PGHPF—An Optimizing High Performance Fortran Compiler for Distributed Memory Machines

ZEKI BOZKUS, LARRY MEADOWS, STEVEN NAKAMOTO, VINCENT SCHUSTER, AND MARK YOUNG

The Portland Group, Inc. (PGI), 9150 SW Pioneer Court, Suite H, Wilsonville, OR 97070;
e-mail: {zeki,lfm,scn,vinces,mark}@pgroup.com

ABSTRACT

High Performance Fortran (HPF) is the first widely supported, efficient, and portable parallel programming language for shared and distributed memory systems. HPF is realized through a set of directive-based extensions to Fortran 90. It enables application developers and Fortran end-users to write compact, portable, and efficient software that will compile and execute on workstations, shared memory servers, clusters, traditional supercomputers, or massively parallel processors. This article describes a production–quality HPF compiler for a set of parallel machines. Compilation techniques such as data and computation distribution, communication generation, run-time support, and optimization issues are elaborated as the basis for an HPF compiler implementation on distributed memory machines. The performance of this compiler on benchmark programs demonstrates that high efficiency can be achieved executing HPF code on parallel architectures. © 1997 John Wiley & Sons, Inc.

1 INTRODUCTION

Recently there have been major efforts to develop programming language and compiler support for distributed memory machines. Based on the initial work on projects like Fortran D [1] and Vienna Fortran [2], the High Performance Fortran Forum produced HPF version 1.0 [3]. HPF incorporates a data-mapping model and associated directives which allow the programmer to specify how data are logically distributed in an application. An HPF compiler interprets these directives to generate code which minimizes interprocessor communication in distributed systems and maximizes data reuse in all types of systems. The result is increased performance without a sacrifice in portability. As processor speeds continue to increase, this ability to effectively specify and exploit data locality in all types of hierarchical memory systems is becoming increasingly important.

This article describes the implementation of a production–quality HPF compiler for a set of parallel machines. The compiler performs source-to-source translation from HPF to Fortran 77 with run-time primitives. The generated code is loosely synchronous single-program multiple-data (SPMD) code.

HPF introduces three distinct data distributions (*block*, *cyclic*(1), and *cyclic*(k)). Support for these distributions is essential for efficiency on dense matrix algorithms on distributed memory machines. How-

Received May 1995
Revised January 1996

© 1997 by John Wiley & Sons, Inc.

Scientific Programming, Vol. 6, pp. 29–40 (1997)
CCC 1058-9244/97/010029-12

ever, naive compilation of a global-name-space HPF program for distributed memory machines under these distributions may cause unacceptable address calculation overhead. A unique approach is described here which eliminates address calculations for *block* distributions and dramatically simplifies address calculations for *cyclic* distributions.

A set of communication primitives has been developed and is used by the HPF compiler to communicate nonlocal data in an efficient, convenient, and machine-independent manner. These primitives take advantage of distribution information provided in HPF directives. For example, primitives move data very efficiently among arrays aligned to the same template. These primitives have been designed to minimize communication overhead. They are treated as true primitives within the compiler, allowing optimizations, such as code motion, to take place which minimize costly interprocessor communication.

The rest of this article is organized as follows. Section 2 presents the HPF language features. Section 3 describes the architecture of the HPF compiler. Section 4 discusses data and computation distribution. Section 5 introduces the communication primitives and describes compiler algorithms for generating calls to the primitives and creating temporary arrays used in communication. Algorithms are given for some communication primitives. Section 6 discusses the run-time support system for intrinsics and subroutine interfaces. Section 7 presents optimization techniques.

Section 8 presents experimental performance results using the pghpf compiler on several benchmark programs. Section 9 summarizes related work and Section 10 provides conclusions.

2 HPF FEATURES

The HPF approach is based on two fundamental observations. First, the overall performance of a program can be increased if operations are performed concurrently by multiple processors. Second, the efficiency of a single processor is highest if the processor performs computations on data elements stored locally. Based on these observations, the HPF extensions to Fortran 90 provide a means for the explicit expression of parallelism and data mapping. An HPF programmer can express parallelism explicitly, and using this information the compiler may be able to tune data distribution accordingly to control load balancing and minimize communication. Alternatively, given an explicit data distribution, an HPF compiler may be able to identify operations that can be executed concurrently. Typically, the programmer will write HPF code that incorporates both data-mapping directives and explicit parallel constructs in order to maximize the information available to the compiler during code generation.

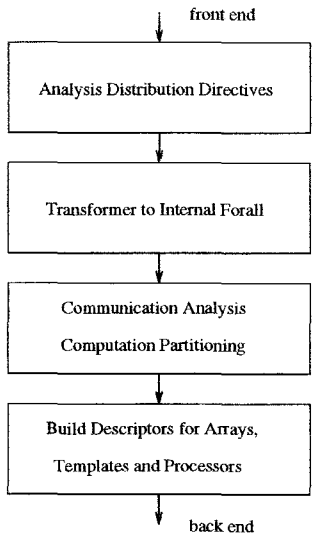


FIGURE 1 The architecture of the HPF compiler.

An HPF program has essentially the same structure as a Fortran 90 program, but is enhanced with data distribution and alignment directives. When writing a program in HPF, the programmer specifies computations in a global data space. Array objects are aligned to abstract arrays with no data, called templates. Templates are distributed according to distribution directives in *block* and *cyclic* fashion. All arrays aligned to such templates are implicitly distributed.

Parallelism can be explicitly expressed in HPF using the following language features: Fortran 90 array assignments, masked array assignments, WHERE statements and constructs, HPF FORALL statements and constructs, HPF INDEPENDENT directives, intrinsic functions, the HPF library, and EXTRINSIC functions [4]. The compiler uses these parallel constructs and distribution information to produce code suitable for parallel execution on each node of a parallel computer.

3 ARCHITECTURE OF THE COMPILER

The major phases of the HPF compiler are shown in Figure 1. The first phase semantically analyzes all HPF directives and stores the resulting information on templates, distribution, alignment, and processor arrangements in the array descriptor portion of the symbol table. This information is used throughout compilation. For variables that are not explicitly mapped (and compiler-created temporaries), the

```
      integer, dimension(100) :: A, B,C
!HPF$ align with B :: A
!HPF$ align with C :: B
!HPF$ distribute C(cyclic)
```

FIGURE 2 An example of HPF directives.

compiler chooses a default distribution and alignment. Figure 2 shows a typical sequence of HPF directives used to align and distribute arrays. The compiler identifies alignment chains and determines that no PROCESSORS directive is present. In the absence of a PROCESSORS directive, the compiler generates code to dynamically determine the number of available processors and uses a default processor arrangement.

The compiler stores the distribution information as shown in Figure 2. It creates a default template T for array C and collapses all alignments to C into alignments to T. The template T is distributed by default onto P, the default processor arrangement.

The next phase of the compiler transforms all parallel constructs: DO loops, array assignments, WHERE statements/constructs, and FORALL statements/constructs into one internal representation which is very similar to a FORALL statement [5]. After transformation, subsequent phases of the compiler (such as optimization) process parallel constructs using only the internal representation.

The communication phase of the compiler selects the communication primitives. It inserts code for allocation of buffers as well as calls to communication primitives. This phase also partitions computations by modifying the bounds of parallel loops and inserting conditional statements which restrict execution of statements to the appropriate processors. Numerous communication optimizations are also performed during this pass. This phase is discussed further in Section 4.

The last phase of the compiler builds descriptors for each processor arrangement, template, and array so that all information available at compile-time is also available at run-time. Processor descriptors include information on the shape and mapping of the processor arrangement. A template data structure describes the shape and distribution of the template. An array descriptor contains all the information necessary to determine the shape of the array, the template to which it is aligned, and how the alignment is specified.

Finally, the code generator produces loosely synchronous SPMD code [6]. The generated code is structured as alternating phases of local computation and calls to communication primitives. Communication primitives are synchronization points. Most of the time the compiler does not know until run-time which groups of processors will communicate for a given parallel construct, so it guarantees that communication primitives will be called by all processors but only sender and receiver processors will perform the communication.

4 PARTITIONING

Array distributions on distributed memory machines generally fall into two categories: canonical and dimensional distributions. Canonical distributions spread the elements on the array across the machine independently of the dimensionality of the array being distributed. This approach may cause more communication on distributed memory machines. With dimensional distributions, each array dimension is distributed independently of all other dimensions. HPF uses dimensional distributions. This approach has better load-balance control and may cause less communication, but complicates address calculation. Address calculation for multidimensional arrays requires numerous integer operations, including several integer divides, multiplications, and additions. This may cause unacceptable address calculation overhead. The challenge is to support dimensional distribution while minimizing its overhead.

4.1 Data Partitioning

A distribution of an array is defined as a two-level mapping in HPF. First, arrays are aligned with templates, then templates are distributed onto the processor grid. The template distribution functions provided by HPF are *block*(m), *cyclic*(m), and replication (denoted by a '*'). In this article, we will discuss *block*(m) distribution only. Figure 4 gives a set of HPF directives that we will use to explain *block*(m) distribution.

Equation 1 of Figure 4 gives the set of local elements of a distributed template T on a particular processor p, denoted by $local^T(p)$ [7]. The calculation presented

```
      integer, dimension(100) :: A, B,C
!HPF$ processors P(number_of_processors())
!HPF$ template T(100)
!HPF$ align with T :: A, B, C
!HPF$ distribute T(cyclic) onto P
```

FIGURE 3 An example of HPF directives.

```
real A(L:U)
!HPF$ processor P(4)
!HPF$ template T(N)
!HPF$ distribute T(block(M)) onto P
!HPF$ align A(i) with T(s*i+o)
```

template distribution	alignment of array to template
$local^T(p) = (L^T : U^T)$ (1)	$local^A_T(p) = (L^A_T : U^A_T)$ (2)
$L^T(p) = (p-1) * M + 1$	$L^A_T(p) = max(L, \lceil \frac{L^T(p)-o}{s} \rceil)$
$U^T(p) = min(p * M, N)$	$U^T(p) = min(U, \lfloor \frac{U^T(p)-o}{s} \rfloor)$

FIGURE 4 An example of distribution calculation.

by Equation 1 will be performed once and used for distribution of all arrays aligned with template T.

Equation 2 gives the set of local elements on processor p of an array $A(L:U)$, denoted by $local^A_T$, which is aligned with the alignment target T by the alignment function $f(i) = s * i + o$ where s and o represent stride and offset, respectively.

Equation 2 is based on the observation that array element $A(i)$ will be owned by processor p if and only if processor p owns $T(i')$ where $i == f^{-1}(i')$, i.e.,

$$local^A_T = f^{-1}(local^T(p)) \cap (L:U)$$

The nice thing about the presented calculation is that local elements of array A on processor p are represented with a lower bound and upper bound pair. This can easily be implemented with the Fortran 90 *allocate* statement. As shown in the equations, all calculations use the array's global index space. The allocate statement will allocate the aligned arrays with respect to their global index space without allocating the entire array in a processor. This approach eliminates the overhead of address calculation between global to local and local to global index space since the local indices are identical to the global indices.

4.2 Computation Partitioning

The source of parallelism using data parallelism is the partitioning of the computation among the processors. One of the most common methods of computation partitioning is to use the owner-computes rule [5, 8, 9]. This implies that assignment statements involving distributed arrays are executed exclusively by those processors which are owners of the lefthand side (lhs) variable. The set of elements of an assignment which have to be computed on a particular processor p is referred to as an execution set, denoted by $exec(p)$. The execution set calculation is shown in Figure 5 for $block(m)$ distribution. For an assignment to a regular array section $A(L:U:S)$ on a particular processor p, the execution set can be determined by the intersection of $(L:U:S)$ and the local set of A, $local^A(p)$, on this processor. Again, the calculation of the execution set is performed according to global index space. This eliminates global to local and local to global conversion overhead. This calculation can be applied to each dimension of the *lhs* independently of all other dimensions.

5 COMMUNICATION

In HPF, data distribution can be specified so that corresponding elements of arrays aligned to the same template are allocated on the same processor. In this way, the programmer can use templates to reduce communication and scheduling overheads without increasing the load imbalance significantly. *Pghpf* uses a set of communication primitives designed to take advantage of cases where arrays are aligned to the same template. There are several communication possibilities.

5.1 no_communication

Arrays aligned to the same template may be located in the same processor depending on their alignment and subscript relations [10]. The compiler pairs the dimensions of a source array and a destination array if they are aligned to the same dimension of the template. Let the pair be (*lhs*, *rhs*). The compiler applies an affine alignment function to this pair. If the result is (i, i) or (s, s), where i is a parallel loop index and

$$exec^A(p) = local^A(p) \cap (L:U:S)$$
$$exec^A(p) = (L^A_T : U^A_T) \cap (L:U:S)$$
$$exec^A(p) = (max(L^A_T, L) : min(U^A_T, U) : S)$$

FIGURE 5 The execution set calculation.

s is a scalar, then the compiler marks that dimension of the source array as *no_communication*. All replicated and collapsed dimensions are also marked as *no_communication*. If all dimensions are marked as *no_communication*, the compiler will not generate any communication for the source array. This is the optimal case, no communication and no temporary arrays are needed.

5.2 overlap_shift

The compiler takes advantage of the overlap concept [11] whenever possible. It creates an overlap area around the local segment of a source array to store nonlocal segments of the source array received during communications. The primitive that corresponds to the overlap concept is called *overlap_shift*. To detect *overlap_shift*, the compiler applies an affine alignment function to a source/destination pair as defined above. If the result is $(i, i + c)$, where c is a compile-time constant, then the overlap concept can potentially be applied. In theory, c could be any scalar variable, but allowing arbitrary scalars could lead to significant memory usage problems. So c is restricted to a small set of values. If all dimensions of the source array are marked *no_communication* or *overlap_shift*, the compiler generates an *overlap_shift* call. This primitive is very efficient since its overhead is low and there is no unnecessary local copying. In this case, the only temporary storage used is that required for the overlap areas.

5.3 collective_communication

This primitive handles those cases where the result of the affine alignment function is $(i, i + s)$, (i, d), or (d, s), where d and s are arbitrary scalars. These results correspond to the *shift*, *multicast*, and *transfer* communication patterns, respectively [10]. *transfer* requires that the elements of a subarray of the source be copied into a temporary which is distributed and aligned with the destination array in such a way that the computations require no communication. Internally, these communications are viewed in relation to the template to which both arrays are aligned. An example of *transfer* is shown in Figure 6a. *multicast* broadcasts data along one dimension of a processor arrangement as shown in Figure 6b. *shift* shifts data an arbitrary distance along one dimension of a processor arrangement, and is the general case of the *overlap_shift* primitive. The compiler generates a call to the *collective_communication* primitive if all dimensions are marked with some combination of *no_communication*, *overlap_shift*, *shift*, *multicast*, or *transfer*. This primitive always requires that temporary arrays be allocated to hold the communicated source subarrays. However, the scalar dimensions are eliminated to conserve memory.

If the compiler can't generate any of the above primitives, it attempts to detect whether the *copy_section* primitive can be used. This primitive copies a source array section to a destination array section as shown in Figure 6c. The source and destination array sections can be aligned to distinct templates in this case, which implies that they can have distinct distributions. The compiler tries to apply this primitive when the result of the affine alignment function is $(i, a * i + b)$ where a and b are scalars. This primitive only applies of the loop index i appears in the same subscript of the source and destination arrays. It does not apply when i appears in more than one subscript of either the source or destination array. As with *collective_communication*, the compiler creates temporary arrays when this primitive is used. The rank of each temporary is determined according to the corresponding source array, with all scalar dimensions being eliminated. The temporaries are distributed and aligned with the destination array in such a way that any computations and the subsequent assignments can be performed internal to the processors. The primitive

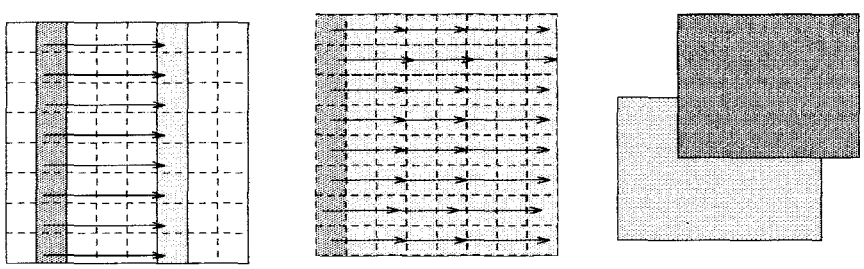


FIGURE 6 Communication primitives.

copies data from the source arrays into each temporary before computation. If there is only one source array on the righthand side of the parallel loop, the source is copied directly to the destination without storing into a temporary.

Additional communication primitives are available to handle transpose, diagonal, and indirect accesses [12, 13]. However, these primitives are not applicable for cases in which multiple loop indices are associated with a particular array subscript or when a subscript is a higher-order function of a parallel index variable. These cases are currently handled by scalarization of the parallel loop.

6 RUN-TIME SUPPORT SYSTEM

The HPF language allows the programmer to write an HPF program with imprecise compile-time information. This situation arises when PROCESSORS, ALIGNMENT, or DISTRIBUTION directives, or ALLOCATE statements, generated to support compiler-created temporaries, depend on run-time variables. An example would be the directive **!hpf distribute templ(cyclic(k))** where k is a run-time parameter. Therefore, the HPF compiler postpones distribution and address calculation until run-time. During the compilation process, the compiler analyzes the global name space. This simplifies the logic in the compiler and shifts the complexity to the run-time support software.

A set of compact data structures was designed to store information on array sections, alignment, templates, and processors. These data structures are called distributed array descriptors (DADs) [14]. DADs pass compile-time information to the run-time system and information between run-time primitives. The run-time primitives query alignment and distribution information from a DAD and act on that information.

Many basic data-parallel operations in Fortran 90 are supported through intrinsic functions which rely on the run-time software. The intrinsics not only provide a concise means of expressing operations on arrays, but they also identify parallel computation patterns that may be difficult to detect automatically. Fortran 90 provides intrinsic functions for operations such as shift, reduction, transpose, reshape, and matrix multiplication. Pghpf parallelizes these intrinsics. Arrays may be redistributed across subroutine boundaries. A dummy argument which is distributed differently from its actual argument in the calling routine is automatically redistributed upon entry to the subroutine by the compiler, and is automatically redistributed to its original distribution at subroutine exit. These operations are performed by the *copy_in* and *copy_out* primitives. These primitives also copy noncontiguous memory spaces specified in array section actual arguments into a contiguous dummy argument. The compiler takes the advantages of *intent*(*in*) and *intent*(*out*) specifiers by not calling *copy_out* and *copy_in*, respectively, where appropriate.

The data-parallel programming model naturally results in the creation of many arrays with identical distributions and alignments. The HPF run-time support system is designed to take advantage of data-parallel problems. Run-time primitives share run-time data structures globally. The compiler provides information to the run-time through interprocedural analysis (IPA). By sharing the run-time data structures, the overhead of creating many identical data structures is eliminated. Analysis extends even to the communication primitives. The overhead of scheduling communications is reduced by reusing scheduling data structures for identical communications. Certain portions of the run-time, in particular those that perform the actual message passing and synchronize the processors, are optimized for the underlying hardware architecture.

7 OPTIMIZATION TECHNIQUES

Several types of communication and computation optimizations can be performed to generate more efficient code. In terms of computation optimization, the scalar node compiler performs a number of classic scalar optimizations within basic blocks. These optimizations include common subexpression elimination, copy propagation (of constants, variables, and expressions), constant folding, useless assignment elimination, and a number of algebraic identities and strength-reduction transformations. However, to use parallelism within the single node (e.g., using an attached vector unit), the compiler propagates information to the node compiler using node directives. Since there is no data dependency between different loop iterations in the original data-parallel constructs such as *forall* statement, vectorization can easily be performed by the node compiler.

Pghpf performs several optimizations to reduce the total cost of communication [10, 15, 16]. This section lists these optimizations.

7.1 Message Aggregation

One of the important considerations for message passing on distributed memory machines is the setup time required for sending a message. Typically, this cost is

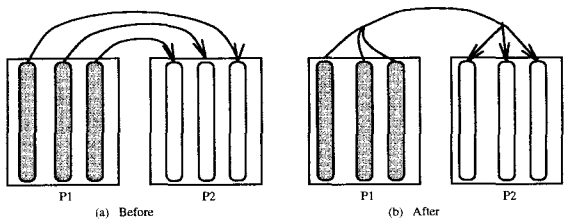


FIGURE 7 Message aggregation.

equivalent to the sending cost of hundreds of bytes. The communication library routines try to aggregate messages [9, 10, 16] into a single larger message, possibly at the expense of extra copying into a continuous buffer. A communication routine first calculates the largest possible array section from this processor to the other processors. These may indicate several continuous blocks of data. Then, it tries to sort the continuous data by destination. Then it aggregates noncontinuous array sections (messages) into a continuous message buffer. Messages with an identical destination processor can then be collected into a single communication operation as shown in Figure 7. The gain from message aggregation is similar to communication vectorization in that multiple communication operations can be eliminated at the cost of increasing message length.

7.2 Evaluating Expression

A drawback to blindly applying the owner_computes rule is that the amount of data motion could be much greater than necessary. For example, consider the following program fragment:

```
      REAL A(N), B(N), C(N), D(N), E(N)
!HPF$ distribute E(BLOCK)
!HPF$ distribute (CYCLIC) :: A, B, C, D
      E = (A + B) * (C + D)
```

In a compiler that blindly applies the owner-computes rule, A, B, C, and D will be redistributed into temporaries that match the distribution of E, then the computation will be performed locally. This may cause four different communications with four different temporaries.

A more intelligent approach might perform the sum of A and B locally into a cyclically distributed temporary, then perform the sum of C and D locally into another cyclically distributed temporary. Next the result is the multiply of these two temporaries locally into a cyclically distributed third temporary. Finally, the result is redistributed (comunicated) into E. This approach may cause one communication with three temporaries (shown in Fig. 8b).

To apply the optimization, the compiler has to evaluate the expression according the partial order induced by the expression tree (shown in Fig. 8a). However, Li and Chen [17] show that the problem of determining an optimal static alignment between the dimension of distinct arrays is NP-complete. Chatterjee et al. [18] and Bouchitte et al. [19] propose some heuristic algorithms to evaluate the expression tree with minimal cost. Thus, this is a worthwhile optimization to implement.

7.3 Communication Parallelization

HPF allows arrays to be replicated in one or more dimensions. When a block of source data is replicated,

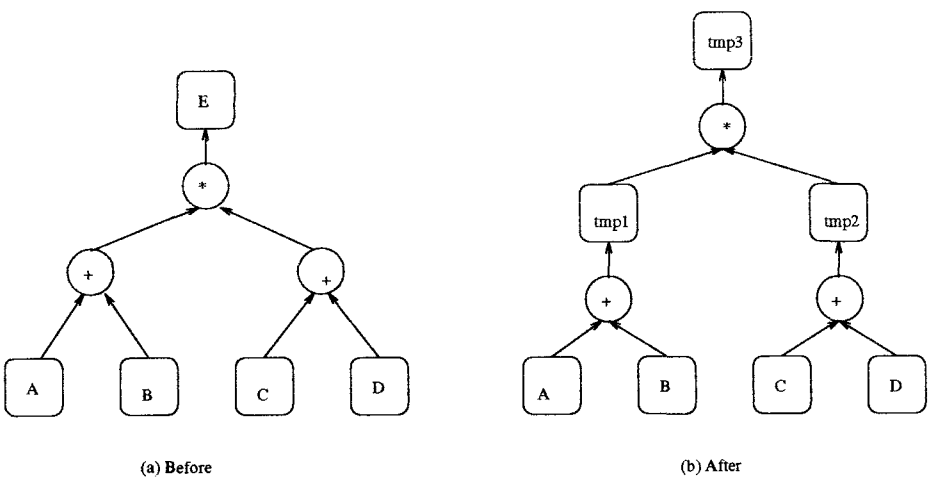
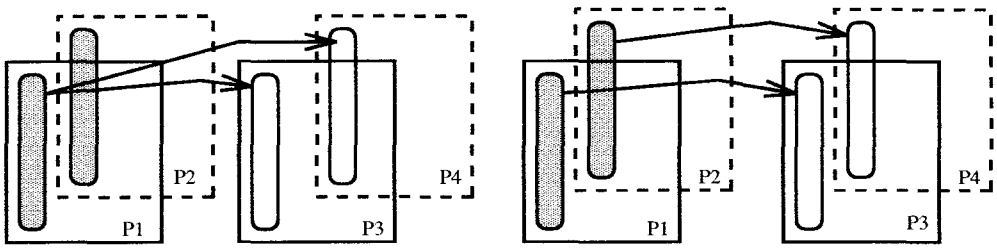


FIGURE 8 Expression evaluation trees.



(a) Before (b) After

FIGURE 9 Parallel communication.

any or all of the processors owning the block can take part in the communication. Alternatively, one of the source processors is chosen to send to all processors owning the destination block. Ideally, the sends should be spread out over as many source processors as possible to utilize the available communication bandwidth (Fig. 9).

The basic idea of this optimization is to divide the set of destination processors among the set of source processors. Each source does a multicast to its assigned subset of destinations. The source and destination sets are computable from information in the template and processor data structures.

7.4 Communications Union

In many cases, communication required for two different operands can be replaced by their union [10]. Clearly, the advantage of communication union is that it reduces the number of communication statements and thus the number of messages. For example, the following code may require two *overlap_shifts*. However, with simple analysis, the compiler can eliminate the shift of size 2.

```
FORALL(I=1:N) A(I)=B(I+2)+B(I+3)
```

The communication union optimization can be applied in a statement as well as between statements. However, the compiler needs a data-flow analysis infrastructure to perform interstatement communication union.

```
!hpf$   distribute (*,block) :: a
!hpf$   distribute (block) :: row
        do k = 1, N
          ...
          forall (i = 1:N, j = k:N, indx(i) .EQ. -1)
     &          a(i,j) = a(i,j) - fac(i)*row(j)
        endo
```

FIGURE 10 Main factorization loop in gauss.

7.5 Reuse of Scheduling Information

The communication routines perform send and receive set calculations according to a scheduling data structure. The schedule, once computed, can also carry out identical patterns of data exchanges on several different, but identically distributed arrays on array sections [12, 20, 21]. The same schedule can be reused repeatedly to carry out a particular pattern of data exchange. In this case, the cost of generating the schedules can be amortized by only executing it once. This analysis can be performed at compile_time. Hence, if the compiler recognizes that a schedule can be reused, it does not need to generate code for scheduling but rather passes a pointer to the already existing schedule. Furthermore, the scheduling computation can be moved up as much as possible by analyzing definition-use chains [22]. Reduction in communication overhead can be significant if the scheduling code can be moved out of one or more nested loops by this analysis.

In addition, *pghpf* performs invariant communication hoisting, reuse of communicated data, and loop fusion.

8 EXPERIMENTAL RESULTS

Benchmark results from five programs are presented to illustrate performance obtained using the HPF compiler. All of these benchmarks were run on a 15-processor Intel Paragon. The processors run at 50 MHz, and each node has 32 MByte of physical memory. The programs were compiled using the 2.0 version of the pghpf compiler with all of pgf77's node-

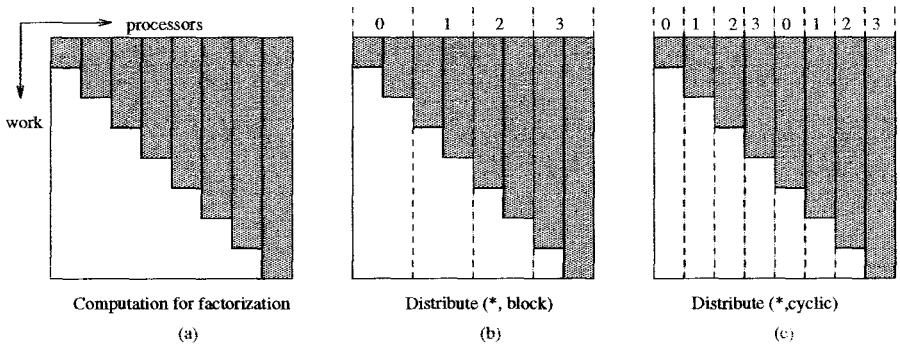


FIGURE 11 Load balancing for gauss.

level optimizations turned on, including the i860 vectorizer. These results are early results of *pghpf*. *Pghpf* performance will continue to improve.

The shallow water (*shallow*) benchmark is a 300-line program abstracting a two-dimensional (2D) flow system. The data are distributed block fashion in one dimension, (*, block). The generated code consists of many computations of order N^2 with communication mostly consisting of overlap shifts of order N. Figure 12 shows the performance of *shallow*. The superlinear speedup on the large data set dramatically exhibits the ability of the HPF compiler to make large problems more tractable simply through efficient use of the larger available core memory on a multiprocessor system.

The partial differential equation benchmark (*pde1*) is a 360-line program from the Genesis Parallel Benchmark Suite that implements a 3D Poisson Solver using red–black relaxation through five iterations. Figure 13 shows the performance of pde1. The data are distributed block fashion in one dimension (*,*,block). Good scalability is exhibited. The communication mostly consists of overlap shifts due to the stencil computations of pde1.

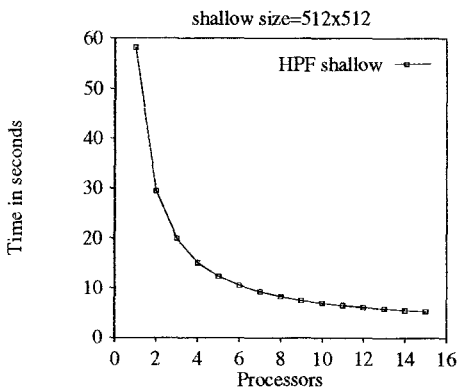


FIGURE 12 Shallow Performance.

The hydflo benchmark is a hydrodynamics program with 2,000 lines. Figure 14 shows the performance of hydroflo. The data are distributed block fashion in one dimension (*,*,block). Good scalability is exhibited. The communication mostly consists of copy-section and collective-communication.

A significant advantage of coding in HPF is the ability to specify different distribution directives and measure performance differences without extensive recording. Some experimentation along these lines was performed on the Gauss benchmark (*gauss*), which is a short program designed to measure the performance of a Gaussian elimination algorithm.

Figure 10 gives the main factorization loop of *gauss* which converts matrix a to upper triangular form. This Gaussian elimination algorithm is suboptimal due to a mask in the inner loop which prevents vectorization. Figure 11a shows the updated values of matrix a in the shaded region after the factorization loop. Since the compiler uses the owner_computes rule to assign computations, only owners of data in the shaded region will participate in the computation. The remaining processors are masked out of the computation. Figure 11b,c show the computation distribution on four processors in block and cyclic fashions, respectively. In this particular benchmark, cyclic distribution results in better load balancing than block distribution. Figure 15 presents the performance using cyclic as well as block distributions. As expected, the cyclic distribution exhibits better performance because of load balancing. The communication requirements for these distributions are identical. Both use a multicast communication primitive.

As shown by the data, benchmark programs written in HPF can achieve reasonable efficiency given a problem of reasonable size. The figures show good scalability when increased numbers of processors are used.

9 SUMMARY OF RELATED WORK

The programming language Fortran D [1] proposes a Fortran language extension in which the programmer specifies the distribution of data by aligning each array to a virtual array, known as a decomposition, and then specifying a distribution of decomposition to a virtual machine. Fortran 77D [15] and Fortran 90D/HPF [5] compilers are developed based on Fortran D by Rice University and Syracuse University. The Fortran 77D compiler introduces and classifies a number of advanced optimizations needed to achieve acceptable performance; they are analyzed and empirically evaluated for stencil computations. The Fortran 90D/HPF compiler introduces techniques for data and computation partitioning, communication detection, and generation to develop a portable compiler. It gives a methodology to HPF implementors.

SUPERB [23] is a semiautomatic parallelization tool designed for multiple-instruction multiple-data (MIMD) distributed memory machines. It supports arbitrary user-specified contiguous rectangular distributions and performs dependence analysis to guide interactive program transformations. Recently, SUPERB has been adapted for a new language called Vienna Fortran 90. Vienna Fortran 90 [2] is a language extension of Fortran 90 which enables the user to write programs for distributed memory machines using global reference only.

KALI [12] is the first compiler system that supports both regular and irregular computations on MIMD machines. KALI requires that the programmer explicitly partition loop iterations onto the processor grid. An inspector/executor strategy is used for run-time preprocessing of the communication.

Data parallel C [24] is a variant of the original C* programming language designed by Thinking Machines Corporation for the Connection Machine (CM) processor array. Data parallel C extends C to provide the programmer access to a parallel virtual machine based on SIMD semantics. These attempt to minimize interprocessor synchronization.

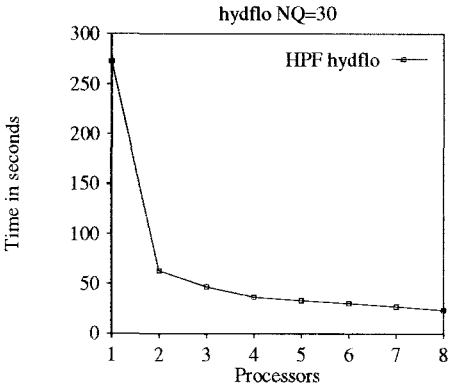


FIGURE 14 Hydflo Performance.

The MIMDizer [25] and ASPAR [26] (within the Express system) are two commercial systems which support the task of generating parallel code. The MIMDizer performs a good deal of program analysis and provides block and cyclic distributions for array dimensions. ASPAR employs pattern-matching techniques to detect common stencils in the code, from which communications are generated.

The PARTI primitives, a set of run-time library routines to handle irregular computations, have been developed by Saltz and coworkers [27]. These primitives have been integrated into an ARF compiler for irregular computations [28] as well as into other systems to handle a wide range of irregular problems in scientific computing.

The ADAPT system [29] compiles Fortran 90 for execution on MIMD distributed memory architectures. ADAPTOR [30] is a tool that transforms data-parallel

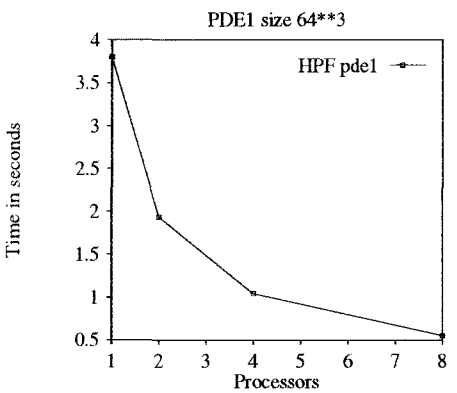


FIGURE 13 PDE1 Performance.

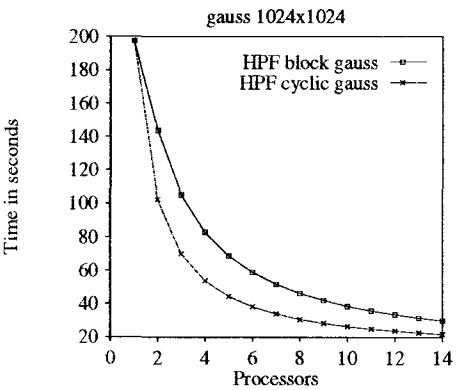


FIGURE 15 Gauss Performance.

programs written in Fortran with array extensions and layout directives to explicit message passing.

CM Fortran language [31, 32] implemented is a subset of Fortran 77, extended by Fortran 8x array features to support a data-parallel programming style for the CM computer system. Sabot [33] describes techniques used by the CM compilers to map the fine-grained array parallelism of languages such as Fortran 90 and C* onto the CM architectures.

The PREPARE HPF compiler [7] is implemented on an engine-like basis. Engines can concurrently work on a common internal program representation called the prepare intermediate representation (PIR). The HPF program to be compiled is transformed into the PIR by the HPF front-end engine. Later, the parallelization engine transforms the PIR into a form that can be efficiently executed on the target parallel machine.

The PTRAN II compiler [34] is a prototype compiler for HPF that is developed as a testbed for experimenting with distributed memory compilation techniques, including automatic data partitioning and parallelization, cost modeling, and global communication optimization.

10 CONCLUSIONS

HPF is rapidly gaining acceptance as an easy-to-use and portable programming language for high-performance scientific applications. The wide variety of current parallel system architectures, however, presents a significant challenge to HPF compiler implementors. Here, we have explored some of the issues and outlined our compilation and execution techniques.

As shown by our experimental results, HPF benchmark programs compiled by *pghpf* can achieve reasonable efficiency given a problem of reasonable size on distributed memory parallel systems. The figures show good scalability when increased numbers of processors are used. The loosely coupled SPMD codes produced by *pghpf* are adaptable to a variety of parallel system architectures.

PGI is committed to making a commercially viable HPF compiler product. The results we have been able to achieve thus far and the potential for further performance we see as we continue our development project give us confidence in this endeavor.

ACKNOWLEDGMENTS

The authors thank Alok Choudhary, Geoffrey Fox, Tomasz Haupt, and Sanjay Ranka of Syracuse University for their suggestions and encouragement. We are also grateful to the other members of our HPF compiler group Duane Dillerud, Doug Miles, John Mudd, Tom Van Raalte, and Cliff Walinsky for their assistance in implementing the pghpf compiler.

REFERENCES

[1] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "Fortran D language specification," Rice and Syracuse University, Tech. Rep. TR90079, 1992.

[2] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," *Sci. Prog.*, vol. 1, pp. 31–50, 1992.

[3] High Performance Fortran Forum, "High performance Fortran language specification," Rice University, Houston, TX, Tech. Rep. Version 1.0, May 1993.

[4] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel, *The High Performance Fortran Handbook*, Boston: MIT Press, 1994.

[5] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M. Wu, "Compiling Fortran 90D/HPF for distributed memory MIMD computers," *J. Parallel Distrib. Comput.*, vol. 21, pp. 15–26, April 1994.

[6] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors*, vol. 1–2, Englewood Cliffs, NJ: Prentice Hall, 1988.

[7] S. Benkner, P. Brezany, and H. Zima, "Compiling high performance Fortran in the prepare environment," presented at the Fourth Int. Workshop on Compilers for Parallel Computers, Delft University of Technology, The Netherlands, 1993.

[8] B. Chapman, H. Herbeck, and H. Zima, "Automatic support for data distribution," *IEEE Trans. Computers*, pp. 51–57, 1991.

[9] C. Tseng, "An optimizing Fortran D compiler for MIMD distributed-memory machines," PhD thesis, Rice University, 1993.

[10] J. Li and M. Chen, "Compiling communication-efficient programs for massively parallel machines," *IEEE Trans. Parallel Distrib. Systems*, vol. 2, pp. 361–376, July 1991.

[11] M. Gerndt, "Updating distributed variables in local computations," *Concurrency Practice Exp.*, vol. 2, pp. 171–193, Sept. 1990.

[12] C. Koelbel and P. Mehrotra, "Supporting compiling global name-space parallel loops for distributed execution," *IEEE Trans. Parallel Distrib. Systems*, Oct. 1991.

[13] J. Saltz, H. Berryman, and J. Wu, "Multiprocessors and run-time compilation," *Concurrency Practice Exp.*, vol. 3, pp. 573–592, Dec. 1991.

[14] M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Trans. Parallel Distrib. Systems*, vol. 3, pp. 179–193, March 1992.

[15] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiler optimization for Fortran D on MIMD distributed-memory machines," in *Proc. Supercomputing '91*, 1991.

[16] D. Palermo, E. Su, J. Chandy, and P. Banarjee, "Communication optimizations used in the paradigm compiler for distributed-memory multicomputers," presented at the Int. Conf. on Parallel Processing, St. Charles, IL, 1994.

[17] J. Li and M. Chen, "The data alignment phase in compiling programs for distributed-memory machines," *J. Parallel Distrib. Comput.*, vol. 13, pp. 213–221, Oct. 1991.

[18] S. Chatterjee, J. R. Gilbert, and R. Schreiber, "Optimal evaluation of array expression on massively parallel machines," presented at the Second Workshop on Languages, Compilers, and Runtime Environments for Distributed Memory Multicomputers, Boulder, CO, 1992.

[19] V. Bouchitte, P. Boulet, A. Darte, and Y. Robert, "Evaluating array expressions on massively parallel machines with communication/computation overlap," Ecole Normale Superieure de Lyon, Tech. Rep. 94-10, 1994.

[20] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, "Run-time scheduling and execution of loops on message passing machines," *J. Parallel Distrib. Comput.*, Dec. 1991.

[21] R. Ponnusamy, J. Saltz, and A. Choudhary, "Compilation techniques for data partitioning and communication schedule reuse," in *Supercomputing '93*. New York: IEEE Computer Society Press, 1993.

[22] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1988.

[23] H. Zima, H. Bast, and M. Gerndt, "Superb: A tool for semi automatic SIMD/MIMD parallelization," *Parallel Comput.*, Jan. 1988.

[24] P. Hatcher, A. Lapadula, R. Jones, M. Quinn, and R. Anderson, "A production-quality C* compiler for hypercube multicomputers," in the *Third ACM SIGPLAN Symp. on PPOPP*, 1991, p. 73.

[25] Pacific Sierra Research Corporation, *MIMDizer User's Guide, Version 7.02*. Pacific Sierra Research Corp., 1991.

[26] K. Ikudome, G. Fox, A. Kolawa, and J. Flower, "An automatic and symbolic parallelization system for distributed memory parallel computers," presented at the 5th Distributed Memory Computing Conference, Charleston, SC, 1990.

[27] J. Saltz, R. Das, and H. Berryman, "A manual for PARTI runtime primitives," in NASA, ICASE Interim Report 17, May 1991.

[28] J. Saltz, J. Wu, H. Berryman, and S. Hiranandani, "Distributed memory compiler design for sparse problems," Interim Report ICASE, NASA Langley Research Center, 1991.

[29] J. H. Merlin, "Techniques for the automatic parallelisation of 'Distributed Fortran 90'," Southampton Novel Architecture Research Centre, Tech. Rep. SNARC 92-02, 1992.

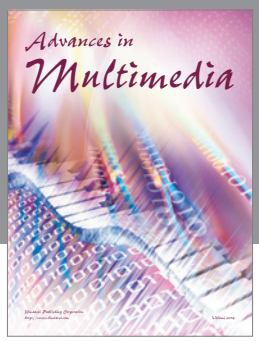
[30] T. Brandes, "ADAPTOR language reference manual," German National Research Center for Computer Science, Tech. Rep. ADAPTOR-3, GMD, 1992.

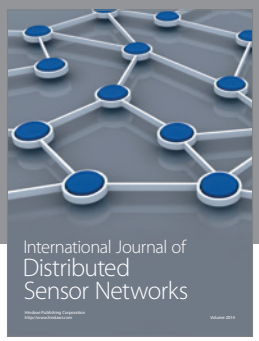
[31] K. Knobe, J. D. Lukas, and G. L. Steele, "Compiling Fortran 8x array features for the Connection Machine computer systemication on SIMD machines," in the *ACM SIGPLAN Symp. on Parallel Programming: Experience with Applications, Languages, and Systems*, 1988.

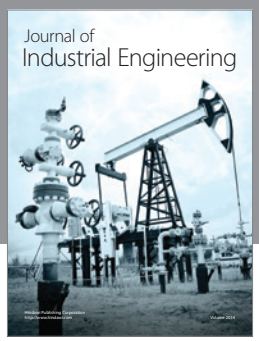
[32] The Thinking Machine Corporation, *CM Fortran User's Guide Version 0.7-f*. Cambridge, MA: Thinking Machine Corp., 1990.

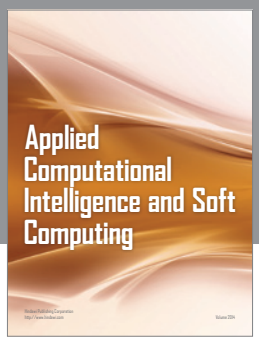
[33] G. Sabot, "A compiler for a massively parallel distributed memory MIMD computer," in the *Fourth Symp. on the Frontiers of Massively Parallel Computation*, pp. 12–20, 1992.

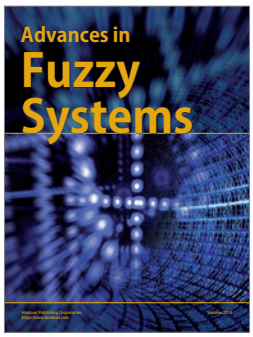
[34] M. Gupta, S. Midkiff, E. Schonberg, P. Sweeney, K. Y. Wang, and M. Burke, "PTRAN II-A compiler for high performance Fortran, presented at the Fourth International Workshop on Compilers for Parallel Computers, Delft University of Technology, The Netherlands, 1993.

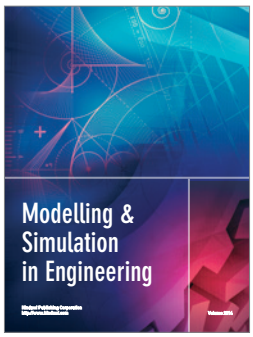


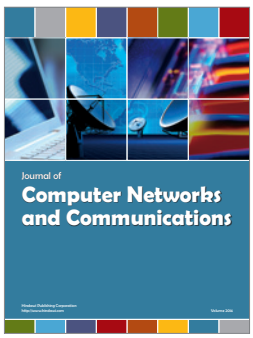












Hindawi
Submit your manuscripts at
http://www.hindawi.com