A Region-based Approach for Sparse Parallel Computing*

Bradford L. Chamberlain   E Christopher Lewis   Lawrence Snyder

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350

Abstract

This paper introduces a technique for parallel sparse computation by extending the array-language concept of *regions*—regular programmer-specified index sets used for specifying array computations. We introduce the notion of *sparse regions* which can represent an arbitrary set of indices. Sparse regions inherit the benefits of regular regions, including conciseness, a direct encapsulation of parallelism, and support for language performance models that highlight parallel overheads. We show that region-based array languages can benefit from the use of sparse regions, both in terms of the semantic richness available to the programmer and the execution times of the resulting program. We also demonstrate that regions result in efficient implementations as compared to array-based approachs, due to their role in amortizing sparse overheads and enabling optimizations.

1 Introduction

Sparse computations form an important class of applications for parallel computing, yet one that has been largely ignored in the parallel programming language community. Few languages exist that contain built-in support for sparsity. Instead, most require programmers to explicitly build and manipulate data structures for representing sparse arrays and matrices. Although considerable effort has been devoted to developing runtime libraries for sparse computations as well as efficient data structures for representing sparse matrices, direct support for sparsity in a programming language can have many benefits. Chief among these is the opportunity to cleanly and unambiguously alert the compiler to the presence of sparsity in order to ensure its efficient implementation and highlight opportunities for parallelism. In addition, language support for sparse computations can result in cleaner programs, increasing a language's readability and the ease with which programmers can express their computations.

For the purposes of this discussion, *sparsity* is characterized by an arbitrary subset of indices from a regular index space. Typically the subset represents only a small fraction of the space. For example, an array or matrix is considered to be sparse when it contains a large number of identical values (typically "0") such that it becomes worthwhile to represent only the values that differ. In this paper, we refer to these as

*This research was supported in part by DARPA Grant E30602-97-1-0152, NSF Grant CCR-9710284 and the Intel Corporation.

the *represented values*. Due to the arbitrary location of the represented values' indices, the overhead of any sparse representation is tied to the representation of, and iteration over, this set of indices. The magnitude of this overhead determines whether a sparse representation will result in savings over a dense one. In any reasonable representation, if the number of represented elements is asymptotically smaller than the overall size of the index space (*e.g.*, $3n$ elements in an n^2 array), the sparse representation should result in a savings.

In this paper, we discuss a *region-based* approach to expressing sparse computations. A *region* is an array language construct used to represent a regular, rectangular set of array indices. In regular array-based computations, the use of regions has resulted in clean, concise code that is readily parallelizable [4]. Furthermore, parallel region-based languages can be designed with clear performance models that allow programmers to trivially detect and quantify the concurrency and parallel overheads associated with their code [2]. In this paper, we relax the regularity characteristic of traditional regions to create *sparse regions* that can represent arbitrary index sets. We find that the use of regions to express sparsity results in the following benefits:

- The advantages obtained in regular region-based computations (*e.g.*, conciseness, concurrency, a clear performance model) [4] are preserved.
- Sparse computations with a wide variety of dynamic and structural characteristics can be expressed clearly and efficiently.
- Overheads associated with representing and traversing sparse arrays can be amortized across expressions that share the same sparsity pattern.
- Sparsity is presented to the compiler at a high level, leveraging the support of traditional parallel array language optimizations and creating opportunities for new optimizations resulting from the presence of sparsity.

Programming language support for sparsity has perhaps been best achieved in the matrix language MATLAB [10]. The extension of its original dense formulation to include sparsity is a model example of language design [6]. MATLAB's dense language concepts were transparently extended to support computations on sparse matrices, while remaining true to the mathematical concept of a matrix. In this work, we have similar goals, seeking to extend the region-based array language ZPL to support computations on *sparse arrays*. However, our work differs in that MATLAB is a serial, interpreted language whereas ZPL is parallel and compiled. Moreover, while MATLAB provides implicit, interpreter-managed sparse arrays, ZPL provides the programmer with an explicit means of specifying and operating over sparse index sets. This enables a rich variety of sparse computing styles that cannot be expressed succinctly in MATLAB.

It should be noted that we distinguish between matrices and arrays in this work—the former being a special case of the latter with inherent mathematical interpretations and properties. Our goal is *not* to design a language that provides sparse matrix support, for example by including factoring and solving routines as primitives. Rather, we provide support for a general sparse array data structure and array-based operators.

By way of analogy, ZPL does not contain an explicit matrix multiplication operator, but instead provides array operators with which users can express standard parallel matrix multiplication routines such as the SUMMA algorithm [14]. Similarly, we view sparse arrays as a tool for representing arbitrary sparse data structures, of which sparse matrices are just one instance.

The rest of this paper is organized as follows: The next section characterizes different sparsity types used in the real-world that we believe a sparse array language should support. Section 3 provides a brief introduction to ZPL, emphasizing those features that we extend to support sparse computation. We discuss these extensions in Section 4. Section 5 discusses implementation issues and gives some preliminary results demonstrating the benefits obtained using sparse regions. Section 6 summarizes related work for expressing sparse computations, and we conclude in Section 7.

2 Characterizing Sparsity

Before describing our region-based approach for sparse computations, it is worthwhile to consider the different sparsity characteristics that a real-world problem might have. To begin with, we classify the dynamic properties of sparsity as falling into four rough categories:

static the sparsity pattern is known and expressible at compile time. For example, a tridiagonal or densely banded matrix.

runtime constant the sparsity pattern is not known until runtime, but will be fixed for the program's duration. For example, the indices representing coastlines in a discretized ocean modeling application.

dynamic predictable the sparsity pattern will change during the course of the program's execution, but in ways that can be anticipated. For example, the fill-in that might be produced by matrix operations on well-ordered sparse matrices.

dynamic chaotic the sparsity pattern will change during the course of the program's execution, but in completely arbitrary ways. For example, a sparse implementation of the game of life.

It is our intention to introduce language features to support computations with any of these dynamic characteristics. It should be clear that the costs associated with each of the above categories can be vastly different. For example, dynamic chaotic sparsity results in runtime overheads stemming from the computation of new sparsity patterns, updating the sparse representation, and managing the changing memory requirements. In contrast, static sparsity requires no dynamic restructuring and gives the compiler access to an array's sparsity pattern, enabling opportunities for optimization. As a result, our intention is to make the distinction between these sparsity types clear at the language level, so that the users have a notion of the computational overhead induced by their code.

In addition, sparse computations have different density characteristics, which we categorize as follows:

sparse complete an operation on all of the elements of a sparse array.

sparse subset an operation over a sparse subset of elements in a dense (or possibly less sparse) array.

dense superset an operation that reads a sparse array as though it were a fully allocated dense array.

sparse-dense mix an index set that is sparse in some dimensions but dense in others. For example, a sparse subset of dense planes from a 3-dimensional index space.

As with the dynamic sparity characteristics, our intent is to support all of these density types in our language. The use of sparse regions makes this trivial and gives programmers full control over the types of sparsity they require.

3 Introduction to ZPL

ZPL is an array-based data-parallel language that contains support for dense arrays and regularly-strided sparse arrays [12]. Its features have proven popular with applications programmers, and ZPL is supported for most modern parallel architectures. ZPL has an associated performance model that allows programmers to reason about the parallel implementation of their code without forcing them to program at a per-processor level [2]. The ZPL compiler produces efficient code that is competitive with hand-coded C [3] and which tends to outperform High Performance Fortran (HPF) [11]. Given ZPL's successes in the domain of regular array computation, it seems only logical to consider extending it to support sparse computations. We do this in the context of Advanced ZPL (A-ZPL), the successor language to ZPL. This section gives a brief introduction to some of ZPL's fundamental concepts. A more thorough presentation is available elsewhere [12].

In addition to standard support for constants and variables, ZPL provides *configuration variables* for defining runtime constants—values that are set at the outset of a program's execution and then remain constant for the duration of the program. Configuration variables are given default initializing values that can be overridden on the command line. These variables tend to be invaluable for describing values such as problem sizes, numeric tolerances, and input files to a program. For example:

```
config var n:integer = 100;            -- problem size
           epsilon:double = 0.0001;    -- tolerance
           filename:string = "A.mat";  -- input filename
```

Configuration variables are commonly used to define regions. As described in the introduction, regions are a programmer-specified representation of a regular rectangular set of indices. For example, the following declaration defines the index set $\{(1,1),(1,2),\ldots,(n,n)\}$:

```
region R = [1..n,1..n];   -- declare an n x n index set
```

ZPL's support for regions includes a set of operators useful for describing boundary conditions, strided regions for use in hierarchical multigrid problems, and other common transformations on index sets [4].

Regions serve two purposes in ZPL. The first is to declare parallel arrays. For example, the following declaration creates three $n \times n$ arrays of integers and an $n \times n$ array of booleans:

```
var A, B, C:[R] integer;   -- n x n integer arrays
    Mask:[R] boolean;      -- an n x n boolean array
```

The second use of regions is to specify the indices over which an array operation should take place. For example, the region preceding the following statement indicates that elements from the i^{th} row of A and B are to be added and assigned to their corresponding element in C:

```
[i,1..n] C := A + B;   -- add the i-th rows of A and B, assigning to C's i-th row
```

These *region specifiers* are dynamically scoped and can inherit from the enclosing scope, allowing code to be written in a region-independent manner.

In addition to simple element-wise operators such as the addition and assignment operators shown above, ZPL provides a number of *array operators* that support more complex array manipulations. These operators include reductions, parallel prefix operations, subarray replication, permutations, stencil operations, and boundary conditions.

Regions have two main benefits in ZPL. The first is to unclutter the user's code by factoring indexing expressions traditionally scattered throughout an array statement into a concise, dynamically scoped prefix. The second is to emphasize *index locality* so that the relationship between the elements accessed by each array operand is clear and obvious to the programmer [4]. This is the basis for ZPL's WYSIWYG performance model, which allows the parallel overheads associated with every statement to be easily detected and reasoned about by the programmer [2]. Our goal in this work is to extend ZPL's regular regions to support sparse index sets without sacrificing the syntactic, semantic, and performance benefits that were achieved in the non-sparse case.

ZPL currently supports a notion of operating on arbitrary indices using *masking*. This is a variation on the region specifier in which a boolean array is used in combination with the region. For example the following statement sums and assigns corresponding elements of A, B, and C, for all indices in R at which the variable Mask is true:

```
[R with Mask] C := A + B;   -- assign A + B to C wherever Mask = true
```

Although ZPL's masking allows for computation over arbitrary index sets, it is not sufficient for efficiently expressing sparse computation. For one thing, due to the presence of the $n \times n$ region R, the computational complexity of the statement above will be $\Theta(n^2)$, even if only $\Theta(n)$ elements of Mask are true. In sparse computations, one would like the asymptotic complexity to be proportional to the number of represented values. Furthermore, ZPL's masking does not support the declaration of sparse arrays, but merely "sparse" computations over dense arrays. This results in wasted memory and poor cache performance in true sparse applications. In the next section, we discuss the design of sparse regions in A-ZPL to support efficient sparse computation.

4 Sparse Computation in A-ZPL

4.1 Sparse Regions and Arrays

A-ZPL's approach for supporting sparse computations is to allow the creation of sparse regions by extending ZPL's masking concept. Whereas masking associates a dense mask with a dense region to select a sparse subset of indices, A-ZPL's sparse regions will explicitly represent the index set, so that all computations over them will be proportional to the number of represented indices rather than the size of the index space.

To this end, declaring a sparse region will be done in a manner similar to masking, substituting the keyword swith for with. For example, the following declaration takes a snapshot of the indices in R where the variable Mask is true and associates those indices with Rs:

```
region Rs = R swith Mask;
```

As a result, although the following two statements achieve the same effect, the time required by the first is $\Theta(n^2)$, whereas the second will be proportional to the number of represented indices:

```
[R with Mask] A := B + C;
[Rs] A := B + C;
```

Sparse regions can be used to declare arrays, just like traditional regions:

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

Arrays declared using a sparse region only allocate storage for elements corresponding to the region's represented indices (plus one additional element to store the unrepresented value, set to 0 by default). Decoupling the sparse indices from the array in this manner allows each array's represented values to be allocated densely in memory (Section 5).

Note that sparse arrays are simply an efficient representation of an array that is conceptually dense. To illustrate this, consider the eight simple assignments expressible using the declarations so far:

```
[R] A := B;       -- DDD: Dense region, Dense LHS, Dense RHS
[R] A := Bs;      -- DDS: Dense region, Dense LHS, Sparse RHS
[R] As := B;      -- DSD: Dense region, Sparse LHS, Dense RHS—illegal
[R] As := Bs;     -- DSS: Dense region, Sparse LHS, Sparse RHS—illegal
[Rs] A := B;      -- SDD: Sparse region, Dense LHS, Dense RHS
[Rs] A := Bs;     -- SDS: Sparse region, Dense LHS, Sparse RHS
[Rs] As := B;     -- SSD: Sparse region, Sparse LHS, Dense RHS
[Rs] As := Bs;    -- SSS: Sparse region, Sparse LHS, Sparse RHS
```

The first case (DDD) is simply a traditional ZPL assignment of n^2 elements. In the second case (DDS), a sparse array is read within the context of a dense region. At first glance, this might seem illegal since Bs does not have memory allocated for all indices in R. However, values that are not explicitly stored are represented implicitly by the unrepresented value. Thus, this statement assigns each element of A its corresponding value from Bs for indices defined in Rs and the unrepresented value for all other indices in R.

The next two cases (DSD and DSS) are illegal, since they try to write to indices in As for which there is no associated memory. This violates ZPL's basic principle that the left-hand side of an assignment must have memory allocated for all indices in the enclosing region specifier.

The following two cases (SDD and SDS) perform a sparse assignment of a dense array. These statements only assign elements of A corresponding to the indices in Rs—the remainder are left unchanged. Whether the right-hand expression is sparse or dense, only elements corresponding to Rs are accessed. The final two statements (SSD and SSS) are similar, except that the left-hand side is sparse, causing all of its elements to be overwritten.

These cases illustrate sparse complete (SSS), sparse subset (SDD, SDS, SSD), and dense superset (DDS) styles of computation. Note that this richness of expression is a direct result from the use of regions to divorce index sets from arrays. By way of contrast, MATLAB only allows users to operate concisely over regular sections of an array's values, or to index explicitly into the arrays when an irregular subset of array values is required. Sparse regions give a programmer direct, concise control over sparse array allocation and computation.

Sparse-dense mixes are also expressible in this scheme, relying on ZPL's concept of flood dimensions for replicated storage. Due to space constraints, we do not introduce flood dimensions and their use in declaring sparse-dense regions here.

Mixing Different Sparsity Patterns The eight basic assignments listed above become more interesting as different sparsity patterns are used in combination. However, the simple rules outlined above remain the same: left-hand side arrays must be defined for all indices in the enclosing region scope, while right-hand side arrays can always be read as a dense set of values, regardless of their allocation. As a result, the conformability rules for sparse computation are a simple extension to those present in ZPL.

Sparse Interpretation of Array Operators By definition, the interpretation of an array operator within a sparse region specifier is identical to its operation within the equivalent densely masked region. Similarly, applying an array operator to a sparse array has a straightforward definition due to the sparse array's dense interpretation. Thus, sparsity represents a transparent extension to the base ZPL language, eliminating the need for detailed rules for complex sparse expressions.

4.2 Expressing Static and Dynamic Sparsity

Sparse regions are sufficient for expressing sparsity patterns with various dynamic characteristics. This is done using A-ZPL's support for array constants, array configuration variables, and *dynamically bound regions*. We begin by showing an example of a region with a constant sparsity pattern:

```
constant TriDiag:[R] boolean = abs(Index1 – Index2) <= 1;
region RTri = R swith TriDiag;
```

In this example, an array constant TriDiag is declared over region R and defined in terms of the compiler-provided constants Index1 and Index2. Index*i* is defined to be a constant array in which the value of each element is equal to its index in the i^{th} dimension. Thus, TriDiag is "true" for all elements on the tridiagonal. TriDiag is then used to define the sparse region RTri to be those indices along the tridiagonal. Note that the definition of TriDiag could have been inlined directly into RTri's declaration to eliminate the dense representation of TriDiag.

If a static sparsity pattern cannot be represented as a constant expression, but is computable at runtime (possibly by reading it from a file), the declaration can be made using an array configuration variable:

```
config var Pat:[R] boolean = ComputePattern();
region RFixed = R swith Pat;
```

One proposed extension for A-ZPL is the promotion of regions to a full type, in which case RFixed could be declared as a configuration variable and read directly from a file rather than being declared in terms of an array.

To express dynamic sparsity patterns, the programmer must use dynamically bound regions as in the following example:

```
region Rdyn = R swith ?;
var Adyn, Bdyn, Cdyn:[Rdyn] integer;
```

Replacing the sparsity pattern with a question mark allows it to be set at any time during the program's execution. This is done by referring to the region's sparsity pattern using the unary $ operator:

```
Mask1 := (A > delta) and (B = 0);
$Rdyn := Mask1;
⋮
Mask2 := (B > delta) and (A = 0);
$Rdyn := Mask2;
```

Each assignment to Rdyn's sparsity pattern results in the reallocation of all arrays defined in terms of it (*i.e.*, Adyn, Bdyn, and Cdyn). All indices that are common to the old and new sparsity patterns will have their values preserved. All new indices will have their values set to that of the unrepresented element (since the element's value is being converted from an implicit representation to an explicit one). This dynamic restructuring of sparse indices and array values is costly by nature and therefore emphasized symbolically to programmers in the $ operator (which also resembles an "S" for "sparsity").

This technique is useful for representing both predictable and chaotic dynamic sparsity patterns. Since the syntax makes programmers aware of the overheads associated with changing a region's sparsity, algorithms with predictable dynamic sparsity patterns may be written in an attempt to reduce the number of restructurings. For example, sparsely banded matrices can be naively represented using a dynamically bound region that is incrementally updated as matrix operations cause fill-in. Or, to avoid the overhead of repeated restructuring, the programmer could statically declare the region to be fully banded, as in the tridiagonal case above. Matrices with more complicated fill-in patterns might be amenable to approaches between these two extremes in which the region is restructured infrequently.

Note that the region sparsity accessor, $, can be used not only to assign a region's sparsity pattern, but also to read it. For example, the following declaration creates a region whose sparsity pattern is the union of two others':

```
region RsTot = $Rs1 | $Rs2;
```

The $ operator can be thought of as returning an implicit boolean array whose values indicate the sparsity of the region. These implicit arrays can be used in general array computation, just like any other.

Using the region syntax described here, we have written codes that implement tridiagonal matrix multiplication, sparse matrix-matrix multiplication using the SUMMA algorithm, coastline computations in the *tropic* portion of the MOM ocean simulator, boundary conditions for irregular instances of solving Laplace's equation, the sparse component of a fuzzy clustering algorithm, and a sparse implementation of the game of life. We expect that once sparsity is implemented in the A-ZPL compiler and made available to users, an abundance of other interesting applications of sparse regions will be found.

5 Implementation

In this section, we discuss the issues that will be involved in supporting sparse regions and arrays in the A-ZPL compiler.

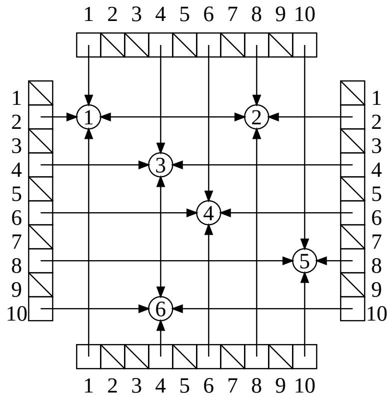
5.1 Runtime Region and Array Representations

The implementation decision of primary importance is related to the language's separation of sparse index sets and arrays. Since regions are distinct from arrays in A-ZPL, they are represented separately at runtime. This yields a great benefit in the sparse case, since much of the overhead in sparse computations is related to the representation of the sparse structure. Sparse index representations need only be stored for each sparse region. Every array declared in terms of the region can simply refer to its sparse structure as necessary. The array values themselves can therefore be allocated densely and accessed using a unique index stored at each node in the sparse representation (Figure 1).

The net effect of this implementation choice is that only a single sparse structure will have to be traversed for each unique sparsity pattern in a statement. The result is that overheads related to sparse traversal are amortized across array expressions with the same sparsity. By way of contrast, if each array stored its own sparsity pattern, multiple sparse structures would have to be traversed, even if they were all identical. In A-ZPL, this worst-case scenario will occur only when it must—when each array has a unique sparsity pattern (and therefore a different defining region).

0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	28	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	44	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	66	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	80
0	0	0	0	0	0	0	0	0	0
0	0	0	104	0	0	0	0	0	0

(a) a sparse array



(b) a representation of its defining region

0	1	2	3	4	5	6
0	21	28	44	66	80	104

(c) the array's sparse representation

Figure 1: An illustration of how sparse regions separate the representation of a sparse index set from the actual sparse data. The sparse array in (a) is shown to be separated into its indices (b) and its data (c). Each node in (b) contains an index for its corresponding data element in the array's packed representation in (c). Note that multiple arrays with the same sparsity pattern can share a single sparsity structure, allowing the overhead of storing and traversing the sparse index set to be amortized between them. Sparse regions enable this implementation. Note that each node in (b) also stores its logical index (not shown here) and that the 0^{th} element of (c) stores the unrepresented element's value (in this case, 0).

Although we have not settled on a precise sparse representation, it is clear that the main characteristic of our choice will have to be flexibility. A-ZPL's array operators require arrays to be traversed in any dimensional and directional order, and for subarrays and nearest neighbors to be located quickly. As a result, we envision a sparse representation in which each represented index can quickly access its neighbors in both directions of each dimension. In addition, we envision a dense indexing subarray allocated for each dimension that points to the first and last indices in that position (Figure 1 (b)).

Each node in the sparse region structure will contain a unique value that can be used to index into the packed representation of sparse arrays declared over the region. Necessarily, the nodes will also store the index position that they represent.

5.2 Parallel Issues

Although the description above could be used for a sequential implementation, we envision it to exist on each node of a parallel machine. The base index set of each sparse region will be distributed across the processor set as with dense regions [4]. Each processor will then allocate the sparse structure described above to store its subset of the global index space.

We recognize that sparse computations are often more sensitive to load balancing than dense computations, and therefore expect to extend A-ZPL's distribution capabilities to handle more complex partitions. In doing this, we intend to preserve ZPL's *region distribution invariant* [2] so that the performance model will extend transparently to the sparse domain. This will allow programmers to quickly identify which operations are completely parallel and which require specific types of communication (*e.g.*, point-to-point, subdimension broadcast, all-to-all).

5.3 Compiler Issues

The current ZPL compiler generates different loops for array statements depending on whether the region specifier's dimensions are dense, strided, or flooded. Although general-purpose loops can be constructed, they tend to contain unnecessary overheads for the simpler cases. Similarly, arrays are accessed in different ways depending on the characteristics of their defining regions. As a result, each statement is implemented by determining the regions that could be involved and generating loops and accesses of the appropriate type (note that this decision is obscured by aliasing and dynamically inherited region scopes).

We expect that the support of sparse regions and arrays will simply be an extension of this code generation phase in which dimensions may also be sparse. Loops will therefore be generated that iterate over a region's sparse structure, and arrays will be accessed appropriately. Though the interplay of sparse and dense regions and arrays can result in a large number of possibilities as demonstrated in the previous section, our current compiler infrastructure is easily extensible to handle each case without explicitly considering the cross product of possibilities.

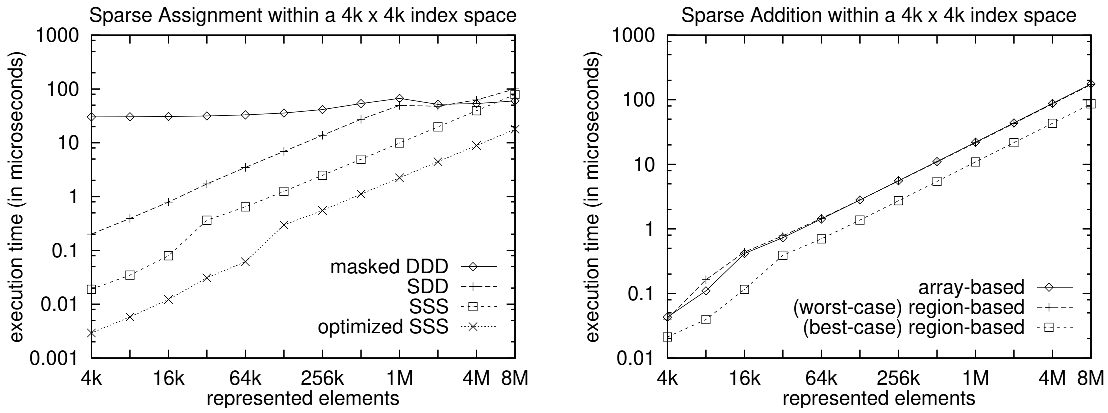
One challenge in supporting sparse computation is that all regular regions and arrays could be described by one "most-general" looping or accessing method, since they were based on rectangular index sets with a high degree of regularity. Since sparse dimensions break this model, procedures written to work in both sparse and dense domains may be candidates for specialization in the compiler, so that sparse and dense versions are generated to ensure the efficiency of each.

In addition to performing our traditional optimizations in a sparse context (most notably array contraction [9] and communication optimizations [5]), we expect that there will be many new opportunities for optimization exposed by language support for sparsity. One simple example is that array statements which involve only a single sparsity pattern can be implemented by ignoring the sparse structure altogether and performing the operation directly on the arrays' dense representations in memory.

5.4 Preliminary Experiments

In this section, we present some simple experiments that demonstrate the benefits of sparse language support. These experiments use a naive sparse representation similar to that described above, in which region nodes are allocated densely in memory and refer to their neighbors via pointers. We hand-generated code similar to that which we would expect a complete A-ZPL compiler to produce. All experiments were run on 16 nodes of a Cray T3E running at 450 MHz.

In the first experiment, we demonstrate the advantage of using sparse regions and arrays over dense regions with masking. A sparse array assignment is implemented in four ways: (i) masked DDD: using



(a) (b)

Figure 2: Initial performance results timing the execution of simple sparse array statements with varying numbers of represented elements. (a) Sparse assignment implemented in four ways: using dense arrays and a masked dense region, using dense arrays and a sparse region, using sparse arrays, and using an optimized sparse array assignment that operates directly on the packed data allocations. (b) Sparse addition comparing the cost of storing sparsity patterns in regions vs. the traditional method of making them part of the array's data structure. Region-based sparsity is shown in its best- and worst-case scenarios: when all arrays share a common region and when they each have a unique region.

dense arrays and dense masking, (ii) SDD: using dense arrays and a sparse region, (iii) SSS: using sparse arrays and a sparse region, and (iv) optimized SSS: an optimized version of SSS in which the assignment is performed directly over the dense allocation (as described in the previous subsection). The assignment is done within the context of a 4096×4096 index space for a varying numbers of represented elements (n to $n^2/2$) distributed uniformly throughout the index space.

Several observations can be made: The masked DDD implementation takes roughly the same amount of time regardless of array density, due to the fact that n^2 elements of the mask must be read. In contrast, all of the sparse versions scale proportionally to the number of represented elements. As hoped, the sparse implementations are significantly cheaper than the dense version until the number of represented elements approaches $n^2/2$. Furthermore, each sparse implementation is approximately an order of magnitude faster than the previous. SDD is faster than SSS due to the fact that the memory footprint of the sparse arrays is smaller than that of their dense counterparts. Similarly, optimized SSS outperforms SSS since the sparse region structure does not need to be traversed.

Our second experiment shows the savings available by associating sparsity with regions rather than arrays. We perform sparse addition using two representations: one in which the sparsity pattern is associated with the region as proposed here, and a second in which sparsity is associated with arrays as is traditional. For the region-based approach, we run two versions: one in which each array has its own region and one in which they all share a common region. These represent the best- and worst-case scenarios for region-based sparsity. The graph indicates that when sparsity patterns are shared by multiple arrays, regions result in faster runtimes. Yet when arrays have different patterns, the region-based approach performs similarly to array-based techniques. The overhead of the region-based approach is noticeable in less dense arrays due to the fact that the region representation is separate from the array values in memory, resulting in a larger footprint. In conclusion, our experiments demonstrate that sparse regions can result in significant performance improvements over more naive sparse array representations.

6 Related Work

Few parallel programming languages have included direct support for sparse computation, instead requiring users to build their own sparse data structures and manipulate them explicitly. One such example is NESL [1], a functional language that allows the construction of nested parallel data structures. These structures can be used to explicitly construct standard sparse array representations such as compressed column storage. This approach puts the burden of the sparse representation on users, forcing them to deal with low-level details that could be handled by the compiler. Furthermore, the compiler has no means of detecting that a data structure represents a sparse array, and therefore cannot perform optimizations specific to the sparse context. In contrast, our approach allows for the clear representation of a sparse computation at a global level, leaving details of representation and optimizations to the compiler.

High-Performance Fortran [7] is perhaps the most prominent parallel language. Although it has no inherent support for sparse arrays, an extension to HPF has been proposed by Ujaldon *et al.*, which allows for the declaration of sparse arrays using a variety of standard storage schemes [13]. Although this exposes the sparse array representation to the compiler, computations over the arrays still require users to directly refer to the underlying sparse data structure. This is an unfortunate burden to place on users, obfuscating a code's meaning. In contrast, the sparse and dense versions of A-ZPL algorithms are quite similar in appearance.

Although parallel libraries have been developed for representing and operating on sparse arrays (*e.g.*, [8]), these have typically assumed a sparse matrix interpretation, providing support for linear algebra operations. We believe that although such libraries are valuable, support for sparse computation at the language level aids in the clear expression of the programmer's computation. Ideally, a sparse array language would provide a means of interfacing to sparse matrix libraries to take advantage of the efforts in this area.

7 Conclusions

In this paper, we have proposed a region-based approach for representing sparse computations. We have shown that regions are a clean, concise means of expressing sparsity. Furthermore, we have argued that sparse regions admit an efficient parallel implementation as well as parallel performance modeling. Our experiments demonstrate that languages using sparse regions can achieve significantly improved execution times, due to the possibility of optimizations as well as the opportunity to amortize sparse overheads between arrays with identical sparsity patterns. In future work, we intend to implement sparse regions and arrays in the A-ZPL compiler, seeking to develop new language concepts and optimizations specific to the sparse context.

Acknowledgements The authors would like to thank Donna Calhoun, Jason Secosky, and Greg Warnes for their help in motivating this work. Additional thanks to Sung-Eun Choi, Derrick Weathersby, and Ruth Anderson for their ideas in the early stages of the design. This research was supported by a grant of HPC time from the Arctic Region Supercomputing Center.

References

[1] 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.

[2] 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 Parallel Programming Models and Supportive Environments*, pages 50–61. IEEE, March 1998.

[3] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Lawrence Snyder, W. Derrick Weathersby, and Calvin Lin. The case for high-level parallel programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–85, July–September 1998.

[4] Bradford L. Chamberlain, E Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: An abstraction for expressing array computation. Technical Report UW-CSE-98-10-02, University of Washington, October 1998.

[5] Sung-Eun Choi and Lawrence Snyder. Quantifying the effect of communication optimizations. In *Proceedings of the International Conference on Parallel Processing*, pages 218–222, August 1997.

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

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

[8] S. A. Hutchinson, J. N. Shadid, and R. S. Tuminaro. *Aztec User's Guide: Version 2.0*. Sandia National Laboratories, September 1998.

[9] E Christopher Lewis, Calvin Lin, and Lawrence Snyder. The implementation and evaluation of fusion and contraction in array languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–59, June 1998.

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

[11] Ton A. Ngo, Lawrence Snyder, and Bradford L. Chamberlain. Portable performance of data parallel languages. In *SC97: High Performance Networking and Computing*, November 1997.

[12] Lawrence Snyder. *Programming Guide to ZPL*. MIT Press (in press), 1998.

[13] 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.

[14] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. Technical Report TR-95-13, University of Texas, Austin, Texas, April 1995.