Parallel Language Support for Multigrid Algorithms*

Bradford L. Chamberlain Steven Deitz Lawrence Snyder

University of Washington, Seattle, WA 98195-2350 USA

{brad,deitz,snyder}@cs.washington.edu

Abstract

Multigrid algorithms are a computational paradigm that enjoy widespread use in the scientific community. While parallel multigrid applications have been in use for quite some time, parallel language support for features common to multigrid algorithms has been lacking. This forces scientists either to express their computations in high-level terms without knowing the parallel impact, or to explicitly manage all the details by hand, thereby diverting their attention from the algorithm itself. In this paper, we enumerate properties that would be desirable for any parallel language that hopes to support multigrid application development. We then explain how these properties influenced the design of support for hierarchical arrays in ZPL, our array-based parallel programming language. We describe the implementation of these features in the ZPL compiler and its runtime system. In addition, we show that our approach performs competitively with hand-coded Fortran + MPI for the NAS MG benchmark, outperforming it on 256 processors. In addition, a static comparison of the two codes demonstrates ZPL to be the more concise, readable, and flexible implementation.

1 Introduction

Multigrid algorithms enjoy widespread use in the scientific community due to their ability to produce accurate results in significantly less time than a direct, non-hierarchical approach [2, 1, 7]. These applications utilize *hierarchical arrays*, which consist of a number of *levels*, each with fewer elements than the previous level — typically half as many elements per dimension. Real-world multigrid applications typically involve large problem sizes and many iterations. For this reason, parallel implementations are often developed to reduce the computational running time. Due to the prevalence of this technique, it seems natural that parallel programming languages should provide users with the ability to create hierarchical arrays that are easy to manipulate, whose parallel implementation is made clear to the programmer, and which result in efficient parallel execution. Yet, most modern parallel languages fail in this respect, either by forcing the programmer to deal with low-level implementation details that are a distraction from the algorithm at hand, or by providing them with high-level features that have no performance model with which to evaluate implementation alternatives.

*This research was supported in part by a grant of HPC time from the Arctic Region Supercomputing Center.

In this paper we describe the design and implementation of ZPL's support for multigrid algorithms. ZPL is an array-based parallel programming language which has proven successful in the domain of non-hierarchical data parallel computations [3, 12]. ZPL programs tend to be concise and easy to read, yet result in execution times that are similar to hand-coded message passing programs. ZPL is unique among parallel programming languages in that it combines a global view of array computations with a performance model that allows users to evaluate the parallel implications of their code at the syntactic level [4].

This paper constitutes the first description of how we extended ZPL's traditional constructs to elegantly express hierarchical computations using *multiregions* and *multiarrays*. Although these concepts have been described elsewhere [19], this is the first discussion of the principles that guided their design, the language design itself, and the implementation of both hierarchical and non-hierarchical regions and arrays in the ZPL compiler and runtime. We evaluate our approach by comparing a ZPL implementation of the NAS MG benchmark with the original hand-coded Fortran + MPI version. Comparisons are made both in terms of the static programs and their parallel runtime performance. This paper is the first presentation of these results [1]

This paper is organized as follows: In the next section we give an enumeration of factors that play a role in the efficient parallel execution of multigrid applications. Section 3 gives a brief introduction to ZPL and describes how these factors influenced the design of hierarchical array support in ZPL. Sections 4 and 5 describe how we implemented multiregions and multiarrays in the ZPL compiler and runtime. We give performance results for a ZPL version of the MG benchmark in Section 6, making comparisons with the hand-coded NAS implementation. In Section 7 we summarize language support for multigrid algorithms in other parallel programming languages, and in Section 8 we conclude.

2 Parallel Multigrid Considerations

We begin by considering multigrid algorithms abstractly, defining a set of properties that would be desirable in parallel languages to aid in the clean expression and efficient implementation of multigrid applications. This list can then be used to evaluate a language's support for multigrid applications, or to guide the design of new concepts that might aid in designing such support. For the purposes of this discussion, let us characterize a multigrid application as having d dimensions per level with $n_1 \times n_2 \times ... \times n_d$ elements at its finest level, and l levels altogether.

Initially, let us restrict ourselves to characteristics that apply not only to parallel multigrid applications, but also to sequential ones. The first property that comes to mind is that the language should allow programmers to write multigrid applications in which the values of n_i, l, and potentially even d are left unbound until runtime. We'll refer to this as *dynamic parameterization*. Forcing programmers to statically specify

[1]ZPL performance figures for NAS MG have been published previously [13], but using an approach that predated the hierarchical language constructs, relying instead on macro expansion and code replication to a degree that clashes with good language design principles.

these values is a severe inconvenience, requiring recompilation for every new problem size or degree of refinement that they care to run.

Once these values have been given names, the language should support their use in declaring hierarchical arrays. Due to the fact that multigrid applications tend to perform the same operations at each level of the array, the hierarchical array should be indexable by level so that the user can iterate over levels of the hierarchy, applying the same computation to each level. In this sense, hierarchical arrays can be thought of as $d+1$-dimensional arrays where the additional dimension corresponds to the level number. However, they differ from traditional arrays in that each level contains a different number of elements. The ability to declare and iterate over such arrays will be called *hierarchical array support*. All operations that a language supports on traditional non-hierarchical arrays — *e.g.*, indexing, reductions, permutations — should also be applicable to a single level of a hierarchical array. Furthermore, a single level of a hierarchical array should serve as a legal actual parameter for any formal parameter whose type is a traditional array of the same element type. We refer to these last two properties as *hierarchical transparency*. Hierarchical array support and hierarchical transparency both reinforce basic principles of code reuse, thereby simplifying the programmer's task and allowing for a more concise, expressive program.

Next, let us consider memory allocation for the hierarchical array. Each level of the array should be allocated densely in memory to ensure good cache performance. Furthermore, the total memory consumption for the hierarchical array should correspond to the total number of elements in the hierarchy. In particular, allocating a $n_1 \times n_2 \times \ldots \times n_d \times l$ rectangular array is completely unacceptable. We refer to these properties as *compact allocation*. Hierarchical arrays should also have *persistent storage* in which array values at all levels of the hierarchy are preserved across iterations over the hierarchy. For instance, this rule would be broken by languages that only provide support for hierarchical arrays by iterating over the levels recursively and allocating a local array for each call that is twice as small (or big) as that of the previous level. Although this approach results in an implicit hierarchical array structure, the array values are lost upon returning from the recursive calls and therefore would not be preserved for subsequent iterations. Maintaining values across iterations is a requirement for many multigrid applications, such as adaptive mesh refinement techniques (AMR) [11]. Finally, each level of the hierarchy should support *sparse membership* so that memory is not wasted if the programmer only wishes to refine certain areas of the problem space, and these sparsity patterns should be dynamically computable.

Finally, the language should support clean *stencil expressions* within a level and between levels since these are the most prevalent operations in multigrid applications. These expressions should include convolutions at a given level of the hierarchy, restrictions from a fine level of the hierarchy to a coarser level, and interpolations from a coarse level to a fine level. To reduce the number of special cases for these expressions, there should be support for *boundary condition specification* and for periodic boundary conditions in particular, due to their prevalence.

Let us now turn our attention to characteristics that are specific to parallel implementations of multigrid applications. To begin with, the programmer should be able to write programs without specifying the number of processors at compile-time. In parallel computing environments, processor availability can change quickly, and recompiling for each potential processor configuration quickly becomes a nuisance for users. Thus, *dynamic processor specification* is key for programmer convenience.

Load balancing is essential for making the best use of parallel computing resources. For example, dense multigrid applications can often be load balanced simply by dividing each level of the hierarchical array into equal-sized blocks.

Minimal communication should be used to access nonlocal data values. For example, in computing a 9-point stencil for a $d = 2$ dense blocked multigrid application, only one nonlocal row/column/element of data needs to be transferred from each of the 9 logical directions, so this should be implemented with a minimal point-to-point data transfer scheme rather than a more general and expensive all-to-all communication. In short, the communication costs should be proportional to the complexity of the stencil and the data distribution. Once communication has completed, the non-local values should be stored in such a way that they can be accessed as cheaply as local data values. We call this *nonlocal value transparency*.

Finally, the language should give the user some sense of the *parallel performance implications* of their design choices: how does the user's implementation affect the communication, load balancing, and concurrency of the program. Preferably, these cues should be given at an intuitive level such as syntactic cues rather than through profiling and post-processing feedback mechanisms.

This completes our "wishlist" of properties for parallel multigrid language support. We make no claims that this list is exhaustive, but rather put it forth as a prototype, believing that it covers the fundamental issues that are common to a wide variety of multigrid codes. Due to limited space and the inherent complexity of the problem, we postpone our discussion of sparse multigrid applications until a later date, focusing on the dense case in this paper. We believe that without sufficient support for the dense case, the elegant and efficient expression of sparse multigrid applications is unlikely, and so we focus on the easier problem for now.

3 Multigrid Support in ZPL

This section gives a brief review of ZPL's region, array, and direction concepts and then describes our design process for extending these concepts to support multigrid applications. For a more detailed introduction to ZPL, see [19].

ZPL's fundamental concept is the *region*. Regions are index sets with no associated data. They are commonly named, though dynamic region specifiers can also be inlined directly into the code. For example:

```
region R    = [1..n,   1..n  ];
       BigR = [0..n+1, 0..n+1];
```

These declarations define region R, which is an $n \times n$ index set, and BigR, which extends R by one row and column in each direction. Regions' sizes can be described using constants or *configuration variables* — runtime constants that are set at the beginning of a ZPL program's execution.

Regions are used to declare *parallel arrays*. To declare two arrays of doubles over the indices specified by BigR, one would write:

```
var A, B: [BigR] double;
```

Regions also specify computation over arrays. For example, the following statement performs elementwise assignment from B to A *only* for those indices described by R:

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

A region's index set is partitioned across the processor set and thus is a source of parallelism. ZPL's performance model specifies that A and B will be distributed identically [4], and therefore the statement above is completely parallel.

Another basic concept in ZPL is the *direction*. A direction is simply an offset vector. The @ *operator* takes an array and a direction as operands and shifts references to the array by the offset. So to replace each element of A with the sum of its northwest and southeast neighbors, one would write:

```
[R] A := A@[-1,-1] + A@[1,1];
```

To improve program readability, directions are typically named. For example, one could rewrite the above computation as:

```
direction nw = [-1,-1];
          se = [ 1, 1];
...
[R] B := B@nw + B@se;
```

ZPL's performance model indicates that the @ operator represents an array reference requiring point-to-point communication [4]. Other operators support reductions, parallel prefix operations, floods, remaps, and boundary condition support, and each of these has a particular form of communication associated with it. This allows the programmer and compiler to trivially identify where data transfer is required and what type of communication will take place. For example, when the programmer sees an @ operator, s/he knows that point-to-point communication is required, whereas the remap operator would indicate an expensive all-to-all communication.

In addition to dense regions, ZPL supports strided regions and arrays. For example, to declare a strided region S2 that omits all of R's even numbered indices and an array C over this region, one writes:

```
region S2 = R by [2,2];
var C : [S2] integer;
```

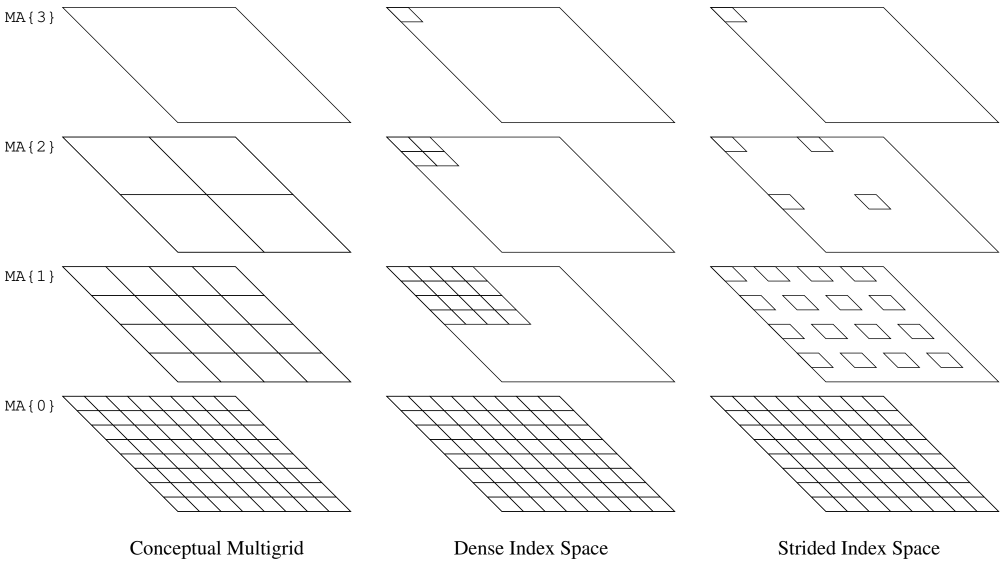

Figure 1: Three possible views of hierarchical array MA: the conceptual view, a view of the data with dense indices, and a view of the data with strided indices as encouraged by ZPL's performance model.

Note that the "step" [2,2] is simply a direction and could be named as usual. This region is partitioned over the processor set in the same way as R. Therefore the statement:

```
[S2] A := B + C;
```

involves no interprocessor communication. Furthermore, the elements of A not referred to by S2 remain unchanged.

Although strided regions are sufficient for describing any individual level in a hierarchical array, each level would have to be explicitly declared and named, violating our notion of hierarchical array support. For this reason, we introduce the concept of parameterizing ZPL directions, regions, and arrays. For example, the following declares a *multidirection*, mse{}:

```
direction mse{0..3} = [1,1] scaledby 2^{};
```

which describes the four directions: [1,1], [2,2], [4,4], and [8,8]. To refer to the direction [2,2], the programmer would simply write mse{1}. This can then be used to define a multiregion:

```
region MR{} = R by mse{};
```

which in turn can define a multiarray:

```
var MA{} :  [MR{}] integer;
```

Thus, MA{} is a 2-dimensional hierarchical array with four levels, and indices corresponding to those described by MR{}.

At this point it is worth noting that there are two different indexing schemes for hierarchical arrays in any language. For example, to express the hierarchical array conceptualized in Figure 1(a) we could either define level i densely as [1..n/(2ˆi),1..n/(2ˆi)] (Figure 1(b)), or we could define each level as being [1..n,1..n], but strided by larger and larger amounts (Figure 1(c)). In order to ensure that hierarchical arrays are load balanced and that stencil operations require minimal communication, ZPL's performance model dictates that the second is the better choice for our language. Though users are often initially resistant to think about indexing in this manner, they quickly realize that once regions and directions are named, the numeric indices can be ignored, resulting in extremely readable stencils.

Computation on multiarrays can be done on any level as though it was a normal ZPL array. For example, the following code performs a 4-point stencil at every level of MA:

```
            for i := 0 to 3 do
[MR{i}]         MA{i} = (MA{i}@north{i} + MA{i}@south{i} +
                         MA{i}@east{i}  + MA{i}@west{i})/4;
            end;
```

Similarly, computations between levels are simple, as illustrated in the following loop which performs a restriction operation up the hierarchy:

```
            for i := 0 to 2 do
[MR{i+1}]       MA{i+1} = (MA{i}@north{i} + MA{i}@south{i} +
                           MA{i}@east{i}  + MA{i}@west{i})/4;
            end;
```

A reader might raise the objection that multiregions are superfluous. Why not simply allow the user to declare an array of regions using ZPL's indexed arrays? The main reason we chose not to do so is that regions are not a standard *type* in ZPL, but rather a concept unto itself. Due to its value in analyzing and optimization of ZPL code, we were hesitant to incorporate regions into the type system although this may be relaxed in ZPL's successor language, Advanced ZPL.

The bottom line is that multidirections, -regions, and -arrays help meet many of the goals that we laid out for ourselves in Section 2: regions can be dynamically parameterized through the use of configuration variables; multiarrays fulfill our expectations for hierarchical array support and transparency, provide persistent storage, stencil expressions, and boundary condition specifications (not described here). ZPL's performance model specifies that multiarrays will be compactly allocated, load balanced, and implemented with minimal communication, and that the parallel performance implications will be visible in the code. In the following sections we will describe what is required to make this work.

4 Compiler Issues

The changes required in the ZPL compiler to implement multidirections, multiregions, and multiarrays fall into two main areas: adjusting the internal representation to support these concepts, and updating the compiler's *fluff analysis* pass. The first of these is an organizational task while the second requires a significant improvement to ZPL's compile-time analysis. Both of these modifications will be addressed here.

The ZPL compiler uses an *Abstract Syntax Tree* (AST) to store the user program during compilation, with standard structures for statements, expressions, datatypes, symboltable entries, *etc.*The compiler uses the AST to maintain ZPL's high-level semantics throughout compilation, only *scalarizing* the code during code generation. This is possible due to the fact that our output language is ANSI C; thus traditional scalar optimizations and issues such as register allocation are delegated to the C compiler.

Modifying the AST to support ZPL's multigrid concepts required three changes: First, we added a new dimension field to each symboltable entry to indicate the range of its multidimensional parameterization if it exists (*e.g.*, $1..l$). In addition, we also tagged symboltable entries for multidirections with the expression that is used to scale them. Second, we store any reference to a multiregion or multiarray in a ZPL statement as a traditional array reference with a flag indicating that it is a multiregion or multiarray reference for typechecking purposes. Since multiregions and multiarrays are implemented using a C array of region or array descriptors at runtime, this makes code generation for references to multiregions and multiarrays completely transparent. Finally, we had to modify our means of referring to a direction in a ZPL expression. Previously, directions could be stored simply as a pointer to their symboltable entry since they were just a name. Now that directions can have a multi-indexing expression, they need to be stored as a more general 2-tuple of (symboltable pointer, expression).

In ZPL, *fluff* refers to memory used to cache any nonlocal array values that are required to implement an @ reference. Fluff is allocated contiguously with a processor's local block of memory so that once its values are set up via communication with remote processors, references to those values are transparent, accessing it as though it was local data. This is important because it eliminates special cases when implementing @ references near the boundaries of a processor's local block of data – local and nonlocal data are accessed identically.

The only trick to implementing fluff is knowing how much fluff to allocate. One approach would be to perform dynamic checks to see if sufficient fluff exists, and to resize the processor's local block of data if it doesn't. This results in significant runtime overhead, especially for local variables that may be resized with every access and then destroyed once the subroutine returns. Instead, the ZPL compiler takes a static *whole program analysis* approach in which we examine all the @ references to an array and annotate that array's symboltable pointer with the directions that extend its storage to the maximal extent in each direction. When provided with interprocedural alias analysis information, the running time of this algorithm is $O(n \cdot a)$, where n is the number of @ references in the program and a is the maximum number of aliases that any array variable has. As a result, fluff analysis tends to be linear in the number of @ references used.

Inserting multidirections into the compiler complicates fluff analysis somewhat since the extent to which an array is extended is no longer statically known. Previously, the ZPL compiler could summarize the fluff analysis for each array by storing two directions per array dimension to describe how many extra columns should be allocated in the low and high directions for that dimension. Now that the exact magnitude of a direction is not known (*e.g.*, for a reference A$\{$i$\}$@east$\{$j$\}$, how many columns of fluff need to be allocated?), a more complex scheme is required for fluff analysis since assuming the worst-case value for j could result in exponential amounts of memory being wasted.

Fortunately, since multigrid codes are typically self-similar at every level of computation, @ references that show up in practice are of the form A$\{$i$\}$@east$\{$i+k$\}$, where k is a small constant (typically -1, 0, or 1). These cases can be handled at compile time using symbolic analysis, resulting in no runtime overhead. Cases where the relationship between the multiarray and multidirection are not so clear would have to be handled dynamically (but are currently not handled at all).

Given the expressive power that multiregions and multiarrays add to the language, these changes seemed very reasonable and presented no great technical challenges to implement. The most complex change was the symbolic analysis required to determine the relationship between a multiarray and multidirection in fluff analysis (or a multiregion and multidirection in implicit storage analysis). The most disappointing effect was the requirement for dynamic array resizing in cases where this relationship could not be determined statically.

5 Runtime Implementation

Support for multidirections, multiregions, and multiarrays constituted a small change to ZPL's runtime system. Traditional ZPL directions, regions, and arrays are each represented by a *descriptor* at runtime. For example, directions have the simplest runtime representation, which is simply an integer array.

Regions' descriptors are significantly more interesting. The indices represented by each dimension of a region are stored in three different formats. The first is a 4-tuple format (l, h, s, a) where l is the low bound, h is the high bound, s is the stride, and a is the alignment of the region. These values correspond directly to the region's formal definition [6] and are used to dynamically create new regions in terms of old ones using ZPL's *region operators*. In addition to this, the descriptor stores the global and local bounds for each dimension. Global bounds are used by a processor to determine its place in the large scheme of things. All computation over a region is expressed by generating loops that iterate over a processor's local bounds for the current region.

Array descriptors are used to implement multidimensional arrays since C has no inherent support for them. Thus, each array descriptor contains a pointer to the chunk of memory used to store its values, as well as striding and offset values so that values can be randomly accessed using global indices. ZPL uses a global indexing scheme so that arrays declared over different regions can be accessed using the same index variable, and these accesses are optimized by the compiler to generally be as fast as a local indexing scheme. In addition to fields used to describe the array layout in memory, array descriptors contain a reference to the region descriptor that defines their size and pointers to functions that can be used to read or write array values (particularly useful for arrays of non-scalar types).

The changes required to adapt these structures to work for multidirections, multiregions, and multiarrays was minimal. The ZPL compiler creates setup code for each descriptor, whether static or dynamic. For multi-descriptors, these setup routines were adjusted to allocate a vector of descriptors, and then to loop over the parameter range, setting up each descriptor as a function of the loop index. As described in Section 4, all references to multidirections, regions, or arrays were generated as array accesses, and simply index into these vectors. All of ZPL's runtime support for I/O and interprocessor communication are written to take its runtime descriptors as parameters, and therefore worked transparently with multidirections, regions, and arrays. This served as verification that our approach of using runtime descriptors to represent these objects was excellent software engineering for allowing extensions such as this to occur so effortlessly.

6 Experiments

To evaluate our multigrid support, we compared version 2.3 of the NAS MG benchmark [16] with a ZPL implementation. This benchmark is a 3d V-cycle multigrid application that computes an approximate solution to the discrete Poisson problem. The NAS MG benchmark is hand-coded and written in Fortran with explicit code for message passing using MPI [8]. As such, it is expected to be a fast portable implementation.

The ZPL code is the most faithful translation of the benchmark possible. The computations are the same and are organized in the same procedural style. One major difference is that the Fortran code contains hand-optimized stencil loops that eliminate redundant computation. Computation that is repeated from one iteration in the inner loop to the next iteration is precomputed outside of the inner loop. This optimization is illustrated as follows. Consider the following 2D nine-point stencil:

```
for i := 1 to n do
  for j := 1 to n do
    A[i,j] := A[i-1,j-1] + A[i-1,j] + A[i-1,j+1] +
              A[i,j-1]   + A[i,j]   + A[i,j+1] +
              A[i+1,j-1] + A[i+1,j] + A[i+1,j+1]
```

There are a $8n^2$ additions in the above code segment. This can be decreased to $6n^2$ additions by using a

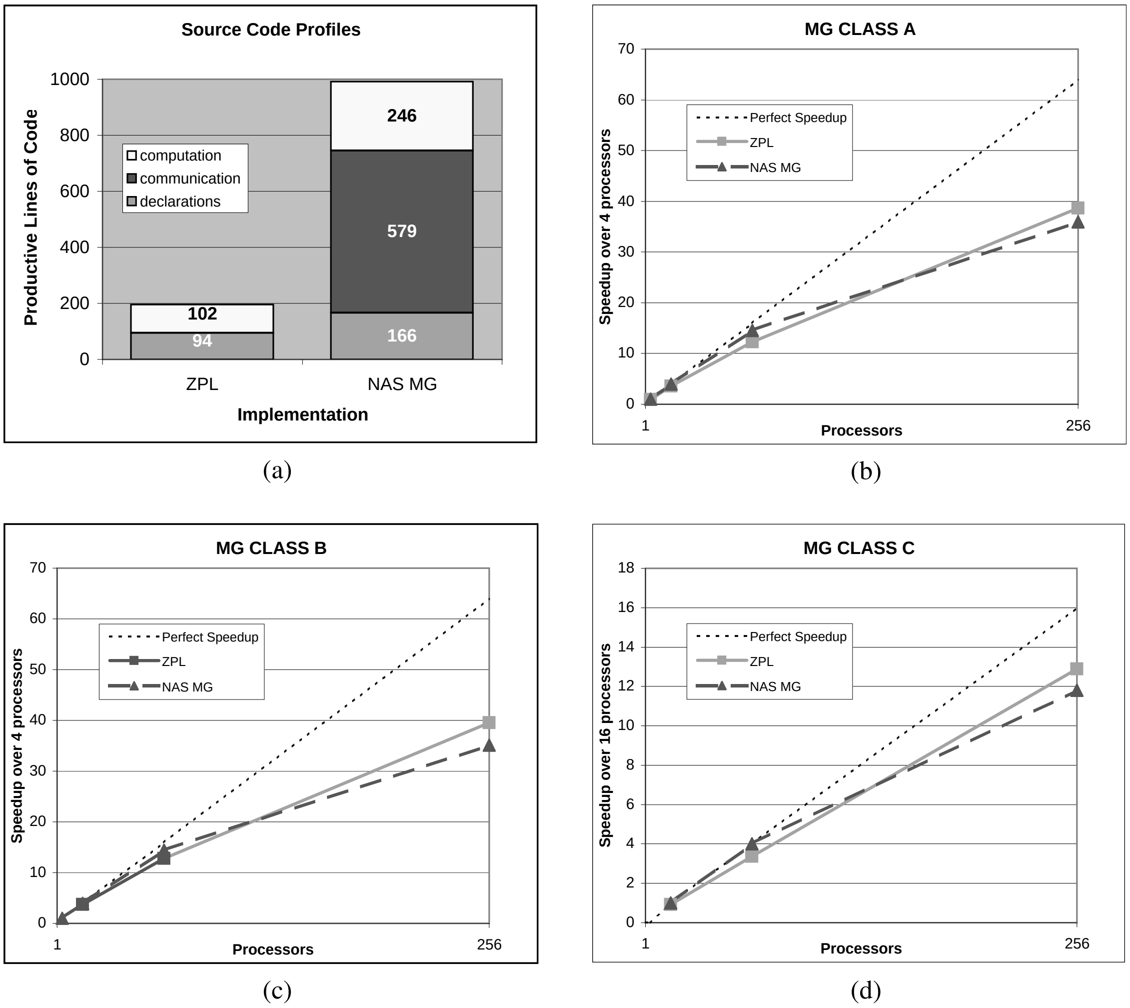

Figure 2: Comparison between ZPL and handcoded versions of the NAS MG benchmark. (a) Difference in the number of productive lines of code (b) Speedup of class A problem size (c) Speedup of class B problem size and (d) Speedup of class C problem size.

temporary one-dimensional array and rewriting the code as follows:

```
for i := 1 to n do
  for j := 0 to n+1 do
    A1[j] := A[i-1,j] + A[i,j] + A[i+1,j]
  for j := 1 to n do
    A[i,j] := A1[j-1] + A1[j] + A1[j+1]
```

Note that the payoff is potentially even greater for a 3d 27-point stencil. Unfortunately, this optimization cannot be coded explicitly in ZPL and is currently not performed by the compiler.

Another difference between the codes is that the ZPL version is problem size and processor set independent, whereas the handcoded version assumes that these values are statically known. This has two implications. First, the NAS code must be recompiled whenever the problem size or the number of processors changes, and second, the NAS code can potentially benefit from optimizations that the ZPL code cannot.

The programs differ significantly in the code length. Figure 2(a) shows the number of lines of code involved in the timed portion of the benchmark. Comments, whitespace and initialization code was removed from both programs. ZPL is more than one-fifth the size of the Fortran code. Note that communication accounts for over half of the Fortran code since the programmer has to manage all the details of data transfer by hand. In addition, computation requires twice as many lines in the Fortran code as in ZPL due to the overhead of setting up local bounds and looping.

The programs were both run on a 400Mhz CRAY T3E with a total of 272 processors, each with 256 MG of memory. We ran three classes of the benchmark, A, B and C, on 4, 16, 64 and 256 processors. There was not enough memory to run classes A or B on 1 processor or class C on less than 8 processors.

Figure 2(b-d) shows the speedups obtained for each class of the problem. In each case speedups are computed using the fastest of the smallest-processor set runs for that class. As can be seen, the performance of the two codes is quite similar, with ZPL scaling better than the Fortran as more processors are added.

Profiling the executions indicates that ZPL is more efficient at communication than Fortran due to its use of the Ironman interface [5]. This interface allows ZPL to take advantage of the SHMEM communication library, which is a faster means of communication on the Cray T3E due to reduced buffering and copying. However, ZPL spends more time than Fortran in computation, due primarily to its lack of the stencil optimization referred to above. This difference explains why ZPL scales better, since the computation becomes a less significant part of the total execution time as more processors are added. We should emphasize that these results were obtained simply by adding multigrid support to the ZPL compiler as it stood — no optimizations specific to multidirections, -regions, or -arrays were added.

From these experiments we conclude that ZPL's multigrid support is competitive with what can be done by hand, but is significantly more concise and cleaner to write since the user doesn't have to hassle with parallel implementation details.

7 Related Work

In addition to ZPL, there are several other modern parallel programming languages in active use and development that should be considered for multigrid application programming. These include High Performance Fortran (HPF) [10], Co-Array Fortran (CAF) [14], and Single-Assignment C (SAC) [17]. Each of these languages has been successfully used to write hierarchical applications [9, 15, 18], and we evaluate each language based on our wishlist of multigrid support from Section 2.

HPF is the most renowned of parallel languages, having received considerable attention throughout this decade. However, the language, as well as current compilers for it, leave much to be desired for multigrid applications. Like ZPL, HPF supports a global view of computation, managing details of communication and data distribution on the user's behalf. However, unlike ZPL, the programmer is given no details either syntactically nor in language definition itself as to how a program will be implemented in parallel [13]. The result is that programmers must tune their HPF program to a specific compiler and/or architecture, perhaps developing their own empirical performance model in the process [9]. Thus, it is impossible to evaluate HPF as a language on the parallel issues such as load balancing, minimal communication, and nonlocal value transparency. Parallel performance implications are clearly not given by the language. On the plus side, HPF supports most of our nonparallel multigrid concepts, which makes sense given that it is based on a sequential language: Hierarchical transparency, compact allocation, persistent storage, and stencil expressions are all supported, for example. It should be noted that HPF's only support for dynamically parameterized hierarchical arrays is via an array of pointers to dynamically allocated data, and that this feature is not yet supported by all implementations, forcing the user to allocate the wasteful $d+1$-dimensional array [9].

Co-Array Fortran is similar to programming in Fortran+MPI in that it requires the programmer to write local per-processor code. However, rather than supporting communication through library calls, it adds *co-arrays* — a concise, elegant extension to Fortran 90 which allows processors to refer to non-local data. The advantage of this approach is that it allows users to express their communication at a semantic level without dealing with details of buffering and setting up communication schedules. In addition, by supporting the syntax at the language level, the CAF compiler is free to optimize communication to a degree that users may be unwilling or unable to do themselves. Moreover, by making nonlocal references explicit in the source code, the user is given a sense of the parallel impact of their code. Since CAF is essentially Fortran-90 plus nonlocal memory access, it is fully capable of expressing all the requirements of a multigrid application. The trouble is that the programmer will have to explicitly handle details such as boundary conditions, data distribution, minimizing communication, etc. Though this is not the end of the world, it leaves much to be desired for scientists who would prefer to concentrate on their algorithms rather than parallel programming.

Single Assignment C is a functional variation on C developed at the University of Kiel. Its extensions provide multidimensional arrays, APL-like operators for dynamically querying array properties, concise specifications of whole-array operations, and functional semantics at procedure calls. SAC supplies a global view of arrays and currently runs only on shared memory computers, allocating the arrays in one contiguous block. This approach makes issues such as minimal communication and nonlocal value transparency moot — all the data is available to all the processors all the time. However, this approach does make parallel performance implications such as cache coherency protocol overhead invisible to the user. SAC's functional semantics make it best-suited for writing multigrid applications in a recursive style, allocating arrays as local variables whose sizes are a function of the function's array arguments. This is an elegant way of expressing multigrid codes such as the NAS MG benchmark, and aggressive compiler optimizations such as inlining and loop unrolling result in competitive execution times [18]. However, the recursive approach does not fulfill our persistent storage requirement as required by many adaptive mesh refinement codes. At this point it remains to be seen whether persistent hierarchical arrays can be expressed in SAC without sacrificing performance.

ZPL's strength over these other languages is that it provides a global view of the computation, yet one in which the parallel performance implications are visible to the programmer at the source level. In addition, ZPL's language features support most of the concepts that were on our wishlist in Section 2.

8 Conclusions

In this paper, we have characterized aspects of a language's semantics and implementation that are important for supporting parallel multigrid applications. Furthermore, we have demonstrated that ZPL's hierarchical features allow it to fulfill most of these requirements with minimal changes to the compiler and runtime system. Our experiments with the NAS MG benchmark show that ZPL is a more concise, readable, and flexible implementation of the code than the handcoded Fortran + MPI version, yet it performs competitively on the Cray T3E, scaling better and outperforming the handcoded benchmark on 256 processors.

The biggest drawback to ZPL's multigrid support is that it currently doesn't have the means to specify or load balance sparse multigrid computations. This is an issue that will be addressed in future work by providing support for sparse regions in Advanced ZPL. We also plan to investigate the stencil optimization discussed in Section 6 as a means of improving our scalar performance without affecting the user or obfuscating the source code.

References

[1] A. Brandt. Multi-level adaptive solutions to boundary value problems. *Mathematics of Computation*, 31(138):333–390, 1977.

[2] W. L. Briggs. *A Multigrid Tutorial*. SIAM, 1987.

[3] Bradford Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. The case for high level parallel programming in zpl. *IEEE Computational Science and Engineering*, 5(3):76–86, July-September 1998.

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

[5] Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. A compiler abstraction for machine independent communication generation. In *Languages and Compilers for Parallel Computing*, pages 261–76. Springer-Verlag, August 1997.

[6] Bradford L. Chamberlain, E Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: An abstraction for expressing array computation. In *ACM SIGAPL/SIGPLAN International Conference on Array Programming Languages*, pages 41–9, August 1999.

[7] W. Davids and G. Turkiyyah. Multigrid preconditioners for unstructured nonlinear 3d finite element models. *Journal of Engineering Mechanics*, 125(2):186–196, 1999.

[8] Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4):169–416, 1994.

[9] 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.

[10] High Performance Fortran Forum. *High Performance Fortran Specification Version 1.1.* November 1994.

[11] R. Leveque and M. Merger. Adaptive mesh refinement for hyperbolic partial differential equations. In *Proceedings of the 3rd International Conference on Hyperbolic Problems*, Uppsala, Sweden, 1990.

[12] C. Lin, L. Snyder, R. Anderson, B. Chamberlain, S. Choi, G. Forman, E. Lewis, and W. D. Weathersby. ZPL vs. HPF: A comparison of performance and programming style. Technical Report 95–11–05, Department of Computer Science and Engineering, University of Washington, 1995.

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

[14] R. W. Numrich and J. K. Reid. Co-array fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, August 1998.

[15] Robert W. Numrich, John Reid, and Kieun Kim. Writing a multigrid solver using co-array fortran. In *Proceedings of the Fourth International Workshop on Applied Parallel Computing*, Umea, Sweden, June 1998.

[16] William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. New implementations and results for the NAS parallel benchmarks 2. In *8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[17] S.-B. Scholz. Single assignment C - functional programming using imperative style. In *Proceedings of IFL '94*, Norwich, UK, 1994.

[18] S.-B. Scholz. A case study: Effects of WITH-loop-folding on the NAS benchmark MG in SAC. In *Proceedings of IFL '98*, London, 1998. Springer-Verlag.

[19] Lawrence Snyder. *The ZPL Programmer's Guide*. MIT Press (in press—available at publication date at ftp://ftp.cs.washington.edu/pub/orca/docs/zpl_guide.ps), 1998.