Compilation of Out-of-Core Data Parallel Programs for Distributed Memory Machines

Rajeev Thakur  Rajesh Bordawekar  Alok Choudhary
Dept. of Electrical and Computer Eng. and
Northeast Parallel Architectures Center
Syracuse University
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 to message passing node programs with explicit parallel I/O. We describe the basic model of the compiler and the various steps involved in the compilation. 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 the example of an out-of-core Laplace equation solver, with performance results on the Intel Touchstone Delta.

1 Introduction

Massively parallel computers 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 [Hig93a], 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 [dRC94]. These figures are expected to increase by orders of magnitude as teraflop machines make their appearance.

The main memory in massively parallel systems is usually 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 [CFPB93, Pie89, BCdR93, DdR92]. del Rosario and Choudhary [dRC94] give an overview of the various issues in high performance I/O.

In this paper, we consider the I/O problem from a language and compiler point of view. Data parallel languages like HPF [Hig93b] and pC++ [BBG+93] 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 large scale scientific computations, 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 [BC93, BGMZ92]. 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.

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 an out-of-core Laplace equation solver as an example to demonstrate the working of the compiler. 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 Out-of-Core Compilation 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. 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) [Hig93b]. 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. 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 a compiler for out-of-core programs 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 [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. 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.

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, 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. 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 array expressions and then describe the compilation method for out-of-core programs and show how it is different. 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 $4 \times 4$ two-dimensional grid.
REAL A(1024,1024), B(1024,1024)
........
!HPF$ PROCESSORS P(4,4)
!HPF$ TEMPLATE T(1024,1024)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO P
!HPF$ ALIGN with T :: A, B
........
FORALL (I=1:N, J=1:N)
    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

3.1 In-core Compilation

This section describes the in-core compilation strategy used in the HPF compiler developed by our group at Syracuse University [BCF+93]. 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 processor memory. Since array B is distributed in block-block fashion over 16 processors, the above assignment requires fetching data from neighboring processors (nearest-neighbor communication). The HPF compiler analyzes the statement and generates a call to a specific collective communication routine. The assignment statement is translated into corresponding DO loops with appropriate calls to the communication routines as shown in Figure 3.
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. The local array has overlap area.

3.2 Out-of-core Compilation

For compiling out-of-core programs, in addition to handling all the issues in 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.

In our model, a local array is stored in the 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.

Some of the issues here are similar to compiler optimizations carried out to gain advantage of processor cache or pipelines. This optimization, commonly known as stripmining [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 compilation, the iteration space of the assignment/forall statement will be sectioned (stripmined) so that each iteration operates on the data that can fit the processor’s memory (ie. the size of ICLA).

3.2.1 Language Support for Out-of-Core Compilation

In order to stripmine the out-of-core assignment statements, the HPF 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 annotated with these directives as shown in Figure 4. The OUT_OF_CORE directive specifies which arrays are out-of-core arrays (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 out-of-core assignment statements. We consider the compilation of the following 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. At any instant, only a constant number of slabs can lie in the local memory of each processor. The same figure shows the overlap area (also called ghost cells) for array B for the above array assignment statement. The overlap area represents the amount of the data to be communicated per processor.

Figure 6 describes two methods for compiling an out-of-core array expression. The two methods, the **Out-of-core Communication Method** and the **In-core Communication Method**, differ in the way data communication is performed.

- **Out-of-core Communication Method**: In this method, the compiler determines what off-processor data will be required for the entire out-of-core array. The shaded region in Figure 5 (C) shows the amount of data to be communicated (**comm.data**) to processor 5 for the above assignment statement. Processor 5 sends/receives data from all of its neighbors. In the **Out-of-core Communication Method**, the entire **comm.data** is communicated in one step (Figure 6(A)) and stored at appropriate places in the local array file. After the data is communicated, the assignment of the local array is performed. This computation is stripmined using the memory size provided by the user. During the computation, each data slab (**in-core local array**) and its **comm.data** is read from and written to the local array file. Each individual
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 Perform local I/O access.
   3.2 Perform Computation.
   3.3 Store Data.
4. If iteration < $k$ then go to 3.

In-core Communication

1. Stripmine Code based on memory size.
2. Repeat $k$ times ($k$ is the stripmine factor).
   2.1 Schedule communication for in-core data (gather).
   2.2 Perform local I/O access.
   2.3 Perform Computation.
   2.4 Schedule communication for in-core data (scatter).
   2.5 Store local Data.
3. If iteration < $k$ then go to 2.
slab computation does not require communication since the necessary \texttt{comm.data} is fetched from the local array file. After the computation on each slab is done, the data slab is written back to the disk.

The \textit{Out-of-core Communication Method} requires extra data storage during program execution. Also in this method, 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 shifts communications in this example. This communication pattern is preserved except that communication also requires disk accesses in addition to data movements. Also, since communication is separated from computation, the compiler can easily perform other code optimizations such as loop fusion.

- \textit{In-core Communication Method}: In this method, the compiler analyzes each slab or (ICLA) instead of the entire out-of-core array (Figure 6(B)). The assignment statement is first strip-mined 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 access the necessary data. Figure 5(D) shows an example of \textit{In-core Communication method}. 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 entire OCLA. Figure 5(D) shows the data to be communicated or \textit{gathered} (\texttt{comm.data}) for each individual slab (each shade represents a different slab). Consider slabs 3 and 4. The amount of data to be communicated and the communication pattern for the two slabs is different. As a result, the structured communication pattern (Figure 5(C)) now becomes an unstructured communication pattern as shown in Figure 5(D).

Since the \texttt{comm.data} will be stored in the ICLA, this method does not require disk accesses to store the \texttt{comm.data}. After the necessary \texttt{comm.data} is fetched, the computation on each slab begins. After the computation is over, the \texttt{comm.data} is again communicated or \textit{scattered}. However in this method, the structured communication pattern gets distributed into several unstructured patterns (for each slab). Hence, the compiler needs to analyze each slab and insert appropriate communication calls to get the necessary data. Optimizing such communication patterns is very difficult. It requires extensive pre-processing and the translated code looks unreadable.
Table 1: Communication Patterns in Assignment Statements for 1-D block distribution

Note: c : compile-time constant; s,d : scalar; f : invertible function; V : indirection array

<table>
<thead>
<tr>
<th>Assignment Statement</th>
<th>Comm. Primitive</th>
<th>Type of Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(i)=A(s)</td>
<td>multicast</td>
<td>structured</td>
</tr>
<tr>
<td>A(i)=A(i+c)</td>
<td>overlap.shift</td>
<td>structured</td>
</tr>
<tr>
<td>A(i)=A(i+s)</td>
<td>temporary.shift</td>
<td>structured</td>
</tr>
<tr>
<td>A(i)=A(s)</td>
<td>transfer</td>
<td>structured</td>
</tr>
<tr>
<td>A(i)=A(f(i))</td>
<td>precomp_read/write</td>
<td>unstructured</td>
</tr>
<tr>
<td>A(i)=A(V(i))</td>
<td>gather/scatter</td>
<td>unstructured</td>
</tr>
</tbody>
</table>

3.3 Compiling Out-of-core Array Assignment Statements

Array assignments involving distributed arrays often result in different communication patterns [CC92, Gup92]. The compiler must recognize the type of communication in the computation in order to generate appropriate runtime calls (communication as well as I/O). Since it is relatively easier to detect and optimize the communication in the Out-of-Core Communication Method, we focus on the Out-of-Core Communication Method for compiling array expressions in this paper.

Detecting the type of communication required in an array assignment statement involves analyzing the relationships among the subscripts of the arrays in the statement [BCF+93, CC92, Gup92]. Table 1 shows some of the communication patterns that arise in assignment statements. In the Out-of-core Communication Method, communication is performed with respect to the entire out-of-core array and since 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.

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 on processors.
- What is the communication pattern in the array expression.
- How the array is stored in the file (e.g. column major/row major).
- How the file is stored on the file system (number of disks, data striping etc).
- How many processors are reading the file.

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, in our model for out-of-core compilation, 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 communication and I/O primitives provided by 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, optimum communication algorithms, communication scheduling, I/O scheduling etc.

4.1 Issues in Runtime Support

Consider the fragment of an HPF program given in Figure 2 which we discussed earlier. 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 example, consider the out-of-core local array on processor P5, shown in Figure 7(B). In this program, 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