Design and Evaluation of a Compiler-Directed Collective I/O Technique*

Gokhan Memik[1], Mahmut T. Kandemir[2], and Alok Choudhary[1]

[1] Department of Electrical and Computer Eng.
Northwestern University, Evanston IL 60208, USA
[2] Dept. of Computer Science and Engineering,
Pennsylvania State University, University Park PA 16802, USA

Abstract. Current approaches to parallel I/O demand extensive user effort to obtain acceptable performance. This is in part due to difficulties in understanding the characteristics of a wide variety of I/O devices and in part due to inherent complexity of I/O software. While parallel I/O systems provide users with environments where large datasets can be shared between parallel processors, the ultimate performance of I/O-intensive codes depends largely on the relation between data access patterns and storage patterns of data in files and on disks. Collective I/O is one of the most popular methods to access the data when the storage and access patterns do not match. In this strategy, each processor does I/O on behalf of other processors if doing so improves the overall performance. While it is generally accepted that collective I/O and its variants can bring impressive improvements as far as the I/O performance is concerned, it is difficult for the programmer to use collective I/O effectively. In this paper, we propose and evaluate a compiler-directed collective I/O approach which detects the opportunities for collective I/O and inserts the necessary I/O calls in the code automatically. An important characteristic of the approach is that instead of applying collective I/O indiscriminately, it uses collective I/O selectively, only in cases where independent parallel I/O would not be possible. We have conducted several experiments using an IBM SP-2 distributed-memory message-passing machine with 128 nodes. Our compiler directed collective I/O scheme was able to perform 18% better in average than an indiscriminate collective I/O scheme in our base configuration.

1 Introduction

Todays' parallel architectures comprise fast microprocessors, powerful network interfaces, and storage hierarchies that typically have multi-level caches, local and remote main memories, and secondary and tertiary storage devices. In going from upper levels of a storage hierarchy to lower levels, average access times

* This work is supported by the Department of Energy's Accelerated Strategic Computing Initiative (ASCI) program under a subcontract No W-7405-ENG-48 from Lawarence Livermore National Laboratories and by NSF CDA-9703228 and NSF ACI-9707074.

increase dramatically. Because of their cost effectiveness, magnetic disks have dominated the secondary storage market for the last several decades. Unfortunately, their access times have not kept pace with performance of the processors used in parallel architectures. Consequently, a large performance gap between secondary storage access times and processing unit speeds has emerged.

To address this imbalance, hardware designers focus on improving parallel I/O capabilities using multiple disks, I/O processors, and large bandwidth I/O busses [6]. An optimized I/O software can also play a major role in bridging this performance gap. In order to eliminate the difficulty in using a parallel file system directly, several research groups proposed high-level parallel I/O libraries and runtime systems that allow programmers to express access patterns of their codes using program-level data structures such as rectilinear array regions [4,13,5]. While all these software supports provide an invaluable help to boost the I/O performance in parallel architectures, it remains still programmer's responsibility to select appropriate I/O calls to use, to insert these calls in appropriate locations within the code, and to manage the data flow between parallel processors and parallel disks.

One of the most important optimizations in MPI-IO [5] is *collective I/O,* an optimization that allows each processor to do I/O on behalf of other processors [4]. This optimization has many variants [12,11,13]; the one used in this study is *two-phase I/O.* In this implementation, I/O is performed in *two phases*: an I/O phase and a communication phase. In the I/O phase, processors perform I/O in a way that is most beneficial from the storage layout point of view. In the second phase, they engage in a many-to-many type of communication to ensure that each piece of data arrives in its final destination. While collective I/O and its variants are very beneficial if used properly, almost all previous studies considered a *user-oriented* approach in applying collective I/O. For example, Thakur et al. suggest programmers to use collective I/O interfaces of MPI-IO instead of easy-to-use Unix-like interfaces [14]. Apart from determining the most suitable collective I/O routine and its corresponding parameters, this also requires, on the user part, analyzing access patterns of the code, detecting parallel I/O opportunities, and finally deciding a parallel I/O strategy.

In this paper, we propose and evaluate a *compiler-directed* collective I/O strategy whereby an optimizing compiler and MPI-IO cooperate to improve I/O performance of scientific codes. The compiler's responsibility in this work is to analyze the data access patterns of individual applications and determine suitable file storage patterns and I/O strategies. Our approach is *selective* because it activates collective I/O selectively, only when necessary. In other cases, it ensures that processors perform *independent parallel I/O,* which has almost the same I/O performance as collective I/O but without extra communication overhead.

The remainder of this paper is organized as follows. In the next section, we review collective I/O. In Section 3, we explain our compiler analyses to detect access patterns and suitable storage patterns for multidimensional datasets considering multiple, related applications together. In Section 4, we describe our experimental framework, our benchmarks, and different code versions, and present our experimental results. In Section 5, we present our conclusions.

2 Collective I/O

In many I/O-intensive applications that access large, multidimensional, disk-resident datasets, the performance of I/O accesses depends largely on the layout of data in files (*storage pattern*) and distribution of data across processors (*access pattern*). In cases where these patterns are the same, potentially, each processor can perform *independent parallel I/O.* However, the term 'independent parallel I/O' might be misleading, as, depending on the I/O network bandwidth, the number of parallel disks available, and the data striping strategies employed by the parallel file system, two processors may experience a conflict in accessing different data pieces residing on the same disk [6]. What we mean by 'independent parallel I/O' instead is that the processors can read/write their portions of the dataset (dictated by the access pattern) using only a few I/O requests *in the code,* each for a large number consecutive data items in a file. These independent source-level I/O calls to files are broken up into several system-level calls to parallel disks. This last aspect, however, is architecture and operating system dependent and is not investigated in this paper. Note that, in independent parallel I/O, there is *no* interprocessor communication or synchronization during I/O.

In cases where storage and access patterns do not match, allowing each processor to perform independent I/O will cause processors to issue many I/O requests, each for a small amount of consecutive data. In this paper, an access pattern which is the same as the corresponding storage pattern is called a *conforming* access pattern. Collective I/O can improve the performance in non-conforming cases by first reading the dataset in question in a conforming (storage layout friendly) manner and then redistributing the data among the processors 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. The idea is that the communication cost is typically small as compared to I/O cost, meaning that the cost of accessing a dataset becomes almost independent from its storage pattern.

Consider Figure 1 that shows both independent parallel I/O and collective I/O for a four processor case using a single disk-resident two-dimensional dataset. In Figure 1(a), the storage pattern is row-major (each circle represents an array element and the arrows denote the linearized file layout of elements) and the access pattern is row-wise (i.e., each of the four processors accesses two full-rows of the dataset). Since the access pattern and the storage pattern match, each processor can perform independent parallel I/O without any need of communication or synchronization. Figure 1(b), on the other hand, shows the case where collective I/O is required. The reason is that in this figure the storage pattern is row-major and the access pattern does not match it. As explained earlier, the I/O is performed in two phases. In the first phase, each processor accesses the data row-wise, as if this was the original access pattern), and in the second step, an all-to-all communication is performed between the processors and each data item is delivered to its final destination.

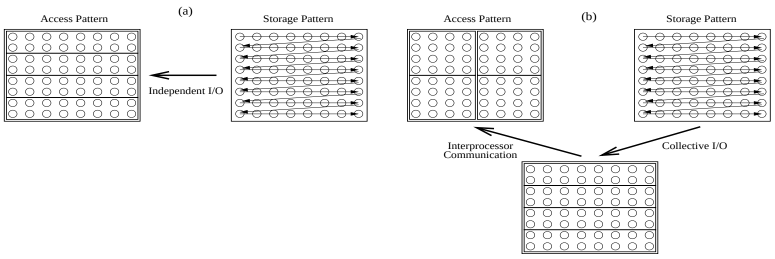

Fig. 1. (a) Independent parallel I/O and (b) Collective (two-phase) I/O.

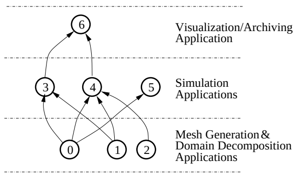
6
3
4
5
0
1
2
Visualization/Archiving Application
Simulation Applications
Mesh Generation & Domain Decomposition Applications

Fig. 2. Scientific working environment.

3 Compiler Analysis

Our approach to collective I/O utilizes a directed graph called *weighted communication graph* (WCG). Each node of a weighted communication graph is a *code block*, which can be defined as a program fragment during which we can keep the datasets in memory; however, between executions of code blocks, the datasets should be stored on disks. Depending on the applications, the datasets in question, and the available memory, a code block can be as small as a loop nest or can be as large as a full-scale application. An example for the latter is shown in Figure 2 that depicts a typical scenario from a scientific working environment. There is a directed edge, $e_{1,2}$, between two nodes, cd_1 and cd_2, of the WCG if and only if there exists at least a dataset that is produced (i.e., created and stored on disk) in cd_1 and used (i.e., read from disk) in cd_2. In such a case cd_1 is called *producer* and cd_2 is called *consumer*. The *weight* associated with $e_{1,2}$ (written as $w_{1,2}$) corresponds to total number of dynamic control-flow transitions between code blocks cd_1 and cd_2 (e.g., how many times cd_2 is run after cd_1 in a typical setup). Depending on the granularity of code blocks, these weights can be calculated using profiling with typical input sets, can be approximated using weight estimation techniques, or can be entered by a user who observed the scientific working environment for a sufficiently long period of time.

3.1 Access Pattern Detection

Access patterns exhibited by each code block can be determined by considering individual loop nests that make up the code block. The crucial step in this process is taking into account the parallelization information [2]. Individual nests can either be parallelized explicitly by programmers using compiler directives [1,3], or can be parallelized automatically (without user intervention) as a result of intra-procedural and inter-procedural compiler analyses [2,8,9]. In either case, after the parallelization step, our approach determines the data regions (for a given dataset) accessed by each processor involved.

For each array reference in each loop nest, our compiler determines an access pattern. Afterwards, it utilizes a *conflict resolution scheme* to resolve intra-nest and inter-nest access pattern conflicts. To achieve a reasonable conflict resolution, we associate a *count* with each reference indicating (or approximating) the number of times that this reference is touched in a typical execution. In addition to that, for each access pattern that exists in the code block, we associate a *counter* that is initialized to zero and incremented by a count amount each time we encounter a reference with that access pattern. In this way, for a given array, we determine the most preferable (or most prevalent) access pattern (also called *representative access pattern*) and mark the code block with that information. Although, at first glance, it seems that, in a typical large-scale application, there will be a lot of conflicting patterns that would make compiler's job of favoring one of them over the others difficult, in reality, most scientific codes have a few preferable access patterns.

3.2 Storage Pattern Detection

Having determined an access pattern for each disk-resident dataset, the next step is to select a suitable storage pattern for each dataset in its producer code block. We have built a prototype tool to achieve this.[1]

For a given dataset, the tool takes the representative access patterns detected in the previous step by the compiler for each code block and runs a storage layout detection algorithm. Without loss of generality, in the following discussion, we focus only on a single dataset. The first step in our approach is to determine *producer-consumer subgraphs* (PCSs) of WCG for the dataset in question. A PCS for a dataset consists of a producer node and a number of consumer nodes that use the data produced by this producer. In the second step, we associate a count

[1] Note that building a separate tool is necessary only if the granularity of code blocks is a full-application. If, on the other hand, the granularity is a single nested-loop or a procedure, the functionality of this tool can be embedded within the compiler framework itself.

with each possible access pattern and initialize it to zero. Then, we traverse all the consumer nodes in turn and for each consumer node add its weight to the count of its access pattern. At the end of this step, for each access pattern, we obtain a count value. In the third step, we set the storage pattern in the producer node to the access pattern with the highest count. Note that, for a given dataset, we need to run the storage pattern detection algorithm multiple times, *one for each producer node* for this dataset.

The next step is to determine suitable I/O strategies for each consumer node. Let us again focus on a specific dataset. If the access pattern (for this dataset) of a consumer node is the same as the storage pattern in the producer node, we perform independent parallel I/O in this consumer node. Otherwise, that is, if the access and storage patterns are different, we perform collective I/O. We perform this step for each dataset and each PCS. Once the suitable I/O strategies have been determined, the compiler automatically inserts corresponding MPI-IO calls in each code block.

3.3 Discussion

Although, our approach is so far mainly discussed for a setting where individual code blocks correspond to individual applications, it is relatively easy to adapt it to different settings as well. If we consider each code block as a procedure in a given application, then a WCG can be processed using algorithms similar to those utilized in processing *weighted call graphs* [7], where each node represents a procedure and an edge between two nodes correspond to dynamic control-flow transitions (e.g., procedure calls and returns) between the procedures represented by these two nodes. In an out-of-core environment, on the other hand, each node may represent an individual loop nest and edges might represent dynamic control-flow between nests; in this case, the WCG is similar to a control-flow graph.

Another important issue that needs to be addressed is what to do (inside a code block) when we come across a reference whose access pattern is not the same as the representative (prevalent) access pattern for this code block. Recall that we assumed that, within a code block, we should be able to keep the datasets in memory. When the access pattern of an individual reference is different from the representative access pattern determined for a code block, we can re-shuffle the data in memory. This is not a correctness issue for shared-memory machines but it may cause performance degradation. In distributed-memory message-passing architectures, on the other hand, this data re-shuffling in memory is necessary to ensure the correct data accesses in subsequent computations.

4 Experiments

In this section, we first describe the experimental environment. Afterwards, we explain the setups for the experiments. Then, the results for the base configuration is given. Finally, we give results for different number of processors and data sizes.

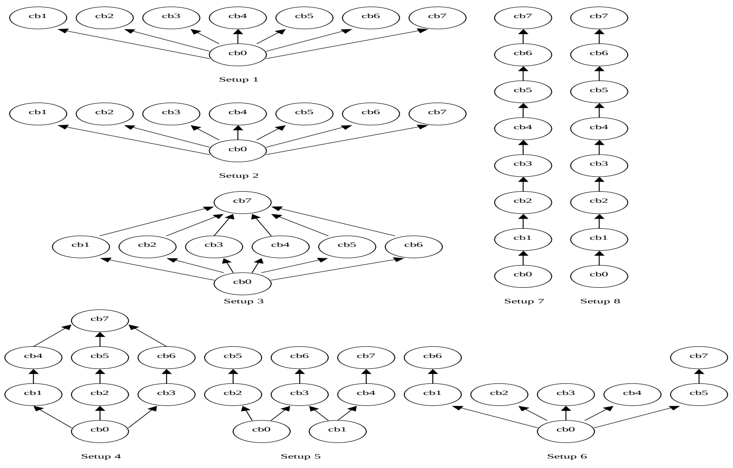

Fig. 3. Setups (communication graphs) used in the experiments.

4.1 Experimental Environment

We used the MPI-2 library and an IBM SP-2 in Argonne National Laboratories to evaluate our scheme proposed in this paper. The IBM SP-2 used in the experiments has 128 processors, 8 of which are I/O processors. The compute nodes are RS/6000 Model 370 processors with 128 MB memory, whereas the I/O nodes are RS/6000 Model 970 processors with 256 MB memory. The nodes are connected via 100 Mbs Ethernet, 155 Mbs ATM and 800 Mbs HiPPI networks. Each I/O server has a 9 GB of storage space 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.

4.2 Setups

To evaluate the possible improvements with our scheme, we have designed 8 different *setups* (*communication graphs*), each built up using 8 different benchmark codes in different ways. We have selected 4 benchmarks from Specfp (tomcatv (cb_0), vpenta (cb_1), btrix (cb_2), and mxm (cb_3)), 2 codes from the Perfect Club benchmark suite (tis (cb_4) and eflux (cb_5)), 1 from Nwchem suite (transpose (cb_6)) and a miscellaneous code (cholesky (cb_7)). The setups built from these benchmarks are given in Figure 3.[2]

[2] Although Setup 1 and Setup 2 look the same, they differ in the access and storage patterns they employ for different benchmarks. Similarly, Setup 7 and Setup 8 differ in their access and storage patterns. The details are omitted due to lack of space.

🟦 Naïve Strategy 🟪 Collective I/O 🟨 Compiler-directed Collective I/O

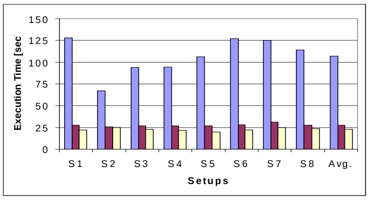

Figure 6: Execution times for the base experiments

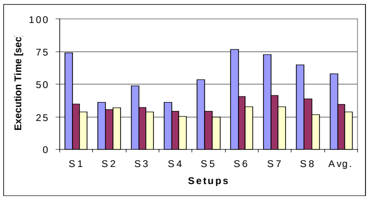

Figure 7: Execution times for 4 processors

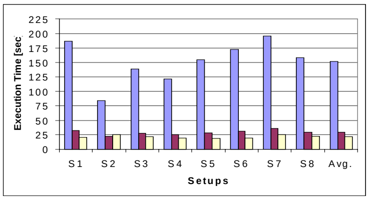

Figure 8: Execution times for 16 processors

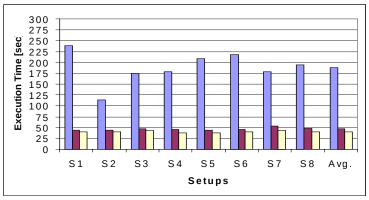

Figure 9: Execution times for larger data size (Data size is doubled)

Three bars for each setup represents the following access strategies, from left to right: naïve strategy (no collective I/O), indiscriminate collective I/O, compiler directed collective I/O

4.3 Base Experiments

For the base experiments, we have executed the setup explained in Section 4.2 using 8 processors of the IBM SP-2. In our experiments we are comparing the execution times of three different code versions as explained below.

Version 1 In this version, each processor performs independent (non-collective) I/O regardless whether the access pattern and the storage pattern are the same or not. We call this version the *naive I/O strategy.*

Version 2 This version performs indiscriminate collective I/O.

Version 3 This is the strategy explained in this paper. The collective I/O is performed selectively, only if the access pattern and the storage pattern do not match. In all other cases, we perform independent parallel I/O.

In Figures 6 through 9, the left-most bar represents the total execution time of version 1, the middle bar represents the total execution time of version 2 and the right-most bar represents the total execution time of version 3.

The results for 8 processors are given in Figure 6. The average improvement over the indiscriminate collective I/O strategy is 18.01%. For setups 5 and 6, we are able to gain more than 21% over the version 2, which performs indiscriminate collective I/O. These two setups give the best results, because a change of the storage pattern effects the most applications. For example, when *cb*0 in Setup 1 changes its storage pattern, the weights of the favoring applications add up to 50% of the sum of all weights, whereas in Setup 6, the sum of weights of the favoring applications constitutes 60% of all the weights. So, there are 10% more favoring applications in Setup 6. Therefore, the improvement of Setup 6 is more than Setup 1 with our scheme.

Note that, both the indiscriminate and selective collective I/O strategies perform well compared to a naive strategy, which does not use collective I/O at all. The improvement of indiscriminate collective I/O is 74.19% over the naive strategy, whereas our scheme brings 78.84% improvement.

4.4 Sensitivity Analysis

We have performed a second set of experiments to see how our strategy is affected by the number of processors and the data size. Figures 7 and 8 give the results for 4 and 16 processors, respectively. For 4 processors, our scheme brings 16.42% improvement over the indiscriminate collective I/O version, and 40.23% over the naive strategy. For 16 processors, on the other hand, our scheme brings an 26.27% improvement over the indiscriminate collective I/O version, and 80.74% over the naive strategy. As the number of processors increase, the improvement of our scheme increases, because with larger number of processors, the synchronization and communication costs increase.

Figure 9 gives the results for a larger data size. For this experiment, we have doubled the size of the input and/or output data of all the benchmarks. When the data size is increased, the synchronization overhead is reduced. Similarly, communication and I/O can be better overlapped because the I/O calls are longer, so the overall communication cost also reduces. These factors decrease the percentage improvements of our scheme, but, it still performs the best by far. It brings a 13.57% improvement over the indiscriminate collective I/O version, and 75.21% improvement over the naive strategy.

5 Conclusions

In this paper, we present and evaluate a compiler-directed collective I/O strategy. Collective I/O plays a major role in parallel I/O systems. Therefore, increasing its performance is very important for many of data-intensive parallel applications. By adopting a selective collective I/O strategy, we are able to bring significant amounts of improvements. In average, our scheme performs 18.01% better than an indiscriminate collective I/O strategy in our base configuration. The scheme performs better as the number of processors increases. Although the improvement decreases with the increased data size, the scheme is still able to perform more than 13% better than an indiscriminate collective I/O strategy.

The interface for parallel I/O systems are usually complex, and they are getting more complex, because the information required by the I/O calls increases. So, it becomes harder for an average user to detect the best possible I/O call for an application. Therefore, detecting the best possible storage and access patterns automatically is very useful for many programmers and increases the performance of I/O-intensive applications significantly.

References

1. A. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. *Scientific Prog.,* 6(1):3–28, Spring 1997.

2. J. Anderson. *Automatic Computation and Data Decomposition for Multiprocessors.* Ph.D. dissertation, Computer Systems Lab., Stanford Univ., March 1997.
3. R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedeljkovic, and J. Anderson. Data-distribution support on distributed-shared memory multi-processors. In *Proc. Prog. Lang. Design and Implementation*, Las Vegas, NV, 1997.
4. A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. Passion: Parallel and scalable software for input-output. *NPAC Technical Report SCCS-636,* Sept 1994.
5. P. Corbett et al. Overview of the MPI-IO parallel I/O interface, In *Proc. Third Workshop on I/O in Par. and Dist. Sys.*, IPPS'95, Santa Barbara, CA, April 1995.
6. J. del Rosario and A. Choudhary. High performance I/O for parallel computers: problems and prospects. *IEEE Computer,* March 1994.
7. N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *Proc. Micro-30,* Research Triangle Park, North Carolina, December 1–3, 1997.
8. M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.
9. M. W. Hall, B. Murphy, S. Amarasinghe, S. Liao, and M. Lam. Inter-procedural analysis for parallelization. In *Proc. 8th International Workshop on Lang. and Comp. for Parallel Computers*, pages 61–80, Columbus, Ohio, August 1995.
10. W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, CS Dept., University of Maryland, College Park, March 1995.
11. D. Kotz. 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.
12. B. J. Nitzberg. *Collective Parallel I/O.* PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995.
13. K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing'95,* December 1995.
14. R. Thakur, W. Gropp, and E. Lusk. A case for using MPI's derived data types to improve I/O performance. In *Proc. of SC'98: High Performance Networking and Computing,* November 1998.