Compiler and Runtime Support for Out-of-Core HPF Programs

Rajeev Thakur **Rajesh Bordawekar** **Alok Choudhary**
Dept. of Electrical and Computer Eng. and
Northeast Parallel Architectures Center
Syracuse University, Syracuse, NY 13244, USA
thakur, rajesh, choudhar @npac.syr.edu

Abstract

This paper describes the design of a compiler which can translate out-of-core programs written in a data parallel language like HPF. Such a compiler is required for compiling large scale scientific applications, such as the *Grand Challenge* applications, which deal with enormous quantities of data. We propose a framework by which a compiler together with appropriate runtime support can translate an out-of-core HPF program to a message passing node program with explicit parallel I/O. We describe the basic model of the compiler and the various transformations made by the compiler. We also discuss the runtime routines used by the compiler for I/O and communication. In order to minimize I/O, the runtime support system can *reuse* data already fetched into memory. The working of the compiler is illustrated using two out-of-core applications, namely a Laplace equation solver and LU Decomposition, together with performance results on the Intel Touchstone Delta.

1 Introduction

Massively parallel computers (MPPs) with a peak performance as high as 100 GFlops have made their advent into the supercomputing arena. As a result, MPPs are increasingly being used to solve large scale computational problems in physics, chemistry, biology, engineering, medicine and other sciences. These applications, which are also referred to as *Grand Challenge Applications* [14], are extremely complex and require several Teraflops of computing power to be solved in a reasonable amount of time. 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 [12]. These figures are expected to increase by orders of magnitude as teraflop machines make their appearance.

Although supercomputers have very large main memories, the memory is not large enough to hold this much amount of data. Hence, data needs to be stored on disk and the performance of the program depends on how fast the processors can access data from disks. A poor I/O capability can severely degrade the performance of the entire program. 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 [10, 17, 4, 11]. A good overview of the various issues in high performance I/O is given in [12]

In this paper, we consider the I/O problem from a language and compiler point of view. Data parallel languages like HPF [15] and pC++ [2] have recently been developed to provide support for high performance programming on parallel machines. These languages provide a framework for writing portable parallel programs independent of the underlying architecture and other idiosyncrasies of the machine. In order that these languages can be used for programming Grand Challenge Applications, it is essential that the compiler can automatically translate out-of-core data parallel programs. Language support for out-of-core programs has been proposed in [3, 8, 20]. We propose a framework by which a compiler together with appropriate runtime support can translate an out-of-core HPF program to a message passing node program with explicit parallel I/O. Although we use HPF as the source language, the translation technique is applicable to any other data parallel language. There has been considerable research on compiling in-core data parallel programs for distributed memory machines [6, 22, 21]. This work, to our knowledge, is one of the first attempts at a methodology for compiling out-of-core data parallel programs.

The rest of the paper is organized as follows. The model for out-of-core compilation is explained in Section 2. Section 3 describes the compiler design including the transformations made by the compiler. The runtime support system is described in Section 4. We use two out-of-core examples to demonstrate the working of the compiler, namely the solution of Laplace's equation and LU Decomposition. We discuss some performance results on the Intel Touchstone Delta in Section 5, followed by Conclusions in Section 6. 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

High Performance Fortran (HPF) is an extension to Fortran 90 with features to specify data distribution, alignment, data parallel execution etc. In distributed memory

Proc. of the 8^{th} ACM International Conference on Supercomputing, Manchester, England, July 1994.

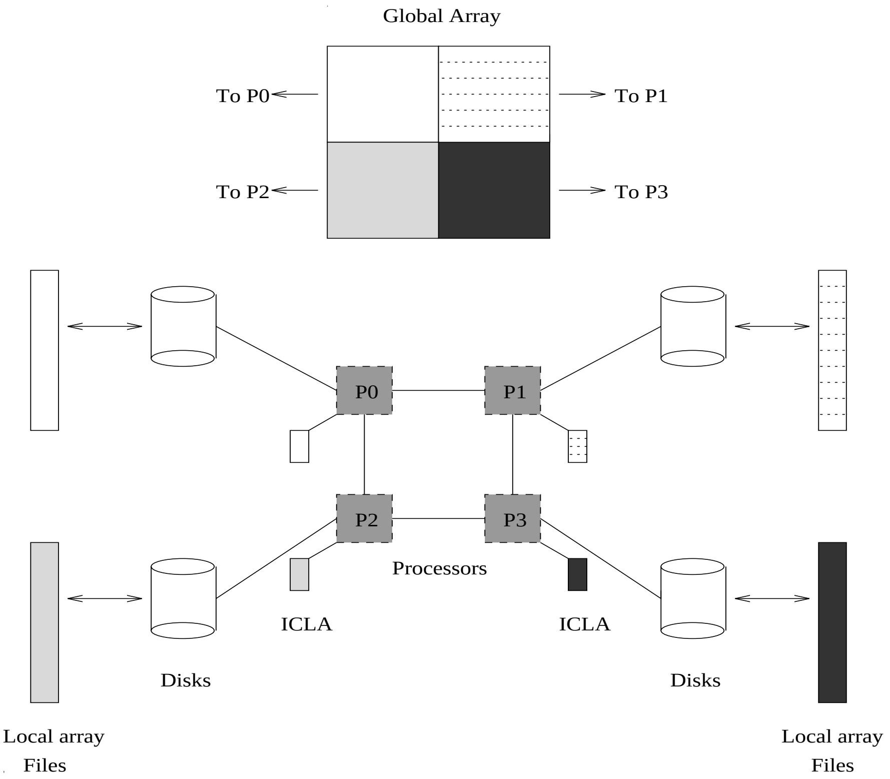

Figure 1: Model for out-of-core compilation

computers, arrays have to be distributed among the processors in some manner. HPF provides directives (ALIGN and DISTRIBUTE) which describe how arrays should be partitioned. Arrays are first aligned to a *template* or index space using the ALIGN directive. The DISTRIBUTE directive specifies how the template is to be distributed among the processors. In HPF, an array can be distributed as either BLOCK(m) or CYCLIC(m). In a BLOCK(m) distribution, contiguous blocks of size m of the array are distributed among the processors. In a CYCLIC(m) distribution, blocks of size m are distributed cyclically.

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 [6]. For large data sets, however, local arrays cannot entirely fit in main memory. Hence, these programs are referred to as out-of-core programs. In such cases, parts of the local array have to be stored on disk. We refer to such a local array as the **Out-of-core Local Array (OCLA)**. Parts of the OCLA have to be swapped between main memory and disk during the course of the computation. If the operating system supports virtual memory on each processor, the OCLA can be swapped in and out of the disk automatically by the operating system. In that case, the HPF compiler for in-core programs could also be used for out-of-core programs.

A major consideration in the design of an HPF compiler is performance. Studies of the performance of 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 degrades the performance of the user code [19]. 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. Hence, the HPF compiler must translate into a code which explicitly performs I/O. Even if 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 where, in order to get good performance, the programmer must explicitly do I/O instead of leaving it to the operating system.

The basic model on which the out-of-core compilation is based is shown in Figure 1. Since the local arrays are out-of-core, they have to be stored in files on disk. The local array of each processor is stored in a separate file called the **Local Array File (LAF)** of that processor. The node program explicitly reads from and writes into the file when required. If the I/O architecture of the system is such that each processor has its own disk, 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, 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. At any time, only a portion of the local array is fetched and stored in main memory. The size of this portion is specified at compile time and it usually depends on the amount of memory available. The portion of the local array which is in main memory is called the **In-Core Local Array (ICLA)**. All computations are performed on the data in the ICLA. Thus, during the course of the program, parts of the LAF are fetched into the ICLA, the new values are computed and the ICLA is stored back into appropriate locations in the LAF.

3 Compiler Design

This section describes the design of the compiler for out-of-core HPF programs. We mainly focus on the compilation of array expressions and FORALL statements for out-of-core arrays. The compilation basically involves the following stages: Data Partitioning, Communication Detection, I/O Detection and finally Code Generation with calls to runtime libraries. We first describe how the compilation is done for in-core programs and then extend the methodology to out-of-core programs. We explain both cases with the help of the HPF program fragment given in Figure 2. In this example, arrays A and B are distributed in (block,block) fashion on 16 processors arranged as a two-dimensional grid of 4 × 4.

3.1 In-core Compilation

This section describes the in-core compilation methodology used in the HPF compiler developed by our group at Syracuse University [6]. Consider the array assignment statement from Figure 2. The compiler translates this statement using the following steps:-

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 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 sequential F77 code.
6. Add calls to runtime libraries to perform collective communication.

The local arrays corresponding to arrays A and B lie in the local memory of each processor. Since array B is distributed in block-block fashion over 16 processors, the above assignment requires fetching data from neighboring processors. The compiler analyzes the statement and inserts a call to the appropriate collective communication routine. The assignment statement is translated into corresponding DO loops with a call to a routine which performs overlap shift type communication [6], as shown in Figure 3.

3.2 Out-of-core Compilation

For compiling out-of-core programs, in addition to handling all the issues involved in compiling in-core programs, the compiler must also 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 explained earlier, the local array of each processor is stored in a separate local array file (LAF) and the portion of the local array currently required for computation 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 the ICLA.

```
parameter (n=1024)
real A(n,n), B(n,n)
..........
!HPF$ PROCESSORS P(4,4)
!HPF$ TEMPLATE T(n,n)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO P
!HPF$ ALIGN with T :: A, B
...........
    FORALL (i=2:n-1, j=2:n-1)
        A(i,j) = (B(i,j-1) + B(i,j+1) + B(i+1,j)
            + B(i-1,j))/4
...........
    B = A
```

Figure 2: HPF Program Fragment

```
Call communication routine to perform overlap shift.
do j = lower_bound, upper_bound
    do i = lower_bound, upper_bound
        A(i,j) = (B(i,j-1) + B(i,j+1) + B(i-1,j) +
            B(i+1,j))/4
    end do
end do
```

Figure 3: Translation of the Array Assignment Statement.

Some of the issues in out-of-core compilation are similar to optimizations carried out in in-core compilers to take advantage of caches or pipelines. This optimization, commonly known as *stripmining* [23, 24], partitions the loop iterations so that data of fixed size (equal to cache size or pipeline stages) can 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, the iteration space of the local array assignment/forall statement is partitioned (*stripmined*) so that each iteration operates on the data that can fit in the processor's memory (ie. the size of ICLA). In other words, there are two levels of data partitioning. Data is first partitioned among processors and then data within a processor is partitioned into slabs which fit in the processor's local memory.

3.2.1 Language Support for Out-of-Core Compilation

In order to stripmine the array assignment statements, the compiler needs information about which arrays are out-of-core and also the amount of memory available to store the ICLA. We propose two directives, **OUT_OF_CORE** and **MEMORY**, using which the user can specify this information to the compiler. The HPF program can be an-

```
    parameter (n=64000)
    !HPF$ OUT_OF_CORE :: D, C
    !HPF$ MEMORY M(n)
```

Figure 4: Proposed Out-of-Core Directives for HPF

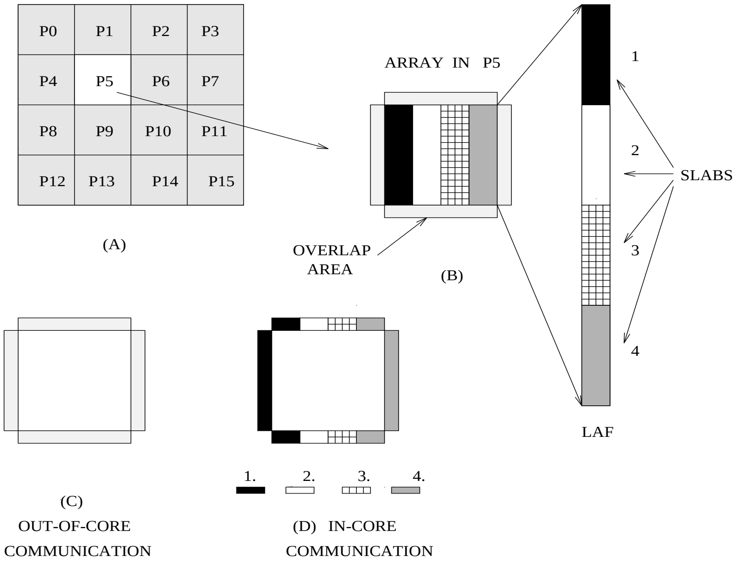

Figure 5: Compilation of Out-of-core Programs

notated with these directives, as shown in Figure 4. The **OUT_OF_CORE** directive specifies which arrays are out-of-core (e.g D, C). The **MEMORY** directive specifies the amount of memory available for the ICLA. In the future, we plan to incorporate some optimizations in the compiler by which the compiler will be able to automatically calculate the memory available for the ICLA on the basis of the amount of memory provided on each processor and the memory used by the program.

3.2.2 Communication Models for Out-of-Core Compilation

Let us now examine the compilation of array assignment statements involving out-of-core arrays. We consider the array assignment statement from the HPF program shown in Figure 2.

A(i,j) = (B(i-1,j) + B(i+1,j) + B(i,j-1) + B(i,j+1))/4

Array B is distributed over 16 processors in (block,block) manner as shown in Figure 5(A). Consider the out-of-core local array (OCLA) and corresponding local array file (LAF) for processor 5, shown in Figure 5(B). The OCLA is divided into slabs, each of which is equal to the size of the in-core local array (ICLA). The slabs are shown using columns with different shades. The same figure shows the overlap area (also called ghost cells) for array B for the above array assignment statement. The overlap area is used to store the data received from other processors.

Each point (i, j) of the array is computed using the values at its four neighboring points. Hence each processor, except those at the boundaries, needs to get one row or one column from each of its four neighboring processors. There are two ways in which this communication can be done, which we call the *Out-of-core Communication Method* and the *In-core Communication Method*. Figure 6 describes how the compilation is done for each of these methods.

- **Out-of-core Communication Method**: In this method, the compiler determines what off-processor data is required for the entire out-of-core local array. The shaded region in Figure 5(C) shows the amount of data to be received by processor 5 (*comm_data*). In the out-of-core communication method, the entire *comm_data* is communicated in one step and stored at appropriate locations in the local array file. The computation is stripmined using the memory size provided by the user. During the computation, each slab along with its *comm_data*, is read from and written to the local array file. No communication is required during the computation on each slab, since the necessary *comm_data* is fetched from the local array file. After the computation, the slab is written back to disk.

  The out-of-core communication method requires extra data storage during program execution. Also, the communication stage requires accessing data from other processors (inter-processor communication) and storing data to the local array file (disk access). However, this method allows the compiler to identify and optimize collective communication patterns because the communication pattern depends on the logical shape of arrays and the access patterns for the entire array. For example, there are four shift type communications required in this example. This communication pattern is preserved except that communication also requires disk accesses in addition to data movement. Also, since communication is separated from computation, the compiler can easily perform other code optimizations such as loop fusion.

- **In-core Communication Method**: In this method, the compiler analyzes each slab instead of the entire out-of-core local array. The assignment statement is first stripmined according to the memory size. Then each data slab is analyzed for communication. If the slab requires off-processor data, appropriate communication primitives are used to fetch the necessary data. This is illustrated in Figure 5(D). In this example, the local array file is divided into four data slabs. The shaded region in Figure 5(C) shows the total amount of data to be communicated for the entire OCLA. Figure 5(D) shows the data to be fetched for each in-

Out-of-core Communication

```
1. Stripmine code based on memory size.
2. Schedule communication for entire out-of-core data.
3. Repeat $k$ times ($k$ is the stripmine factor).
     3.1 Read data from disk to ICLA.
     3.2 Do the computation on the data in ICLA.
     3.3 Write data from ICLA back to disk.
```

In-core Communication

```
1. Stripmine code based on memory size.
2. Repeat $k$ times ($k$ is the stripmine factor).
     2.1 Read data from disk to ICLA.
     2.2 Schedule communication for in-core data.
     2.3 Do the computation on the data in ICLA.
     2.4 Write data from ICLA back to disk.
```

Figure 6: Compiling for out-of-core and in-core communication

dividual slab (*comm_data*). Each shade represents a different slab. Consider the last two slabs. The last slab needs data from three other processors whereas the slab before it needs data from two other processors. Thus, the communication patterns for the slabs within the same local array are different.

Since the *comm_data* is stored in the ICLA, this method does not require disk accesses to store the *comm_data*. After the necessary *comm_data* is fetched, the computation on each *slab* is done. Since the communication pattern for each slab may be different, the compiler needs to analyze each slab separately and insert appropriate communication calls to get the necessary data. Optimizing such communication patterns can be difficult. It requires extensive pre-processing and the translated code looks unreadable.

3.3 Compiling Out-of-core Array Assignment Statements

Array assignments involving distributed arrays often result in different communication patterns [9, 13]. The compiler must recognize the type of communication in order to generate appropriate runtime calls (communication as well as I/O). It is relatively easier to detect and optimize the communication in the out-of-core communication method than in the in-core communication method. Also, since communication is performed with respect to the entire out-of-core array and for each assignment statement there is a single call to a communication routine, the overall communication overhead is independent of the number of slabs and the size of the ICLA. Hence, we prefer to use the out-of-core communication method.

Detecting the type of communication required in an array assignment statement involves analyzing the relationships among the subscripts of the arrays in the statement [6, 16, 13]. I/O pattern detection involves analyzing I/O characteristics of array expressions. There are many factors that influence the I/O access patterns. Important among these are :-

- How the array is distributed among the processors.
- What is the communication pattern in the array expression.
- How the array is stored in the local array file (eg. column major/row major).
- How the file system stores the local array file (number of disks, data striping etc).
- How many processors read the files.

After detecting the type of communication and I/O, the compiler performs basic code optimizations. These optimizations rearrange the code so that the overhead of communication and I/O can be reduced. The compiler then inserts calls to appropriate runtime routines depending on the I/O access pattern and communication.

4 Runtime Support

As discussed earlier, each processor has an out-of-core local array (OCLA) stored in a local array file (LAF) and there is an in-core local array (ICLA) which is used to store the portion of the OCLA currently being used for computation. During program execution, it is necessary to fetch data from the LAF into the ICLA and store the newly computed values from the ICLA back into appropriate locations in the LAF. Since the global array is distributed, a processor may need data from the local array of another processor. This requires data to be communicated between processors. Thus the node program needs to perform I/O as well as communication, both of which are not explicit in the source HPF program.

The compiler does basic code transformations such as partitioning of data and computation, and inserts calls to runtime library routines for disk accesses and communication. The runtime support system for the compiler consists of a set of high level specialized routines for parallel I/O and collective communication. These routines are built using the native communication and I/O primitives of the system and provide a high level abstraction which avoids the inconvenience of working directly with the lower layers. For example, the routines hide details such as buffering, mapping of files on disks, location of data in files, synchronization, optimum message size for communication, best communication algorithms, communication scheduling, I/O scheduling etc.

Runtime support has been used previously as an aide to the compiler. Runtime primitives for the initial reading of data from a file for an in-core program are discussed in [5]. The in-core HPF compiler developed by our group at Syracuse University uses runtime support [7, 1]. Ponnusamy et al [18] describe how runtime support can be integrated with a compiler to solve unstructured problems with irregular communication patterns. These projects only deal with compilation of in-core programs, so the runtime support is mainly limited to communication libraries. The runtime support for our out-of-core compiler is different in the sense that in addition to having routines which perform only I/O, even the communication routines need to do I/O.

4.1 Issues in Runtime Support

Consider the HPF program fragment given in Figure 2. This has the array assignment statement

$$A(i,j) = (B(i,j-1) + B(i,j+1) + B(i+1,j) + B(i-1,j))/4$$

Suppose the arrays A and B are distributed as (block,block) on a 4×4 grid of processors as shown in Figure 7. As an

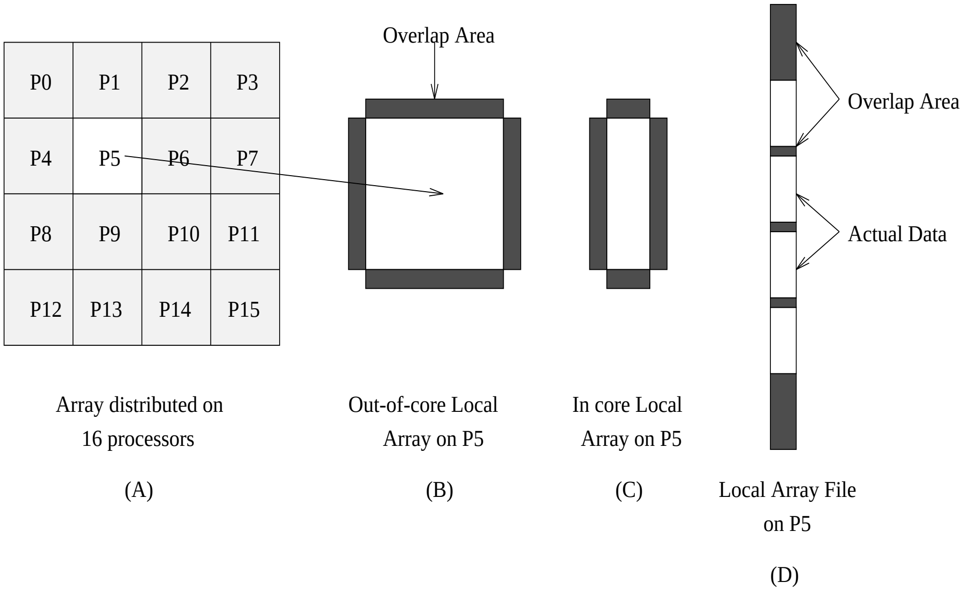

Figure 7: Example of OCLA, ICLA and LAF

example, consider the out-of-core local array on processor P5, shown in Figure 7(B). The value of each element (i, j) of A is calculated using the values of its corresponding four neighbors in B, namely $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ and $(i, j+1)$. Thus to calculate the values at the four boundaries of the local array, P5 needs the last row of the local array of P1, the last column of the local array of P4, the first row of the local array of P9 and the first column of of the local array of P6. Before each iteration of the program, P5 gets these rows and columns from its neighboring processors. If the local array was in-core, these rows and columns would have been placed in the overlap areas shown in the Figure 7(B). This is done so as to obtain better performance by retaining the DO loop even at the boundary. Since the local array is out-of-core, these overlap areas are provided in the local array file. The local array file basically consists of the local array stored in either row-major or column major order. In either case, the local array file will consist of the local array elements interspersed with overlap area as shown in Figure 7(D). Data from the file is read into the in-core local array and new values are computed. The in-core local array also needs overlap area for the same reason as for the out-of-core local array. The example shown in the figure assumes that the local array is stored in the file in column major order. Hence, for local computation, columns have to be fetched from disks and then written back to disks.

At the end of each iteration, processors need to exchange boundary data with neighboring processors. In the in-core case, this would be done using a shift type collective communication routine to directly communicate data from the local memory of the processors. In the out-of-core case, there are two options:-

- **Direct File Access**: Since disks are a shared resource, any processor can access any disk. In the direct file access method, a processor directly reads data from the local array file of some other processor as required by the communication pattern. This requires explicit synchronization at the end of each iteration.
- **Explicit Communication**: Each processor accesses only its own local array file. Data is read into memory and sent to other processors. Similarly, data is received from other processors into main memory and then saved on disk. This is similar to what would be done in in-core compilation methods.

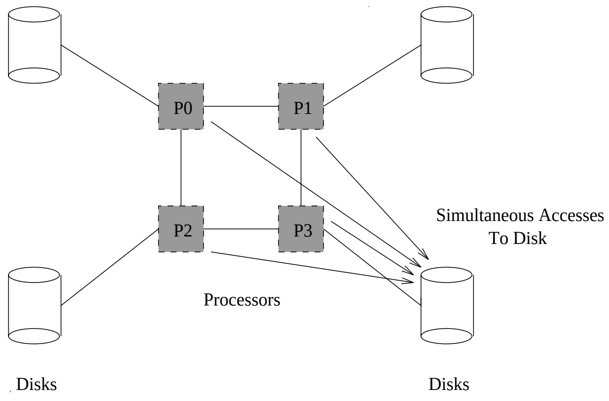

Figure 8: Direct File Access Method

Consider a situation in which each processor needs to communicate with every other processor (all-to-all communication). In the direct file access method, this will result in several processors trying to simultaneously access the same disk as shown in Figure 8, resulting in contention for the disk. A minimum of one block of data, the size of which is system dependent, is transferred during each disk access. Even if a processor actually needs a small amount of data, one whole block will be transferred for each access from every processor. So the direct file access method has the drawback of greater disk contention and higher granularity of data transfer. Also, in some communication patterns (eg. broadcast), the same piece of data may be fetched repeatedly by several processors. In the explicit communication method, each processor accesses only its own local file and reads the data to be sent to other processors into its local

	Dimension						
	1	2	3	4	5	6	7
Incore lb							
Incore ub							
Incore lbo							
Incore ubo							
Global sz							
OCLA size							
Procs							
OOC storage	x	y					
Distribution							
Block sz							

Figure 9: Out-of-Core Array Descriptor (OCAD)

memory. This data is communicated to other processors. Thus, there is no contention for a disk and since the data to be sent to all other processors has to be read from disk, the high granularity of data access from disk is less of a problem. In addition, the time to communicate data between processors is at least an order of magnitude less than the time to fetch data from disk. However, this requires a communication phase in addition to I/O. The relative performance of these two methods on the Touchstone Delta is discussed in Section 5.

4.2 Out-of-Core Array Descriptor (OCAD)

The runtime routines require information about the array such as its size, distribution among the nodes of the distributed memory machine, storage pattern etc. All this information is stored in a data structure called the Out-of-Core Array Descriptor (OCAD) and passed as a parameter to the runtime routines. Before any of the runtime routines are called, the compiler makes a call to a subroutine which fills in the OCAD on the basis of some parameters. The structure of the OCAD is given in Figure 9. Rows 1 and 2 contain the lower and upper bounds of the in-core local array (excluding overlap area) in each dimension. The lower and upper bounds of the in-core local array in each dimension including overlap area are stored in rows 3 and 4. The size of the global array in each dimension is given in row 5. Row 6 contains the size of the out-of-core local array. Row 7 specifies the number of processors assigned to each dimension of the global array. The format in which the out-of-core local array is stored in the local array file is given in Row 8. The array is stored in the order in which array elements are accessed in the program, so as to reduce the I/O cost. The entry for the dimension which is stored first is set to 1, the entry for the dimension which is stored second is set to 2 and so on. For example, for a two-dimensional array, x,y = 1,2 means that the array is stored on disk in column major order and x,y = 2,1 means that the array is stored in row major order. This enables the runtime system to determine the location of any array element (i,j) on the disk. Row 9 contains information about the distribution of the global array. Since the array can be distributed as BLOCK(m) or CYCLIC(m), where m is the block-size, the value of m is stored in Row 10 of the OCAD.

4.3 Runtime Library

We are developing a library of runtime routines using which we can compile any general out-of-core HPF program. The routines are divided into two categories — Array Management Routines and Communication Routines. The Array Management Routines handle the movement of data between the in-core local array and the local array file. The Communication Routines perform collective communication of data in the out-of-core local array. Some of the basic routines are described below.

4.3.1 Array Management Routines

1. **read_vec(file, A, OCAD, i, j, start_vec, end_vec, stride)**
   This routine reads vectors from the local array file to the in-core local array A. The vectors are assumed to be rows if the array is distributed along rows and columns if the array is distributed along columns. The vectors are read starting from number 'start_vec' in the out-of-core local array till vector number 'end_vec', with the specified stride. The vectors are placed in the in-core local array starting from the location (i,j).

2. **write_vec(file, A, OCAD, i, j, start_vec, end_vec, stride)**
   This routine writes vectors starting from location (i,j) in the in-core local array A to the local array file. The location in the file is specified by 'start_vec' and 'end_vec', which are the starting and ending vector numbers in the out-of-core local array, together with a stride.

3. **write_vec_with_reuse(file, A, OCAD, i, j, start_vec, end_vec, stride, left_shift, right_shift)**
   This routine writes vectors from the in-core local array to the local array file as in write_vec. In addition it **reuses data** from the current ICLA slab for the computation involving the next ICLA slab. This is done by moving some vectors from the end of the in-core local array to the front of the in-core local array, in addition to writing all the vectors to the file. This can be explained with the help of Figure 10 and the Laplace equation solver discussed earlier.

   Suppose the array is distributed along columns. Then the computation of each column requires one column from the left and one column from the right. The computation of the last column requires one column from the overlap area and the computation of the column in the overlap area cannot be performed without reading the next column from the disk. Hence, instead of writing the column in the overlap area back to disk and reading it again with the next set of columns, it can be reused by moving it to the first column of the array and the last column can be moved to the overlap area before the first column. If this move is not done, it would be required to read the two columns again from the disk along with data for the next slab. The reuse thus eliminates the reading and writing of two columns in this example. The number of columns to be moved is specified by 'left_shift' and 'right_shift'. 'left_shift' refers to the number of columns from the left that are needed for the computation of any column and 'right_shift' refers to the number of columns from the right. In general, the amount of data reuse would depend on the intersection of the sets of data needed for computations involving two consecutive slabs.

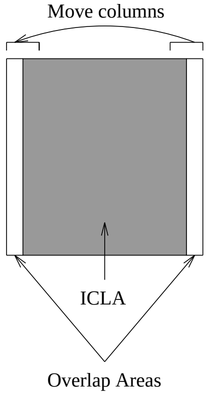

Figure 10: Data Reuse

4.3.2 Communication Routines

1. **out_of_core_shift(file, OCAD, nvec, direction)**
   This is a collective communication routine which does a shift type communication for out-of-core local arrays. It shifts a specified number of vectors to the processor in the specified direction. For example, if the array is distributed along columns and n vectors have to be shifted to the right, each processor (except the last) reads the last n columns in its local array from the local array file and sends them to the processor on the right. Each processor (except the first) receives n columns from the processor on the left and places them in the overlap area at the beginning of the local array file. Data in the local array file is not moved.

2. **out_of_core_multicast(file, OCAD, i, j, nelements, vec, source, proclist)**
   This routine does an out-of-core multicast operation. "source" specifies the source processor and "proclist" is the list of destination processors. A broadcast operation in which data has to be sent to all other processors can be specified by setting proclist(1) to –1. The source processor reads "nelements" from its local array file starting from the element at location (i, j) in the out-of-core local array. These elements are broadcast (or multicast) to the processors specified by proclist. At the destination, the data is stored in the in-core vector "vec".

We have described only a subset of the runtime library in this paper because of space limitations.

5 Examples: Laplace Equation Solver and LU Decomposition

We illustrate the working of the compiler using two out-of-core applications — the first is a Laplace equation solver by Jacobi iteration method and the second is LU decomposition. The Laplace solver program is discussed previously in Sections 3 and 4 (see Figure 2). For simplicity, we consider the case in which the arrays are distributed only in one dimension, along columns. The translated Fortran 77 code using the Explicit Communication Method is given in Figure 11. In the Jacobi iteration method, the new values in each iteration are computed using the values from the previous iteration. This requires the newly computed array to be copied into the old array for the next iteration. In the out-of-core case, this would require copying the local array file. We do an optimization in which instead of explicitly copying the file, the file unit numbers are exchanged after each iteration. This is equivalent to dynamically changing the virtual addresses associated with arrays. Hence the program uses the correct file in the next iteration.

```
do k=1 to no_of_iterations
   call oc_shift(unit1,OCAD,1,right) !right shift
   call oc_shift(unit1,OCAD,1,left) !left shift
   do l=1, no_of_slabs
      call read_vec(unit1, B, OCAD, i, j, start_vec,
                    end_vec, stride)
      do j=j1, j2
         do i=i1, i2
            A(i,j) = (B(i,j-1) + B(i,j+1) +
                        B(i+1,j) + B(i-1,j))/4
         end do
      end do
      call write_vec(unit2, A, OCAD, i, j, start_vec,
                     end_vec, stride)
   end do
C        exchange file unit numbers instead of explicitly
C        copying files (optimization)
   unit1 ↔ unit2
end do
```

Figure 11: Translated code for the Laplace Equation Solver

```
parameter(n=1024, m=16)
real A(n,n), mult(n), maxNum, row(n)
!HPF$ PROCESSORS P(m)
!HPF$ TEMPLATE D(n)
!HPF$ DISTRIBUTE D(CYCLIC) ONTO P
!HPF$ ALIGN (*,:) with D :: A
!HPF$ OUT_OF_CORE :: A
!HPF$ MEMORY(4096)
do k=1, n
     maxNum = A(k,k)
     mult(k+1:n) = A(k+1:n,k)/maxNum
     A(k+1:n,k) = mult(k+1:n)
     forall (i=k+1:n, j=k+1:n)
          a(i,j) = a(i,j) - mult(i) × a(k,j)
end do
```

Figure 12: LU Decomposition without pivoting

The performance of the Laplace equation solver on the Intel Touchstone Delta is given in Table 1. We compare the performance of the three methods — direct file access, explicit communication and explicit communication with data reuse. The array is distributed in one dimension along columns. We observe that the direct file access method performs the worst because of contention for disks. The best performance is obtained for the explicit communication method with data reuse as it reduces the amount of I/O by reusing data already fetched into memory. If the array is distributed in both dimensions, the performance of the direct file access method is expected to be worse because in this case each processor, except at the boundary, has four neighbors. So, there will be four processors contending for a disk when they try to read the boundary values.

We also consider an out-of-core LU decomposition pro-

Table 1: Performance of Laplace Equation Solver (time in sec. for 10 iterations)

	Array Size: $2K \times 2K$		Array Size: $4K \times 4K$	
	32 Procs	64 Procs	32 Procs	64 Procs
Direct File Access	73.45	79.12	265.2	280.8
Explicit Communication	68.84	75.12	259.2	274.7
Explicit Communication with data reuse	62.11	71.71	253.1	269.1

Table 2: Performance of LU Decomposition ($1K \times 1K$ array)

Processors	16	32	64
Time (sec.)	1256.5	1113.9	1054.5

gram without pivoting. The HPF code for this is given in Figure 12 and the pseudo-code for the translated program is given in Figure 13. The array is distributed cyclically along columns for load balancing purposes. In the translated program, for each column, every processor has to reduce some of the rows in its out-of-core local array. This requires the local array to be fetched from the disk. Hence, it is necessary to perform I/O as many times as the number of columns. The performance of the translated code on the Touchstone Delta for an array of size $1K \times 1K$ is given in Table 2. Since the problem size is small, the I/O costs dominate. We were not able to study the performance for larger arrays because of system constraints.

6 Conclusions

We have described the design of a compiler and associated runtime support to translate out-of-core programs written in a data-parallel language like HPF into node programs for distributed memory machines with explicit communication and parallel I/O. Such a compiler is necessary for compiling large 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 model of out-of-core compilation and the various transformations which the compiler makes. We have also described the runtime support used by the compiler for communication and I/O. The working of the compiler was illustrated using two applications, namely a Laplace Equation solver and LU decomposition. For fetching off-processor data, the Explicit Communication method is found to perform better than the Direct File Access method as it reduces contention for disks. An improvement in performance is also obtained by reusing data already fetched in memory, which reduces the amount of I/O.

All the runtime routines described in this paper have already been implemented. A subset of the compiler has been implemented and a full implementation is in progress. We believe that this paper provides an important first step in techniques for automatically translating out-of-core data parallel programs.

```
do k=1, n
   if (column k lies in mynode) then
C     get the slab containing column k from LAF
     call read_vec(unit, A, OCAD, i1, j1, start_col,
                 end_col, stride)
     do i=k+1, n
        A(i,pivot_col) = A(i,pivot_col)/A(k,pivot_col)
        mult(i) = A(i,pivot_col)
     end do
     Broadcast the multipliers to other processors
     do j=pivot_col+1, end_incore_col
        do i=k+1, n
           A(i,j) = A(i,j) - A(k,j)×A(i,pivot_col)
        end do
     end do
     call write_vec(unit, A, OCAD, i2, j2, start_col,
                 end_col, stride)
C     get remaining slabs from LAF and reduce the rows
     do slab=slab_no+1, no_of_slabs
        call read_vec(unit, A, OCAD, i3, j3, start_col,
                    end_col, stride)
        do j=1, end_incore_col
           do i=k+1, n
              A(i,j) = A(i,j) - A(k,j)×mult(i)
           end do
        end do
        call write_vec(unit, A, OCAD, i3, j3, start_col,
                    end_col, stride)
     end do
   else
       Read the column of multipliers broadcast by the
       owner of column k
C      fetch the columns after k from LAF and reduce
C      appropriate rows
       do slab=slab_no+1, no_of_slabs
          call read_vec(unit, A, OCAD, i4, j4, start_col,
                      end_col, stride)
          do j=1, end_incore_col
             do i=k+1, n
                A(i,j) = A(i,j) - A(k,j)×mult(i)
             end do
          end do
          call write_vec(unit, A, OCAD, i4, j4, start_col,
                      end_col, stride)
       end do
   end if
end do
```

Figure 13: Translated code for LU Decomposition

Acknowledgments

We would like to thank Geoffrey Fox, Ken Kennedy, Chuck Koelbel, Ravi Ponnusamy and Joel Saltz for many enlightening discussions. We also thank our compiler group at Syracuse University for their help with the basic infrastructure of the HPF compiler. 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.

References

[1] Ahmad, I., Bordawekar, R., Bozkus, Z., Choudhary, A., Fox, G., Parasuram, K., Ponnusamy, R., Ranka, S., and Thakur, R. Fortran 90D Intrinsic Functions on Distributed Memory Machines: Implementation and Scalability. In *Proceedings of* 26^{th} *Hawaii International Conference on System Sciences* (January 1993).

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

[3] Bordawekar, R., and Choudhary, A. Language and Compiler Support for Parallel I/O. In *IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems* (Apr. 1994).

[4] Bordawekar, R., Choudhary, A., and del Rosario, J. An Experimental Performance Evaluation of Touchstone Delta Concurrent File System. In *Proceedings of International Conference on Supercomputing, Tokyo, Japan* (July 1993).

[5] Bordawekar, R., del Rosario, J., and Choudhary, A. Design and Evaluation of Primitives for Parallel I/O. In *Proceedings of Supercomputing '93* (November 1993), pp. 452–461.

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

[7] Bozkus, Z., Choudhary, A., Fox, G., Haupt, T., Ranka, S., Thakur, R., and Wang, J. Scalable Libraries for High Performance Fortran. In *Proceedings of Scalable Parallel Libraries Conference* (October 1993), Mississippi State University.

[8] Brezany, P., Gerndt, M., Mehrotra, P., and Zima, H. Concurrent File Operations in a High Performance Fortran. In *Proceedings of Supercomputing '92* (November 1992), pp. 230–238.

[9] Chen, M., and Cowie, J. Prototyping Fortran-90 Compilers for Massively Parallel Machines. In *Proceedings of the Conference on Programming Language Design and Implementation* (1992), pp. 94–105.

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

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

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

[13] Gupta, M. *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, September 1992.

[14] 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.

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

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

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

[18] Ponnusamy, R., Saltz, J., and Choudhary, A. Runtime-Compilation Techniques for Data Partitioning and Communication Schedule Reuse. In *Proceedings of Supercomputing '93* (November 1993), pp. 361–370.

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

[20] Snir, M. Proposal for IO. Posted to HPFF I/O Forum by Marc Snir, July 1992.

[21] Su, E., Palermo, D., and Banerjee, P. Automatic Parallelization of Regular Computations For Distributed Memory Multicomputers in the PARADIGM Compiler. In *Proceedings of International Conference on Parallel Processing* (August 1993), pp. II-30—II-38.

[22] Tseng, C. *An Optimizing Fortran D Compiler for MIMD Distributed Memory Machines.* PhD thesis, Dept. of Computer Science, Rice University, January 1993.

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

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