A Comparative Study of the NAS MG Benchmark across Parallel Languages and Architectures*

Bradford L. Chamberlain Steven J. Deitz Lawrence Snyder

University of Washington, Seattle, WA 98195-2350 USA
{brad,deitz,snyder}@cs.washington.edu

Abstract

Hierarchical algorithms such as multigrid applications form an important cornerstone for scientific computing. In this study, we take a first step toward evaluating parallel language support for hierarchical applications by comparing implementations of the NAS MG benchmark in several parallel programming languages: Co-Array Fortran, High Performance Fortran, Single Assignment C, and ZPL. We evaluate each language in terms of its portability, its performance, and its ability to express the algorithm clearly and concisely. Experimental platforms include the Cray T3E, IBM SP, SGI Origin, Sun Enterprise 5500, and a high-performance Linux cluster. Our findings indicate that while it is possible to achieve good portability, performance, and expressiveness, most languages currently fall short in at least one of these areas. We find a strong correlation between expressiveness and a language's support for a global view of computation, and we identify key factors for achieving portable performance in multigrid applications.

1 Introduction

Hierarchical algorithms such as the *multigrid method* lie at the core of many large-scale scientific computations [5, 4, 14]. These algorithms accelerate the solution of large discrete problems by solving a coarse approximation of the original problem and then refining that solution until it forms a sufficiently precise answer to the original problem. Our long-term goal in this work is to identify important abstractions and optimizations for parallel languages that wish to support a wide variety of hierarchical techniques, including *adaptive mesh refinement* (AMR) [25] and the *fast multipole method* (FMM) [16]. This study constitutes a first step toward that goal by evaluating the support that current parallel programming languages and compilers offer for the NAS MG benchmark.

Though NAS MG contains simplifications that may not reflect the assumptions of more complex multigrid solvers, its authors explain that they "chose it for its portability and simplicity, and expect that a supercomputer which can run it effectively will also be able to run more complex multigrid problems" [3]. In this work, we adopt a similar philosophy, assuming that in order to support complex adaptive hierarchical algorithms, a parallel language must first demonstrate success with simpler, more regular examples.

To this end, we sought out implementations of NAS MG written in various parallel programming languages and studied them on a number of diverse parallel architectures. The languages represented in this study are Co-Array Fortran [28], High Performance Fortran [21], Single Assignment C [31], and ZPL [6]. Though not technically a

*This research was supported by a scholarship from the USENIX Association and NSF Grant GER–9454434. The results were made possible thanks to grants of HPC time from the Arctic Region Supercomputing Center, the Maui High Performance Computing Center, and Los Alamos National Laboratory.

parallel language, we also consider Fortran 90+MPI [26] since MPI is currently the *de facto* standard for scientific parallel programming. Our parallel platforms include the Cray T3E, Sun Enterprise 5500, SGI Origin, IBM SP, and a high-performance Linux cluster.

Each implementation of NAS MG was evaluated in terms of its performance, portability, and expressiveness. The evaluation of performance is obvious since it is often considered the bottom line in parallel programming. Portability constitutes another important factor due to the wide variety of parallel architectures in use today as well as the rate at which their characteristics evolve. In this study, "expressiveness" describes the degree to which a language allows scientific programmers to express their algorithms in a straightforward manner. Though often underrated in the parallel computer science community, expressiveness is essential if scientists who are not well-versed in the intricacies of parallel programming are to enjoy its benefits. Note that we form our evaluation using a specific set of compilers and NAS MG implementations, and therefore may fail to capture the maximum potential of each language. With this in mind, we have taken pains to make our study consider each language as fairly as possible using current technology.

This paper constitutes the first cross-language, cross-platform study of a parallel multigrid algorithm. Previous studies have typically concentrated on the performance of a single language on a single platform [18, 29, 32]. As such, this work should be of interest to the supercomputing community as a whole, surveying the state-of-the-art in compiling and running a multigrid application using a variety of languages and architectures. In addition, developers of parallel languages and compilers can use our results to identify key features for producing expressive and portable multigrid applications that perform well. This paper also presents the first published performance results of the NAS MG benchmark for the CAF and ZPL parallel programming languages.

This paper is organized as follows: In the next section, we give an overview of the languages being compared, emphasizing their support for hierarchical applications. In Section 3 we give an introduction to the NAS MG benchmark as well as a summary of each language's implementation of it. Sections 4 and 5 contain our evaluation of the implementations' expressiveness, performance, and portability. In Section 6 we summarize related work, and in Section 7 we offer some conclusions.

2 The Languages

In this section, we give a brief introduction to the languages that form the basis of our study: Fortran 90+MPI, High Performance Fortran, Co-Array Fortran, Single Assignment C, and ZPL. Inclusion in the study was based on the following criteria: each language had to (1) support parallel hierarchical programming in a reasonable form; (2) be currently available, in use, and supported; and (3) be readily available to the public.

We give a brief introduction to the philosophy of each language and give a concrete sense of what the language entails by showing a simple parallel computation written in it. We also describe each language's support for *hierarchical arrays*, the fundamental data structure of multigrid computations. Hierarchical arrays are composed of multiple levels, each of which typically has half as many elements per dimension as the previous.

This section also makes an important distinction between whether a language supports a local or global view of parallel computation. *Local-view* languages are those that require programmers to write a per-processor SPMD code, explicitly managing details of communication and data distribution. In contrast, *global-view* languages allow programmers to express their computations globally, relying on the compiler to handle the details of its parallel implementation. A brief summary of the languages described in this section is given in Table 1.

Language	*Programmer view*	*Data distribution*	*Communication*	*Platforms*
F90+MPI	local	manual	manual	most
HPF	global	directive-based	invisible	most
CAF	local	manual	manual++	Cray T3E
SAC	global	automatic	invisible	Linux/Solaris SMPs
ZPL	global	stipulated	visible	most

Table 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 with a global view. *Data distribution* indicates whether programmers specify and manage data distribution manually, provide the compiler with suggestions via directives, rely on the compiler to automatically manage the distribution, or stipulate a distribution for the runtime to manage. *Communication* indicates whether communication and synchronization are performed manually by the programmer or, when managed by the compiler, whether it is visible or invisible to the programmer at the source level ("manual++" indicates that the language aids the programmer significantly). The last column indicates the hardware platforms on which each language is supported.

2.1 Fortran 90+MPI

Though not strictly a language, programming in Fortran 90 (or C) using the Message Passing Interface (MPI) library [26] 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 programming using MPI is that programmers must explicitly manage all the details of their parallel computations since they are writing local-view code that will be run on each processor. In spite of this programmer effort, MPI's availability, stability, and portability have caused it to enjoy widespread use. MPI is often 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 a sample Fortran 90+MPI (F90+MPI) computation, consider the following code, designed to replace 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 > 0) then
  call MPI_RECV(a(1), 1, MPI_REAL, index-1, 1, MPI_COMM_WORLD, ierr)
end
if (index < nprocs-1) 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.

It should be noted that this simple parallel computation is brimming with details that manage data distribution, boundary cases, and the MPI interface itself. Handling these details correctly in a larger program is a significant distraction for scientific programmers who are primarily interested in developing and expressing their algorithms (assuming 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 described in the following sections.

While Fortran 90 has no implicit support for hierarchical arrays, the fact that programmers are writing at the local-view allows them to create and manipulate parallel hierarchical arrays manually. They simply must take care of managing all the related details. In the F90+MPI version of NAS MG, the implementors accomplished this by allocating a 1D array per hierarchical array with enough memory to store all the elements in the hierarchy. A subrange of the array can then be sent into the kernel routines which interpret it as a 3D array of the appropriate size.

2.2 High Performance Fortran

High Performance Fortran (HPF) [21, 38] 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 program from the previous section 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 two 1000-element vectors of floating point values. It then suggests that vector `a` be distributed across the processors 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 inserting 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 degradation from one compiler or architecture to the next as a result of their differing implementation choices [27].

HPF has no specific support for hierarchical arrays apart from Fortran 90's array language concepts. One consequence of this is that in order to specify a hierarchical array at the global view such that one may iterate over its levels and parallelize it effectively, an array of pointers to dynamically allocated arrays must be used [18]. Although this is possible in Fortran 90, it is not currently supported by all HPF compilers, forcing users to resort to allocating a 4D array in which the additional dimension represents the levels of the hierarchy [18]. This has the effect of allocating a number of elements at each level equal to that of the finest — an extremely wasteful approach at best.

2.3 Co-Array Fortran

Developed at Cray Research, Co-Array Fortran (CAF) [28] 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 sample 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 F90+MPI example due to the fact that both are Fortran-based 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 subsequent communication steps do not begin until every processor has finished updating `a` (cheaper synchronization might also be used, but is less concise for an example such as this). The subsequent 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 in F90+MPI, CAF's local-view allows programmers to create and manage their parallel hierarchical arrays manually.

2.4 Single Assignment C

Single Assignment C (SAC) is a functional variation of ANSI C developed at the University of Kiel [31]. 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. It also 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 perhaps 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 provided with `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 with-loop, resulting in parallel execution.

SAC's functional nature makes it natural to express hierarchical applications using a recursive approach. This allows programmers to implement each level of a hierarchal array using local arrays whose dimensions are each half as big as 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 often 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 [6]. It was designed from first principles rather than by extending or modifying an existing language based on the assumption that traditional sequential applications and operations cannot 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 syntax-based performance model that indicates where interprocessor communication is required, as well as the type of communication that is needed [7].

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.

Hierarchical arrays can be expressed in ZPL using its concept of *multi-regions*. These are regions whose indices can be parameterized to yield a series of related index sets. For example, the following line declares a set of increasingly small regions:

```
region MR{0..k} = [1+{} .. 1000-{}];
```

This declaration creates k+1 regions, each of which has its upper and lower bounds adjusted by its index. In this example, MR{0} is equivalent to the region R above, MR{1} is equivalent to Int, and MR{5} would represent [6..995]. ZPL programmers can use multi-regions to create hierarchical arrays simply by adjusting a region's stride or upper bound by powers of two. For example, the following declarations create two sets of hierarchical regions and arrays:

```
region HierReg1{0..num_levels} = [1..1000] by [2^{}]; -- strides of 1, 2, 4, etc.
       HierReg2{0..num_levels} = [1..1000/(2^{})];    -- [1..1000], [1..500], etc.
var HierArr1{}:[HierReg1{}] float;
    HierArr2{}:[HierReg2{}] float;
```

Although both of these approaches are legal and each results in a hierarchical array with the same number of elements, ZPL's performance model indicates that the first approach is likely to result in better load balancing and significantly less communication, making it preferable. For further details on multi-regions, refer to [9, 33].

3 The NAS MG Benchmark

Version 2 of the NAS Parallel Benchmark (NPB) suite [1] was designed to evaluate the performance of parallel computers using hand-coded F90+MPI implementations of the version 1 benchmarks. 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. The prevalence of the NAS MG benchmark makes it an ideal application for our initial study of parallel language support for hierarchical applications.

The bulk of the work in MG is done in four routines, each of which is implemented using one or more 27-point stencils. Two of these stencils — *resid* and *psinv* — operate at a single level of the hierarchy, whereas the other

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

Table 2: Characteristics of the three production grade classes of the MG benchmark. *Problem Size* gives the number of elements at the finest level of the hierarchy. *Iterations* indicates the number of times that the hierarchy is traversed.

two — *interp* and *rprj3* — interpolate and project between adjacent levels of the hierarchy, respectively. Parallel implementations of these stencils require point-to-point communication to update every processor's boundary values for each dimension that is distributed. Note that this communication may be with more "distant" processors at coarser levels of the hierarchy. In addition, the benchmark requires periodic boundary conditions to be maintained, and for global reductions to be performed over the finest grid.

There are five classes of MG, each of which is characterized by the number of elements in its finest grid and the number of iterations to be performed. Two of the classes — S and W — are designed for developing and debugging MG, and are therefore not considered in this study. The other three classes — A, B, and C — are production grade problem sizes and their defining parameters are summarized in Table 2. Note that A and B use the same problem size, but a different number of iterations.

3.1 Finding Implementations

We required two things while searching for candidate implementations of NAS MG. The first was motivated by the emphasis in NPB version 2 on portability rather than algorithmic cleverness [1]. In particular, we required each implementation to follow the spirit of the NAS implementation as closely as the source language and compiler would allow. For example, an implementation could not use a sparse array to store the twenty nonzero values in the $n \times n \times n$ input array, as this would constitute a radical departure from the original algorithm. However, it could certainly change loop nest iteration orders, strive to minimize communication, etc. All of the implementations used in this study met this requirement without modification.

Our second requirement was that each benchmark's programmer be familiar with the language and compiler in question. Our goal was to ensure that our own ignorance of a language or compiler would not adversely affect that language's evaluation. This requirement was easy to meet due to the widespread study of NAS MG: all of the implementations in this study were coded by the language's development team with the exception of the HPF version which was written at NAS.

As in any programming context, there is a potential tension between expressiveness and performance. In general, the implementors strove to optimize for performance without unduly compromising expressiveness. For example, the F90+MPI implementation of NAS MG often relies on Fortran 77 loops and indexing rather than the equivalent (and arguably more expressive) Fortran 90 array statements. This decision was motivated by the desire to ensure that the crucial stencil code would be implemented efficiently rather than relying on each platform's compiler to get it right [37, 1]. Our efforts to make the code more expressive using array statements proved to adversely affect performance as anticipated by the code's authors.

Since the results in Sections 4 and 5 evaluate specific implementations of NAS MG on a specific set of compilers, they may not necessarily reflect the maximum potential of each compiler or language (in truth, one would be hard-

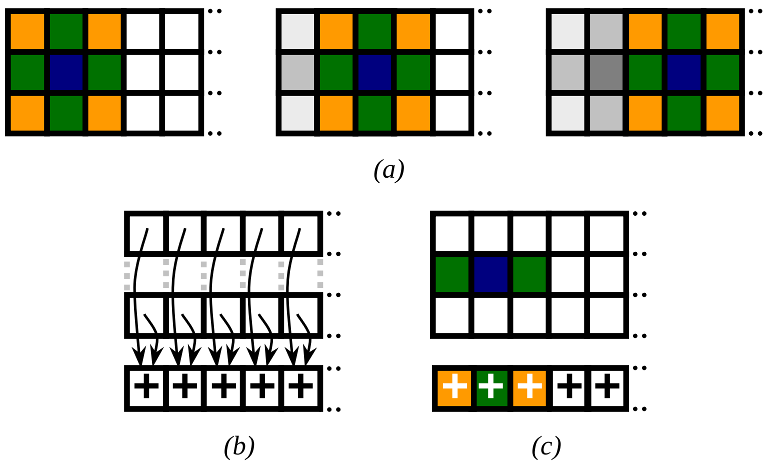

Figure 1: A 2D illustration of the stencil optimization hand-coded in the Fortran versions of NAS MG. *(a)* Adjacent applications of the stencil. Note that the sum of the rightmost top and bottom values in the first application are re-used in the second and third. *(b)* To take advantage of this, a vector of sums is pre-computed. *(c)* To compute the stencil, the sum values are weighted and added to the weighted values from the middle row. Note that the benefits for 3D stencils are even greater.

pressed to design an experiment that would). However, our methodology strives to ensure that while one might conceive of a faster or more expressive implementation, our implementations represent reasonable examples of how a knowledgeable, performance-minded programmer might implement a specific multigrid algorithm in each language using current compiler technology.

3.2 The NAS MG Implementations

In this section, we give a brief description of each implementation, emphasizing ways in which it differs from the original NAS version.

F90+MPI The F90+MPI code that we use is the NAS implementation, version 2.3. It serves as the baseline for this study. It should be mentioned that this implementation utilizes a clever hand-optimization in each of the 27-point stencils. The optimization computes subexpressions that are reused in adjacent applications of a stencil and caches them to minimize redundant FLOPs (Figure 1). This significantly improves performance at the cost of obscuring the source code's intent (and it was a wise decision: we found that current Fortran compilers do not perform this transformation when the stencils are written in a naive manner, causing a performance degradation of up to 17% or more).

HPF The HPF implementation is obtained from NASA Ames [18] and stems from an effort to implement all of the NAS benchmarks in HPF. PGI identifies this implementation as the best-known publicly-available version of NAS MG for their compiler [30], which serves as the HPF compiler in our experiments[1]. This implementation follows the F90+MPI implementation of MG very closely with one major exception: as alluded to in Section 2, the authors had to represent their hierarchical arrays using 4D arrays in order to achieve a concise, flexible parallel implementation [18, 19]. This requires extensive use of HPF's `HOME` directive in order to align the arrays in a distributed and load

[1]We regret that other HPF compilers are not represented in this study. We had a difficult time obtaining versions of other HPF compilers, much less ones that supported an implementation of NAS MG. We welcome any implementations that the reader might bring to our attention.

Language	*Author*	*# Processors unbound?*	*Problem Size unbound?*	*Data Distribution*	*Contains Stencil Optimization*
F90+MPI	NAS	no (2^x)	no (2^x)	3D blocked	yes
HPF	NAS	yes	no	1D blocked[2]	yes
CAF	CAF group	no (2^x)	no (2^x)	3D blocked	yes
SAC	SAC group	yes	yes[3] (2^x)	1D blocked	no
ZPL	ZPL group	yes	yes	3D blocked	no

Table 3: Summary of the MG implementations used in this study. *Author* indicates the origin of the code. The next two columns indicate whether the number of processors and/or problem size can be specified at runtime rather than fixed at compile-time. If the implementation also constrains the number of processors or problem size to be a power of two, this is indicated in parenthesis. *Data distribution* indicates the way in which arrays are distributed across processors. The last column indicates whether or not the implementation includes the hand-coded stencil optimization described in Section 3.2

balanced manner. Other than this issue, the implementation is extremely true to the F90+MPI version and includes its hand-coded stencil optimizations. Future work should consider how these 4D arrays might be avoided in other HPF compilers [20, 2] or using HPF extensions for sparse or irregular problems [22, 35].

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 90 as their base language, the implementation simply involved removing the MPI calls and replacing them with the equivalent co-array syntax. Although solutions more tailored to CAF could be imagined, this implementation is as true to the original F90+MPI implementation as one could imagine and was a fairly trivial port for its authors.

SAC The SAC implementation of MG comes as part of the SAC distribution [36] and forms the most radical departure from the NAS F90+MPI implementation. This is primarily due to the decision to use recursion to express hierarchies as discussed in the previous section. This differs from the NAS implementation's iterative approach in which the hierarchical arrays are allocated at the outset of the program 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 shift in paradigm would neither be in the spirit of SAC nor in its best interests performance-wise. The NAS stencil optimization could not be cleanly hand-coded in SAC without disabling other essential optimizations, so all stencils are expressed in their direct but redundant form.

ZPL The ZPL implementation of MG was written by mimicking the F90+MPI implementation as carefully as possible. The hierarchical arrays, procedural organization, and computations all follow the original scheme. The main difference is that the stencil optimization could not easily be hand-coded in ZPL without wasting a significant amount of memory. Therefore, the more direct but redundant means of expressing the stencils was used.

Table 3 summarizes the versions of the benchmark that we used for our experiments. Included in this table is information about whether each implementation fixes or constrains the problem size and/or number of processors at compile-time. While a completely dynamically-specified MG could be written in each language, doing so could adversely affect expressiveness and/or performance by making the code more general and providing less information to

[2]The implementors found that distributing more than one dimension hurt performance in HPF, so chose to distribute just one [18].
[3]Though the problem size may be specified dynamically, the code is written such that only a few problem sizes are possible.

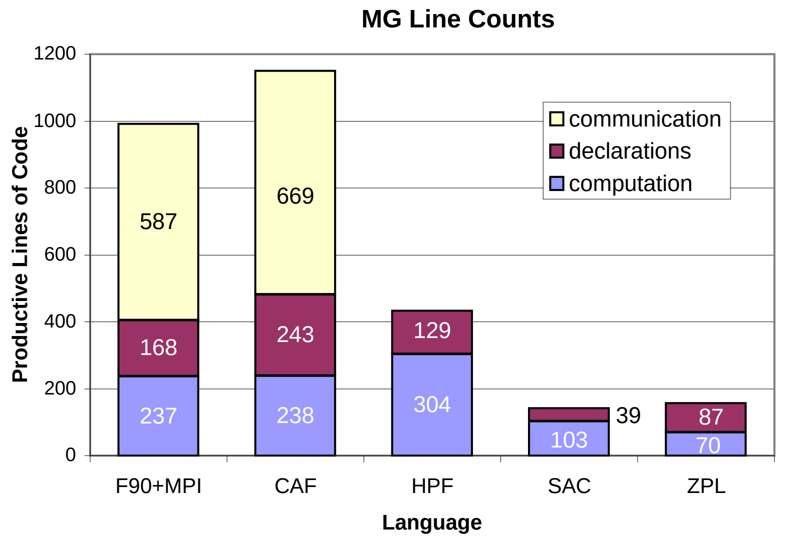

Figure 2: An indication of the number of useful lines of code in each implementation of MG. *Communication* indicates the number of lines devoted to communication and synchronization. *Declarations* indicates code used to declare variables, constants, and identifiers. *Computation* indicates the number of lines used to express the computation itself. Note that the local-view languages require 2–8 times as many lines of code, due primarily to the fact that programmers must explicitly manage communication.

the compiler. For example, the F90+MPI implementation uses its knowledge of the number of processors and problem size to allocate its hierarchical arrays statically. Similarly, by constraining the problem sizes in SAC, inlining and loop unrolling optimizations are enabled. The point here is not to argue whether or not such constraints are reasonable, but rather to highlight that the more dynamic implementations start with a slight disadvantage. The table also indicates the data distribution used by each implementation. The current distribution of SAC always uses a 1D decomposition. For all other languages, the distribution was specified by the programmer.

4 Evaluation of Expressiveness

To evaluate the expressiveness of the MG implementations, we perform both quantitative and qualitative analysis. In the quantitative stage, we classify the lines of each benchmark as one of four types: *declarations*, *communication*, *non-essential*, and *computation*. Declarations include all lines of code that are used for the declaration of variables, constants, and other identifiers. Communication lines are those that are used for synchronization or interprocessor data transfer. Code related to comments, initialization, timings, and I/O are considered non-essential and excluded from the analysis. The remaining lines of code form the timed, computational kernel of the benchmark and are considered computation. Because linecounts will vary somewhat due to a programmer's style, we use these figures only as a coarse indicator of expressiveness.

Figure 2 gives a summary of the quantitative classification, showing the number of essential lines of code and how they break down into our taxonomy. The most immediate observation is that languages with a local view of computation require 2 to 8 times as many lines of code as those providing a global view, and that the majority of these lines implement communication. Inspection of this communication 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 at all levels of the hierarchy. The difference in communication counts between F90+MPI and CAF stems from MPI's built-in support for collective communication routines such as broadcasts, reductions, etc. In

```
      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(r,m1k,m2k,m3k,s,
     >                                  m1j,m2j,m3j,k)

      implicit none
      include 'globals.h'
      include 'rprj3.h'
      integer j3, j2, j1, i3, i2, i1, d1, d2, d3, j
      double precision x1(m), y1(m), x2,y2
      double precision w000(m1j,m2j,m3j), w100(m1j,m2j,m3j),
     >                 w010(m1j,m2j,m3j), w110(m1j,m2j,m3j),
     >                 w001(m1j,m2j,m3j), w101(m1j,m2j,m3j),
     >                 w011(m1j,m2j,m3j), w111(m1j,m2j,m3j)

!hpf$   align w000(i1,i2,i3) with r(2*i1,2*i2,2*i3)
!hpf$   align w100(i1,i2,i3) with r(2*i1-1,2*i2,2*i3)
!hpf$   align w010(i1,i2,i3) with r(2*i1,2*i2-1,2*i3)
!hpf$   align w110(i1,i2,i3) with r(2*i1-1,2*i2-1,2*i3)
!hpf$   align w001(i1,i2,i3) with r(2*i1,2*i2,2*i3-1)
!hpf$   align w101(i1,i2,i3) with r(2*i1-1,2*i2,2*i3-1)
!hpf$   align w011(i1,i2,i3) with r(2*i1,2*i2-1,2*i3-1)
!hpf$   align w111(i1,i2,i3) with r(2*i1-1,2*i2-1,2*i3-1)

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

C TWO CONDITIONALS OF A SIMILAR FORM DELETED TO SAVE SPACE

!hpf$ independent, on home(w000(i1,i2,i3))
         do  i3=1,m3j-1
!hpf$ independent
            do  i2=1,m2j-1
               do i1=1,m1j-1
                 w000(i1,i2,i3) = r(2*i1,2*i2,2*i3)
     >                           + r(2*i1,2*i2+2,2*i3)
     >                           + r(2*i1+2,2*i2,2*i3)
     >                           + r(2*i1+2,2*i2+2,2*i3)
               end do
            end do
         end do

C 7 LOOPS OF A SIMILAR FORM DELETED TO SAVE SPACE

         s(2:m1j-1,2:m2j-1,2:m3j-1) =
     >                0.5D0* w111(2:m1j-1,2:m2j-1,2:m3j-1)
     >          + 0.25D0 * (  w011(1:m1j-2,2:m2j-1,2:m3j-1)
     >                   +    w011(2:m1j-1,2:m2j-1,2:m3j-1)
     >                   +  w101(2:m1j-1,1:m2j-2,2:m3j-1)
     >                   +  w101(2:m1j-1,2:m2j-1,2:m3j-1)
     >                   +      w110(2:m1j-1,2:m2j-1,1:m3j-2)
     >                   +      w110(2:m1j-1,2:m2j-1,2:m3j-1) )
     >          + 0.125D0 * ( w001(1:m1j-2,1:m2j-2,2:m3j-1)
     >                   +    w001(2:m1j-1,1:m2j-2,2:m3j-1)
     >                   +  w010(1:m1j-2,2:m2j-1,1:m3j-2)
     >                   +  w010(1:m1j-2,2:m2j-1,2:m3j-1)
     >                   +    w100(2:m1j-1,1:m2j-2,1:m3j-2)
     >                   +    w100(2:m1j-1,1:m2j-2,2:m3j-1) )
     >         + 0.0625D0 * ( w000(1:m1j-2,1:m2j-2,1:m3j-2)
     >                   +  w000(1:m1j-2,1:m2j-2,2:m3j-1)  )

       s(m1j,:,:) = s(2,:,:)
       s(1,:,:) = s(m1j-1,:,:)
       s(:,m2j,:) = s(:,2,:)
       s(:,1,:) = s(:,m2j-1,:)
       s(:,:,m3j) = s(:,:,2)
       s(:,:,1) = s(:,:,m3j-1)

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

(d) **ZPL version**

Figure 3: The *rprj3* stencil as expressed in each implementation. Note that to save space, repeated idioms are condensed as comments in the Fortran versions and that communication for the F90+MPI/CAF version is omitted completely.

Machine	*Location*	*Processors*	*Speed*	*Memory*	*Memory Model*
Linux cluster[4]	LANL	128 dual P-IIIs	500 MHz	0.938 GB	Distributed Memory
IBM SP	MHPCC	96	222 MHz	0.5 GB	Distributed Memory
Cray T3E	ARSC	256	450 MHz	0.256 GB	Dist. Glb. Address Space
Sun Enterprise 5500	UT Austin	14	400 MHz	0.143 GB	Shared Mem. Multiproc.
SGI Origin	LANL	2048 (16×128)	250 MHz	0.25 GB	Shared Mem. Multiproc.

Table 4: A summary of the machines used in our experiments. *Location* indicates the institution that owns the machine and donated the computer time. *Processors* indicates the total number of processors available to a single user. *Speed* gives the clock speed of the processors. *Memory* tells the amount of memory available per processor. *Memory Model* indicates the organization of memory on the platform.

the CAF version, such operations were manually implemented as functions, adding to both its communication and declaration counts.

The next thing to note is that of the global-view languages, HPF is considerably lengthier than either SAC or ZPL. This is an unfortunate consequence of the extra code and directives required to implement its 4D hierarchical arrays as described in Section 3.2. Examples of such code can be seen in lines 14–21 and 31–44 of Figure 3. Though the SAC and ZPL linecounts are similar to one another, SAC takes a 30-line computation hit by explicitly replicating top-level function calls for each problem size to enable inlining and unrolling. On the other hand, it requires 48 fewer lines for declarations due to its support for implicit variable declarations.

In terms of computation, one observes that the F90+MPI and CAF implementations have 2 to 3 times the number of lines as the SAC and ZPL benchmarks. This can largely be attributed to the looping and indexing overhead associated with hand-coding the stencil optimization. Although converting the stencils in these implementations to an unoptimized slice-notation brings the computation line count closer to that of SAC and ZPL, we found that this caused the performance to degrade significantly (Section 3.2), and that communication continues to vastly dominate the overall line counts and complexity for these codes.

Our qualitative evaluation consists of reading through the code by hand and looking to see if the line counts seem inversely related to the clarity with which each implementation expresses the algorithm. Our conclusion is that they are. Looking at Figure 3 as an example, and noting that much of the Fortran code has been omitted for brevity, one finds that SAC and ZPL are not only concise, but also clear representations of the projection stencil using functional and imperative styles, respectively. The clarity of each can be attributed to its global view, its support for whole-array operations and language-level index sets, and its lack of hand-coded optimizations. In contrast, the Fortran-based codes are obscured by looping structures, computations related to management of data distribution, hand-coded optimizations, and communication specification. The experiments of the next section indicate whether this additional coding pays off in terms of performance.

5 Performance Evaluation

5.1 Methodology

To evaluate performance, we ran the MG implementations on five hardware platforms: a Linux cluster, a Cray T3E, a Sun Enterprise 5500, an IBM SP, and an SGI Origin. The machines represent the diversity of parallel architectures in use today. Relevant details are summarized in Table 4.

[4]The Linux cluster is linked by Myrinet as well as 100Mb ethernet. Both networks are represented in our experiments.

	F90+MPI	CAF	HPF	SAC	ZPL
Linux cluster	•	×	•	×	•
IBM SP	•	×	•	×	•
Cray T3E	•	•	•	×	•
Sun Enterprise 5500	•	×	⊖	•	•
SGI Origin	•	×	•	×	•

Table 5: A summary of the language/hardware combinations studied in these experiments. "×" indicates that the language is not currently supported on the machine. "•" indicates that it is supported and included in our results. "⊖" indicates that it is supported, but that we do not have access to a compiler for it.

Currently, no single platform supports all of the languages in our study since CAF only runs on the Cray T3E and SAC only runs on the Sun Enterprise. Table 5 summarizes the hardware/language combinations that we studied.

Appendix A summarizes information about the compilers and command-line flags that we used to collect our results. In most cases we used the highest level of single-flag optimizations available. For the HPF benchmark, we used the flags suggested by its implementors. Both the SAC and ZPL compilers use ANSI C as an intermediate language, so we also provide information about the C compilers and flags used by their back ends.

5.2 Performance Results

The graphs in Figures 4 and 5 give speedup results for each machine. Timings were obtained by running each configuration (*language* × *machine* × *number of processors*) a handful of times over the course of several days and taking the best time for each. Appendix B contains this data. Note that most of the machines did not have sufficient memory to run the production grade classes on small numbers of processors. Thus, speedup numbers for each graph are computed relative to the fastest implementation on the smallest number of processors. Classes B and C are shown for each machine except the Sun Enterprise which had insufficient memory to run a class C problem. We begin our discussion of the performance results by focusing on the languages that are supported on multiple platforms.

HPF The most striking observation is that the HPF implementation performs poorly as compared to all other languages. This is due to the overheads associated with its 4D hierarchical array implementation, as anticipated in Section 3.2. The amount of storage required by these arrays prevents many of the HPF configurations from running due to memory limitations. Notable exceptions are the Linux cluster, which has the largest amount of memory per node, and the SGI Origin, whose shared memory architecture allows small processor configurations to utilize memory from other processors on the machine that are not involved in the computation. The only class C HPF results were obtained on the SGI Origin using 8–16 processors. Running on fewer nodes causes each processor to exceed the system-specified limit for how much memory it can use. Running on more processors exhausts the global amount of memory available to the group of 128. Note that while the Linux cluster has a greater amount of memory per processor, its distributed memory architecture prevents it from running a class C problem using any number of processors.

When the HPF implementation does have sufficient memory to run, the 4D arrays impact overall performance due to their large memory footprints and the extra computation required for their correct distribution and alignment. Note that the HPF implementations *are* in fact scaling, just not at a rate that makes them competitive with the other implementations. At this point, it is beyond our ability to judge whether these results are a symptom of the youth of HPF compilers or whether the language itself poses significant obstacles to efficient compilation of multigrid applications.

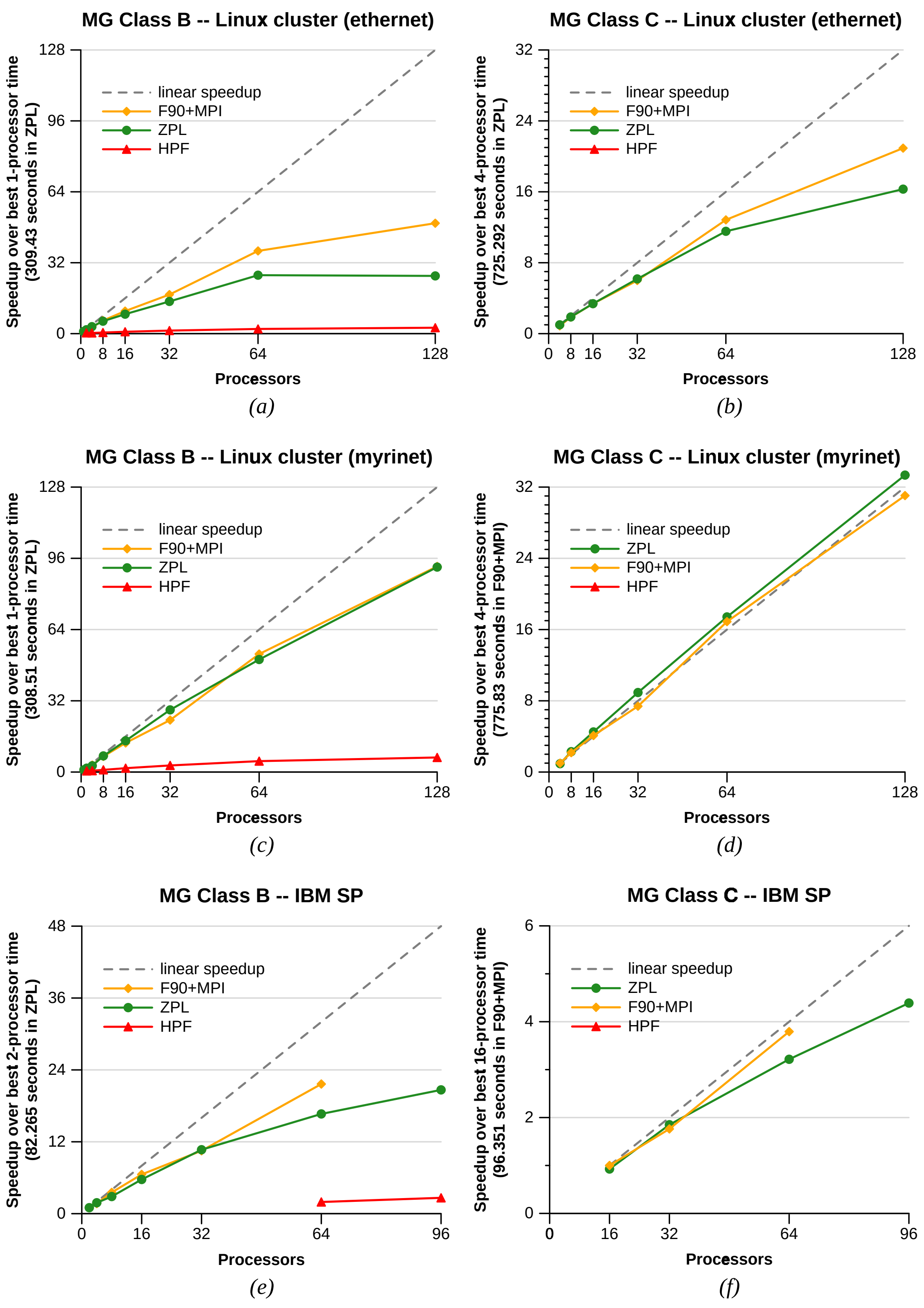

Figure 4: Speedup curves for classes B and C of MG on the Linux cluster using both ethernet and myrinet, and on the IBM SP. Note that there is not enough memory to run these problem sizes on small processor sets. Thus, we compute speedups for each graph using the fastest execution time on the smallest number of processors (indicated in the *y*-axis label). Due to its excessive memory allocation, the HPF version is unable to run on most configurations. Note that the F90+MPI version cannot run on 96 processors since it is not a power of two.

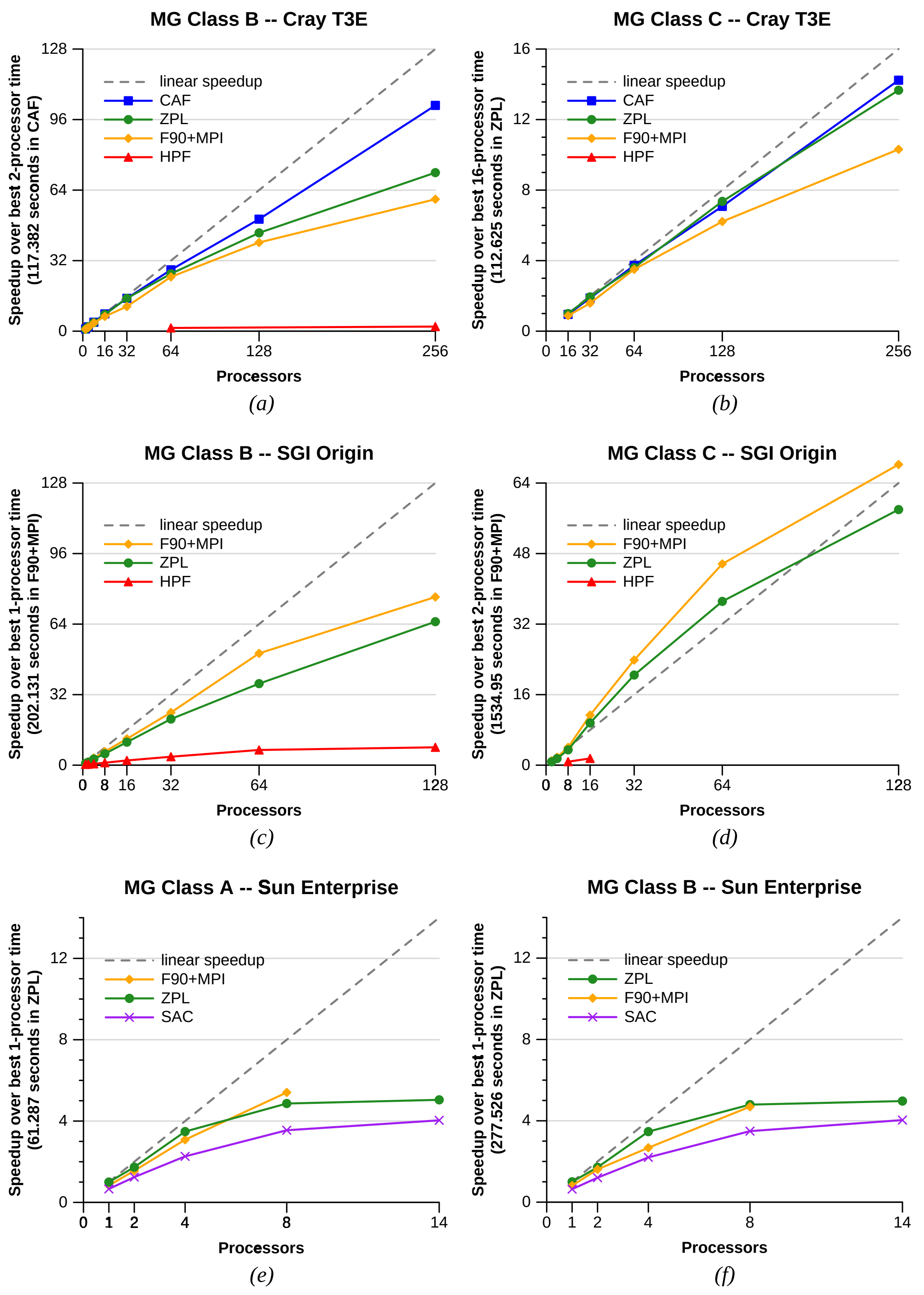

Figure 5: Speedup graphs for MG classes B and C on the Cray T3E and SGI Origin, and for classes A and B on the Sun Enterprise 5500. As in the previous figure, speedups for each graph are computed using the fastest execution time on the smallest number of processors. The superlinear speedup on the Origin is due to the memory traffic required to run a class C problem on 2 processors. We were unable to obtain reasonable timings on more than 128 Origin processors due to the network traffic involved in crossing between machines. See Appendix B for details.

F90+MPI & ZPL By way of comparison, the F90+MPI and ZPL implementations perform reasonably well. As the processor set size increases, F90+MPI tends to outpace ZPL due to communication overhead. Recall that the F90+MPI implementation is hand-coded at a local-view, allowing the programmer to explicitly and precisely indicate when and where communication is required. In contrast, ZPL's data transfers are compiler-generated. Although the compiler performs several communication optimizations including vectorization, redundancy elimination, and pipelining [12, 11], inspection shows that it fails to insert the minimal amount of communication required by the benchmark. ZPL's communication overheads are apparent in the fact that the gap between F90+MPI and ZPL tends to be greater for class B problems than class C.

In addition, the fact that ZPL does not fix or constrain the number of processors and problem size at compile-time results in a certain amount of runtime overhead to determine which data elements must be sent and received for each communication. However, this flexibility pays off on machines such as the IBM SP and Sun Enterprise which do not have a power-of-two processor set. On these machines, the ZPL implementation can run using all of the processors while the F90+MPI implementation must stick to powers of two. If the amount of computation is great enough (as for class C on the IBM SP), this can result in performance unobtainable by the F90+MPI version.

On smaller numbers of processors the two implementations are fairly comparable. This may come as a surprise since the F90+MPI code contains hand-coded stencil optimizations that were not efficiently expressible in the ZPL source. However, the ZPL compiler automatically optimizes these stencils, giving the user similar performance benefits without the hassle of hand-coding the optimization [15]. The ZPL optimization differs from the hand-coded version by computing its subexpressions on an as-needed basis and caching the results in a small set of scalar temporaries. This results in improved temporal and spatial locality. It also unrolls the inner loop by the stencil's width to minimize shifts between these temporaries. These tweaks represent optimizations that a programmer may not be willing to do by hand, yet which have a measurable positive impact on performance.

Note that the Cray T3E results are exceptional due to the fact that the ZPL implementation significantly outperforms F90+MPI. This is a result of the fact that ZPL's IRONMAN communication interface allows it to map to the SHMEM interface on the T3E rather than MPI [8]. Previous work has shown that the message passing paradigm is a poor match for architectures that support one-sided communication, due to extraneous buffering and synchronization overheads [34, 8]. By using the one-sided SHMEM interface, ZPL can avoid these overheads and use a communication style that more closely matches the target architecture. Presumably, a shared-memory port of IRONMAN could result in similar improvements on the SGI Origin and Sun Enterprise, but at this time only tentative steps in that direction have been taken [11]. It should be noted that while the HPF implementation was also compiled to the SHMEM interface, it failed to hoist synchronization out of the stencils' inner loops, negating its benefits.

CAF The CAF implementation also benefits from one-sided communication on the Cray T3E. The CAF compiler directly generates assembly instructions to perform the remote puts and gets required by the benchmark. The result is performance that is unmatched by the other implementations. For configurations that are not computation bound, ZPL falls away from CAF much as it did from F90+MPI on other platforms due once again to the overhead of flexible compiler-generated communication as compared to precise programmer-specified communication.

It should be noted that while it would certainly be possible to support CAF on any platform, its co-array syntax predisposes it towards one-sided communication styles. This presents an interesting challenge for efficiently implementing CAF on distributed memory machines where one-sided communication is not inherently supported by the architecture. This paradigm mismatch could cause performance degradations similar to those seen when running MPI on the T3E.

SAC Looking at the Sun Enterprise, we see that SAC scales similarly to ZPL and F90+MPI, but lags slightly in performance. The reason for this is the lack of the stencil optimization, either hand-coded or by the compiler. Detection and optimization of stencils in SAC should be no more difficult than in ZPL, and would likely be a worthwhile addition to the SAC compiler. This optimization combined with its implicit support for shared-memory should allow it to easily outperform both the F90+MPI implementation and an MPI-based ZPL implementation.

6 Related Work

For each of the languages in this paper, work has been done on expressing and optimizing hierarchical computations in that language [1, 18, 29, 32, 9]. However, each of these papers studies the language independently, and typically on a single platform. Our study is unique in its cross-language and cross-platform comparison.

Another 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 [17], 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 impact of using a library rather than a language for hierarchical applications not only in terms of performance, expressiveness, and portability, but also opportunity for optimization.

POOMA (Parallel Object-Oriented Methods and Applications) [24] is a template-based C++ class library for large-scale scientific computing. Although POOMA does not directly support multigrid computations, its *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 [13] 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 [23], and future studies should evaluate its suitability for hierarchical multigrid-style problems.

7 Conclusions

In this work, we have used the NAS MG benchmark to study parallel language support for multigrid applications. Our study used a variety of parallel languages and architectures that represent the current diversity in parallel computing. In doing so, we made a best effort to find NAS MG implementations that accurately represent how a savvy programmer would implement a particular multigrid algorithm in each language. We evaluated each implementation's performance, expressiveness, and portability, and summarize these findings here.

Scalar performance was seen to be primarily affected by two factors: The first is that hierarchical arrays should use the minimal required amount of memory. The F90+MPI, CAF, SAC, and ZPL implementations all use the same amount of memory — SAC allocates it recursively while the others allocate it explicitly at the program's start. In contrast, the HPF implementation used an excessive amount of memory and suffered as a result, both in terms of performance and its ability to run large problem sizes on small processor sets.

The second factor in scalar performance is the ability to optimize stencils, whether it is done explicitly by the programmer or automatically by the compiler. While this does not affect performance as much as a poor memory allocation, we found that it could cause significant differences in execution time, especially for problem sizes that are computation-bound.

In terms of parallel performance, the most important factor is to use a communication paradigm that matches that of the architecture, as shown by ZPL and CAF on the Cray T3E. Apart from that, traditional communication optimizations that minimize latency, numbers of communications, data transferred, and runtime overhead all proved to be valuable.

In terms of expressiveness, we found that supporting a global view of computation allows for a huge benefit in terms of a code's size and clarity. While explicit specification of communication and data distribution proved to benefit performance in this study, each programmer must decide whether the effort required to develop and maintain a local-view parallel code, as well as the distraction that it causes from the algorithm at hand, is worth the potential benefit in performance. ZPL and SAC indicate that good to exceptional parallel performance can be obtained without sacrificing the global view. Other factors that influenced expressiveness include high-level array operations, source-level index set representations, and compiler-support for stencil optimizations.

Portability in these languages is achieved primarily via the widespread support for MPI. MPI not only allows the F90+MPI implementation to run on every platform, but it also provides a useful mechanism for portable compilation of languages, as seen with HPF and ZPL. The ZPL compiler's IRONMAN library demonstrates that compiler support for non-message-passing communication styles can result in portable performance that cannot be achieved by strictly using MPI. CAF presents similar challenges to portable performance due to its reliance on one-sided communication, which typically underperforms message passing on distributed memory architectures. Though SAC is currently supported on only two architectures, the similarity between its with-loops and ZPL's regions gives reason to hope that it could be compiled portably using runtime support similar to that of ZPL.

To conclude, we use our experiences with the NAS MG benchmark to classify whether or not each language used in this study is expressive, has reasonable performance, and is portable. Looking at Table 5 and the graphs in Figures 2, 4, and 5, we find that the languages are naturally bimodal in each category. At present, F90+MPI, HPF, and ZPL are far more portable than CAF and SAC. Local-view languages are seen to be far less expressive than those supporting a global view. And all languages other than HPF currently obtain reasonable performance for this benchmark. The result is the Venn diagram in Figure 6, which summarizes our findings for the current state-of-the-art in parallel language support for dense multigrid computations.

In future work, we plan to continue our study of language support for hierarchical applications, looking toward more complex applications such as AMR and FMM [25, 16]. These applications differ from MG in that different areas of the hierarchical problem space are refined to differing degrees. Our approach in this work will be based on mixing traditional ZPL multi-regions with Advanced ZPL's support for *sparse regions* [10] in an attempt to preserve the expressiveness, performance, and portability demonstrated in this first experiment.

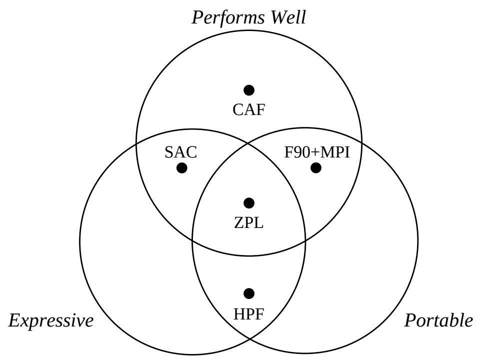

Figure 6: A Venn diagram that summarizes the results of our paper. Comparing the languages to one another using the NAS MG benchmark and current compilers, we find that they fall into distinct groups in terms of expressiveness, portability, and performance. For each characteristic, we make a simple binary decision as to whether or not each language has the property in question. It should be emphasized that this diagram will certainly vary when using different benchmarks and as the compiler technology for each language matures.

Acknowledgments. The authors would like to thank Sven-Bodo Scholz, Clemens Grelck, Alan Wallcraft, and NASA Ames for providing 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 NASA employees who helped us obtain the HPF implementation. This work was performed using grants of supercomputer time from the Alaska Region Supercomputer Center, the University of Texas at Austin, Los Alamos National Lab, and the Maui High Performance Computing Center. We are extremely grateful for the use of their computers and support from their respective staffs. A debt of gratitude is owed to Sung-Eun Choi, Calvin Lin, and E Christopher Lewis for their help in developing the ZPL language and for providing feedback on drafts of this manuscript. In addition, Ton A. Ngo must be thanked for his invaluable help in motivating and designing the 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] P. Banerjee, J. A. Chandy, M. Gupta, , E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[3] Eric Barszcz and Paul Frederickson. *NAS MG 2.3 release notes*. December 1995.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] Bradford L. Chamberlain, Steven Deitz, and Lawrence Snyder. Parallel language support for multigrid algorithms. Technical Report UW-CSE 99-11-03, University of Washington, Seattle, WA USA, November 1999.

[10] 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.

[11] Sung-Eun Choi. *Machine Independent Communication Optimization*. PhD thesis, University of Washington, Department of Computer Science and Engineering, March 1999.

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

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

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

[15] Steven J. Deitz. On eliminating redundant computation from high-level array statements. Technical Report UW-CSE 2000-04-02, University of Washington, Seattle, WA USA, April 2000.

[16] 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.

[17] 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.

[18] 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.

[19] Michael A. Frumkin. *Personal communication.* NASA Ames Research Center, April 2000.

[20] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Y. Wang, W. M. Ching, and T. Ngo. An HPF compiler for the IBM SP-2. In *Proceedings of Supercomputing '95*, December 1995.

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

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

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.

[29] 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.

[30] Portland Group technical reporting service. *Personal communication.* October 1999.

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

[32] 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.

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

[34] T. Stricker, J. Subhlok, D. O'Hallaron, S. Hinrichsand, and T. Gross. Decoupling synchronization and data transfer in message passsing systems of parallel computers. In *9^{th} International Conference on Supercomputing*, July 1995.

[35] Manuel Ujaldon, Emilio L. Zapata, Barbara M. Chapman, and Hans P. Zima. Vienna-fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(10), October 1997.

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

[37] Rob Van der Wijngaart. *Personal communication.* MRJ Technology Solutions, April 2000.

[38] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna fortran - a language specification version 1.1. Technical Report ACPC/TR 92-4, Austrian Center for Parallel Computation, March 1992.

A Compiler Specifications

The following is a summary of the compilation process used in these experiments. The compiler, version number, and command-line arguments used are given for each language and platform. In addition, the communication mechanism used at runtime is noted.

Language	*Compiler*	*Version*	*Command-line arguments*	*Comm. Mechanism*
			Linux cluster compilers	
F90+MPI	GNU g77	0.5.25	–O3	MPI (MPICH 1.2)
HPF	PGI pghpf	3.1-2	–O3 –Mautopar –Moverlap=size:1 –Mmpi	MPI (MPICH 1.2)
ZPL	U. Wash. zc	1.16a		MPI (MPICH 1.2)
	GNU gcc	2.95.2	–O3	
			IBM SP compilers	
F90+MPI	IBM mpxlf	2.4	–O3	MPI (IBM)
HPF	PGI pghpf	2.4-4	–O3 –Mautopar –Moverlap=size:1 –Mmpi	MPI (IBM)
ZPL	U. Wash. zc	1.16a		MPI (IBM)
	IBM mpcc	2.4	–O3	
			Cray T3E compilers	
F90+MPI	Cray f90	3.3.0.0	–O3	MPI (Cray)
CAF	Cray f90	3.3.0.0	–O3 –X 1 –Z *nprocs*	E-registers
HPF	PGI pghpf	3.0-1	–O3 –Mautopar –Moverlap=size:1 –Msmp	SHMEM
ZPL	U. Wash. zc	1.16a		SHMEM
	Cray cc	6.3.0.0	–O3	
			Sun Enterprise 5500 compilers	
F90+MPI	WorkShop f90	5.0	–fast	MPI (Sun)
SAC	U. Kiel sac2c	0.90alpha	–O3 –v1 –noLIR –noTSI –maxlur 3 –mt	shared memory
	SUNW cc	5.0	–fast	
ZPL	U. Wash. zc	1.16a		MPI (Sun)
	SUNW cc	5.0	–fast	
			SGI Origin compilers	
F90+MPI	MIPSpro 7 f90	7.3.1.1m	–O3	MPI (SGI)
HPF	PGI pghpf	2.4-4	–O3 –Mautopar –Moverlap=size:1 –Mmpi[5]	MPI (SGI)
ZPL	U. Wash. zc	1.16a		MPI (SGI)
	MIPSpro cc	7.3.1.1m	–O3	

B Experimental Timings

The following tables contain the best observed times for each configuration and were used to compute the speedup graphs of Section 5.

Linux cluster (ethernet) — Class B

processors	1	2	4	8	16	32	64	128
F90+MPI	362.60	194.51	110.41	52.51	30.39	17.55	8.29	6.21
ZPL	309.43	175.12	97.83	55.14	35.24	21.30	11.73	11.88
HPF	—.—	896.02	964.65	618.68	362.18	223.62	146.13	115.47

Linux cluster (ethernet) — Class C

processors	1	2	4	8	16	32	64	128
F90+MPI	—.—	—.—	828.18	390.26	215.02	121.04	56.51	34.66
ZPL	—.—	—.—	725.29	384.99	214.52	117.45	62.82	44.49
HPF	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—

[5] Although –Msmp would be more appropriate for the Origin, it was not supported with our installation.

Linux cluster (myrinet) — Class B

processors	1	2	4	8	16	32	64	128
F90+MPI	359.33	181.84	99.13	44.02	23.59	13.21	5.82	3.34
ZPL	308.51	176.92	108.82	42.57	21.96	11.05	6.10	3.35
HPF	—.—	616.38	516.15	314.54	177.57	104.10	62.95	47.15

Linux cluster (myrinet) — Class C

processors	1	2	4	8	16	32	64	128
F90+MPI	—.—	—.—	775.83	356.45	189.27	104.95	45.93	24.99
ZPL	—.—	—.—	827.56	338.99	172.15	86.87	44.53	23.27
HPF	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—

IBM SP — Class B

processors	1	2	4	8	16	32	64	96
F90+MPI	—.—	83.48	48.86	23.19	12.55	7.83	3.80	—.—
ZPL	—.—	82.27	45.12	28.75	14.41	7.71	4.94	3.98
HPF	—.—	—.—	—.—	—.—	—.—	—.—	42.17	31.10

IBM SP — Class C

processors	1	2	4	8	16	32	64	96
F90+MPI	—.—	—.—	—.—	—.—	96.35	54.65	25.40	—.—
ZPL	—.—	—.—	—.—	—.—	104.01	52.08	29.96	21.95
HPF	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—

Cray T3E — Class B

processors	1	2	4	8	16	32	64	128	256
CAF	—.—	117.38	58.86	28.98	14.98	7.87	4.22	2.31	1.15
F90+MPI	—.—	127.22	68.94	31.47	17.52	10.49	4.77	2.92	1.96
ZPL	—.—	127.79	59.08	30.39	15.55	7.88	4.51	2.63	1.63
HPF	—.—	—.—	—.—	—.—	—.—	—.—	80.16	—.—	56.35

Cray T3E — Class C

processors	1	2	4	8	16	32	64	128	256
CAF	—.—	—.—	—.—	—.—	118.42	59.89	30.23	15.91	7.91
F90+MPI	—.—	—.—	—.—	—.—	128.59	70.89	32.14	18.12	10.92
ZPL	—.—	—.—	—.—	—.—	112.63	57.85	31.24	15.30	8.24
HPF	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—	—.—

SGI Origin — Class B

processors	1	2	4	8	16	32	64	128	256
F90+MPI	202.13	130.04	61.88	32.84	16.93	8.46	3.98	2.65	264.17
ZPL	244.88	147.27	72.37	39.06	19.37	9.66	5.46	3.10	310.15
HPF	556.13	488.93	341.50	183.94	94.67	52.80	29.33	24.96	—.—

SGI Origin — Class C

processors	1	2	4	8	16	32	64	128	256
F90+MPI	—.—	1534.95	853.55	380.58	135.18	64.36	33.62	22.51	340.35
ZPL	—.—	2004.99	1026.91	441.74	160.35	75.09	41.31	26.47	445.16
HPF	—.—	—.—	—.—	1949.62	1006.18	—.—	—.—	—.—	—.—

Sun Enterprise — Class A

processors	1	2	4	8	14
F90+MPI	73.49	39.36	19.87	11.34	—.—
ZPL	61.29	35.37	17.59	12.60	12.15
SAC	92.56	49.27	27.07	17.27	15.18

Sun Enterprise — Class B

processors	1	2	4	8	14
F90+MPI	341.77	172.33	103.78	59.13	—.—
ZPL	277.53	162.00	80.02	57.87	55.83
SAC	434.02	230.80	126.08	79.49	68.76