ZPL *vs* HPF: A Comparison of Performance and Programming Style*

Calvin Lin
Lawrence Snyder
Ruth E. Anderson
Bradford L. Chamberlain
Sung-Eun Choi
George Forman
E Christopher Lewis
W. Derrick Weathersby

Department of Computer Science and Engineering
Box 352350
University of Washington
Seattle, WA 98195-2350

August 1, 1995

Abstract

This paper compares two data parallel languages, ZPL and HPF, in terms of programming style and performance. The results show that for eight programs from a number of standard benchmark suites, ZPL generally outperforms HPF, and ZPL expresses problems at higher levels of abstraction, yielding programs that are shorter, less error prone and easier to maintain. ZPL's better performance comes from its clean expression of parallelism that allows for better compiler analysis.

1 Introduction

ZPL is a new data parallel array language designed at the University of Washington to provide high performance parallel computation for all MIMD parallel computers [7]. HPF was recently defined by the High Performance Fortran Forum to provide "data parallel programming" and "top performance on MIMD and SIMD computers" [4, p.1]. With such similar ambitions, it is natural to compare the two languages. This paper compares ZPL and HPF, concentrating on performance and programming style.

Though their high-level objectives are similar, the languages have dissimilar design goals:

> **ZPL Design Goal:** to design from first principles a language maximizing performance and portability. Indeed, ZPL is the data parallel subset of a more general parallel language, Advanced ZPL,[1] which is based on an efficient parallel machine model and programming model [10] with demonstrated performance and portability [6].

*This research was supported in part by ARPA Grant N00014-92-J-1824 and NSF Grant CDA-9211095

[1] Advanced ZPL is the new name for Orca C. This change has been made to avoid confusion with a similarly named language for distributed computing.

HPF Design Goal: "to develop extensions to Fortran that provide support for high performance programming on a wide variety of machines ..." [4, p.*vii*]. The HPF Forum included groups with large installed Fortran bases, vendors intending to support Fortran on their parallel hardware, and researchers developing compilation technology for Fortran sources.

Thus, ZPL is a new language focused on the best practical way to express parallelism, while HPF is a new language extension focused on the best practical way to extract performance—particularly parallelism—from Fortran. There are numerous effects that these two design goals have had on their respective languages, but one is immediate: To use ZPL, it is necessary to write a new program; to use HPF, it may be possible to embellish an existing Fortran program.

Since HPF provides Fortran compatibility while ZPL does not, is a comparison of these languages fair or even sensible? Yes, for the following reason: For problems where no Fortran program exists, either ZPL or HPF can be used to solve the problem. *The comparison presented here thus applies to the case of writing new parallel programs.* The evidence presented will favor ZPL over HPF, for the programs will likely run faster and be easier to understand and maintain. As a crude measure of readability, the SIMPLE benchmark is 2400 lines of Fortran 77 without HPF directives but only 500 lines in ZPL [7]. The discussion in Section 4 illustrates this point more clearly.

In cases where a Fortran program exists, our experience suggests that the effort to understand it well enough to select the proper HPF annotations can be comparable to the effort required to rewrite it in ZPL. Moreover, after adding HPF directives, considerable effort can be required to restructure a Fortran program to get good parallel performance [3]. After this effort, the HPF approach is finished, if the performance is satisfactory, while the ZPL program must be validated to show it equivalent to the original Fortran. Tools can assist in this validation [1], and the additional effort is generally repaid in better performance and a cleaner, more easily maintained program.

2 A Brief Introduction to HPF

HPF was designed by the High Performance Fortran Forum in 1993 [4]. HPF extends sequential Fortran by adding data decomposition directives that do not change the semantics of the program, but serve as hints to the compiler to improve performance. Thus HPF provides data parallelism by adding data decomposition information to sequential programs. HPF programmers can also use the `FORALL` statement to identify loops that can execute in parallel. HPF directives can be added to programs written in either Fortran 77 or Fortran 90. In this paper, we refer to programs written in either Fortran 77 or Fortran 90 simply as HPF programs.

To illustrate the basics of the language, Figure 1 shows fragments of HPF code. The declaration of the array `a` is shown followed by a data distribution directive. This directive, prefixed by the special characters `HPF$`, will have the effect of allocating a rectangular block of the array to each processor. Following are two alternative ways of updating the elements of an array with the average of their four nearest neighbors. In the first case the **FORALL** statement is used to direct that all iterations can be performed in parallel. In the second case array indices are used, which specify the portion of the array that is to be referenced in the computation. Both schemes allow the programmer to specify computation at a larger granularity than is possible in basic F77. Larger units of computation are more efficiently performed in parallel.

Further examples of HPF program segments are given below.

```
REAL a(0:n+1,0:n+1)
!HPF$ DISTRIBUTE a(BLOCK,BLOCK)
!
! The Jacobi kernel
!
! Variant 1: use FORALL statement
FORALL (i=1:n, j=1:n)   a(i,j) = (a(i-1,j)+a(i+1,j)+a(i,j-1)+a(i,j+1))/4
!
! Variant 2: use array assignment
!
a(1:n, 1:n) = (a(0:n-1,1:n) + a(2:n+1,1:n) + a(1:n,0:n-1) + a(1:n,2:n+1))/4
```

Figure 1: HPF program fragments for the Jacobi iteration.

3 A Brief Introduction to ZPL

ZPL is an array language with the standard data types (e.g. `real`, `integer`, `char`), dense arrays of arbitrary dimension, the usual arithmetic and logical operators which can be applied to either scalars or point-wise to arrays, parallel prefix operators (e.g. reduction and scan), and the standard control structures (`if`, `for`, `while`, *etc.*), including recursion. The language has sequential semantics. There are *no* parallel directives, and all concurrency is derived by the compiler from the semantics of the language constructs. Our current compiler is implemented as a source-to-source translator that produces machine independent ANSI C code, which is then compiled on the target machine (via the native C compiler) and linked with a machine-specific library to produce executable code. Performance that matches hand-coded, explicitly parallel C code has been demonstrated on both shared and distributed memory parallel computers [8].

Program Walk-through. Perhaps the quickest introduction is to walk-through a ZPL program. Figure 2 shows the ZPL code for the 4-point Jacobi computation [11]. The program begins by defining two problem-specific *configuration variables*, `n` and `epsilon`, and their default values (line 2-3). These defaults can be overridden at load time from the command line.

Next, a *region* is declared (line 5). *Regions* are rectilinear sets of array indices that are fundamental to ZPL programming. In line 13 the region specifier `[R]` is used to declare two `n` $\times$ `n` arrays of `real`'s. Specifically, the `A` and `Temp` arrays are declared to have the indices {`1..n`} $\times$ {`1..n`}. In line 16 the region specifier `[R]` defines the indices over which computation takes place: The elements of `A` are initialized to `0.0` for the indices in `R`. Naming regions in this manner is a syntactic convenience which simplifies compiler analysis because the regions do not change during a program's execution. ZPL also supports *dynamic regions*

```
 1  program Jacobi;
 2  config var         n: integer = 512;             -- Configuration defaults
 3                     epsilon: real = 0.000001;
 4
 5  region             R = [1..n, 1..n];             -- Declarations
 6
 7  direction          north   = [-1, 0];
 8                     east    = [ 0, 1];
 9                     west    = [ 0,-1];
10                     south   = [ 1, 0];
11
12  procedure Jacobi();
13  var                A, Temp: [R] real;
14                     err: real;
15  begin
16    [R]              A := 0.0;                     -- Initialization
17    [north of R]     A := 0.0;
18    [east  of R]     A := 0.0;
19    [west  of R]     A := 0.0;
20    [south of R]     A := 1.0;
21
22    [R]              repeat                        -- Body
23                         Temp := (A@north+A@east+A@west+A@south)/4.0;
24                         err := max\abs(A-Temp);
25                         A := Temp;
26                     until err < epsilon;
27    [R]              writeln(A);                   -- Print the result
28  end;
```

Figure 2: ZPL program for the Jacobi iteration.

in which the region's size can change at runtime. For example, `[1..n, i]` is a region consisting of the $\mathtt{i}^{th}$ column of an index space, where the value of `i` is bound at runtime.

Line 17 shows how boundary conditions can be initialized using the `of` operator to refer to neighboring regions. Using the direction vector `north`, defined to be `[-1, 0]` in line 7, `[north of R]` is a region whose indices are disjoint from `R` and offset from it by `north`, i.e. `[north of R] = [0, 1..n]`. (More precise definitions can be found in the literature [5].) Line 17 not only initializes the northern border of `A`, but also implicitly allocates storage for this border region.

The program body computes the Jacobi iteration using a repeat statement (lines 22-26). Since the statement is prefixed by the region specifier `[R]`, all array operations within it apply to indices in `R`. The averaging of each element's four nearest neighbors is accomplished in line 23 using the `@` operator. `A@north` is an array of the same size and shape as `R` but with indices that are computed by adding the `north` vector to each index, i.e. `A@north` includes the $\mathtt{0}^{th}$ row of `A` but not the $\mathtt{n}^{th}$ row of `A`. The scalar operations in line 23 are applied to corresponding elements of each array. Thus, the statement

```
Temp := (A@north + A@east + A@west + A@south) / 4.0;
```

succinctly produces the 4-point stencil average for all elements in `A`. Line 24 shows the application of the scalar subtraction operation to array operands (`A - Temp`), the promotion of a scalar procedure (`abs()`) that is applied to each element of an array, and the use of the maximum reduction operator (`max\`) to find the largest element of an array within a region. Notice that code which uses scalars within the scope of a region, e.g. the `until` clause (line 26), is unaffected by that region and operates in the usual (sequential) manner. Many interesting and powerful features of the language are omitted for the sake of brevity. For more details, see the *ZPL Programmer's Guide* [11] and the *ZPL Reference Manual* [5].

Parallel Execution. Arrays declared over regions are called *parallel arrays* because they are the mechanism for achieving parallelism. In the ZPL programming model, parallelism is not expressed via loops [10]. Instead, ZPL derives concurrency from parallel array and parallel prefix operations. By default, parallel arrays are distributed in a block allocation for 1D and 2D arrays; higher dimensional arrays are distributed across two dimensions. Thus, to increment the elements of a parallel array `A`, for example, one simply writes `A := A+1`. Indexing is neither needed nor allowed[2], and full concurrency is achieved as each processor increments its block of `A`.

By adopting a clear definition of which constructs do and do not provide concurrency, and guaranteeing that they are executed as defined, ZPL programmers have an unambiguous picture of the parallelism in their code. This is not the case in HPF, where obscure details of the source program can disable parallelization by the compiler [3]. It is for this reason that Fortran compilers must often be accompanied by parallelization tools designed to help the programmer restructure their code for parallelism. Such parallelization tools are not needed in ZPL because the parallelism is clear from the source program.

The Context. As noted earlier, ZPL is the data parallel subset of a more general MIMD parallel language, Advanced ZPL, which is now being implemented. (Since ZPL is self-contained, it is reasonable to treat ZPL as a stand-alone language.) As a sublanguage of Advanced ZPL, ZPL derives two crucial benefits:

(a) Any features not available in ZPL can be written in Advanced ZPL and included in the program as required. This allows for a simple ZPL language design, without great concern for "completeness." For example, ZPL provides a default "block" partitioning of arrays. Advanced ZPL provides full control over memory allocation, so this default can be overridden.

(b) Advanced ZPL has been created from first principles, and has a well specified machine model (the CTA) and a well specified programming model (Phase Abstractions) [10]. These foundations have simplified the development of ZPL and the construction of its machine independent compiler.

Clearly, advantage (a) awaits completion of the Advanced ZPL compiler. Advantage (b) is the topic of the remainder of this paper.

[2]ZPL also provides indexable arrays, called *indexed arrays*, that are replicated on each processor (as are scalars) and thus are not a source of concurrency.

4 Differences In Programming Style

Though both ZPL and HPF are data parallel languages, programs written for the same task in each language can be more than just syntactically different, for the languages employ different conceptual mechanisms to express the parallelism. This section compares ZPL and HPF code fragments of three programs from the Applied Parallel Research (APR) xHPF Benchmark Suite [9]:

- Relaxation loop from PDE1—illustrates very similar approaches, but ZPL expresses it more mnemonically.
- Invocation of vrand() from Embar—ZPL uses direct data parallelism, while in HPF, a serial loop is parallelized.
- Boundary updates from Shallow—another case where ZPL raises the conceptual level above Fortran 90's array assignments.

PDE1. The PDE1 benchmark is a 3D Poisson solver using red/black successive over-relaxation written in the Fortran 90 style of HPF. This iterative stencil calculation is a classic example of a computation that is easily parallelized.

The inner loops of the HPF and ZPL versions of PDE1 are given in Figure 3. There are strong similarities between the two: Both loop from `1` to `iter`, and both use masking (`with/without` in the ZPL, `where/elsewhere` in the HPF) to select the proper values of array `U` to update. What is striking is the clarity of the ZPL code in comparison to the HPF.

```
   DO NREL=1,ITER
     WHERE(RED(2:NX-1,2:NY-1,2:NZ-1))
!
!  RELAXATION OF THE RED POINTS
!
    U(2:NX-1,2:NY-1,2:NZ-1) =                       &
  &            FACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+ &
  &   U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+ &
  &   U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+ &
  &   U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))

    ELSEWHERE
!
!  RELAXATION OF THE BLACK POINTS
!
    U(2:NX-1,1:NY-2,2:NZ-1) =                       &
  &            FACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+ &
  &   U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+ &
  &   U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+ &
  &   U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))

     END WHERE
   ENDDO
```

```
for nrel := 1 to iter do

                    /* Relax the red points */
  [I with Red]    U := factor*(hsq*F+U@top+U@bot+U@left+
                                U@right+U@front+U@back);

                    /* Relax the black points */
  [I without Red] U := factor*(hsq*F+U@top+U@top+U@left+
                                U@right+U@front+U@back);
end;
```

Figure 3: Red/black relaxation loop from HPF and ZPL versions of PDE1.

Much of the clarity comes from the region construct. The region specifier `I` in ZPL encapsulates the same information as the triple, `2:NX-1, 2:NY-1, 2:NZ-1`, which specifies an array slice in HPF, except the region applies to all arrays in a statement while the slice must be specified for each individual array. Because this "interior" region is used throughout the computation, it is both convenient and conceptually simplifying to declare it once and use it symbolically thereafter.

The improved readability of the ZPL code reduces the potential for errors. In the ZPL solution *different things look different*—in this case using different direction and region names—whereas all of the HPF slices look similar except for easily obscured details such as a `-2` or a `-3`. To illustrate this point, *an error has been introduced into the "black half" of each loop.* How difficult is each error to find?

Embar. Embar is a NAS parallel benchmark kernel that "generates pairs of Gaussian random deviates (according to a specific scheme) and tabulates the number of pairs in successive square annuli" [2]. Embar is considered "embarrassingly parallel" because it requires very little communication. This benchmark is interesting as an upper bound on FLOPS rates for parallel computers.

Of interest here is the way in which ZPL and HPF specify the computation's "unlimited" concurrency. Embar generates pseudo-random numbers in independent batches, with the $\texttt{i}^{th}$ batch ($\texttt{1} < \texttt{i} \leq \texttt{nn}$) seeded by a number of the form `2*(i-1)*nk`, for an instance-specific constant `nk`.

In the HPF version of Embar (written in the Fortran 77), the subroutine that produces a batch of random pairs, `vrand`, is called in a loop

```
DO 500 i = 1 , nn
  CALL vrand((i - 1) * nk * 2, x, 2 * nk, ...)
    . . .
  gc = gc + 1
    . . .
500 CONTINUE
```

Each call fills the array `x` with `2*nk` random numbers. This code is logically sequential, but by adding a directive to the compiler, separate calls can be executed concurrently. In APR HPF, the compiler-specific directive, `capr$ do par`, precedes the loop. More generally, HPF would identify this "loop parallelism" by the `INDEPENDENT` directive to assert that the loop's iterations could be executed concurrently. Since different iterations would all be assigning to the same array `x`, the `NEW` modifier would have to be included to specify additional storage for `x` (and other variables) for each concurrent iteration. This does not quite solve the problem, since there is some tabulation code, e.g. `gc = gc+1`, following the call to accumulate counts which *does* require interaction among the iterations. Special attention (beyond the scope of this discussion) is required to ensure that the concurrent execution preserves the intended semantics.

To achieve concurrent execution of the batches in ZPL, the programmer simply sets up a one dimensional region

```
region B = [1..nn];
```

corresponding to the number of batches planned, declares arrays

```
var     Seed: [B] integer;      -- seed for each batch
        Gc:   [B] integer;      -- global count of "good" pairs in each batch
```

to have that number of items, and initializes `Seed` with the proper values

```
[B]     Seed := 2 * (Index1-1) * nk;
```

This array expression uses the compiler generated constant `Index1`, which is an array containing the indices of `B`, that is, `1`, `2`, . . ., `nn`, to produce the batch initializations. Now, the `RandPairs()` procedure, which has been written to expect scalar arguments, is applied to the array `Seed`

```
[B]     Gc := RandPairs(Seed, nk, ...)
```

causing `RandPairs()` to be called in parallel for all batches. An array of values, being the independent contributions to the tabulation of each parallel call, is returned by the function into `Gc`. These values are simply added by a sum reduction, `gc:=+\Gc`, to complete the computation.

Thus, ZPL achieves concurrency by directly applying the principles of data parallelism, while the HPF solution decorates the original program to direct the compiler to perform the loop iterations concurrently. In HPF, careful attention must be paid to memory usage to ensure that concurrent execution of separate iterations does not cause unintended interactions but *does* cause the intended interactions.

Shallow. The Shallow benchmark, a finite-difference calculation to predict weather using the shallow-water equations, illustrates how ZPL and HPF handle periodic boundary conditions. Here, the ZPL code is compared to a Fortran 90 style HPF program from APR [9]. This program contains arrays with periodic boundaries, so each array is allocated an extra row and column that is kept equal to the row or column on the opposite edge.

Figure 4 shows the HPF and ZPL code to update the boundaries of `m+1` $\times$ `n+1` arrays. Each of the first six lines of the HPF code (on the left) copies the first row of an array to the last row of the same array. The next six lines copy the first column to the last column, and the last six lines copy the upper left corner item to the lower right. The right side of Figure 4 shows how ZPL uses the `wrap` statement to copy items from one side of an array to the other. For example, given `region I = [1..m, 1..n]` and direction `east = [0,1]`, the statement

```
[east of I] wrap U;                  -- copy first column into last column
```

assigns to the region `[east of I]` (that is, `[1..m, n+1]`) the data from the same-sized region on the opposite end of the array, namely, the region `[1..m, 1]`. The programmer's intent is evident from the text, reducing the chance for error. Thus, ZPL raises the level of abstraction by providing a direct solution for periodic (and mirrored, using `reflect`) boundary computations. This is another instance where ZPL programmers define names for indices once and thereafter need not worry about getting them right.

```
       uold(m + 1,:n) = uold(1,:n)
       vold(m + 1,:n) = vold(1,:n)
       pold(m + 1,:n) = pold(1,:n)
       u(m + 1,:n) = u(1,:n)
       v(m + 1,:n) = v(1,:n)
       p(m + 1,:n) = p(1,:n)

 CAPR$ DO PAR on POLD<:,1>
       uold(:m,n + 1) = uold(:m,1)
       vold(:m,n + 1) = vold(:m,1)
       pold(:m,n + 1) = pold(:m,1)
       u(:m,n + 1) = u(:m,1)
       v(:m,n + 1) = v(:m,1)
       p(:m,n + 1) = p(:m,1)

       uold(m + 1,n + 1) = uold(1,1)
       vold(m + 1,n + 1) = vold(1,1)
       pold(m + 1,n + 1) = pold(1,1)
       u(m + 1,n + 1) = u(1,1)
       v(m + 1,n + 1) = v(1,1)
       p(m + 1,n + 1) = p(1,1)
```

```
/* Periodic Continuation */
[south of I]  wrap U, Uold, V, Vold, P, Pold;
[east  of I]  wrap U, Uold, V, Vold, P, Pold;
[se of I]     wrap U, Uold, V, Vold, P, Pold;
```

Figure 4: Setting boundary conditions in HPF and ZPL versions of Shallow.

5 Performance Comparison

ZPL has been used to solve real-world problems including a hierarchical fast N-body solver, a synchronous routing simulator, a conjugate gradient program for sparse matrices, and a simulation of bacteria formation, but for these programs direct comparisons with HPF are difficult to obtain. Thus, we use the APR xHPF Benchmark Suite [9]—currently the only published set of comprehensive HPF performance results—as the basis for comparison.

5.1 Methodology

Of the fifteen benchmarks in the xHPF suite, we compare ZPL results against eight of the top HPF performers on the Intel Paragon and the Cray T3D (See Table 1). To produce comparable results, we translated each of the benchmarks to ZPL as directly as possible, in each case timing the same portions of code, consuming the same inputs, and producing the same outputs as the original HPF programs.[3] The HPF performance numbers are those published in March, 1995 by Applied Parallel Research (APR) for their commercial HPF compiler [9].

The ZPL results were gathered on the San Diego Supercomputing Center's (SDSC) Paragon and the Arctic Region Supercomputing Center's (ARSC) Cray T3D, while the HPF results used the Paragon at NASA's Ames Research Center (Ames) and the Cray T3D at the Pittsburgh Supercomputing Center (PSC). The machines are similarly configured, as shown in Table 2.

[3]Interestingly, the HPF version of PDE1 contains a bug—the mask is initialized to consist of parallel red and black planes instead of a 3D checkerboard—reinforcing our complaint that HPF's slice notation is error-prone.

name	*origin*	*description*	*nature of communication*
Embar	NAS	Embarrassingly parallel	Global Sum
Grid	APR	Grid relaxation	Nearest Neighbor
Ora	SPEC	Ray tracing	Global Sum
PDE1	GENESIS	3D red/black SOR	Nearest Neighbor
Scalgam	LANL	Monte Carlo transport	Global Sum
Shallow	NCAR	2D finite difference calculation	Nearest Neighbor
Swm256	SPEC	Version of Shallow	Nearest Neighbor
Tomcatv	SPEC	2D Grid generation	Nearest Neighbor
X42	APR	Fourth order differencing	Nearest Neighbor

Table 1: Summary of Benchmarks

Machine	nodes	CPU	clock rate	memory/node	OS
ARSC T3D	128	Alpha	150MHz	64 MB	UNICOS 8.0.2.2/MAX 1.2.0.2
PSC T3D	512	Alpha	150MHz	64 MB	UNICOS 8.0.3/MAX 1.2.0.3
SDSC Paragon	416	i860	50MHz	16 MB	OSF/1 R1.2.4
Ames Paragon	208	i860	50MHz	32 MB	OSF/1 R1.2 (patch # unknown)

Table 2: Hardware Configuration.

5.2 Results and Discussion

Figure 5 shows the speedup for each of the eight benchmarks. Raw execution times are given in Table 3. Each plot shows curves for HPF (blue lines) and ZPL (red lines) speedups on the T3D (solid lines) and the Paragon (dashed lines). For each benchmark and machine, both the HPF and ZPL curves use the same sequential time to compute speedup, where speedup on P processors is the best sequential time divided by the execution time of the HPF or ZPL program on P processors. This best sequential time is the *fastest* of the following: (a) sequential Fortran 77 execution time, (b) HPF execution time for P=1, or (c) ZPL execution time for P=1.

A detailed comparison of HPF and ZPL compilation techniques is not possible because the languages are different. With this constraint in mind, the remainder of this section comments on the relative performance of the languages and explains why ZPL generally outperforms APR's HPF compiler and identifies some of ZPL's performance shortcomings.

To summarize the data in Figure 5, the ZPL programs delivered greater speedup compared to their HPF counterparts for a substantial majority of the experiments. ZPL is unambiguously superior on both machines for half of the benchmarks: Embar, Swm256, PDE1 and X42 (Figure 5(a)-(d)). Embar is, perhaps, most significant since it was developed to bound the performance of parallel computations [2]. For Shallow (Figure 5(e))—a larger single precision version of Swm256 in which only the inner loops are timed—both ZPL and HPF match until P=32, where there is an advantage for HPF on the T3D and where the HPF Paragon entry is unexpectedly superlinear, suggesting a possible transcriptional error. Grid and Ora (Figure 5(f)-(g)) show conflicting data, with ZPL consistently superior for Ora on the T3D and for Grid on the Paragon,

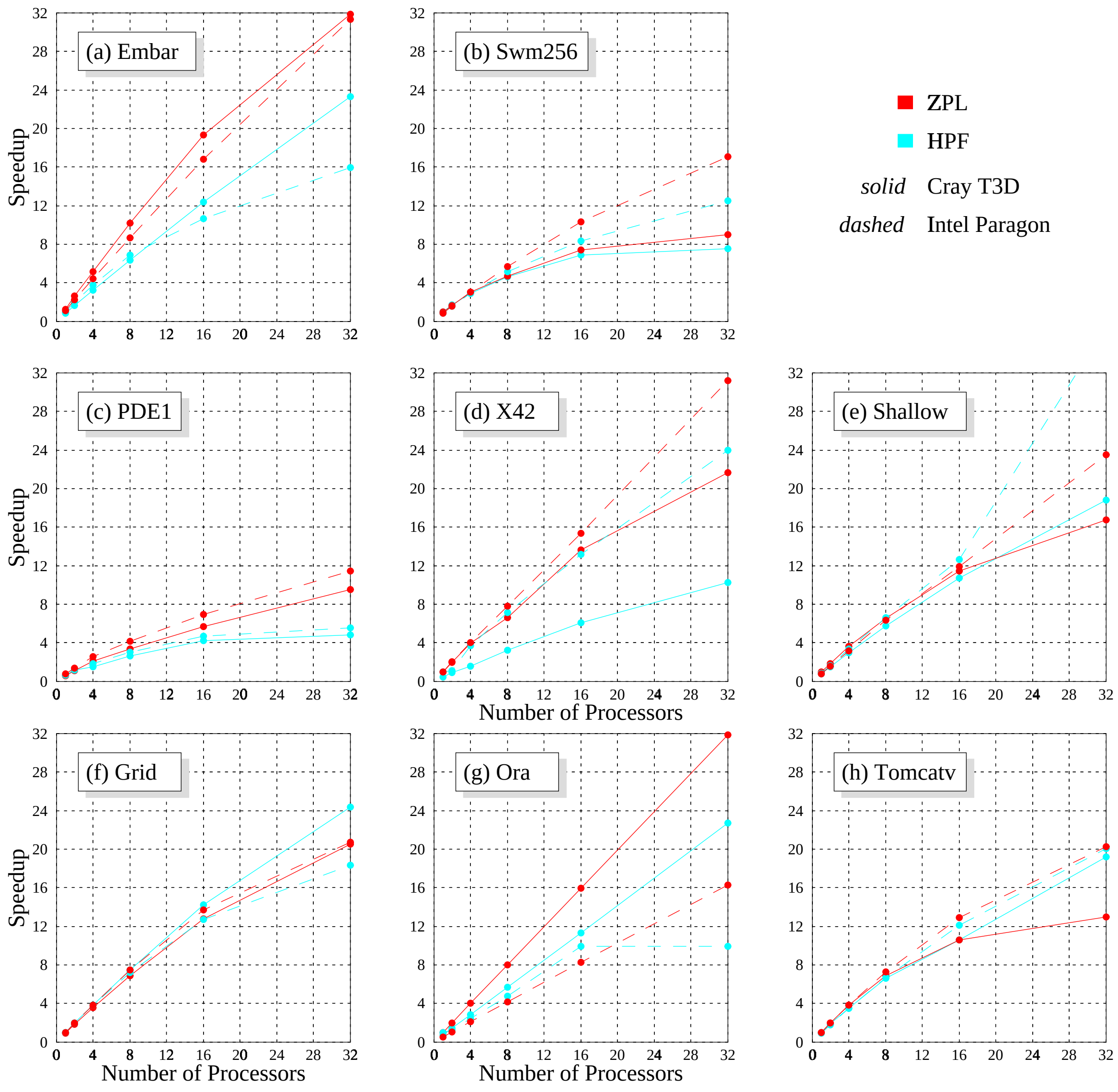

Figure 5: Speedup on the Paragon and T3D.

while HPF is superior for the other comparisons. Finally, the two languages were comparable in Tomcatv until P=32 on the T3D where there was an advantage for HPF.

We believe that much of the performance advantage of ZPL comes from the use of regions, which allows the compiler to exploit a simple dependence-based algorithm to determine when and where communication is needed. The runtime overhead is small since each process can cache information that indicates when communication is necessary. HPF's more general semantics, in which arrays can change representation on every procedure call, would likely require extremely sophisticated analysis to achieve the same performance.

Another ZPL advantage is in the compilation of reduction operations. The ZPL compiler produces efficient code for collective communication, as evidenced by its nearly linear speedup for the Embar benchmark in 5(a). No compiler analysis is required to recognize parallel prefix or reduction operators since they are primitives in the language. Though HPF provides intrinsic functions for collective communication, none of the benchmarks in the xHPF benchmark suite use them. In the Embar benchmark, for example, the HPF code embeds a reduction in other parallel code, making recognition and parallelization difficult, while the ZPL code distinguishes the reduction operator for the compiler.

Three effects, which are largely by-products of the small benchmark problem sizes, explain the cases where the ZPL speedup is lower.

In Tomcatv ZPL uses dynamic regions to implement the tridiagonal systems solver. Though these are not as efficient as static regions, ZPL optimizes the code extensively. Thus, ZPL and HPF perform equivalently for all experiments, except P=32 on the Cray T3D. At this point the problem solved per processor consists of only eight data points whose computation is not sufficient to amortize the overhead of manipulating dynamic regions. The specific problem, initialization of dynamic regions, could be further optimized, but problems are rarely this small.

The Grid benchmark measures both the computation and the program's I/O. The present ZPL I/O implementation is inefficient in that it uses a "token passing" scheme to guarantee that all data is written out consistently and in row major order. Though the I/O is not an issue for the Paragon, the faster clock of the T3D exposes this effect. Future ZPL runtime systems will include parallel I/O and will likely exploit vendor-provided routines.

Finally, the ZPL compiler emits ANSI C, which is compiled using the native compiler of the parallel machine, while HPF emits Fortran. In experiments to understand the performance of Shallow, we have found that for extremely small loops, sequential F77 code outperforms the equivalent sequential C code by as much as a factor of two. This effect vanishes for larger loops, and in any case the ZPL compiler's aggressive optimizations compensate for Fortran's advantage. But, the timed portion of Shallow is composed entirely of small loops, and this, with the small data when P=32, explains the Cray data point.

6 Conclusions

We have shown that ZPL leads to higher level programs that are more mnemonic and readable than HPF, while still typically achieving better performance. These properties of code readability are extremely important since software engineers report that program maintenance represents 70% of software costs. This number may be conservative for long-lived scientific codes, and in any case raises the question of whether perpetuating an existing Fortran code is economic.

This is the first comparison between ZPL and HPF, and to our knowledge the first performance comparison of any language to HPF. We acknowledge that it is early in the compiler development cycle for both languages, and we expect that both will continue to improve. The ZPL compiler effort began in late March 1993, which is roughly when the HPF design was frozen and HPF compiler development could commence in earnest. Of course, Fortran compilation and loop parallelization have been studied for years. More extensive tests are needed, for larger numbers of processors, for a wider array of computers, and most significantly, for more substantial applications, but these results represent an important first step.

Acknowledgments. We gratefully acknowledge the help of Jason Secosky, Kurt Partridge, Vassily Litvinov, and Peter Van Vleet. We thank the Arctic Region Supercomputing Center and the San Diego Supercomputing Center for providing access to their parallel computers.

References

[1] D.A. Abramson and R. Sosic. A debugging tool for software evolution. In *(to appear) CASE-95, 7^{th} International Workshop on Computer-Aided Software Engineering*, July 1995.

[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks (94). RNR Technical Report RNR-94-007, March 1994.

[3] Richard Friedman, John Levesque, and Gene Wagenbreth. *Fortran Parallelization Handbook, Preliminary Edition.* Applied Parallel Research, April 1995.

[4] High Performance Fortran Forum. *High Performance Fortran Specification.* November 1994.

[5] Calvin Lin. ZPL language reference manual. Technical Report 94–10–06, Department of Computer Science and Engineering, University of Washington, 1994.

[6] Calvin Lin and Lawrence Snyder. A portable implementation of SIMPLE. *International Journal of Parallel Programming*, 20(5):363–401, 1991.

[7] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 96–114. Springer-Verlag, 1993.

[8] Calvin Lin and Lawrence Snyder. SIMPLE performance results in ZPL. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 361–375. Springer-Verlag, 1994.

[9] Applied Parallel Research. XHPF benchmark results. Technical report, March 1995.

[10] Lawrence Snyder. Foundations of practical parallel programming languages. In *Proceedings of the Second International Conference of the Austrian Center for Parallel Computation.* Springer-Verlag, 1993.

[11] Lawrence Snyder. A ZPL programming guide. Technical Report 94–12–02, Department of Computer Science and Engineering, University of Washington, 1994.

Embar			seq	$p = 1$	2	4	8	16	32
Cray	HPF	time	6.589	—	—	—	—	—	—
T3D		speedup	1.00	0.81	1.62	3.23	6.37	12.35	23.29
	ZPL	time		5.025	2.516	1.271	0.646	0.341	0.207
		speedup		1.27	2.62	5.18	10.20	19.32	31.83
Intel	HPF	time	14.904	—	—	—	—	—	—
Paragon		speedup	1.00	0.99	1.94	3.77	6.84	10.68	15.96
	ZPL	time		13.474	6.754	3.392	1.716	0.885	0.476
		speedup		1.11	2.21	4.39	8.69	16.84	31.31

Swm256			seq	$p = 1$	2	4	8	16	32
Cray	HPF	time	113.	80.65	47.64	27.68	17.46	11.68	10.68
T3D		speedup	—	1.00	1.69	2.91	4.62	6.90	7.55
	ZPL	time		84.949	49.934	26.889	17.262	10.850	8.968
		speedup		0.95	1.62	3.00	4.67	7.43	8.99
Intel	HPF	time	193.	231.	122.	66.13	37.46	23.14	15.46
Paragon		speedup	—	0.84	1.58	2.92	5.15	8.34	12.48
	ZPL	time		236.281	122.886	63.340	34.173	18.675	11.295
		speedup		0.82	1.57	3.05	5.65	10.33	17.09

PDE1			seq	$p = 1$	2	4	8	16	32
Cray	HPF	time	3.296	5.3	2.8	2.2	1.25	0.78	0.68
T3D		speedup	1.00	0.62	1.18	1.50	2.64	4.23	4.85
	ZPL	time		5.459	2.935	1.575	0.971	0.580	0.346
		speedup		0.60	1.12	2.09	3.39	5.68	9.53
Intel	HPF	time	3.145	5.1	2.67	1.73	1.03	0.67	0.53
Paragon		speedup	1.00	0.62	1.18	1.81	3.05	4.69	5.93
	ZPL	time		4.207	2.276	1.226	0.755	0.454	0.275
		speedup		0.75	1.38	2.57	4.17	6.93	11.44

X42			seq	$p = 1$	2	4	8	16	32
Cray	HPF	time	10.75	7.64	4.08	2.28	1.12	0.59	0.35
T3D		speedup	—	0.47	0.88	1.57	3.21	6.08	10.26
	ZPL	time		3.590	1.830	0.906	0.544	0.263	0.166
		speedup		1.00	1.96	3.96	6.60	13.65	21.63
Intel	HPF	time	27.87	14.96	13.22	3.96	2.06	1.11	0.61
Paragon		speedup	—	0.98	2.22	3.70	7.11	13.19	24.00
	ZPL	time		14.641	7.280	3.671	1.875	0.955	0.469
		speedup		1.00	2.01	3.99	7.79	15.33	31.22

Shallow			seq	$p = 1$	2	4	8	16	32
Cray	HPF	time	132.8	80.31	40.53	20.97	10.77	5.79	3.3
T3D		speedup	—	0.77	1.53	2.96	5.76	10.71	18.80
	ZPL	time		62.028	33.335	17.109	9.534	5.409	3.704
		speedup		1.00	1.86	3.63	6.51	11.47	16.75
Intel	HPF	time	165.6	193.9	99.85	49.47	25.71	13.13	4.5
Paragon		speedup	1.00	0.85	1.66	3.35	6.44	12.61	36.80
	ZPL	time		208.413	104.499	52.340	26.253	13.864	7.051
		speedup		0.79	1.59	3.16	6.31	11.94	23.49

Grid			seq	$p = 1$	2	4	8	16	32
Cray	HPF	time	33.34	33.4	16.91	8.67	4.45	2.34	1.37
T3D		speedup	1.00	1.00	1.97	3.85	7.49	14.25	24.34
	ZPL	time		36.019	18.186	9.367	4.854	2.615	1.626
		speedup		0.93	1.83	3.56	6.87	12.75	20.50
Intel	HPF	time	51.046	51.1	25.8	13.3	7.11	4.01	2.78
Paragon		speedup	1.00	1.00	1.98	3.84	7.18	12.73	18.36
	ZPL	time		51.541	25.947	13.270	6.813	3.726	2.467
		speedup		0.99	1.97	3.85	7.49	13.70	20.69

Ora			seq	$p = 1$	2	4	8	16	32
Cray	HPF	time	50.25	48.93	24.45	12.33	6.11	3.06	1.53
T3D		speedup	—	0.71	1.42	2.81	5.69	11.34	22.68
	ZPL	time		34.698	17.351	8.677	4.340	2.173	1.090
		speedup		1.00	2.00	4.00	8.00	15.97	31.83
Intel	HPF	time	58.00	56.40	46.20	23.30	11.80	5.70	5.70
Paragon		speedup	—	1.00	1.22	2.42	4.78	9.89	9.89
	ZPL	time		109.011	54.516	27.268	13.636	6.821	3.457
		speedup		0.52	1.03	2.07	4.14	8.27	16.31

Tomcatv			seq	$p = 1$	2	4	8	16	32
Cray	HPF	time	41.68	31.59	16.26	8.71	4.64	2.89	1.59
T3D		speedup	—	0.97	1.88	3.51	6.59	10.58	19.23
	ZPL	time		30.580	15.287	8.049	4.466	2.883	2.353
		speedup		1.00	2.00	3.80	6.85	10.61	13.00
Intel	HPF	time	91.27	98.	49.81	25.67	13.45	7.4	4.45
Paragon		speedup	—	0.91	1.79	3.48	6.65	12.08	20.08
	ZPL	time		89.376	44.889	23.295	12.316	6.922	4.406
		speedup		1.00	1.99	3.84	7.26	12.91	20.29

Table 3: Performance Results Summary