Parallel Array Object I/O Support on Distributed Environments

Jenq Kuen Lee[*] Ing-Kuen Tsaur
Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan
San-Yih Hwang
Department of Information Management, National Sun Yat-Sen University, Taiwan

Abstract

This paper presents a parallel file object environment to support distributed array store on shared-nothing distributed computing environments. Our environment enables programmers to extend the concept of array distributions from memory levels to file levels. It allows parallel I/O that facilitates the distribution of objects in an application. When objects are read and/or written by multiple applications using different distributions, we present a novel scheme to help programmers to select the best data distribution pattern according to minimum amount of remote data movements for the store of array objects on distributed file systems. Our selection scheme, to our best knowledge, is the first work to attempt to optimize the distribution patterns in the secondary storage for HPF-like programs with inter-application cases. This is especially important for a class of problems called multiple disciplinary optimization (MDO) problems. Our testbed is built on an 8-node DEC Farm connected with an ethernet, FDDI, or ATM switch. Our experimental results with scientific applications show that not only our parallel file system can provide aggregate bandwidths, but also our selection scheme effectively reduce the communication traffics for the system.

1 Introduction

High-performance distributed computing environments, which consist of a collection of high-performance machines connected via a high-speed network, can provide the aggregate computing powers necessary for large-scale scientific applications. One of the critical issues in effectively using these systems is the efficient transfer of data to and from secondary storage. In this paper, we investigate the parallel I/O issues on a shared-nothing architecture from the viewpoints of languages and programming environments. Figure 1 shows an architecture of a shared-nothing high-performance distributed computing environment that we are dealing with in this paper. In the architecture, each processor is associated with a local disk, and the information exchange between disks of different processors has to go through an interconnection network.

[*]The correspondence author's e-mail address is jklee@cs.nthu.edu.tw. This paper is submitted to Journal of Parallel and Distributed Computing. A preliminary version of this work appeared in the Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, S.F., Feb. 1995.

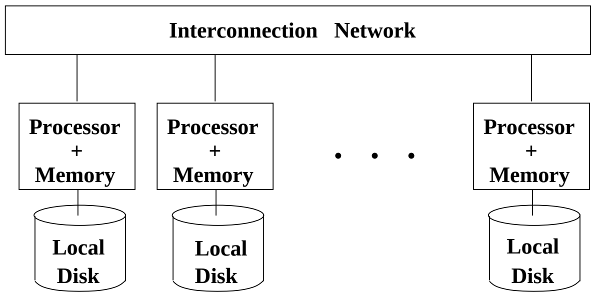

Figure 1: Architecture for Shared-Nothing Environment

Recent research efforts in parallel programming languages have been concentrated on specifying regular dimension-wise data distribution patterns for arrays among parallel machines with distributed memories. Languages supporting the distributed array and data distribution concepts include F90-D[10], HPF[13], PC++[14][3], and up-coming HPC++[22]. The data distribution concepts and SPMD programming model deliver scalable performance on parallel machines for the computational part. It however does not solve the problem with I/O part. The lack of parallel I/O supports creates two problems. First, I/O is executed serially, which results in performance bottlenecks according to Amdahl's law[1]. Second, in parallel languages such as HPF and PC++, the array is distributed among different processors. Due to the lack of support of distribution in the file level, each processor has to read the whole set of data from disk to memory and store them in a temporary buffer, and then assign the data into the distributed array it owns. This creates extra burdens on programmers to keep two set of data structures, and results in extra storage uses and program codes.

In this paper, we extend the concept of array distributions in HPF[13] and pC++[14][21][3] from memory levels to file levels. There are three key elements in this work. First, we support parallel file objects with random access. There is a unique name for each array object in a parallel I/O unit. A parallel I/O unit can be either a file or a pipe in the conventional Unix sense, but now it is a parallel file or pipe and has to be accessed by parallel file operators provided by our libraries. (For example, we now need to use pcat to cat names and contents of all array objects in a parallel I/O unit.) The access of array objects in a parallel I/O unit is no longer by sequential order but instead by the name of objects. The use of a unique name for object in the secondary storage environment helps us to track down the access patterns to a particular object and allows us to do efficient implementation of parallel array object I/O. Second, when objects are read and/or written by multiple applications using different distributions, we provide an interactive environment for programmers to specify the inter-application I/O dependence, represented as a graph. We further provide a novel scheme to select the best data distribution pattern for array store in disk according to the minimum amount of remote data movements calculated from the I/O dependence graphs. This optimization is particularly important for a class of problems called multiple disciplinary optimization problem (MDO) in which various discipline codes interacting with one another to analyze the data[24]. For example, a realistic multidisciplinary optimization of a full aircraft configuration would require a number of discipline codes including aerodynamic analysis, structural design analysis, controls, performance analysis, etc, to interact with each other. Various discipline codes can be executed as a pipeline or executed asynchronously with data file being exchanged at various points in the code. The data distribution pattern of grid data that various discipline codes are using might be different from each other, so it's crucial to select a good disk distribution pattern to optimize total execution time for a full set of applications. Our selection scheme, to our best knowledge, is the first work to attempt to optimize the distribution patterns in the secondary storage for HPF-like programs with multiple disciplinary optimization problems. The third key element of our work is to support an implementation of parallel I/O libraries which can be used effectively on shared nothing storage environments. This set of libraries can support array object with described distribution patterns for both disks and memories. When programs want to read/write array object with different distribution patterns between disk and memory, the system provides a collective communication library to efficiently support the functionalities. In addition, our system is capable of supporting a situation we call "out of configuration" execution, which can be described by an example as follows. Suppose we have an application running with 4 processors. Let the processor set be $P = \{P_0, P_1, P_2, P_3\}$. After the execution, the application uses our parallel write operator to write an data X[:] into the disks of processor set P. At a later time, another application is again running but with processor set $Q = \{P_0, P_1\}$, and is reading the data X[:] from disks of processor set P. We call the parallel I/O operations beyond the processor set of the current SPMD programs as "out of configuration" operations. In such an environment, the set of processors running in a traditional SPMD mode have difficulties in accessing the disks of the processor set which are not in the current running processor set. Our system solves this problem by installing a disk server on each processor, and providing a three phase protocols. This kind of "out of configuration" support is crucial for parallel file systems on shared nothing environments.

Our parallel I/O system is currently incorporated into experimental HPF[19][11] and parallel C++ programming environments[14][15] based on a 8-node DEC Farm system. The parallel I/O system and programming environment is currently being used in a joint work with Power Mechanics Department, Tsing-Hua university to develop scalable methods to model three-dimensional gas turbine combustor model[17][27]. Our experimental results with these applications show that not only our parallel file system can provide aggregate bandwidths, but also our selection scheme effectively reduces the communication traffics for the system.

The remainder of the paper is organized as follows. Section 2 describes the related work. Section 3 presents the parallel I/O operations of our file system for distributed arrays. Section 4 describes a framework to select distribution patterns for arrays stores in the secondary storage. Next, Section 5 gives the design and implementation of parallel I/O libraries. Finally, Section 6 discusses experimental results, and Section 7 concludes this paper.

2 Related Work

The concept of extending the data distribution patterns from memory levels to file levels is first pioneered by Brezany et al.[4]. Their work supports concurrent file operations on Vienna FORTRAN. The system in supporting data distribution for files can be basically classified into two categories: shared-storage system and shared-nothing environment. Brezany's work is based on a shared storage system, and is supported by concurrent file System (CFS) on Intel Paragon machines, while our work in this paper supports a parallel file system on shared nothing environments. The support of data distribution of a parallel file in a shared nothing environment is generally considered more difficult than that in a shared storage system, as more efforts are needed to enforce data sharing on shared-nothing storages. Follow-up work in supporting data distributions on shared-storage systems can also be seen in the work[2][7][8][26] by Choudhary et al. Choudhary's work provides the store of distributed array in the file system with a two phase access strategy scheme on CFS and has an extensive performance analysis of the file system. Both Brezany and Choudhary work are based on Fortran languages. The extension of parallel I/O with a C++ language on shared storage environment is proposed by Gotwals[9]. Gotwals's work provides a parallel stream for a parallel extension of C++ language on Intel Paragons. Gotwals's work is done in parallel with our mechanism[16] which can be used in both HPF and parallel C++ languages. However, in contrast to Brezany, Choudhary, and Gotwals work, the access of array objects in our parallel I/O unit is not by sequential order but instead by the name of objects. The use of a unique name for an object in the secondary storage environment helps us to track down the access patterns to a particular object among a set of applications and allow us to do optimizations of parallel array object I/O.

Related work in shared-nothing storage system includes IBM's Vesta[5], a scalable parallel I/O system at Argonne National Laboratory[23], and PIOUS[25] systems. Our work also falls into the category of shared nothing environments. IBM Vesta puts all the disks on the IO nodes so that their system is nonsymmetrical compared to our system. Vesta has two layer partitions. Programmers not only have the global view of the file but also specify these two layer partition patterns. If programmer uses these two layer partition properly, system can get better performance. Argonne's system is also built based on IBM SP2 system. This system provides a communication library helps applications to manage hundreds of I/O streams and to checkpoint and restart with different number of processors. PIOUS is a parallel file system provide by Steven A. Moyer and V. S. Sunderam. This system consists of a set of data servers, a service coordinator and library routines. PIOUS directly works compatibly with PVM environments, and does not support to the higher level of file distributions of HPF programs.

Our system supports the data distribution of objects in the file level on shared nothing environment, and work compatible with experimental HPF and parallel C++ compilers. In addition, we address the issues of multiple disciplinary optimization problem. When objects are read and/or written by multiple applications using different distributions, we provide a novel scheme to select the best data distribution pattern for the array store. The pattern is chosen with the minimum amount of remote data movements calculated from the I/O dependence graphs. This optimization is particularly important for a class of problems called multiple disciplinary optimization problem (MDO) in which various discipline codes interacting with each other to analyze the data files[24]. None of the existing work attempts to optimize the distribution patterns in the secondary storage for HPF-like programs with multiple disciplinary optimization problems.

3 Parallel Object I/O for Distributed Array

In this section, we present parallel I/O operations for distributed array objects. We will deal with the selection of I/O distributions for inter-application cases in the next section.

Function Name	Functionality
PARALLEL-OPEN	Open a parallel I/O unit for reading or writing
PARALLEL-READ	Read from the named parallel I/O unit
PARALLEL-WRITE	Write from the named parallel I/O unit
PARALLEL-PIPE	Create an inter-application communication channel
PARALLEL-CLOSE	Close a I/O unit of the current application
PCAT	Cat the names and contents of array objects in a parallel I/O unit.
PARALLEL-RESHAPE	Explicitly reshape the array distributions in a parallel file

Table 1: Parallel Operators and their Functionalities for a Parallel I/O Unit

A parallel write is demonstrated in the following extended HPF programs,

```
        REAL A(64,128),B(32,16),C(24,24),D(36,36)
!HPF$   PROCESSORS MESH(4,4)
!HPF$   DISTRIBUTE (BLOCK,*) ONTO MESH:: A,B,C,D
!HPF$   PARALLEL-FILE-OBJECT A,B,C
        PARALLEL-WRITE (u,'A','B') A,B
```

A read operation to one or more distributed arrays is specified by a statement of the following form:

```
!HPF$   PARALLEL-FILE-OBJECT A,B,C
        PARALLEL-READ (u,'A','B','C') A,B,C
```

where u is a parallel I/O unit, and A,B,C,D are array identifiers. In a parallel I/O unit, an array declared as a *PARALLEL-FILE-OBJECT* can be uniquely identified and accessed by its name. The name tag is given in the parenthesis following the parallel I/O unit in an I/O statment. A parallel I/O unit can be either a parallel file or a pipe. If it is a pipe, the reader of the pipe has to use the unique name to access the object. If the named object in the pipe is not stored yet, the reader will be pending until the object is stored. Similarly, the access of array objects in a parallel I/O file is no longer by sequential order but instead by the name of objects. Only those arrays declared as "PARALLEL-FILE-OBJECT" can be used in operations of a parallel I/O unit. Table 1 gives a complete list of operators for a parallel file. These parallel I/O operators can be extended to parallel C++ language constructs in a similar way. The code segments shown later in Section 6 with our experiments will demonstrate the use of parallel I/O operators in a parallel C++ program.

4 Select Distributions on Inter-Applications Environments

4.1 I/O Models for Inter-Application Environments

Suppose we have three applications, F1, F2, F3, and each application accesses a distributed array A and is reading or writing A from/to the same I/O unit using the parallel I/O operators. Suppose A is of

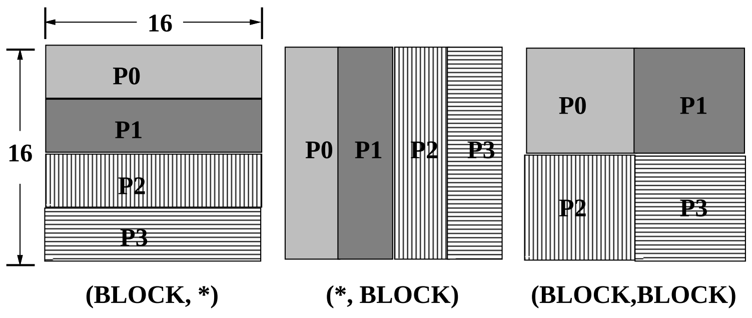

Figure 2: Three Different Data Distribution Patterns of Array Objects Using READ/WRITE in Different Applications.

size 16×16, and is distributed as (BLOCK, *), (*, BLOCK), and (BLOCK, BLOCK) respectively in the memory in applications F1, F2, and F3. Figure 2 shows these three different distributions with 4 processors. The question is with what distribution should the array A reside in secondary storage. Our system helps programmers to select the best data distribution pattern for the store of array objects on distributed file systems according to the minimum amount of remote data movements. If (BLOCK, *) is chosen as the distribution scheme, application F1 incurs no remote data movements. However, there will be 192 remote data movements for (*,BLOCK) distribution in application F2, and 128 for (BLOCK,BLOCK) distribution in F3, and totally there are 320 data movements from remote disks. Similarly, there will be 384 remote data movements for the store of A in secondary storage if we select (*,BLOCK) as the distribution pattern, and 320 remote data movements for (BLOCK, BLOCK) distribution.

Figure 3 shows the programming environment that we have for the selection of I/O distribution patterns for multiple applications. When a set of applications that access an array object are ready to be executed, a programmer first goes through an interactive I/O control panel (ICP), where they can describe the inter-application I/O dependence graphs. These dependence graphs are then solved by a solver to select the best data distribution schemes for the store of the array object on secondary storage. These distributions schemes are then fed to a compiler to generate efficient I/O codes. After the execution, profiling information can be sent back to ICP to update the inter-application I/O dependence graphs if necessary.

Currently, the solver is able to select the best data distribution schemes for two types of inter-application I/O dependence graphs:

- *Independent Event Model*

  Each application is annotated with a frequency to denote how frequently the application is invoked. Also each application is associated with a distribution pattern in which the distributed array object is distributed among processor memories. The inter-application I/O dependence graph in Figure 4 is an example of the Independent Event Model. The read/write access patterns to an file object are independent events in this model.

- *Multiple-Stage Precedence Model*

  The inter-application I/O dependence graph in Figure 5 is an example of the multiple-stage precedence

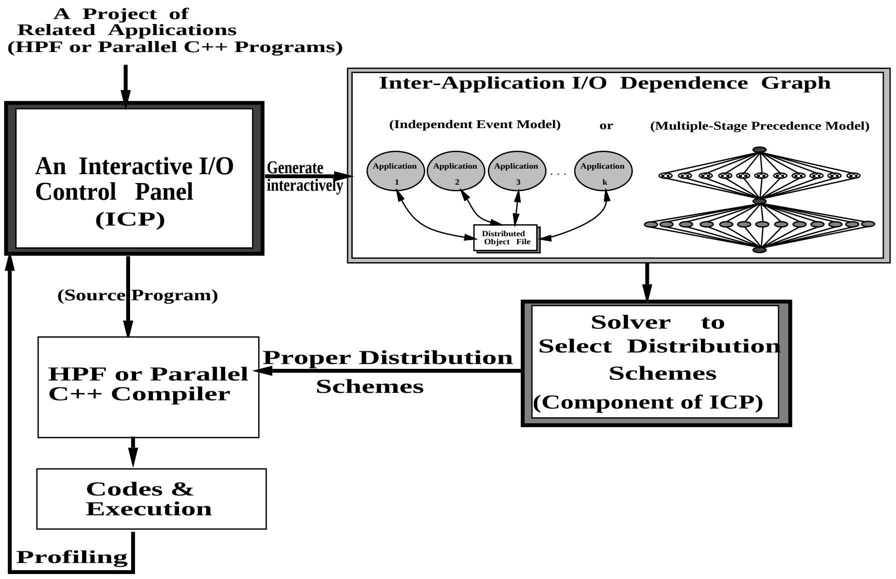
A Project of Related Applications (HPF or Parallel C++ Programs)
An Interactive I/O Control Panel (ICP)
Generate interactively
Inter-Application I/O Dependence Graph
(Independent Event Model)
or
(Multiple-Stage Precedence Model)
Application 1
Application 2
Application 3
. . .
Application k
Distributed Object File
(Source Program)
HPF or Parallel C++ Compiler
Proper Distribution Schemes
Solver to Select Distribution Schemes (Component of ICP)
Codes & Execution
Profiling

Figure 3: System Flow for Interactive Selection of Distributions of Array File Objects

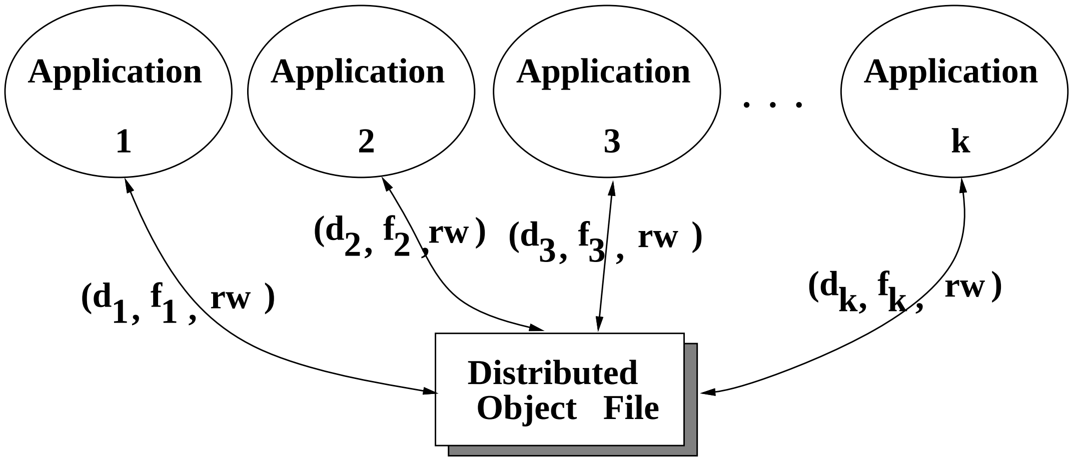

Figure 4: Independent Event Model for Inter-Application Stores

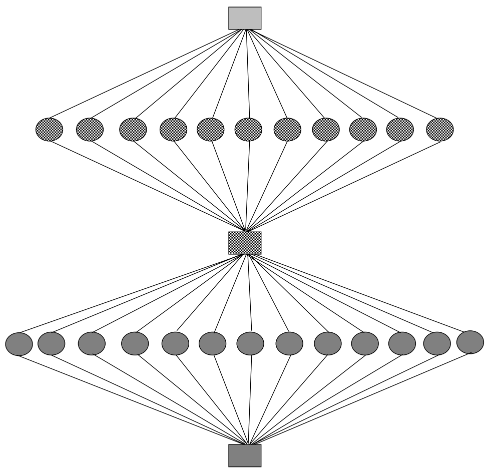

Figure 5: Multi-stage Model for Inter-Application Stores

model. Each round node in the graph represents an application. The rectangular node represents the data distribution of an object at each stage. The multiple stage model represents a temporal relationship, and at each time step, there is an set of applications accessing an array object in the file level. The multi-stage graph models the I/O graphs at each stage also as an independent event model. Each application at each stage is associated with a distribution pattern in which the distributed array object is distributed among processor memories. Multiple-stage precedence graph represents a static precedence relation between applications. It is useful for applications running for a particular order of procedures.

4.2 Solvers

A solver selects the best data distribution scheme for the store of the array object on secondary storage according to the specified I/O graphs. Here, we first introduce the solver for the Independent Event Model. For simplicity, we first assume that there is only one parallel I/O unit in the system. Our scheme works under the assumption that the processor allocation scheme is statically decided, and our goal is to find a distribution τ such that

$$Minimize : \Sigma_{i=1}^{m} \ f_i \times \Delta(\tau, D_i) \quad \forall \tau \in D^d$$

where f_i is the execution frequency of ith Application, D_i is the memory distribution pattern of the object in the ith application, m is the number of applications, and

$$\tau = (\tau_1, \tau_2, ..., \tau_d) \in D^d.$$

where D is the domain of all possible distribution schemes, d is the number of processor dimension, and $\Delta(\tau, D_i)$ is the amount of remote data movements for a given distribution τ under the condition that the application is using D_i at the memory distribution scheme, and it can be calculated by

$$\Delta(\tau, D_i) = \Sigma_{p=1}^{\#P} \quad NumOfRemoteRef_p(\tau, D_i).$$

The function *NumOfRemoteRef*$_p$ calculates the number of remote references for any given processor p. The function *NumOfRemoteRef*$_p$ can be calculated as follows. Suppose the distribution of an array on a dimension is BLOCK or CYCLIC, and then the element in processor p can be described as $begin_1 : end_1 : stride_1$ and similarly the elements resided in the disk of node p can also be described as $begin_2 : end_2 : stride_2$. The number of local references is equivalent to the number of non-negative integer pairs $(x,\, y)$ such that

$$begin_1 + stride_1 * x = begin_2 + stride_2 * y, \quad where\ begin_1 + stride_1 * x \leq\ MIN(end_1, end_2).$$

The equation can be converted into a form of diophantine equation as

$$a * x + b * y = c, \quad where\ a = stride_1, b = -stride_2, c' = begin_2 - begin_1$$

This equation can be solved with standard diophantine equation solver[28] to get $x = uc/g + t * b/g, y = vc/g - t * a/y$, where $g = gcd(a, b) = a * u + b * v$. The variable u and v can be calculated by an extended form of Euclid's algorithm[6] as follows

```
Extended-Euclid(a,b)
    if $b = 0$ then $return(a, 1, 0)$
    $(d', x', y')$ ←Extended-Euclid$(b, a\ mod\ b)$
    $(d, x, y) \leftarrow (d', y', x' - \lfloor a/b \rfloor \times y')$
    $return(d, x, y)$
END-Euclid
```

Procedure Extended-Ecluid takes as input an arbitrary pair of integers and return a triple of the form (g, u, v), where $g = a \times u + b \times v$. The complexity of the Extended-Ecluid algorithm is the recursive depth of the algorithm, which is O(log MIN(a,b)). a and b are strides and will be 1 in the case of BLOCK distribution and equal to the number of processor in the case of CYCLIC distribution.

Thus $\Delta(\tau, D_i)$ can be computed in $O(p \times Log(p))$ time, where p is the number of processors in the system. The above complexity is certainly true for the one-dimension case. For multiple dimensional array, we need further explanation below.

Suppose we have a d dimensional array, and the processor array in the memory and disk is $(p_1, p_2, ..., p_d)$ and $(p_1^{'}, p_2^{'}, ..., p_d^{'})$, respectively.

$$p = p_1 * p_2 * ... * p_d$$

$$p = p_1^{'} * p_2^{'} * ... * p_d^{'}$$

Assume the complexity is h, then

$$h < \Sigma\ p_i * log(Max(p_i, p_i^{'}))$$

Therefore,

$$\begin{aligned} h < \Sigma\, p * log(Max(p_i, p_i^{'})) &= p * \Sigma\, log(Max(p_i, p_i^{'})) \\ &= p * log(\Pi\ Max(p_i, p_i^{'})) \\ &< p * log(\Pi\ p_i * p_i^{'}) \\ &< p * log(p^2) \\ &= 2 * p * log(p) \end{aligned}$$

Thus $\Delta(\tau, D_i)$ can be computed in $O(p \times Log(p))$ time for arbitrary dimensional arrays.

In the case that the distribution is a Block-Cyclic distributions, we need extra efforts to calculate the function $NumOfRemoteRef_p$. Suppose that we have memory distribution $B(N_1)$ and disk partition as $B(N_2)$, we will try to find x, y, i, j, such that

$$begin_1 + stride_1 * x + i = begin_2 + stride_2 * y + j, \quad where\ 0 \leq i < N_1, 0 \leq j < N_2$$

Therefore,

$$begin_1 + stride_1 * x = begin_2 + stride_2 * y + (j - i), \quad where\ 0 \leq i < N_1, 0 \leq j < N_2$$

We need to solve the diophantine equation $MAX(N_1, N_2)$ times. Assume N_1 is bigger than N_2, $\Delta(\tau, D_i)$ can be computed in O(p $\times$ N_1 $\times$ Log(p)) time, where p is the number of processors in the system.

Solver for Multiple-Stage Precedence Model

Now let's develop the solver for Multiple-Stage Precedence Model. In this model, we have an adaptive distribution scheme for then array object in different stages. Our goal is to find an arbitrary sequence of $(x_1, x_2, ..., x_k)$, to

$$Minimize\ \ \Sigma_{j=1}^{k}\ \Omega(j, x_j) + \delta(x_j, x_{j+1})$$

where $\Omega(j, x_j)$ is the amount of remote data movement at stage j, given the distribution pattern x_j, k is the number of stages, $x_i\ \in D^d, 1 \leq i \leq k$, and δ is a reshape cost from one shape to another shape.

$$\Omega(j, \tau) = \Sigma_{i=1}^{m}\ f_{i,j} \times \Delta(\tau, D_{i,j}) \quad \forall \tau \in D^d$$

where $f_{i,j}$ is the execution frequency of ith application in the jth stage, $D_{i,j}$ is the distribution pattern of the object in the ith application of the jth stage, and τ and Δ can be calculated in the same way as those in the Independent Model.

4.3 Extended Frameworks for Solvers

Figure 6 shows a standard three level data mapping models in the processing of an array object in a HPF language. Our work in the previous section can only work with distribution directives, and here we extend our framework to work with alignments.

Assume that array A is aligned with B, and B is distributed by "BLOCK" distribution as shown in the program code segment below.

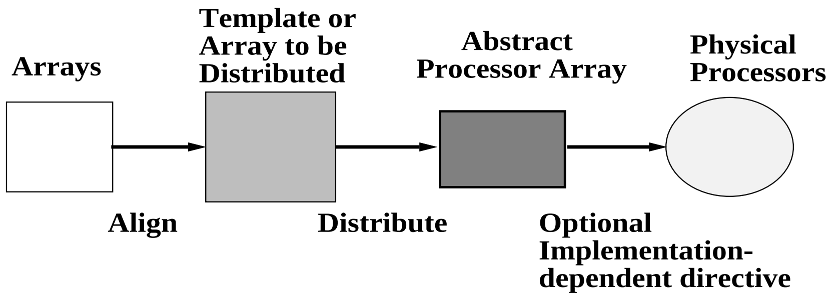
Arrays
Template or Array to be Distributed
Abstract Processor Array
Physical Processors
Align
Distribute
Optional Implementation-dependent directive

Figure 6: Three Level Data Mapping Model of HPF Language

```
        REAL  A(N)
!HPF$ ALIGN A(i)  with B(a*i+ d)
```

We represent the index set of the elements of array A resided in processor *pid* as

$$Index_A(pid) = \{i \mid a_i \text{ is a member of} A \text{ and } a_i \text{ is stored on processor } pid\}$$

$Index_A(pid)$ can be described by a triplet, $[begin : end : stride]$, when the array B aligned by A is distributed by "Block" or "Cyclic" distribution. For example, assume that the array B is distributed by "Block" distribution, we have

$$a * i + d = pid * B_S + j, \;\; 0 \leq i < N, \; 0 \leq j < B_s$$

where B_s is the block size of elements on each processor. The above equation can be transformed into an diophantine equation as shown below.

$$-a * i + j = d - pid * B_S, \;\; 0 \leq i < N, \; 0 \leq j < B_s$$

By solving the diophantine equation, we get

$$i = u * c/g + b/g * t, \;\; 0 \leq i < N,$$

where $c = d - pid * B_s, \; g = gcd(-a, 1), \; (-a) * u + 1 * v = g, \; b = 1.$

Therefore we have the triplet $[begin : end : stride]$ as below:

$$begin = u * c/g + b/g * ceil(u * c/g), \; stride = 1, \; end = u * c/g + b/g * floor((N * g - u * c)/b)$$

Similarly, when array B is distributed by "Cyclic" distribution the $Index_A(pid)$ can be described by a triplet, $[begin : end : stride]$. We can get the triplet by solving the diophantine equation below

$$a * i + b = pid + j * NPROC, \;\; 0 \leq i < N, \; 0 \leq j < N/NPROC$$

Since the element index in the processor *pid* can be described as a triplet, we can calculate the function $NumOfRemoteRef_p$ by the method used in Section 4.2. Suppose the distribution of an array on a dimension is BLOCK or CYCLIC, and the element in processor p can be described as $begin_1 : end_1 : stride_1$ and similarly the elements resided in the disk of node p can also be described as $begin_2 : end_2 : stride_2$. The number of local references is equivalent to the number of non-negative integer pairs (x, y) such that

$$begin_1 + stride_1 * x = begin_2 + stride_2 * y, \quad where\ begin_1 + stride_1 * x \leq\ MIN(end_1, end_2).$$

In the case that the distribution is a Block-Cyclic distribution, we need extra efforts to calculate the function $NumOfRemoteRef_p$. Suppose that array B is distributed by block-cyclic distribution, say B(k_1). Then $Index_A(pid)$ can no longer be described a a triplet, but instead it can be described by k_1 triplets,

$$c * i + d = pid + j * NPROC + offset$$

$$0 \leq offset \leq k_1$$

Suppose that we have memory distribution $B(k_1)$ and disk partition as $B(k_2)$. Then the element in processor p can be described as k_1 triplet, $begin_i : end_i : stride_i, 0 \leq i < k_1$ and similarly the elements resided in the disk of node p can also be described by k_2 triplet, $begin'_j : end_j^{'} : stride'_j,\ 0 \leq j < k_2$. The intersection of the elements can be calculated by solving $k_1 * k_2$ diophantine equations. The number of local references is equivalent to the number of non-negative integer pairs (x, y) such that

$$begin_i + stride_i * x = begin_j^{'} + stride_j^{'} * y$$

$$where\ begin_i + stride_i * x \leq\ MIN(end_i, end_j^{'}),\ 0 \leq i < k_1, 0 \leq j < k_2.$$

The complexity in finding $\Delta(\tau, D_i)$ in this case is O(p $\times$ k_1 $\times$ k_2 $\times$ Log p).

Finally, our scheme works well with or without the optional mapping between abstract processor array and physical array described in Figure 6. If the abstract processor array is the physical array, our scheme can be applied immediately. If the optional mapping exists, we only need to do a mapping between the physical processor number and the abstract processor number when calculating the remote reference numbers. All of our frameworks can be applied directly.

5 Design of Parallel I/O Libraries

5.1 File Structure Design

In our parallel file system, each parallel file is distributed among disks of machines. In the disk of each machine, we have a parallel file structure and a data set of distributed array objects for each file. The parallel file structure records the information of a parallel file. The information includes the type of the file, the number of objects in the file, the distribution configuration for each object in the file, the disk address to denote where the object is located, the name tag for each object, the processor configuration, and the size of each object. These information can be used to locate the positions of the data set of objects among processors and disks.

When users want to create a parallel file in our system, the system creates an ordinary unix file on each node named as user specified parallel file name and appended node number as postfix. Then each corresponding file of each processor will have only partial data set of distributed array objects. This file structure is used as the basis of our parallel file system.

5.2 Collective Communication Library

In the case that we want to read an array object from disks, the system first pack data from local disks into memory buffers. If the distribution of the object in the memory is the same as that in the disk, each processor read the data from the memory buffers with SPMD mode. On the other hand, if the distribution of the object in the memory is different from that in the disk, our system provides a collective communication library to exchange data into its correct positions. The collective communication library will compute the sending set and receiving set. Sending set is the set of data that that each local processor will send to other processors, and receiving set is the set of data that each local processor will receive from other processors. When the collective communication is done, data is then placed into the correct memory addresses from the memory buffers. A parallel write operation can be done similarly.

The sending set and receiving set in our collective communication library for parallel I/O operations can be modeled by extending the concepts in the data movements in the memory level on distributed memory environments[12]. We will explain our revision as follows. Suppose A is an array, and p and q denote processor numbers. The essential set operations involved in a collective communication library implementation are described as follows.

$local_A(p)$ which represents the set of data owned by processor p is defined as follows.

$$local_A(p) = \{a \mid a\ is\ a\ member\ of A\ and\ a\ is\ stored\ on\ p\}$$

$send\_set(p, q)$ that represents the set of data processor p will send to processor q is defined as follows.

$$send\_set_A(p, q) = \{a \mid a\ is\ a\ member\ of\ A,\ and\ must\ be\ sent\ from\ p\ to\ q\}$$

$recv\_set_A(p, q)$ which represents the set of data processor p will receive from processor q is defined as follows.

$$recv\_set(p, q) = \{a \mid a\ is\ a\ member\ of\ A,\ and\ p\ must\ receive\ a\ from\ q\}$$

We then have the following two basic properties.

Lemma 1 Assume that the owner computing rule is used, and let A be the array object to be written from memory to a file, and denoted by $Disk_A[:] = Memory_A[:]$. Then

$$recv\_set(p, q) = local_{Memory_A}(q) \cap local_{Disk_A}(p)$$

$$send\_set(p, q) = local_{Memory_A}(p) \cap local_{Disk_A}(q)$$

Lemma 2 Assume that the owner computing rule is used, and let A be the array object to be read from a file to the memory, and denoted by $Memory_A[:] = Disk_A[:]$. Then

$$recv\_set(p,q) = local_{Disk_A}(q) \cap local_{Memory_A}(p)$$

$$send\_set(p,q) = local_{Disk_A}(p) \cap local_{Memory_A}(q)$$

The local functions for "Block" and "Cyclic" distributions are

$$local_{\mathbf{block}}(p) = \{i \mid (p-1) * \lceil N/P \rceil + 1 \le i \le p * \lceil N/P \rceil\}$$

$$local_{\mathbf{cyclic}}(p) = \{i \mid i \equiv (p \; mod \; \#P)\}$$

The above form can be further simplified as a triplet. Let's assume that A(N) is with Block distribution and distributed on #P processors. Then we can get

$$local\_set_A(pid) = (L : U : S)$$

where $L = B * \#P,\ U = B * (\#P + 1) - 1,\ B = N/\#P,\ and\ S = 1.$

Similarly, if A(N) is with Cyclic Distribution and distributed on #P processors, the elements on processor *pid* can also be described by a triplet.

We will give an example in calculating *send_set* and *recv_set* below:

Example 1 Assume that the owner computing rule is used, and let A(10) be the array object to be read from file to memory, and denoted by $Memory_A[:] = Disk_A[:]$. Let $Memory_A$ is distributed by block distribution and $Disk_A$ is distributed by cyclic distribution. Furthermore, we assume that we have only two processors in a system. Then

$$\begin{aligned} recv\_set(0,1) &= local_{Disk_A}(1) \cap local_{Memory_A}(0) \\ &= (1:9:2) \cap (0:4) \\ &= (1:3:2) \\ send\_set(0,1) &= local_{Disk_A}(0) \cap local_{Memory_A}(1) \\ &= (0:8:2) \cap (5:9) \\ &= (6:8:2) \end{aligned}$$

5.3 Out of Configuration Support

Suppose we have application A running with 4 processors. Let the processor set be $P = \{P_0, P_1, P_2, P_3\}$. After the execution, application A uses our parallel write operator to write an data X[:] into the disks of processor set P. Later, application B is running but with processor set $Q = \{P_0, P_1\}$, and is reading the data X[:] from disks of processor set P. Figure 7 illustrates the data movements of such an environment. We call the parallel I/O operations beyond the processor set of the current SPMD programs as "out of configuration support". In such an environment, the processors running in traditional SPMD mode have difficulties in accessing the disks of the processor set outside the current running processor set. Our system

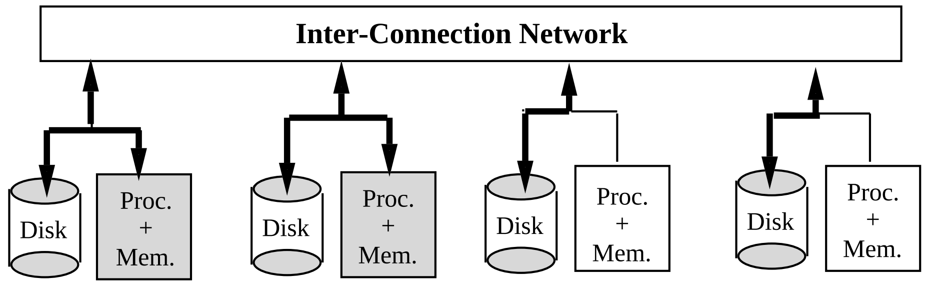

Figure 7: Processors Read/Write Objects with Out of Configuration Cases

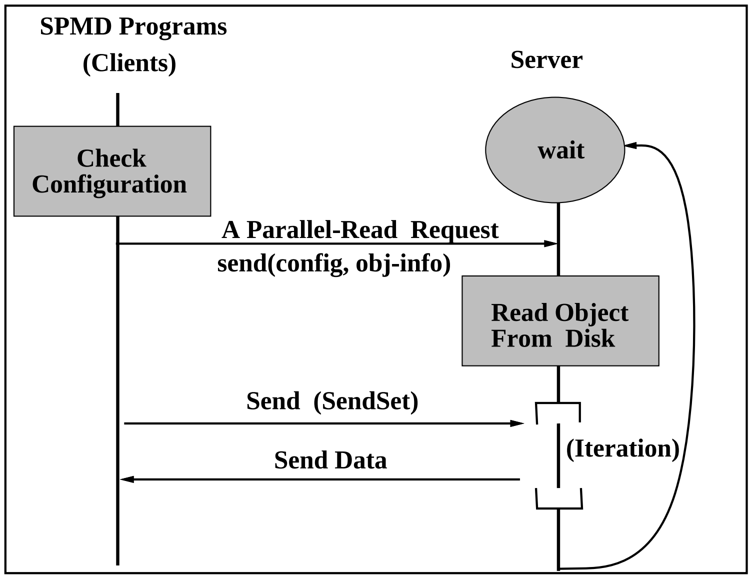

Figure 8: Protocol for a Parallel Read Operation with Interactions between SPMD Programs and a File Server

solves this problem by installing a disk server on each processor and providing a protocol to support out of configuration operations.

Figure 8 describes our three phase protocols, OCP (Out of Configuration Protocols) between the program and the server when a parallel read operation is issued. The program first check if a "out of configuration" parallel read operation is invoked. If an out of configuration read operation does happen, processor 0 then sends a parallel read request to the server of each processor which is not in the current configuration, but possesses data. This daemon in the server site first waits and listens to request. Upon receiving the parallel read request, it does handshake with processor 0 to get the configuration and object information. In the second phase, the server reads the local collection of the array object from the disk, and waits to receive the sending set and receiving set information from each process running SPMD program. Finally after the server receives the sending set, it then sends data to each of the SPMD program according to the sending set it receives.

Similarly, when a parallel write operation is issued, a program first checks if a "out of configuration" parallel write operation is invoked. If an out of configuration write operation does get invoked, processor 0 then sends a parallel write request to the server of each processor which is outside the current configuration, but the data will be written to. This daemon in the server site first waits and listens to request. Upon it receives the parallel write request, one of the SPMD processor continues to send the configuration and object information to each of the servers. In the second phase, the server receives receiving set and the data from each of the SPMD processors. Finally in the third phase, it stores the data into local disk according to the receiving set.

6 Experimental Results

Our parallel file system is currently being incorporated with an experimental parallel C++ and High-Performance Fortran compiler on a 8-node DEC Farm. Figure 3 in Section 4.1 showed the full system view for the parallel programming environments. This system is also used for the development of computational fluid dynamic applications in a joint effort with scientists in the Power Mechanics Department, Tsing-Hua university[17][27]. In the remainder of this section, we will present three sets of experiments done on our parallel file systems.

Experiments with Primitive Operations

The first set of our experiments is conducted to evaluate the basic functionalities of our parallel I/O operators. Table 2 shows the time spent and the speedup figures for parallel write operations. The experiment is done with the memory and disk having the same distribution patterns. The size of the array in the experiment is 128 by 128 with each element of the array having 64 bytes. We experiment with basic distribution patterns including (Block,Block), (Block,Cyclic), (Cyclic,Cyclic), (Cyclic,Block), and (Block,*). All of them exhibit close to linearly speedup and deliver aggregate bandwidth as processor number grows. The time listed is measured by running the parallel read and write operations over 100 times. We observe their behavior similar to each other in spite of different distribution patterns used. Similarly, parallel-read operations also exhibits similar performance speedup in our experiment.

The experiment listed above is done when the data distribution pattern of the object in the memory is the same as the distribution in the disk. Our system also supports the parallel I/O operator when the memory object has different distribution from the disk object. In the following, we conduct an experiment in our system with such cases. In our experiment, we have an array object stored in the disk with (Block, Block) distribution, and has to be read from the disk into the memory with (Block, Cyclic) distribution pattern. The array sizes listed used are 256 by 128, 128 by 128, and 128 by 64, respectively. The size of each element is again 64 bytes. The disk data is first fetched into the local processor, and then our collective communication library described in Section 5 is invoked to exchange data elements between different processors. The collective communication time is bound by the total number of remote references and the network bandwidths. Table 3 shows the communication time spent with our collective communication library in such cases. Out system works on an 8 node DEC Farm with both Ethernet and FDDI connections. Our

Pattern	W(B,B),R(B,B)		W(C,C),R(C,C)		W(B,C),R(B,C)		W(C,B),R(C,B)		W(B,*),R(B,*)	
NPROC	time	speedup	time	speedup	time	speedup	time	speedup	time	speedup
1 node	40.74	1.0	41.16	1.0	42.22	1.0	41.37	1.0	40.90	1.0
2 nodes	23.12	1.76	23.09	1.78	23.04	1.83	22.76	1.82	23.10	1.77
4 nodes	13.93	2.92	13.72	2.99	13.74	3.07	14.12	2.93	13.86	2.95
8 nodes	6.72	6.06	6.64	6.19	6.72	6.28	6.82	6.06	6.80	6.01

Table 2: Time (Seconds) Spent When Parallel-Write with the Same Data Distribution Pattern.

Network/ ArraySize	256x128	128x128	128x64
EtherNet	171.7	93.9	62.2
FDDI	52.8	29.7	24.8

Table 3: Communication Time with Data Moved From (Block,Block) into (Block, Cyclic) Distribution on an 8 Node DEC Farm

collective communication library is improved with a factor of 3 when we move from Ethernet to FDDI network. In these experiments, our parallel file system can not only provide aggregate bandwidths when the data distribution pattern of an object in the memory is the same as the distribution in the disk, but also provide efficient supports of a collective communication library on high-speed networks when the memory object is associated with a different distribution from that of the disk object.

Scientific Applications Using Parallel I/O

In our second set of experiments, our parallel file system is being used for actual software development with parallel computational fluid dynamic applications to reduce I/O bottlenecks. This is a joint work with the Power Mechanics Department, Tsing-Hua university. Currently, two fluid dynamic codes are employing the parallel I/O operations to parallelize the original serial part of I/O statements. These codes are being developed by using a parallel C++ language[14][15] with our parallel I/O library.

The first application in our experiment is a gas-turbine-combustor model for simulating dilution jets[17]. The 3D grid is partitioned among processors and the data distribution scheme is illustrated in Figure 9. Two of the three dimensions are distributed among processors while the dimension along the k direction is not distributed. The data distribution is [Block,Block,*]. The data structure represented in a parallel C++ programs is shown below.

```
class SubGrid {
    double  U1[NZ],V1[NZ],W1[NZ],P[NZ],VIS[NZ],DEN[NZ], . . .
     . . .
};
DistributedArray<SubGrid>  Volume([MAXPROC],[NX,NY],[Block,Block]);
```

The above is an extended syntax of an experimental parallel C++ language[14]. We declare a distributed array of "SubGrid" with the array being distributed by [Block,Block] distribution. The element of an array

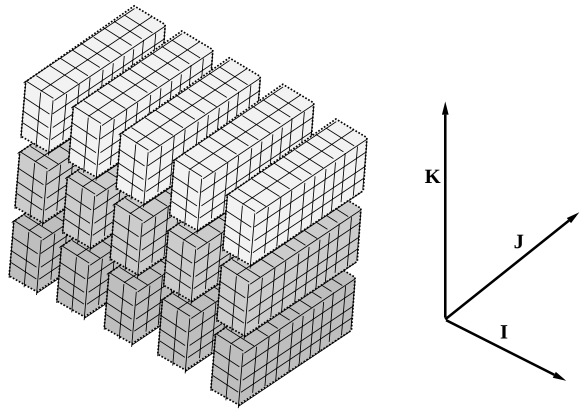

Figure 9: Distributed Array Used in Combustion-Chamber Model.

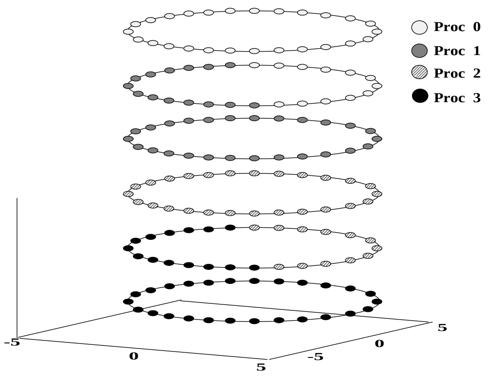

Figure 10: The Structure decomposition for a Parallel Vortex Filament Method.

is a SubGrid consisting a vector of grid points. The information in a subgrid includes positions, velocity, pressures, viscosity, densities, etc.

The algorithm is basically divided into three steps. In the first step, a finite difference scheme (QUICK scheme) in a 3D plane with 32 points in the stencil pattern is used. After the finite difference scheme is completed, we need to solve a group of linear systems using ADI methods. At this point in each iteration, the fluid data of the three-dimensional grids are written out by our parallel I/O operators. The periodical output of the fluid data can make the checkpoints to restart the program in case the machine is down and can be used for programmers to observe (or visualize) the progress of a program. Using the parallel I/O library we can parallelize both the read of the initial grid data and the output of the grid data in each iteration. The parallelization of I/O statements can reduce the time with I/O statements, simplify program coding efforts, and reduce the storage consumptions. Without using a parallel write operations, the program originally had to collect all the data of the distributed array into one processor and then write the data to a disk from one processor. Table 4 compares the time spent between a parallel and sequential write operation with the gas-turbine-combustor model. In this experiment, the size of the distributed array is 32 by 16, and the size of each SubGrid is 3296 bytes, as it contains the third dimension of the structures and many essential information for fluid computations. The total size of the distributed array is around 1.68MB. With parallel write operations, it only takes 0.10 seconds on a 8 processor DEC ALPHA Farm with Ethernet connections to write the data into the disk in parallel. With sequential write operations, all of the processors has to first move the data through network into processor 0, which takes 3.40 seconds. And then it takes 0.62 seconds to finish the disk write operations. A parallel write operation significantly out-perform a sequential write operation.

The second application in using our parallel I/O libraries is a parallel vortex method to simulate the turbulence of a three-dimensionally evolving jet. Vortex method[20] is widely used to simulate vortex induced flow problems, and is adopted to simulate the evolution of three-dimensionally periodical jet under axial perturbations. The global interaction of all point vortexes is a typical N-body problem with an operation

Write/Time	Write Operations	Communication	Total(seconds)
Parallel Write with 8 Nodes	0.10	0.0	0.10
Sequential Write with 8 Nodes	0.62	3.40	4.02

Table 4: Parallel Write v.s. Sequential Write in the Combustion-Chamber CFD Application.

cost proportional to N*(N-1). Our whole data structures are composed by a group of rings distributed among three dimensional spaces, with each ring consisting of a collection of vortexes. Figure 10 shows a system of six rings distributed among four processors. There are four fragments in this system, one for each processor. Each fragment consists of two sub-rings, with one being a complete ring and the other being a half ring. The whole algorithm is then carried out by each fragment computing the interactions among each other, stretching the vortexes, and saving the data on every M iterations, for some constant M. The main program loop looks like

```
class Fragment {
      double pos1[m+1][4],vel[m+1][4],cir[m+1],nber[mnr+1], . . .
      . . .
 };.
DistributedArray<Fragment>      Volume([MAXPROC],[MAXPROC],[Block]);
main() {
 pfp=pCreate("/pfs/user/local/project/vortex.iof","w+r");
 for (iter=1; iter<max_iteration; i++){
      calculate velocity (0);   calculate velocity (1);
      stretch();
      Volume.pWrite(pfp,"Volume"); /* parallel I/O to save data */
 }
 pClose(pfp);
}
```

Similar to the first application, this application is using parallel I/O to read the initial vortex data and output the vortex data in each iteration. In this experiment, there are 16 rings and each ring can hold up to 1000 vortex elements. The total size of the structure is around 1.1MB. With parallel write operations, it only takes 0.07 seconds on a 8 processor DEC ALPHA Farm with Ethernet connections. With sequential write operations, it first moves the data into one processor which takes 2.13 seconds, and then it takes 0.42 seconds to finish the disk write operations. Totally, a parallel write operation is almost 36 times faster than a sequential write operation after adding the factor of network traffics. This is calculated from the formula below.

$$Speedup = (T_{SequentialWrite} + T_{communication})/T_{ParallelWrite} = (0.42 + 2.13)/0.07 = 36$$

Write/Time	Write Operations	Communication	Total(seconds)
Parallel Write with 8 Nodes	0.07	0.0	0.07
Sequential Write with 8 Nodes	0.42	2.13	2.55

Table 5: Parallel Write v.s. Sequential Write in the Parallel Vortex Method.

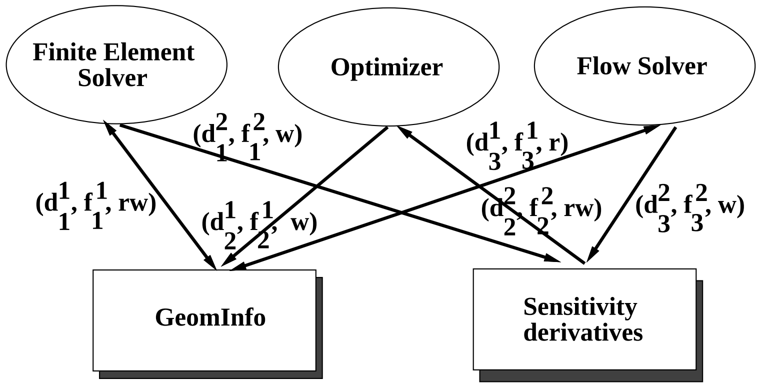

Figure 11: An Example of Multidisciplinary Optimization Application

Experiments on Selecting Distribution Pattern

The selection of a data distribution pattern for the store of array object in the file system is particularly useful in a multiple disciplinary optimization problem (MDO) in which various discipline codes interacting with each other to analyze the data. Figure 11 shows the I/O precedence graphs in our model for a simplified sample of MDO applications[24] with an aircraft design. A realistic multidisciplinary optimization of a full aircraft configuration would require a number of discipline codes including aerodynamic analysis, structural design analysis, control system analysis, performance analysis, etc, to interact with each other. Various discipline codes can be executed as a pipeline or executed asynchronously with data file being exchanged at various points in the code. In the sample application in Figure 11, optimizer, finite element solver, and flow solver all access the surface geometry information object which is stored in a parallel file. The data distribution pattern of the surface geometry information used in the optimizer, finite element solver, and flow solver might be different. In this case, selecting a good disk distribution pattern by using our proposed scheme in Section 4 can decrease total execution time of a full system.

Figure 12 plugs in a template data set to model MDO problem. Suppose the optimizer, flow solver, and finite element solver uses the grid object with (Block,Block), (Block,Cyclic), and (Cyclic,Cyclic) distribution patterns, respectively and there are totally four processors in use. Figure 12 shows the basic data movement ratio between different distributions. The number associated with the edge represents the data movement ratio from memory to disk for the two different distributions. The number such as 1/2 represents the amount of remote data movement over the size of the array object. Let's assume that each application has the same frequency in accessing the object in disk. By employing our dynamic programming algorithm in selecting the best data distribution pattern, a selection of (Block,Cyclic) as the disk distribution will minimize the

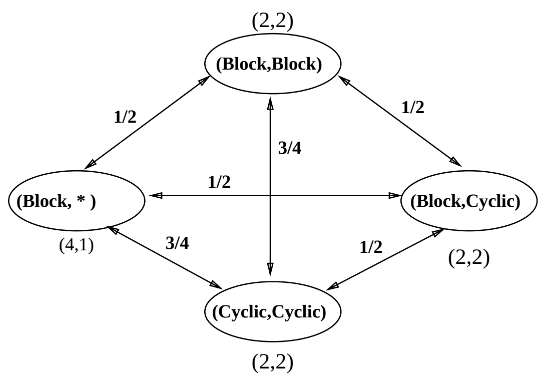

Figure 12: Data Movement Ratio between Different Distribution Pattern with 4 Nodes

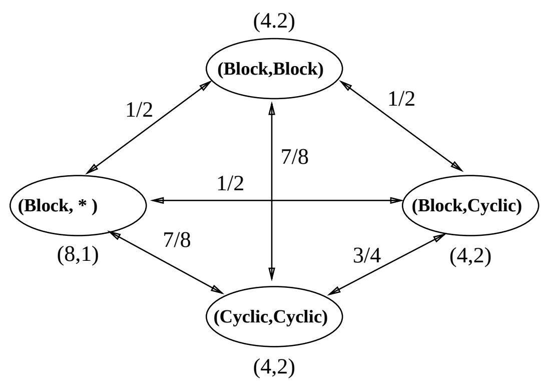

Figure 13: Data Movement Ratio between Different Distribution Pattern with 8 Nodes

Disk/Memory Pattern		(Block,Block)	(Block,Cyclic)	(Cyclic,Cyclic)	Total
(Block,Cyclic)	remote move ratio	1/2*N	0	1/2*N	N
	communication time	25.8	0	27.3	53.1
(Block,Block)	remote move ratio	0	1/2 *N	3/4*N	5/4*N
	communication time	0	25.7	56.5	82.2
(Cyclic,Cyclic)	remote move ratio	3/4*N	1/2*N	0	5/4*N
	communication time	54.5	26.9	0	81.4
(Block,*)	remote move ratio	1/2*N	1/2*N	3/4*N	7/4*N
	communication time	26.6	25.6	52.7	104.9

Table 6: Predicted Data Movement Ratio vs. Experimented Performance on 4 Nodes

remote references. The optimization is based on the formula in Section 4 as shown below.

$$Minimize : \Sigma_{i=1}^{m} \ f_i \times \Delta(\tau, D_i)$$

where f_i is the execution frequency of ith application, and $\Delta(\tau, D_i)$ is the amount of remote data movements for a given distribution τ under the condition that the application is using D_i at the memory distribution scheme.

Similarly, Figure 13 illustrates the same idea but with 8 processors. The diagram shows the remote data movement from memory to disk with different distributions. The (Block, *) distribution is to distribute the first dimension of the array by block, but there is no distribution for the second dimension of the array. It's used as one extra reference point to show the data movement rate with different distribution pattern. Suppose we still use the same assumption with the four processor case, and apply the selection algorithm in Section 4, we will find the (Block,Cyclic) distribution is the best distribution.

Focus is now directed to Table 6, where we compare the predicted performance by our selection scheme with actual performance results. The left hand side of the table represents the disk patterns of the objects if chosen. The first row of the table represents a template data set of the MDO problem. Suppose the

Disk/Memory Pattern		(Block,Block)	(Block,CYclic)	(Cyclic,Cyclic)	Total
(Block,Cyclic)	remote move ratio	1/2*N	0	3/4*N	5/4*N
	communication time	39.5	0	79.9	119.4
(Block,Block)	remote move ratio	0	1/2 *N	7/8*N	11/8*N
	communication time	0	40.4	110.8	151.2
(Cyclic,Cyclic)	remote move ratio	7/8*N	3/4*N	0	13/8*N
	communication time	109.2	82.2	0	191.4
(Block,*)	remote move ratio	1/2*N	1/2*N	7/8*N	15/8*N
	communication time	41.0	40.3	111.1	192.4

Table 7: Predicted Data Movement Ratio vs. Experimented Performance on 8 Nodes

optimizer, flow solver, and finite element solver uses the grid object with (Block,Block), (Block,Cyclic), and (Cyclic,Cyclic) distribution patterns in the memory respectively, and let N be the size of the array object. The predicted smallest remote movement ratio is N if we choose (Block,Cyclic) as the disk distribution. The actual execution time is more than 30% improvement over the next best selection. The array size in this experiment is 64 by 64, and each element of the array is again of size 64 bytes. The time measured in the table is in the unit of 10^{-2} second. Note that due to CSMA/CD behavior in the ethernet, the movement ratio is not linearly proportional to the actual execution time. In one instance in our experiment, when the data movement ratio increases from one 1/2*N to 3/4*N, the actual communication time increases nearly 100 percents. This observation further underlines the importance of pattern selections. Table 7 shows a similar result, but experimented on an 8-node DEC Farm.

Finally, our solver for selecting proper distributions for parallel file system is very efficient in our experiments. For example, in one of the test run, with array size 4000×4000, processor array size 4×4, array object distribution scheme [B(10), B(10)], and file array distribution [Cyclic,Cyclic], it only takes 0.034 seconds at a SUN Sparc 10/30 machine to decide the number of remote references needed.

7 Conclusion

This paper describes a parallel file object environment to support distributed array store on shared-nothing distributed computing environments. Our environment enables programmers to extend the concept of array distribution from memory levels to file levels. It allows parallel I/O according to the distribution of objects in an application. Currently, our parallel I/O library is incorporated into an experimental parallel C++ and a subet set of HPF compiler on a 8-node DEC Farm with Ethernet or FDDI connections. In addition, this paper proposes a selection scheme to help programmers to select the best data distribution pattern according to the minimum amount of remote data movements for the store of array objects on distributed file systems, when objects are read and/or written by multiple applications using different distributions. This selection scheme, to our best knowledge, is the first work to optimize the distribution patterns in the secondary storage with HPF-like programs for multiple disciplinary applications. Our experimental result shows that not only our parallel file system can provide aggregate bandwidths, but also our selection scheme effectively reduce the communication traffics for the system.

References

[1] G.M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities*, Proc. AFIPS 1967 Spring Joint Computer Conference, Atlantic City, N.J., pp. 483–485, Apr. 30, 1967.

[2] Rajesh R. Bordawekar, Alok N. Choudhary, Juan M. del Rosario. *An Experimental Performance Evaluation of Touchstone Delta Concurrent File System*, International Conference on Supercomputing, Tokyo, Japan, July 20-22, 1993.

[3] F. Bodin, P. Beckman, D. Gannon, Srinivas Narayana, and S. Yang. *Distributed pC++: Basic Ideas for an Object Parallel Language*, Scientific Programming, 2(3), Fall 1993.

[4] Peter Brezany, Michael Gerndt, Piyush Mehrotra, Hans Zima. *Concurrent File Operations in a High Performance Fortran*, Proceedings of Supercomputing '92, 1992.

[5] P. F. Corbett and D. G. Feitelson. *Design and Implementation of the Vesta Parallel File System*, Proceedings of IEEE SHPCC, June 1994.

[6] T. L. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, The MIT Press.

[7] Rajesh Bordawekar, Juan Miguel del Rosario, Alok Choudhary. *Design and Evaluation of Primitives for Parallel I/O*, Proceeding of Supercomputing 93, Oregon, November 15-19, 1993.

[8] Alok Choudhary, *Parallel I/O Systems*, Journal of Parallel and Distributed Computing, January/February, 1993.

[9] Jacob Gotwals, Suresh Srinivas, Dennis Gannon. *pC++/streams: a LIbrary for I/O on COmplex DIstributed Data Structures*, Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming (PPoPP '95), Santa Barbara, July 19-21, 1995. (Also in SIGPLAN Notices, Aug 1995).

[10] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. *Fortran D Language Specification*, Department of Computer Science, TR90079, Rice University, March 1991.

[11] Gwan-Hwan Hwang, Jenq Kuen Lee, and Dz-Ching Ju. An Array Operation Synthesis Scheme to Optimize Fortran 90 Programs, Proceedings of ACM SIGPLAN Conference on Principles and Practice of Parallel Programming, pp. 112-122, July 1995.

[12] C. Koelbel. *Compile-Time Generation of Regular Communications Patterns*, Proceedings of Supercomputing '91, Albuquerque, Nov. 18-22, 1991.

[13] C. Koelbel , D. Loveman and Schreiber, G. Stelle and M. Zosel. *The High Performance Fortran Handbook*, MIT-press, Cambridge, 1994.

[14] Jenq Kuen Lee and D. Gannon. *Object-Oriented Parallel Programming: Experiments and Results, Proceedings of Supercomputing '91*, New Mexico, November, 1991.

[15] Jenq Kuen Lee and Dan Ho. *Integrating Lattice Parallelism into a Collection-Oriented Language*, *Proceedings of HPC Asia '95*, Taipei, Taiwan, September 1995.

[16] Jenq Kuen Lee, Ing-Kuen Tsaur, San-Yih Huang. *Language and Environment Support for Parallel Object I/O on Distributed Memory Environments*, *Proceedings of the 7th SIAM Conference for Parallel Processing in Scientific Computing*, S.F., Feb. 1995.

[17] Jenq Kuen Lee, Y. Chen, C. A. Lin, and C. M. Lu. *Modelling Three-Dimensional Gas-Turbine-Combustor-Model Flows on Parallel Machine with Distributed Memory*, *Parallel Computational Fluid Dynamics: New Trends and Advances*, A. Ecer et al. editors, Elsevier Science, 1995.

[18] Lee et al. '*Scalable Computing: Algorithm, Application, System*, National Science Counsel Project Proposal, Tsing-Hua University, 1995.

[19] Jenq Kuen Lee, et al. '*A National Science Counsel Project Report on A High-Performance Fortran Compiler, 1993-1995*, Tsing-Hua University, June 1995.

[20] Leonard, A. *Computing Three-Dimensional Incompressible Flows with Vortex Elements* Ann. Rev. Fluid Mech. 17 (1985) 523.

[21] Dennis Gannon. *Libraries and Tools for Object-Oriented Parallel Programming*, Technical Report, Indiana University, 1992.

[22] Dennis Gannon. *HPC++ Level 1. Future Directions for Parallel C++*, The Draft of the HPC++ Discussion Group, Indiana University, September 17, 1995.

[23] N. Galbreath, W. Gropp, D. Levine. *Applications-Driven Parallel I/O*, Proceedings of Supercomputing, Oregon, November 15-19, 1993.

[24] Mathew Haines, Brian Hess, Piyush Mehrotra. *Exploiting Parallelism in Multidisciplinary Application Using Opus*, Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, David Bailey et al, editors, SIAM 1995.

[25] S. A. Moyer and V. S. Sunderam. *PIOUS: A Scalable Parallel I/O Systems for Distributed Computing Environments*, Proceedings of IEEE SHPCC, June 1994.

[26] J. Roseario, Alok N. Choudhary. *High Performance I/O for Massively Parallel Computers: Problems and Prospects*, IEEE Computer, Volume 27, Number 3, March 1994.

[27] Y. L. Shieh, Jenq Kuen Lee, J. H. Tsai, and C. A. Lin. *Computations of Three-Dimensionally Evolving Jets with Vortex Methods on Parallel Machines with Distributed Memory*, Parallel CFD '94, Kyoto, Japan, May 1994.

[28] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*, Addison-Wesley Publishing Company, May 1990.

[29] Rajeev Thakur, Rajesh Bordawekar, Alok Choudhary. *Compiler and Runtime Support for Out-of-Core HPF Programs*, Proceedings of International Conference on Supercomputing, Manchester, U.K., 1994.