Performance Portability of XL HPF Compiler on IBM SP2 and SMP Multiprocessors

Abderrazek Zaafrani[1], Xinmin Tian[2*]

[1] Motorola Labs
1301 E Algonquin Rd.
Schaumburg, IL 60196, USA
{caz012@email.mot.com}
[2] Intel Microcomputer Software Labs
3600 Juliette Lane, SC12-301
Santa Clara, CA 95052, USA
{Xinmin.Tian}@ intel.com

Abstract—

High Performance Fortran (HPF) is a data-parallel programming language that allows the programmer to specify the data decomposition onto the processors while the compiler takes care of the tedious tasks of communication generation and computation partitioning. Shifting some of the complex tasks from the user to the compiler should encourage programmers to write and port code to parallel machines especially if the compiler implements these tasks efficiently. In this paper, performance results and analysis of a subset of the SPEC92 is presented for the XL HPF compiler on IBM SP2 machines. In addition to obtaining good performance from the compiler, one of the the main concerns of HPF users is portability. Experimental results and analysis are presented in this paper to investigate performance portability (consistency) first across multiprocessor architectures and then across compilers. For performance portability across multiprocessor machines, the same XL HPF compiler used for the IBM SP2 distributed memory machine experiment is also used to compile and execute the same applications but on IBM SMP machines. The comparable speedup and behaviour obtained for both machines indicates that HPF compilers can be portable across different architectures. For performance portability across compilers, various HPF programming techniques and recommendations are introduced to increase the chances of obtaining performance consistency with different HPF compilers.

Keywords— **HPF, performance, portability, compiler, SPMD, distributed memory, shared memory.**

I. Introduction

High Performance Fortran (HPF) is a programming language designed to support the data-parallel programming style by introducing a set of directive extensions to Fortran 90 [HPF 94]. It allows the programmer to specify the data decomposition onto the processors while the compiler takes care of the tedious tasks of communication generation and computation partitioning. In addition to the data decomposition directives, HPF provides other directives to help the compiler generate efficient code. Even though the main target machines for HPF users are distributed memory parallel machines, HPF code should also execute efficiently on shared memory machines.

Shifting some of the complex tasks from the user to the compiler encourages programmers to write and port code to parallel machines provided that the implementation of these tasks is done efficiently. Hence, the existence of good compilers is necessary to make the use of HPF widespread among the parallel programming community. Early users of HPF requested robust compilers, full implementation of the language, good performance from compilers, availability of debuggers and performance analysis tools. In addition, one of the main concerns of parallel programmers is portability which was a driving force behind the creation of HPF and its standard committee [HPF 94].

In this paper, we analyze the performance of some HPF benchmarks using the XL HPF compiler on IBM SP2 machines. The speedup obtained for some applications is quite large. For many other parallel programming languages and environments such as MPI [GRO 94], obtaining good speedup is achievable [SUB 98] [ELI 98]. However, porting the code to other target machines and obtaining similar performance is usually a challenging task that requires substantial tuning effort [JIA 97]. We investigate in this paper the issue of performance portability of HPF code across machines: the same XL HPF compiler used for the IBM SP2 distributed memory machine experiment is also used to compile and execute the same applications (without any change or tuning) but on IBM SMP machines, a shared memory parallel machine. Our experiments indicate that comparable speedup results and behaviour are obtained for both platforms. In addition to performance consistency across machines, this paper addresses performance portability across compilers. While it is widely known that the performance obtained for same applications on same platform using different compilers is occasionally not consistent [NGO 97] for currently available HPF compilers, we attempt to reduce these inconsistencies by introducing HPF programming techniques and recommendations that should increase the chances of obtaining comparable performance with different HPF compilers.

The remainder of this paper is organized as follows. Sec-

*work was done when authors were at IBM Toronto Lab

tion 2 presents an overview of the XL HPF compiler by explaining the Single Program Multiple Data (SPMD) code generated by the compiler and presenting some of the important optimization techniques implemented by the compiler. Section 3 analyzes the performance results obtained for a subset of SPEC92 benchmarks using the XL HPF compiler on an IBM SP2 machine. In Section 4, experiments similar to the ones in Section 3 are presented but for IBM SMP target machines. machines. Section 5 introduces some HPF programming techniques that should make compilers more likely to generate efficient code. Section 6 shows the performance numbers for the subset of SPEC92 benchmarks on an IBM SMP machine but using a compiler specifically targeting the SMP model. Finally, concluding remarks can be found in Section 7.

II. An Overview of XL HPF Compiler

The main target platforms for HPF are distributed memory parallel machines. Given the importance of data distribution in such environment, this task is performed by the user through HPF directives. The compiler uses this data distribution information and the owner computes rule (every processor is defining only the data it owns) to extract parallelism from HPF code. Hence, the speedup obtained for an application depends mainly on:

- the data distribution (BLOCK, CYCLIC, or replicated) described by the user with directives such as *align* and *distribute*.
- the code generated by the compiler after transforming the HPF code into SPMD code. The main tasks performed by the compiler in this transformation is the creation of local data on each processor, automatic generation of communication statements, and computation partitioning [GUP 95] [BOZ 94] [HIR 94].

Figure 1.a shows a simple example of an HPF program. The SPMD code generated by the XL HPF compiler for this example is shown in Figure 1.b: The compiler first generates code to compute descriptor information about local arrays A and B before their allocation on each processor. When executing the SPMD code, processor p ($0 <= p < N$) allocates an array A and an array B both of size $(LB : UB)$ such that $LB = p * ceiling(100/N) + 1 - overlap$ and $UB = (p+1) * ceiling(100/N) + overlap$. The overlap amount is two by default but can be set to any value by using the appropriate compiler option. This additional amount of memory allows the compiler to sometimes avoid the creation of communication buffer for nearest neighbor communication [WOL 96]. In this case, data is received into the overlap entries, and computation then proceeds with local accesses instead of a communication buffer. After the allocation of local arrays A and B, the compiler generates code to compute the sets of data to be received, the sets of data to be sent, the processor sets involved in the communication. After generating calls to *send* and *receive* communication routines, the compiler generates code to perform the actual computation. For the latter step, computation partitioning is achieved through loop bound shrinking. Finally, storage for local arrays A and B are deallocated.

```
      integer, dimension (100) :: A, B

!hpf$ DISTRIBUTE (BLOCK) onto P :: A, B

      do i = 2,100
         A(i) =  B(i-1) + A(i)
      end do

      end
```

(a) Simple Example of HPF Code

```
/* Num_PE and Proc_ID are 1-dimensional arrays of size equal to the
/* dimensionality of the processor configuration. */
call  _xlhpf_get_pe_index(1, Num_PE, Proc_ID)

/* Determine information about array  A and B. */
Global(1) = 1
Global(2) = 100
Distribution(1) = (100 + Num_PE(1) - 1) / Num_PE(1)     /* BLOCK Size */
Distribution(2) = 0                                     /*  0 indicates a BLOCK Distr. */
iown_l = 1 + ((100 + Num_PE(1) - 1) / Num_PE(1)) * Proc_ID(1)
iown_u = ((100 + Num_PE(1) - 1) / Num_PE(1)) + iown_l - 1

/* Each processor allocates it own chunk of array A and B.*/
call _xlhpf_allocate (A, Proc_ID, Global, Distribution)
call _xlhpf_allocate (B, Proc_ID, Global, Distribution)

/* Communication code */
if (Proc_ID(1) .gt. 0) then
   /* Code computing the arrays PSS (Processor Set Send) and DSR (Data */
   /* Set Receive) is not shown.                                       */
   call _xlhpf_nbreceive_sections (Proc_ID, PSS, B, DSR)
end if

if (Proc_ID(1) .lt. Num_PE(1)-1) then
  /* Code computing the arrays PSR (Processor Set Receive) and DSS     */
  /* (Data Set Send) is not shown.                                     */
  call _xlhpf_send_sections (Proc_ID, PSR, B, DSS)
end if

/* Computation partitioning done through loop bound shrinking*/
do I = max(iown_l,2), min(iown_u,100)
  A (I) = B(I - 1) + A(I)
end do

/* Each processor frees its local chunk of array A and B. */
call _xlhpf_deallocate(A)
call _xlhpf_deallocate(B)
end
```

(b) Generated SPMD code for Above Example

Fig. 1. Generation of SPMD Code

Figure 2 shows a pseudo code that emphasizes the changes obtained if no overlap region is used (ex. B[i-3] is used on the rhs of the assignment statement of Figure 1.a instead of B[i-1]). Since the default overlap region of two array elements is not large enough to allow storing the values of B that need to be received into the overlap entries, a communication buffer for B is allocated for this purpose. Given that all use of array B in the computation part of the code is done through the communication buffer, local communication has to be performed by packing local elements of array B into the buffer (i.e elements of B that exist locally are copied into the communication buffer).

Given the high communication overhead for a distributed memory machine, it is essential for an HPF compiler to generate efficient communication by implementing many of the optimizing communication techniques [BOZ 94] [GUP 94]. Among the important optimization techniques implemented in the XL HPF compiler are:

- Message vectorization: Communication analysis is performed to move communication to the outermost possible loop. This enables the compiler to send the data of an array in a single message instead of sending the data of that array element by element. This is the most important optimization technique done by the compiler in terms of improvement to execution time. Any improvement to the compiler to recognize and perform communication vectorization, and any code tuning by the user to expose message vectorization to the compiler can contribute significantly to the decrease in execution time of an application.
- Elimination of redundant communication statements: Communication analysis within single loopnests and across loopnests (within the same compilation unit) is performed to eliminate unnecessary communication. The compiler attempts to eliminate a message with a source processor set P_s, a destination processor set P_d, and a data set D if there exists an earlier message from P_s to P_d with a data set D_1 where $D \subseteq D_1$ and the data in D_1 is not invalidated by write statements between the two messages.
- Recognizing communication Patterns: Broadcast operations, nearest neighbor communication, and reduction operations are recognized in order to generate efficient communication. The reduction operations that are recognized are $+$, $-$, $*$, $/$, min, max, $maxval$, and $minval$.
- Message Combining: In order to reduce the overhead of message startup time, communication statements for different array variables are combined into a single message when possible. This is, in essence, similar to communication vectorization: message vectorization combines a large number of small messages into one large message, while message combining coalesces a small number of large messages into one larger message. Given that the number of message combined is small, the improvement is not substantial, especially if we consider the drawback of having a very big message.

In addition to communication generation, the compiler is also responsible for computation partitioning which, in its simplest form, consists of adding guard statements when necessary to ensure the owner computes rule. A guard IF statement is needed around every defining statement to make sure that it will be executed on the processors owning the data being defined (owner computes rule does not apply to reduction). In its ideal form, computation partitioning consists of parallelizing loops by shrinking loop bounds so that every processor iterates over a small set of iterations and avoid adding any guard statement. In Figure 1.b, the loop bounds have been successfully reduced and no guard statement is needed around the assignment statement. Sometimes, a combination of loop bound shrinking and guard statement insertion are needed to ensure a correct execution of the code.

```
/* Same code as in Figure 1.b for determining information about local  */
/* arrays A and B and allocating them is not shown here.               */
                .
                .
/*allocate communication buffer*/
Size(1) = iown_1 + (-3)
size(2) = min (iown_u,100) + (-3)
call _xlhpf_allocate_computation_buffer(buff, Size)
                .
                .
/* Communication code */
if (Guard_expression1) then
  do isr=...
     if (...) then
                ...
        call _xlhpf_nbreceive_sections (Proc_Id, PSS, buff, DSR)
     end if
  end do
end if
if (Guard_expression2) then
  do isr= ....
    if (non_local_expression) then
                ...
      call _xlhpf_send_sections (Proc_Id, PSR, b, DSS)
    else
      call _xlhpf_pack_local_blocks(b, buff, DSS, PSR)
    end if
  end do
end if

/* Computation partitioning */
do I = max (iown_l,4),min (iown_u,100),1
  a (I) =  buff (I - 3) + a (I)
end do

/* Freeing local data */
call _xlhpf_deallocate(buff)
call _xlhpf_deallocate(a)
call _xlhpf_deallocate(b)
end
```

Fig. 2. Overview of the Changes Without Overlap Region

The new expressions created by loop bound shrinking and guard statement can sometimes be too complex. Hence optimization techniques for computation partitioning are also needed. Among the techniques used by XL HPF compiler, we can mention:

- Simplification of expressions used in guards and shrunk loop bounds. These expressions contains several calls to max, min, and other routines.
- Guard motion: If the expressions in a guard statement is loop-independent, then the guard is moved outside the loop (it is moved to the outermost possible loop).
- Merging guard statements: If the guard expressions of two adjacents statements are similar, then only one guard is used for both statements.

III. HPF Performance on SP2

A. Benchmark Analysis

In order to analyze the performance of XL HPF compiler, a subset of the SPEC92 benchmarks are selected for execution on a 16-node SP2 machine. The speedup obtained for these benchmark applications shown in Table I ranges from almost linear speedup (*grid2*) to poor speedup (*pde1par*). This speedup is obtained by dividing the execution time of the serial version (which is obtained by using the XL HPF compiler with the option *-nohpof* specified and which generates code similar to the one generated by the XLF90 compiler) by the execution time of the parallel version. Grid2 has

a computational intensive loopnest where most of the execution time is spent. By analyzing the SPMD code generated for this critical part of the code, it can be noticed that there are three factors that contributed to the efficient execution of the code: 1) loops in the loopnest are parallelized, 2) all communications have been pulled outside the loops and vectorized. 3) only nearest-neighbor communications are used. This type of communication is fast because the code generated by the compiler is simple and communication is done through the overlap region and not through communication buffers as explained in the previous section.

Among the rest of the applications, both Tomcatv2 and Shallpar90 show good speedup. For Tomcatv2, the code is computation intensive but there is no particular loopnest or part of the code that is consuming most of the execution time. Hence, the entire code needs to be analyzed. The arrays used in Tomcatv2, all two-dimensional arrays, are distributed in a BLOCK manner in one dimension and collapsed in the other dimension. This reduces communication overhead especially when large number of communication statements are needed but disables parallelism in the collapsed dimension. Disabling parallelism in one dimension is not a drawback as long as enough parallelism can be extracted from the other dimension. It is usually a good practice to distribute arrays involved in communication (especially for non-nearest neighbor communication) in one dimension only and collapse the rest so that the overhead of communication does not become too expensive (computation of data sets to be sent and received, processors sets involved in the communication, etc are performed in one dimension only). Even though the data distribution for this application is efficient and loops have been parallelized by the compiler, the large number of reduction communications and, to a lesser extent, the large number of nearest neighbor communication are the major factors contributing to the non-linear speedup of Tomcatv2. Similar analysis can be done for Shallpar90 to explain its reasonably good but not linear speedup.

TABLE I
SPEEDUP FOR A SUBSET OF SPEC92 ON IBM SP2 WITH XL HPF COMPILER

Benchmark	Speedup				
	1 PE	2 PEs	4 PEs	8 PEs	16 PEs
Grid2	1.00	1.98	3.94	7.75	15.10
Shpar90	0.89	1.64	3.18	5.77	NA
Tomcatv2	0.98	1.93	3.68	6.78	11.74
Swm256par	0.97	1.58	2.61	3.86	5.02
Pdelpar	0.93	1.55	1.95	2.63	3.11
Translpar	0.14	0.19	0.30	0.52	0.81

As we move down the list in Table I, Pdelpar and Smw256par show poor speed up. For Pdelpar, only nearest neighbor communications are generated by the compiler. However, the code is not computationally intensive. Hence, communication overhead had more negative effect on the speedup for Pdelpar than for the previously analyzed applications. In addition to communication, there are few other factors that contributed to the unsatisfactory speedup: 1) An array is redistributed when entering the major subroutine (relaxation subroutine) and then restored back to the old distribution after returning from the subroutine. 2) temporary arrays are generated by the compiler with sizes determined at runtime (expensive memory overhead because of runtime allocation on the heap). Similar observations can be noticed to justify the poor speedup for Smw256par.

Finally, the last application in Table I shows a big slowdown. Translpar is a small program that just consists of transposing a large array and assigning the result to another array. Thus, the whole program consists mainly of doing expensive communications. The two-dimensional arrays used in this application are distributed along both dimensions. This increases the overhead of communication given that send data set information, receive data set information, processor set information, etc. are computed for both dimensions. The big slowdown for translpar should be expected when executed on a distributed memory parallel machine. However, it can be noticed from Table I that the execution time of Translpar on 16 processors is getting close to the serial execution time. Hence, if this application included some regular computational code after the array transposition, then satisfactory speedup may be obtained for large number of processors.

B. Serial Execution vs Parallel Execution on One Processor

A important observation that can be seen in Table I is the difference in execution time between the serial versions of the applications and their corresponding parallel versions executed on one processor (1 PE column in the table represents the serial execution time divided by the parallel execution time on one processor). Both versions are obtained using the same XL HPF compiler. This difference is due to the complex SPMD code generated by the compiler for the parallel version. The code portability principle expected by HPF programmers is violated when the difference between both versions becomes large. From Table I, this difference seems acceptable for most applications (except for Translpar). It is usually high for code that requires non-nearest-neighbor communication. This type of communication is performed using buffers and the actual computation uses the data from the buffers. Hence, even for code executing on one processor, local communication is performed for non-nearest-neighbor communication through the creation of communication buffers and packing data into the buffers. Another factor contributing to the difference in execution time between the two versions is the low level optimizer (back-end com-

piler) which makes a better job optimizing the serial version than the parallel version because of the complexity of the latter one. Future improvement to the code generation and optimization of the SPMD code should reduce the difference between the two versions.

IV. Performance Portability across Architectures: SMP Case

Obtaining acceptable speedup for a parallel application is usually achievable [SUB 98] [ELI 98]. However, obtaining similar speedup without any large tuning effort when the application is ported to another machine is quite often a challenge. This has turned away many potential parallel programming users because of the extra overhead associated with porting code. By being the first standard parallel high level language, HPF solved many portability problems for parallel programmers. But, does it solves the performance portability problem?. In order to investigate this issue, we execute our SPEC92 benchmark subset on an IBM SMP, and on a cluster of IBM SMP machines using the same XL HPF compiler which has been mainly used by programmers, so far, for the SP distributed memory machines.

An HPF compiler creates SPMD code. In order to run this code, a parallel environment creates N processes (N is specified by the user as the number of processors to be used). Each Process is executing the same object code. Processes communicate among themselves using the Message Passing Interface (MPI) library. In an SMP environment, threads are attached to these processes. Data communication among threads is more efficient than the traditional communication between processes: The MPI, in a thread based environment, recognizes when communication is local within the SMP node and when communication is between two different nodes in a cluster of SMP nodes. For local communication, a simple copy of the data is performed instead of the the more costly send/receive.

TABLE II
Speedup for a Subset of SPEC92 Using XL HPF Compiler on a Cluster of two 8-processor IBM SMP Machines

Benchmark	Speedup					
Number of PEs	1 PE	2 PEs	4 PEs	8 PEs	12 PEs	16 PEs
Grid2	1.01	2.00	3.94	7.54	10.77	13.79
Shpar90	0.95	1.89	3.40	6.43	7.78	12.61
Tomcatv2	0.97	1.79	3.25	5.00	6.74	7.60
Swm256par	0.82	1.43	2.46	3.32	3.11	3.44
Pdelpar	0.93	1.61	2.33	2.85	2.85	3.22

Table II shows the speedup for the subset of SPEC92 using XL HPF compiler on a cluster of two 8-processor SMP machines. It can be noticed from the table that the speedup results obtained are comparable to the ones for the SP2 machine (Table I). No major difference in speedup can be noticed between the two platforms. This is encouraging for HPF users who consider performance consistency as a major concern. By focusing only on the results of a single SMP node (1, 2, 4, 8 PEs columns in Table II), it can be noticed that the difference in speedup between an SMP machine and an SP2 machine is small. However, we would have expected to get more improvement in speedup in favor of SMP given the promise of the parallel environment to recognize same node communication and execute them efficiently. The parallel environment we are using is still in its early development stage and we may get better results for a single SMP node in the future. By analyzing the speedup for the cluster of two 8-processor SMP nodes (last two columns in Table II), it can be noticed that the difference in speedup between an SP2 and a cluster of SMP gets bigger (and is in favor of SP2). This is expected because of the expensive communication overhead across SMP nodes in a cluster.

V. Performance Portability across HPF Compilers

The experiments described in the previous section show encouraging results for the performance portability of HPF code across platforms. Another major aspect of performance consistency is portability across compilers. Parallel programmers demand to get approximately the same performance for their applications when they switch to another compiler. While this goal has not been currently reached [NGO 97] as HPF compilers are still immature and are in early stages of development, we should be able to approach this objective in the future once most of the data parallel optimization techniques are well understood and implemented.

Many data parallel compiler optimization techniques (some of which are briefly described at the beginning of the paper) are implemented in various HPF compilers. However, an attempt to implement a particular optimization on some code may not always be successful by all compilers especially with the early version of the HPF compiler releases. While waiting for these optimizations to be more aggressive and their implementation to be more robust, the user may need to write better code, add more directives provided by the language so that the compiler can generate better code. This results in a better portability among compilers given that a well written code is more likely to be portable across compilers. In the remainder of this section, we present some techniques and recommendations that the programmer can use to write code more likely to be portable across compilers.

A. Data Distribution

Given the importance of data distribution on the performance of HPF code, this task is performed by the user. The main objective of good data distribution is to achieve load balancing without introducing excessive communica-

tion overhead. In order to achieve this and increase the chance of having portable code across compilers, the user should be aware of some rules and hints about data distribution:

- Dimensions with expensive communication should be collapsed. In the example of Figure 3, communication is done inside the j loop because of the data dependence. Fortunately, the compiler interchanges the j loop with the i loop and vectorizes communication by pulling it outside the i loop. This vectorization would not have been legal without loop interchange. However, this is still an expensive communication given that it is still nested within a loop (j loop). Collapsing the elements of an array dimension with expensive communication (such as communication inside loops, irregular communication, ...) instead of distributing them should result in a faster execution of the code. Hence, a distribution of $(BLOCK, *)$ for array A in the code in Figure 3 is more efficient than a $(BLOCK, BLOCK)$ distribution.

```
      integer, dimension (100,100) :: A

!hpf$ DISTRIBUTE (BLOCK, *)  :: A

      Do i = 1,100
         Do j = n+1,100
            A(i,j) =  A(i,j-n)* c
         End do
      End do

      End
```

Fig. 3. Code with Expensive Communication if Distribution Changes to (BLOCK,BLOCK)

- Theoretically, it is beneficial to switch a distribution of an array from $BLOCK$ to $CYCLIC$ when the $BLOCK$ distribution causes load imbalance. However, the expressions used to compute the guards, the new loop bounds, the send/receive processor sets, and send/receive data sets are too complex for the $CYCLIC$ distribution. The load balancing benefits obtained by using the $CYCLIC$ distribution may be canceled by its complex code generation. Hence, the $CYCLIC$ distribution should not be used unless a large load imbalance exists using other distributions. In the example of Figure 4, a $CYCLIC$ distribution may seem appropriate because of the triangular iteration space. However, the granularity of the code is not large enough to get any load balancing benefits because of the complex code generated by the $CYCLIC$ distribution. Experimentation with the code indicates that the granularity of the code in Figure 4 should be at least ten times the current granularity to get any improvement with the $CYCLIC$ distribution. No communication is needed for the code in Figure 4. An even larger granularity would be needed for the $CYCLIC$ distribution to be beneficial if the code involves communication.

```
      integer, dimension (100,100) :: A, B

!hpf$ PROCESSORS P(4,4)
!hpf$ DISTRIBUTE (BLOCK) onto P :: A, B

      Do i = 1,100
         Do j = 1,i
            A(i,j) =  B(i,j) * A(i,j)
         End do
      End do

      End
```

Fig. 4. $CYCLIC$ vs $BLOCK$ Distribution

- Small arrays should be replicated instead of distributed. This results in some code being redundantly executed by every processor because of the owner computes rule. However, the benefits of avoiding communication and avoiding generating complex code for computation partitioning should outweigh the loss of parallelism for small arrays.
- For a P number of processors available, it is usually preferable to use them as a one dimensional processors grid and distribute arrays in just one dimension while collapsing the other dimensions. As indicated in section 2, complex expressions are sometimes generated in the new loop bounds. Extracting the parallelism in a loop-nest from one loop only by modifying the bounds of that loop may reduce the overhead of computation partitioning. In addition, communication overhead may be reduced by computing data and processor sets in one dimension only. Better yet, communication may be eliminated when collapsing dimensions (as is the case in the example of Figure 3).

When considering to apply the above techniques and others for a large program, they may conflict with each others in different parts of the code. A distribution that is beneficial in one part of the code may become inefficient in another part. The user should investigate the code and determine the parts that are more critical to the overall execution time. Critical parts of the code should be given a priority in the data distribution choice.

B. *Use of Directives*

- The INDEPENDENT Directive can be used by the programmer before a *DO* or *Forall* construct to indicate to the compiler that the iterations of a loop can execute in any order because of the inexistence of data dependence inside the loop. This should be especially used for loops with complex code for which the compiler may not determine that no data dependence exists in the loop and consequently generates inefficient code. For the exam-

ple in Figure 5.a, the code generated by the compiler is inefficient given that communication occurs at the innermost loop. For a 4×4 processor grid, each processor generates 25×25 send/receive messages of size n where n is the size of one element of A (for some of the processors, these messages are local). The user can transform the loopnest in Figure 5.a into the equivalent two loopnests in Figure 5.b using the parallel region transformation presented in [ZAA 94]. For the transformed code, the compiler uses the assertion provided by the INDEPENDENT directives to pull communication outside the loopnests. For each loopnest, every processor vectorizes all of its send/receive messages into a single message. Hence, every processor generates two send/receive messages only (one for each loopnest) but of size $\frac{25^2}{2}$. The transformed code clearly outperforms the initial code because of the large overhead of communication startup time. Without the INDEPENDENT directives in Figure 5.b, compilers may not be able to recognize that communication can be pulled outside the loop.

```
integer, dimension (100,100) :: A,B
!hpf$ DISTRIBUTE (BLOCK,BLOCK)  :: A,B
   Do i = 1,100
      Do j = 1,100
         A(i,j) = A(j,i) + B(i,j)
      End do
   End do
   End
```

(a) Complex Code

```
integer, dimension (100,100) :: A, B
!hpf$ DISTRIBUTE (BLOCK,BLOCK)  :: A, B
!hpf$ Independent
   Do i = 1,100
!hpf$ Independent
      Do j = i,100
         A(i,j) = A(j,i) + B(i,j)
      End do
   End do
!hpf$ Independent
   Do i = 1,100
!hpf$ Independent
      Do j = 1,i-1
         A(i,j) = A(j,i) + B(i,j)
      End do
   End do
   End
```

(b) Transf. of Code in (a)

Fig. 5. Use of INDEPENDENT in Complex Code

- Loop containing procedure calls are not parallelized. However, the user can use the PURE directive to assert to the compiler that the procedure has no side effect. With this directive, calls to a procedure within a loop can be executed in any order and hence be parallelized without any need for a complex interprocedural analysis.

C. Compiler Temporary Arrays

The compiler needs sometimes to create temporary arrays because of data dependence in array assignments, FORALL constructs, WHERE constructs, etc. A compiler temporary array is aligned with a user array chosen by the compiler through some heuristics. The compiler may not choose an efficient alignment especially for complex code. In addition, the compiler may need to create and allocate temporary arrays at runtime (expensive operation) if the size of the temporary array can not be determined at compile time. The programmer can transform the code so that creating temporary arrays, finding an appropriate alignment, and storing into them are explicitly done by the user in the program.

D. Code Simplification

Compilers occasionally do not generate efficient communication statements or code partitioning for irregular code. The user should simplify the code even at the expense of writing longer programs. In Example 6.a, the complex loop bounds results in complex communication generation for array B. The user can transform his code into two loops as shown in 6.b. The first loop has the role of making the communication generated by the compiler simpler. More data than needed are transferred but the compiler is able to generate efficient communication because of the simple loop bounds. The second loop does the actual computation.

```
integer, dimension (100,100) :: A, B
!hpf$ DISTRIBUTE(BLOCK,BLOCK) onto P::A,B
   Do I = 1,100
      Do J = 2*I,100-i
         A(I,J) = B(J,I) + A (I,J)
      End do
   End do
   End
```

(a) Simplify Code

```
integer, dimension (100,100) :: A, B
integer, dimension (100,100) :: Temp
!hpf$ DISTRIBUTE(BLOCK,BLOCK) onto P :: A, B
!hpf$ DISTRIBUTE(BLOCK,BLOCK) onto P :: Temp
   Do I = 1,100
      Do J = 1,100
         Temp(I,J) = B(J,I)
      Endo
   Endo
   Do I = 1,100
      Do J = 2*i,100-i
         A(I,J) = Temp(I,J) + A (I,J)
      Endo
   Endo
   End
```

(b) Overcommunication But simpler

Fig. 6. Simplify Code

VI. Performance Comparison with other Fortran Compilers

The experiments presented in section 4 indicates that there is no major performance changes noticed between executing HPF code on a distributed memory SP2 machine and executing the same code on an SMP machine (or a cluster of SMP machines) using the same XL HPF compiler for both executions. The issue addressed in the experiment of this section is how much improvement (if any) we can obtain by using a compiler specifically designed for an SMP environment. For this matter, we use the XL Fortran 90 parallelizing compiler that automatically parallelizes code by creating parallel threads to be executed on IBM SMP machines. The same subset of Spec92 benchmark applications are used here without any change in the code (HPF directives in the code are ignored by the XL Fortran 90 parallelizing compiler).

Table III shows the speedup obtained for both compilers (the speedup for XL HPF are reproduced from Table II). For some applications, the XL HPF Compiler gives slightly better results. For others the XL Fortran 90 Compiler gives better speedup (The speedup for XL HPF is taken from Table

II). We can conclude form this experiment that an HPF compiler can generate SPMD code that is efficient enough for execution on a shared memory system, and that it can compete with compilers specifically targeting the shared memory model. The overhead of communication code generated by an HPF compiler seems not te be more expensive than the overhead of synchronization generated by the SMP compiler. Note that the XL Fortran 90 and XL HPF compilers both come in the same product. With the appropriate compiling options, the user can either invoke the HPF compiler, or the Fortran 90 SMP compiler.

Another experiment to consider in the future is for the user to tune the benchmark application code and parallelize the code by using the OpenMP directives. OpenMp is a set of compiler directives and callable runtime library routines that extend Fortran to express shared memory parallelism. The OpenMP directives have the same role as HPF directives and are designed to exploit distributed memory parallelism. Such an experiment would be more fair for a performance comparison between XL HPF and XL Fortran 90 compilers. Given that this paper only includes experiments with minimal code changes, this experiment is beyond the scope of the paper.

TABLE III
PERFORMANCE COMPARISON FOR A SUBSET OF SPEC92 BETWEEN XL HPF AND XL FORTRAN 90 SMP COMPILERS ON AN 8-CPU IBM SMP MACHINE

Benchmark	Speedup			
	1 PE	2 PEs	4 PEs	8 PEs
Grid2: XL SMP	0.93	1.77	3.53	6.89
XL HPF	1.00	2.00	3.94	7.54
Shpar90: XL SMP	1.00	1.69	3.43	7.08
XL HPF	0.95	1.89	3.40	6.43
Tomcatv2: XL SMP	0.99	1.78	2.07	NA
XL HPF	0.97	1.79	3.25	5.00
Pdelpar: XL SMP	0.93	1.78	3.00	3.36
XL HPF	0.93	1.61	2.33	2.85
Swm256par: XL SMP	1.00	1.89	3.51	5.95
XL HPF	0.82	1.43	2.46	3.32

VII. CONCLUSION

In this paper, performance results and analysis for a subset of the SPEC92 benchmarks are presented for an SP2 machine, Obtaining good speedup for the code is not enough as parallel programmers require portability of the code. One of most complex aspects of code portability is performance. Results from this paper indicates the performance obtained from HPF compilers can be portable across different architectures. This has been shown by running the applications on a distributed memory system and on a shared memory system using the same compiler. Comparable speedup are obtained for both systems. For performance portability across compilers, some HPF programming techniques are presented in this paper to allow the user to write efficient and simple HPF code which increases the chances of obtaining portable code across compilers.

REFERENCES

[BOZ 94] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M. Wu. Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers. *Journal of Parallel and Distributed Computing*, 15-26, April 1994.

[BOZ 95] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. Compiling High Performance Fortran. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, 1995

[CHA 92] B. Chapman, P. Mehrotra, J. Van Rosendale, H. Zima. Programming in Vienna Fortran. In *Scientific Programming*, 31-50, August 1992.

[ELI 98] V. Elisseev. Parallelization of Three-dimensional Spectral Laser-plasma Interaction Code Using High Performance Fortran. In *Journal of Computers in Physics*, 173-180, March 1998

[FRU 98] M. Frumkin, H. Jin, J. Yan. Implementation of NAS Parallel Benchmarks in High Performance Fortran. In *NAS Technical Report NAS-98-009*, September 1998.

[GRO 94] W. Gropp, E. Lusk, A. Skjellum. *Using MPI: Portable Parallel Processing with the Message-Passing Interface*. MIT Press, 1994.

[GUP 94] M. Gupta, E. Schonberg, H. Srinivasan. A unified data-flow framework for optimizing communication. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[GUP 95] M. Gupta, S. Midkiff, E. Schonberg, V. Scshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF Compiler for the IBM SP2. In *Proceedings of Supercomputing'95*, December 1995.

[HIR 94] S. Hiranandani, K. Kennedy, C. Tseng, Young. Evaluating Compiler Optimizations for Fortran D. In *Journal of Parallel and Distributed Computing*, 1994

[HPF 94] *High Performance Fortran Language Specification*, Version 1.1. CRPC-TR92225. Center for Research on Parallel Computation, Rice University. Houston, 1994.

[JIA 97] D. Jiang, H. Shan, J. Singh. Performance Portability of Applications and Optimizations Across Shared Address Space Multiprocessors. In *ACM/SIGPLAN Symposium on Principles and Practices of Parallel Programming*, June 1997

[NGO 97] T. Ngo, L. Snyder, B. Chamberlain. Portable Performance Of Data Parallel Languages. In *Proceedings of Supercomputing 97*, 1997

[SUB 98] J. Subhlok, P. Steenkiste, J. Stichnoth, P. Lieu. Airshed Pollution Modeling: A Case Study in Application Development in an HPF Environment. In *IPPS/SPDP 98 Proceedings*, April 1998.

[WOL 96] M.J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.

[ZAA 94] A. Zaafrani, M. Ito. Parallel Region Execution of Loops with Irregular Dependencies. In *International Conference on Parallel Processing*, August 1994.