A Cross-Language Comparison of Support for Parallel Multigrid Applications

Bradford L. Chamberlain Steven Deitz Lawrence Snyder

University of Washington, Seattle, WA 98195-2350 USA

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

Abstract

In this study we perform the first cross-language comparison of support for parallel multigrid applications. The candidate languages include High Performance Fortran, Co-Array Fortran, Single Assignment C, and ZPL. The NAS MG benchmark is used as the basis for comparison. Each language is evaluated not only in terms of its parallel performance, but also for its ability to express the computation clearly and concisely. Our findings demonstrate that the decision of whether to support a local per-processor view of computation or a global view affects a language's expressiveness and performance — the local view approaches tend to achieve the best performance while those with a global view have the best expressiveness. We find that ZPL represents a compelling tradeoff between these two extremes, achieving performance that is competitive with a locally-based approach, yet using syntax that is far more concise and expressive.

1 Introduction

Scientific programmers often make use of *hierarchical arrays* in order to accelerate large-scale *multigrid* computations [4, 3, 11]. These arrays typically have a number of *levels*, each of which contains roughly half as many elements per dimension as the previous level. Computations performed at the coarsest level provide a rough approximation to the overall solution and can quickly be computed due to the relatively small number of array elements. The coarse solution can then be refined at each of the finer levels in order to produce a more precise solution. Hierarchical computations generally take significantly less time than directly computing a solution at the finest level, yet can often be designed so that the accuracy of the solution is only minimally affected. Scientific computations that use hierarchical arrays include multigrid-style applications such as *adaptive mesh refinement* (AMR) [19] and the *fast multipole method* (FMM) [12], as well as algorithms such as the Barnes-Hut algorithm for n-body simulations [2] and wavelet-based compression.

Although hierarchical algorithms are fast compared to direct methods on the finest grid, parallel computation is often used to further accelerate hierarchical computations. Until recent years, parallel language support for hierarchical computation was lacking. In previous work we have described desirable properties for parallel languages that hope to support hierarchical computation [8]. In this paper we continue that work

Language	*Programmer view*	*Data distribution*	*Communication*	*Platforms*
F90+MPI	local	manual	manual	most
HPF	global	undefined	invisible	most
CAF	local	manual	manual++	Cray T3E[1]
SAC	global	defined	invisible	Linux/Solaris
ZPL	global	defined	visible	most

Figure 1: A summary of the parallel characteristics of the languages in this study. *Programmer view* indicates whether the programmer codes at the local per-processor level or at a global level. *Data distribution* indicates whether the data distribution is done manually by the programmer, is defined by the language, or is undefined and left up to the compiler. *Communication* indicates whether communication and synchronization are done manually by the programmer ("++" indicates that the language aids the programmer significantly), or are done by the compiler in a manner visible or invisible to the programmer at the source level.

by evaluating the hierarchical support of four modern parallel programming languages: High Performance Fortran, Co-Array Fortran, Single Assignment C, and ZPL. The languages are compared both in terms of their expressiveness and their parallel performance. Though not technically a language, hand-coded Fortran using MPI is also evaluated, being the *de facto* standard for most scientific parallel computing today. In this study we use the NAS MG benchmark as the basis for comparison due to its widespread use and concise expression of idioms that are fundamental for hierarchical scientific programming. This paper constitutes the first cross-language comparison of parallel language support for hierarchical array computation.

The rest of this paper is laid out as follows: In the next section, we give an overview of the languages being compared, looking at their support for hierarchical applications and for parallel computation in general. In Section 3 we give an introduction to the NAS MG benchmark as well as an evaluation of each language's implementation of it. Section 4 contains our experimental results for the benchmarks. In Section 5 we give an overview of related work, and in Section 6 we conclude.

2 The Languages

In this section, we give a brief introduction to the languages that will form the basis of our study: Fortran+MPI, High Performance Fortran, Co-Array Fortran, Single Assignment C, and ZPL. We selected these languages based on the following criteria: (1) each must support parallel hierarchical programming in some reasonable form; (2) each must be currently available, in use, and supported; (3) each must be readily available to the public. For each language, we give a brief introduction to its philosophy and approach, a sample parallel computation written in it, and a summary of the language's support for hierarchical computation. We pay particular attention to whether each language supports a local view of computation, in which the programmer must manage details of communication and data distribution, or a global view, in which the compiler takes care of these tasks. The results of this section are summarized in Figure 1.

[1] In addition, a subset CAF to OpenMP translator is available.

2.1 Fortran+MPI

Though not strictly a language, programming in Fortran 90 (or C) using the Message Passing Interface (MPI) Library [14] must be mentioned as a viable approach to hierarchical programming due to the fact that it remains the most prevalent approach to parallel computing. MPI was designed to be a completely portable message-passing library, supporting various types of blocking and non-blocking sends and receives. It has been widely supported across diverse parallel architectures and has a well-defined interface that forms a solid foundation on which to build parallel applications. The chief disadvantage to this approach is that the programmer must explicitly manage all the details of the parallel computation since they are writing *local* code that will be run on each processor. In spite of this, MPI's availability, stability, and portability have caused it to enjoy widespread use despite the effort it requires from programmers. Fortran+MPI can be considered the "portable assembly language" of parallel computing, since it provides a portable means for expressing parallel computation, albeit in an often painstaking, low-level, error-prone manner.

As an example F90+MPI computation, consider the following code, designed to assign each interior element of a 1000-element vector `b` with the sum of its neighboring elements in vector `a`:

```
 call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
 call MPI_COMM_RANK(MPI_COMM_WORLD, index, ierr)
 vals_per_proc = (1000/nprocs)+2
 ...
 real a(vals_per_proc), b(vals_per_proc)
 ...
 if (index < nprocs-1) then
   call MPI_SEND(a(vals_per_proc-1), 1, MPI_REAL, index+1, 1,
>       MPI_COMM_WORLD, ierr)
 end
 if (index > 0) then
   call MPI_SEND(a(2), 1, MPI_REAL, index-1, 2,
>       MPI_COMM_WORLD, ierr)
 end
 if (index < nprocs-1) then
   call MPI_RECV(a(1), 1, MPI_REAL, index+1, 1,
>       MPI_COMM_WORLD, ierr)
 end
 if (index > 0) then
   call MPI_RECV(a(vals_per_proc), 1, MPI_REAL, index-1, 2,
>       MPI_COMM_WORLD, ierr)
 end
 b(2:vals_per_proc-1) = a(1:vals_per_proc-2) + a(3:vals_per_proc)
```

This code begins by querying MPI to determine the number of processors being used and the unique index of this instantiation of the program. It then computes the number of values that should be stored on each processor, adding two additional elements to store boundary values from neighboring processors. Next it declares two vectors of floating point values, each with the appropriate number of values for a single processor. The next four conditionals perform the appropriate MPI calls to make each processor exchange

boundary values with its neighbors. Finally the computation itself can be performed as a completely local operation. Note that this code assumes that `nprocs` will divide 1000 evenly and fails to check the MPI calls for error codes. Taking care of these problems would result in a larger, even more general code.

The observation that should be made is that this simple parallel computation is brimming with details that handle data distribution, boundary cases, and the MPI interface itself. Managing these details correctly in a larger program is a huge distraction for scientific programmers who are interested in developing and expressing their science (assuming that they have the expertise to write a correct MPI program in the first place). This is the primary motivation for developing higher-level approaches to parallel programming such as the languages in the following sections.

2.2 High Performance Fortran

High Performance Fortran (HPF) [16] is an extension to Fortran 90 which was developed by the High-Performance Fortran Forum, a coalition of academic and industrial experts. HPF's approach is to support parallel computation through the use of programmer-inserted *compiler directives*. These directives allow users to give hints for array distribution and alignment, loop scheduling, and other details relevant to parallel computation. The hope is that with a minimal amount of effort, programmers can modify existing Fortran codes by inserting directives that would allow HPF compilers to generate an efficient parallel implementation of the program. HPF supports a global view of computation in the source code, managing parallel implementation details such as array distribution and interprocessor communication in a manner that is invisible to the user without the use of compiler feedback or analysis tools.

As a sample parallel HPF computation, consider the same example from above written in HPF:

```
      REAL a(1000), b(1000)
!HPF$ DISTRIBUTE a(BLOCK)
!HPF$ ALIGN b(:) WITH a(:)
      ...
      b(2:999) = a(1:998) + a(3:1000)
```

This code starts by declaring the two vectors of floating point values. It then suggests that vector `a` be distributed across the processor set in a blocked fashion and that vector `b` be aligned with `a` such that identical indices are allocated on the same processor. The computation itself is then specified using a traditional Fortran 90 statement. The compiler is responsible for taking care of the details of distributing `a` and `b` as well as the interprocessor communication required to exchange elements of `a` at processor boundaries. Although this serves as a very concise representation of our parallel computation, the primary disadvantage is that the HPF specification makes no guarantees as to how our directives will be interpreted and implemented. Although this example is simple enough that we can be reasonably confident that our intentions will be carried out, more complicated programs can suffer significant performance degradations from one compiler or architecture to the next as a result of their differing implementation choices [20].

HPF has no specific support for hierarchical programming apart from the array language concepts

supported by Fortran 90. One consequence of this is that in order to specify a hierarchical array in a way that one may iterate over the levels, an array of pointers to dynamically allocated arrays must be used in order to allocate the different number of elements per level. Although this is possible in Fortran 90, it is not always supported by current HPF compilers, forcing users to resort to allocating a 4D array in which each level has a number of elements equal to that of the finest level [15] — an extremely wasteful approach at best.

2.3 Co-Array Fortran

Developed at Cray Research, Co-Array Fortran (CAF) [21] is another extension to Fortran 90 designed for parallel computing. However, unlike HPF, CAF requires users to program at the local view, writing code that will execute on each processor. To express cooperative parallel computation, CAF introduces the notion of a *co-array*. This is simply a variable with a special array dimension in which each element corresponds to a single processor's copy of that variable. Thus, indexing a variable in its co-array dimension specifies a reference to data on a remote processor. This serves as a concise representation of interprocessor communication which is simple and elegant, yet extremely powerful. CAF also provides a number of synchronization operations which are used to keep a consistent global view of the problem. As with HPF, there is some hope that an existing sequential Fortran code can be converted into a parallel CAF code with a minimal amount of work.

Our simple example computation would appear in CAF as follows:

```
nprocs = num_images()
index = this_image()
vals_per_proc = (1000/nprocs)+2
...
real :: a(vals_per_proc)[nprocs], b(vals_per_proc)[nprocs]
...
call sync_all
if (index > 1) then
  a(1) = a(vals_per_proc-1)[index-1]
end
if (index < nprocs) then
  a(vals_per_proc) = a(2)[index+1]
end
b(2:vals_per_proc-1) = a(1:vals_per_proc-2) + a(3:vals_per_proc)
```

This code is similar to our Fortran+MPI code due to the fact that both are local views of the computation. It begins by querying the number of available processors, querying the unique index of this instantiation, and then computing the number of values that should be stored on each processor. Next it declares two co-array vectors, `a` and `b`, each of which has the appropriate number of values on every processor. Next we perform a `sync_all` which serves as a barrier to make sure that the following communication steps do not begin until everyone has finished updating `a` (cheaper synchronization might also be used, but is less concise for an example such as this). The following two conditionals cause each processor to update its

boundary values with the appropriate values from its neighboring processors. Note that unlike F90+MPI, this communication is one-sided, being initiated only by the remote data reference. Finally the computation itself can be performed as a completely local operation.

While CAF's local view has the disadvantage of forcing the user to specify data transfer manually, the syntax is concise and clear, saving much of the headache associated with MPI. Furthermore, co-array references within the code serve as visual indicators of where communication is required. Note that, as in the F90+MPI example, this code assumes that `nprocs` divides the global problem size evenly. If this was not the case, the code would have to be written in a more general style.

As with HPF, CAF does not have any specific support for hierarchical programming. However, since the programmer has much finer control over the low-level programming details and relies much less on the compiler to support data allocation and distribution, it is possible for the programmer to create and manipulate parallel hierarchical arrays manually. They simply must take care of managing all the details.

2.4 Single Assignment C

Single Assignment C (SAC) is a functional variation of ANSI C developed at the University of Kiel [24]. Its extensions to C provide multidimensional arrays, APL-like operators for dynamically querying array properties, *forall*-style statements that concisely express whole-array operations, and functional semantics. The SAC compiler benefits from the reduced data dependences inherent in its functional semantics and aggressively performs inlining and loop unrolling to minimize the number of temporary arrays that would be required by a naive implementation. SAC programs support a global view of array computation and tend to be concise and clean algorithmic specifications. SAC currently runs only on shared-memory machines, so issues such as array distribution and interprocessor communication are invisible to the programmer and somewhat less of an issue than the other languages discussed here.

Our sample computation would take the following form in SAC:

```
a = with ([0] <= x <= [999])
    genarray([1000], (float)(...));
b = with ([1] <= x <= [998])
    modarray(a, x, a[x-[1]] + a[x+[1]]);
```

The first statement generates a new array of floating point values with indices 0–999 whose values are initialized by an arbitrary scalar expression (omitted here). This new array of values is assigned to `a` using a *with-loop* that iterates over indices 0–999. Note that declarations are automatic in SAC — `a`'s size is inferred from the with loop while its type is inferred from the expression used in genarray. The second statement creates a modified version of vector `a` in which each element in the range 1–998 is replaced by the sum of its neighboring values. This is again achieved using a with-loop, assigning the result to `b`. The SAC compiler utilizes a number of worker threads (specified by the user on the command line) to implement each of these with loops, resulting in parallel execution.

While SAC does not support hierarchical arrays as a primitive type, its functional nature makes it natural

to express multigrid codes by writing the solver recursively and declaring each level of the hierarchy as a local array in the recursion, where each dimension is half as big as those of the incoming arrays. While this approach is natural for multigrid solvers like the NAS MG benchmark, it should be noted that it is insufficient for techniques like Adaptive Mesh Refinement (AMR) in which the coarse approximation grids may need to be preserved from one iteration to the next.

2.5 ZPL

ZPL is a parallel programming language developed at the University of Washington [5]. It was designed from first principles rather than as an extension or modification to an existing language under the assumption that it is dangerous to assume that applications and operations designed for sequential computing can effortlessly be transformed into an efficient parallel form. ZPL's fundamental concept is the *region*, a user-defined index set that is used to declare parallel arrays and to specify concurrent execution of array operations. ZPL provides a global view of computation, yet has a syntactically-based performance model that indicates where interprocessor communication is required and the type of communication that is needed [6].

In ZPL, our sample computation would be expressed as follows:

```
region R = [1..1000];
       Int = [2..999];
var A,B:[R] float;
...
[Int] B := A@[-1] + A@[1];
```

The first two lines declare a pair of regions, `R`, which forms the base problem size of 1000 indices, and `Int` which describes the interior indices. In the next line, `R` is used to declare two 1000-element vectors of floating point values, `A` and `B`. The final statement is prefixed by region `Int`, indicating that the array assignment and addition should be performed over the indices 2–999. The @ *operator* is used to shift the two references to `A` by –1 and 1 respectively, thereby referring to neighboring values. ZPL's performance model tells the user that these uses of the @ operator will require point-to-point communication to implement, yet the compiler manages all of the details on the user's behalf.

ZPL provides direct support for hierarchical programming in the form of multi-regions. These are regions whose defining characteristics can be parameterized yielding a series of similar index sets. Thus, to declare a hierarchical array, one could parameterize the defining region's stride and use it to define a multiarray as follows:

```
region MR{0..num_levels} = [1..1000] by [2^{}];
var A{}:[MR{}] float;
```

The first line is a traditional region declaration that is modified to take a parameter range (0–`num_levels`) and to be strided as a function of this parameter, referenced via the empty braces. This results in a series of index sets, each of which is strided by twice that of the previous. The second line creates a series of arrays

Class	*Problem Size*	*Iterations*
A	256^3	4
B	256^3	20
C	512^3	20

Figure 2: Characteristics of the three production grade classes of the MG benchmark. *Problem Size* tells the size of the finest grade in the hierarchy. *Iterations* indicates the number of times that the hierarchy is iterated over.

over these index sets such that each will contain half the data of the previous. Note that this "pyramid" of regions could also be declared such that each was dense but half as big as the previous as follows:

```
region MR{0..num_levels} = [1..(1000/2^{})];
```

Though this is legal and creates index sets with the same numbers of elements, ZPL's performance model indicates that the first approach will result in a computation with significantly less communication overhead and better load balancing. For more details on multiregions and multiarrays, refer to [8, 26].

3 The NAS MG Benchmark

3.1 Overview

Version 2 of the NAS Parallel Benchmark (NPB) suite was designed to evaluate the performance of parallel computers using portable F90+MPI implementations of the version 1 benchmarks [1]. Each benchmark contains a computation that represents the kernel of a realistic scientific computation. The MG benchmark uses a multigrid computation to obtain an approximate solution to a scalar Poisson problem on a discrete 3D grid with periodic boundary conditions. Due to the widespread distribution of the NPB suite, we chose this benchmark as an ideal candidate for comparing the hierarchical support of modern parallel languages.

MG has five main operations and one constraint which pose challenges to parallel implementation. Four of the operations are stencil computations in which a 27-point stencil is used to compute values at an arbitrary level of the grid. Two of these stencils — *resid* and *psinv* — operate at a single level of the grid. The other two — *interp* and *rprj3* — interpolate from a coarse grid to a fine grid and restrict from a fine grid to a coarse grid, respectively. In a parallel implementation, each of these stencils requires point-to-point communication to update a processor's boundary values. Note that this communication may not be nearest-neighbor at coarse levels of the hierarchy. The fifth operation is *norm2u3* which computes approximate L2 and uniform norms for the finest grid. In a parallel implementation, this requires a reduction over the processor set. In addition, an implementation of MG must maintain periodic boundary conditions, which requires additional point-to-point communications.

There are five classes of MG, each of which is characterized by the number of elements in the finest grid and the number of iterations to be performed. Three of the classes — A, B, and C — are production grade problem sizes and their defining parameters are summarized in Figure 2. The other two classes — S

Language	*Author*	*# Processors Known?*	*Problem size known?*	*Data Distribution*
F90+MPI	NAS	yes[2]	yes	3D blocked
HPF	NAS	no	yes	1D blocked
CAF	CAF group	yes[2]	yes	3D blocked
SAC	SAC group	no[3]	yes[4]	1D blocked
ZPL	ZPL group	no	no	2D blocked

Figure 3: Summary of the implementations of MG used in this study. *Author* indicates the origin of the code. The next two columns indicate whether the number of processors and problem size are statically known to the compiler. *Data distribution* indicates the way in which arrays are distributed across processors.

and W — are designed for developing and debugging MG, and therefore are not considered in this study.

3.2 MG Implementations

In the spirit of NPB version 2's decision to measure portability rather than algorithmic cleverness, we attempted to use versions of the MG benchmark which are as true to the original NAS implementation as the source language and compilers would allow. Thus, the benchmarks' data layout, procedural breakdown, and operations are as close to the NAS Fortran+MPI implementation as possible. We summarize the differences between each implementation and the original NAS code below. Within this constraint, we sought to use versions of the benchmark that made reasonable use of each language's features so that our inexperience with a language wouldn't factor into the experiments. For all the languages other than HPF, the benchmarks were written by members of the language's design/implementation team. The HPF version was written by members of NAS who expended considerable effort to make it as efficient as possible [15].

F90+MPI The F90+MPI code that we used was the NAS implementation, version 2.3. It served as the baseline for this study.

HPF The HPF implementation was obtained from NASA Ames [15] and stems from a project to implement the NAS benchmarks in HPF. This implementation was identified by PGI as the best known publicly-available implementation of MG for their compiler (which serves as our experimental HPF compiler) [27]. It follows the F90+MPI implementation of MG very closely, making a few changes that allow the code to be more amenable to HPF's semantics and analysis. Unfortunately, the authors had to make some rather drastic concessions in order to implement it in a manner that was concise and independent of problem size. Chief among these was the fact that they had to allocate arrays of the finest grid at *every* level in the hierarchy — a tremendous waste of memory that will also have consequences on spatial locality. This

[2] Additionally, the number of processors must be a power of two.
[3] Although the *maximum* number of threads must be specified on the compile-time command line.
[4] Though the problem size may be specified dynamically, the code is written such that only a few problem sizes are possible.

required the use of the *HOME* directive in order to align the arrays in a manner that would obtain the desired distribution and load balancing. Other than this issue, which is fundamental to achieving a reasonable implementation of MG, the implementation is extremely true to NAS MG.

CAF The CAF implementation was written using the F90+MPI implementation as a starting point. Since both of these languages use a local per-processor view and Fortran as their base language, the implementation simply involved removing the MPI calls and replacing them with the equivalent co-array syntax. Thus, though solutions more tailored to CAF could be imagined, this implementation is extremely true to the NAS version.

SAC The SAC implementation was part of the SAC distribution [23] and forms the most drastic variation from the NAS implementation. This is primarily due to the fact that SAC's support for hierarchical programming is most easily and efficiently expressed using a recursive approach in which the coarse grids are local array variables within the recursive functions. In contrast, the NAS implementation uses an iterative approach in which the hierarchical arrays are allocated once at the outset of program execution and reused thereafter. In spite of this difference, we used the official SAC implementation of MG rather than implementing an iterative solution for fear that a change in paradigm may not be in the spirit of SAC or in its best interests performance-wise.

ZPL The ZPL implementation of MG was written by mimicking the NAS implementation as carefully as possible. The hierarchical arrays, functional breakdown, and computations all follow the original scheme. The main source of difference is that the Fortran implementation optimizes the stencil computations by precomputing common subexpressions and storing them for re-use by an adjacent stencil. This uses a negligible amount of memory to store the precomputed values, but results in far fewer floating point additions. This optimization is not easily hand-coded in ZPL without wasting memory, and therefore the more direct, but redundant means of expressing the stencils was used.

The table in Figure 3 summarizes the versions of the benchmark that we used for our experiments. Included in this table is information about whether the benchmark fixes the problem size and/or number of processors at compile-time or whether they may be set at runtime. Specifying either piece of information at compile-time gives the compiler the ability to generate code that may be more highly optimized. For example, a code that knows both the problem size and number of processors statically can allocate static arrays and generate loops with fixed bounds. In contrast, if either of these factors are unknown, arrays must be dynamically allocated and loop bounds will remain unknown. Note that these are characteristics of the implementations themselves and that each language could support a version of MG in which neither parameter was known at compile-time. Note that the ZPL benchmark is the most general, with neither parameter supplied to the compiler. The table also gives the data distribution used by each implementation.

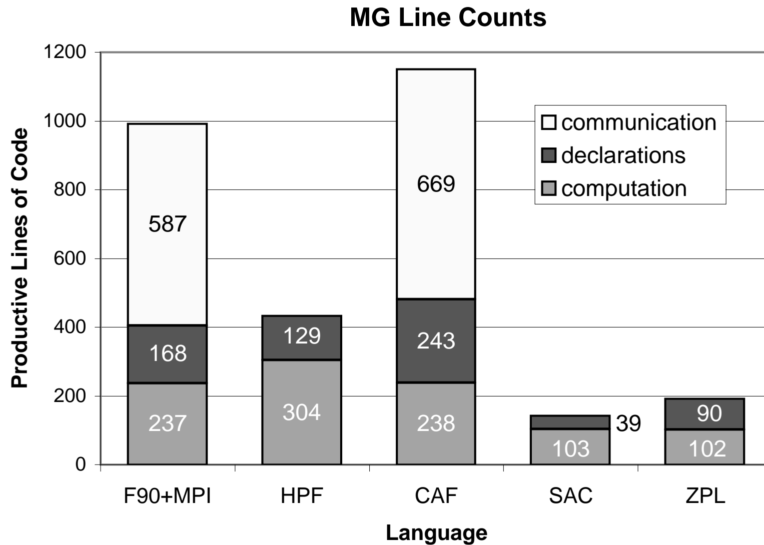


Figure 4: An indication of the number of useful lines of code in each implementation of MG. *Communication* indicates the number of lines devoted to communication and synchronization. *Declarations* indicates code used to declare functions, variables, and constants. *Computation* indicates the number of lines used to express the computation itself.

For F90+MPI, CAF, and HPF, this was chosen by the implementors, whereas SAC and ZPL are both limited by their compiler implementations.

3.3 Evaluation of Expressiveness

To evaluate the expressiveness of each code, we performed both quantitative and qualitative analysis. For the quantitative stage, each benchmark had its lines of code classified as being one of four types: *declarations*, *communication*, *non-essentials*, and *computation*. Declarations include all lines of code that are used for the declarations of functions, variables, constants, and other identifiers. Communication lines are those that are used for synchronization or to transfer data between processors. Code related to comments, initialization, performing timings, and generating output are considered non-essential. The remaining lines of code form the timed, computational kernel of the benchmark and are considered computation.

Figure 4 gives a summary of this classification, showing all non-essential code and how it breaks down into our categories. The first observation that should be made is that communication is responsible for over half the lines of code in F90+MPI and CAF where the user must code using a local view. This puts their line counts at 5 to 6 times that of SAC or ZPL. Inspection of this code reveals that it is not only lengthy, but also quite intricate in order to handle the exceptional cases that are required to maintain a processor's local boundary values in three dimensions. The difference in lines of communication between CAF and F90+MPI are due to MPI's support of higher-level communication calls for broadcasts, reductions, etc. In the CAF version, such operations were encapsulated as functions, adding several lines to its communication

```
      subroutine rprj3( r,m1k,m2k,m3k,s,m1j,m2j,m3j,k )
      implicit none
      include 'mpinpb.h'
      include 'globals.h'

      integer m1k, m2k, m3k, m1j, m2j, m3j,k
      double precision r(m1k,m2k,m3k), s(m1j,m2j,m3j)
      integer j3, j2, j1, i3, i2, i1, d1, d2, d3, j
      double precision x1(m), y1(m), x2,y2

      if(m1k.eq.3)then
        d1 = 2
      else
        d1 = 1
      endif

C TWO CONDITIONALS OF SIMILAR FORM DELETED TO SAVE SPACE

      do  j3=2,m3j-1
         i3 = 2*j3-d3
         do  j2=2,m2j-1
            i2 = 2*j2-d2

            do j1=2,m1j
              i1 = 2*j1-d1
              x1(i1-1) = r(i1-1,i2-1,i3  ) + r(i1-1,i2+1,i3  )
     >                 + r(i1-1,i2,  i3-1) + r(i1-1,i2,  i3+1)
              y1(i1-1) = r(i1-1,i2-1,i3-1) + r(i1-1,i2-1,i3+1)
     >                 + r(i1-1,i2+1,i3-1) + r(i1-1,i2+1,i3+1)
            enddo

            do  j1=2,m1j-1
              i1 = 2*j1-d1
              y2 = r(i1,  i2-1,i3-1) + r(i1,  i2-1,i3+1)
     >           + r(i1,  i2+1,i3-1) + r(i1,  i2+1,i3+1)
              x2 = r(i1,  i2-1,i3  ) + r(i1,  i2+1,i3  )
     >           + r(i1,  i2,  i3-1) + r(i1,  i2,  i3+1)
              s(j1,j2,j3) =
     >               0.5D0 * r(i1,i2,i3)
     >             + 0.25D0 * ( r(i1-1,i2,i3) + r(i1+1,i2,i3) + x2)
     >             + 0.125D0 * ( x1(i1-1) + x1(i1+1) + y2)
     >             + 0.0625D0 * ( y1(i1-1) + y1(i1+1) )
            enddo

         enddo
      enddo

      j = k-1
      call comm3(s,m1j,m2j,m3j,j)

      return
      end
```

(a) F90+MPI/CAF version

```
      extrinsic (HPF) subroutine rprj3(XXXXXXXXXXXXXXXX
     >                                XXXXXXXXXXXXXX
      XXXXXXXX XXXX

      XXXXXXX XXXXXXXXXXX
      XXXXXXX XXXXXXXXX
      XXXXXXX XXX XXX XXX XXX XXX XXX XXX XXX XXX X
      XXXXXX XXXXXXXXX XXXXXX XXXXXX XXXXX
      XXXXXX XXXXXXXXX XXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXX
     >                 XXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXX
     >                 XXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXX
     >                 XXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX

!hpf$   XXXXX XXXXXXXXXXXXXX XXXX XXXXXXXXXXXXXXXXX
!hpf$   XXXXX XXXXXXXXXXXXXX XXXX XXXXXXXXXXXXXXXXXXX
!hpf$   XXXXX XXXXXXXXXXXXXX XXXX XXXXXXXXXXXXXXXXXXX
!hpf$   XXXXX XXXXXXXXXXXXXX XXXX XXXXXXXXXXXXXXXXXXXXX
!hpf$   XXXXX XXXXXXXXXXXXXX XXXX XXXXXXXXXXXXXXXXXXX
!hpf$   XXXXX XXXXXXXXXXXXXX XXXX XXXXXXXXXXXXXXXXXXXXX
!hpf$   XXXXX XXXXXXXXXXXXXX XXXX XXXXXXXXXXXXXXXXXXXXX
!hpf$   XXXXX XXXXXXXXXXXXXX XXXX XXXXXXXXXXXXXXXXXXXXXXX

      XXXXXXXXXXXXXXXX
        XX X X
      XXXX
        XX X X
      XXXXX
C TWO BLOCKS LIKE THE ABOVE DELETED TO SAVE SPACE

!hpf$ XXXXXXXXXXXX XX XXXXXXXXXXXXXXXXXXXX
         XX  XXXXXXXXXX
!hpf$ XXXXXXXXXXX
            XX  XXXXXXXXXX
               XX XXXXXXXXXX
                 XXXXXXXXXXXXXX X XXXXXXXXXXXXXXXXX
     >                          X XXXXXXXXXXXXXXXXXXX
     >                          X XXXXXXXXXXXXXXXXXXX
     >                          X XXXXXXXXXXXXXXXXXXXXX
               XXX XX
            XXX XX
         XXX XX

!hpf$ XXXXXXXXXXXX XX XXXXXXXXXXXXXXXXXXXX
         XX  XXXXXXXXXX
!hpf$ XXXXXXXXXXX
            XX  XXXXXXXXXX
               XX XXXXXXXXXX
                 XXXXXXXXXXXXXX X XXXXXXXXXXXXXXXXXXX
     >                          X XXXXXXXXXXXXXXXXXXXXX
               XXX XX
            XXX XX
         XXX XX

C 2 BLOCKS LIKE THE ABOVE DELETED TO SAVE SPACE

!hpf$ XXXXXXXXXXXX XX XXXXXXXXXXXXXXXXXXXX
         XX  XXXXXXXXXX
!hpf$ XXXXXXXXXXX
            XX  XXXXXXXXXX
               XX XXXXXXXXXX
                 XXXXXXXXXXXXXX X XXXXXXXXXXXXXXXXXXXXX
               XXX XX
            XXX XX
         XXX XX

C 3 BLOCKS LIKE THE ABOVE DELETED TO SAVE SPACE

C DELETED 23 ADDITIONAL LINES HERE TO SAVE SPACE

      return
      end
```

(b) HPF version

```
#define P gen_weights( [ 1d/2d , 1d/4d , 1d/8d  , 1d/16d])

inline double[] gen_weights( double[] wp)
{
  res = with( . <= iv <= . ) {
          off = with( 0*shape(iv) <= ix < shape(iv)) {
                  if( iv[ix] != 1)
                    dist = 1;
                  else
                    dist = 0;
                } fold( +, dist);
        } genarray( SHP, wp[[off]]);
  return( res);
}


inline double weighted_sum( double[] u, int[] x, double[] w)
{
  res = with( 0*shape(w) <= dx < shape(w) )
        fold( +, u[x+dx-1] * w[dx]);
  return(res);
}


double[] fine2coarse( double[] r)
{
  rn = with( 0*shape(r)+1 <= x<= shape(r) / 2 -1)
       genarray( shape(r) / 2 + 1, weighted_sum( r, 2*x, P));
  rn = setup_periodic_border(rn);
  return(rn);
}
```

(c) SAC version

```
procedure rprj3(var S,R: [,,] double; lvl:integer);
begin
  S := 0.5000 * R
    + 0.2500 * (R@dir100{} + R@dir010{} + R@dir001{}
             + R@dirN00{} + R@dir0N0{} + R@dir00N{})
    + 0.1250 * (R@dir110{} + R@dir1N0{} + R@dirN10{} + R@dirNN0{}
             + R@dir101{} + R@dir10N{} + R@dirN01{} + R@dirN0N{}
             + R@dir011{} + R@dir01N{} + R@dir0N1{} + R@dir0NN{})
    + 0.0625 * (R@dir111{} + R@dir11N{} + R@dir1N1{} + R@dir1NN{}
             + R@dirN11{} + R@dirN1N{} + R@dirNN1{} + R@dirNNN{});
  wrap_boundary(S,lvl);
end;
```

(d) ZPL version

Figure 5: The *rprj3* operations from each benchmark, excluding communication for the F90+MPI/CAF version. Note that the HPF implementation is edited down for size and blocked out for the time being due to an outstanding nondisclosure agreement.

Machine	*Location*	*Processors*	*Speed*	*Memory*	*Memory Model*
Cray T3E	ARSC	256	450 MHz	65.5 GB	Dist. Glob. Address Space
Sun Enterprise 5500	UT Austin	14	400 MHz	2 GB	Shared Mem. Multiproc.

Figure 6: A summary of the machines used in these experiments. *Location* indicates the institution that donated the computer time. *Processors* indicates the total number of usable processors to a single user. *Speed* tells the clock speed of the processors. *Memory* gives the total amount of memory available across all processors and *Memory Model* indicates just that.

and declaration counts.

The next thing to note is that the declarations in all of the languages are reasonably similar with the exception of SAC which has the fewest. This is due to SAC's implicit variable declarations, described in the previous section. Thus, SAC's declaration count is limited to simply those lines required to declare functions or the program as a whole.

In terms of computation, one observes that the Fortran-based languages have 2 to 3 times the number of lines as the SAC and ZPL benchmarks. This can be largely attributed to the fact that much of the stencil-based code was written using Fortran 77 loops and indexing rather than Fortran 90's more concise slice notation. This was done by the NAS implementors due to performance issues in Fortran compilers which made a Fortran 90 solution infeasible [1]. It is expected that conversion of these statements to an array-based syntax would bring the computation portion of the linecount closer to that of SAC and ZPL, which benefit from using whole-array statements as their basic unit of operation. HPF's computation linecount is higher than those of its counterparts due to the directives required to achieve the desired load balancing and parallelism.

However, linecounts alone do not make or break a language, so we spent a fair amount of time looking through the codes to see how cleanly they expressed the MG computation. As a representative example, Figure 5 lists the routines that implement the *rprj3* operation in each benchmark. As can be seen, CAF and ZPL require the fewest number of lines, due primarily to their support of whole-array operations, a global view of computation, and the lack of directives required by HPF. In contrast, the Fortran-based codes are complicated by looping structures and local bound computations. Though omitted here for brevity, the communication routines called by the CAF and F90+MPI codes are similarly complex. It is safe to say that the goal of code reuse sought by HPF and CAF are not met in this benchmark — obtaining a good parallel implementation would require quite a bit of work on the programmer's part to instrument and tune a sequential implementation of MG.

Language	*Compiler*	*Version*	*Command-line arguments*	*Comm. Mech.*
Cray T3E Compilers				
F90+MPI	Cray f90	3.2.0.1	–O3	MPI
HPF	PGI pghpf	2.4-4	–O3 –Mautopar –Moverlap=size:1 –Msmp	SHMEM
CAF	Cray f90	3.2.0.1	–O3 –X 1 –Z *nprocs*	E-regs
ZPL	U. Wash zc	1.15a		SHMEM
	Cray cc	6.2.0.1	–O3	or MPI
Sun Enterprise 5500 Compilers				
F90+MPI	SUNW f90	2.0	–O3	MPI
SAC	U. Kiel sac2c	0.8.0	–mtdynamic 14	shared
	GNU gcc	2.95.1	–O1	memory
ZPL	U. Wash zc	1.15a		MPI
	SUNW cc	5.0	–fast	

Figure 7: A summary of the compilers used in these experiments. The compiler, version number, command-line arguments used are given for each. In addition, the communication mechanism used by the compiler is noted.

4 Performance Evaluation

4.1 Methodology

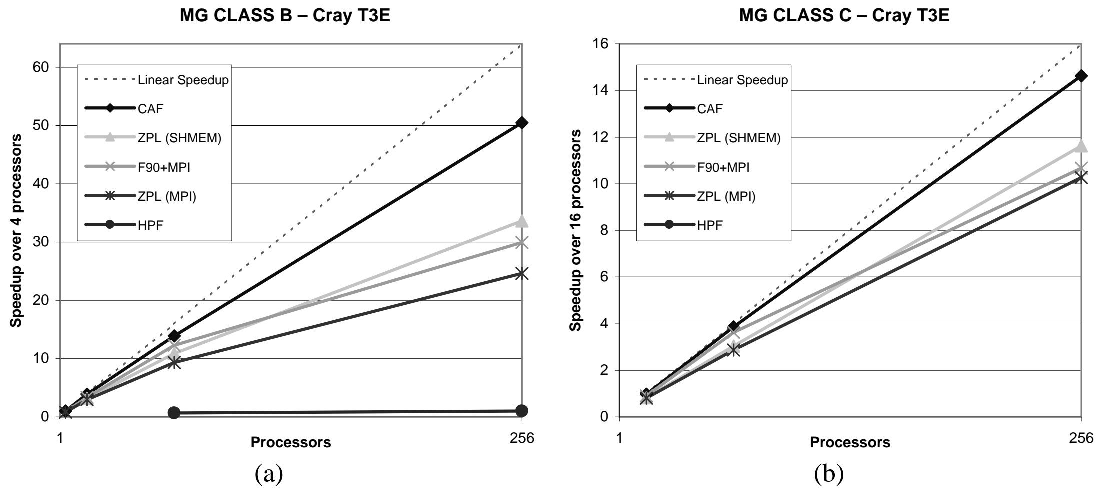
To evaluate performance, we ran experiments on two hardware platforms, the Cray T3E and the Sun Enterprise 5500. No single platform supports all languages since CAF currently only runs on the Cray T3E which is not supported by SAC. In addition, we did not have access to an HPF compiler on the Enterprise 5500. The machines are of vastly different sizes — the Cray has 256 processors while the Enterprise 5500 only has 14. Other relevant details are summarized in Figure 6.

Figure 7 summarizes information about the compilers and command-line flags that were used to compile the benchmarks. In all cases we used the highest degree of single-flag optimizations that was both available and worked. For the HPF compiler, we used the default flags that came with the NAS code, changing –Mmpi to –Msmp in order to take advantage of the Cray's optimized SHMEM communication library. Both the SAC and ZPL compilers compile to ANSI C, so we also give information about the C compilers and flags used by their back ends. With the SAC compiler we found that using optimization levels greater than –O1 caused the multithreaded version of the code to crash. [5]

4.2 Performance Results

The graphs in Figure 8 give speedup results for class B and C runs of F90+MPI, HPF, CAF, and ZPL on the Cray T3E (class A was virtually identical to class B and therefore omitted to save space). ZPL can compile to a variety of communication mechanisms including the Cray's SHMEM interface and the MPI standard [7], so both were used for these experiments. The SHMEM interface has a lower overhead due to

[5] This has apparently been fixed for the next release, and we hope to obtain a copy for the final version of this paper for fairer comparison.



(a) (b)

Figure 8: Speedup curves for the T3E. Note that There was not enough memory to run the smaller processor set sizes for these classes. The fastest running time on the smallest number of processors is used as the basis for computing speedup for all benchmarks. Note that the class C HPF code was unable to run due to its memory requirements.

reduced buffering and synchronization requirements and is always preferable to MPI, so the MPI numbers are included for comparison to F90+MPI.

There was insufficient memory on the Cray T3E to run these classes on a single processor, so all speedup calculations are with respect to the fastest 4-node time for class B and 16-node time for class C. Thus, the best possible speedups were $64\times$ for class B and $16\times$ for class C. Due to the HPF implementation's excessive memory requirements (Section 3.2), it could not run class B with fewer than 16 processors and could not run class C at all.

CAF performed the best on all three problem sizes, achieving speedups of 50.4 on class B and 14.6 on class C. The ZPL and F90+MPI implementations clustered close to one other significantly below it. HPF performed poorly, barley achieving a speedup of 1 on 256 processors, no doubt due to its poor use of memory. While it has been demonstrated that HPF programs can benefit from tuning for each particular architecture and/or compiler [20], we decided not to pursue this avenue due to (1) our assumption that a good implementation on PGI's Origin 2000 compiler should remain a good implementation on PGI's Cray T3E compiler; and (2) a belief that until a true hierarchical array can be allocated with the PGI compiler, performance will remain significantly lacking.

Profiling demonstrated that two primary factors contributed to the differences between the CAF, ZPL, and F90+MPI versions: the different communication mechanisms used by each compiler, and the hand-coded stencil optimization described in Section 3.2. As described above, MPI tends to perform significantly worse than SHMEM on the T3E and so this puts the SHMEM ZPL times ahead of the ZPL MPI version and the F90+MPI implementation. The CAF compiler uses an even lower-level communication mechanism, writing code that directly uses GET and PUT instructions on the Cray's E registers. Though the SHMEM

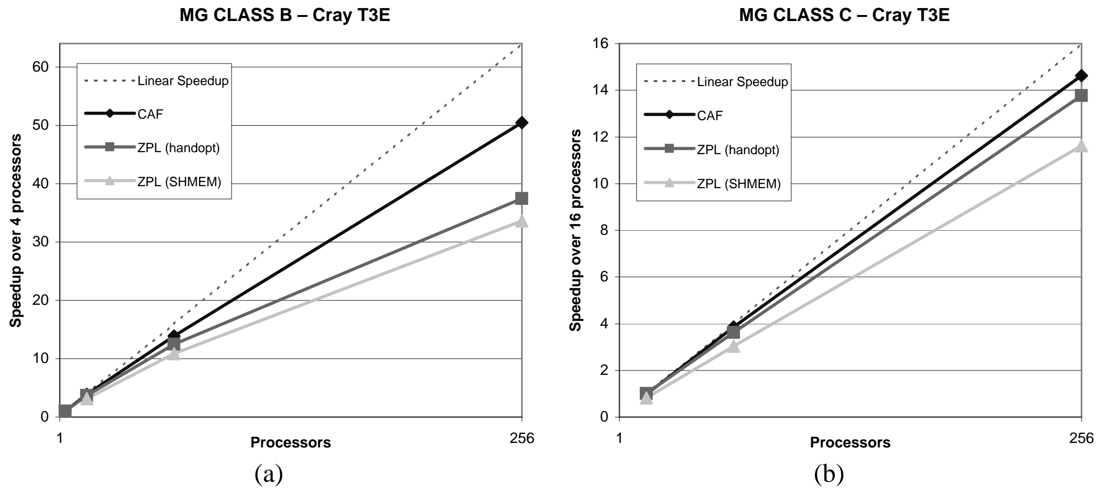


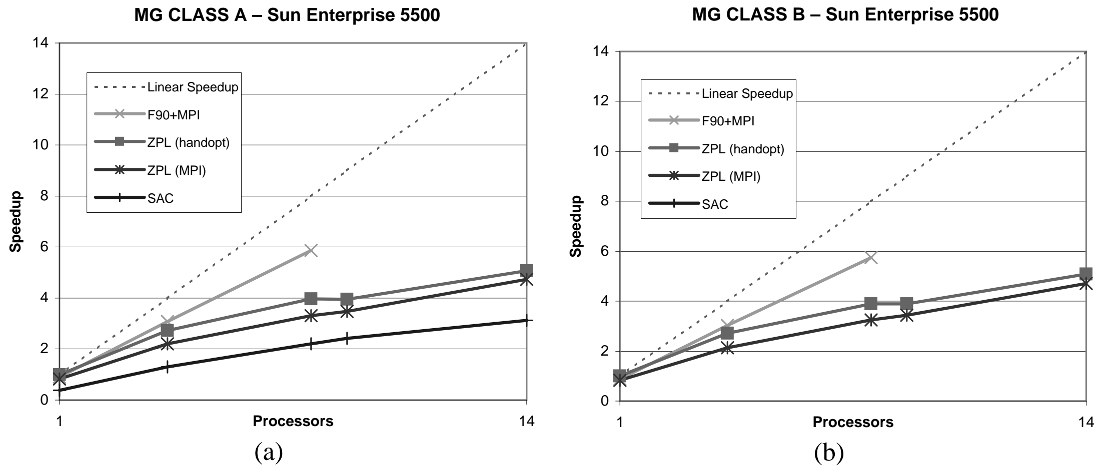
Figure 9: Additional speedup curves for the T3E demonstrating how much the SHMEM ZPL version can gain by eliminating redundant computation in stencil-based codes.

interface is described as being thin, calling into it incurs some amount of overhead.

The hand-coded stencil optimization is implemented in the CAF and F90+MPI codes, but not the ZPL codes. Profiling shows that for the class C problem size, roughly half of the execution time is spent in *resid* alone, thus Amdahl's law indicates that this optimization has the potential to make a significant difference. To determine the impact of this optimization, we implemented it by hand in the C code generated by the ZPL compiler and re-ran the SHMEM version of the benchmark. Results are shown in Figure 9. This small change allows ZPL to almost catch up to CAF for the class C problem size and allows it to get a bit closer in class B where computation is not dominating communication as heavily. We are currently developing this stencil optimization in the ZPL compiler due to the frequent use of stencils in ZPL applications.

On the Sun Enterprise 5500, the smaller number of processors makes the results less interesting, particularly since the F90+MPI implementation can only run on processor sets that are powers of two. Memory limitations only allow the class A and B problem sizes to be run. On this platform, F90+MPI significantly beats the MPI version of ZPL with or without the hand-coded optimization. This is most likely due to the relative quality of code produced by the Fortran and C compilers on this platform as well as the differing surface-to-volume characteristics induced by the benchmarks' allocation schemes. In coming weeks we will be performing more analysis of these numbers.

Note that SAC lags slightly behind F90+MPI and ZPL on the class A problem size. This is certainly due in part to the bug that forced it to be compiled using –O1; we saw nearly a 50% improvement in the single processor running times when SAC was compiled with –O3. In addition, a memory leak was discovered in the class B problem size which prevented it from completing at all. No doubt this memory leak also impacted the performance of the class A runs. We anticipate a fixed version of the compiler in the near future to update these numbers.



(a) (b)

Figure 10: Speedup curves for benchmarks on the Sun Enterprise 5500. Note that the F90+MPI version is unable to run on more than 8 processors due to its requirement that a power of two processors be used.

5 Related Work

For each of the languages in this paper, work has been done on expressing and optimizing hierarchical computations in that language [1, 15, 22, 25, 8]. Each of these papers studies the language independently, however, as opposed to a cross-language comparison as we have done here.

The other main approach to parallel hierarchical programming is to use libraries that support efficient routines for array creation and manipulation in place of a programming language. Most notable among these projects is KeLP [13], a C++ class library that not only supports dense multigrid computations such as MG, but also adaptive hierarchical applications where only a subset of cells are refined at each level. Future work should consider the expressive and performance tradeoffs for using a library rather than a language in hierarchical applications.

POOMA (Parallel Object-Oriented Methods and Applications) [18] is a template-based C++ class library for large-scale scientific computing. Though POOMA does not directly support multigrid computations, the *domain* abstraction can be used to specify operations on strided subsets of dense arrays which can then interact with (smaller) arrays that conform with the domain.

OpenMP [10] is a standard API that supports development of portable shared memory parallel programs and is rapidly gaining acceptance in the parallel computing community. In recent work, OpenMP has been used to implement irregular codes [17], and future studies should evaluate its suitability for hierarchical multigrid-style problems.

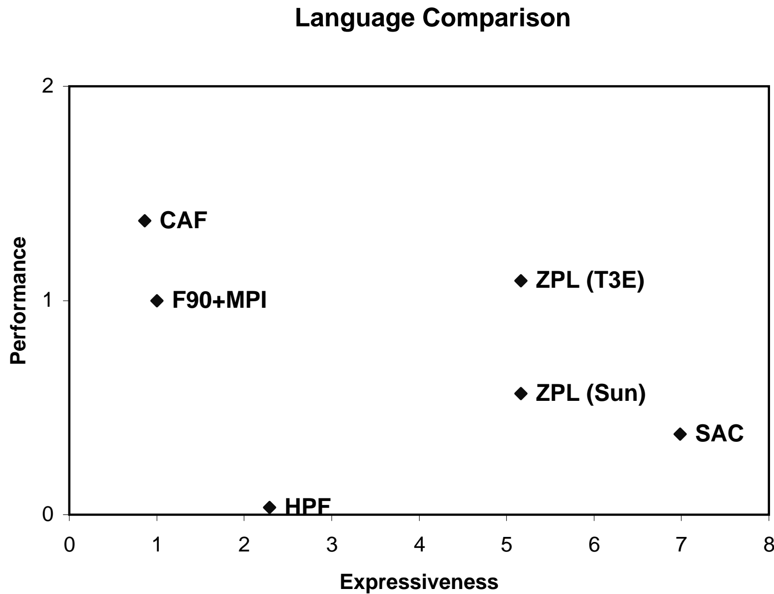


Figure 11: An interpretation of this paper's experiments by plotting performance vs. expressiveness. *Performance* is summarized for each benchmark using the running time of the largest problem size on the largest number of processors since parallel performance for large applications is the goal. *Expressiveness* is measured in total lines of productive code. Both numbers are normalized to the F90+MPI implementation of MG such that higher numbers imply better performance and expressiveness.

6 Conclusions

We have undertaken a study of parallel language support for hierarchical programming using the NAS MG benchmark. We have made an effort to study languages that are available and in use on current parallel computers, and to choose implementations, compiler settings, and hardware platforms that would benefit these languages.

To summarize the results of our study, Figure 11 gives a rough map of expressiveness vs. performance as determined by our experiments. Since our qualitative analysis of the benchmarks indicated an inverse correlation between line counts and a clean expression of the MG benchmark, we use the total number of productive lines of code as the measure of a benchmark's expressiveness. Since parallel performance on large problems is the ultimate goal of this work, performance is judged using the execution time of the largest problem size using the largest number of processors for each language. Both numbers are normalized to the NAS F90+MPI benchmark in such a way that higher numbers indicate better expressiveness/performance.

The first observation to make is that expressiveness varies far more than performance across the languages, with ZPL and SAC being the most concise benchmarks due to their array operators and global scope. Our qualitative analysis of these codes shows that though concise, they are still quite readable, and

form clear descriptions of the algorithms involved. While CAF and F90+MPI could be improved somewhat by a more widespread use of F90 syntax, the amount of code devoted to communication in each constitutes a significant amount of overhead, not only in lines of code, but also in coding and distracting from the heart of the algorithm at hand.

In terms of performance, CAF is the fastest due to a combination of its fast communication mechanism, hand-coded stencil optimizations, its use of Fortran as a base language, and its 3D data decomposition. ZPL rivals F90+MPI on the T3E, but lags behind on the Sun for reasons yet to be determined. SAC and HPF suffer from compiler bugs and unimplemented language features and should both see improvement in the future.

Looking at performance and expressiveness in combination, we see that CAF and F90+MPI demonstrate that great performance is achievable through hard work on the part of the programmer. ZPL represents an interesting data point in that the expressiveness provided by its global view of the problem does not compromise its performance, which rivals that of the local-view languages. ZPL also represents the only benchmark that used neither the problem size nor processor set size at compile time, and which demonstrated portable performance. Though this may seem like a small detail, anyone who has spent considerable amount of time developing, tuning, and running codes on various platforms with varying problem sizes and numbers of processors knows that such factors can be a huge convenience, especially when performance is maintained.

In future work, we plan to continue our study of language support for hierarchical applications, moving towards larger and more realistic applications such as AMR and FMM that are not dense at every level of the hierarchy [19, 12]. Our approach will be based on extending ZPL's region concept to allow for *sparse regions* [9]. We are also in the process of implementing the stencil-based optimization described in Section 3.2 in order to give the user the expressive power of writing stencils naturally while achieving the same performance as a hand-coded scalar implementation.

Acknowledgments. The authors would like to thank Sven-Bodo Scholz, Clemens Grelck, and Alan Wallcraft for their help in writing the versions of MG used in this paper and for answering questions about their respective languages. We'd also like to thank the technical reporting service at PGI and the NAS researchers who helped us obtain the HPF implementation. This work was done using a grant of supercomputer time from the Alaska Region Supercomputer Center and the University of Texas at Austin, for which we are extremely grateful. Finally, Sung-Eun Choi, E Christopher Lewis, and Ton A. Ngo must be thanked for their invaluable help in helping inspire and design the ZPL language features which led to this work.

References

[1] 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.

[2] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] Bradford L. Chamberlain, Steven Deitz, and Lawrence Snyder. Parallel language support for multigrid algorithms. In submitted to *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, November 1999.

[9] 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.

[10] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1), January/March 1998.

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

[12] 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.

[13] S. J. Fink, S. R. Kohn, and S. B. Baden. Efficient run-time support for irregular block-structured applications. *Journal of Parallel and Distributed Computing*, 50:61–82, May 1998.

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

[15] 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.

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

[17] Dixie Hisley, Gagan Agrawal, Punyam Satya-narayana, and Lori Pollock. Porting and performance evaluation of irregular codes using openmp. In *Proceedings of the First European Workshop on OpenMP*, September/October 1999.

[18] Steve Karmesin, James Crotinger, Julian Cummings, Scott Haney, William Humphrey, John Reynders, Stephen Smith, and Timothy J. Williams. Array Design and Expression Evaluation in POOMA II. In D. Caromel, R.R. Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 231–238. Springer-Verlag, 1998.

[19] 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.

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

[21] 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.

[22] 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.

[23] University of Kiel. *SAC website*. http://www.informatik.uni-kiel.de/~sacbase/ (Current December 11, 1999).

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

[25] 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.

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

[27] Portland Group tecnical reporting service. *Personal communication.* October 1999.