n2, n1/np2, 0:np2-1),
   >    xout(n2*np2, n1/np2)
76  integer i, j, p
77  do p = 0, np2-1
78     ioff = p*n2
79     do j = 1, n1/np2
80        do i = 1, n2
81           xout(i+ioff, j) = xin(i, j, p)
82        end do
83     end do
84  end do
85  return
86  end

87  call transpose_xy_z(2, 3, x1, x2)
    ...
88  call transpose_x_yz(3, 2, x1, x2)
```

B. NAS CG COLUMN ROW TRANSPOSE

The ZPL and Fortran + MPI codes for implementing the column to row transpose in the NAS CG benchmark follow.

B.1 ZPL Version

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

B.2 Fortran Version

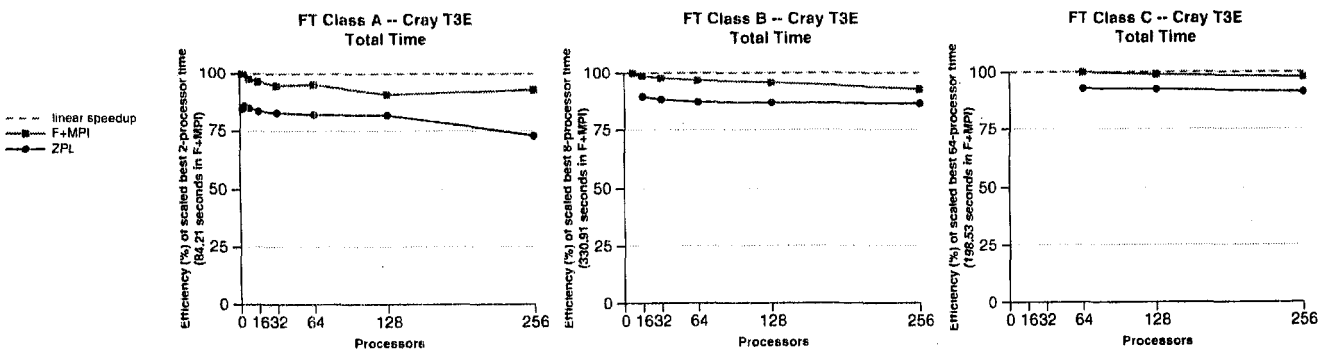
```
1   if( l2npcols .ne.  0 )then
2      call mpi_irecv(q, exch_recv_length,
    >      dp_type, exch_proc, 1,
    >      mpi_comm_world, request, ierr)
3      call mpi_send(w(send_start), send_len,
    >      dp_type, exch_proc, 1,
    >      mpi_comm_world, ierr)
4      call mpi_wait( request, status, ierr )
5   else
6      do j=1,exch_recv_length
7         q(j) = w(j)
8      enddo
9   endif
```

C. EXPERIMENTAL RESULTS

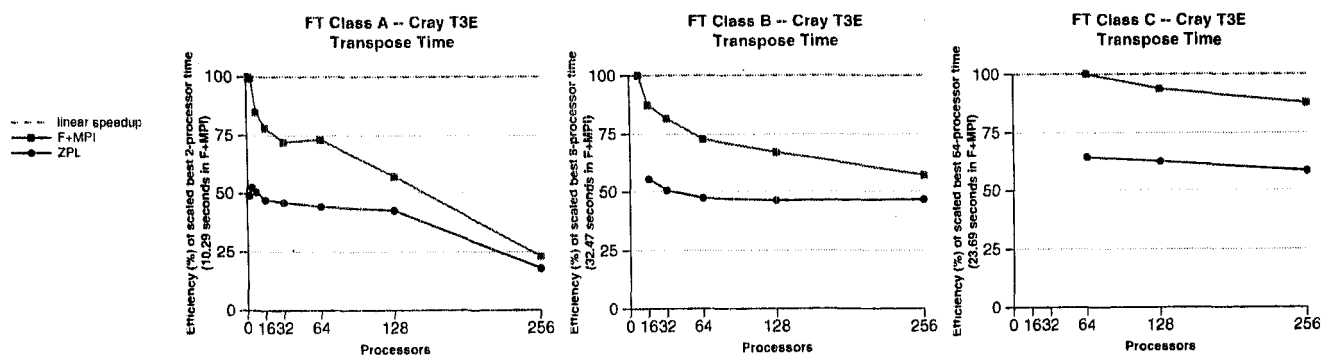
The following graphs show the complete results (classes A, B, and C) for our experiments discussed in Section 5. We show efficiency graphs except where more information can be gleamed from graphs of execution times. We calculate the efficiencies against the perfectly-scaled time of the best implementation time on the fewest number of processors for which any implementation could complete without exhausting the memory or our time allottment.

C.1 NAS FT Results

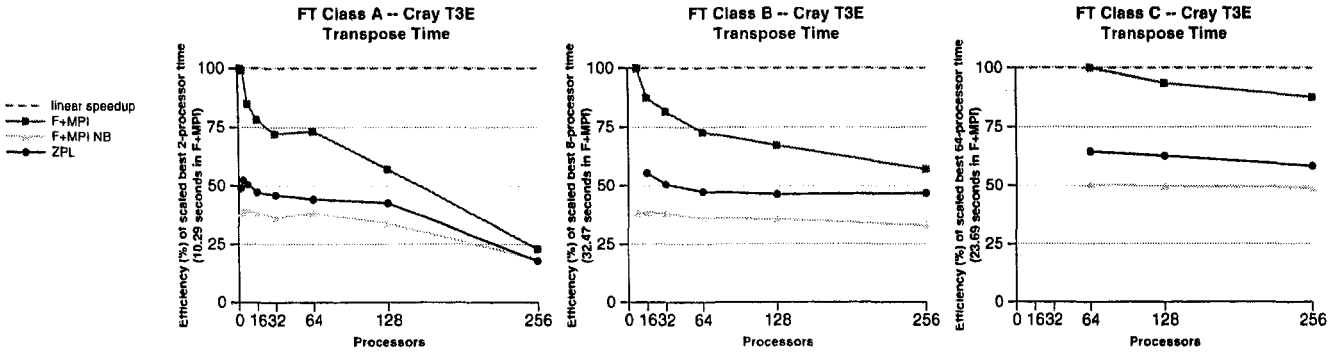
The graphs below compare the total execution time of the ZPL and Fortran + MPI implementations.



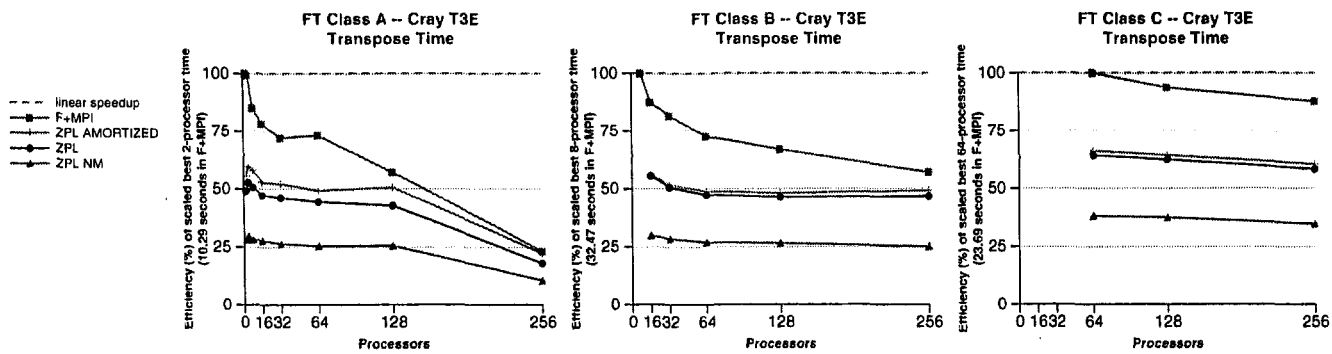
The following graphs compare the performance of the transpose part of the benchmark.



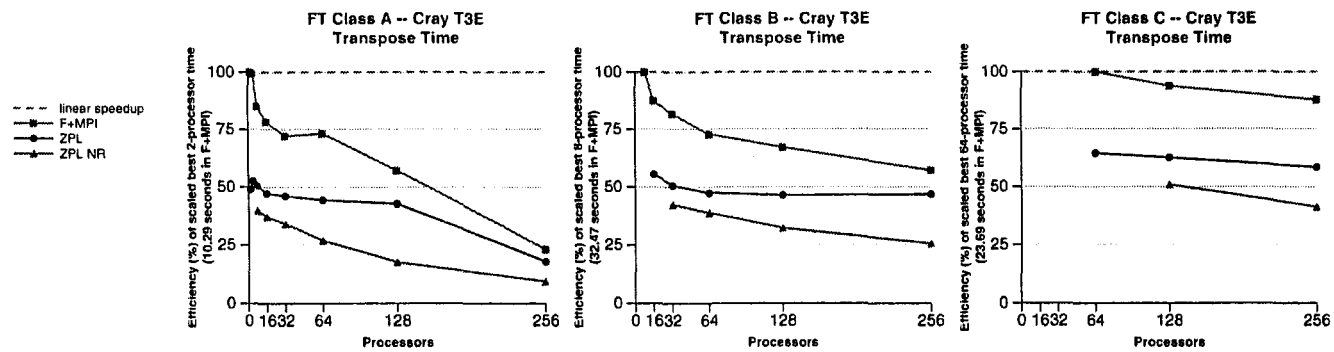
The following graphs show the effect of the cache-blocking optimization in the Fortran + MPI implementation.



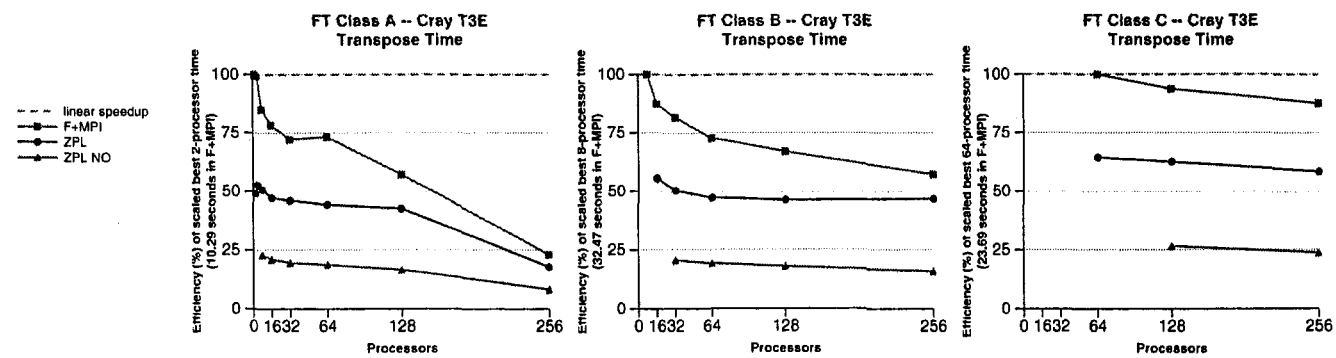
The following graphs show the effect of the map saving optimization.



The following graphs show the effect of the run length encoding optimization.

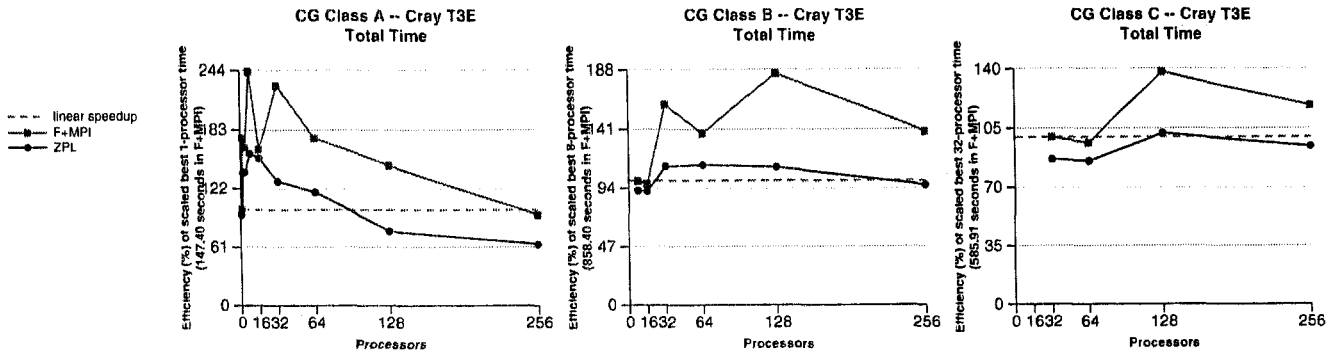


The following graphs show the total effect of the optimizations discussed in this paper.

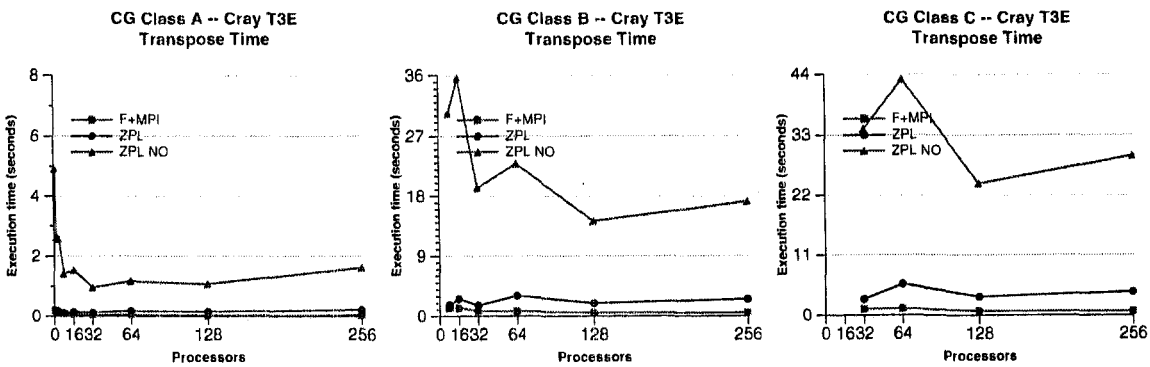


C.2 NAS CG Results

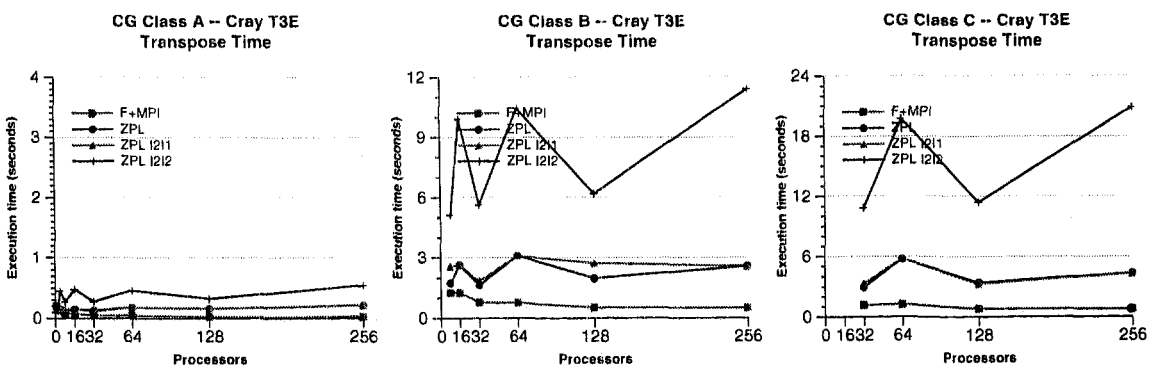
The graphs below compare the total execution time of the ZPL and Fortran + MPI implementations.



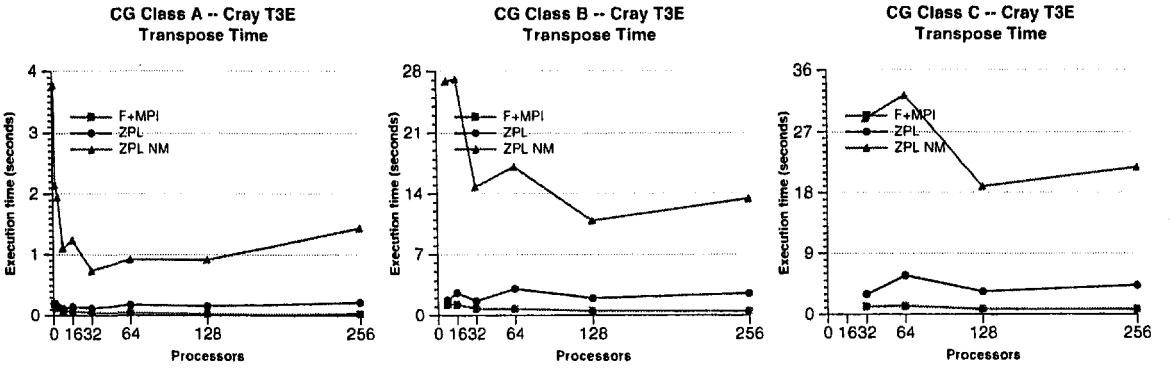
The following graphs compare the performance of the transpose part of the benchmark and show the total effect of the optimizations discussed in this paper.



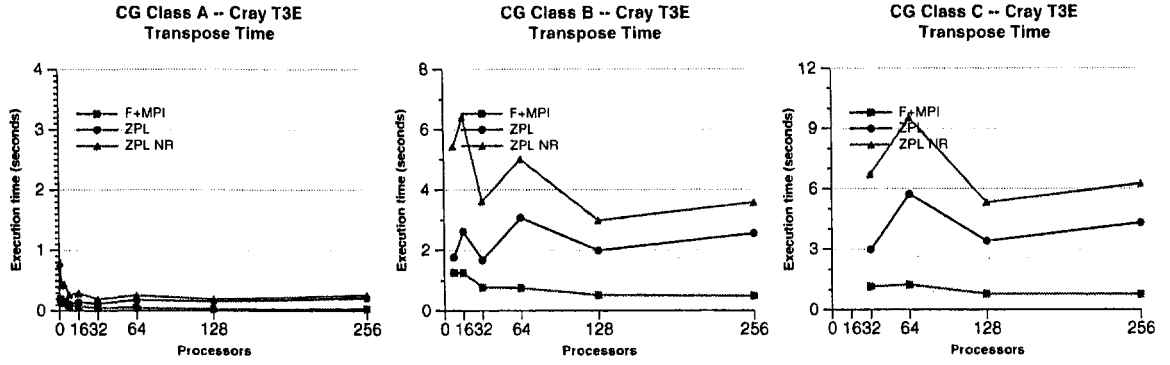
The following graphs compare the various ZPL implementations of the transpose.



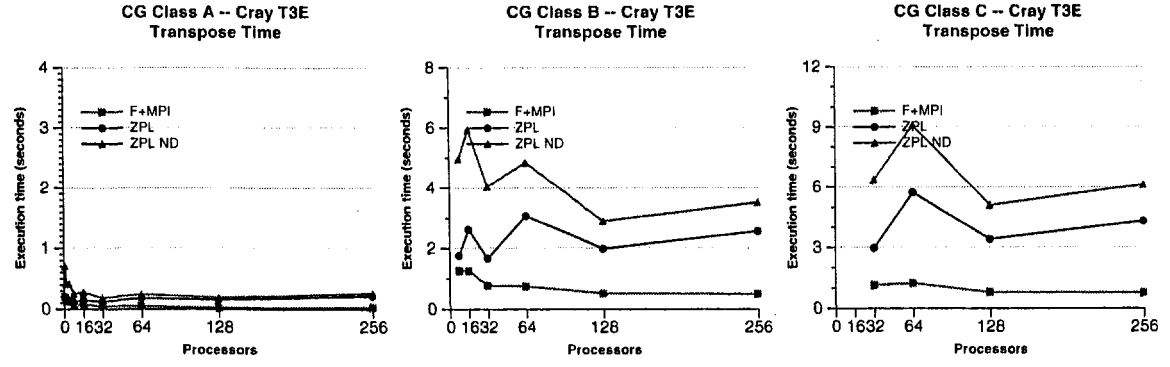
The following graphs show the effect of the map saving and sharing optimizations.



The following graphs show the effect of the run length encoding optimization.



The following graph show the effect of the direct sending and receiving optimizations.



D. EXPERIMENTAL TIMINGS

The following tables contain the minimum observed times for each configuration of the experiments reported on in Section 5 and Appendix C.

FT Class A - Cray T3E (Total Time)

processors	2	4	8	16	32	64	128	256
F+MPI	84.209	42.138	21.560	10.909	5.566	2.769	1.450	0.710
F+MPI NB	100.841	50.283	24.997	12.619	6.394	3.170	1.642	0.789
ZPL	99.477	48.967	24.737	12.562	6.339	3.199	1.605	0.903
ZPL NM	115.429	56.775	28.885	14.505	7.383	3.752	1.876	1.228
ZPL NR	—.—	—.—	25.998	13.304	6.800	3.650	2.173	1.284
ZPL NO	—.—	—.—	30.907	15.971	8.178	4.225	2.210	1.417
ZPL AMORTIZED	96.893	47.892	24.106	12.253	6.169	3.112	1.550	0.816

FT Class A - Cray T3E (Transpose Time)

processors	2	4	8	16	32	64	128	256
F+MPI	10.293	5.188	3.027	1.643	0.892	0.439	0.281	0.350
F+MPI NB	26.949	13.312	6.434	3.347	1.768	0.840	0.474	0.431
ZPL	21.031	9.780	5.080	2.725	1.396	0.725	0.377	0.452
ZPL NM	36.660	17.449	9.169	4.654	2.446	1.268	0.631	0.776
ZPL NR	—.—	—.—	6.456	3.466	1.894	1.198	0.912	0.870
ZPL NO	—.—	—.—	11.235	6.117	3.275	1.711	0.966	0.974
ZPL AMORTIZED	18.548	8.604	4.431	2.440	1.236	0.653	0.318	0.363

FT Class B - Cray T3E (Total Time)

processors	8	16	32	64	128	256
F+MPI	330.907	167.841	84.769	42.862	21.665	11.183
F+MPI NB	383.536	191.053	96.059	48.524	24.312	12.465
ZPL	—.—	184.957	93.796	47.481	23.827	11.964
ZPL NM	—.—	210.262	106.583	53.931	27.128	13.829
ZPL NR	—.—	—.—	96.791	49.282	25.741	13.755
ZPL NO	—.—	—.—	117.338	59.825	30.752	16.154
ZPL AMORTIZED	—.—	184.691	93.222	47.194	23.662	11.835

FT Class B - Cray T3E (Transpose Time)

processors	8	16	32	64	128	256
F+MPI	32.466	18.565	9.941	5.576	3.025	1.773
F+MPI NB	85.364	41.963	21.278	11.233	5.703	3.085
ZPL	—.—	29.228	16.053	8.566	4.381	2.172
ZPL NM	—.—	54.490	28.625	15.091	7.666	4.048
ZPL NR	—.—	—.—	19.186	10.464	6.318	3.980
ZPL NO	—.—	—.—	39.513	20.983	11.250	6.356
ZPL AMORTIZED	—.—	28.822	15.733	8.286	4.220	2.062

FT Class C - Cray T3E (Total Time)

processors	64	128	256
F+MPI	198.529	100.312	50.717
F+MPI NB	221.686	111.393	56.064
ZPL	214.605	107.772	54.579
ZPL NM	240.188	120.640	61.675
ZPL NR	—.—	112.174	58.751
ZPL NO	—.—	133.824	69.359
ZPL AMORTIZED	213.452	107.031	54.164

FT Class C - Cray T3E (Transpose Time)

processors	64	128	256
F+MPI	23.686	12.639	6.752
F+MPI NB	46.922	23.829	12.107
ZPL	36.751	18.915	10.123
ZPL NM	62.315	31.656	17.135
ZPL NR	—.—	23.293	14.385
ZPL NO	—.—	44.703	24.907
ZPL AMORTIZED	35.668	18.352	9.758

CG Class A - Cray T3E (Total Time)

processors	1	2	4	8	16	32	64	128	256
F+MPI	147.399	42.216	22.447	7.588	5.683	2.019	1.321	0.791	0.609
ZPL	155.291	53.358	26.573	11.679	6.026	3.556	1.946	1.473	0.900
ZPL NM	160.675	56.551	28.965	13.060	7.300	4.291	2.724	2.272	2.150
ZPL NR	157.546	54.607	27.342	12.108	6.329	3.811	2.042	1.583	0.955
ZPL ND	156.261	54.087	27.049	12.033	6.243	3.716	2.038	1.550	0.950
ZPL NO	161.559	56.795	29.402	13.210	7.466	4.424	2.923	2.367	2.305
ZPL I2I2	155.916	53.829	27.092	12.085	6.483	3.837	2.334	1.695	1.283
ZPL I2I1	155.831	53.885	26.791	11.951	6.141	3.684	1.992	1.533	0.925

CG Class A - Cray T3E (Transpose Time)

processors	1	2	4	8	16	32	64	128	256
F+MPI	0.198	0.128	0.126	0.078	0.075	0.047	0.046	0.026	0.028
ZPL	0.200	0.143	0.149	0.116	0.146	0.123	0.178	0.156	0.214
ZPL NM	3.774	2.145	1.951	1.105	1.241	0.733	0.920	0.909	1.436
ZPL NR	0.789	0.432	0.445	0.266	0.294	0.200	0.256	0.194	0.261
ZPL ND	0.706	0.416	0.422	0.252	0.277	0.186	0.242	0.195	0.259
ZPL NO	4.910	2.624	2.560	1.407	1.533	0.962	1.170	1.076	1.610
ZPL I2I2	0.201	0.142	0.450	0.268	0.477	0.275	0.461	0.321	0.550
ZPL I2I1	0.200	0.196	0.149	0.155	0.146	0.134	0.177	0.160	0.216

CG Class B - Cray T3E (Total Time)

processors	8	16	32	64	128	256
F+MPI	858.405	440.781	133.984	78.033	28.934	19.279
ZPL	931.290	468.454	193.313	95.748	48.582	27.822
ZPL NM	965.250	497.998	209.422	111.673	58.915	39.715
ZPL NR	951.269	475.753	195.777	98.758	50.054	29.803
ZPL ND	952.145	475.015	198.186	99.031	50.999	29.187
ZPL NO	977.309	503.068	210.391	115.029	60.812	41.047
ZPL I2I2	952.405	479.898	200.653	106.481	54.699	37.979
ZPL I2I1	949.701	471.528	196.332	97.410	51.104	28.534

CG Class B - Cray T3E (Transpose Time)

processors	8	16	32	64	128	256
F+MPI	1.264	1.258	0.780	0.757	0.521	0.494
ZPL	1.758	2.623	1.669	3.080	1.986	2.559
ZPL NM	26.865	27.117	14.782	17.119	10.900	13.479
ZPL NR	5.439	6.419	3.613	5.008	2.986	3.577
ZPL ND	4.961	5.936	4.038	4.844	2.902	3.521
ZPL NO	30.377	35.620	19.232	22.893	14.264	17.203
ZPL I2I2	5.124	9.906	5.641	10.446	6.194	11.401
ZPL I2I1	2.570	2.614	1.850	3.079	2.713	2.552

CG Class C - Cray T3E (Total Time)

processors	32	64	128	256
F+MPI	585.906	303.845	105.895	61.941
ZPL	678.068	343.936	143.390	77.602
ZPL NM	714.300	373.965	162.857	96.736
ZPL NR	690.091	348.150	149.784	83.218
ZPL ND	690.182	349.968	148.285	80.848
ZPL NO	716.261	378.467	164.740	102.905
ZPL I2I2	694.190	362.423	155.265	97.550
ZPL I2I1	686.806	346.857	146.884	79.143

CG Class C - Cray T3E (Transpose Time)

processors	32	64	128	256
F+MPI	1.158	1.244	0.799	0.777
ZPL	2.978	5.733	3.411	4.316
ZPL NM	29.016	32.351	18.931	21.785
ZPL NR	6.720	9.528	5.321	6.253
ZPL ND	6.358	9.082	5.113	6.116
ZPL NO	34.067	43.117	24.160	29.237
ZPL I2I2	10.837	19.741	11.417	20.890
ZPL I2I1	3.317	5.733	3.297	4.313