Array Language Support for Parallel Sparse Computation *

Bradford L. Chamberlain
University of Washington
Dept of CSE; Box 352350
Seattle, WA 98195-2350 USA
brad@cs.washington.edu

Lawrence Snyder
University of Washington
Dept of CSE; Box 352350
Seattle, WA 98195-2350 USA
snyder@cs.washington.edu

ABSTRACT

This paper describes an array-based language-level approach to parallel sparse computation. Our approach is unique due to its separation of sparse index sets from arrays, both syntactically and in the implementation. This design allows users to express their computation using dense array syntax, making the code easier for readers to understand and for compilers to parallelize and optimize. This work is done within the context of Advanced ZPL, retaining its crisp syntax and source-level performance model. Our implementation uses a novel sparse storage format that supports general operations such as arbitrary iteration and slicing. We describe how our compiler automatically optimizes this data structure into more compact forms based on the operations required by the program. We demonstrate our approach using the NAS CG and MG benchmarks, comparing our implementations with the original Fortran+MPI versions in terms of clarity and performance. We present performance results on the Cray T3E indicating that our implementation compares favorably to the hand-coded NAS versions in terms of memory requirements and often surpasses them in terms of execution speed.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; E.1 [**Data**]: Data Structures

General Terms

Languages, Design, Performance, Experimentation

Keywords

sparse array, sparse matrix, parallel computing, parallel language, Advanced ZPL, MPI, NAS Parallel Benchmarks

*This research was supported by a scholarship from the USENIX Association and a grant of HPC time from the Arctic Region Supercomputing Center.

0	0	0	0	0	0
0	3	0	4	0	9
0	0	0	0	0	0
0	0	5	0	2	0
0	1	0	0	0	0
0	0	0	7	0	0

Figure 1: A sample sparse array in which $d = 2$, $n = 6$, $nnz = 7$, and the IRV is 0.

1. INTRODUCTION

The *sparse array* is a fundamental data structure in scientific computing due to the pervasiveness of sparsity in the universe and the models that scientists use to represent aspects of it. Sparse arrays are commonly used to represent *sparse matrices*, which have innumerable applications in mathematics, engineering, and the sciences. Sparse arrays can also represent *structural sparsity* such as the placement of particles in space or of coastlines on the earth's surface. Furthermore, *sparse iteration* may be used with traditional dense arrays in order to focus on particular values of interest.

Since sparse scientific applications typically have high computational demands, it is often desirable to apply parallelism to them in order to employ memory and cycles beyond that which typical uniprocessors can provide. In this paper we present a novel approach to portable parallel sparse computation that provides a combination of clarity, performance, and semantic richness not present in other approaches. This work is done within the context of the Advanced ZPL (A-ZPL) parallel programming language, which is under development at the University of Washington and freely available on the web[1].

For this paper, we consider a sparse array to be an array of arbitrary dimension d, in which a single value appears sufficiently frequently that it benefits the user to store just a single copy of it (along with any values that differ). This *implicitly replicated value* (IRV) is often 0, but can take on any value in practice. Typically, sparse arrays are used when the number of explicitly stored values, nnz, is asymptotically less than the number of values represented by the array (*e.g.* $nnz = o(n^d)$, where n is the number of elements per dimension[2]). Figure 1 illustrates a sample sparse array.

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

[2] In practice each dimension can have a different number of

Challenges to Parallel Sparse Computation

Two of the primary challenges to providing reasonable facilities for sparse computation are clarity and performance. Performing sparse computation in parallel presents additional obstacles. In the following paragraphs, we consider several of these challenges in turn.

Conceptually, sparse arrays are no different than normal arrays; they represent a dense set of values, but use a nontraditional manner to do so. Most programming languages have no built-in support for sparse arrays, forcing users to implement their own sparse array representations by hand. This results in the obfuscation of conceptually simple operations, garbling the code's intent.

A prime example of this obfuscation can be found in the sparse matrix-vector multiplications of the NAS CG benchmark [2]. Conceptually, matrix-vector multiplication is simple, represented densely as follows:

```
do j = 1,numrows
  sum = 0.d0
  do k = 1,numcols
    sum = sum + a(j,k)*p(k)
  enddo
  w(j) = sum
enddo
```

However, since the matrix in NAS CG is sparse and Fortran doesn't provide support for sparse arrays, the benchmark's implementors represent it using a hand-coded *compressed sparse row* (CSR) storage format [23]. This format requires modifying the dense code above as follows:

```
do j = 1,numrows
  sum = 0.d0
  do k = rowstr(j),rowstr(j+1)-1
    sum = sum + a(k)*p(colidx(k))
  enddo
  w(j) = sum
enddo
```

The result is a code fragment that fails to reflect the logical operation being performed and obfuscates it to the point that it is difficult to recognize.

Performance represents another challenge to effective sparse computation. The use of the CSR format in the example above disguises the code's intent not only for human readers, but also for the compiler. In particular, the code uses *index arrays*—arrays which act as indices for other arrays. Index arrays obscure data dependences and are a notorious obstacle for compiler analysis and automatic parallelization [25, 7]. Although many compiler techniques have been developed to combat this problem, a breakdown in communication has occurred: the language has failed to communicate the programmer's intent to the compiler as clearly as it could. For example, CSR guarantees certain useful properties about the `rowstr` and `colidx` arrays used above, but the compiler has no way of detecting these properties by analyzing the code.

elements n_i, but for simplicity in this discussion we assume that they are all equal.

Performing sparse computations in parallel presents a number of additional challenges. Dividing the computation between a number of processors may not prove terribly difficult (the code above only requires changing the upper bound of the outer loop to `lastrow-firstrow+1`), but the code required to implement data transfer and synchronization between processors is lengthy, tedious, and complex. Even simple dense parallel array operations such as boundary value updates become significantly more complicated in the sparse context.

Writing portable parallel sparse codes that perform well *can* be done—the NAS CG benchmark is one such example. However, this is achieved only through great programmer care, and the result is often an extremely brittle piece of code. For example, if we were to extend NAS CG in a way that required efficient iteration over sparse matrix columns as well as rows, an immense amount of code would have to be reformulated and rewritten.

All of these issues—clarity, performance, parallel implementation, and flexibility—call out for a language-based solution. In particular, they motivate a language that provides a high-level, global means of specifying sparse computation using syntax which resembles the equivalent dense computation. This syntax should be clear to the programmer as well as the compiler and should result in an efficient parallel implementation. This paper describes work in which we strive to achieve these goals.

Our Work

ZPL is an array-based parallel programming language that has shown promise in its ability to support clean, concise code without sacrificing portability or performance [9]. The key to this success has been its use of the *region* as its basis for computation. Regions are language-level index sets used to declare arrays and specify parallel computation, and they provide numerous benefits to both the programmer and compiler [10, 8]. In this paper we extend the traditional region concept to support parallel sparse computation while preserving ZPL's benefits. The work in this paper is done within the context of ZPL's successor language, A-ZPL.

This paper constitutes the first published work that describes A-ZPL's support for sparse computation. As such, its contributions include:

- a detailed description of *sparse regions*
- code excerpts showing the use of sparse regions in the NAS CG and MG benchmarks
- a discussion of our compiler's implementation of sparse regions and arrays
- experimental results comparing the memory requirements and execution times of sparse A-ZPL implementations of CG and MG with the hand-coded NAS implementations
- a discussion of optimizations and future optimization opportunities

The rest of this paper is organized as follows. In the next section we give a brief overview of other techniques for sparse scientific computation. Section 3 gives a brief description of traditional ZPL regions and their benefits. Section 4 explains how sparse regions are used in A-ZPL, showing their application in the NAS CG and MG benchmarks. Section 5 describes the implementation of sparse regions and arrays in the A-ZPL compiler and explains our strategy for optimizing their memory requirements. Section 6 gives experimental results using the NAS CG and MG benchmarks. Finally in Section 7 we draw conclusions and sketch out our future work.

2. RELATED WORK

The past few decades have seen a considerable amount of research and development in the area of sparse computation. The vast majority of this work has involved the development of library support for sparse matrix computations [24, 27, 20, 19, 18]. Each of these systems exports a set of highly-tuned sparse matrix operations to the user via a custom library interface. These interfaces are subject to the standard tradeoffs: they can be small and extremely special-purpose or they can be general but "wide," either in terms of the number of routines exported or the number of parameters that their routines require [16]. In contrast, our approach is a language-based solution. It provides users with a small set of general operators that can be used to build sparse array codes (of which sparse matrices are but a subset). In doing so, we make the standard tradeoffs between libraries and languages: A-ZPL is unlikely to outperform a specific isolated library operation, yet its generality allows users to solve a larger set of problems, and the compiler can perform optimizations that span consecutive operations.

Although other parallel languages such as NESL [5] and HPF [17] have supported sparse matrix computations [6, 14], they have required users to implement their own sparsity structures by hand as in our introductory Fortran example. This results in the same obstacles to clarity and performance seen in that example. To aid with the performance problem, Ujaldon *et al.* have proposed extensions to HPF which provide language support for declaring sparse matrices using a variety of storage schemes [28]. This approach solves the problem of communicating to the compiler that a sparse matrix is being used. However, it still requires users to refer explicitly to the underlying sparse matrix representation rather than allowing them to use a traditional dense array syntax as in our work.

One approach that addresses this problem is a compiler technique by Bik and Wijshoff [4, 3] in which dense matrix programs are automatically transformed into an equivalent sparse program. The compiler automatically selects an appropriate sparse format depending on the placement of the non-zeroes. More recently, the Bernoulli compiler group has developed a technique which also allows users to specify sparse matrix operations using traditional dense syntax. In their approach, the compiler uses a form of *generic programming* to implement the code using a programmer-specified sparse matrix implementation [1, 21]. Our work shares the goals of both of these projects, but takes a different approach by using an array-based syntax and having the compiler automatically generate a sparse array representation optimized according to the operations applied to the sparse array rather than its structure. In doing so, we do not expect to compete with matrix representations that are highly specialized to a particular application, but do hope to support a broader class of sparse codes at a high level. Our approach also differs in that it supports higher-dimensional sparse arrays (rather than simply 2D matrices), and uses a unique optimizable sparse array representation.

One other language-based approach that deserves mention is Matlab [22], which supports seamless interactions between sparse and dense matrices [15]. The philosophy of our approach is very much like Matlab's, since we also seek to express sparse computation using a high-level array-based syntax. However, our application contexts are quite different in that Matlab is interpreted, sequential, and matrix-oriented whereas A-ZPL is compiled, parallel, and array-based.

3. INTRODUCTION TO REGIONS

This section provides a brief introduction to regions as well as other ZPL concepts used in this paper. For a more complete introduction, please refer to the literature [10, 26].

3.1 Regions

Regions are fundamental to ZPL's clean syntax, its performance model, and its efficient implementation. Regions are conceptually simple: they are user-defined index sets that can be named and manipulated using high-level operations. For example, the following lines of code create and name two regions:

```
region R  = [1..n,1..n];
       R2 = R by [2,2];
```

The first line declares an $n \times n$ region `R`. The second creates a new region `R2` that is the same size as the original, but strided by 2 in each dimension.

Regions have two uses in ZPL. The first is to declare arrays. For example, the following lines declare two arrays of integers over each of the previous regions:

```
var A , B : [R]  integer;
    A2, B2: [R2] integer;
```

The second use of regions is to provide indices for ZPL's array expressions. For example, consider the following code fragment:

```
[R]  A := 0;
[R2] begin
       A := Index1;
       A2 := 2*A;
     end;
[R] B2 := 0;   --illegal!!
```

The first statement assigns the value 0 to elements of `A`. Since it is preceded by the region scope `R`, all elements of `A` will be initialized to 0. The next line opens a new region scope `R2`, which specifies indices for the two statements that it encloses. Thus, the next assignment to `A` modifies only those values whose indices are specified by `R2`—those whose row and column indices are odd. These values are set using the implicit array `Index1`, whose values equal their indices in the first dimension. The next statement replaces each

	Simple Assignment	Partial reduction	Transpose
ZPL:	`[R] A := B;`	`[1..n,1] B := +<<[R] A;`	`[R] A := B#[Index2,Index1];`

F77:

Simple Assignment:

```
do i = 1,n
  do j = 1, n
    A(i,j) = B(i,j)
  enddo
enddo
```

Partial reduction:

```
do i = 1,n
  B(i,1) = 0
  do j=1,n
    B(i,1) = B(i,1) + A(i,j)
  enddo
enddo
```

Transpose:

```
do i = 1,n
  do j = 1,n
    A(i,j) = B(j,i)
  enddo
enddo
```

Figure 2: An example of the way in which regions emphasize different array access styles. In ZPL the three statements look quite distinct, alerting the programmer to the differences in their parallel implementations. By contrast, the statements appear much less distinct in Fortran.

value of A2 with twice its corresponding value in A. Note that the final assignment is illegal because it refers to B2 using indices not included in its defining region R2.

3.2 Array Operators

ZPL provides a number of array operators to describe array computations that are not simply elementwise in nature. These array operators modify the indices of the enclosing region scope for their array arguments. A simple example is the *reduction operator* which collapses one or more array dimensions. For example, the following code sums the values in each row of A and stores the result in the first column of array B:

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

The outer region scope [1..n,1] provides indices for the assignment to B in the usual manner. The reference to A is modified using the reduction operator +<< and a second region scope, [R]. In the context of the outer region scope, this specifies that all the values in A should be reduced to a single column using addition across the rows.

As a second example, the *remap operator* can be used to arbitrarily gather and scatter values of an array. For example, the following statement assigns the transpose of array A to array B:

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

The remap operator specifies that an array reference should be evaluated using the provided array arguments as index arrays. In scalar terms, the statement above is equivalent to:

```
forall i,j in R do
  B(i,j) = A(Index2(i,j),Index1(i,j))
             -- which equals A(j,i)
```

3.3 Benefits of Regions

Regions provide several benefits to the user. The first is syntactic. Regions have the effect of factoring the indices that describe a computation away from the array references and into an enclosing scope. This eliminates redundancy that occurs in traditional array indexing and slicing notations. Furthermore, the use of array operators serves to emphasize differences in the way that arrays are accessed. As an example, consider the ZPL statements in Figure 2 and their equivalent Fortran loop nests. In Fortran the operations look reasonably similar, though they are quite different in nature.

This last point may seem superficial, but it is crucial in the context of parallel computing. When arrays are distributed across the local memories of a processor set, their reference patterns have a critical impact on performance. ZPL's use of different operators to signify different array access patterns allows users to evaluate their programs' parallel implementation simply by looking at the source code [8]. For example, ZPL programmers know that the first statement in Figure 2 will execute completely in parallel. Furthermore, they know that the assignment involving the reduction operator requires vector reductions across the processor rows and that the assignment which uses the remap operator could potentially result in an all-to-all communication. In contrast, the syntactic similarity of these operations in Fortran disguises the fact that their parallel implementations might be vastly different.

This syntactic similarity causes potential confusion not only for the programmer, but also for the compiler since it must decipher the array references and loop bounds to determine the user's intent before generating optimized parallel code. In contrast, regions enable ZPL compilers to understand exactly what the programmer has requested, allowing compiler writers to concentrate on more interesting optimizations than simply locating and classifying a code's parallelism. The final benefit of the region is therefore its role in generating efficient parallel code.

3.4 Flood Dimensions

Interactions between parallel arrays of different dimensions (*e.g.* vectors and matrices) are expensive when the arrays' distributions are not properly aligned. To circumvent this problem, ZPL programmers typically use higher-dimensional arrays which are *flooded* in one or more dimensions to indicate their distribution in the higher-dimensional space. For example, the following code declares flood regions that are aligned with the rows and columns of the full 2D region R:

```
region R   = [1..n,1..n];
       Row = [ *  ,1..n];
       Col = [1..n, *  ];
```

```
region Diag = Tri where (Index1 = Index2);    -- limit Tri to its main diagonal
       Rs   = R   where Pattern;              -- Pattern is a boolean array
       Rs2  = R   where foo(Index1,Index2);   -- foo(i,j) returns true/false
       Rs3  = R   where read(infile);         -- read pattern from a file
       Rs4  = R   where ?;                    -- dynamic sparsity pattern that
                                              -- will be computed in code body
```

Figure 3: Various styles of sparse region declarations. Note that the base region may either be sparse or dense and that the boolean expression may be an array, a promoted scalar function, file I/O, or left unspecified until runtime.

These flood dimensions are conformable to any index in that dimension, allowing them to interact naturally with traditional arrays. Note that these declarations differ from a "flat" dimension like `[1,1..n]` since flat dimensions are not conformable with arbitrary indices and therefore could not be read within the context of region `R`. Flood arrays are implemented by replicating their defining values across the appropriate dimensions of the logical processor grid. For more details about flood dimensions and their implementation, refer to the literature [12].

3.5 Dense Matrix-Vector Multiply

Let us return now to our introductory example of dense matrix-vector multiplication. Using the flood regions of the previous section, a dense matrix-vector multiplication can be written in ZPL as follows:

```
var A:    [R] double;
    P, Q: [Row] double;
    W:    [Col] double;

[Col] W := +<<[R] (A * P);
[Row] Q := W#[Index2,Index1];
```

The reduction operator performs the actual matrix-vector multiplication, storing the result in the column array `W`. We then use a remap operator to transpose the result back into a row for use in subsequent operations.

ZPL's performance model tells the programmer that running this code on a 2D grid of processors will require a reduction across processor rows to perform the multiplication and all-to-all style communication to implement the vector transpose. It also indicates that when the number of processor columns is set to 1, the reduction will be completely local, but all `Row` computations will be performed redundantly on all processors. Dually, if the number of processor rows is set to 1, no redundant vector computation will take place, but the reductions will involve vectors of size n. ZPL programmers can weigh these tradeoffs or simply try different processor grid topologies at runtime. In the following section, we will see how the sparse version of this code would be written in A-ZPL.

4. SPARSE REGIONS

4.1 Motivation

One of the chief limitations to ZPL's regions is that they must be rectangular and regular (though possibly strided). ZPL provides the ability to select a subset of a region's indices using a boolean *mask*, but this concept costs n^d in both storage and iteration, regardless of the mask's density. For many applications, such as red-black SOR, this may be completely reasonable. However, for others in which the number of indices is $o(n^d)$, masks require excessive memory and execution time.

The goal of this work is to extend regions to efficiently support general sparse index sets in A-ZPL without sacrificing the benefits of traditional ZPL regions. In particular, the syntax should remain crisp, the parallel overheads should remain apparent to the user, and the time and space requirements should be proportional to the number of indices represented. The rest of this section describes the syntax used to represent sparse regions and shows its use in some sample applications.

4.2 Sparse Region Declarations

A-ZPL programmers declare sparse regions by modifying a traditional region with a boolean expression that describes the sparsity pattern. For example:

```
region Tri = [1..n,1..n] where
                (abs(Index1-Index2) < 2);
```

This declaration bounds `Rs` to an $n \times n$ index set, but restricts it to those locations whose row and column indices differ by no more than 1. Thus, this declaration creates a tridiagonal region.

Sparsity patterns need not be statically defined as in this first example. Moreover, the base region itself may be sparse. Figure 3 shows a variety of different sparse region declarations. Each declaration is evaluated at runtime by iterating over the base region and using a sparse representation to store those indices where the defining expression evaluates to "true."

A-ZPL programmers declare sparse arrays in the traditional manner, using sparse regions to define their size:

```
var As, Bs: [Rs] integer;
```

This declaration causes each processor to allocate a dense vector of memory for `As` and `Bs` whose size is equal to the number of indices in `Rs` that it owns, plus one additional value to represent the IRV. See Figure 6a for an illustration. Note that A-ZPL associates sparse representations with regions at runtime rather than with arrays. This allows the space and time required to store and iterate over sparse representations to be amortized across multiple arrays that share the same sparsity pattern.

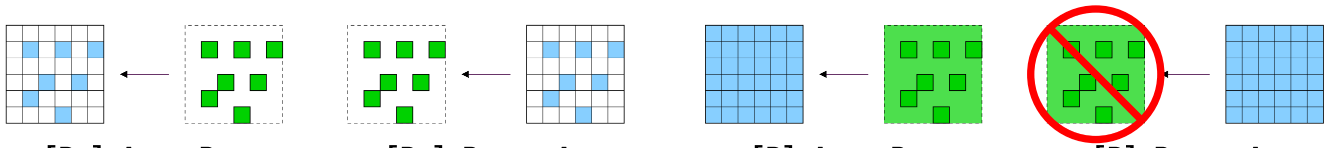

Figure 4: A variety of sparse/dense assignments. Assume that R and A are dense, and that Rs and Bs are sparse. Note that sparse reads and writes of both arrays are legal, referring only to the elements in the sparsity pattern. Dense reads of sparse arrays are also legal—the IRV is used for values that are not in the sparsity pattern. However, a dense write to a sparse array is not legal due to the fact that a unique memory location is not allocated for each index. These examples extend in the obvious way to statements which refer to multiple sparsity patterns.

4.3 Using Sparse Regions

Sparse regions concisely support a rich variety of semantics. As a brief example, consider the assignments in Figure 4. In the first two assignments, the controlling region is sparse. The effect is that references to dense array `A` only read or write those values indicated by `Rs`. When `Bs` is referenced, all of its values are accessed since its sparsity pattern is defined by `Rs`.

The third assignment illustrates a dense read of a sparse array. In this statement, all values of `A` will be written even though `Bs` is sparse. This is because `Bs` logically represents a full $n \times n$ array of values. Thus, when reading an index in `R` $-$ `Rs`, the IRV of `Bs` will be referenced, causing all of `A`'s values to be assigned. Note that this statement will take $\mathrm{O}(n^2)$ time as compared to the first statement which, though similar, requires only $O(|\mathtt{Rs}|)$ time.

The last assignment is illegal for reasons similar to the one in Section 3.1: it attempts to store values in `As` for which no storage has been allocated.

Although these examples are quite simple, they illustrate the basic rules of reading and writing sparse and dense arrays in the context of a sparse or dense region. These same principles naturally extend to statements which mix sparsity patterns. Furthermore, the traditional ZPL array operators extend to sparse regions and arrays in the natural manner. To understand the elegance of this approach, one only needs to consider how a detailed sparse array operation like the following would look in a traditional language:

```
[1..n,1]  A  :=  As#[1,Index1] +
                 +<<[Rs] (Bs * Di);
                -- Di is declared over Diag
```

4.4 Benchmark Examples

At this point, we turn our attention to two real-world examples that will form the basis of our experiments: the NAS CG and MG benchmarks [2].

NAS CG

Consider the changes required to convert the dense ZPL matrix-vector multiplication from Section 3.5 to a sparse format. The changes to the declarations are simple: a sparse region needs to be declared to specify matrix `A`'s sparsity pattern:

```
region RS = R where ...;
var A:[RS] double;
```

The `Row` and `Col` regions could also be made sparse if it seemed worthwhile. However, the properties of matrix `A` in NAS CG are such that all vectors will be dense, and therefore we choose to retain the dense format.

No change is required to the computation itself since `A` can be read within the context of `R` in spite of the fact that it is sparse. However, the savvy programmer will realize that performing `A * P` within the context of `R` will require $O(n^2)$ multiplication whereas the sparsity pattern of `A` is $o(n^2)$. This represents a lot of wasted computation, so we change the region controlling the multiplication to `Rs`:

```
[Col] W := +<<[RS] (A * P);
[Row] Q := W#[Index2,Index1];
```

Note that the logical expression of the matrix-vector multiplication remains the same as in the dense case. This is in sharp contrast to the Fortran version in the introduction. The A-ZPL user continues using traditional ZPL operations, allowing the compiler to manage all of the details required to change this parallel computation from dense to sparse.

NAS MG

The NAS MG benchmark solves a discrete Poisson problem using the *multigrid method*. By most standards, it is considered a dense computation, using a series of 27-point stencils on a *hierarchical array* to achieve its result. However, it turns out that the input to the NAS MG benchmark is extremely sparse. It consists of ten positive and ten negative charges stored at the finest discretization of the problem space. For the class C version of the benchmark, this implies that only 20 of the 512^3 elements in the input array are nonzero—15 millionths of a percent! This application demonstrates the use of sparse arrays to represent structural sparsity rather than a sparse matrix.

If this input array was used only at the outset of the program, the drawbacks to using a dense representation would be minimal since its storage could simply be deallocated or reused to avoid wasting memory. However, the MG benchmark actually computes two residual stencils against the input array every iteration, which require walking across its entire memory footprint to find the 20 values that are of interest. It therefore seems that using a sparse format for the input would be worthwhile. NAS MG programmers have traditionally not done so, presumably due to the effort that changing the dense array into a sparse one would require. In A-ZPL the change is minimal and therefore worthwhile.

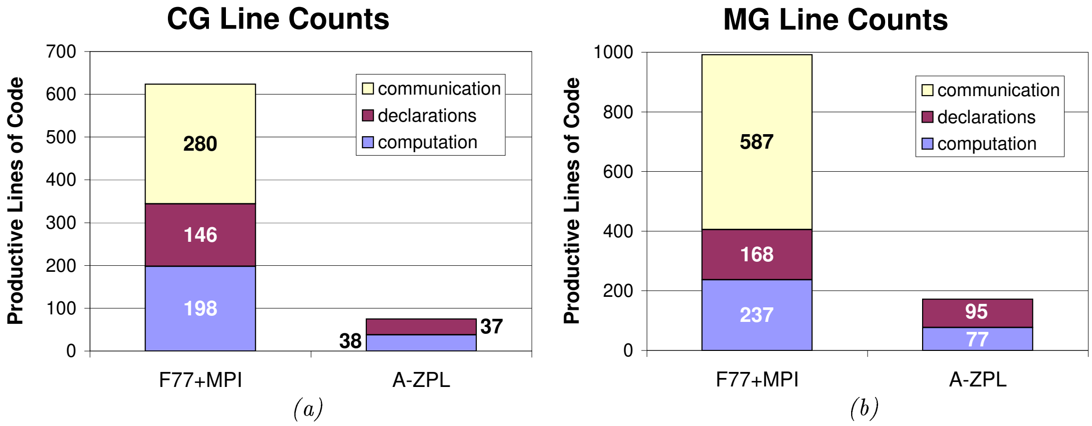

Figure 5: These graphs indicate the number of productive lines of code required to implement the two benchmarks in each language. Each line is classified as being communication, declarations, or computation. Initialization, I/O, and timings are omitted from the analysis. The F77+MPI codes are the original NAS benchmarks. The A-ZPL codes are our sparse versions of the benchmarks.

Here is the code from our dense ZPL implementation [9] that refers to the input array `V`:

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

var V:[RBase] double;

procedure resid(R,V,U:[,,] double);
begin
  R := V + ... -- 27-point stencil on U;
end;
```

The `resid()` procedure is used not only for the input array, but also for all levels of the hierarchical arrays. Therefore, the regions defining the computational range and the sizes of `R`, `V`, and `U` are all inherited from the callsite.

To change the input array into a sparse format, one need simply modify the declarations as follows:

```
region RS = RBase where ...;

var V:[RS] double;
```

The A-ZPL compiler must then automatically create two versions of `resid()`: the traditional dense version for use within the hierarchy, and a second for use when parameter `V` is sparse.

As in NAS CG, the savvy programmer would realize that this code could be improved. In particular, `V` is referenced over the full n^3 problem space in spite of the fact that only 20 of its values are nonzero when it is sparse. Therefore, the programmer may choose to write a specialized version of the procedure that eliminates this useless computation:

```
procedure residSps(R,V,U:[,,] double);
begin
  R := ... -- 27-point stencil of U;
  [RS] R += V;
end;
```

This modification requires the programmer to create two copies of `resid()`, but the change is fairly trivial as compared to implementing the sparsity by hand. In future work, we intend to have the compiler recognize such opportunities for optimization and perform them automatically.

4.5 Evaluation of Clarity

Figure 5 indicates the number of lines required to implement CG and MG in A-ZPL as compared to the NAS F77+MPI implementations. More than half of the F77+MPI lines are devoted to handling parallel communication and data distribution issues by hand. This code is tedious and error-prone. In contrast, the A-ZPL compiler handles these details automatically, resulting in cleaner, clearer implementations of the benchmarks. A-ZPL's region-based syntax also contributes greatly to its succinct representation of the benchmark. While we do not have permission to reproduce long excerpts from the NAS benchmark, we encourage readers to download the codes and confirm for themselves that these line counts reflect the relative readability of the benchmarks.

It is reasonable to wonder whether rewriting the NAS benchmarks in Fortran 90 would improve their clarity. We believe that while it would result in some improvement to the code's readability, the details of communication, which form the bulk of their complexity, would remain largely unchanged. Furthermore, there is some question as to whether the slice notation could more cleanly express the array indexing used by NAS CG or the scalar stencil optimization applied in NAS MG [9] without sacrificing performance.

5. IMPLEMENTATION

In this section we describe our implementation of sparse regions and arrays in A-ZPL. We begin by giving a broad overview of the implementation and then focus on our sparse representation.

5.1 Overview

As mentioned in Section 4.2, our implementation strategy is to associate sparsity patterns with regions rather than arrays. This gives us the ability to amortize overheads related to storing and iterating over sparse representations when multiple arrays share the same pattern. Note that neither NAS CG nor NAS MG benefit from this design since each

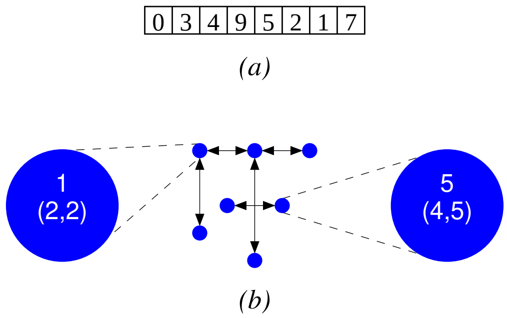

Figure 6: The sparse array of Figure 1 shown as it would be stored in A-ZPL. *(a)* The dense vector of values for this array. The initial element is the IRV; all other elements correspond to an index in the sparse region. *(b)* The lattice for the region that describes the array's sparsity pattern. A few of the nodes are enlarged to indicate their sparse ID and logical indices. Note that other arrays with this sparsity pattern would share the same lattice.

uses just a single sparse array. However, in previous work we have demonstrated its effectiveness [11], and we remain confident that larger sparse applications will benefit from this approach.

Our sparse region data structure is a traditional ZPL region with additional fields tacked on to represent the sparsity pattern. The traditional region information is used for operations related to the sparse region's bounding box, such as determining which processors own the region. The sparsity pattern is represented using a sparse array representation, described in the next section. Each element in the sparse representation has a unique ID from 1 to nnz. This ID is used to access values in any sparse arrays that are declared using the region.

Our sparse array data structure uses a traditional 1D ZPL array to represent a vector of values. Its size is equal to the number of sparse indices owned by the processor, plus one to store the IRV. The IRV is stored at position 0. All other elements are accessed using unique IDs from the region nodes as indices into the vector (Figure 6).

Implementing sparse A-ZPL statements is simply a matter of supporting iteration over a region's sparsity structure within the context of a dense or sparse region. A loop nest is generated for each statement's controlling region, with iterators established for each sparse and dense array referenced by the statement. If multiple arrays have the same defining region, they share a single set of iterators.

ZPL's runtime libraries required modification to work with sparse regions. These changes could either be done by specializing each call to work with sparse regions, or by supporting generalized iteration over regions via function pointers. Presently, we are taking the first approach for simplicity, though we are moving toward a hybrid approach to optimize the tradeoffs between code replication and runtime overheads.

5.2 Our Sparse Representation

To understand our choice of sparse representation, it is useful to understand the operations that ZPL's regions and arrays must support. This list summarizes the most time-crucial operations and the situations that require them:

region operations:

- row-major iteration
  (the common case for most region references)
- iteration in arbitrary directions, dimensions
  (certain uses of the `@`, `'@`, `@^` operators)
- operations on region slices
  (`in`, `of` region operators; dynamic region slices)

array operations:

- ordered array access
  (general iteration)
- random array access
  (`#` operator)

The generality of our design requirements presents an obstacle to using many of the standard sparse array representations since they are typically optimized for a particular access pattern. For example, the need for efficient iteration in arbitrary dimensions eliminates the possibility of using formats that support a particular iteration order such as CSR. Our need for arbitrary iteration combined with the fast random access required by slicing and random array accesses forces us to support a very general-purpose sparse representation. Our strategy is therefore to design a format which can automatically be optimized based on each program's specific requirements.

In our sparse representation, every index is logically represented by an associated node that has: (1) a unique ID used to access sparse arrays, (2) the logical index that it represents, and (3) pointers to the next and previous nodes in each dimension (Figure 6b). This *lattice* of nodes is a generalization of a multilist structure [29] and supports the ability to iterate quickly from any node to its neighbors in any dimension. The space required by this lattice is $O(nnz)$.

Since the lattice may not be strongly connected, some sort of *sparse directory* structure is required to support iteration over all of the nodes. This is achieved by recursively adding dummy nodes to the head and tail of each list in the multilist (Figure 7a). For the dimensions in which these nodes serve as dummies, their indices are set to be minimal or maximal integers to provide a simple termination condition during iteration. The sparse directory supports the ability to iterate over the entire region in $O(nnz)$ time. A very loose bound on the required space is $(3^d - 1) \cdot nnz$ which is $O(nnz)$ if d is considered a constant.

To provide the random access required by region slicing and random array accesses, a *dense directory* structure is added for each dimension which provides random access to its lists (Figure 7b). This structure is represented using a hash table or array, depending on the number of lists to be stored. Since each list contains $o(n)$ elements, this provides a means for accessing any index within the region in $O(1) + o(n)$ time—not quite constant, but reasonably close. An upper bound

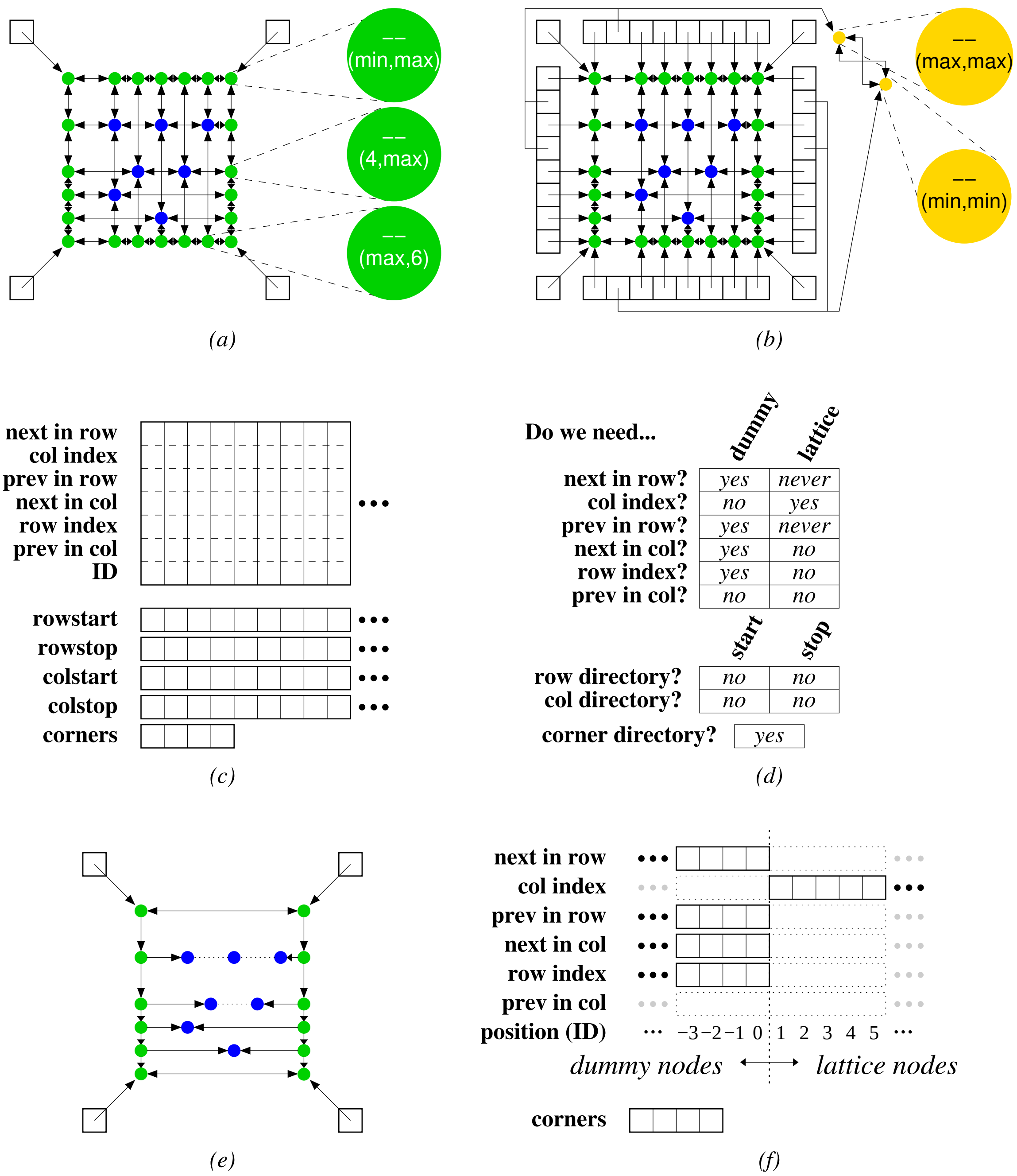

Figure 7: *(a)* The lattice of Figure 6b, extended to include a sparse directory of dummy header nodes. Details for a few nodes are shown to indicate their sentinel values. An array of pointers to corner nodes is also stored. *(b)* The same data structure, extended to include a dense directory structure. Since most rows and columns are non-empty, the directories for each dimension are arrays rather than hash tables. Empty rows and columns refer to a shared dummy list to eliminate special cases. *(c)* A naive implementation of the data structure in part *b* which uses an array of records to store the node data and additional arrays for the dense directory and corners. *(d)* A sample output from our compiler, indicating which fields are needed for each node type, as well as whether or not the dense and corner directories are required. This output corresponds to programs that only require row-major iteration orders, such as NAS CG and MG. Note that next and previous pointers for a row are never required for lattice nodes because they are stored in row-major order. *(e)* The simplified data structure that meets the requirements of the compiler analysis in part *d*. *(f)* The layout of this data structure in memory. Each node field is implemented as a separate array, allocated only for the nodes that require that field. Nodes with positive IDs are lattice nodes. A negative or zero ID indicates a dummy node. Note that an explicit ID no longer needs to be stored for each node.

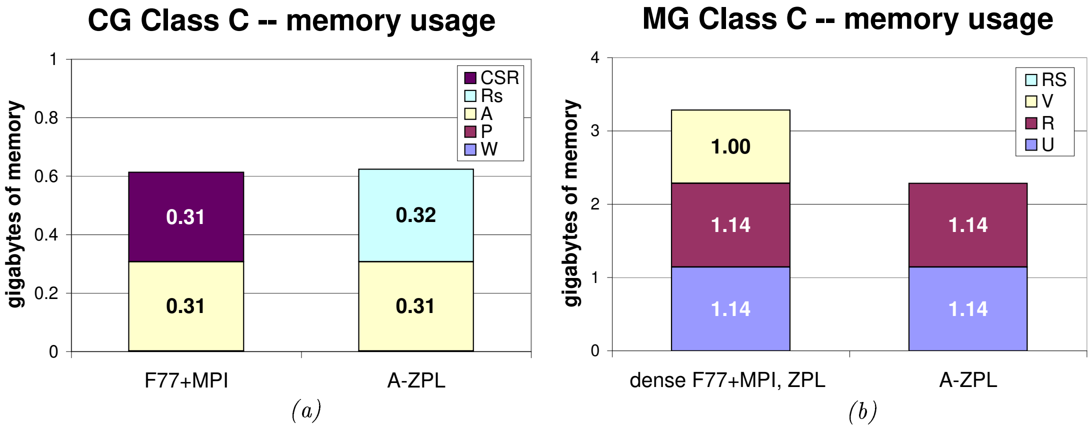

Figure 8: These graphs indicate the amount of memory used by the major data structures for class C of each benchmark on a single processor. *(a)* For the CG benchmark, `A` stores the sparse matrix values. `CSR` is the memory required for the CSR format used by F77+MPI. `RS` is the memory required for the sparse region in A-ZPL. Note that the memory required for vectors `P` and `W` is negligible. *(b)* In the MG benchmark, `U` and `R` are the two hierarchical arrays and `V` is the input array. Note that the storage required for `V` and its sparse region `RS` is negligible for our sparse implementation.

on the space required by the dense directory structure is $2d \cdot \min(nnz, n^{d-1})$ or $O(nnz)$ if d is considered a constant.

The sum of these parts is a very flexible data structure that supports all of our required operations using space proportional to the number of indices represented by a region. While the generality of this structure has a certain amount of overhead (illustrated in Figure 7c), a clever implementation of these components admits an implementation whose memory requirements can be optimized to rival that of CSR.

5.3 Optimizing our Sparse Representation

Our actual implementation differs from this logical description in several respects. Most importantly, we represent the lattice nodes using a collection of integer vectors rather than a vector of records. This decision causes us to represent pointers in the lattice as vector indices rather than true pointers. We order the nodes so that all the dummy nodes are at the front of the vector, and all of the lattice nodes are at the end in row-major order. The vector is shifted so that the lattice nodes start at logical index 1 to eliminate the need for an explicitly-stored node ID.

This approach has several benefits: First, it improves the spatial locality of lattice traversals, since the "pointers" being chased are now contiguous in memory rather than strided by the node size. Second, it allows us to eliminate the "next-in-row pointers" for the lattice nodes due to the fact that they are implicit in the memory layout. Third, it allows us to eliminate the allocation of fields that are not required, or to allocate them only for the dummy or header nodes, as we will now describe.

As our compiler generates loop nests and calls to the runtime libraries, it keeps track of whether the pointers and indices in each sparse node are required by the generated code. The same is done for the dense and corner directories. For each region, the compiler emits a summary of this information as shown in Figure 7d. This summary is used by the region's

Table 1: Technical details for the Cray T3E

Location	Arctic Region Supercomp. Center
Processors	256
Speed	450 MHz
Memory per proc.	0.143 GB
Memory model	Distributed Global Address Space

runtime constructor to allocate only the structures that the program requires. For example, in NAS CG and MG the compiler detects that the sparse regions are only traversed in row-major order. Therefore, the setup code will eliminate the dense directory, much of the sparse directory, and most of the fields for the lattice and dummy nodes (Figure 7e, f). The result is a representation that is similar to CSR.

6. EXPERIMENTAL RESULTS

The graphs in Figure 8 compare the amount of memory used by the F77+MPI and A-ZPL implementations of NAS CG and MG. For CG, our compiler optimized our naive sparse representation as described in the previous section, causing the memory usage to rival that of the hand-coded CSR format. For MG, the memory required for the optimized sparse region and input array is negligible as compared to the dense hierarchical arrays, reducing the overall memory footprint by 30%.

To evaluate the performance obtainable by our compiler, we ran the A-ZPL implementations of CG and MG on a Cray T3E and compared the results to the original F77+MPI implementations. Technical details for the T3E are given in Table 1. Speedup curves for the class C versions of each benchmark are shown in Figure 9.

For the CG benchmark, the A-ZPL implementation outperforms the F77+MPI version for most of the processor sets ($p \leq 64$), but then loses ground as the number of processors increases. Its speed advantage on smaller numbers of pro-

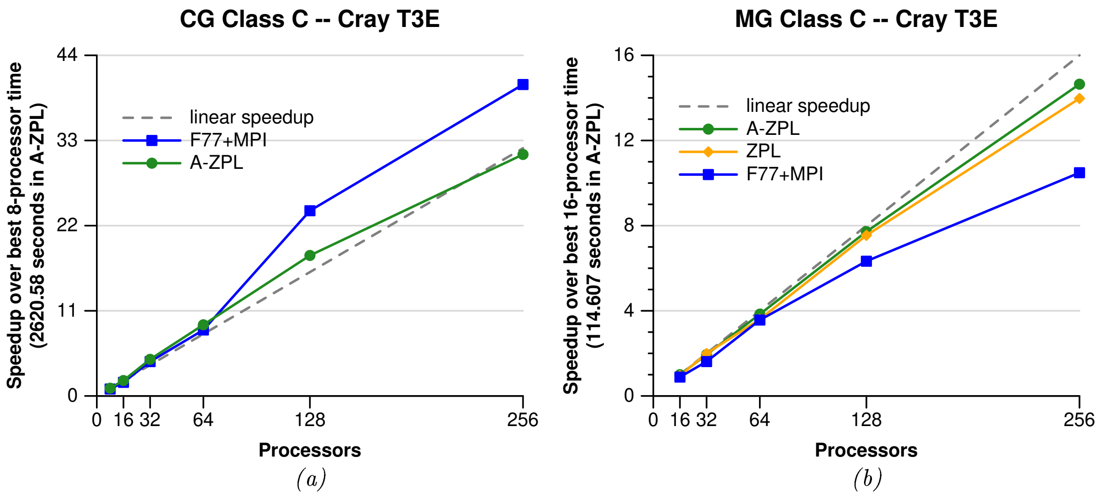

Figure 9: Speedup curves for class C of the NAS CG and MG benchmarks on the Cray T3E. Note that there is not enough memory to run these problem sizes on small processor sets. Thus, we compute speedups for each graph using the fastest execution time on the smallest number of processors (indicated in the y-axis label).

cessors is due to two factors: First, the A-ZPL compiler can generate crisp loop nests to traverse the sparse region and array due to its knowledge of their relation to one another. Second, on the T3E, the A-ZPL compiler implements communication using the SHMEM library, which tends to have lower overhead than MPI since its communication paradigm matches that of the architecture.

A-ZPL loses ground on the larger processor sets due to the overheads associated with the current implementation of its remap operator. This operator has never been tuned to take advantage of readily-available static information such as the flood dimensions of its arguments or its use with the compiler-defined `Index`i arrays. These optimizations are currently under development, and we expect that their completion will improve the performance of CG on the larger processor sets. For the time being, we are pleased to note that A-ZPL's speedup is near-linear across all runs.

The A-ZPL implementation of NAS MG does not use the remap operator, and scales quite nicely for all processor set sizes. We include speedup results for the traditional dense ZPL implementation as a basis for comparison. The ZPL implementation outperforms the F77+MPI version for reasons outlined in previous work [9]. We find that converting the input array to a sparse format in A-ZPL further reduces the execution time, though not by much. This is due to the T3E's hardware support for aggressive data prefetching in the presence of regular memory access patterns. The references to the input array in the dense implementation benefit greatly from this prefetching, reducing the advantages of a sparse representation. For this platform, the greater benefit to users is in the 30% reduction in memory, allowing them to run larger problem sizes on smaller numbers of processors.

7. CONCLUSIONS

In this work, we have described how sparse regions support language-based expression of sparse computation in a variety of forms—sparse matrices, structural sparsity, and sparse iteration over dense arrays. We have succeeded in our goal of supporting expressive sparse parallel computation by maintaining a dense array syntax and exposing parallel overheads in the program's text. Our approach results in sparse matrix codes that are a fraction the size of their hand-coded counterparts and, more importantly, are uncluttered by parallel programming details such as communication and data distribution.

We have described the implementation of our sparse representation and its support for general array operations such as arbitrary iteration and slicing. We have also demonstrated that our compiler can automatically specialize the implementation to reduce its memory footprint while fulfilling a program's requirements. In our implementation of NAS CG, this optimization results in memory use that is similar to that of a hand-coded CSR format. For NAS MG, the sparse implementation significantly reduces memory requirements as compared to the dense implementations.

Our experiments show that the scalar performance of sparse A-ZPL codes rivals or exceeds that of their hand-coded MPI counterparts. Scalability of CG's sparse matrix-vector multiplication in A-ZPL is currently limited by communication overhead on larger processor sets. In spite of this, the performance is near-linear and outperforms the hand-coded version for 1–64 processors. With NAS MG, A-ZPL demonstrates good scalability and performance improvements by using a sparse input array rather than a dense one.

The net result of our work is a language that supports sparse computation using a clean, high-level syntax which results in good parallel performance. While this paper represents a good first step toward supporting language-level sparse parallel computation, there is still much work to do. Some of our next steps will include optimizing the remap operator and broadening the parallel platforms used in this experiment. In the future we also hope to expand our set of sparse benchmarks to include others that utilize a greater number of array operations and sparsity patterns. One algorithm of particular interest will be the Fast Multipole Method [13], which will require combining the 3D sparsity of this paper with the hierarchical arrays of our previous work [9] to implement sparse hierarchical computation.

8. ACKNOWLEDGMENTS

The authors would like to thank E Christopher Lewis, Sung-Eun Choi, W. Derrick Weathersby, and Ruth Anderson for their contributions during the early stages of this work. Thanks also to the Arctic Region Supercomputing Center for the use of their Cray T3E in our experiments.

9. REFERENCES

[1] Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghill. A framework for sparse matrix code synthesis from high-level specifications. In *Proceedings of the 2000 ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC2000)*, November 2000.

[2] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS–95–020, NASA Ames Research Center, Moffet Field, CA, December 1995.

[3] A. J. C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, Leiden University, 1996.

[4] A. J. C. Bik and H. A. G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the International Conference on Supercomputing*, pages 416–424, 1993.

[5] Guy E. Blelloch. NESL: A nested data-parallel language (3.1). Technical Report CMU–CS–95–170, Carnegie Mellon School of Computer Science, September 1995.

[6] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[7] T. Brandes, F. Bregier, M. C. Counilh, and J. Roman. Contribution to better handling of irregular problems in HPF2. In *Proceedings of Europar '98*, number 1470 in LNCS. Springer-Verlag, 1998.

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

[9] Bradford L. Chamberlain, Steven J. Deitz, and Lawrence Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Proceedings of the 2000 ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC2000)*, November 2000.

[10] Bradford L. Chamberlain, E Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: An abstraction for expressing array computation. In *ACM/SIGAPL International Conference on Array Programming Languages*, pages 41–49, August 1999.

[11] Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. A region-based approach for sparse parallel computing. Technical Report UW-CSE-98-11-01, University of Washington, November 1998.

[12] Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. Problem space promotion and its evaluation as a technique for efficient parallel computation. In *Proceedings of the 13th ACM International Conference on Supercomputing*, pages 311–318, June 1999.

[13] M. A. Epton and B. Dembart. Multipole translation theory for the three-dimensional laplace and helmholtz equations. *SIAM Journal on Scientific Computing*, 16(4):865–97, July 1995.

[14] Michael Frumkin, Haoqiang Jin, and Jerry Yan. Implementation of NAS parallel benchmarks in high performance fortran. Technical Report NAS-98-009, NASA Ames Research Center, Moffet Field, CA, September 1998.

[15] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIMAX*, 13(1):333–356, January 1992.

[16] Samuel Z. Guyer and Calvin Lin. Broadway: A software architecture for scientific computing. In *IFIPS Working Group 2.5: Working Conference on Software Architectures for Scientific Computing Applications*, October 2000.

[17] High Performance Fortran Forum. *High Performance Fortran Specification Version 2.0*, January 1997.

[18] Mark T. Jones and Paul E. Plassmann. *BlockSolve95 Users Manual: Scalable Library Software for the Parallel Solution of Sparse Linear Systems.* Argonne National Laboratory, December 1995. (revised June 1997).

[19] A. R. Krommer. Parallel sparse matrix computations in the industrial strength PINEAPL library. In *Applied Parallel Computing: Proceedings of PARA'98*, volume 1541 of *LNCS*, pages 281–285. Springer-Verlag, 1998.

[20] K. J. Maschhoff and D. C. Sorensen. P_ARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures. In *Applied Parallel Computing: Proceedings of PARA'96*, pages 478–486. Springer-Verlag, 1996.

[21] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *Proceedings of the International Conference on Suprecomputing*, May 2000.

[22] Mathworks. *MATLAB User's Guide*, 1993.

[23] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[24] Y. Saad and A. Malevsky. PSPARSLIB: A portable library of distributed memory sparse iterative solvers. In V. E. Malyshkin et al., editor, *Proceedings of Parallel Computing Technologies (PaCT-95), 3rd international conference*, LNCS. Springer-Verlag, Sept. 1995.

[25] Z. Shen, Z. Li, and P. C. Yew. An empirical study on array subscripts and data dependences. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, 1989.

[26] Lawrence Snyder. *The ZPL Programmer's Guide*. MIT Press, 1999.

[27] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid. *Official Aztec User's Guide: Version 2.1*. Sandia National Laboratories, December 1999.

[28] M. Ujaldon, E. L. Zapata, B. M. Chapman, and H. Zima. Vienna fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(10):1068–1083, October 1997.

[29] Mark Allen Weiss. *Data Structures & Algorithm Analysis in C++*. Addison-Wesley, second edition edition, 1999.

APPENDIX

A. COMPILER SPECIFICATIONS

The following table summarizes the compilation process used in our experiments. The compiler, version number, and command-line arguments used are given for each language. In addition, the communication mechanism used at runtime is noted.

Table 2: The compilers and options used for each language in our experiments.

Cray T3E compilers

Language	*Compiler*	*Version*	*Args.*	*Comm.*
F77+MPI	Cray f90	3.3.0.0	–O3	MPI (Cray)
ZPL/	UW zc	1.17a		SHMEM
A-ZPL	Cray cc	6.4.0.0	–O3	

B. EXPERIMENTAL TIMINGS

Table 3 contains the best observed times for each language and benchmark. These were used to compute the speedup graphs of Section 6.

Table 3: The raw timings used to compute speedups in our experiments. All times are in seconds.

Cray T3E — Class C — NAS CG

processors	8	16	32	64	128	256
F77+MPI	2922.03	1489.15	590.65	307.45	109.53	65.14
A-ZPL	2620.58	1314.66	553.95	285.05	144.40	84.01

Cray T3E — Class C — NAS MG

processors	16	32	64	128	256
F77+MPI	129.08	70.95	32.12	18.11	10.93
ZPL	120.09	57.46	31.80	15.19	8.20
A-ZPL	114.61	58.99	29.81	14.85	7.83