NPAC Technical Report, April 1994

# Efficient Compilation of Out-of-core Data Parallel Programs[*]

**Rajesh Bordawekar** **Rajeev Thakur** **Alok Choudhary**
Dept. of Electrical and Computer Eng. and
Northeast Parallel Architectures Center
Syracuse University
rajesh, thakur, choudhar @npac.syr.edu

**Abstract**

Large scale scientific applications, such as the *Grand Challenge* applications, deal with very large quantities of data. The amount of main memory in distributed memory machines is usually not large enough to solve problems of realistic size. This limitation results in the need for system and application software support to provide efficient parallel I/O for out-of-core programs. This paper describes techniques for translating out-of-core programs written in a data parallel language like HPF to message passing node programs with explicit parallel I/O. We describe the basic compilation model and various steps involved in the compilation. The compilation process is explained with the help of an out-of-core matrix multiplication program. We first discuss how an out-of-core program can be translated by extending the method used for translating in-core programs. We then describe how the compiler can optimize the code by estimating the I/O costs associated with different array access patterns and selecting the method with the least I/O cost. This optimization can reduce the amount of I/O by as much as an order of magnitude. Performance results on the Intel Touchstone Delta are presented and analyzed.

[*]This work was sponsored in part by NSF Young Investigator Award CCR-9357840 with a matching grant from Intel SSD, and also by ARPA under contract no. DABT63-91-C-0028. The content of the information does not necessarily reflect the position or policy of the Government and no official endorsement should be inferred. This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by CRPC.

## 1 Introduction

The use of massively parallel machines to solve large scale computational problems in physics, chemistry, biology, engineering, medicine and other sciences has increased considerably in recent times. This is primarily due to the tremendous improvements in the computational speeds of parallel computers in the last few years. Many of these applications, also referred to as *Grand Challenge Applications* [CR93], have computational requirements which stretch the capabilities of even the fastest supercomputer available today. For example, in Computational Fluid Dynamics, a real simulation of the air flow past an aircraft in flight, without any simplifying assumptions, would take several months to solve.

In addition to requiring a great deal of computational power, these applications usually deal with large quantities of data. At present, a typical Grand Challenge Application could require 1Gbyte to 4Tbytes of data per run [dRC94]. These figures are expected to increase by orders of magnitude as teraflop machines make their appearance. Main memories are not large enough to hold this much amount of data; so data needs to be stored on disks and fetched during the execution of the program. Unfortunately, the performance of the I/O subsystems of massively parallel computers has not kept pace with their processing and communications capabilities [CFPB93]. Hence, the performance bottleneck is the time taken to perform disk I/O. The need for high performance I/O is so significant that almost all the present generation parallel computers such as the Paragon, iPSC/860, Touchstone Delta, CM-5, SP-1, nCUBE2 etc. provide some kind of hardware and software support for parallel I/O [CFPB93, Pie89, BC93, DdR92]. An overview of the various issues involved in high performance I/O is given in [dRC94].

Data parallel languages like HPF [For93] and pC++ [BBG$^+$93] have recently been developed to provide support for portable high performance programming on parallel machines. In order that these languages can be used for large scale scientific computations, it is essential that the compiler can automatically translate out-of-core programs efficiently. In this paper we describe the design of a compiler which can translate an out-of-core HPF program to a message passing node program with explicit parallel I/O. We also discuss how the compiler can estimate the cost associated with different I/O access patterns in the translated program, in terms of the number of I/O requests and the amount of data to be fetched from disk. This estimate is used to select the method which requires the least amount of I/O. We find that this optimization can reduce the amount of I/O by as much as an order of magnitude. The compilation process is explained with the help of an out-of-core matrix multiplication program which uses a distributed GAXPY algorithm. This program is used only as an example to illustrate the various issues involved in compiling out-of-programs and optimizing the I/O requirements. The techniques described in this paper are applicable to any other program in general. Performance results on the Intel Touchstone Delta are presented and analyzed.

The rest of the paper is organized as follows. Section 2 describes the basic model used for out-of-core compilation. The compilation methodology is discussed in Section 3. Section 4 describes the I/O cost estimation and optimizations performed by the compiler, followed by conclusions in Section 5. In this paper, the term *in-core compiler* refers to a compiler for in-core programs and the term *out-of-core compiler* refers to a compiler for out-of-core programs.

## 2 Model for Out-of-Core Compilation

### 2.1 Programming Model

The most widely used programming model for large-scale scientific and engineering applications on distributed memory machines is the Single Program Multiple Data (SPMD) model. In this model, parallelism is achieved by partitioning data among processors which effectively represents parallelism in a class of applications called *loosely synchronous* applications [Fox91]. To achieve load-balance, express locality of access, reduce communication and other optimizations, several distribution and data alignment strategies are often used (eg., block, cyclic, along rows, columns, etc.). Many parallel programming languages or language extensions have been developed which support such distributions. These languages provide directives that enable the expression of mappings from the problem domain to the processing domain and allow the user to align and distribute arrays in the most appropriate fashion for the underlying computation. The compiler uses the information provided by these directives to compile global name space programs for distributed memory computers. Examples of parallel languages which support data distribution include Vienna Fortran [ZBC$^+$92], Fortran D [FHK$^+$90] and High Performance Fortran (HPF) [For93]. In this paper, we describe the compilation of out-of-core HPF programs, but the discussion is applicable to any other data parallel language in general.

The DISTRIBUTE directive in HPF specifies which elements of the array are mapped to each processor. This results in each processor having a *local array* associated with it. In an in-core program, the local array resides in the local memory of the processor. Our group at Syracuse University has developed a compiler for in-core HPF programs [BCF$^+$93]. For large data sets, however, local arrays cannot entirely fit in main memory. In such cases, parts of the local array have to be stored on disk. We refer to such a local array as an **Out-of-core Local Array (OCLA)**. Parts of the OCLA need to be swapped between main memory and disk during the course of the computation. If the operating system supports node virtual memory on each processor, the OCLA can be swapped in and out of the disk automatically by the operating system.

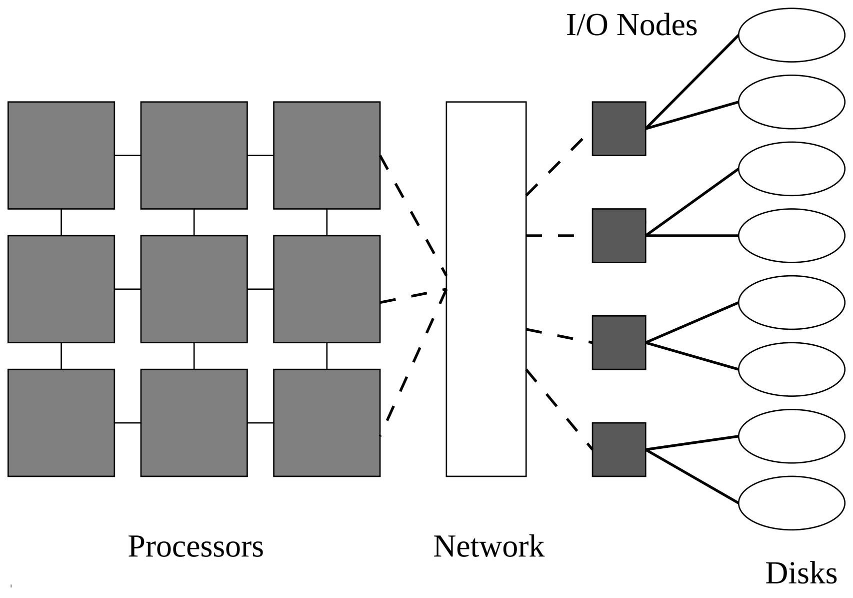

Figure 1: Architectural Model

Performance studies of the virtual memory provided by the OSF/1 operating system on the Intel Paragon have shown that the paging-in and paging-out of data from the nodes drastically degrade the user code performance [SS93]. Also, most of the other massively parallel systems at present, such as the CM-5, iPSC/860, Touchstone Delta, nCUBE-2 etc. do not support virtual memory on the nodes. Thus, a compiler must translate into a code which explicitly performs I/O. Even if the node virtual memory is supported, paging mechanisms are not known to handle different access patterns efficiently. This is true even in the case of sequential computers.

## 2.2 Architectural Model

Figure 1 describes the architectural model used by the compiler. It assumes any general distributed memory computer in which the processors are connected together in some fashion. The system is provided with a set of disks. Each processor may either have its own local disk or all processors may share the set of disks. The system is provided with dedicated I/O processors which control the flow of data between the compute processors and the disks. The I/O subsystem may have a separate interconnection network or it can share the same network which connects the processors together.

## 2.3 Data Storage Model

The Data Storage Model shown in Figure 2 specifies how the out-of-core array is placed on disks and how it is accessed by the processors. The out-of-core local array of each processor is stored

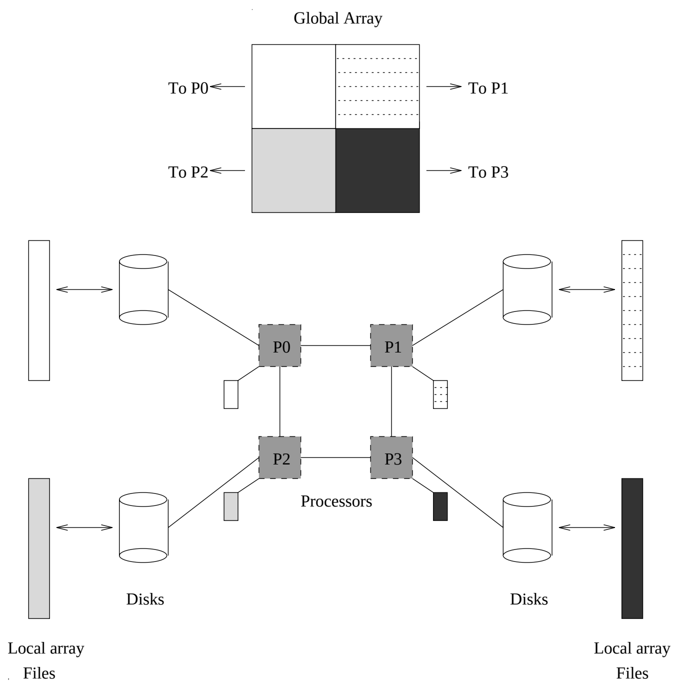

Figure 2: Local Placement Model for out-of-core compilation

in separate file called the **Local Array File (LAF)** of that processor. The LAF can be assumed to be *owned* by that processor. The node program explicitly reads from and writes into the LAF when required. If the I/O architecture of the system is such that each processor has its own disk, such as in the IBM SP-1, the LAF of each processor will be stored on the disk attached to that processor. If there is a common set of disks for all processors, such as on the Intel Paragon, the LAF will be distributed across one or more of these disks. In other words, we assume that each processor has its own *logical disk* with the LAF stored on that disk. The mapping of the logical disk to the physical disks is system dependent.

A simple way to view this model is to think of each processor as having another level of memory (logical disk) which is much slower than the main memory. Both the main memory and this additional memory cannot be directly accessed by any other processor. Hence, a processor cannot directly access some other processor's LAF. If a processor needs data from the LAF of another processor, the required data will be first read by the owner processor and then communicated to the requesting processor.

In order to store data on the disks based on the distribution pattern specified in the program, redistribution of data may be needed in the beginning when data is first stored on disk. This is because the way data arrives (eg. from archival storage, satellite or over the network) may not conform to the distribution specified in the program. Redistribution requires reading data from disks, communicating data between processors and writing the data to the local array files. This involves some additional overhead which can be amortized if the array is used several times (eg. for many iterations).

## 3 Compilation Methodology

This section describes the methodology used for compiling out-of-core HPF programs. We use an out-of-core matrix multiplication program example to illustrate the compilation and optimization process. We have chosen this example because it clearly brings out many of the important issues involved in compiling out-of-core programs. This example is used only for explanatory purposes and the methodology described in this paper is applicable to any other program in general.

We first describe the global name space matrix multiplication program and then explain how it is translated assuming that all the matrices are in-core. This is helpful in understanding how the program needs to be compiled when the arrays are out-of-core.

### 3.1 GAXPY Algorithm for Matrix Multiplication

Let $A$, $B$ and $C$ be $n \times n$ matrices such that $C = A \times B$. $A$, $B$ and $C$ can be represented in terms of their individual columns as

$$A = [a_1, \cdots, a_n],\ a_j \in \mathcal{R}^n$$
$$B = [b_1, \cdots, b_n],\ b_j \in \mathcal{R}^n$$
$$C = [c_1, \cdots, c_n],\ c_j \in \mathcal{R}^n$$

Then the GAXPY algorithm for computing $C = A \times B$ is

$$c_j = \sum_{k=1}^{n} b_{kj} a_k,\ \ j = 1 : n \tag{1}$$

We can see that in order to compute the $j^{th}$ column of $C$, we need the $j^{th}$ column of $B$ and all columns of A. Figure 3 shows the HPF program for GAXPY matrix multiplication. Arrays A and C are distributed in column-block fashion whereas array B is distributed in row-block fashion over 4 processors. A temporary array is needed to store the products of element $b_{kj}$ and column $a_k$, which can be computed for all $k$ in parallel. The $j^{th}$ column of the result is computed using the intrinsic function SUM.

```
1            parameter (n=64, nprocs=4)
2            real a(n,n), b(n,n), c(n,n), temp(n,n)
3       !hpf$      processors Pr(nprocs)
4       !hpf$      template d(n)
5       !hpf$      distribute d(block) on Pr
6       !hpf$      align (*,:) with d :: a, c, temp
7       !hpf$      align (:,*) with d :: b
8            do j=1, n
9                FORALL (k=1:n)
10                    temp(1:n,k) = b(k,j)×a(1:n,k)
11                end FORALL
12                c(1:n,j) = SUM(temp,2)      ! Sum Intrinsic
13             end do
14             end
```

Figure 3: GAXPY Matrix Multiplication in HPF

### 3.2 In-core Compilation

Our research group at Syracuse University has developed a compiler to translate in-core HPF programs to message passing node programs for distributed memory machines [BCF$^+$93]. Figure 4 shows the steps followed by the compiler in translating a FORALL or array assignment statement in an HPF program.[1] A FORALL statement is essentially a parallel loop with copy-in-copy-out semantics [For93]. According to the HPF specifications, the following steps describe a correct sequential implementation of a FORALL statement; 1) copy rhs, 2) synchronize, 3) evaluate expression, 4) synchronize, 5) assign to lhs. Note that synchronization and copying are only part of the specification and can be avoided in most cases with appropriate compiler analysis [BCF$^+$93].

The compiler uses distribution directives (Figure 3, lines 3–7) in the source program to find the distribution pattern of the arrays. Using the data distribution information, the arrays are partitioned into local arrays. After data distribution, the compiler analyzes the array operations (Figure 3, lines 8–13). The compiler checks that the outer loop (lines 8–13) is a sequential DO loop whereas the inner loop (lines 9–11) is a parallel FORALL construct. The inner FORALL

[1]Any array assignment statement can be converted into a corresponding FORALL statement, so we will use them interchangeably [FHK$^+$90, For93].

```
1    Analyze the distribution pattern of each array used in the array expression.
2    Depending on the distribution, detect the type of communication required.
3    Perform data partitioning and calculate local lower and upper bounds
       for each participating processor.
4    Use temporary arrays if the same array is used in both LHS and RHS
       of the array expression.
5    Generate the corresponding loosely synchronous SPMD node program.
6    Add calls to runtime libraries to perform collective communication.
```

Figure 4: Steps in translating array assignment statements (in-core)

loop (indexed by variable $k$) is sequentialized into local DO loops. After the local computation is done, the temporary results are added to give the $j^{th}$ column of the resultant C. This operation is performed using a global sum reduction routine. Using the knowledge that the index $j$ is in global name space and that C is distributed in column-block fashion, the compiler computes the owner of the resultant column which stores the result in the appropriate location in the local C array. Figure 5 shows the resultant node plus message passing program.

### 3.2.1 Comparison with a Hand-coded Program

Equation 1 can be rewritten as a sum of $p$ partial sums as follows

$$c_j = \underbrace{\sum_{k=1}^{\lfloor\frac{n}{p}\rfloor} b_{kj}a_k + \sum_{k=\lfloor\frac{n}{p}\rfloor+1}^{2\lfloor\frac{n}{p}\rfloor} b_{kj}a_k + \cdots + \sum_{k=((p-1)\times\lfloor\frac{n}{p}\rfloor)+1}^{n} b_{kj}a_k}_{p\ Sums},\ j = 1:n,\ a_k \in \mathcal{R}^n \tag{2}$$

Each of these partial sums can be obtained on individual processors. Consider the partial sum $\sum_k b_{kj}a_k$. Each partial sum returns an intermediate vector in $\mathcal{R}^n$. Each vector is a linear combination of $\lfloor\frac{n}{p}\rfloor$ columns of $A$ and $\lfloor\frac{n}{p}\rfloor$ elements of a column $j$ of $B$. These intermediate vectors are then added to give the $j^{th}$ column of matrix $C$. This process is repeated $n$ times. It can be observed that to obtain the intermediate vectors, the best way to distribute $A$ is in column-block form and $B$ in row-block form. For this distribution, the number of rows of $B$ in each processor is equal to the number of columns of $A$ in that processor. Moreover, since in each step $j$ of the summing process column $c_j$ of array $C$ is computed, the natural distribution for $C$ is the same as that for $A$, namely column-block.

```
        parameter (n=64, nproc=4, local_n=16)
C       Partition the arrays using the distribution information.
        real a(n,local_n), b(local_n,n), c(n,local_n)
        do j=1, n
            Initialize the temporary array.
            do i= 1, local_n
                do k=1, n
                    temp(k,i) = a(k,i)×b(i,j)
                end do
            end do
C       Perform Global Sum of the temporary arrays along dimension 2.
            result = global_sum(temp, 2)
C       Find the owner of the $j^{th}$ column and store the column.
            owner = global_to_processor(j)
            local_index = global_to_local(j)
            if (mynode = owner) then
                store the result as $(local\_index)^{th}$ column of C
            end if
        end do
        end
```

Figure 5: Translated code for in-core matrix multiplication

```
1 do j=1, columns_b ($n$)
2       do k=1, rows_b ($\frac{n}{p}$)
3           do i=1, rows_a ($n$)
4               temp(i) = b(k,j)$\times$a(i,k) + temp(i)      ! Find Partial Sum
5           end do
6       end do
7       temp_sum = global sum of temp.
8       if (mynode is owner of column $j$) then
9           store temp_sum as column c($j'$), where $j'$ = global_to_local($j$)
10       end if
11 end do
```

Figure 6: Hand-coded Distributed GAXPY Program

Figure 6 shows a hand-coded distributed memory GAXPY matrix multiplication program. The outer-most loop ($j$) varies from 1 to columns_b ($n$). In each iteration ($j$), the column $j$ of array $B$ is used for computation. Two inner loops multiply the $k^{th}$ column of $A$ by the $k^{th}$ element of column $j$ of $B$ (lines 2-6). The intermediate vector *temp* is then added by all processors to give the global sum (temp_sum in line 7). Using the global index $j$, the owner of column $c(j)$ is calculated. This processor stores temp_sum as the $j^{th}$ column of array $C$ in the corresponding local array position. Note that the two inner loops operate in the local index space whereas the outer loop operates in the global index space. Figure 7 illustrates the computation in the $j^{th}$ iteration of the algorithm. The elements of array $B$ and the corresponding columns of array $A$ are shown using the same shade.

A comparison of the programs in Figures 5 and 6 shows that the code generated by the in-core compiler is similar to the hand-coded version. That is, in-core compilation produces a good code in comparison with a hand-coded program.

## 3.3 Out-of-core Compilation

The out-of-core HPF compiler follows an approach similar to the in-core HPF compiler. In order to translate out-of-core programs, in addition to following the steps in Figure 4, the compiler also has to schedule explicit I/O accesses to fetch/store appropriate data from/to disks. The compiler has to take into account the data distribution on disks, the number of disks used for storing data and the prefetching/caching strategies used. As stated earlier, the local array of each processor is stored in a local array file (LAF). The portion of the local array currently required for computation

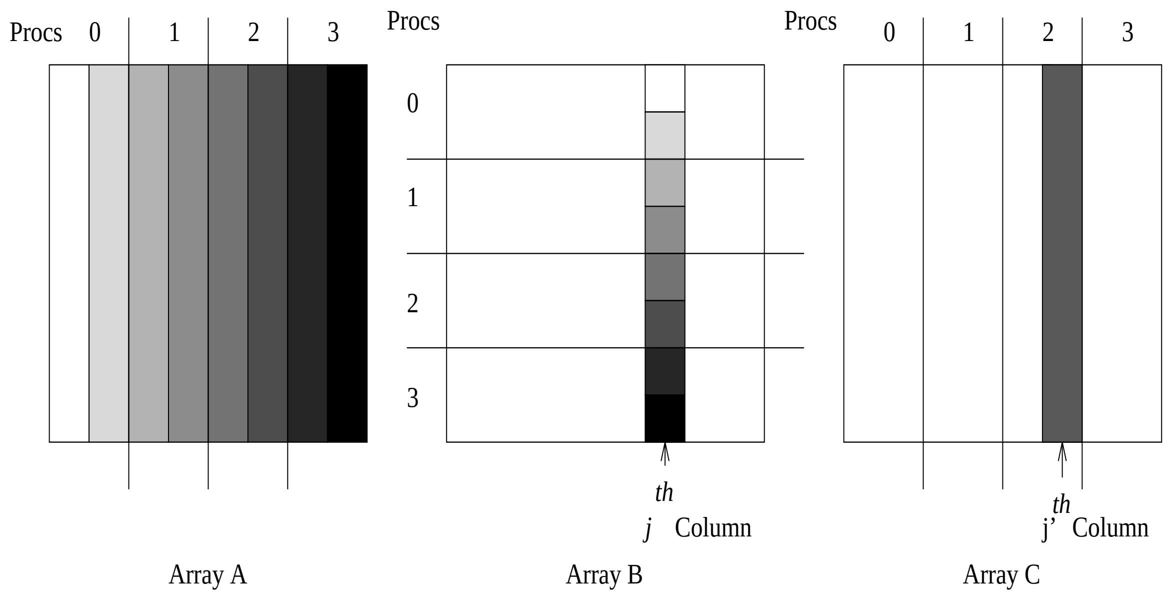

Figure 7: Distributed GAXPY for in-core matrices

is fetched from disk into the in-core local array (ICLA). The size of the ICLA is specified at compile time and usually depends on the amount of memory available. The larger the ICLA the better, as it reduces the number of disk accesses. Each processor performs computation on the data in its ICLA.

Some of the issues in out-of-core compilation are similar to compiler optimizations carried out to gain advantage of processor caches or pipelines. This optimization, commonly known as *stripmining* [AK87, Wol89, ZC91], sections the loop iterations so that data of a fixed size (equal to cache size or pipeline stages) could be operated on in each iteration. In the case of out-of-core programs, the computation involving the entire local array is performed in stages, where each stage operates on a different part of the array called a *slab*. The size of each slab is equal to the size of the ICLA. As a result, during the compilation, the iteration space of a FORALL statement is sectioned (*stripmined*) so that each iteration operates on the data that can fit in the processor's memory (ie. the size of ICLA).

Figure 8 shows the various steps involved in translating an out-of-core HPF program. The compilation consists of two phases. In the first phase, called the *in-core phase*, the arrays in the source HPF program are partitioned according to the distribution information (provided by the HPF directives) and local lower/upper bounds for each local array are calculated. Array expressions are then analyzed for detecting communication. In other words, the compilation in this phase proceeds in the manner compilation is done for in-core programs. The second phase, called the *out-of-core phase*, involves adding appropriate statements to perform I/O and communication. The local array

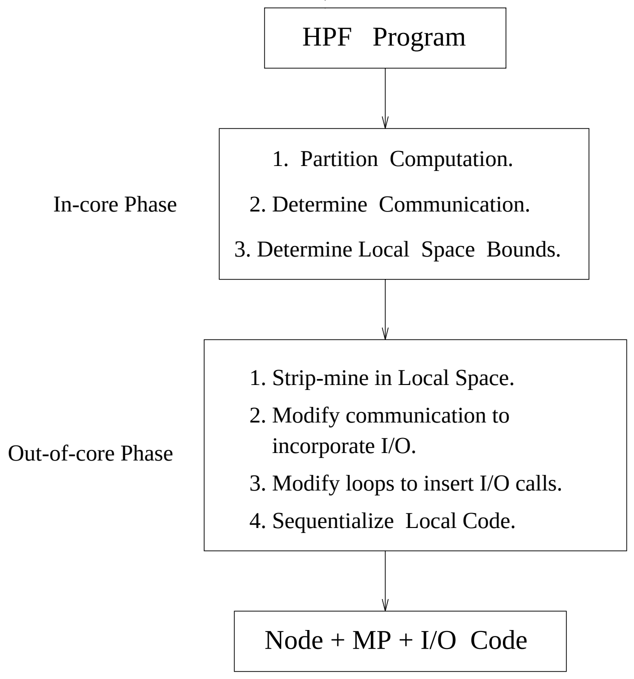
HPF Program
In-core Phase
1. Partition Computation.
2. Determine Communication.
3. Determine Local Space Bounds.
Out-of-core Phase
1. Strip-mine in Local Space.
2. Modify communication to incorporate I/O.
3. Modify loops to insert I/O calls.
4. Sequentialize Local Code.
Node + MP + I/O Code

Figure 8: Flow chart for out-of-core compilation

is first stripmined according to the memory available in each processor. The resulting *slabs* are analyzed for communication. The local FORALL loop is then sequentialized and the loops are modified to insert necessary I/O calls. Note that I/O is performed in the local name space.

### 3.3.1 Compiling the Out-of-core Matrix Multiplication Program

We illustrate the out-of-core compilation process using the HPF matrix multiplication example given in Figure 3 and assume that the arrays A, B and C are out-of-core. Arrays A and C are distributed in column-block fashion over $p$ processors, whereas array B is distributed in row-block fashion. Figure 9 shows the global arrays and their local array files. Figure 9(A) shows array A, Figure 9(B) shows array B and Figure 9(C) shows array C. Consider processor 3 in Figures 9(A) and 9(C). Figure 9(D) shows the local array corresponding to either array A or C and the corresponding OCLA. The OCLA of processor 3 is divided into *slabs*, each of which is equal to the size of the in-core local array (ICLA). The slabs are shown with different shades. Figure 9(E) shows the local array corresponding to array B for processor 3. The OCLA is divided into two slabs where each slab contains four subcolumns.

The compilation is performed in two phases. In the in-core phase, the compiler obtains the necessary information about the arrays from the HPF directives (lines 3–7, Figure 3). Using this information, the compiler analyzes the array operations (lines 8–13, Figure 3). The compiler

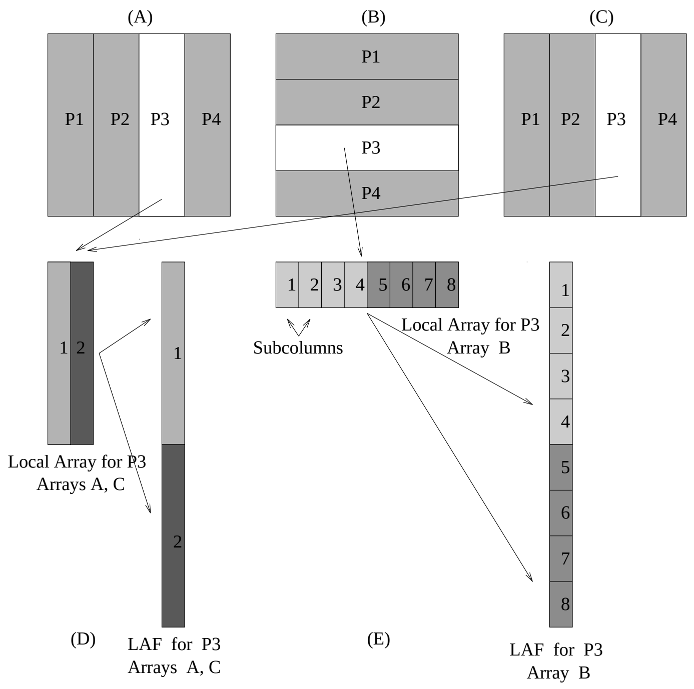

Figure 9: Compiling the Out-of-core Matrix Multiplication Program

analyzes the outer loop (line 8, Figure 3), finds that the loop is a sequential DO loop and hence does not partition it. The inner loop consists of a FORALL construct which is parallelized. Using the array distribution information, the compiler computes the local array bounds and partitions the computation. The compiler analyzes the array assignment statement in line 10 to determine if communication is required. In this case, no communication is required in the innermost loop, but the outer loop requires a global sum operation.

In the second phase, stripmining of the index spaces is carried out using the memory (ICLA) size. Since the outer loop successively fetches elements of B, an I/O routine for fetching the slabs of array B is inserted. The inner loop is also stripmined and another I/O routine is inserted for fetching the slabs of array A. After the execution of the inner loop, all processors add their temporary results to obtain the corresponding columns of C. Using the distribution information of array C and the value of the outer loop index ($j$), the index of the processor that owns these columns is computed. This processor computes the local indices of these columns and stores the columns in the local array file. The resulting node program with the communication and I/O calls is shown in Figure 10.

```
C       (N,N) Arrays distributed over $p$ processors.
C       Stripmine code based on the slab size $M$.
C       Repeat operation $k$ times, $k$=(no. of cols. in OCLA of A/no. of cols. in slab) = $N^2/(M\,P)$
C       Initialize global index.
          global_index=0
          do l=1, $k$
              Call I/O routine to read the ICLA of array B.
              do m=1, no_columns_in_icla_of_B
                  global_index=global_index+1
                  column_count = 0
                  do n=1, $k$
                      Call I/O routine to read the ICLA of array A.
                      do i=1, no_columns_in_icla_of_A {M/N}
                          column_count = column_count + 1
                          do j=1, no_elements_per_column {N}
                              temp(j,i) = temp(j,i) + A(j,i)$\times$B(column_count,m)
                          end do
                      end do
                  end do
                  Call Global Sum routine to obtain the (global_index)$^{th}$ column of C
                  if (mynode is owner of this column) then
                      Store the column in the corresponding ICLA.
                          if ICLA is full then
                              Call I/O routine to write the ICLA of array C.
                          end if
                  end if
              end do
          end do
```

Figure 10: Node+MP+I/O Pseudo-Code for the Matrix Multiplication Program

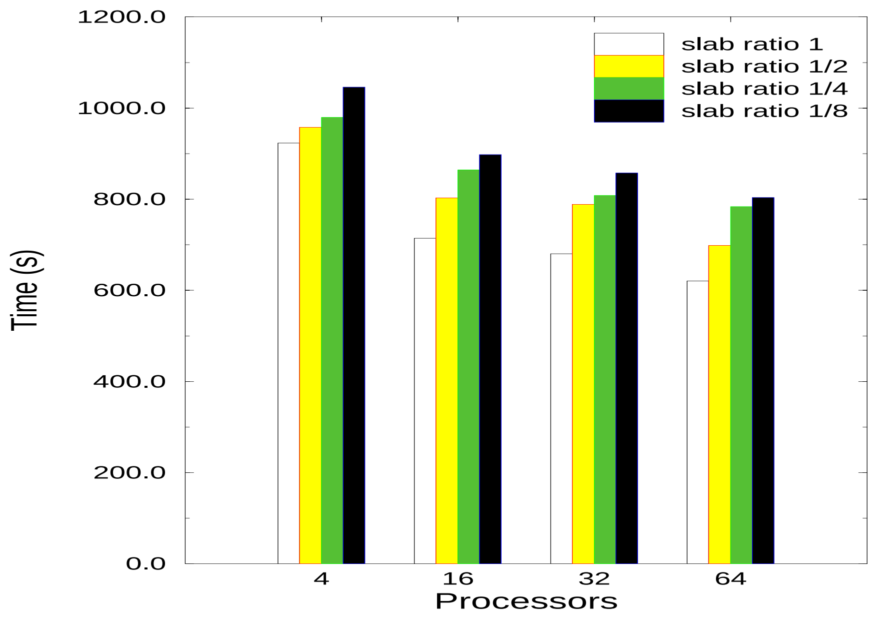

Figure 11: Effect of slab size variation

### 3.4 Experimental Results

Figure 11 shows the performance for multiplication of 1K×1K real arrays on 4, 16, 32 and 64 processors. The slab ratio, which is the ratio of the slab size to the out-of-core local array size is varied from (1/8) to 1. The slab-size for array A is chosen to be equal to the slab-size for array B. Note that the case when the slab-size is equal to the OCLA size (slab ratio = 1) is different from the case when the entire data is stored in main memory. When the slab-size equals the size of the OCLA, the slab-ratio ($k$ in Figure 10) is 1. Even so, data is still accessed from disk, but only once for each column of C. We observe that as the slab ratio is decreased, the time taken increases. This is because a lower slab ratio means a smaller slab size and more number of slabs. This increases the number of I/O requests, though the total amount of data fetched from disk remains the same. The larger number of I/O requests increases the time taken for I/O which results in higher overall execution time. In the next section, we present optimizations which reduce the I/O time significantly.

## 4 Data Access Optimization

For in-core programs, interprocessor communication is often the bottleneck which can degrade the overall performance considerably. Hence, an important optimization to be performed by any compiler for in-core programs is to minimize the communication overhead. This is usually done by aggregating small messages into a single long message so as to reduce communication latency, using collective communication routines etc. For an out-of-core compiler, it is very important to minimize the I/O cost because the time required to fetch data from disk is at least an order of magnitude more than the time required to communicate data between processors.

We measure I/O cost in terms of two metrics, namely the number of I/O requests per processor and the total amount of data fetched from disk per processor. Since the cost associated with physically accessing data (e.g. seek time, latency time etc.) is dictated by the hardware and to a certain extent by the parallel file system, these two metrics can be used to effectively analyze the I/O costs associated with a user program.

## 4.1 I/O Cost Estimation

The previous section presented a simple extension of the in-core compilation method to compile out-of-core programs. Specifically, the extension of the in-core compilation technique did not explicitly consider the I/O costs associated with array assignment statements involving out-of-core arrays. In this section, we describe a framework for estimating the I/O costs in such statements and using this estimate to determine better access patterns which reduce the I/O cost.

In order to estimate the I/O cost associated with the compiled code, we analyze the node+MP+I/O program generated by the out-of-core compiler, which was described in the previous section (Figure 10). Using the local loop bounds, slab sizes and index variables for each out-of-core array, we compute the number of I/O accesses and the total amount of data accessed. We illustrate this by computing the dominant I/O costs in the out-of-core program in Figure 10. We call this version of the translated program as the *column slab version* because the out-of-core local array is divided into slabs along columns as shown in Figure 12(I). Note that for each column of B, the entire local array of A is required. Thus the dominant I/O cost is given by I/O accesses associated with the array A. Further, note that arrays B and C are accessed once during the entire computation, one slab at a time.

The I/O cost for accessing array A in the column slab version can be calculated as follows. Let

$N$ = number of rows and columns in the global array A,

$P$ = number of processors,

$M$ = number of array elements in a slab (This depends on the available node memory).

To calculate one column of C, all columns of A are required. Therefore, the number of I/O requests per processor for one column of C is $(N/P)/(M/N) = N^2/(PM)$. The number of I/O requests

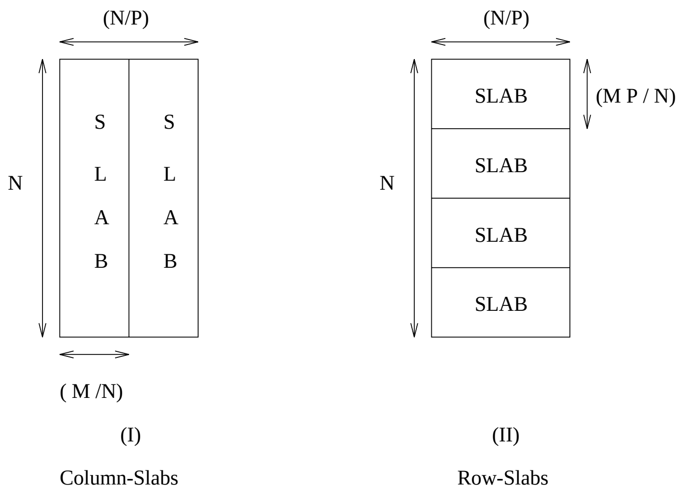

Figure 12: Column slabs versus row slabs

per processor to calculate all columns of C is given by

$$T_{fetch}(A) = N^3/(M\,P) \tag{3}$$

The number of elements of array A that are fetched by each processor to compute one column of C is $N(N/P) = N^2/P$. Hence, the total number of elements of A that are accessed by each processor to compute all columns of C is given by

$$T_{data}(A) = N^3/P \tag{4}$$

Clearly, the I/O cost associated with this code is as large as the amount of computation. As explained in the previous section, it is important to note that the in-core version of this compiled code is as optimized as the hand-coded version.

In this example, another way of accessing the array A is to create slabs along rows as shown in Figure 12(II). This would require reordering the loops as illustrated in Figure 13. We call this version of the translated program as the *row slab version*. The I/O cost associated with this version can be calculated as follows. We note that a row slab of array A consists of all the columns of A within a particular set of rows. Thus when a processor fetches a slab, it has all the subcolumns of the out-of-core local array and the size of the subcolumns is the same as the number of rows in the slab. Since each slab has all the subcolumns necessary to calculate one subcolumn of C, each slab needs to be fetched only once to compute all the columns of C. Hence, in the row slab version, the number of I/O requests per processor is equal to the number of slabs which can be calculated as

$$T_{fetch}(A) = N/((M\,P)/N) = N^2/(M\,P) \tag{5}$$

Since the array A is accessed only once, the number of data elements of A fetched per processor to compute all columns of C is given by

$$T_{data}(A) = N(N/P) = N^2/P \tag{6}$$

Comparing equations 3 and 4 with equations 5 and 6, we observe that the row slab version requires an order of magnitude less number of I/O requests and an order of magnitude less amount of data to be fetched from disk than the column slab version. The I/O costs associated with arrays B and C are the same in both versions. So, the row slab version is clearly the method of choice for translating the out-of-core GAXPY matrix multiplication program. This data access pattern and the corresponding computation reorganization is further illustrated using Figure 14. The elements of array B that are multiplied to the corresponding columns of array A are shown using the same shade. The same slab of array A is multiplied with the $j^{th}$ column to $k^{th}$ column of B to produce the $j^{th}$ to $k^{th}$ subcolumns of C. Thus, the repeated I/O accesses to array A in the column slab version are eliminated.

In general, this approach for estimating the I/O cost requires analyzing the storage and access patterns along each dimension of the distributed out-of-core array. Based on this analysis, the loops are reorganized and the corresponding I/O costs are computed. The version with the minimum I/O cost is selected. This is summarized in the algorithm given in Figure 15.

## 4.2 Performance Results

Table 1 compares the performance of the row and column slab versions of the out-of-core matrix multiplication program. Two arrays of 1K$\times$1K real numbers are multiplied on 4, 16, 32 and 64 processors. The slab ratio, which is the ratio of the slab size to the OCLA size, is varied from (1/8) to 1. We have also measured the time for an in-core version of the program which requires only an initial read of the arrays from disk. As explained earlier, this is different from the case when the slab size is 1 because in the latter case, the array is assumed to be out-of-core even though the entire out-of-core local array is stored in one slab. This slab is fetched from disk whenever necessary.

We observe that the row slab version performs considerably better than the column slab version for any number of processors and any slab size. This is because it requires an order of magnitude less amount of I/O, as proved earlier. In both versions, the time taken increases as the slab ratio (or slab size) is decreased. A smaller slab size results in higher number of I/O requests, which increases the I/O cost. The difference between the in-core version and any of the out-of-core versions shows the corresponding time spent in performing I/O.

```
C       (N,N) Arrays distributed over $p$ processors.
C       Stripmine code based on the slab size $M$.
C       Repeat Operation $k$ times, $k$=(no. of rows in OCLA of A/no. of rows in slab)=$N^2/(M\ P)$
          do l=1, $k$
               *Call I/O routine to read the ICLA of array A.*
               global_index=0
               do n=1, $k$
                    *Call I/O routine to read the ICLA of array B.*
                    do m=1, no_columns_in_icla_of_B
                         global_index=global_index+1
                         do i=1, no_columns_in_icla_of_A {N/P}
                              do j=1, no_elements_per_column {(M P)/N}
                                   temp(j,i) = temp(j,i) + A(j,i)$\times$B(i,m)
                              end do
                         end do
                         Call Global Sum intrinsic to obtain the (global_index)$^{th}$ subcolumn of C
                         if (mynode is owner of this subcolumn) then
                              Store the subcolumn in the corresponding ICLA.
                              if ICLA is full then
                                   *Call I/O routine to write the ICLA of array C.*
                              end if
                         end if
                    end do
               end do
          end do
```

Figure 13: Row Slab Version of the translated code

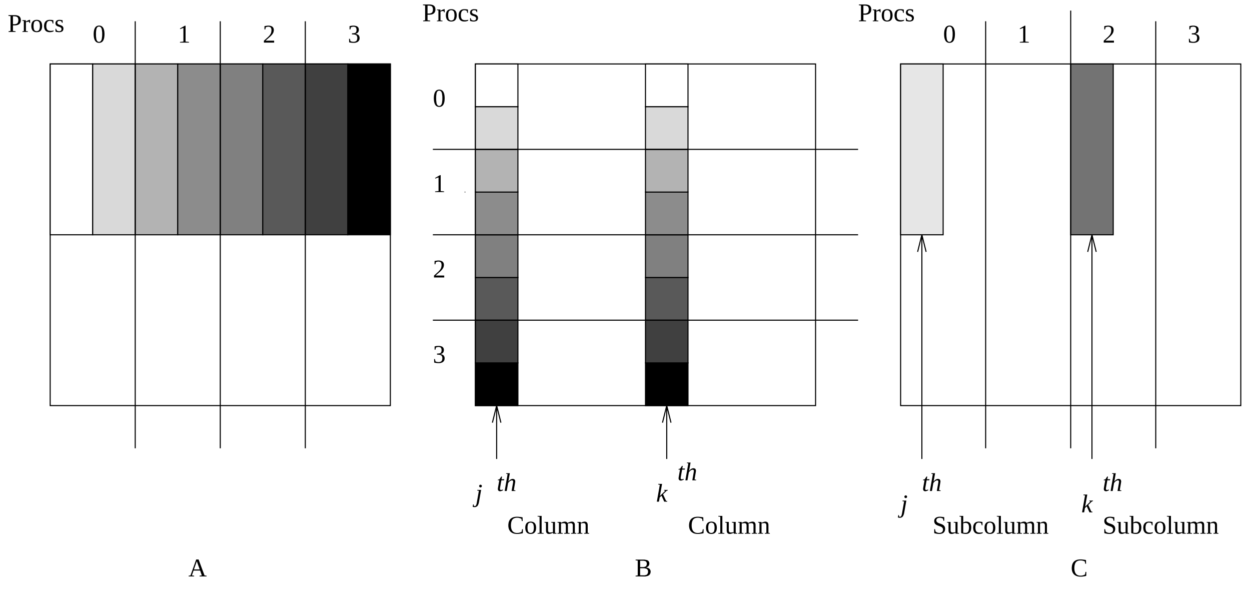

Figure 14: Row slab version

```
Determine the amount of available memory.
For each array used in the array assignment statement do
    For each dimension of the out-of-core array do
        Use index variables to analyze access patterns.
        Compute the I/O costs for stripmining using slabs along this dimension.
    end for
end for
Determine which array requires the largest amount of I/O.
Select the stripmining strategy which results in lowest I/O cost for that array.
```

Figure 15: General Algorithm for I/O Cost Estimation

Table 1: Performance of matrix multiplication for various slab sizes, time in seconds

| Slab Ratio | 4 Procs | | 16 Procs | | 32 Procs | | 64 Procs | |
|---|---|---|---|---|---|---|---|---|
| | Col. slab | Row slab | Col. slab | Row slab | Col. slab | Row slab | Col. slab | Row slab |
| 1/8 | 1045.84 | 239.97 | 897.59 | 161.02 | 857.62 | 97.08 | 803.57 | 90.29 |
| 1/4 | 979.20 | 226.08 | 864.08 | 118.20 | 807.99 | 92.43 | 783.79 | 75.56 |
| 1/2 | 958.17 | 205.91 | 802.69 | 96.79 | 788.47 | 80.45 | 698.29 | 66.70 |
| 1 | 923.11 | 194.15 | 714.15 | 84.77 | 680.40 | 66.94 | 620.70 | 60.11 |
| In-core | 140.91 | | 40.40 | | 20.14 | | 9.58 | |

### 4.2.1 Selecting Slab Sizes for Multiple Arrays

The compiler has to choose the slab sizes to be used for all arrays in the program depending on the amount of available memory. One approach is to distribute the available memory equally among all the arrays, so that they all have the same slab size. Another approach is to analyze the I/O access patterns of the arrays and assign a larger slab size to the array with more frequent accesses. We have studied the effect of different slab sizes on the overall performance. Table 2 shows the performance of the matrix multiplication program for different slab sizes for arrays A and B. The arrays are chosen to be of size 2K$\times$2K. In the first experiment, the slab size for array A is fixed and the slab size for array B is varied. The second experiment is performed by keeping the slab size for array B fixed. While in the first experiment, the execution time improves from 826.94 seconds to 493.04 seconds, in the second experiment the best performance observed is 452.29 seconds. Hence, instead of equally dividing the available memory between the slabs of A and B, if a larger portion is allocated to the slab of A, better performance is obtained. This is because A is accessed more often than B or C. Hence the compiler should allocate more memory to array A. As explained earlier, using the loop bounds and index variables, the compiler can determine which array requires more I/O accesses and accordingly allocate the available memory.

## 5 Conclusions

We have described how an out-of-core program written in a data parallel language like HPF can be translated into a message passing node program with explicit communication and parallel I/O. Such a compiler is necessary for compiling large scale scientific applications written in a data parallel language. These applications typically handle large quantities of data which results in the program being out-of-core.

We have discussed the basic compilation model and various steps involved in the compilation. The compilation process was illustrated using an out-of-core matrix multiplication example. We

Table 2: Performance of the row slab version for different slab sizes of arrays A and B

2K $\times$ 2K arrays, 16 processors, time in seconds

| Slab_B size | Slab_A=256 Time (s) | Slab_A size | slab_B=256 Time (s) | Total Memory (Slab_A + Slab_B) |
|---|---|---|---|---|
| 256 | 826.94 | 256 | 826.94 | 512 |
| 512 | 548.13 | 512 | 510.02 | 768 |
| 1024 | 507.01 | 1024 | 492.87 | 1280 |
| 2048 | 493.04 | 2048 | 452.29 | 2304 |

described how the basic in-core compilation method can be extended to compile out-of-core programs. However, the code generated this way may not give good performance. We have proposed an optimization by which the compiler can improve the code generated by the above method. The compiler estimates the I/O costs associated with different array access patterns and selects the method with the least I/O cost. This can reduce the amount of I/O by as much as an order of magnitude. We also discussed how the performance of the program varies with slab size. Instead of dividing the available memory equally among all arrays, the best performance is obtained when the most frequently accessed array is allocated a larger slab size.

Some of the compilation techniques described in this paper are currently being done by hand. We are in the process of implementing them in the compiler. Due to stability problems with the hardware, we have done experiments with relatively small data sets. We will include complete results on much larger data sets in the final version of this paper.

## Acknowledgments

We would like to thank Ken Kennedy and Chuck Koelbel for many enlightening discussions. We would also like to thank our compiler group at Syracuse University for their help with the basic infrastructure of the in-core HPF compiler.

## References

[AK87] Randy Allen and Ken Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Trans. Programs. Lang. Syst.*, 9(4):491–542, October 1987.

[BBG+93] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. In *Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 1–24, April 1993.

[BC93] Rajesh R. Bordawekar and Alok N. Choudhary. Compiler and Runtime Support for Parallel I/O. In *Fourth Workshop on Compilers for Parallel Computers*, pages 171–189, Delft, The Netherlands., December 1993. Delft University of Technology.

[BCF+93] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results. In *Proceedings of Supercomputing '93*, pages 351–360, November 1993.

[CFPB93] P. Corbett, D. Feitelson, J. Prost, and S. Baylor. Parallel Access to Files in the Vesta File System. In *Proceedings of Supercomputing '93*, pages 472–481, November 1993.

[CR93] High Performance Computing and Communications: Grand Challenges 1993 Report. A Report by the Committee on Physical, Mathematical and Engineering Sciences, Federal Coordinating Council for Science, Engineering and Technology, 1993.

[DdR92] E. DeBenedictis and J. del Rosario. nCUBE Parallel I/O Software. In *Proceedings of* $11^{th}$ *International Phoenix Conference on Computers and Communications*, pages 117–124, April 1992.

[dRC94] J. del Rosario and A. Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. *IEEE Computer*, pages 59–68, March 1994.

[FHK+90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. Fortran D Language Specifications. Technical Report COMP TR90-141, Rice University, 1990.

[For93] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Version 1.0, May 1993.

[Fox91] Geoffrey Fox. The Architecture of Problems and Portable Parallel Software Systems. Technical Report SCCS-78b, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY 13244, 1991.

[Pie89] P. Pierce. A Concurrent File System for a Highly Parallel Mass Storage Subsystem. In *Proceedings of* $4^{th}$ *Conference on Hypercubes, Concurrent Computers and Applications*, pages 155–160, March 1989.

[SS93] S. Saini and H. Simon. Enhancing Applications Performance on Intel Paragon through Dynamic Memory Allocation. In *Proceedings of the Scalable Parallel Libraries Conference, Mississippe State University*, October 1993.

[Wol89] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[ZBC$^+$92] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - a Language Specification. Technical Report ICASE Interim Report 21, MS 132c, ICASE, NASA, Hampton VA 23681, 1992.

[ZC91] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, NY, 1991.