Compiler and Runtime Support for Parallel I/O *

Rajesh Bordawekar  Alok Choudhary,
Dept. of Electrical and Computer Engineering & NPAC,
3-211 CST, Syracuse Univ., Syracuse, NY 13244

Abstract

This paper addresses the problem of performing parallel input-output in data parallel computations. Specifically, we present runtime primitives, language support (as extensions to HPF), and initial compiler implementation in the Fortran90D/HPF compiler for distributed memory machines. In this paper we only address the problem of reading and writing data in parallel. We do not address the problem of out-of-core computations. We suggest a set of compiler directives for parallel I/O which can be used in addition to standard HPF directives. The runtime primitives along with the compiler directives provide a common I/O platform for parallel languages like HPF.

1 Introduction

Parallel computers have become the preferred computational instrument of the scientific community due to their immense processing capacities. As scientists expand their models to describe physical phenomena of increasingly large extent, the memory capacity of parallel machines, although immense, become insufficient to contain all the required computational data, and I/O becomes important [2]. A system with limited I/O capacity can severely limit the performance of the entire program - this is known as the I/O bottleneck problem. Some of the key I/O sensitive issues include Input, Debugging, Checkpoint and Output [11]. This problem has become critical, and the need for high I/O bandwidth has become significant enough that most parallel computers such as the Intel iPSC/2 [10], Intel iPSC/860 [18], Intel Touchstone Delta [12, 6], Paragon [13] and the nCUBE [14] now provide some measure of support for parallel I/O.

*This work was sponsored by ARPA under contract # DABT63-91-C-0028 and by an NSF Young Investigator Award CCR-9357840 (Alok Choudhary). The content of the information does not necessarily reflect the position or the 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 the Center for Research on Parallel Computation (CRPC).
In recent years, data distribution is used by several languages to map the arrays on the distributed memory machines. Important among these languages are HPF Fortran [9], Vienna Fortran [4] and Fortran D [7]. Data decomposition often requires data to be accessed (read/written) from the disks in a distributed manner. The inconvenience of having to explicitly specify and control file access for a given data distribution has prompted recent proposals for the inclusion of parallel I/O support primitives into parallel programming languages such as HPF Fortran [9] and Vienna Fortran [4].

However, these languages do not take into account the cost of data distribution. Parallel file systems vary in their level of support for data distribution; some provide no support whatsoever. The cost of distributing data from the disks depends on both the disk mapping and processor mapping. We have proposed an efficient strategy - two phase strategy - to perform optimal I/O [15],[5].

In this paper, we present a set of primitives which we have implemented in a runtime library that makes use of the two-phase access strategy. This runtime system supports a number of parallel file systems, thus providing a common I/O interface for parallel programs. We also propose a set of language directives for parallel I/O, which can be used concurrently with other HPF directives. We also describe an initial implementation of these directives in our Fortran 90D/HPF compiler [8].

1.1 Organization

In this paper, we only consider input and output of data (and not out-of-core computations). The paper has the following organization. In Section 2, we present an overview of current parallel language support for data distribution. Section 3 describes the design of the language directives introduced to provide the necessary information to the runtime I/O library as well as it describes the runtime library. Section 4 presents a sample HPF code that uses the I/O directives. Corresponding compiler generated F77+MP code with appropriate parallel I/O calls is also explained. In Section 5, we present experimental performance results for the runtime primitives. Finally, we summarize in Section 6.

2 Languages Supporting Data Distribution

We concentrate on parallel programs which use the Single Program Multiple Data (SPMD) programming paradigm for MIMD machines. This is the most widely used model for large-scale scientific and engineering applications. In such applications, parallelism is exploited by a decomposition
of the data domain. To achieve load-balance, express locality of access, reduce communication, and to achieve other optimizations, several decompositions and data alignment strategies are often used (e.g., block, cyclic, along rows, columns, etc.) (figure 1). To enable such decompositions to be expressed in a parallel program, several parallel programming languages or language extensions have emerged. These languages provide intrinsics that permit the expression of mappings from the problem domain to the processing domain, allow a user to decompose, distribute and align arrays in the most appropriate fashion for the underlying computation. An example of parallel languages which support data distribution includes Vienna Fortran [4], Fortran I [7] and High Performance Fortran or (HPF) [9, 8].

In order to address the I/O bottleneck problem, these languages propose to provide some support for parallel I/O operations. Important examples include Vienna Fortran [4] and High Performance Fortran [9, 8].

We review the Vienna Fortran and High Performance Fortran proposals for concurrent I/O operations by presenting some simple examples.

2.1 Vienna Fortran

Vienna Fortran (VF) is a data-parallel language which supports the SPMD model of computation. Vienna Fortran provides explicit expressions for data mapping. Using these expressions, VF allows the user to configure the processor map and the corresponding array distributions. The following directive shows an array A(64,64) distributed in Block-Block form over 4 processors arranged in a 2x2 grid. The second statement shows the same array distributed in a row-cyclic fashion over a 2x2 processor grid.

```
  PROCESSORS P(2,2)
  REAL A(64,64) DIST(BLOCK,BLOCK) TO P
  REAL A(64,64) DIST(CYCLIC,:) TO P
```

Vienna Fortran also provides a wide range of facilities for distributing and aligning data arrays. A detailed explanation of the Vienna Fortran can be found in [3]. For concurrent file I/O, the language distinguishes between two types of files: standard Fortran files, and array files for which access is only available via concurrent I/O operations. For example, the array A, above could be written into a file, f, by using the following statement

```
  CWRITE(f, PROCESSORS='P(2,2)',
      DIST=('BLOCK,BLOCK) TO P') A
```

CREAD is used in a similar fashion. Other parallel I/O statements include COPEN, CCLOSE,
Figure 1: Data Distributions in Fortran 90D/HPF

CSKIP. For further detail, the reader is referred to [3].

2.2 High Performance Fortran

High Performance Fortran or HPF was designed to provide language support for a variety of machines including SIMD, MIMD, and vector machines. The HPF is syntactically similar to Fortran D [7], however, HPF incorporates many more features including new run-time library functions. Figure 1 provides a simple illustration of some data distributions (for four processors) that can be easily specified in Fortran D, HPF or VF.

HPF provides an excellent toolset for data mapping. The mapping toolset includes a set of compiler directives. Important directives are ALIGN, REALIGN, DISTRIBUTE, TEMPLATE, PROCESSORS. HPF describes the data-to-processor mapping in two stages: the DISTRIBUTE and ALIGN operations.

DISTRIBUTE is an HPF directive that describes how an array is distributed to the processors. The configuration of the target processor grid is provided by the PROCESSORS directive. TEMPLATE and ALIGN directives allows HPF to distribute several arrays in a similar fashion. Figure 2 shows a sample HPF program. Arrays A and B are distributed in a BLOCK fashion over 4 processors whereas array C is distributed in COLUMN-BLOCK fashion.

HPF has also proposed (but does not currently support) parallel I/O statements. These statements include array mapping and file pointer manipulation statements. Unlike the Vienna Fortran
REAL A(100), B(100), C(100,100)
$CHPF PROCESSORS PRC(4)
$CHPF TEMPLATE DUMMY(100)
$CHPF DISTRIBUTES DUMMY(BLOCK)
$CHPF ALIGN (i) with DUMMY(j): A,B
$CHPF ALIGN (*,i) with DUMMY(i): C

Figure 2: A Sample HPF Program

proposal, HPF proposals include common support for parallel and sequential files.

In the HPF proposal, a FILMAP directive is used to specify a mapping for files. ALIGN and
DISTRIBUTES statements are then used to map FILEMAPs onto nodes. Following example shows
a sample code to map files.

!!HPF$ FILEMAP :: F1(2, 4)
!!HPF$ DISTRIBUTES F1(*, BLOCK) ONTO D1

READs and WRITEs, or PREADs and PWRITEs are subsequently used to access distributed
files. For further information, the reader is referred to [9, 19].

3 Runtime Compiler Support for Parallel I/O

A number of high level programming languages have recently introduced intrinsics that support
parallel I/O through a runtime library. By using these primitives, I/O operation instructions
within applications become portable across various parallel file systems. Further, the primitives
are convenient to use; the instructions for carrying out parallel I/O operations don’t involve much
more than a declaration of the data decomposition mapping and the use of open, close, read, and
write routines.

Yet, these language supported I/O primitives suffer from a serious drawback. Because they use
a direct access mechanism to perform the I/O, the user data distribution mapping remains tightly
linked to the file mapping to disks. Thus, they are susceptible to the same performance fluctuations
and limitations (e.g., unsupported data distributions) that are observed of the parallel file systems.

Motivated by these facts we have implemented a runtime system for parallel I/O. This runtime
system can be accessed by the programmer by using a set of compiler directives. This system will
provide the portability and convenience of language supported I/O primitives. In addition, because
it makes use of the two-phase access strategy (discussed below) to carry out I/O, it effectively
decouples user mappings from the file mappings of the parallel file system, and provides consistently high performance independent of the underlying data decompositions used.

Advantages of Runtime I/O Primitives:

1. The runtime system can be easily ported on various machines which provide parallel file systems.
2. Complex data distributions (Block-Block or Block-Cyclic) are made available to the user.
3. Primitives allow the user to control the data mapping over the disks. This is a significant advantage since the user can vary the number of disks to optimize the data access time.
4. The primitives allow the programmer can change the data distribution on the processors dynamically.
5. The data access time is significantly improved and is made more consistent since the primitives use two-phase access strategy.

3.1 Approach

Our I/O strategy involves a division of the parallel I/O task into two separate phases. In the first phase, we perform the parallel data access using a data distribution, stripe size, and set of reading nodes (possibly a subset of the computational array) which conforms with the distribution of data over the disks.

By employing the two-phase redistribution strategy, the costs inherent in many of the I/O configurations are avoided. The redistribution phase improves performance because it can exploit the higher bandwidths made available by the higher degree of connectivity present within the interconnection network of the computational array.

In the subsection that follows, we discuss the runtime I/O primitives. A brief description of the purpose of each primitive, its functional flow, and syntax is provided followed by some performance results.

3.2 General Description

The runtime primitives library provides a set of simple I/O routines. These include `popen`, `pclose`, `array_map`, `proc_map`, `pread` and `pwrite`. Though the exact syntax of these routines varies from C to Fortran, the basic data structures remain the same. This section presents a brief overview of each primitives. Syntax description and implementation details are presented in [5].
3.2.1 popen

The *popen* primitive concurrently opens a file using a specified number of processors \( P' (P' \in P) \), where \( P \) is the number of processors on which the program is executing (Figure 3). The choice of \( P' \) is important in the systems like Intel Paragon [13] which provide I/O dedicated compute nodes. That is, often the number of processors involved in generating I/O requests must be smaller than the number of processors requiring the data to achieve better performance [6].

The user passes file information to the *popen* primitive which is then stored in a two dimensional array called File Descriptor Array (FDA) using the file unit number as a key. The file information includes file name, file status, file form, access pattern and the number of disks on which the file will be distributed. For example, the following statement

\[
popen(3, 'TEST', 'SEQUENTIAL', 'UNFORMATTED', 'NEW', -1, 4)
\]

opens a file called TEST over 4 processors. The corresponding FDA is shown in table 1. The file will be distributed over all the disks (number of disks = -1). If the number of processors is -1, the file will be opened by default number of processors (P).
Table 1: The File Descriptor Array (FDA): Fortran Version

<table>
<thead>
<tr>
<th>Unit</th>
<th>Access</th>
<th>Form</th>
<th>Status</th>
<th>No. of disks</th>
<th>No. of proc</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>64</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2: The Array Description Table (ADT)

<table>
<thead>
<tr>
<th>Info</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Size</td>
<td>64</td>
<td>64</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Distr. Code</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Block Size</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>nprocs</td>
<td>2</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

3.2.2 pclose

The `pclose` primitive performs concurrent closing of the parallel files. The `pclose` primitive gets the unit number of the file as an input. Using this as a key, the primitive obtains the number of processors (\(P'\)). Using the file unit number, these processors close the file.

3.2.3 array_map

This primitive is semantically similar to the compiler directives in HPF or VF. A Fortran D or HPF compiler would directly extract this information from the distribution directives.

The `array_map` primitive returns an integer called `array descriptor` which will be used by `pread` and `pwrite` routines for acquiring the necessary array information. A table called the Array Description Table or (ADT) is used to store the array information. The user provides the global size of the array, the distribution type, the processor distribution along each dimension and the block size (for CYCLIC distributions).

For example, consider array A(64,64) distributed in BLOCK-BLOCK form over 4 processor arranged in a 2x2 grid. The corresponding Array Description Table is shown in table 2. The value -1 is used to denote don’t-care entries.

3.2.4 proc_map

The `proc_map` primitive is used for mapping the processors from the physical to the logical domain. The `proc_map` initializes the logical processor grid according to the user specifications. The
dimension of the logical processor grid can vary from 1 to 7. The proc.map routine allows two kinds
of mappings, one is the system-defined mapping and the second is the user-defined mapping. The
user has to pass the number of processors in each dimension, the mapping mode and (or) processor
mapping information. proc.map initializes a global data structure called P.INFO array, which is
used by pread and pwrite routines. Using the proc.map primitive the programmer can change
the logical processor configuration during the execution of the program.

3.2.5 pread

The pread primitive reads a distributed array from the corresponding file. The pread primitive
reads the data from the file using \( P' \) processors and distributes the data over \( P \) processors \((P' \in P)\). The
pread primitive uses the unit number as a key to access the file information from the FDA.
The global array information is obtained using the array.descriptor. In general, the runtime system
would use a distribution for intermediate access which performs the best, given a specific file
distribution. For our experiments, we use column-block distribution for I/O access because we
assume that the files are stored in the column-major fashion on the disk arrays. The two-phase
access is used by pread to read the data from the file using \( P' \) processors. Then the data is
redistributed over the \( P \) processors to obtain the target data distribution. Figure 4 shows the
pread algorithm.

3.2.6 pwrite

The pwrite primitive is used to write a distributed array using \( P' \) processors to the file that was
opened (created) by popen. pwrite uses the array.descriptor to get the array information, the
unit number to get the file information and proc.map to obtain the logical grid information. The
runtime primitive will choose a distribution for intermediate access which performs the best for a
specific file distribution. If the processor distribution and the conformal distribution don’t match,
data is first distributed from \( P \) to \( P' \) processors. After the distribution, data is written by \( P' \)
processors using the conforming distribution. Figure 5 shows the pwrite algorithm implemented
in primitive.

4 Language Interface for I/O primitives

High Performance Fortran (HPF) has been designed to be an informal standard programming lan-
guage for a variety of high-performance computers, such as vector machines and massively parallel
MIMD and SIMD multiprocessors [9, 7]. HPF uses data distribution and alignment for mapping and
1. Read the input parameters.
2. Get the global array information using ADT.
3. Obtain the logical mapping of the processors
   \((P)\) participating in array distribution from proc_map.
4. Use the unit number to acquire information
   on the file such as the number of disks, number of processors \((P')\).
5. If the target data distribution is same as the conformal access distribution and
   \(P' = P\) then read the data using the conforming distribution and go to 10.
6. If the two-phase data access is used, then read the data using
   the conforming distribution. The reading is performed by \(P'\) processors.
7. From the global array distribution, calculate the
   data that needs to be communicated.
8. Compute the communication schedule for data redistribution.
9. Distribute the data over \(P\) processors to obtain the
   target data distribution.
10. Stop.

Figure 4: pread Algorithm
1. Read the input parameters.
2. Get the global array information using ADT.
3. Obtain proc_map to obtain the logical mapping of the processors 
   \((P)\) participating in array distribution.
4. Use the unit number to acquire information 
   on the file such as number of disks, number of processors \((P')\).
5. If the target data distribution is same as the conformal access distribution and 
   \(P' = P\) then write the data using the same access distribution and go to 11.
6. If the two-phase data access is used, then redistribute 
   the data over \(P'\) processors using 7,8,9.
7. From the global array information, calculate the 
   data that needs to be communicated.
8. Compute the communication schedule for data distribution.
9. Distribute the data over \(P'\) processors in conforming access fashion.
10. Write the data on the disks using the conforming access distribution.
11. Stop.

Figure 5: pwrite Algorithm
decomposing the computational domain on processors. HPF provides data distribution directives which can be used by the user to effectively declare and map distributed arrays. Important directives include PROCESSORS, TEMPLATE, DISTRIBUTE and ALIGN. Arrays can be distributed in either in BLOCK or CYCLIC form in each dimension.

A large number of scientific problems including many grand challenge problems are I/O intensive [1]. Therefore, in order to achieve good scalability in speed and problem size, support for high performance I/O to perform reads/writes and out-of-core computations is necessary. Currently, HPF does not provide any support for explicit parallel I/O, although some proposals were made [9, 3].

This paper proposes some directives for parallel I/O that can be used in conjunction with other HPF directives. These directives have proven to be useful in the initial implementation of runtime and compiler support for parallel I/O in our HPF compiler [8].

4.1 Parallel I/O Directives

Currently, we are addressing two parallel I/O problems; 1) parallel reads/writes from files, and 2) support for out-of-core computations. A brief description of the directives to support these problems is as follows.

- DISKS: This directive is used for describing the logical mapping of disks over which one or more files may be distributed and/or which are used to distribute scratch files for out-of-core computations. The syntax for this directive is similar to the PROCESSORS directive in HPF.

For example,

**DISKS D(8,8)**

indicates that disks are logically arranged as a two-dimensional logical grid of size 8×8. This directive aids a compiler associate a disk (or a set of disks) with processors for file distributions and out-of-core computations. Many processors are allowed to be associated with one disk and many disks are allowed to be associated with one processor. For example, if processor grid size is 16×16, each disk can be associated to maintain scratch files of a 2×2 processor sub-array.

- FILEPROC: This directive is also similar to the PROCESSORS directive in HPF except that it specifies the processors which really participate in performing I/O. From our earlier studies [6, 21], we observed that the best performance need not necessarily be obtained when all processors performing computations also perform I/O. Thus, this provides the user the flexibility to specify a set of processors to perform I/O. This directive is optional, and if not specified, the default is the number of processors specified in the PROCESSORS directives.
For example,

**FILEPROC FP(2,2)**

specifies that a 2×2 array of processors participates in I/O.

- **FILEDISTR**: This directive declares a file-template and distributes it over the specified number of disks declared in the DISK directive. It also uses the optional FILEPROC parameter. This directive uses names declared in DISKS and FILEPRC as pointers to the corresponding topologies. For example,

  **FILEDISTR F(D,[FP])**

  declares a file-template F which is distributed over D disks, and it associates this template with the processors declared in FP. Thus a file distributed over a set of disks can be associated with different sets of processors by using this directive. For example, when declared together,

  **FILEDISTR F(D, FP1)**
  **FILEDISTR F(D, FP2)**

  permit two different processor configurations to access files on the same set of disks.

- **FILEALIGN**: This directive is similar to the ALIGN directive of HPF. FILEALIGN aligns the list of associated files to the template declared using FILEDISTR directive. However, there is a fundamental difference between ALIGN and FILEALIGN. File may not have a size at declaration time. Thus the same file may be aligned to more than one file-templates as illustrated above. This is quite logical since a file can be opened by two different processor grids. Following example illustrates the FILEALIGN directive.

  **FILEALIGN F :: F1, F2, F3**

- **ASSOCIATE**: This directive describes the relationships between an array’s and the corresponding file’s mapping. That is, the ASSOCIATE directive associates a file-template with the corresponding array template. ASSOCIATE directive has the following form

  **ASSOCIATE :: (file-template, array-template)**

  For example,

  **ASSOCIATE :: (F,A),(),...**
associates the file-template F with the array-template A.

Thus, this directive provides an HPF compiler a list of files to be used for I/O for a set of arrays aligned to the corresponding array template.

- **OUT_OF_CORE**: This directive declares an array as an out-of-core array. Following example declares array A as an out-of-core array.

```
OUT_OF_CORE :: C
```

Figure 6 illustrates the relationships between these directives. File-template declares an abstract template (F), which is distributed over (logical) disks (D) and processors (FP). Note that FP specifies a set of processors which may be a subset of processors declared in the processor directive or a may be a different set (e.g., I/O nodes). Implementation, therefore, may vary from one architecture to another. Files to be accessed in the HPF program are aligned to this template F. Array template, to which arrays C and D are aligned, is distributed over a set of processors (PROCS). As illustrated, the array template A and file template F are associated using the associate directive. Thus a set of files mapped to the a file-template will have a set of arrays (may have different sizes) associated with them. As a result, a compiler can optimize the parallel accesses (e.g., read/write) of distributed arrays from/to the associated files using various strategies (e.g., [21]).

### 4.2 Compiler and Runtime Support for Parallel I/O

Information provided by the compiler directives is used to extract parameters about array and file distributions, which in turn are used in the runtime primitives. In the following we briefly discuss the primitives and how they are used by the compiler.

Input and output operations include reading/writing arrays from/to files. The cost of reading/writing files in parallel may vary tremendously as a function of the distribution of data on compute nodes [6, 21].

We have developed a set of runtime primitives to perform parallel read and write operations [5, 17]. These primitives provide consistent (and high) performance independent of the type of distribution [17]. The parallel I/O primitives include parallel read (pread), parallel write operation (pwrite) and several supporting primitives including popen, pclose, array_map and proc_map.

We have developed compiler support in our HPF compiler [8] to automatically embed the runtime primitives in the compiler code (F77+MP+I/O) using the directives and constructs specified in the source HPF program. Figure 7 illustrates a set of directives and HPF code fragment as well as the corresponding compiled code in F77+MP+I/O. We only show the pertinent code for
Figure 6: File Distribution and Association in HPF
Table 3: Comparing Direct Access with Two-phase Access (64 Processors, 5K*5K Array, time in msec)

<table>
<thead>
<tr>
<th>Distr. Mode</th>
<th>Best Read</th>
<th>Re Distrib.</th>
<th>Total Read</th>
<th>Direct Read</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3324</td>
<td>-</td>
<td>3324</td>
<td>3357</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3324</td>
<td>703</td>
<td>4027</td>
<td>11407</td>
<td>2.83</td>
</tr>
<tr>
<td>3</td>
<td>3324</td>
<td>246</td>
<td>3570</td>
<td>38018</td>
<td>10.65</td>
</tr>
<tr>
<td>4</td>
<td>3324</td>
<td>708</td>
<td>4092</td>
<td>*</td>
<td>&gt; 879</td>
</tr>
</tbody>
</table>

the sake of brevity. Note that in the source HPF code, the user uses very simple constructs called pread and pwrite (figure 7(I)) which are automatically converted into the appropriate calls to the runtime routines. Also, the compiler performs all the necessary transformations.

5 Experimental Results

In this section we present performance results for the runtime primitives when used in conjunction with a variety of data distributions. The tables below contain Best Read, Redistribute, Total Read, and Direct Read times for the four 1-dimensional distributions considered in this paper.

For a given array size, the Best Read time represents the minimum of the read times of the four distributions; the Best Read time is derived from the distribution that most closely conforms to the disk storage distribution for the given file. The Redistribution time is the time it takes to redistribute data from the conforming distribution to the one desired by the application. The Total Read time is the sum of the Best Read and Redistribution times; it denotes the time it takes for the data to be read using the optimal Read access and then be redistributed (two-phase access). The Direct Read time is the time it takes to read the data with the selected distribution using direct access. The last row of each table shows the speedup obtained from using the two-phase access strategy over the direct access strategy. Note that the Block-Block distribution is not supported by CFS, hence table 5 do not present any performance numbers for direct access. (1 denotes Column Block access, 2 column cyclic access, 3 denotes Row Block and 4 represents Row cyclic access.)

Tables 3 and 4 shows access times for 5Kx5K and 10Kx10K arrays, read and distributed over 64 processors. The reduction in cost ranged from 7.4 secs, to over 60 minutes for the 5Kx5K Row-Cyclic case. Note that the variation in Total Read time is again very small (at most a factor of 1.27). However, for all the four types of distribution, the total read time is nearly consistent (of the same order). Thus using the two-phase access we are able to get the data distribution performance which is independent of both the disk distribution and the processor distribution.
(I) HPF Program

real A(64,64),B(64,64),C(64,64)
CHPF$ processors p(1,4)
CHPF$ template R(64,64)
CHPF$ distribute R(block,block)
CHPF$ align (I,J) with R(I,J) :: A,B
CHPF$ disks d(8,8)
CHPF$ fileproc H(1,4)
CHPF$ filedist F(H,d)
CHPF$ filealign F :: F1,F2
CHPF$ associate :: F,R
CHPF$ out_of_core :: A
call popen(3,'F1',SEQUENTIAL,UNFORMATTED,OLD)
call pread (A,3)
call pwrite(A,3)

(II) F77+MP+I/O Program

REAL A (64, 16), B (64, 16), C (64, 64)
INTEGER array_map, map_info(1536)
INTEGER size_info(7),proc_info(7),distr_info(7),block_size(7)
COMMON /INFO/F_INFO, P_INFO, A_INFO
[Initialize the data structures]
CALL popen (3,'F1', 0, 0, 1, disks, procs)
TT3temp = array_map (A, size_info, distr_info, block_size, proc_info, proclist)
CALL pread (A, 64, 16, TT3temp, 3, TT2temp, 1024, 1024)
CALL pwrite (A, 64, 16, TT3temp, 4, TT4temp, 1024, 1024)

Figure 7: (I) shows a sample $HPF^{io}$ program fragment which uses the proposed I/O directives. Disks are logically arranged as an 8×8 logical grid (D) and I/O processors are arranged as 1×4 grid (H). File-template F is distributed over H and D. Files F1 and F2 are aligned to the file-template F. Finally, file-template F is associated with array template A. Array A is declared out_of_core. HPF compiler automatically generates F77+MP+IO code (Shown in (II)).
Table 4: Comparing Direct Access with Two-phase Access (64 Processors, 10K*10K Array, time in msec)

<table>
<thead>
<tr>
<th>Distr. Mode</th>
<th>Best Read</th>
<th>Re Distr.</th>
<th>Total Read</th>
<th>Direct Read</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11395</td>
<td>-</td>
<td>11395</td>
<td>11395</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11395</td>
<td>2478</td>
<td>13873</td>
<td>63400</td>
<td>4.57</td>
</tr>
<tr>
<td>3</td>
<td>11395</td>
<td>1028</td>
<td>11623</td>
<td>78767</td>
<td>6.78</td>
</tr>
<tr>
<td>4</td>
<td>11395</td>
<td>3092</td>
<td>14487</td>
<td>*</td>
<td>&gt; 248</td>
</tr>
</tbody>
</table>

Table 5: Block-Block Distribution over 64 Processors using the Runtime Primitives (time in msec)

<table>
<thead>
<tr>
<th>Size</th>
<th>Best Read</th>
<th>Redistr.</th>
<th>Total Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K*1K</td>
<td>350</td>
<td>82</td>
<td>432</td>
</tr>
<tr>
<td>2K*2K</td>
<td>1100</td>
<td>166</td>
<td>1286</td>
</tr>
<tr>
<td>4K*4K</td>
<td>2462</td>
<td>577</td>
<td>3039</td>
</tr>
</tbody>
</table>

Tables 5 show access times for arrays distributed in the Block-Block fashion 64 processors. Again, note that the read time is consistent with the times obtained for other distributions.

5.1 Discussion

The results above show that for every case, regardless of the desired data distribution, performance is improved to within a factor of 2 of the Best Read Time performance for all distributions. Further, the cost of redistribution is small compared with the Total Read Times. This indicates an effective exploitation of the additional degree of connectivity available within the interconnection network of the computational array. Further, the results also show that by using the runtime primitives, the data can be distributed in Block-Block fashion effectively.

6 Conclusions

The need for high performance parallel I/O has become critical enough that most manufacturer's have provided some support for parallel I/O within their file systems. Recently, several high performance languages have proposed the inclusion of primitives to support parallel I/O.

We proposed a set of runtime primitives which make use of the two-phase access strategy, and showed that the runtime primitives achieve consistent performance across a variety of data distributions, and allows the user to avail of complex data distributions. We also presented a set of
parallel I/O directives which allow the user to access (read/write) any distributed HPF array from the underlying file system. The parallel I/O directives along with the runtime routines make HPF a very suitable language for parallel I/O.

References


