Exploiting Inter-File Access Patterns Using Multi-Collective I/O*

Gokhan Memik
EE Dept.
UCLA
memik@ee.ucla.edu

Mahmut Kandemir
CSE Dept.
Penn State
kandemir@cse.psu.edu

Alok Choudhary
ECE Dept.
Northwestern University
choudhar@ece.nwu.edu

Abstract

This paper introduces a new concept called Multi-Collective I/O (MCIO) that extends conventional collective I/O to optimize I/O accesses to multiple arrays simultaneously. In this approach, as in collective I/O, multiple processors co-ordinate to perform I/O on behalf of each other if doing so improves overall I/O time. However, unlike collective I/O, MCIO considers multiple arrays simultaneously; that is, it has a more global view of the overall I/O behavior exhibited by application. This paper shows that determining optimal MCIO access pattern is an NP-complete problem, and proposes two different heuristics for the access pattern detection problem (also called the assignment problem).

Both of the heuristics have been implemented within a runtime library, and tested using a large-scale scientific application. Our preliminary results show that MCIO outperforms collective I/O by as much as 87%. Our runtime library-based implementation can be used by users as well as optimizing compilers. Based on our results, we recommend future library designers for I/O-intensive applications to include MCIO in their suite of optimizations.

1 Introduction

Significant strides made in microprocessor performances in the last decade have increased the importance of optimizing I/O performance. While a highly-optimized I/O platform/hardware is a must for optimal I/O, software optimizations can also play a significant role. This is particularly true for a class of large-scale scientific applications whose access patterns can be analyzed by users/compilers and optimized for the best I/O performance. Previous work has attacked this growing I/O problem at the operating system, runtime libraries, compilers and application levels. One of the major characteristics of most of the previous approaches to I/O is that they attempt to optimize I/O access pattern for a single data structure (e.g., disk-resident array) at a time. While this approach can be useful for some applications, we believe that considering the interactions between different data structures manipulated by the same application opens new opportunities for optimization. For example, considering accesses to multiple arrays manipulated by an application can enable inter-array optimizations such as co-locating arrays in disk space or performing prefetching based on history of array accesses.

This paper introduces a new concept called Multi-Collective I/O (MCIO) that extends conventional collective I/O (CIO) to optimize I/O accesses to multiple arrays simultaneously. In this approach, as in collective I/O, multiple processors co-ordinate to perform I/O on behalf of each other if doing so improves overall I/O time. However, unlike collective I/O, MCIO considers multiple arrays simultaneously; that is, it has a more global view of the overall I/O behavior exhibited by application. This paper shows that determining optimal MCIO access pattern is an NP-complete problem, and proposes two different heuristics for the access pattern detection problem (also

*This research was supported in part by Department of Energy under the Accelerated Strategic Computing Initiative (ASCI) Academic Strategic Alliance Program (ASAP) Level 2, under subcontract No W-7405-ENG-48 from Lawrence Livermore National Laboratories, by DOE SCIDAC SDM center and by a grant for NSF, EIA-0103023.

called the assignment problem).

Figure 1 highlights the difference between MCIO and traditional collective I/O. In MCIO, we optimize the accesses from multiple processors to *multiple files* taking into account the inter-file access patterns. In CIO, on the other hand, accesses to a *single file* from different processors are combined to a single and larger I/O request to improve the request time. In MCIO, accesses from several processors to several files are combined to increase the efficiency.

MCIO might be very useful in a number of cases. Many large-scale scientific applications access multiple files in a single run. Several of these applications generate separate files for the simulated values. This forces the application to access data from multiple files to gather the required information. A simple analysis of the code reveals this information, which can be used by the MCIO. MCIO can also be utilized when data for different variables are accessed from a single file. For example, *astro3d* [15], the scientific application we have used in this study, accesses six different variables from a single file. These variables are stored in separate buffers, hence six different I/O calls have to be performed to access the information required by the application. MCIO can be utilized in this case. Note that, accessing different variables from a single file is not a separate problem, it is just a special case for the processor-file configuration in MCIO. Most of the scientific applications fall either in the first (accessing multiple files in a single run) or into the second (performing consecutive I/O calls to a single file to access different variables) category. Therefore, a large majority of the large-scale scientific applications can effectively utilize MCIO.

Another usage of MCIO is related to sub-filing. Memik et al. [16] uses sub-filing for optimizing random accesses to the tape-residing data. In this framework, a large global file is stored as several independent so called *sub-files*. In general, each access to the global file might involve several sub-files. In other words, an access to the global file brings only the smallest subset of sub-files (from tape to disk) that (collectively) contain the required data portion. This not only reduces the effective latency observed from the tape device, but also allows customized management of individual sub-files across storage hierarchy depending on data reuse. MCIO can be utilized to access data that has been brought to the disks.

In this paper, we make the following major contributions:

- We introduce the MCIO technique that improves over the current state-of-the art collective I/O technique;
- We discuss the complexity of the MCIO, and present two different heuristics (one based on sorting and the other based on maximal matching) to perform effective MCIO; and
- We evaluate the effectiveness of the MCIO by using both synthetic workloads and a complete large-scale scientific application.

Both of the heuristics have been implemented within a runtime library, and tested using a large-scale scientific application. Our preliminary results show that MCIO outperforms conventional collective I/O by as much as 87%. Based on our results, we recommend future library designers for I/O-intensive applications to include MCIO in their suite of optimizations.

The remainder of this paper is organized as follows. In the next section, we discuss related work on software-based I/O optimizations. In Section 3, we summarize collective I/O. In Section 4, we discuss the MCIO in detail, and show that finding the optimal access pattern for MCIO is NP-complete. In Section 5, we discuss two different heuristics for determining suitable access patterns. Section 6 introduces our experimental environment and discusses our preliminary results. Finally, we conclude the paper with a summary and an outline of future work in Section 7.

2 Related Work

In collective I/O, individual I/O requests are combined to create a single, big I/O request and sent to the storage system. As a result, the effective I/O bandwidth is

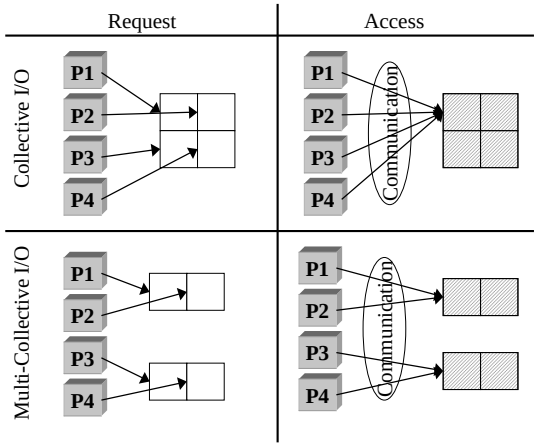

Figure 1: Overview of the MCIO.

significantly increased. This optimization has many variants [19, 13, 22]; although any CIO technique can be used for MCIO, the one used in this study is *two-phase I/O* as implemented in ROMIO, a portable implementation of MPI-IO from Argonne National Laboratories [26]. ROMIO has been incorporated into several MPI-libraries, including the MPI implementations of several vendors (e.g., HP, SGI, NEC) and MPICH and LAM, two widely-used, freely available, portable MPI implementations.

Numerous techniques have been proposed in literature to optimize I/O accesses. The run-time system optimizations [14, 12, 6, 4, 5] are the most relevant among these studies. Although these techniques share the same goal with our work (optimizing I/O accesses at the run-time), they try to optimize only a single file at a time.

Several researchers focused on implementing easy-to-use interfaces that include optimizations for data-intensive scientific applications [24, 3, 22, 23]. These interfaces try to improve the I/O performance with little input from the user. MCIO can be employed by these interfaces with little or no modification.

Characterizing the I/O behavior of the scientific applications has been extensively studied. Cypher et al. [9] studied individual parallel scientific applications, measuring temporal patterns in I/O rates. Crandall et al. [8] performed an analysis based on Pablo [1] on three scientific applications. Nieuwejaar et al. [18] characterized a mix of user programs on Intel iPSC and CM-5. All these studies implicitly motivate the usage of MCIO by illustrating that multiple files are accessed within a single run of the studied application. According to these studies, the number of files accessed varies according to the application and the file system studied but can be as high as 2000.

Several parallel I/O APIs provide routines for accessing multiple files with a single call. For example, HPSS [7] uses the notion of a 'file set' to define a collection of files. The files in a file set can be manipulated as if they constitute a single file. Similarly, SRB [2] uses 'collections' to define a set of files.

3 Collective I/O

Since MCIO is an extended form of conventional collective I/O, in this section we briefly review the fundamental idea behind collective I/O. In many parallel I/O-intensive applications that access large, multidimensional, disk-resident datasets, the performance of I/O accesses depends largely on both the layout of data in files (*storage pattern*) and distribution of data across processors (*access pattern*). In cases where storage and access patterns do not match, allowing each processor to perform I/O independently might cause processors to issue many I/O requests, each for a small amount of consecutive data. Collective I/O can improve the performance in such cases by first reading the dataset in question in a *layout-conforming* (storage layout friendly) manner, and then distributing the data among the processors (using the inter-processor communication network) to obtain the target access pattern. Of course, in this case, the total data access cost should be computed as the sum of I/O cost and communication cost. Since, in I/O-intensive applications, I/O costs in general dominate communication costs, collective I/O might lead to large savings in overall execution times.

Consider an example file access pattern (to a two-dimensional array) depicted in Figure 2. In this figure, the circles correspond to data elements and arrows indicate the storage pattern of the elements. Note that in this case, the storage pattern is row-major whereas the access pattern is column-major. If no collective I/O technique is used, every processor make 8 small requests (each for

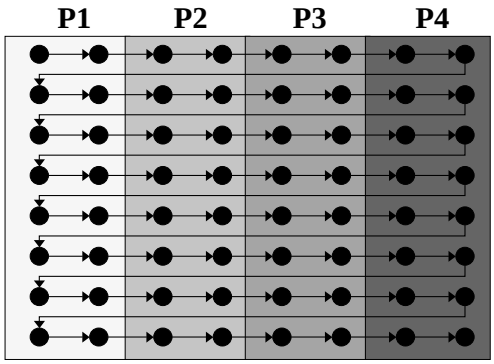

Figure 2: A file access by four processors.

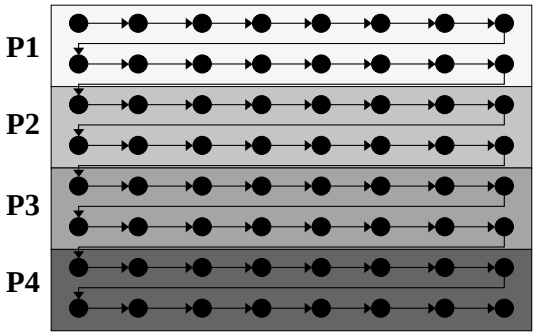

Figure 3: File access using the collective I/O.

two array elements) for a total of 24 small requests to the I/O subsystem (the numbers are for illustrative purposes only). Obviously, this will result in a poor performance as the number of elements read per I/O request is very small. Collective I/O combines these small requests, and sends larger requests to the I/O subsystem to improve the performance. If, for example, the *two-phase I/O* technique is used, in the first step, the processors access the data using a row-major access pattern which is compatible with the (row-major) storage pattern of the array (see Figure 3). Note that this reduces the number of I/O calls as each processor can read as many consecutive data as possible (limited only by available buffer capacity) in a single I/O call. In the second step, the processors engage in all-to-all inter-processor communication so that each processor receives the portion of array it originally requested (that is, each data item is delivered to its final destination).

Several variations of the collective I/O technique have been proposed in previous research. In node grouping [20], nodes making I/O requests are partitioned into groups. Then, they take turns in performing the I/O. In disk-directed I/O [13], one compute node sends the collective request to all I/O nodes. I/O nodes optimize the access, perform the request, and send the data directly back to all compute nodes.

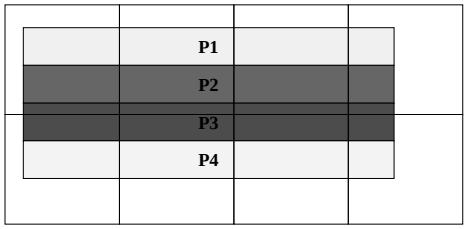

Figure 4: An access pattern involving four processors and eight files.

4 Multi-Collective I/O

In this section, we explain the MCIO technique in detail. Consider Figure 4 where four processors are requesting different portions of eight files. Note that, these files might contain independent data. The problem is to read the corresponding data portions from the corresponding files as fast as possible. There are several methods to achieve this. In a naive method, we would use the CIO for each of the files. For our example in Figure 4, this will result in 8 different I/O calls (each involving two or three processors). In this method, as the number of files increase, the number of I/O calls will increase linearly regardless of the number of processors available. In this paper, we explore methods that improve over this naive method. The main idea behind these methods is to assign different files to different processors, thereby increasing the I/O parallelism available in the system.

In the next subsection, we define the assignment problem of MCIO in detail and show that this problem is NP-complete [11]. There are two important questions that need to be answered. First, how many files should we assign to each processor, or vice versa? Second, once we have determined the number of processors to be assigned for each file, how do we decide which specific processor to assign for each file? In the following subsections, we try to answer these questions. In the rest of this section, we concentrate on the I/O accesses where the number

of files exceeds the number of processors. Note that the problem we try to solve here is symmetric. If the number of files is larger, we try to assign files to processors; otherwise, if the number of processors exceeds the number of files, we try to assign the processors to files. It is easy to see that these two problems are actually duals of each other.

4.1 The Assignment Problem

In this subsection, we first define the assignment problem in detail. Then, we show that the problem of deciding the files to be read by a processor is NP-complete. After this, we build a Linear Programming (LP) model of the problem. As we will show in the following, the assignment variables can either be 0 or 1, hence the model we build is a zero-one integer LP. Since it is slow to construct and solve an LP model in a run-time system, we have also developed two heuristics to solve the problem, which we discuss in the next section.

4.1.1 Definition of the Assignment Problem

In the assignment problem, p processors are requesting data from n different files. For simplicity, we assume that $n > p$. The assignment problem is to assign a processor for each file, such that when processors access their assigned files, the overall response time for the requests (total of I/O and communication times) is minimized. We estimate the I/O time by the amount of data accessed by a processor. The communication time is estimated by the amount of data that has to be transfered to/from other processors. So, the assignment problem is to find the optimal assignment of the processors to the files, such that $max_i\{\alpha \times I/O_i + \beta \times comm_i\}$ is minimized. α and β are constant values indicating the relative cost of I/O and communication in the system. I/O_i and $comm_i$ are the estimated I/O and communication times of processor i, respectively. They are estimated using the following formulas:

- $I/O_i = \sum_{j=1}^{n} a_{i,j}$ (the amount of data accessed by processor i)
- $comm_i = \sum_{j=1}^{n} |r_{i,j} - a_{i,j}|$ (the amount of data accessed by processor i subtracted from the amount of data requested by processor i).

In the above formulas, $r_{i,j}$ corresponds to the amount of data requested by processor i from file j. And, $a_{i,j}$ corresponds to the amount of data to be accessed by processor i from file j.

4.1.2 Complexity of the Assignment Problem

In this subsection, we show that the assignment problem as defined in the previous subsection is NP-complete. We prove that optimizing the I/O time (when $\alpha = 1$ and $\beta = 0$) is an NP-complete problem.

Claim: For arbitrary number of processors, number of files, and file sizes, finding the optimal assignment to minimize the I/O time is an NP-complete problem.

Sketch of the Proof: We prove the NP-completeness of the assignment problem using restriction. More specifically, we show that if we can solve the assignment problem in polynomial time, then we can solve the multiprocessor scheduling problem [11] in polynomial time, too. Multiprocessor scheduling problem is finding the m disjoint partitions of a finite task set A (A_1, A_2, ..., A_m), such that

$$max_i \quad \{\sum_{a \in A_i} length(a)\}$$

is minimized. Assume that we have a solver for the assignment problem that finds the optimal solution in polynomial time. Then, given a multiprocessor scheduling problem, we can easily transform the length of each task to a corresponding file size and use the assignment problem solver to solve the multiprocessor scheduling problem. Hence, we would be able to solve an arbitrary multiprocessor scheduling problem in polynomial time. Therefore, the assignment problem is NP-complete in the strong sense, because the multiprocessor scheduling problem is NP-complete in the strong sense for arbitrary m. □

4.2 Deciding the Number of Processors

Although the general assignment problem is NP-complete for arbitrary file sizes, for files of equal size, it can be

solved in polynomial time. In this section, we discuss how the assignments should be done in such a case. Specifically, we try to find the number of processors that will be used for accessing each file. In finding this value, we use the following basic model for the I/O time:

$$c_1 + c_2 * ((DataSize)/(Number\ of\ Processor))$$

where c_1 (corresponding to constant costs independent of the request size such as seek time) and c_2 (corresponding to transfer rate) are system dependent constant values. The intuition behind the model is that in a parallel environment the amount of data read by each processor will be reduced by increasing the number of processor. Hence if we omit the communication, there will be a decrease in the I/O time.

The total time for the request will be determined by the processor for which the I/O time is maximum (i.e., the one that completes the last). If we assume that the data sizes to be read from each file to be equal and that we assign k processors to each file, the response time will be equal to

$$m * c_1 + m * c_2 * ((DataSize)/(k)) \quad \text{Eq. 1}$$

where m is the number of files that the slowest processor is assigned to. If the assignments are done homogeneously among the processors, then

$$k = \frac{m*p}{n}$$

where p is the number of processors and n is the number of files. So, Equation 1 can be re-written as

$$m * c_1 + \frac{n}{p} * c_2 * DataSize$$

For a given number of processors and number of files, this expression takes its minimum for the minimum value of m. Since we have to assign at least one processor for each file the minimum value for m is $(n\ /\ p)$. That means, we have to assign only one processor to each file to minimize the response time. Note that, in the naive strategy, several processors are assigned to the files, which will increase the response time according to our model.

If the number of processors is larger than the number of files, then we have to assign one file to each processor only (the reverse of one processor for each file). The calculations are similar to the calculations given above. Note that in the calculations, we assume a homogeneous distribution of the files to the processors. If the file sizes are different, this might affect the response time significantly. This case is discussed in detail in Section 4.3.

Although we have found that we have to assign one processor to each file, we still have to determine which processor to assign to each file. Our selection of the processor will have a marginal effect on the I/O times, but it will have a significant effect on the communication times.

4.3 LP Model of the Assignment Problem

We are given a two-dimensional matrix $R = [r_{i,j}]$ such that an entry $r_{i,j}$ in R gives the amount of data requested by processor i from file j. In our ILP formulation, we would like to find the entries of a matrix $X = [x_{i,j}]$. An entry $x_{i,j}$ indicates whether the processor i is assigned to file j ($x_{i,j} = 1$) or not ($x_{i,j} = 0$). As mentioned earlier, although our selection of the processor for each file will not affect the I/O time, it will effect the communication time significantly. In the following discussion, n corresponds to the number of files to be accessed, p corresponds to the number of processors involved and we assume that $n > p$. So, we try to minimize

$\sum_{i=1}^{p} \sum_{j=1}^{n} r_{i,j} \times (1 - x_{i,j})$

This is because minimizing this expression will give us the total amount of data to be communicated. We will have the following constraints on $x_{i,j}$:

- $x_{i,j} \in \{0,1\}, \quad \forall i, j$

$x_{i,j}$ is a decision variable and should be either *assigned* or *not assigned*. Hence, the LP model of the problem is boolean or zero-one integer LP.

- $\sum_{i=1}^{p} x_{i,j} = 1, \quad \forall j$

In the previous section, we have found that each file will be assigned to exactly one processor. The above equation corresponds to this constraint. If the number of processors exceeds the number of files, this constraint becomes

- $\sum_{j=1}^{n} x_{i,j} = 1, \quad \forall i$

resulting in each processor being assigned to only one file.

- $\sum_{j=1}^{n} x_{i,j} = \frac{n}{p}, \quad \forall i$

This constraint makes sure that the assignments are homogeneous. It states that the number of files assigned to processor i equals to $\frac{n}{p}$. Here, we assume that regardless of the data size read from the files, we are going to assign same number of files to each processor. Although this assumption makes the calculation of the assignment variables easier, it might decrease the performance if the variance between the data sizes read is large. In such a case we can use the constraint

- $\sum_{j=1}^{n}(\sum_{k=1}^{p} r_{k,j}) \times x_{i,j} \leq P, \quad \forall i$

which means that the total data to be read by each processor should approximately be the same. In this expression, P corresponds to the average data size each processor should access. If the number of processors exceeds the number of files, then the constraint becomes

- $\sum_{i=1}^{p} x_{i,j} = F_j, \quad \forall j$

where F_j represents the number of processors to be assigned to file j. This number can easily be found using the amount of data read from the file j and all the other files. To calculate F_j, we use the following formula

$$F_k = p \times \frac{\sum_{i=1}^{p} r_{i,k}}{\sum_{i=1}^{p}\sum_{j=1}^{n} r_{i,j}}$$

p processors are distributed according to the portion of the file j with respect to all the data read.

Note that, in this model, the number of variables is equal to (*number of processors*) $\times$ (*number of files*). The above ZO-ILP model helps us understand the nature of the problem. But, we cannot use this model in a run-time library to make the processor-file assignments, since even the fastest LP-solvers will require extremely long running times to find a solution. Consequently, we developed two heuristics to solve the problem. We discuss these heuristics in the next section.

5 MCIO Heuristics

In this section, we explain two heuristics for making the processor-file assignments. The first one uses a sorting algorithm to find the assignment. The second one uses a maximal matching solver to make the assignments.

5.1 Greedy Heuristic

The first heuristic uses sorting for making the assignments. The main idea behind the heuristic is to assign the specific processors to the file, from which they are reading the largest amount of data. To achieve this, we first create an entry for every case, where processor i is reading a part of file j. Then, all these entries are sorted according to non-increasing values of the amount of data the processor i is reading from file j. As an example, the request list for the pattern shown in Figure 4 is given in Table 2. This list is formed using the request sizes given in Table 1.

The algorithm tries to pick the first element from the list and assign the processor to the corresponding file. If the file has already been assigned or if the processor has already completed the number of assignments it should have (i.e. the processor is full), then the entry is skipped and the next entry is checked. The algorithm continues until we make n assignments. If the entries in the list are finished before we make n assignments, we make the rest of the assignments randomly from the remaining processor pool. We do not have to pay attention in this step, because if the entries are finished, this means that all of the remaining processors are not reading any data from the remaining files. Thus, we can make the assignments arbitrarily because it will not change the amount of data communicated. The pseudo code of the algorithm is given in Figure 5. Returning to our example, the resultant assignments for the access in Figure 4 are given in Figure 6. Consequently, file 1 and file 4 are assigned to processor 2, file 2 and file 3 are assigned to processor 1, file 5 and file 8 are assigned to processor 3, and file 6 and file 7 are assigned to processor 4.

The insert_entry function in Figure 5 inserts a new entry into the list with the processor number i, file number j, and weight $r_{i,j}$. Similarly, remove_entry function deletes the entry from the list that is equal to its input parameter. After execution, the algorithm returns the list of processor–file pairs that have been assigned.

Table 1: The amount of data requested by each processor from each file for the access in Figure 4. The files are named row-major order starting from the upper left corner.

Processor Number	File Numbers							
	1	2	3	4	5	6	7	8
1	10 MB	12.8 MB	12.8 MB	4.4 MB	0 MB	0 MB	0 MB	0 MB
2	10 MB	12.8 MB	12.8 MB	4.4 MB	0 MB	0 MB	0 MB	0 MB
3	2.5 MB	3.2 MB	3.2 MB	1.1 MB	7.5 MB	9.6 MB	9.6 MB	3.3 MB
4	0 MB	0 MB	0 MB	0 MB	10 MB	12.8 MB	12.8 MB	4.4 MB

Table 2: The list formed for the example access in Figure 4.

Access List
$P1 \rightarrow F2, P1 \rightarrow F3, P2 \rightarrow F2, P2 \rightarrow F3, P4 \rightarrow F6, P4 \rightarrow F7, P1 \rightarrow F1, P2 \rightarrow F1, P4 \rightarrow F5, P3 \rightarrow F6, P3 \rightarrow F7,$ $P3 \rightarrow F5, P1 \rightarrow F4, P2 \rightarrow F4, P4 \rightarrow F8, P3 \rightarrow F8, P3 \rightarrow F2, P3 \rightarrow F3, P3 \rightarrow F1, P3 \rightarrow F4$

5.2 Maximal Matching Heuristic

The second heuristic uses a maximal matching solver. We have used the solver from the Netflow Solver Package [25]. The matching problem solver inside Netflow implements Gabow's N-cubed weighted matching algorithm [10]. This program is written by Ed Rothberg. To be able to use an existing maximal matching solver, we first need to build a graph representing the $r_{i,j}$s. Then, we need to modify the graph so that the solver gives the answer we seek.

The first step is straightforward. We build a graph G (V, E), where V contains a vertex for each processor and file. More specifically, if there are p processors and n files, then the graph will have $(n + p)$ vertices. The resulting vertices for the example access in Figure 4 are given in Figure 8(a). Then, we put an edge $(u, v) \in E$ with weight w, when the processor u reads w bytes of data from file v. The resulting graph is shown in Figure 8(b).

The existing matching problem solvers do not solve the exact problem we are interested in. They instead solve the maximum flow problem, given an input graph. For a bipartite graph G ($V_1 \bigcup V_2$, E) (note that, the graph given to the algorithm is bipartite), they try to find a different vertex in V_2 for each vertex in V_1 such that the sum of the weight of the edges between the selected node pairs is maximized (i.e., the flow is maximized). If the number of vertices in V_2 is larger then the number of vertices in V_1, then some of the vertices in V_2 will be left out. Similarly, if the number of vertices in V_1 is larger then the number of vertices in V_2, then some of the vertices in V_1 will be left out. Therefore, one has to make sure that the number of vertices for processors equals to the number of processors for files. We replicate the processor nodes to be able to make the number of processor vertices equal to the number of file vertices. For each file node, we replicate F_j times the processor nodes that have an edge to the node[1]. Since the solver assigns only one node for each file vertex, it gives the result we are looking for. This way, we guarantee that the solver makes n assignments. The resulting graph is given in Figure 8(c). Note that, in this example, we assume that F_j equals to two for every file node.

Once the graph has been constructed, we give it to the matching problem solver as input. The result of the solver is used as the assignments between the processor and the files. The assignments for the replicated nodes are interpreted as if they are assignments to the original node. The assignments made by the maximal matching heuristic for the access given in Figure 4 are given in Figure 7. Note that, if the number of processors is larger than the number of files, we replicate the file nodes instead of the processor nodes so that in the final result several processors might be assigned to the same file.

[1] The calculation of F_j is explained in Section 4.3.

```
GREEDY_ASSIGN (p, n, r_{i,j})
1.  /*p is the number of processors, n is
2.  the number of files, r_{i,j} represents
3.  the amount of data read by processor i
4.  from file j.  */
5.  begin
6.    for i=0 to p do
7.      for j=0 to n do
8.        if r_{i,j} ≠ 0 then
9.          insert_entry (i, j, r_{i,j})
10.       end if
11.     end for
12.   end for
13.   sort_entries_according_to (last_field)
14.   while (!list empty) AND
15.   (assignments_made < n) do
16.     entry = list_top
17.     if entry.processor full OR
18.       entry.file assigned then
19.       remove_entry (entry)
20.     else
21.       assign(entry.processor,entry.
22.       assignments_made ++
23.     end if
24.   end while
25.   assign the remaining files to
26.   remaining processors arbitrarily
27.   return all_assignments
28. end.
```

Figure 5: Greedy algorithm for making assignments algorithm sorts the requests from processors to file cording to the access size. Then, it tries to assign the processors to the files such that the resulting assignment will result in a small communication overhead.

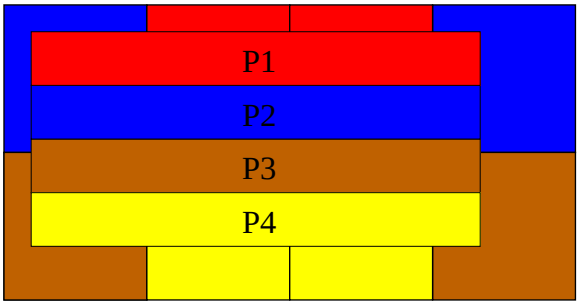

Figure 6: The result of the greedy heuristic. The colors denote the processor assigned.

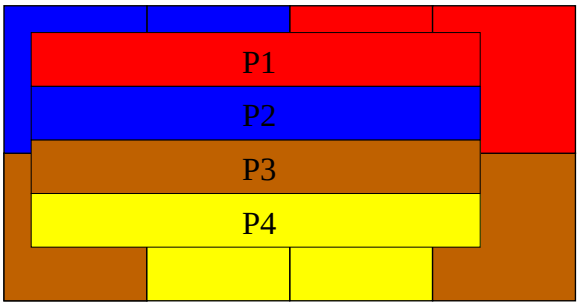

Figure 7: The result of the maximal matching heuristic. The colors denote the processor assigned.

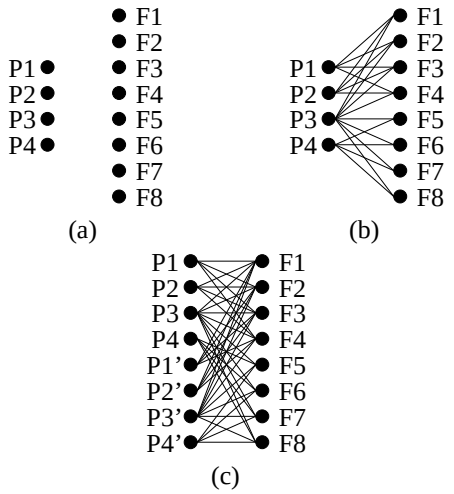

Figure 8: The creation of the input graph for the matching problem solver. The weights on the edges are left out for simplicity. (a) The initial graph with a node for each processor and a file. (b) The graph with an edge added for each case where processor i is reading data from file j. (c) The graph after the replication of the processor vertices to make the number of processor vertices and the file vertices to be equal.

Table 3: Platform used in the experiments.

Number of Processors	128 (120 compute nodes, 8 I/O nodes)
Processor Type	Compute Nodes: RS/6000 Model 370, I/O Nodes: RS/6000 Model 970
Clock Rate	332 MHz
L1 Cache	32 KB split, 2-way set-associative
L2 Cache	256 KB unified, 2-way set-associative
Memory Capacity	128 MB per compute node, 256 MB per I/O node
Network	100 Mbs Ethernet, 155 Mbs ATM and 800 Mbs HiPPI
Disk Space	9 GB per I/O node
Operating System	AIX 4.2.1
Parallel File System	PIOFS

6 Experiments

In this section, we discuss the experimental environment and discuss our preliminary results. We report experimental data for both synthetic access patterns and a large-scale scientific code.

6.1 Experimental Environment

We used the MPI-2 library [17] and an IBM SP-2 in Argonne National Laboratories to evaluate our scheme proposed in this section. The important characteristics of our experimental platform are shown in Table 3.

The IBM SP-2 used in the experiments has 128 processors, 8 of which are I/O processors. Each I/O server is attached to a 9 GB SSA disk, resulting in 72 GB of total disk space. The operating system on each node is AIX 4.2.1. PIOFS provides the parallel access to files. It distributes a file across multiple I/O server nodes.

The MCIO calls are similar to MPI-IO [5] calls. For example, to perform a read operation, the processors call the

```
int MCIO_File_read_all (MPI_File *fh,
int filecount, void **buf, int *count,
MPI_Datatype *datatype, MPI_Status
*status)
```

routine. Note that the syntax of the call is very similar to `MPI_File_read_all` call in MPI-IO, except that MCIO routine takes an array of files (similarly, an array of buffers, an array of number of elements, and an array of data types) as argument. In addition, it takes an integer argument `filecount` indicating the number of files involved in the I/O request. Each array element corresponds to a request from a different file. Without the MCIO, calls to all the different files would have resulted in a separate MPI-IO call. In all experiments, we compare the performance of MCIO with that of traditional collective I/O.

6.2 Results for Synthetic Patterns

Figure 9 gives examples of the access patterns we experiment with. Two major types of experiments are conducted: row-major access and column-major access. In a row-major access, each processor accesses consecutive rows of the underlying data. In a column-major access, the array is distributed column-wise among the processors (i.e., each processor accesses a group of consecutive columns). For each category, we experiment with different number of processors and files.

For all the access patterns we experimented with, we evaluated the assignments resulting from the LP-model, greedy heuristic, and the maximal matching heuristic. The objective function and the constraints for the access in Figure 4 are given in Appendix A. The assignments for all the three methods were the same. This is mainly because of the constant file size we have used in the experiments. Due to this invariance of the results, we do not present separate execution times for these methods, but discuss the advantages and disadvantages of each. Note that, the assignments of greedy algorithm and the maximal matching heuristic will differ only in their communication time, not the I/O time. Therefore, there is some difference between the response times of the assignments made by the two heuristic. The greedy algorithm is faster to find the assignments than the maximal matching algorithm. Hence, in cases where the access sizes from different processors to different files are the same (similar to the access patterns in Figure 9), we recommend the use of the greedy algorithm. On the other hand, if the request sizes from different files have a significant difference, the maximal matching algorithm always gives a better communication time. Also, if the total amount of data requested by differ-

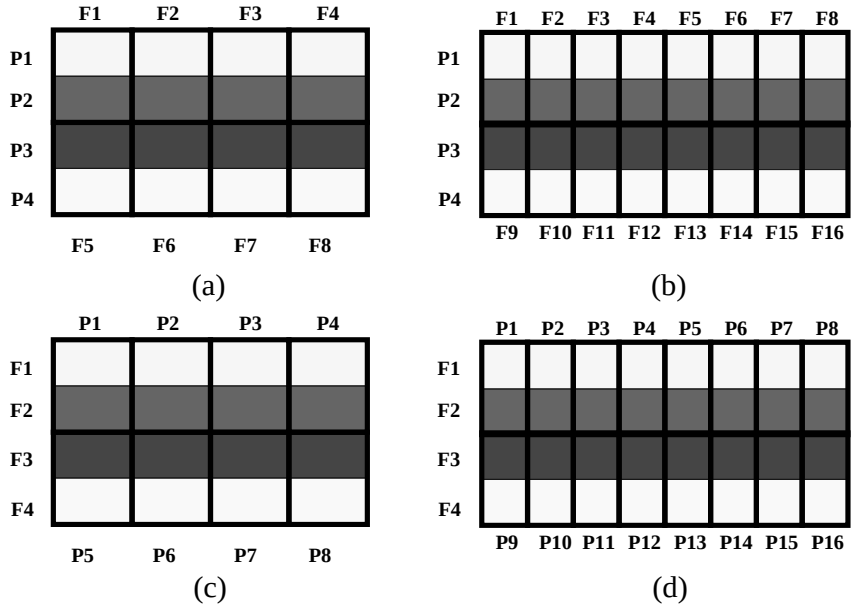

Figure 9: Examples of the experimented access patterns: (a) Four processor accessing data row-wise from 8 files, (b) Four processor accessing data row-wise from 16 files, (c) 8 processor accessing data column-wise from four files, (d) 16 processor accessing data column-wise from four files.

Table 4: The improvements for row major accesses (Figure 9(a) and (b)) in [%] with respect to the naive I/O access.

Number of Files	Number of Processors		
	4	8	16
4	49.97	51.22	49.97
8	52.28	78.33	77.41
16	53.42	80.12	86.67
32	54.33	80.52	87.18

Table 5: The improvements for column major accesses (Figure 9(c) and (d)) in [%] with respect to the naive I/O access.

Number of Files	Number of Processors		
	4	8	16
4	41.26	55.76	49.04
8	43.90	76.49	79.26
16	45.21	77.59	80.28
32	45.84	78.07	80.82

ent processors have a large variance, maximal matching algorithm gives better performance. Hence, in such cases maximal matching algorithm should be employed.

In the first set of experiments conducted, the file size is set to 32 MB, representing a two-dimensional matrix of 1024×2048 floating points for the case of four processors reading data from four files. Therefore, the total amount of data read in this case equals to 128 MB. When the number of files is increased, the total amount of data accessed increases linearly.

6.2.1 Row-Major Accesses

The results for the experiments with row-wise access patterns are summarized in Table 4. The table gives the improvements achieved by MCIO technique compared to a naive access using the CIO technique for each accessed file. The MCIO is able to improve the response time in the base case (4 processors reading data from 4 files) by 49.97% over a naive access pattern, which performs CIO for each of the 4 files separately. The results also reveal that as the number of processors increases, the improvement also increases. In addition, when the number of files is increased, the improvement increases. An exception

occurs when the number of processors is increased from 8 to 16 when 4 files are accessed. The reason for a reduction in the improvement is the small parallelism available for this access. Note that, when 16 processors are accessing 4 files, each file is assigned to 4 processors. This increases the synchronization cost of the access and reduces the advantages of MCIO, which utilizes the available parallelism. Similarly, with 16 processors, there is an insignificant reduction in the improvement when the number of files is increased from 8 to 16. These results indicate that MCIO brings significant improvement even in the case where the number of processors is less than the number of files.

6.2.2 Column-Major Accesses

Table 5 summarizes the experimental results for column-wise access patterns. Although, the performance improvement by MCIO is slightly less for column-major accesses, MCIO still brings substantial amount of improvement of the naive CIO technique. Specifically, MCIO is able to improve the CIO performance by as much as 80.82% when 16 processors are requesting data from 32 files. Similar to row-major accesses, MCIO brings better improvement over the naive CIO performance when the number of processors or files are increased. The case where the number of processors is increased from 8 to 16 for reading 4 files is again an exception to the general trend.

6.3 Results for a Scientific Application

We have also applied the MCIO technique to improve the I/O performance of the *astro3d* [15] application. Table 6 shows the results for this three-dimensional astrophysics application. *Astro3d* accesses six different variables from a single file. In the original application, this corresponds to six different collective I/O calls. Using MCIO, same data can be accessed using a single call, thereby the parallelism in the system is better utilized. For 4 processors, this results in a reduction of the I/O time by 38.52%. For 8 processors, the improvement of MCIO over traditional collective I/O increases to 62.96%.

Table 6: Total I/O times (in seconds) for *astro3d* application (Data set size is 8 MB).

	4 processors	8 processors
Collective I/O	3.33	3.51
MCIO	2.04	1.30

To summarize, these results show that the MCIO strategy brings significant amount of improvement over a naive CIO access method. Specifically, we are able to improve the response time by as much as 87.18%.

7 Conclusions and Future Work

In this paper, we have introduced an I/O optimization technique called multi-collective I/O. As the gap between the performance of the processor and the storage subsystem increases, more aggressive optimizations are needed to be able to feed the processor with enough data. Several scientific applications exhibit poor storage access patterns, and hence optimizations like MCIO can bring significant improvement in the execution time of such applications.

We have first shown that finding the optimal access pattern in MCIO is an NP-complete problem. Then, we have presented two heuristics to perform this task: a greedy algorithm that uses sorting and a graph algorithm that uses a matching problem solver. Then, using synthetic benchmarks and a scientific application, we have shown that MCIO can bring substantial amount of improvement in the I/O response time over a collective I/O technique. Specifically, MCIO was able to improve the response time by as much as 87.18%.

Our current work focuses on using the scheduling obtained through our approach in developing powerful I/O prefetching techniques. Also in our agenda is evaluating the effectiveness of MCIO in a sub-file based environment. We also plan to design and implement an optimizing compiler framework for generating I/O-optimized code automatically using the MCIO interface provided by our runtime library. Such a compiler will relieve applica-

tion developers from low-level details of file systems and runtime libraries, and let them focus instead on high-level (application-specific) aspects of their codes.

References

[1] Bagrodia, R., Chien, A., Hsu, Y., Reed, D. Input/output: Instrumentation, Characterization, Modeling and Management Policy. *Tech. Rep. CCSF-41*, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[2] Baru, S. *Storage Resource Broker (SRB) Reference Manual*. Enabling Technologies Group, San Diego Supercomputer Center, La Jolla, CA, August 1997.

[3] Beynon, M., Kurc, T., Sussman, A., Saltz, J. Design of a Framework for Data-Intensive Wide-Area Applications. In *Proc. of the 9th Heterogeneous Computing Workshop (HCW2000)*, Cancun, Mexico, May 2000.

[4] Choudhary, A., Bordawekar, R., Harry, M., Krishnaiyer, R., Ponnusamy, R., Singh, T., and Thakur, R. PASSION: parallel and scalable software for input-output. *NPAC Technical Report SCCS-636*, Sept 1994.

[5] Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J., Snir, M., Traversat, B., and Wong, P. Overview of the MPI-IO parallel I/O interface, In *Proc. Third Workshop on I/O in Parallel and Distributed Systems*, IPPS'95, Santa Barbara, CA, April 1995.

[6] Corbett, P., Feitelson, D., Prost, J-P., Almasi, G., Baylor, S. J., Bolmarcich, A., Hsu, Y., Satran, J., Snir, M., Colao, R., Herr, B., Kavaky, J., Morgan, T., and Zlotek, A. Parallel file systems for the IBM SP computers, *IBM Systems Journal*, Vol. 34, No. 2, pp. 222–248, Jan 1995.

[7] Coyne, R. A., Hulen, H., and Watson R.. The high performance storage system. In *Proc. Supercomputing 93*, Portland, OR, November 1993.

[8] Crandall, P., Aydt, R., Chien, A., Reed, D. Input/output characteristics of scalable parallel applications. In *Proc. of Supercomputing'95*, Dec. 1995.

[9] Cypher, R., Ho, A., Konstantinidou, S., Messina, P. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *Proc. of the 20th International Symposium on Computer Architecture*, 1993, pp. 2-13.

[10] Gabow, H. Implementation of Algorithms for Maximum Matching on Non-bipartite Graphs. *Ph.D Thesis*, Stanford University, 1973.

[11] Garey, M. R. and Johnson, D. S. Computers and Intractability, *A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[12] Karpovich, J. F., Grimshaw, A. S., and French, J. C. Extensible file systems (ELFS): An object-oriented approach to high performance file I/O. In *Proc. the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 191–204, Oct 1994.

[13] Kotz, D. Disk-directed I/O for MIMD multiprocessors. In *Proc. the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth *TR PCS-TR94-226* on November 8, 1994.

[14] Kotz, D. Multiprocessor file system interfaces. In *Proc. of the Second International Conference on Parallel and Distributed Information Systems*, pp. 194-201, 1993.

[15] Malagoli, A. ASTRO3D - 2.0. The ASCI Flash Center at Chicago, IL, Sep. 1998.

[16] Memik, G., Kandemir, M. T., Choudhary, A. APRIL: A Run-Time Library for Tape Resident Data. In *Proc. of 8.NASA Goddard Conference on Mass Storage Systems and Technologies*, Baltimore, MD, April 2000.

[17] Message Passing Interface Forum. MPI-2: Extension to the message passing interface. *Technical report*, University of Tennessee, July 1997. URL: http://www.mpi-forum.org.

[18] Nieuwejaar, N., Kotz, D., Purakayastha, A., Ellis, C., Best, M. File-Access Characteristics of Parallel Scientific Workloads. In *IEEE Transactions on Parallel and Distributed Systems*, October 1996 (Vol. 7, No. 10), pp. 1075-1089.

[19] Nitzberg, B. J. Collective Parallel I/O. *PhD thesis*, Department of Computer and Information Science, University of Oregon, December 1995.

[20] Nitzberg, B. Performance of the iPSC/860 Concurrent File System. *Technical Report RND-92-020*, NAS Systems Division, NASA Ames, Dec. 1992.

[21] Rosario, J., Bordawekar, R., Choudhary, A. Improved parallel I/O via a two-phase run-time access strategy. In *Proc. the 1993 IPPS Workshop on Input/Output in Paral. Comp. Sys.*, April 1993.

[22] Seamons, K. E., Chen, Y., Jones, P., Jozwiak, J., Winslett, M. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing'95*, December 1995

[23] Seamons, K. E., Winslett, M. Efficient sub-image extraction using Panda. URL: http://drl.cs.uiuc.edu/panda/demo/

[24] Shen, X., Liao, W., Choudhary, A., Memik, G., Kandemir, M., More, S., Thiruvathukal, G., and Singh, A. A Novel Application Development Environment for Large-Scale Scientific Computations. In *Proc. of International Conference on Supercomputing*, Santa Fe, New Mexico, May, 2000.

[25] URL: ftp://dimacs.rutgers.edu/pub/netflow/

[26] Thakur, R., Lusk, E., Gropp, W. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. *Technical Memorandum ANL/MCS-TM-234*, Mathematics and Computer Science Division, Argonne National Laboratory, IL, Revised July 1998.

APPENDIX A *An Example LP Model*

Ee give the LP model for the access in Figure 4 and also present the result of the solver. We have used the CPLEX as the LP-solver, hence the model and the results are given using the CPLEX format. The LP-model is as follows:

```
minimize
M
subject to
c1:  x1_1 + x2_1 + x3_1 + x4_1 = 1
c2:  x1_2 + x2_2 + x3_2 + x4_2 = 1
c3:  x1_3 + x2_3 + x3_3 + x4_3 = 1
c4:  x1_4 + x2_4 + x3_4 + x4_4 = 1
c5:  x1_5 + x2_5 + x3_5 + x4_5 = 1
c6:  x1_6 + x2_6 + x3_6 + x4_6 = 1
c7:  x1_7 + x2_7 + x3_7 + x4_7 = 1
c8:  x1_8 + x2_8 + x3_8 + x4_8 = 1
c9:  x1_1 + x1_2 + x1_3 + x1_4 + x1_5 +
x1_6 + x1_7 +x1_8 = 2
c10:  x2_1 + x2_2 + x2_3 + x2_4 + x2_5 +
x2_6 + x2_7 + x2_8 = 2
c11:  x3_1 + x3_2 + x3_3 + x3_4 + x3_5 +
x3_6 + x3_7 + x3_8 = 2
c12:  x4_1 + x4_2 + x4_3 + x4_4 + x4_5 +
x4_6 + x4_7 + x4_8 = 2
c13:  M + 10x1_1 + 12x1_2 + 12x1_3 +
5x1_4 + 10x2_1 + 12x2_2 + 12x2_3 + 5x2_4
+ 2x3_1 + 3x3_2 + 3x3_3 + 2x3_4 + 7x3_5
+ 9x3_6 + 9x3_7 + 3x3_8 + 10x4_5 +
12x4_6 + 12x4_7 + 5x4_8 = 155
binary
x1_1 x1_2 x1_3 x1_4 x1_5 x1_6 x1_7 x1_8
x2_1 x2_2 x2_3 x2_4 x2_5 x2_6 x2_7 x2_8
x3_1 x3_2 x3_3 x3_4 x3_5 x3_6 x3_7 x3_8
x4_1 x4_2 x4_3 x4_4 x4_5 x4_6 x4_7 x1_8
end
```

Note that, we have rounded the request sizes (`c13`) from Table 1. In the above model, $x_{i,j}$'s are denoted by xi_j and M is the objective function. The first 8 constraints correspond to the $\sum_{i=1}^{p} x_{i,j} = 1$ constraint in Section 4.3. Constraints 9 through 12 correspond to the $\sum_{j=1}^{n} x_{i,j} = \frac{n}{p}$ constraint which guarantees homogeneous distribution of the files among processors.

The result for the above model is as follows:

```
 Variable Name  Solution Value
 M              82.000000
 x2_1           1.000000
 x2_2           1.000000
 x1_3           1.000000
 x1_4           1.000000
 x3_5           1.000000
 x4_6           1.000000
 x4_7           1.000000
 x3_8           1.000000
All other variables in the range 1-33
are zero.
```

The value of 82 for `M` is the optimal communication that can be achieved. The variables $x_{2,1}$, $x_{2,2}$, $x_{1,3}$, $x_{1,4}$, $x_{3,5}$, $x_{4,6}$, $x_{4,7}$, and $x_{3,8}$ are 1. The remaining variables are zero. Note that, this assignment is the same as the maximal matching heuristic has made, which is shown in Figure 7.