NPAC Technical Report SCCS 735, August 1995

Low Level HPF Compiler Benchmark Suite

Tomasz Haupt, Shravanthi G. Reddy, Giriraj Vengurlekar

Northeast Parallel Architectures Center at Syracuse University

Abstract

In this paper we discuss a methodology for evaluation of emerging HPF compilers, based on a dedicated benchmark suite, and present an early experience using DEC Fortran 90 compiler with HPF extensions.

1 Introduction

The high cost of code development and lack of the software investment protection made parallel computing unattractive for many users, in spite of spectacular successes of this technology demonstrated by many research groups. The High Performance Fortran[1], the industry standard data parallel extension to Fortran 90 promises to be easy to use (as compared to the message passing paradigm), yet delivering top performance, scalable codes for MIMD and SIMD computers with non-uniform memory access. The programs written in HPF are expected to be portable, i.e., their efficiency is to be preserved on different machines with comparable number of processors.

2 Evaluation of HPF Compilers

The compilation process of a HPF code can be logically divided into several phases. Two of those have fundamental impact on performance of the resulting code: transformation of a single threaded, global name space program into a SPMD parallel one and subsequent generation of the node code. Typically, the resulting code requires a runtime support (RTS) to facilitate interprocessor communication and therefore an efficient RTS plays an important role in any HPF compilation system.

In our approach, we do not address quality of the node code generation, even though, admittedly, we witness an exciting development in the compiler technology forced by Fortran 90 standard. Instead, we concentrate on parallel transformations and efficiency of RTS. Since the parallelism is inherent in a problem and not its representation, we anticipate many commonalities in different parallel languages and corresponding compiler technologies, including sharing a common runtime support.

3 Low Level HPF Compiler Benchmarks

To evaluate HPF compilers we developed the Low Level HPF Compiler Suite, included in the PARKBENCH suite[2]. The suite comprises of 10 short, synthetic kernels that address selected aspects of the HPF compilation process. They concentrate on testing implementation of explicitly parallel statements, such as array assignments including FORALL and transformational intrinsic functions with different mapping directives. In addition, a few kernels address problem of passing distributed arrays as arguments to subprograms. The codes included in this suite are either adopted from existing benchmark suites, NAS suite, Livermore Loops, and the Purdue Set, or has been developed at Syracuse University, and represent typical computation patterns found in many real applications. Since they are short, they do not necessarily reflect a realistic computation to communication ratio, though.

4 Description of the Kernels

4.1 FORALL statement - kernel FL

FORALL statement provides a convenient syntax for simultaneous assignments to large groups of array elements. Such assignments lie at the heart of the data parallel computations that HPF is designed to express. The idea behind introducing FORALL in HPF is to generalize Fortran 90 array assignments to make expressing parallelism easier. Kernel FL provides several examples of FORALL statements that are difficult or inconvenient to write using Fortran 90 syntax and it is inspired by examples given in the HPF specification[1].

4.2 Explicit template - kernel TL

Parallel implementation of the array assignments, including FORALL statements, is a central issue for an early HPF compiler. Given a data distribution, the compiler distributes computation over available processors. An efficient compiler achieves an optimal load balance with minimum interprocessor communication.

Sometimes, the programmers may help the compiler to minimize interprocessor communication by suitable data mapping, in particular by defining a relative alignment of different data objects. This may be achieved by aligning the data objects with an explicitly declared template. Kernel TL provides an example of this kind.

4.3 Communication detection in array assignments - kernels AA, SH, ST, and IR

Once the data and iteration space is distributed, the next step that strongly influences efficiency of the resulting codes is communication detection and code generation to execute data movement. In general, the off-processor data elements must be gathered before execution of an array assignment, and the results are to be scattered to destination processors after the assignment is completed. In other words, some of the array assignments may require a preprocessing phase to determine which off-processor data elements are needed and execute the gather operation. Similarly, they may require postprocessing (scatter). Many different techniques may be used to optimize these operations. To achieve high efficiency, it may be very important that the compiler is able to recognize structured communication patterns, like shift, multicast, etc. Kernels AA, SH, and ST introduce different structured communication patterns, and kernel IR is an example of an array assignment that requires unstructured communication (because of indirections).

4.4 Non-elemental intrinsic functions - kernel RD

Fortran 90 intrinsics and HPF library functions offer yet another way to express parallelism. Kernel RD tests implementation of several reduction functions.

4.5 Passing distributed arrays as subprogram arguments - kernels AS, IT, IM

The last group of kernels, demonstrate passing distributed arrays as subprogram arguments. They represents three typical cases:

- a known mapping of the actual argument is to be preserved by the dummy argument (AS).
- mapping of the dummy argument is to be inherited from the actual argument, thus no remapping is necessary. The mapping is known at compile time (IT).
- mapping of the dummy argument is to be identical to that of the actual argument, but the mapping is not known at compile time (IM).

5 Performance of the Kernels

In this section we describe an early results obained using DEC Fortran 90 compiler (V2.0-1) on the 8 node cluster of DEC AXP workstations. The results are shown in terms of parallel speedup, defined as a ratio of execution time on one node[†] to that on N nodes for one-dimensional data distributions: BLOCK or (BLOCK,*), fig.1. In addition, fig.2 presents the same data in terms of a speedup relative to execution time on 2 nodes demonstrating scalability of codes without additional overhead associated with transition from sequential to parallel execution.

As shown in fig.1, kernels AA, AS, TM and RD demonstrate linear (or even superlinear) speedup. For AA and TM kernels it is not surprising, since they do not require any communication. Much more interesting is a good performance of the RD kernel which involves MINVAL, COUNT, ANY, ALL, and two SUM reductions. AS kernel involves 2 dimensional shift (5 point stensil, a naive implementation of a Laplace solver), and the relaxation is implemented as a subroutine with dummy argument declared as an assumed shape array with a descriptive mapping.

Nominally less impressive results have been obtained for SH and ST kernels, with speedup of roughly 5 and 6, respectively, on 8 processors. These kernels, however, characterize a relatively high communication to computation ratio, i.e., high communication overhead difficult to avoid even with hand coded message passing implementations. The heart of the SH kernel is a multiple shift

```
FORALL (K=1:N-6)   X(K) = U(K) + R*( Z(K) + R*Y(K)) +
&                         T*( U(K+3) + R*( U(K+2) + R*U(K+1)) +
&                         T*( U(K+6) + Q*( U(K+5) + Q*U(K+4))))
```

while ST involves complicated though regular communication pattern:

```
FORALL (I=2:M) A(I,K1) = A(2*I-1,K1) * B(I,1)
```

The quality of DEC implementation can be deduced from fig.2 (speedup against execution time on two nodes). Both kernels scale, promising a very good performance on large systems.

At this time we were not able to achieve a decent performance of the IR kernel because of irregular communications which are very expensive. The execution time on 8 processors is larger than that of sequential implementation. However, it scales well (see fig.2). This may indicate that the poor performance comes from the extremely large communication overhead inherent to the algorithm (irregular communication pattern with only two floating point operations per iteration) rather than deficiencies of the compiler and its RTS:

```
FORALL (I=1:ME2) Y(L(I)) = 0.5 * (Y(L(I)) + Y(R(I))
```

[†]i.e., using -wsf 1 compiler switch

Kernel FL provides several examples of FORALL statements that are difficult or inconvenient to write using Fortran 90 syntax. It turns out that they can be difficult to compile, too. For example, the version of the compiler we use serializes

```
FORALL (i=1:m-1) s(i) = SUM(z(i:n:m))
```

statement. In addition to "hidden" array transpose in

```
FORALL (k=1:n) x(k,1:n)=y(1:n,k)
```

it leads to a very poor performance of this kernel (not shown in figs.1 or 2). On the other hand, a modified version of this kernel without SUM reduction, and one dimensional mapping with array x(i,j) aligned with y(j,i) to avoid transposition exhibits superlinear speedup. It is another example demonstrating the importance of selecting the right data mapping. For large applications it certainly will require some compromises.

We do not present results on kernel IT and IM, since they require language features not yet implemented in the compiler we used.

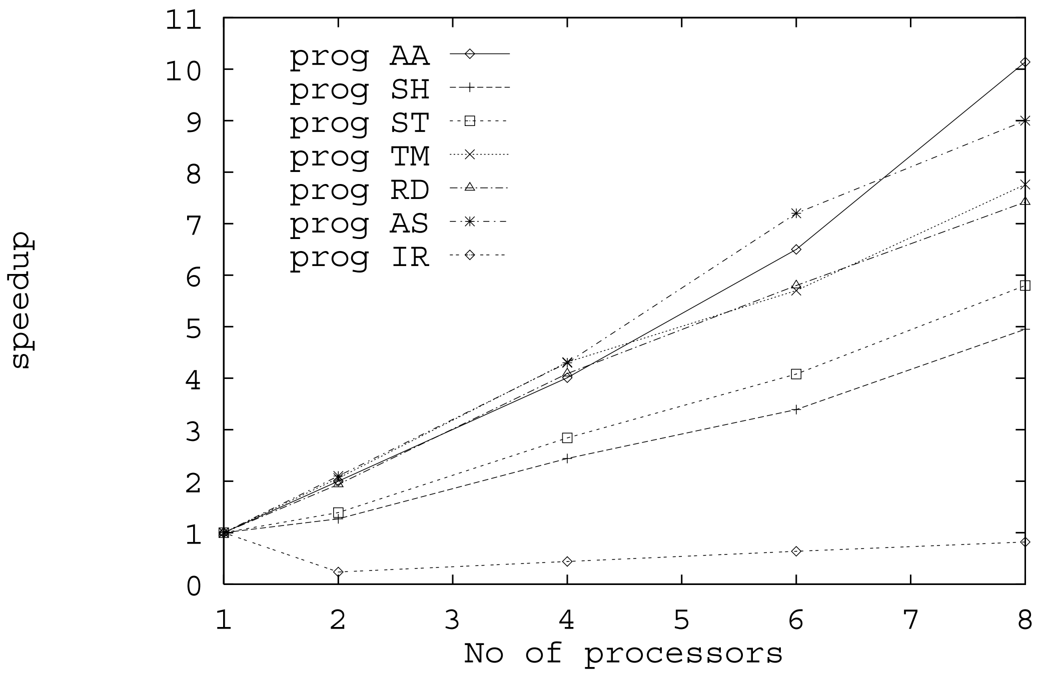

Figure 1: Speedup vs. No. of processors

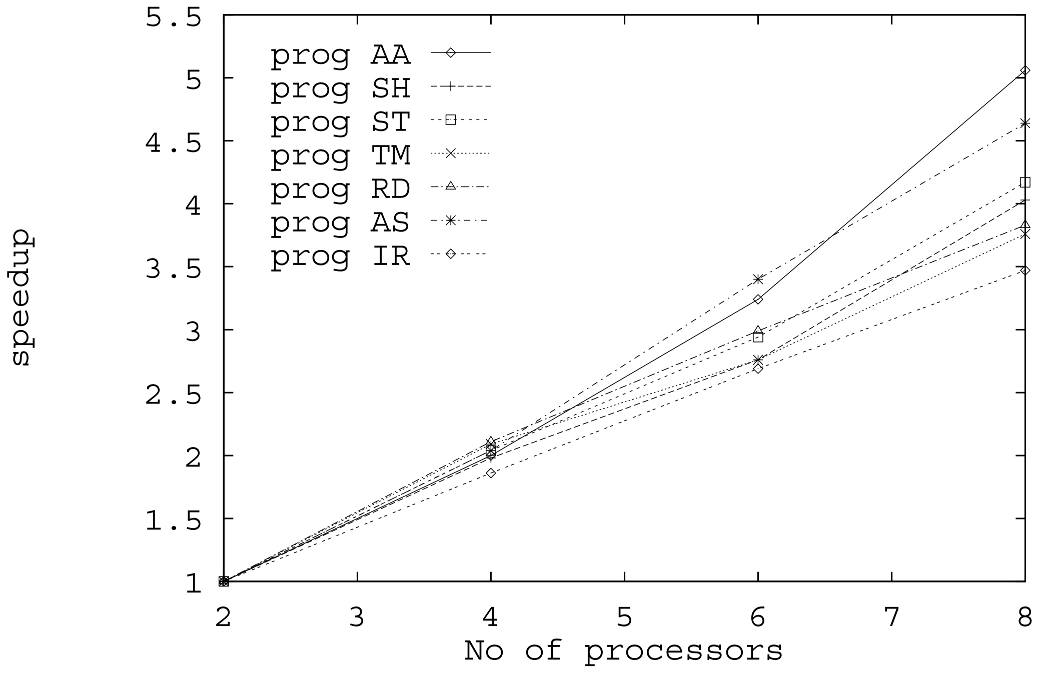

Figure 2: Speedup vs. No. of processors

6 Passing Distributed Arrays as Subprogram Arguments

It is hard to imagine a real life application without subroutines and distributed data objects being passed to them. Since many users will migrate directly from Fortran 77 to HPF (thus not from Fortran 90 to HPF), a word of caution seems to be in place. To obtain a decent performance, the distributed arrays must be passed as assumed-shape arrays (a Fortran 90 feature, therefore never discussed in HPF specification), and not in a traditional Fortran 77 way, i.e., as assumed size arrays. Note, that using assumed-shape arrays, according to Fortran 90 rules, requires specifying an explicit interface to the subroutines. As an example, here are two versions of the AS kernel, one written in old Fortran 77 way, the other using assumed shape arrays. The former is about 10 times slower than the other.

```
      SUBROUTINE SUBA(X,N,M)
      REAL, DIMENSION(N,M) :: X
      INTEGER, INTENT(IN) :: N,M
CHPF$ PROCESSORS P(2,2)
CHPF$ TEMPLATE TMP(N,M)
CHPF$ DISTRIBUTE TMP(BLOCK,BLOCK) ONTO P
CHPF$ ALIGN X(:,:) WITH TMP(:,:)

      FORALL (I=2:N-1,J=2:M-1) X(I,J)=0.25*(X(I+1,J)+X(I-1,J)+
     &         X(I,J-1)+X(I,J+1))

      RETURN
      END
```

```
SUBROUTINE SUBA(X,N,M)
REAL, DIMENSION(:,:) :: X
INTEGER, INTENT(IN) :: N,M

!HPF$ PROCESSORS P(2,2)
!HPF$ DISTRIBUTE X *(BLOCK,BLOCK) ONTO *P

FORALL (I=2:N-1,J=2:M-1) X(I,J)=0.25*(X(I+1,J)+X(I-1,J)+ X(I,J-1)+X(I,J+1))

RETURN
END
```

7 Summary

The synthetic compiler benchmark suite presented here may serve as a test of the base features of HPF compilers. The kernels comprise of code fragments likely to be found in many actual applications. Therefore, the compiler that can handle these simple problems should be capable to deliver high parformance for more realistic codes as well.

It is important to note that we developed the kernels using the Fortran 90 programming style, and expressing parallelism explicitly. In our opinion, this is the best way to understand performance of the code, and to actually achieve expected efficiency. It is our experience, that adopting "dusty" Fortran 77 decks may be tedious, and without appropriate restructuring it often results in poor performance.

The first experience of using DEC Fortran 90 (with HPF extensions) indicates very good performance of codes that are easy to parallelize, and decent performance for more complex applications. Taking into account the fact that the HPF codes have been developed in a small fraction of time as compared to message passing equivalents (the codes are more compact, easier to read and maintain, and much easier to debug), we see it as a beginning of the success story of the data parallel fortran.

Finally, it is worth noticing that we have not addressed here other features supported by the HPF/Fortran 90 programming model such as management of global or dynamical data objects (modules, allocatable arrays, derived types, etc.) which make HPF even more attractive.

More information on the HPF language and compiler evaluation projects at Syracuse University can be found in our Web pages[3,4]

References

[1] High Performance Fortran Forum, "High Performance Fortran Language Specififcation", Scientific Programming 2(1), 1, 1993.

[2] R. W. Hockney and M. W. Berry, "Public International Benchmarks for Parallel Computers", Scientific Programming 3(2), 101-146, 1994. The suite is also available from http://www.npac.syr.edu/users/haupt/parkbench/parkbench.html

[3] K. Hawick, http://www.npac.syr.edu/hpfa

[4] T. Haupt, http://www.npac.syr.edu/projects/bbh/AMR/amr.html