1

COMPILING HPF FOR DISTRIBUTED MEMORY MIMD COMPUTERS

Zeki Bozkus, Alok Choudhary*, Geoffrey Fox, Tomasz Haupt and Sanjay Ranka**

Northeast Parallel Architectures Center
Syracuse University, Syracuse, NY, 13244-4100

** Computer Engineering Dept. Syracuse University*
*** Computer Science Dept. Syracuse University*

{*zbozkus, choudhar, gcf, haupt, ranka*}*@npac.syr.edu*

ABSTRACT

This paper describes the design of a High Performance Fortran (HPF/Fortran 90D) compiler, a source-to-source translator for distributed memory systems. HPF is a data parallel language with compiler directives to enable users to specify data alignment and distributions. A systematic methodology to process distribution directives of HPF is presented. Furthermore, techniques for data and computation partitioning, communication detection and generation, and the run-time support for the compiler are discussed. Finally, initial performance results for the compiler are presented which show that the code produced by the compiler is portable, yet efficient. We believe that the methodology to process data distribution, computation partitioning, communication system design and the overall compiler design can be used by the other HPF compiler implementors.

1 INTRODUCTION

Nowadays, when increasing the speed of processors become more and more difficult, more and more computer experts admit that the future of high performance computing belongs to parallel computers. Many machines that allow for concurrent execution are commercially available for several years. Nevertheless, this is a very rapidly developing technology, and vendors come with newer, better concepts almost every year. Hardly ever parallel computers coming from different vendors have a similar architecture. To exploit specific features of the machine, vendors develop specific extensions to existing languages (Fortran, C, ..., etc.) and/or develop vendor specific runtime libraries for interprocessor communication. As a result, codes developed on these machines are not portable from one platform to another. Even worse, moving to the next version of the machine from the same vendor usually requires recoding to obtain performance. Consequently, it is not surprising that they are not widely used, in particular, for commercial purposes. Users who traditionally require tremendous amount of computing power still prefer conventional supercomputers, recognizing that parallel computing is still a high risk technology, which does not protect software investment.

To overcome this deficiency, we have designed Fortran D language [2] with our colleagues at Rice University. Fortran D is a version of Fortran enhanced with a rich set of data decomposition specifications to provide a simple machine-indepentent programming model for most data-parallel computations. Recently, the High Performance Fortran Forum, an informal group of people from academia, industry and national labs, led by Ken Kennedy, developed a language called HPF (High Performance Fortran) [3] based on Fortran D. Companies that have already committed to developing compilers and/or supporting HPF include Intel, TMC, PGI, DEC, IBM, and others.

The idea behind HPF is to develop a minimal set of extensions to Fortran (ISO/ANSI standard known informally as Fortran 90) to support data parallel programming model, defined as single threaded, global name space, loosely synchronous parallel computation. The purpose of HPF is to provide software tools (i.e., HPF compilers) that produce performance codes for MIMD and SIMD computers with non-uniform memory access cost. The portability of the HPF codes means that the efficiency of the code is preserved for different machines with comparable number of processors.

This paper presents the design of a prototype HPF compiler for distributed memory systems. The compiler transforms codes written in HPF to SPMD (Single Program Multiple Data) program with appropriate data and computation partitioning and communication calls for MIMD machines. Therefore, the user can still program using a data parallel language but is relieved of the responsibility to perform data distribution and communication.

The rest of this paper is organized as follows. Section 2 briefly presents HPF language. The compiler architecture is described in Section 3. Data partitioning, and computation partitioning are discussed in Sections 4. Section 5 presents the communication primitives and communication generation for HPF programs. In Section 6, we present the runtime support system including the intrinsic functions. Some optimization techniques are given in Section 7. Section 8 summarizes our initial experience using the current version of the compiler. It also presents a comparison of the performance with hand written parallel code. Section 9 presents a summary of related work. Finally, summary and conclusions are presented in Section 10.

2 HPF LANGUAGE

The HPF extensions to the Fortran 90 standard fall into four categories: compiler directives, new language features, library routines and restrictions to Fortran 90. The HPF compiler directives are structured comments that suggest implementation strategies or assert facts about a program to the compiler. They may affect the efficiency of the computation performed, but they do not alter the semantics. In analogy to Fortran 90 statements, there are declarative directives, to be placed in the declaration part of a scoping unit, and executable directives, to be placed among the executable Fortran 90 statements. The HPF directives are designed to be consistent with Fortran 90 syntax except for the directive prefix !HPF$, CHPF$ or *HPF$.

The new language features are FORALL statement and construct as well as modifications and additions to the library of intrinsic functions. In addition to the intrinsic functions, HPF introduces new functions that may be used to express parallelism, like new array reduction functions, array combining scatter functions, array suffix and prefix functions, array sorting functions and others. Those functions are collected in a separate library, the HPF library. Finally, HPF imposes some restrictions on Fortran 90 definition of storage and sequence associations.

The HPF approach is based on two key observations. First, the overall efficiency of the program can be increased, if many operations are performed concurrently by different processors, and secondly, the efficiency of a single processor is likely be the highest, if the processor performs computations on data elements stored in its local memory. Therefore, the HPF extensions provide means for explicit expression of parallelism and data mapping. It follows that an HPF programmer expresses parallelism explicitly, and the data distribution is tuned accordingly to control the load balance and minimize communication. On the other hand, given a data distribution, an HPF compiler may be able to identify operations that can be executed concurrently, and thus generate even more efficient code.

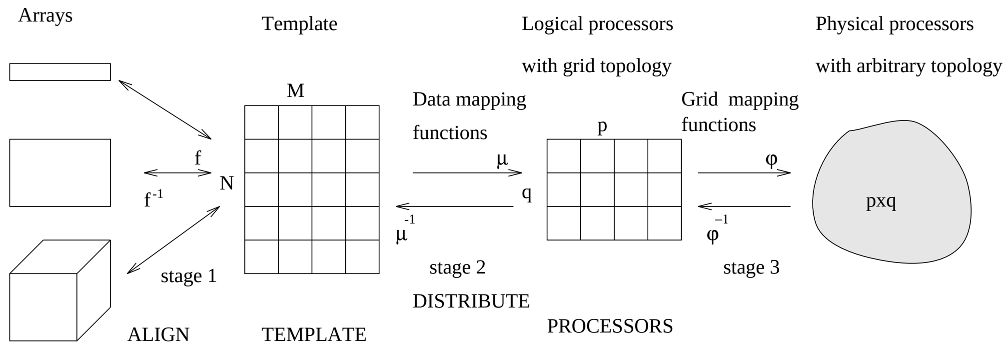


Figure 1 Array mapping model at HPF.

HPF data alignment and distribution directives allow the programmer to advise the compiler how to assign data object (typically array elements) to processors' memories. The model is that there is a two-level mapping of data objects to memory regions (as shown at Figure 1), referred to as "abstract processors": arrays are first aligned relative to one another, and then this group of arrays is distributed onto a user defined, rectilinear arrangement of abstract processors. The final mapping, abstract to physical processors is not specified by HPF and it is language-processor dependent. The alignment itself is logically accomplished in two steps. First, the index space spanned by an array that serves as an align target defines a natural template of the array. Then, an alignee is associated with this template. In addition, HPF allows users to declare a template explicitly; this is particularly convenient when aligning arrays of different size and/or different shape. It is the template (either a natural or explicit one) that is distributed onto abstract processors. This means, that all arrays' elements aligned with an element of the template are mapped to the same processor. This way locality of data is forced. Arrays and other data objects that are not explicitly distributed using the compiler directives are mapped according to an implementation dependent default distribution. One possible choice of the default distribution is replication: each processor is given its own copy of the data.

The data mapping can be declared using declarative directives: PROCESSORS, ALIGN, DISTRIBUTE, and, optionally, TEMPLATE. In addition, arrays may be remapped during the runtime. To this end, array must be declared using DYNAMIC directive, and the actual remapping is triggered by executable directives REALIGN and REDISTRIBUTE.

In HPF, an array may be aligned with another in many ways. The repertoire includes shifts, strides, or any other linear combination of a subscript (i.e., n*i + m), transposition of indices, and collapse or replication of array's dimensions. Skewed or irregular alignments are, however, not allowed. The template may be distributed in BLOCK, CYCLIC, BLOCK(n), and CYCLIC(n) fashion. In addition, any dimension of the template may be collapsed or replicated onto a processor grid (note, that it does not change the relative alignment of the arrays!). The BLOCK distribution specifies that the template should be distributed across set of abstract processors by slicing it uniformly into blocks of contiguous elements. The BLOCK(n) distribution specifies that groups of exactly n elements should be mapped to successive abstract processors, and there must be at least (array size)/n abstract processors if the directive is to be satisfied. The CYCLIC(n) distribution specifies that successive array elements' blocks of size n are to be dealt out to successive abstract processors in round-robin fashion. Finally, CYCLIC distribution is equivalent to the CYCLIC(1) distribution.

3 HPF COMPILER

Our HPF compiler exploits only the parallelism expressed in the data parallel constructs. We do not attempt to *parallelize* other constructs, such as *do* loops and *while* loops, since they are used only as naturally sequential control constructs in this language. The foundation of our design lies in recognizing commonly occurring computation and communication patterns. These patterns are then replaced by calls to the optimized run-time support system routines. The run-time support system includes parallel intrinsic functions, data distribution functions, communication primitives and several other miscellaneous routines. This approach represents a significant departure from traditional approaches where a compiler needs to perform in-depth dependency analyses to recognize parallelism, and embed all the synchronization and low-level communication functions inside the generated code.

The basic structure of our HPF compiler is organized around four major modules–parsing, partitioning, communication detection and insertion, and code generation. Given a syntactically correct HPF program, the first step of the compilation is to generate a parse tree. The front-end to parse Fortran 90 for the compiler was obtained from ParaSoft Corporation. In this module, our compiler also transforms each array assignment statement and *where* statement into equivalent *forall* statement with no loss of information [7]. In this way, the subsequent steps need only deal with *forall* statements.

The partitioning module processes the data distribution directives; namely, template, distribute and align. Using these directives, it partitions data and computation among processors.

After partitioning, the parallel constructs in the node program are sequentialized since it will be executed on a single processor. This is performed by the sequentialization module. Array operations and *forall* statements in the original program are transferred into loops or nested loops. The communication module detects communication requirements and inserts appropriate communication primitives.

Finally, the code generator produces *loosely synchronous* [6] SPMD code. The generated code is structured as alternating phases of local computation and global communication. Local computations consist of operations by each processor on the data in its own memory. Global communication includes any transfer of data among processors, possibly with arithmetic or logical computation on the data as it is transferred (e.g. reduction functions). In such a model, processes do not need to synchronize during local computation. But, if two or more nodes interact, they are implicitly synchronized by global communication.

4 PARTITIONING

The distributed memory system solves the memory bottleneck of vector supercomputers by having separate memory for each processor. However, distributed memory systems demand high locality for good performance. Therefore, the distribution of data and computations across processors is of critical importance to the efficiency of a parallel program in a distributed memory system.

Data Partitioning

Even thoug HPF language models two stage mapping, in our implementation we choosen to map arrays to physical processors by using a three stage mapping as shown in Figure 1 which is guided by the user-specified HPF directives.

Stage 1 : The alignment of arrays to template is determined by their subscript expressions (f an affine function) in the ALIGN directive. The compiler computes f and f^{-1} function from the directive and applies f functions for the corresponding array indices to bring them onto common template index domain. The original indices can be calculated by f^{-1} if they are required. The algorithm to compile align directive can be found in [9].

Stage 2 : Each dimension of the template is mapped onto the logical processor grid, based on the DISTRIBUTE directive attributes. *Block* divides the template into contiguous chunks. *Cyclic* specifies a round-robin division of the template. The mapping functions μ and μ^{-1} to generate relationship between global and local indices are computed.

Stage 3 : The logical processor grid is mapped onto the physical system. The mapping functions φ and φ^{-1} can change from one system to another but the data mapping onto the logical processor grid does not need to change. This enhances portability across a large number of architectures.

By performing the above three stage mapping, the compiler is decoupled from the specifics of a given machine or configuration. Compilation of distribution directives is discussed in detail in [9].

Computation Partitioning

Once the data is distributed, there are several alternatives to assign computations to processing elements (PEs) for each instance of a *forall* statement. One of the most common methods is to use the *owner computes rule*. In the owner computes rule, the computation is assigned to the PE owning the *lhs* data element. This rule is simple to implement and performs well in a large number of cases. Most of the current implementations of parallelizing compilers uses the owner computes rule [12, 13]. However, it may not be possible to apply the owner computes rule for every case without extensive overhead. The following examples describe how our compiler performs computation partitioning.

Example 1 (canonical form) Consider the following statement, taken from the Jacobi relaxation program

```
   forall (i=1:N, j=1:N)
&    B(i,j) = 0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
```

In the above example, as in a large number of scientific computations, the *forall* statement can be written in the canonical form. In this form, the subscript value in the *lhs* is identical to the forall iteration variable. In such cases, the iterations can be easily distributed using the owner computes rule. Furthermore, it is also simpler to detect structured communication by using this form. (This will be elaborated in Section 5.2.)

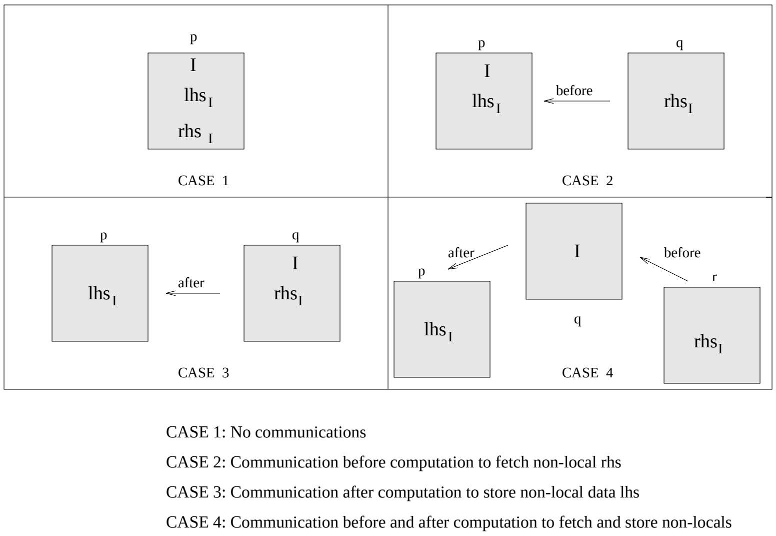


Figure 2 I shows the processor on which the computation is performed. lhs_I and rhs_I show the processors on which the lhs and rhs of instance I reside.

Figure 2 shows the possible data and iteration distributions for the $lhs_I = rhs_I$ assignment caused by iteration instance I. Cases 1 and 2 illustrate the order of communication and computation arising from the owner computes rule. Essentially, all the communications to fetch the off-processor data required to execute an iteration instance are performed before the computation is performed. The generated code will have the following communication and computation order.

```
Communications  ! some global communication primitives
Computation     ! local computation
```

Example 2 (non-canonical form) Consider the following statement, taken from an FFT program

```
 forall (i=1:incrm, j=1:nx/2)
&   x(i+j*incrm*2+incrm) = x(i+j*incrm*2) - term2(i+j*incrm*2+incrm)
```

The *lhs* array index is not in the canonical form. In this case, the compiler equally distributes the iteration space on the number of processors on which the *lhs* array is distributed. Hence, the total number of iterations will still be the same as the number of *lhs* array elements being assigned. However, this type of forall statement will result in either Case 3 or Case 4 in Figure 2. The generated code will be in the following order.

```
Communications  ! some global communication primitives to read
Computation     ! local computation
Communication   ! a communication primitive to write
```

For reasonably simple expressions, the compiler can transform such index expressions into the canonical form by performing some symbolic expression operations [14]. However, it may not always be possible to perform such transformations for complex expressions.

Having presented the computation partitioning alternatives for various reference patterns of arrays on the *lhs*, we now present a primitive to perform global to local transformations for loop bounds.

```
set_BOUND(llb,lub,lst,glb,gub,gst,DIST,dim) ! computes local lb, ub, st
```

The *set_BOUND* primitive takes a global computation range with global lower bound, upper bound and stride. It distributes this global range statically among the group of processors specified by the *dim* parameter on the logical processor dimension. The *DIST* parameter gives the distribution attribute such as *block* or *cyclic*. The *set_BOUND* primitive computes and returns the local computation range in local lower bound, local upper bound and local stride for each processor. The algorithm to implement this primitive can be found in [7].

In summary, our computation and data distributions have two implications.

- The processor that is assigned an iteration is responsible for computing the *rhs* expression of the assignment statement.
- The processor that owns an array element (*lhs* or *rhs*) must communicate the value of that element to the processors performing the computation.

5 COMMUNICATION

Our HPF compiler produces calls to collective communication routines [17] instead of generating individual processor send and receive calls inside the compiled code. There are three main reasons for using collective communication to support interprocessor communication in the HPF compiler.

1. *Improved performance of HPF programs.* To achieve good performance, interprocessor communication must be minimized. By developing a separate library of interprocessor communication routines, each routine can be optimized. This is particularly important given that the routines will be used by many programs compiled through the compiler.
2. *Increased portability of the Fortran 90D/HPF compiler.* By separating the communication library from the basic compiler design, portability is enhanced because to port the compiler, only the machine specific low-level communication calls in the library need to be changed.
3. *Improved performance estimation of communication costs.* Our compiler takes the data distribution for the source arrays from the user as compiler directives. However, any future compiler will require a capability to perform automatic data distribution and alignments [18, 19, 10]. Such techniques usually require computing trade-offs between exploitable parallelism and the communication costs. The costs of collective communication routines can be determined more precisely, thereby enabling the compiler to generate better distributions automatically.

In order to perform a collective communication on array elements, the communication primitive needs the following information 1-) send processors list, 2-) receive processors list, 3-) local index list of the source array and, 4-) local index list of the destination array.

There are two ways of determining the above information. 1) Using a preprocessing loop to compute the above values or, 2) based on the type of communication, the above information may be implicitly available, and therefore, not require preprocessing. We classify our communication primitives into *unstructured* and *structured* communication.

Our structured communication primitives are based on a logical grid configuration of the processors. Hence, they use grid-based communications such as shift along dimensions, broadcast along dimensions etc. The following summarizes some of the structured communication primitives implemented in our compiler.

- **transfer:** Single source to single destination message.
- **multicast:** broadcast along a dimension of the logical grid.
- **overlap_shift:** shifting data into overlap areas in one or more grid dimensions. This is particularly useful when the shift amount is known at compile time. This primitive uses that fact to avoid intra processor copying of data and directly stores data in the overlap areas [20].
- **temporary_shift:** This is similar to overlap shift except that the data is shifted into a temporary array. This is useful when the shift amount is not a compile time constant. This shift may require intra-processor copying of data.
- **concatenation:** This primitive concatenates a distributed array and the resultant array ends up in all the processors participating in this primitive.

We have implemented two sets of unstructured communication primitives: One, to support cases where the communicating processors can determine the send and receive lists based only on local information, and hence, only require preprocessing that involves local computations [21], and the other, where to determine the send and receive lists preprocessing itself requires communication among the processors [22]. The primitives are as follows.

- **precomp_read:** This primitive is used to bring all non-local data to the place it is needed before the computation is performed.
- **postcomp_write:** This primitive is used to store remote data by sending it to the processors that own the data after the computation is performed. Note that these two primitives requires only local computation in the preprocessing loop.

- **gather:** This is similar to *precomp_read* except that preprocessing loop itself may require communication.
- **scatter:** This is similar to *postcomp_write* except that preprocessing loop itself may require communication.

The compiler must recognize the presence of collective communication patterns in the computations in order to generate the appropriate communication calls. Specifically, this involves a number of tests on the relationship among subscripts of various arrays in a forall statement. These tests should also include information about array alignments and distributions. We use pattern matching techniques similar to those proposed by Chen [23]. Further, we extend the above tests to include unstructured communication. Table 1 shows the patterns of communication primitives used in our compiler. The detail of communication detection algorithm can be found in [7].

Steps	(lhs,rhs)	Comm. primitives
1	(i, s)	multicast
2	$(i, i+c)$	overlap_shift
3	$(i, i-c)$	overlap_shift
4	$(i, i+s)$	temporary_shift
5	$(i, i-s)$	temporary_shift
6	(d, s)	transfer
7	(i, i)	no_communication
8	$(i, f(i))$	precomp_read
9	$(f(i), i)$	postcomp_write
10	$(i, V(i))$	gather
11	$(V(i), i)$	scatter
12	$(i, unknown)$	gather
13	$(unknown, i)$	scatter

Table 1 Communication primitives based on the relationship between *lhs* and *rhs* array subscript reference pattern for block distribution. (c: compile time constant, s, d: scalar, f: invertible function, V: an indirection array).

5.1 Communication Generation

Having recognized the type of communication in each dimension of an array for structured communication or each array for unstructured communication in a forall statement, the compiler needs to perform the appropriate program transformations. We now illustrate these transformations with the aid of some examples.

Structured Communication

All the examples discussed below have the following mapping directives.

```
CHPF$ PROCESSORS(P,Q)
CHPF$ DISTRIBUTE TEMPL(BLOCK,BLOCK)
CHPF$ ALIGN A(I,J) WITH TEMPL(I,J)
CHPF$ ALIGN B(I,J) WITH TEMPL(I,J)
```

Example 1 (transfer) Consider the statement

```
FORALL(I=1:N) A(I,8)=B(I,3)
```

The first subscript of B is marked as no_communication because A and B are aligned in the first dimension and have identical indices. The second dimension is marked as *transfer*.

```
1.  call set_BOUND(lb,ub,st,1,N,1) ! compute local lb, ub, and st
2.  call set_DAD(B_DAD,.....)        ! put information for B into B_DAD
3.  call transfer(B, B_DAD, TMP,src=global_to_proc(8), dest=global_to_proc(3))
4.  DO I=lb,ub,st
5.     A(I,global_to_local(8)) = TMP(I)
6.  END DO
```

In the above code, the *set_BOUND* primitive (line 1) computes the local bounds for computation assignment based on the iteration distribution (Section 4). In line 2, the primitive set_DAD is used to fill the Distributed Array Descriptor (DAD) associated with array B so that it can be passed to the *transfer* communication primitive at run-time. The DAD has sufficient information for the communication primitives to compute all the necessary information including local bounds, distributions, global shape etc. Note that transfer performs one-to-one send-receive communication based on the logical grid. In this example, one column of grid processors communicate with another column of the grid processors as shown in Figure 3 (a).

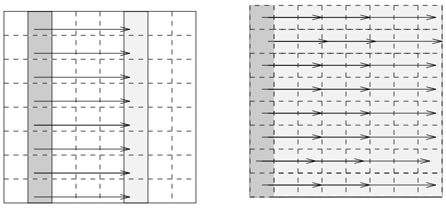
Example 2 (multicast) Consider the statement

```
FORALL(I=1:N,J=1:M) A(I,J)=B(I,3)
```

The second subscript of B marked as *multicast* and the first as no_communication.

```
1.    call set_BOUND(lb,ub,st,1,N,1) ! compute local lb, ub, and st
2.    call set_BOUND(lb1,ub1,st1,1,M,1) ! compute local lb, ub, and st
3.    call set_DAD(B_DAD,.....)        ! put information for B into B_DAD
4.    call multicast(B, B_DAD, TMP,source_proc=global_to_proc(3), dim=2)
5.    DO I=lb,ub,st
6.    DO J=lb1,ub1,st1
7.       A(I,J) = TMP(I)
8.    END DO
```

Line 4 shows a broadcast along dimension 2 of the logical processor grid by the processors owning elements $B(I,3)$ where $1 \leq I \leq N$ (Figure 3 (b).)



(a) transfer (b) multicast

Figure 3 Structured communication on logical grid processors.

Unstructured Communication

In distributed memory MIMD architectures, there is typically a non-trivial communication latency or startup cost. Hence, it is attractive to vectorize messages to reduce the number of startups. For unstructured communication, this optimization can be achieved by performing the entire preprocessing loop before communication so that the schedule routine can combine the messages to the maximum extent. The preprocessing loop is also called the "inspector" loop [25, 26].

Example 1 (precomp_read) Consider the statement

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

The array B is marked as *precomp_read* since the distributed dimension subscript is written as $f(i) = 2 * i + 1$ which is invertible as $g(i) = (i - 1)/2$.

```
1      count=1
2      call set_BOUND(lb,ub,st,1,N,1) ! compute local lb, ub, and st
3      DO I=1, N/P
4         receive_list(count)=global_to_proc(f(i))
5         send_list(count)= global_to_proc(g(i))
6         local_list(count) = global_to_local(g(i))
7         count=count+1
8      END DO
9      isch = schedule1(receive_list, send_list, local_list, count)
10     call precomp_read(isch, tmp,B)
11     count=1
12     DO I=1, N/P
13        if((I.ge.lb).and.(I.le.ub).and.(mod(I,st).eq.0))  ! mask
     &      A(I) = tmp(count)
14        count= count+1
15     END DO
```

The preprocessing loop is given in lines 1-9. Note that this preprocessing loop executes concurrently in each processor. The loop covers entire local array bounds since each processor has to calculate the *receive_list* as well as the *send_list* of processors. Each processor also fills the local indices of the array elements which are needed by that processor.

The *schedule1* routine does not need to communicate but only constructs the scheduling data structure *isch*. The schedule *isch* can also be used to carry out identical patterns of data exchanges on several different but identically distributed arrays or array sections. The same schedule can be reused repeatedly to carry out a particular pattern of data exchange on a single distributed array. In these cases, 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 the same schedule can be reused, it does not generate code for scheduling but it passes a pointer to the already existing schedule. Furthermore, the preprocessing computation can be moved up as much as possible by analyzing *definition-use chains* [27]. 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 the above example, *local_list* (line 6) is used to store the index of one-dimensional array. However, in general, *local_list* will store indices from a multi-dimensional Fortran array by using the usual column-major subscript calculations to map the indices to a one-dimensional index.

The *precomp_read* primitive performs the actual communication using the schedule. Once the communication is performed, the data is ordered in a one dimensional array, and the computation (lines 12-15) uses this one dimensional array. The *precomp_read* primitive brings an element into *temp* for each local array element since preprocessing loops coves entire local array. The *if* statement masks the assignment to preserve the semantic of original loop.

Example 2 (gather) Consider the statement

```
FORALL(I=1:N) A(I)=B(V(I))
```

The array B is marked as requiring *gather* communication since the subscript is only known at runtime. The receiving processors can know what non-local data they need from other processors, but a processor may not know what local data it needs to send to other processors. For simplicity, in this example, we assume that the indirection array V is replicated. If it is not replicated, the indirection array must also be communicated to compute the receive list on each processor.

```
count=1
call set_BOUND(lb,ub,st,1,N,1) ! compute local lb, ub, and st
DO I=lb,ub,st
   receive_list(count)=global_to_proc(V(i))
   local_list(count) = global_to_local(V(i))
   count=count+1
END DO
isch = schedule2(receive_list, local_list, count)
call gather(isch, tmp,B)
count=1
DO I=lb,ub,st
   A(I) = tmp(count)
   count= count+1
END DO
```

Once the scheduling is completed, every processors knows exactly which non-local data elements it needs to send to and receive from other processors. Recall that the task of *scheduler2* is to determine exactly which send and receive communications must be carried out by each processor. The scheduler first figures out how many messages each processor will have to send and receive during the data exchange. Each processor computes the number of elements (*receive_list*) and the local index of each element it needs from all other processors. In *schedule2* routine, processors communicate to combine these lists (a fan-in type of communication). At the end of this processing, each processor contains the send and receive list. After this point, each processor transmits a list of required array elements (*local_list*) to the appropriate processors. Each processor now has the information required to set up the send and receive messages that are needed to carry out the scheduled communication. This is done by the gather primitives. Note that *schedule1* does not need to communicate to form scheduling unlike schedule2.

Example 3 (scatter) Consider the statement

```
FORALL(I=1:N) A(U(I))=B(I)
```

The array A is marked as requiring *scatter* primitive since the subscript is only known at runtime. Note that owner computes rule is not applied here. The processor performing the computation knows the processor and the corresponding local-offset at which the resultant element must be written.

```
count=1
call set_BOUND(lb,ub,st,1,N,1) ! compute local lb, ub, and st
DO I=lb,ub,st
   send_list(count)=global_to_proc(U(i))
   local_list(count) = global_to_local(U(i))
   count=count+1
END DO
isch = schedule3(proc_to, local_to, count)
call scatter(isch, A, B)
```

Unlike the *gather* primitive, in this case each processor computes a *send_list* containing processor ids and *local_list* containing the local index where the communicated data must be stored. The *schedule3* is similar to *schedule2* of gather primitives except that schedule3 does not need to send local index in a separate communication step.

The gather and scatter operations are powerful enough to provide the ability to read and write distributed arrays with vectorized communication facility. These two primitives are available in PARTI (Parallel Automatic Runtime Toolkit at ICASE) [25] designed to efficiently support irregular patterns of distributed array accesses. The PARTI and other communication primitives and intrinsic functions form the run-time support system of our Fortran 90D compiler.

6 RUN-TIME SUPPORT SYSTEM

The Fortran 90D/HPF compiler relies on a very powerful run-time support system. The run-time support system consists of functions which can be called from the node programs of a distributed memory machine. Intrinsic functions support many of the basic data parallel operations in Fortran 90. They do not only provide a concise means of expressing operations on arrays, but 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. The intrinsic functions that may induce communication can be divided into five categories as shown in Table 2.

1. Structured communication	**2. Reduction**	**3. Multicasting**	**4. Unstructured communication**	**5. Special routines**
CSHIFT EOSHIFT	DOTPRODUCT ALL, ANY COUNT MAXVAL, MINVAL PRODUCT SUM MAXLOC, MINLOC	SPREAD	PACK UNPACK RESHAPE TRANSPOSE	MATMUL

Table 2 Some HPF Intrinsic Functions.

The first category requires data to be transferred using with less overhead structured shift communications operations. The second category of intrinsic functions require computations based on local data followed by use of a reduction tree on the processors involved in the execution of the intrinsic function. The third category uses multiple broadcast trees to spread data. The fourth category is implemented using unstructured communication patterns. The fifth category is implemented using existing research on parallel matrix algorithms [17]. Some of the intrinsic functions can be further optimized for the underlying hardware architecture.

Nproc	ALL (1K x 1K)	ANY (1K x 1K)	MAXVAL (1K x 1K)	PRODUCT (256K)	TRANSPOSE (512 x 512)
1	580.6	606.2	658.8	90.1	299.0
2	291.0	303.7	330.4	50.0	575.0
4	146.2	152.6	166.1	25.1	395.0
8	73.84	77.1	84.1	13.1	213.0
16	37.9	39.4	43.4	7.2	121.0
32	19.9	20.7	23.2	4.2	69.0

Table 3 Performance of some HPF Intrinsic Functions (time is milliseconds).

Table 3 presents a sample of performance numbers for a subset of the intrinsic functions on iPSC/860. A detailed performance study is presented in [11]. The times in the table include both the computation and communication times for each function. For most of the functions we were able to obtain almost linear speedups. In the case of TRANSPOSE function, going from one processor to two or four actually results in increase in the time due to the communication requirements. However, for larger size multiprocessors the times decrease as expected.

Arrays may be redistributed across subroutine boundaries. A dummy argument which is distributed differently than its actual argument in the calling routine is automatically redistributed upon entry to the subroutine by the compiler, and is automatically redistributed back to its original distribution at subroutine exit. These operations are performed by the redistribution primitives which transform from *block* to *cyclic* or vice versa.

When a distributed array is passed as an argument to some of the run-time support primitives, it is also necessary to provide information such as its size, distribution among the nodes of the distributed memory machine etc. All this information is stored into a structure which is called *distributed array descriptor* (DAD) [11].

In summary, parallel intrinsic functions, communication routines, dynamic data redistribution primitives and others are part of the run-time support system.

7 OPTIMIZATIONS

Several types of *communication* and *computation* optimization can be performed to generate a more efficient code. In terms of *computation* optimization, it is expected that 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 attached vector units), our compiler propagates the information to the node compiler using node directives. Since in the original data parallel constructs such as *forall* statement, there is no data dependency between different loop iteration, vectorization can be performed easily by the node compiler.

Our compiler performs several optimizations to reduce the total cost of communication. Some of *communication* optimizations [23, 28, 29] are as follows.

- *Vectorized communication.* 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 equivalent to the sending cost of hundreds of bytes. Vectorization combines messages for the same source and destination into a single message to reduce this overhead. Since in Fortran 90D we are only parallelizing array assignments and forall loops, there is no data dependency between different loop iterations. Thus, all the required communication can be performed before or after the execution of loop on each of the processors involved.

- *Eliminate unnecessary communications.* In many cases, communication required for two different operands can be replaced by their union. For example, the following code may require two *overlapping_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)
  ```

- *Reuse of scheduling information. Unstructured* communication primitives are required by computations which require the use of a preprocessor. As discussed in Section 6.3.2, the schedules can be reused with appropriate analysis.

- *Code movement.* The compiler can utilize the information that the runtime support routines do not have procedural side effects. For example, the preprocessing loop or communication routines can be moved up as much as possible by analyzing *definition-use chains* [27]. This may lead to moving of the scheduling code out of one or more nested loops which may reduce the amount of communication required significantly.

We are incrementally incorporating many more optimizations in the compiler.

8 EXPERIMENTAL RESULTS

A prototype compiler is complete (it was demonstrated at Supercomputing'92). In this section, we describe our experience in using the compiler.

One of the principal requirements of the users of distributed memory MIMD systems is some "guarantee" of the portability for their code. Express parallel programming environment [30] guarantees this the portability on various platforms including, Intel iPSC/860, nCUBE/2, networks of workstations etc. We should emphasize that we have implemented a collective communication library which is currently built on the top of Express message passing primitives. Hence, in order to change to any other message passing system such as PVM [31] (which also runs on several platforms), we only need to replace the calls to the communication primitives in our communication library (not the compiler). However, it should be noted that a penalty must be paid to achieve portability because portable routines are normally built on top of the system routines. Therefore, the performance also depends on how efficient are the communication primitives on the top of which the communication library is built.

As a test application we use Gaussian Elimination, which is a part of the FortranD/HPF benchmark test suite [32]. Figure 4 shows the execution times obtained to run the same compiler generated code on a 16-node Intel/860 and nCUBE/2 for various problem sizes. Due to space limitations, we do not present performance of many other programs, and some of them can be found in [?].

Table 4 shows a comparison between the performance of the hand-written Fortran 77+MP code with that of the compiler generated code. We can observe that the performance of the compiler generated code is within 10% of the hand-written code. This is due to the fact that the compiler generated code produces an extra communication call that can be eliminated using optimizations. However as Figure 5 shows, the gap between the performance of the two codes increases as the number of processors increases. This is because the extra communication step is a broadcast which is almost $O(log(P))$ for a P processor hypercube system.

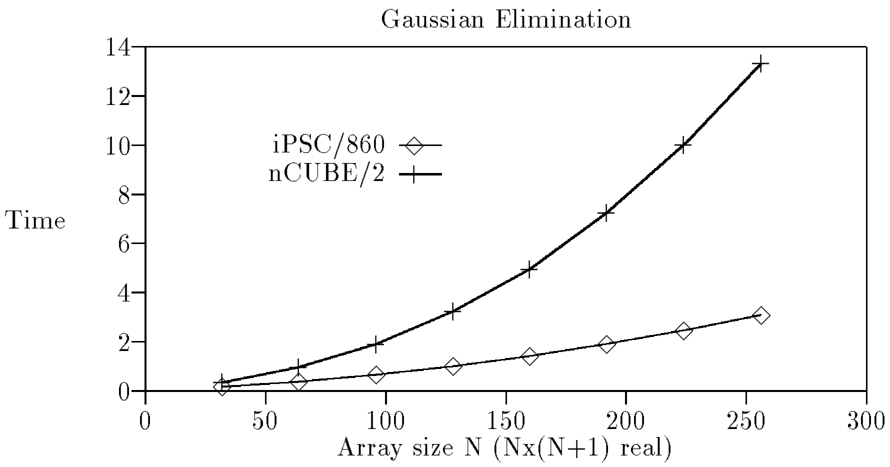


Figure 4 Execution time of HPF compiler generated code for Gaussian Elimination on a 16-node Intel iPSC/860 and nCUBE/2 (time in seconds).

	Number of PEs				
	1	2	4	8	16
Hand Written	623.16	446.60	235.37	134.89	79.48
HPF	618.79	451.93	261.87	147.25	87.44

Table 4 Comparison of the execution times of the hand-written code and HPF compiler generated code for Gaussian Elimination. Matrix size is 1023x1024 and it is column distributed.(Intel iPSC/860, time in seconds).

9 SUMMARY OF RELATED WORK

The compilation technique of Fortran 77 for distributed memory systems has been addressed by Callahan and Kennedy [13]. Currently, a Fortran 77D compiler is being developed at Rice [28, 34]. Superb [12] compiles a Fortran 77

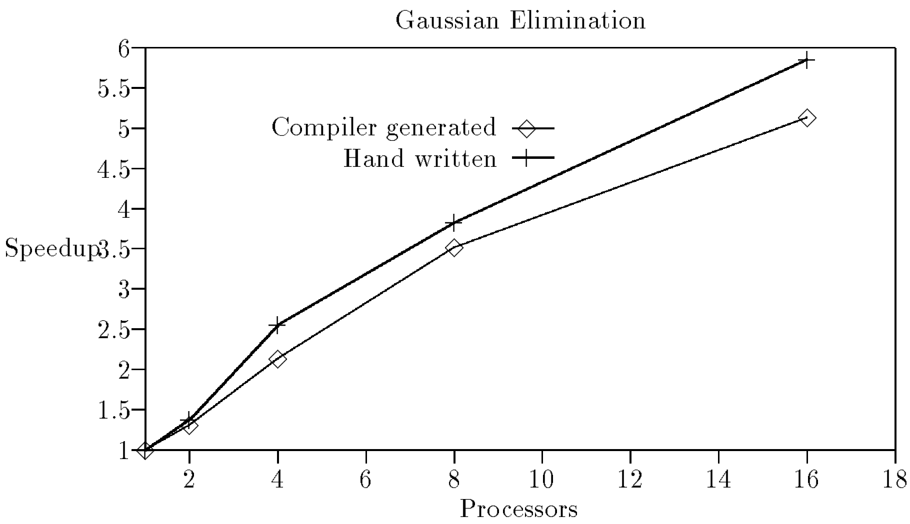


Figure 5 Speed-up against the sequential code (corresponds to Table 4 of the hand-written code and Fortran 90D compiler generated code for Gaussian Elimination).

program into a semantically equivalent parallel SUPRENUM multiprocessor. Koelbel and Mehrotra [26, 21] puts a great deal of effort on run-time analysis for optimizing message passing in implementation of Kali. Quinn *et al.* [35, 36] use a data parallel approach for compiling C* for hypercube machines. The ADAPT system [37] compiles Fortran 90 for execution on MIMD distributed memory architectures. The ADAPTOR [38] is a tool that transform data parallel programs written in Fortran with array extension and layout directives to explicit message passing. Chen [23, 39] describes general compiler optimization techniques that reduce communication overhead for Fortran-90 implementation on massivelly parallel machines. Many techniques especially unstructured communication of our compiler are adapted from Saltz *et al.* [40, 29, 22]. Gupta *et al.* [24, 41] use collective communication on automatic data partitioning on distributed memory machines. Due to space limitations, we do not elaborate on various other related projects.

10 SUMMARY AND CONCLUSIONS

HPF are languages that incorporate parallel constructs and allow users to specify data distributions. In this paper, we presented a design for HPF compiler for distributed memory machines. Specifically, techniques for processing distribution directives, computation partitioning, communication detection and generation were presented. We show that our design is portable, yet efficient.

We believe that the methodology presented in this paper to compile HPF can be used by the other designers and implementors for HPF language.

APPENDIX A

A.1 GAUSSIAN E. WITH HPF

```
      integer, dimension(N) :: indx
       integer, dimension(1) :: iTmp
       real, dimension(N,NN) :: a
       real, dimension(N) :: fac
       real, dimension(NN) :: row
       real :: maxNum
CHPF$  PROCESSORS  PROC(P)
CHPF$  TEMPLATE   T(NN)
CHPF$  DISTRIBUTE T(BLOCK)
CHPF$  ALIGN row(J) WITH T(J)
CHPF$  ALIGN a(*,J) WITH T(J)
      indx = -1
      do k = 0, N-1
        iTmp = MAXLOC(ABS(a(:,k)), MASK = indx .EQ. -1)
        indxRow = iTmp(1)
```

```
      maxNum = a(indxRow,k)
      indx(indxRow) = k
      fac = a(:,k) / maxNum
      row = a(indxRow,:)
      forall (i = 0:N-1, j = k:NN-1, indx(i) .EQ. -1)
&       a(i,j) = a(i,j) - fac(i) * row(j)
    end do
```

Acknowledgements

We are grateful to Parasoft for providing the Fortran 90 parser and Express without which the prototype compiler could have been delayed. We would like to thank the other members of our compiler research group I. Ahmad, R. Bordawekar, R. Ponnusamy, R. Thakur, and J. C. Wang for their contribution in the project including the development of the run-time library functions, testing, and help with programming. We would also like to thank K. Kennedy, C. Koelbel, C. Tseng and S. Hiranandani of Rice University and J. Saltz and his group of Maryland University for many inspiring discussions and inputs that have greatly influenced this work.

This work was supported in part by NSF under CCR-9110812 (Center for Research on Parallel Computation) and DARPA under contract # DABT63-91-C-0028. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. Alok Choudhary is also supported by an NSF Young Investigator Award.

REFERENCES

[1] American National Standards Institue. Fortran 90: X3j3 internal document s8.118. *Summitted as Text for ISO/IEC 1539:1991*, May 1991.

[2] G. C. Fox, S. Hiranadani, K. Kenndy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical report, Rice and Syracuse University, 1992.

[3] High Performance Fortran Forum. High performance fortran language specification version 1.0. *Draft, Also available as technical report CRPC-*

TR92225 from the Center for Research on Parallel Computation, Rice University., Jan. 1993.

[4] The Thinking Machine Corporation. *CM Fortran User's Guide version 0.7-f*, July 1990.

[5] Maspar Computer Corporation. *MasPar Fortran User Guide version 1.1*, Aug. 1991.

[6] G. Fox. The architecture of problems and portable parallel software systems. Technical Report SCCS-78b, Syracuse University, 1991.

[7] Z. Bozkus et al. Compiling the FORALL statement on MIMD parallel computers. Technical Report SCCS-389, Northeast Parallel Architectures Center, July 1992.

[8] D. Padua B. Leasure D. Kuck, R. Kuhn and M. Wolf. Dependence graph and compiler optimizations. *Proc. of 8th ACM Symp. Principles on Programming Lang.*, September 1981.

[9] Z. Bozkus et al. Compiling Distribution Directives in a Fortran 90D Compiler. Technical Report SCCS-388, Northeast Parallel Architectures Center, July 1992.

[10] S. Chatterjee, J.R. Gilbert, R. Schreiber, and S.H Tseng. Automatic Array Alignment in Data-Parallel Programs. *Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, January 1993.

[11] I. Ahmad, R. Bordawekar, Z. Bozkus, A. Choudhary, G. Fox, K. Parasuram, R. Ponnusamy, S. Ranka, and R. Thakur. Fortran 90D Intrinsic Functions on Distributed Memory Machines: Implementation and Scalability. Technical Report SCCS-256, Northeast Parallel Architectures Center, March 1992.

[12] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi Automatic SIMD/MIMD Parallelization. *Parallel Computing*, January 1988.

[13] D. Callahan and K. Kennedy. Compiling programs for Distributed Memory Multiprocessors. *The Journal of Supercomputing*, pages 171–207, 1988.

[14] M. Wu and G. Fox et al. Compiling Fortran 90 programs for distributed memory MIMD paralelel computers. Technical Report SCCS-88, Northeast Parallel Architectures Center, May 1991.

[15] R. Allen. Dependency analysis for Subscripted Variables and its Application to Program Transformation. Technical Report PhD thesis, Rice University, 1983.

[16] J. Ng, V. Sarkar, and J.F. Shaw. Optimized execution of Fortran 90 array constructs on supercomputer architectures. Supercomputing'91, 1991.

[17] G. C. Fox, M.A. Johnson, G.A. Lyzenga, S. W. Otto, J.K. Salmon, and D. W. Walker. In *Solving Problems on Concurent Processors*, volume 1-2. Prentice Hall, May 1988.

[18] K. Knobe, J. D. Lukas, and G. L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, pages 102–118, Feb 1990.

[19] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, pages 213–221, Oct 1991.

[20] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, September 1990.

[21] C. Koelbel and P. Mehrotra. Supporting Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

[22] H. Berryman J. Saltz, J. Wu and S. Hiranandani. Distributed Memory Compiler Design for Sparse Problems. *Interim Report ICASE, NASA Langley Research Center*, 1991.

[23] J. Li and M. Chen. Compiling Communication -Efficient Programs for Massively Parallel Machines. *IEEE Transactions on Parallel and Distributed Systems*, pages 361–376, July 1991.

[24] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE: Transaction on Parallel and Distributed Systems*, pages 179–193, March 1992.

[25] R. Das, J. Saltz, and H. Berryman. A Manual For PARTI Runtime Primitives. *NASA,ICASE Interim Report 17*, May 1991.

[26] C. Koelbel, P. Mehrotra, and J. V. Rosendale. Supporting Shared Data Structures on Distributed Memory Architectures. *PPoPP*, March 1990.

[27] A.V. Aho, R. Sethi, and J.D Ullman. *Compilers Principles, Techniques and Tools*. March 1988.

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

[29] R. Mirchandaney J. Saltz, K. Crowley and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, December 1991.

[30] ParaSoft Corp. *Express Fortran refernce guide Version 3.0*, 1990.

[31] A. Beguelin, J. Dongarra, A. Geist, R. Mancheck, and V. Sunderam. A Users Guide to PVM Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[32] A. G. Mohamed, G. Fox, G. V. Laszewski, M. Parashar, T. Haupt, K. Mills, Y. Lu, N. Lin, and N. Yeh. Application Benchmark Set for Fortran-D and High Performance Fortran. Technical Report SCCS-327, Northeast Parallel Architectures Center, May 1992.

[33] S. L. Johnsson. Performance Modeling of Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, pages 300–312, August 1991.

[34] S. Hiranandani, K. Kennedy, and C.W. Tseng. Compiler support for machine-indepentet Parallel Programming in Fortran D. *Compiler and Runtime Software for Scalable Multiprocessors*, 1991.

[35] Michael Quinn, Philip Hatcher, and Karen Jourdenais. Compiling C* Programs for a Hypercube Multicomputer. *Parallel Computing Laboratory, University of New Hampshire*, PCL-87-12, December 1987.

[36] Philip Hatcher, Anthony Lapadula, Robert Jones, Michael Quinn, and Ray Anderson. A Production-Quality C* Compiler for Hypercube Multicomputers. *Third ACM SIGPLAN symposium on PPOPP*, 26:73–82, July 1991.

[37] J.H Merlin. Techniques for the Automatic Parallelisation of 'Distributed Fortran 90'. Technical Report SNARC 92-02, Southampton Novel Architecture Research Centre, 1992.

[38] T. Brandes. ADAPTOR Language Reference Manual. Technical Report ADAPTOR-3, German National Research Center for Computer Science, 1992.

[39] M. Chen and J.J Wu. Optimizing FORTRAN-90 Programs for Data Motion on Massivelly Parallel Systems. Technical Report YALEU/DCS/TR-882, Yale University, Dep. of Comp. Sci., 1992.

[40] H. Berryman J. Saltz and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, December 1991.

[41] M. Gupta. Automatic Data Partitioning on Distributed Memory Multicomputers. Technical Report PhD thesis, University of illinois at Urbana-Champaign, 1992.