Appears in "Proceedings of the SC '97: High Performance Networking and Computing," 1997.

Portable Performance of Data Parallel Languages *

Ton Ngo
P.O. Box 704
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598

Lawrence Snyder, Bradford Chamberlain
Dept. of Computer Science & Engineering
University of Washington
Seattle, WA 98195

Abstract

A portable program executes on different platforms and yields consistent performance. With the focus on portability, this paper presents an in-depth study of the performance of three NAS benchmarks (EP, MG, FT) compiled with three commercial HPF compilers (APR, PGI, IBM) on the IBM SP2. Each benchmark is evaluated in two versions: using DO loops and using F90 constructs and/or HPF's Forall statement. Base-line comparison is provided by versions of the benchmarks written in Fortran/MPI and ZPL, a data parallel language developed at the University of Washington.

While some F90/Forall programs achieve scalable performance with some compilers, the results indicate a considerable portability problem in HPF programs. Two sources for the problem are identified. First, Fortran's semantics require extensive analysis and optimization to arrive at a parallel program; therefore relying on the compiler's capability alone leads to unpredictable performance. Second, the wide differences in the parallelization strategies used by each compiler often require an HPF program to be customized for the particular compiler. While improving compiler optimizations may help to reduce some performance variations, the results suggest that the foremost criteria for portability is a concise performance model that the compiler must adhere to and that the users can rely on.

1 Introduction

Portability is defined as the ability to use the same program on different platforms and to achieve consistent performance. Developing a parallel program that is both **portable** and **scalable** is well recognized as a challenging endeavor. However, the difficulty is not necessarily an intrinsic property of parallel computing. This assertion is especially clear in the case of data parallel algorithms which provide abundant parallelism and tend to involve computation that is very regular. The data parallel model is not adequate for general parallel programming. However, its simplicity coupled with the prevalence of data parallel problems in scientific applications has

*This research was supported by the IBM Resident Study Program and DARPA Grant N00014-92-J-4041 and F30602-97-2-0152

motivated the development of many data parallel languages, all with the goal of simplifying programming while achieving scalable and portable performance.

Of these languages, High Performance Fortran [10] constitutes the most widespread effort, involving a large consortium of companies and universities. One of HPF's distinctions is that it is the first parallel language with a recognized standard — indeed, HPF can be regarded as the integration of several similar data parallel languages including Fortran D, Vienna Fortran and CM Fortran [6, 14, 22]. The attractions of HPF are manifold. First, its use of Fortran as a base language promises quick user acceptance since the language is well established in the target community. Second, the use of directives to parallelize sequential programs implies ease of programming since the directives can be added incrementally without affecting the program's correctness. In particular cases, the compiler may even be able to parallelize the program without user assistance.

On the other hand, HPF also has potential disadvantages. First, as an extension of a sequential language, it is likely to inherit language features that are either incompatible with parallelization or difficult for a compiler to analyze. Second, the portability of a program must not be affected by differences in the technology of the compilers or the machines since the principal purpose for creating a standard is to ensure that programs are portable. HPF's design presents some potential conflicts with the goal of portability. For instance, hiding most aspects of communication from the programmer is convenient, but it forces the user to rely completely on the compiler for generating efficient communication.

Differences between compilers will always be present. However, to maintain the program portability in the language, the differences must not force the users to make program modifications to accommodate a specific compiler. In other words, the user should be able to use any compiler to develop a program that scales, then have the option of migrating to a different machine or compiler for better scalar performance. This requires a tight coupling between the language specification and the compiler in the sense that the compiler implementations must provide a consistent behavior for the abstractions provided in the language. To this end, the language specification must serve as a consistent contract between the compiler and the programmer. We call this contract the **performance model** of the language [18]. A robust performance model has a dual effect: the program performance is (1) predictable to the user and (2) portable across different platforms.

With the focus on the portability issue, we study in-depth the performance of three NAS benchmarks compiled with three commercial HPF compilers on the IBM SP2. The benchmarks are: Embarrassingly Parallel (EP), Multigrid (MG), and Fourier Transform (FT). The HPF compilers include Applied Parallel Research, Portland Group, and IBM. To evaluate the effect of data dependences on compiler analysis, we consider two versions of each benchmark: one programmed using DO loops, and the second using F90 constructs and/or HPF's Forall statement.

For the comparison, we also consider the performance of each benchmark written in MPI and ZPL [16], a data parallel language developed at the University of Washington. Since message passing programs yield scalable performance but are not convenient, the MPI results represent a level of performance that the HPF programs should use as a point of reference. The motivation for including the ZPL results is as follows. ZPL is a data parallel language developed from first principles. The lack of a parent language allows ZPL to introduce new language constructs and incorporate a robust performance model, creating a concrete delineation between parallel and sequential execution. Consequently, the programming model presented to the user is clear, and the compiler is relatively unhindered by artificial dependencies and complex interactions between language features. One may expect that it is both easier to develop a ZPL compiler and to write a ZPL program that scales well. Naturally, the downside of designing a new language without a legacy is the challenge of gaining user acceptance. For this study, the ZPL measurement gives an indication as to whether consistent and scalable performance can be achieved when the compiler is not hampered by language features unrelated to parallel computation.

Our results show that programs that scale well using a particular HPF compiler may not perform similarly with a different compiler, indicating a lack of portability. Some F90/Forall programs achieve scalable performance, but the results are not uniform. For the other programs, the results suggest that Fortran's sequential nature leads to considerable difficulties in the compiler's analysis and optimization of the communication. By analyzing in detail the implementations by the HPF compilers, we find that the wide differences in the parallelization strategies and their varying degrees of success contribute to the portability problem of HPF programs. While improving compiler optimizations may help to reduce some performance variations, it is clear that a robust solution will require more than a mature compiler technology. The results suggest that the foremost criteria for portability is a concise performance model that the compiler must adhere to and that the users can rely on. This performance model will serve as an effective contract between the users and the compiler.

In related work, APR published the performance of its HPF compiler for a suite of HPF programs, along with detailed descriptions of their program restructuring process using the APR FORGE tool to improve the codes [3, 11]. The programs are well tuned to the APR compiler and in many cases rely on the use of APR-specific directives rather than standard HPF directives. Although the approach that APR advocates (program development followed by profiler-based program restructuring) is successful for these instances, the resulting programs may not be portable with respect to performance, particularly in cases that employ APR directives. Therefore, we believe that the suite of APR benchmarks is not well suited for evaluating HPF compilers in general.

Similarly, papers by vendors describing their individual HPF compilers typically show some performance numbers; however it remains difficult to make comparisons across compilers [8, 12, 13].

Lin et al. used the APR benchmark suite to compare the performance of ZPL versions of the programs against the corresponding HPF performance published by APR and found that ZPL generally outperforms HPF [17]. However, without access to the APR compiler at the time, detailed analysis was not possible, limiting the comparison to the aggregate timings.

This paper makes the following contributions:

1. An in-depth comparison and analysis of the performance of HPF programs with three current HPF compilers and alternative approaches (MPI, ZPL).

2. A comparison of the DO loop with the F90 array syntax and the Forall construct.

3. An assessment of the parallel programming model presented by HPF.

The remainder of the paper is organized as follows: Section 2 describes the methodology for the study, including a description of the algorithms and the benchmark implementations. In Section 3, we examine and analyze the benchmarks' performance, detailing the communication generated in each implementation and quantifying the effects of data dependences in the HPF programs. Section 4 provides our observations and our conclusions.

2 Methodology

2.1 ZPL Overview

ZPL is an array language designed at the University of Washington expressly for parallel execution. In the context of this paper, it serves two purposes. First, it sets a bound on the performance that can be expected from a high level data parallel language that is not an extension of an existing sequential language. Second, it illustrates the importance of the performance model in a parallel language.

ZPL is implicitly parallel — i.e. there are no directives. The concurrency is derived entirely from the semantics of the array operations. Array decompositions, specified at run time, partition arrays into either 1D or 2D blocks. Processors perform the computations for the values they own. Scalars are replicated on all processors and kept coherent by redundantly computing scalar computations.

ZPL introduces a new abstraction called region which is used to allocate distributed arrays and to specify distributed computation. ZPL provides a full complement of operations to define regions relative to each other (e.g. [east of R]), to refer to adjacent elements (e.g. A@west), to perform full and partial prefix operations (e.g. big := max << A), to express strided computations, to establish boundary conditions (e.g. wrap A), and to accomplish other powerful operations (e.g. flooding). ZPL also contains standard operators, data types and control structures, using a syntax similar to Modula-2.

2.2 Benchmark selection

To emphasize the portability issue, we establish these criteria:

1. The benchmarks should be derived from an independent source to insure objectivity.

2. A message passing version should be included in the study to establish the target performance.

3. For HPF, there should be separate versions that employ F77 DO loop and F90/Forall because there is a significant difference between the two types of constructs.

4. The algorithm should be parallel and there should be no algorithmic differences between versions of the same benchmark.

5. Tuning must adhere to the language specification rather than any specific compiler capability.

6. Because support for HPF features is not uniform, the benchmarks should not require any feature that is not supported by all HPF compilers.

Following these criteria proves to be challenging given the disparate benchmark availability. The NAS benchmark version 1.0 (NPB1) was implemented by computer vendors and was intended to measure the best performance possible on a parallel machine without regard to portability. The available sources for NPB1 are generally sequential implementations. Although they are valid HPF programs, the sequential nature of the algorithms may be too difficult for the compilers to parallelize and may not reflect a natural approach to parallel programming. For instance, a programmer may simply choose a specific parallel algorithm to implement in HPF.

The NAS version 2.1 (NPB2.1), intended to measure portable performance, is a better choice since the programs implement inherently parallel algorithms and they use the same MPI interface as the compilers in the study. NPB2.1 contains 7 benchmarks[1], all of which should ideally be included in the study. Unfortunately, a portable HPF version of these benchmarks is not available, severely limiting an independent comparison. While APR and other HPF vendors publish the benchmarks used to acquire their performance measurements, these benchmarks are generally tuned to the specific compiler and are not portable. This limitation forces us to carefully derive HPF versions from the NPB2.1 sources with the focus on portability while avoiding external effects such as algorithmic differences.

Among the benchmarks, CG is not available in NPB2.1. SP, BT and LU are not included in this study because they require block cyclic and 3-D data distributions that are not supported by all HPF compilers. These limitations do not prevent these benchmarks from being implemented in HPF and ZPL; however, the implementations will have an algorithmic difference that cannot be factored from the performance. This leaves only FT and MG as potential candidates. Fortunately, EP is by definition highly parallel; therefore its sequential implementation can be trivially parallelized.

2.3 Benchmark implementation

The HPF implementations are derived by **reverse engineering** the MPI programs: communication calls are removed and the local loop bounds are replaced with global loop bounds. To parallelize the programs, HPF directives are then added to recreate the data partitioning of the MPI versions. The HPF compilers are thus presented with a program that is fully data parallel (not sequential) and ready to be parallelized. Conceptually, the task for the compilers is to repartition the problem as specified by the HPF directives and to regenerate the communication. The benchmarks we chose require only the basic BLOCK distribution which is supported by all three compilers and use only the basic HPF intrinsics. Therefore they can stress the compilers without exceeding their capability.

The implementations in the ZPL language are derived from the same source as the HPF implementations, but in the following manner: the sequential computation is translated directly from Fortran to the corresponding ZPL syntax, while the parallel execution is expressed using ZPL's parallel constructs.

[1] One was added recently.

2.3.1 EP

The benchmark EP generates N pairs of pseudo-random floating point values (x_j, y_j) in the interval (0,1) according to the specified algorithm, then redistributes each value x_j and y_j onto the range (-1,1) by scaling them as $2x_j - 1$ and $2y_j - 1$. Each pair is tested for the condition:

$$t_j \leq 1 \ \ where\ t_j = x_j^2 + y_j^2$$

If true, the independent Gaussian deviates are computed:

$$X = x_j\sqrt{(-2\log t_j)/t_j}$$

$$Y = y_j\sqrt{(-2\log t_j)/t_j}$$

Then the new pair (X, Y) is tested to see if it falls within one of the 10 square annuli and a total count is tabulated for each annulus.

$$l \leq max(|X|, |Y|) < l + 1 \ \ where\ 0 \leq l < 9$$

The pseudo-random numbers are generated according to the following linear congruential recursion:

$$x_k = ax_{k-1} \bmod 2^{46} \ \ where\ a = 5^{15}, x_0 = 271828183$$

The values in a pair (x_j, y_j) are consecutive values of the recursion. To scale to the (0,1) range, the value x_k is divided by 2^{46}.

The computation for each pair of Gaussian deviates can proceed independently. Each processor would maintain its own counts of the Gaussian deviates and communicate at the end to obtain the global sum. The random number generation, however, presents a challenge. There are two ways to compute a random value x_k:

1. x_k can be computed quickly from the preceding value x_{k-1} using only one multiplication and one *mod* operation, leading to a complexity of $O(n)$. However, the major drawback is the true data dependence on the value x_{k-1}.

2. x_k can be computed independently using k and the defined values of a and x_0. This will result in an overall complexity of $O(n^2)$. Fortunately, the property of the *mod* operation allows x_k to be computed in $O(\log k)$ steps by using a binary exponentiation algorithm [4].

The goal then is to balance between method (1) and (2) to achieve parallelism while maintaining the $O(n)$ cost. Because EP is not available in the NPB2.1 suite, we use the implementation provided by APR as the DO loop version. This version is structured to achieve the balance between (1) and (2) by batching: the random values are generated in one sequential batch at a time and saved; the seed of the batch is computed using the more expensive method (2), and the remaining values are computed using the less expensive method (1). A DO loop then iterates to compute the number of batches required, and this constitutes the opportunity for parallel execution.

The F90/Forall version is derived from the DO loop version with the following modifications:

- All variables in the main DO loop that cause an output dependence are expanded into arrays of the size of the loop iteration. In other words, the output dependence is eliminated by essentially renaming the variables so that the computation can be expressed in a fully data parallel manner. Since the iteration count is just the number of sequential batches, the expansion is not excessive.
- Directives are added to partition the arrays onto a 1-D processor grid.
- The DO loop for the final summation is also recoded using the HPF reduction intrinsic.

A complication arises involving the subroutine call within the Forall loop, which must be free of side effects in order for the loop to be distributed. Some slight code rearrangement was done to remove a side effect in the original subroutine, then the PURE directives were added to assert freedom from side effects.

The ZPL version is translated in a straightforward manner from the DO loop version. The only notable difference is the use of the ZPL region construct to express the independent batch computations.

2.3.2 MG

Multigrid is interesting for several reasons.

First, it illustrates the need for data parallel languages such as HPF or ZPL. The NPB2.1 implementation contains over 700 lines of code for the communication – about 30% of the program – which are eliminated when the program is written in a data parallel language.

Second, since the main computation is a 27-points stencil, the reference pattern that requires communication is simply a shift by a constant, which results in a simple neighbor exchange in the processor grid. All compilers (ZPL and HPF) recognize this pattern well and employ optimizations such as message vectorization and storage preallocation for the nonlocal data [3, 8, 9, 12]. Therefore, although the benchmark is rather complex, the initial indication is that both HPF and ZPL should be able to produce efficient parallel programs.

The benchmark is a V-cycle multigrid algorithm for computing an approximate solution to the discrete Poisson problem:

$$\nabla^2 u = v$$

where ∇^2 is the Laplacian operator $\nabla^2 u = (\frac{\delta^2}{\delta x^2}, \frac{\delta^2}{\delta y^2}, \frac{\delta^2}{\delta z^2})$

The algorithm consists of 4 iterations of the following three steps:

$$\begin{array}{lll} r = v - A \cdot u & (1) & \textit{evaluate residual} \\ z = M^k r & (2) & \textit{compute correction} \\ u = u + z & (3) & \textit{apply correction} \end{array}$$

where A is the trilinear finite element discretization of the Laplace operator ∇^2, M^k is the V-cycle multigrid operator as defined in the NPB1 benchmark specification [1, 4].

The algorithm implemented in the NPB2.1 version consists of three phases: the first phase computes the residual, the second phase is a set of steps that applies the M^k operator to compute the correction while the last phase applies the correction.

The HPF DO loop version is derived from the NPB2.1 implementation as follows:

- The MPI calls are removed.
- The local loop bounds are replaced with the global bounds.
- The use of a COMMON block of storage to hold the set of hierarchical arrays (different sizes) is incompatible with HPF; therefore the arrays are renamed and declared explicitly.
- HPF directives are added to partition the arrays onto a 3-D processor grid. The array distribution is maintained across subroutine calls by using the transcriptive directives to prevent unnecessary redistribution.

The HPF F90/Forall version requires the additional step of rewriting all data parallel loops in F90 syntax.

The ZPL version has a similar structure to the HPF F90/Forall version, the notable difference being the use of *strided region* to express the hierarchy of 3-D grids. A strided region is a sparse index set over which data can be declared and computation can be specified.

2.3.3 FT

Consider the partial differential equation for a point x in 3-D space:

$$\frac{\delta u(x,t)}{\delta t} = \alpha \nabla^2 u(x,t)$$

The FT benchmark solves the PDE by (1) computing the forward 3-D Fourier Transform of $u(x,0)$, (2) multiplying the result by a set of exponential values, and (3) computing the inverse 3-D Fourier Transform. The problem statement requires 6 solutions, therefore the benchmark consists of 1 forward FFT and 6 pairs of dot products and inverse FFTs.

The NPB2.1 implementation follows a standard parallelization scheme [5, 2]. The 3-D FFT computation consists of traversing and applying the 1-D FFT along each of the three dimensions. The 3-D array is partitioned along the third dimension to allow each processor to independently carry out the 1-D FFT along the first and second dimension. Then the array is transposed to enable the traversal of the third dimension. The transpose operation constitutes most of the communication in the program. The program requires moving the third dimension to the first dimension in the transpose so that the memory stride is favorable for the 1-D FFT; therefore the HPF REDISTRIBUTE function alone is not sufficient[2].

The HPF DO loop implementation is derived with the following modifications:

1. HPF directives are added to distribute the arrays along the appropriate dimension. Transcriptive directives are used at subroutine boundaries to prevent unnecessary redistribution.
2. The communication for the transpose step is replaced with a global assignment statement.
3. A scratch array that is recast into arrays of different ranks and sizes between subroutines is replaced with multiple arrays of constant rank and size. Although passing an array section in a formal argument is legitimate in HPF, some HPF compilers have difficulty managing array sections.

[2]HPF data distribution specifies the partition to processor mapping, not the memory layout.

The HPF F90/Forall version requires the additional step of rewriting all data parallel loops in F90 syntax.

The ZPL implementation allocates the 3-D arrays as regions of 2-D arrays; the transpose operation is realized with the ZPL permute operator.

2.4 Platform

Program portability should be evaluated across multiple platforms of different compilers and machines. In this study, we elect to factor out the difference due to the machine architecture and focus on the compiler difference. Although the machine architecture sets the upper bound on the possible performance, a language and its compiler determine the achievable performance. The targeted parallel platform is the IBM SP2 at the Cornell Theory Center. The HPF compilers used in the study include:

- Portland Group pghpf version 2.1
- IBM xlhpf version 1.0
- Applied Parallel Research xhpf version 2.0

All compilers generate MPI calls for the communication and use the same MPI library, ensuring that the communication fabric is identical for all measurements. The PGI and APR HPF compilers generate intermediate Fortran code that is then processed by the standard IBM Fortran compiler (xlf). The IBM HPF compiler generates machine code directly but otherwise is based on the same xlf compiler. The ZPL compiler generates intermediate C code. All measurements use the same compiler options and system environment that NAS and APR specified in their publications (using SP2 wide nodes only), and spotchecks confirmed that the published NAS and APR performances are reproduced.

3 Parallel Performance

In this section we examine the performance of the programs. Figure 1 shows the aggregate timing for all versions (MPI, HPF, ZPL) and for the small and large problem size (class S, class A). Notes:

- Because the execution time may be excessive depending on the success of the compilers, we first examine the small problem size (class S), then only the programs with a reasonable performance and speedup with the large problem size (class A).
- The time axis uses a log scale because of the wide performance spread.
- For EP, an MPI version is not used; for class S, the F90/Forall version is not shown.
- For MG class A, the APR F90/Forall version does not scale and is not included.
- For FT class A, only one data point is available for the IBM F90/Forall version.

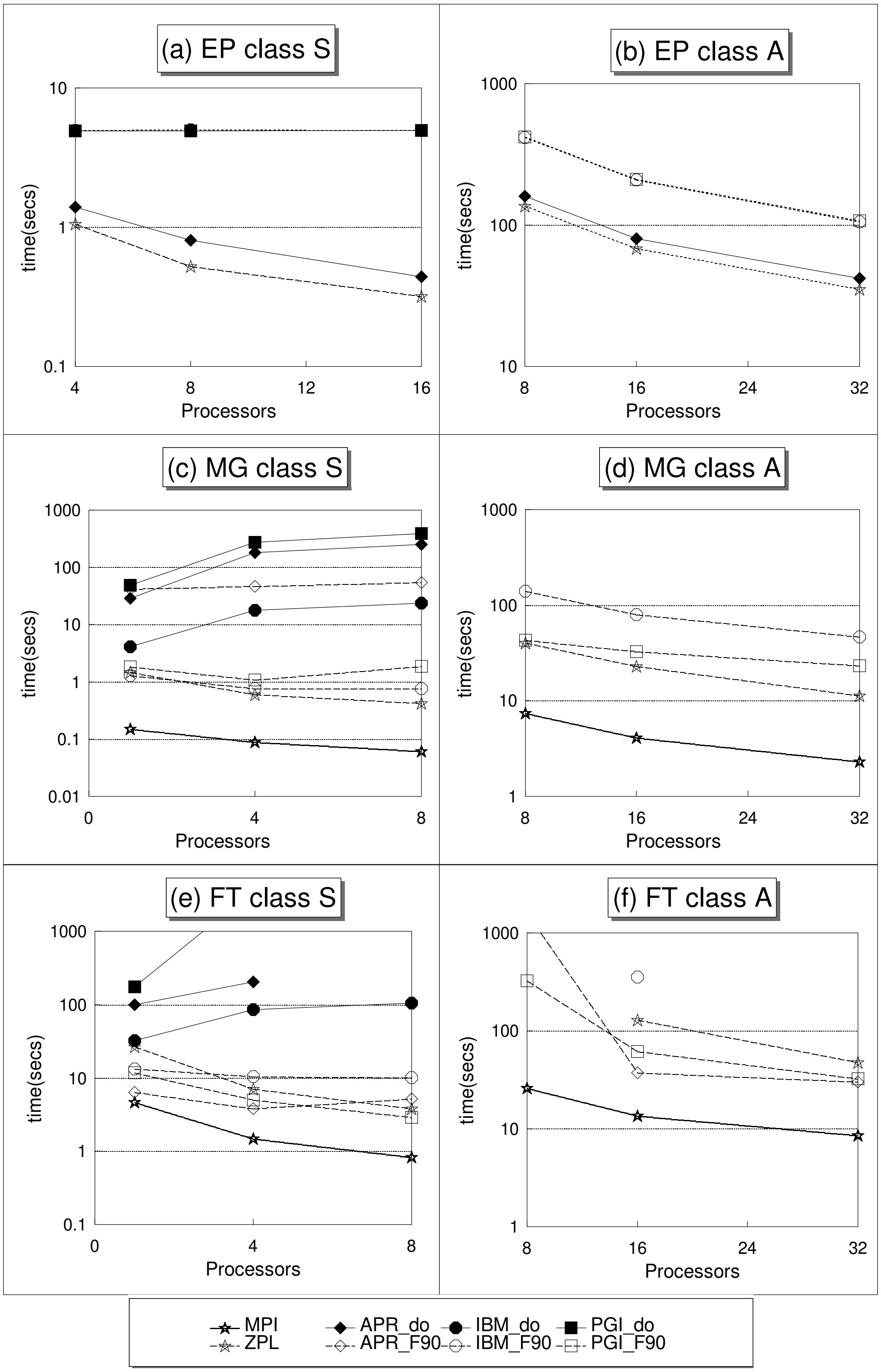

Figure 1: Performance for EP, MG and FT
(see notes in Section 3)

3.1 NAS EP benchmark

In Figure 1(a), the first surprising observation is that the IBM and PGI compilers achieve no speedup with the HPF DO loop version although the APR compiler produces a program that scales well (recall that the EP DO loop version is from the APR suite). Inspecting the code reveals that no distribution directives were specified for the arrays, resulting in a default data distribution. Although the default distribution is implementation dependent, the conventional choice is to replicate the array. The IBM and PGI compilers distribute the computation strictly by the owner-computes rule; therefore, in order for the program to be parallelized, some data structures must be distributed. Since the arrays in EP are replicated by default, no computation is partitioned among the processors: each processor executes the full program and achieves no speedup. By contrast, the APR parallelization strategy does not strictly adhere to the owner-computes rule. This allows the main loop to be partitioned despite the fact that none of the arrays within the loop are distributed. Note that the HPF language specification does not specify the default distribution for the data nor the partitioning scheme for the computation. The omission was likely intended to maximize the opportunity for the compiler to optimize; however the observation for EP suggests that the different schemes adopted by the compilers may result in a portability problem with HPF programs.

When directives were inserted to distribute the arrays, it was found that the main array in EP is intended to hold pseudo-random values generated sequentially, therefore there exists a true dependence in the loop computing the values. If the array is distributed, the compiler will adjust the loop bounds to the local partition, but the computation will be serialized.

The HPF F90/Forall version corrects this problem by explicitly distributing the arrays and the IBM and PGI compilers were able to parallelize. The class A performance in Figure 1(b) shows that all compilers achieve the expected linear speedup. However, expanding the arrays to express the computation into a more data parallel form seems to introduce overhead and degrade the scalar performance. It is possible for advanced compiler optimizations such as loop fusion and array contraction to remove this overhead, but these optimizations were either not available or not successful in this case.

The ZPL version scales linearly as expected and the scalar performance is slightly better than the APR version.

3.2 NAS MG benchmark

Compared to EP, MG allows a more rigorous test of the languages and compilers. We first discuss the performance for the class S in Figure 1(c).

The p=1 column shows considerable variation in the scalar performance with all versions showing overhead of 1 to 2 orders of magnitude over the MPI performance.

For the base cases, both the original MPI program and the ZPL version scale well. The ZPL compiler partitions the problem in a straightforward manner according to the region and strided region semantics, and the communication is vectorized with little effort. The scalar performance does however show over a 6x overhead compared to the MPI version.

The HPF DO loop version clearly does not scale with any HPF compiler, as explained below.

The PGI compiler performs poorly in vectorizing the communication when the computation is expressed with DO loops: the communication calls tend to remain in the innermost loop, resulting in a very large number of small messages being generated. In addition, the program uses guards within the loop instead of adjusting the loop bound.

The APR compiler only supports a 1-D processor grid, therefore the 3-D distribution specified in the HPF directives is collapsed by default to a 1-D distribution. This limitation affects the asymptotic speedup but does not necessarily limit the parallelization of the 27-point stencil computation. For one subroutine, the compiler detects through interprocedural analysis an alias between two formal arguments, which constitutes an inhibitor for the loop parallelization within the subroutine. However, the analysis does not go further to detect from the index expressions of the array references that no dependence actually exists. For most of the major loops in the program, the APR compiler correctly partitions the computation along the distributed array dimension, but generates very conservative communication before the loop to obtain the latest value for the RHS and after the loop to update the LHS. As a result, the performance degrades with the number of processors.

The IBM compiler does not parallelize because it detects an output dependence on a number of variables although the arrays are replicated. In this case, the compiler appears to be overly conversative in maintaining the consistency of the replicated variables. Other loops do not parallelize because they contain an IF statement.

The INDEPENDENT directive is treated differently by the compilers. The PGI compiler interprets the directive literally and parallelizes the loop as directed. The IBM compiler on the other hand ensures correctness by performing a more rigorous dependence check nevertheless and because of detected dependence, it does not parallelize the loop.

For the HPF F90/Forall version, the IBM and PGI compilers are more successful: the IBM compiler performance and scalability approach ZPL's, while the PGI compiler now experiences little problem in vectorizing the communication; indeed its scalar performance exceeds IBM's. The APR compiler does not result in slowdown but does not achieve any speedup either. It partitions the computation in the F90/Forall version similarly to the DO loop version, but is able to reduce the amount of communication. It continues to be limited by its 1-D distribution as well as an alias problem with one subroutine. Note that the version of MG from the APR suite employs APR's directives to suppress unnecessary communication. These directives are not used in our study because they are not a part of HPF, but it is worth noting that it is possible to use APR's tools to analyze the program and manually insert APR's directives to improve the speedup with the APR compiler.

Given that the DO loop version fails to scale with any compiler, one may conjecture whether the program may be written differently to aid the compilers. The specific causes for each compiler described above suggest that the APR compiler would be more successful if APR's directives are used, that the PGI compiler may benefit from the HPF INDEPENDENT directive, and that the IBM compiler would require actual removal of some data dependences. Therefore, it does not appear that any single solution is portable across the compilers.

3.3 NAS FT benchmark

FT presents a different challenge to the HPF compilers. In terms of the reference pattern, FT consists of a dot product and the FFT butterfly pattern. The former requires no communication and is readily parallelized by all compilers. For the latter, the index expression is far too complex to optimize the communication, but fortunately the index variable is limited to one dimension at a time; therefore the task for the compiler is to partition the computation along the appropriate dimensions. The intended data distribution is 1-D and is thus within the capability of the APR compiler.

Figure 1(e) shows the full set of performance results for the small problem size. As with MG, the MPI and ZPL versions scale well and the scalar performance of all data parallel implementations shows an overhead of 1 to 2 orders of magnitude over the MPI implementation.

For the HPF DO loop version, the APR compiler exhibits the same problem as with MG: it generates very conservative communication before and after many loops. In addition, the APR compiler does not choose the correct loop to parallelize. The discrepancy arises because APR's strategy is to choose the partitioning based on the array references within the loop. In this case the main computation and thus the array references are packaged in a subroutine called from the loop, the intention being that the loop is parallelized and the subroutine operates on the local data. When the compiler proceeds to analyze the loops in this subroutine (1-D FFT), it finds that the loops are not parallelizable.

The PGI compiler also generates suboptimal communication, although its principal limitation is in vectorizing the messages. The IBM compiler does not parallelize because of assignments to replicated variables.

The HPF F90/Forall version requires considerable experimentation and code restructuring to arrive at a version that is accepted by all compilers, partly because of differences in supported features among the compilers and partly because of the nested subroutines structure of the original program. All HPF compilers achieve speedup to varying degrees. APR is particularly successful since the principal parallel loop has been moved to the innermost subroutine. Its scalar performance approaches MPI performance, although communication overhead limits the speedup. PGI shows good speedup while IBM's speedup is more limited.

3.4 Communication

The communication generated by the compilers is a useful indication of the effectiveness of the parallelized programs. For the versions of the benchmarks that scale, table 1 shows the total number of MPI message passing calls and the differences in the communication scheme employed by each compiler. The APR and PGI compilers only use the generic send and receive while the IBM compiler also uses the the nonblocking calls and the collective communication; this may have ramifications in the portability of the compiler to other platforms. The ZPL compiler uses nonblocking MPI calls to overlap computation with communication as well as MPI collective communication.

Benchmark	version	point-to-point	collective	type of MPI calls
EP (class A)	ZPL	0	120	Allreduce, Barrier
	APR DO loop	31	0	Send, Recv
	IBM F90	70	120	Send, Recv, Bcast
	PGI F90	240	0	Send, Recv
MG (class S)	MPI	2736	40	Send, Irecv, Allreduce, Barrier
	ZPL	9504	56	Isend, Recv, Barrier
	APR F90	126775	8	Send, Recv, Barrier
	IBM F90	9636	32	Send, Recv, Irecv, Bcast
	PGI F90	22191	0	Send, Recv
FT (class S)	MPI	0	104	Alltoall, Reduce
	ZPL	1064	32	Isend, Recv, Barrier
	APR F90	58877	8	Send, Recv, Barrier
	IBM F90	728	258048	Send, Irecv, Bcast
	PGI F90	64603	0	Send, Recv

Table 1: Dynamic communication statistics for EP class A, MG class S and FT class S: p=8

3.5 Data Dependences

HPF compilers derive parallelism from the data distribution and the loops that operate on the data. Loops with no dependences are readily parallelized by adjusting the loop bounds to the local bounds. Loops with dependences may still be parallelizable but will require analysis; for instance, the IBM compiler can recognize the dependence in a loop that performs a reduction and generate the appropriate HPF reduction intrinsic. In other instances, loop distribution may isolate the portion containing the dependence to allow the remainder of the original loop to be parallelized. To approximately quantify the degree of difficulty that a program presents to the parallelizing compiler in terms of dependence analysis, we use the following simple metric:

$$\frac{count\ of\ all\ loops\ with\ dependences}{count\ of\ all\ loops}$$

A value of 0 would indicate that all loops can be trivially parallelized, while a value of 1 would indicate that any parallelizable loops depend on the analysis capability of the compiler. Using the KAPF tool, we collect the loop statistics from the benchmarks for the major subroutines; they are listed in Table 2. This metric is not complete since it does not account for the data distribution; for instance, for 3 nested loops and a 1-D distribution, only 1 loop needs to be partitioned to parallelize the program and 2 loops may contain dependences with no ill effects. In addition, this metric is static and may not correlate directly with the dynamic characteristics of the programs. However, the metric gives a coarse indication for the demand on the compiler.

The loop dependence statistics show clear trends that correlate directly with the performance data. We observe the expected reduction in dependences from the DO loop version to the F90/Forall version. The reduction greatly aids the compilers in parallelizing the F90/Forall programs, but also highlights the difficulty with parallelizing programs with DO loops.

	subroutine	DO	F90/Forall
EP	embar	3/5	1/31
	vrandlc	1/1	-
	get_start_seed	-	1/1
FT	fftpde	2/16	2/16
	cfft3	0/6	0/6
	cffts1	2/6	1/7
	cffts2	2/6	1/7
	cfftz	3/5	1/4
	fftz2	3/3	3/4

	subroutine	DO	F90/Forall
MG	hmg	1/2	1/27
	mg3P_up	1/1	1/22
	mg3P_down	1/1	1/1
	psinv	4/4	0/6
	resid	4/4	0/6
	rprj3	4/4	0/6
	interp	7/21	0/30
	norm2u3	3/3	0/0
	comm3	0/6	0/18

Table 2: Statistics on dependences for EP, MG and FT: $\frac{m}{n}$, where m is the count of loops with data dependences or subroutine calls, and n is the total loop count; for F90/Forall, the counts are obtained after the array statements have been scalarized

For MG, the difference is significant; the array syntax eliminates the dependence in most cases. Some HPF compilers implement optimizations for array references that are affine functions of the DO loop indices, particularly for functions with constants. These optimizations should have been effective for the MG DO loop version, however it does not appear that they were successful. Note that the loops in the subroutine **norm2u3** are replaced altogether with the HPF reduction intrinsics.

For FT, the low number of dependences in **fftpde** comes from the dot-products which are easily parallelized. The top-down order of the subroutines listed also represents the nesting level of the subroutines. The increasing dependences in the inner subroutine reflect the need to achieve parallelism at the higher level. As explained earlier, this proves to be a challenge to the APR compiler which focuses on analyzing individual loops to partition the work.

4 Conclusion

For data parallel applications, the recent progress in languages and compilers allows us to experimentally evaluate an important issue: program portability. We recognize that many factors affect the development and success of a parallel language and our study only focuses on the portability factor.

Three NAS benchmarks were studied across three current HPF compilers. We examined different styles of expressing the computation in HPF and we also consider the same benchmarks written in MPI and ZPL to understand the interaction between performance, portability and convenient programming.

The HPF compilers show a general difficulty in detecting parallelism from DO loops. The compilers are more successful with the F90 array syntax and Forall construct, although even in this case the success in parallelization is not uniform. Significant variation in the scalar performance also exists between the compilers.

While the HPF directives and constructs provide information on the data and computation partitioning, the sequential semantics of Fortran leave many potential dependences in the program. An HPF compiler must analyze these dependences, and when unable to do so, it must make a conservative assumption. This analysis capability differentiates the various vendor implementations. However, because it is difficult for the compilers to parallelize reliably, a user cannot consistently estimate the parallel behavior and thus the speedup of the program. In addition, because the parallelization strategy by the compilers varies widely, different ways to express the same computation can lead to drastically different performance. The unpredictable variations reflect a shortcoming in the performance model of HPF; as a result, a user needs to continually experiment with each compiler to learn the actual behavior. In doing so, the user is effectively supplementing the performance model provided by the language with empirical information. Yet, such an enhanced model tends to be platform specific and not portable.

ZPL programs show consistent scalable performance, illustrating that it is possible to incorporate a robust performance model in a high level language. The language design ensures that the language abstractions behave in a predictable manner with respect to parallel performance. Although ZPL is supported on multiple parallel systems, the results in this study do not directly show ZPL's portability across platforms because multiple independent compiler implementations for ZPL are not available. However, the existence of the performance model, evident in the predictable performance behavior, ensures that ZPL programs will be portable across independently developed platforms.

The results also show that significant overhead in the scalar performance remains in all implementations compared to the MPI programs. One source for the overhead is the large number of temporary arrays generated by the compiler across subroutine calls and parallelized loops. They require dynamic allocation/deallocation and copying, and generally degrade the cache performance. The index computation also contributes significantly to the overhead. It is clear that to become a viable alternative to explicit message passing, compilers for data parallel languages must achieve a much lower scalar overhead.

References

[1] Ramesh Agarwal, Bowen Alpern, Larry Carter, Fred Gustavson, David Klepacki Rick Lawrence, and Mohammad Zubair. High performance parallel implementation of the NAS kernel benchmarks on the IBM SP2. *IBM Systems Journal*, 34(2):263–272, 1995.

[2] Ramesh Agarwal, Fred Gustavson, and Mohammad Zubair. An efficient parallel algorithm for the 3-D FFT NAS parallel benchmark. In *Proceedings of SHPCC 1994*, pages 129–133. IBM Thomas J. Watson Research Center, 1994.

[3] Applied Parallel Research. *XHPF User's Guide*, version 2.0 edition, January 1995.

[4] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Finebertg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, CA, March 1991.

[5] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmark 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[6] S. Benkner, Barbara Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of Scalable High Performance Computing Conference 1992*, pages 51–59. Department for Statistics and Computer Science, Vienna University, Austria, 1992.

[7] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, Sanjay Ranka, and Min-You Wu. Compiling Fortran 90D/HPF for distributed memory mimd computers. *Journal of Parallel and Distributed Computing*, 21:15–26, 1994.

[8] Zeki Bozkus, Larry Meadows, Steven Nakamoto, Vincent Schuster, and Mark Young. Compiling High Performance Fortran. In *Proceedings ofthe Seventh SIAM Conference on Parallel Processing for Scientific Computing.* The Portland Group, Inc., 1995.

[9] Bradford Chamberlain, Sung-Eun Choi, E Lewis, Calvin Lin, Lawrence Snyder, and Derrick Weathersby. Factor-Join: A unique approach to compiling array languages for parallel machines. In *9th Workshop on Languages and Compilers for Parallel Computing*, pages 481–500. University of Washington, August 1996.

[10] High Performance Fortran Forum. HPF language specification version 1.0. Technical Report CRPC-TR92225, Rice University, May 1993.

[11] Richarcd Friedman, John Levesque, and Gene Wagenbreth. Fortran parallelization handbook. Technical report, Applied Parallel Research, Sacramento, CA, April 1995.

[12] Manish Gupta, Sam Midkiff, Edith Schonberg, Ven Seshadri, David Shields, KoYang Wang, Waimee Ching, and Ton Ngo. An HPF compiler for the IBM SP2. In *Supercomputing 1995*, San Diego, December 1995. IBM T. J. Watson Research Center, IEEE.

[13] Jonathan Harris, John Bircsak, Regina Bolduc, Jill Diewald, Israel Gale, Neil Johnson, Shin Lee, Alexander Nelson, and Carl Offner. Compiling High Performance Fortran for distributed-memory systems. *Digital Technical Journal*, 7(3):5–38, 1995.

[14] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Evaluating compiler optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21:27–45, 1994.

[15] Calvin Lin. ZPL language reference manual. Technical report, University of Washington, 1994.

[16] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In *Languages and Compilers for Parallel Computing.* 1993.

[17] Calvin Lin, Lawrence Snyder, Ruth Anderson, Brad Chamberlain, Sung-Eun Choi, George Forman, E Lewis, and Derrick Weathersby. ZPL vs. HPF: A comparison of performance and programming style. Technical Report UW-CSE-95-11-05, University of Washington, Seattle, Wa 98195, November 1995.

[18] Ton Ngo. *The Role of Performance Models in Parallel Programming and Languages.* PhD thesis, University of Washington, June 1997.

[19] Ton Ngo and Lawrence Snyder. On the influence of programming models on shared memory computer performance. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, pages 284–291. Dept of Computer Science and Engineering, University of Washington, Seattle, Wa, May 1992.

[20] William Saphir, Alex Woo, and Maurice Yarrow. NAS parallel benchmark 2.1 results: 8/96. Technical Report NAS-96-010, NASA Ames Research Center, Moffett Field, CA, December 1995.

[21] Lawrence Snyder. A ZPL programming guide. Technical report, University of Washington, 1994.

[22] Thinking Machines Corporation. *CM Fortran Programming Guide*, version 2.2 edition, October 1994.