Appears in the International Conference on Parallel Processing 2003

Scalable Implementations of MPI Atomicity for Concurrent Overlapping I/O

Wei-keng Liao[†], Alok Choudhary[†], Kenin Coloma[†], George K. Thiruvathukal[‡], Lee Ward[*], Eric Russell[*], and Neil Pundit[*]

[†] ECE Department
Northwestern University

[‡] CS Department
Loyola University

[*] Scalable Computing Systems Department
Sandia National Laboratories

Abstract

For concurrent I/O operations, atomicity defines the results in the overlapping file regions simultaneously read/written by requesting processes. Atomicity has been well studied at the file system level, such as POSIX standard. In this paper, we investigate the problems arising from the implementation of MPI atomicity for concurrent overlapping write access and provide two programming solutions. Since the MPI definition of atomicity differs from the POSIX one, an implementation that simply relies on the POSIX file systems does not guarantee correct MPI semantics. To have a correct implementation of atomic I/O in MPI, we examine the efficiency of three approaches: 1) file locking, 2) graph-coloring, and 3) process-rank ordering. Performance complexity for these methods are analyzed and their experimental results are presented for file systems including NFS, SGI's XFS, and IBM's GPFS.

1. Introduction

Concurrent file access has been an active research topic for many years. Efforts were contributed in both software development as well as hardware design to improve the I/O bandwidth between computational units and storage systems. While most of these works only consider exclusive file access among the concurrent I/O requests, more scientific applications nowadays require data partitioning with overlap among the requesting processes [1, 6, 9, 10]. For instance, ghost cells are commonly used in multi-dimensional array partitioning such that the sub-array partitioned in one process overlaps with its neighbors near the boundary. A couple of examples that use this ghosting technique are large scale simulations in earth climate and N-body astrophysics, hydrodynamics using Laplace equations, both where a strong spatial domain partitioning relationship is present. Figure 1 illustrates an example of a two-dimensional array in a block-block partitioning pattern in which a ghost cell represents data "owned" by more than one process. A typical run of this large-scale type of applications can take from days to months and usually output data periodically for the purposes of check-pointing as well as progressive visualization. During check pointing, the output of ghost cells creates overlapping I/O from all processes concurrently. The outcome of the overlapped file regions from a concurrent I/O is commonly referred as *atomicity*.

In this paper, we examine the implementation issues for concurrent overlapping I/O operations that abide the MPI atomicity semantics. We first differentiate the MPI atomicity semantics from the definition in POSIX standard. The POSIX definition only considers atomicity at the granularity of `read()/write()` calls in which only a contiguous file space can be specified in a single I/O request. In MPI, a process can define a non-contiguous file view using MPI derived data types and subsequent I/O calls can then implicitly access non-contiguous file regions. Since the POSIX definition is not aware of non-contiguous I/O access, it alone cannot guarantee atomic access in MPI, and additional efforts are needed above the file system to ensure the correct implementation of atomic MPI access. In this work, we study two approaches for atomicity implementation: using byte-range file locking and a process handshaking strategy. Using a byte-range file locking mechanism is a straightforward method to ensure the atomicity. In many situations, however, file locking can serialize what were intended to be concurrent I/O calls and, therefore, it is necessary to explore alternative approaches. Process handshaking uses inter-process communication to determine the access sequence or agreement on the overlaps, in which two methods are studied: graph-coloring and process-rank ordering methods. These two methods order the concurrent I/O requests in a sequence such that no two overlapping requests can perform at any instance. Experimental performance results are provided for running a test code using a column-

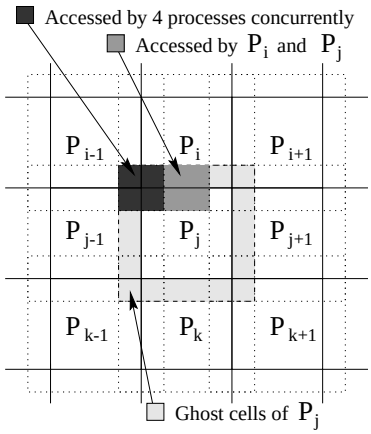


Figure 1. A 2D array partitioned with overlaps. The ghost cells of P_j overlaps with its 8 neighbors resulting in some areas accessed by more than one process.

wise partitioning pattern on three machine platforms: an Linux cluster running an extended NFS file system, an SGI Origin2000 running XFS, and an IBM SP running GPFS. The results show that, in general, using file locking generates the worst performance and using the process-rank ordering performs the best on all three machines.

The rest of the paper is organized as follows. Section 2 describes the difference between POSIX and MPI atomicity semantics. We explore three potential approaches for implementing MPI atomicity in depth in Section 3. In Section 4, we present performance results and the paper is concluded in Section 5.

2. Concurrent Overlapping I/O

The *concurrent overlapping I/O* referred to in this paper occurs when I/O requests from multiple processes are issued simultaneously to the file system and overlaps exist among the file regions accessed by these requests. If all the requests are read requests, the file system can use the disk cache to duplicate the overlapped data for the requesting processes and no conflict will exist when obtaining file data among the processes. However, when one or more I/O requests are write requests, the outcome of the overlapped regions, either in file or in process's memory, can vary depending on the implementation of the file system. This problem is commonly referred as the *I/O atomicity*.

2.1. POSIX Atomicity Semantics

POSIX standard defines atomicity such that all the bytes from a single file I/O request that start out together end up together, without interleaving from other I/O requests [3, 4]. The I/O operations confined by this definition include the system calls that operate on regular files, such as `open()`, `read()`, `write()`, `chmod()`, `lseek()`, `close()`, and so on. In this paper, we focus on the effect of the read and write calls on the atomicity.

The POSIX definition can be simply interpreted as that either all or none of the data written by a process is visible to other processes. The none case can be either the write data is cached in a system buffer and has not been flushed to the disk or the data is flushed but over-written by other processes. Hence, when POSIX semantics is applied to the concurrent overlapping I/O operations, the data resulted in the overlapped regions in disk shall consist of data from only one of the write requests. In other words, no mixed data from two or more requests shall appear in the overlapped regions. Otherwise, in non-atomic mode, the result of the overlapped region is undefined, i.e. it may comprise mixed data from multiple requests. Many existing file systems support the POSIX atomicity semantics, such as NFS, UFS, IBM PIOFS, GPFS, Intel PFS, and SGI XFS.

POSIX atomicity mainly considers the I/O calls defined within the POSIX scope in which its read and write calls share a common characteristic: one I/O request can only access a contiguous file region specified by a file pointer and the amount of data starting from the pointer. Therefore, the overlapped data written by two or more POSIX I/O calls can only be a contiguous region in file. Many POSIX file systems implement the atomic I/O by serializing the process of the requests such that the overlapped regions can only be accessed by one process at any moment. By considering only the contiguous file access, the POSIX definition is suitable for file systems that mainly handle non-parallel I/O requests. For I/O requests from parallel applications that frequently issue non-contiguous file access requests from multiple processes, POSIX atomicity may improperly describe such parallel access patterns and impose limitation for the I/O parallelism.

2.2. MPI Atomicity Semantics

MPI standard 2.0 [5] extends the atomicity semantics by taking into consideration of the parallel I/O operations. The MPI atomic mode is defined as: in concurrent overlapping MPI I/O operations, the results of the overlapped regions shall contain data from only one of the MPI processes that participates in the I/O operations. Otherwise, in the MPI non-atomic mode, the result of the overlapped regions is undefined. The difference of the MPI atomicity from POSIX definition lies on the use of MPI file view, a new file concept introduced in MPI 2.0. A process' file view is created by calling `MPI_File_set_view()` through an MPI derived data type that specifies the visible file range to the process.

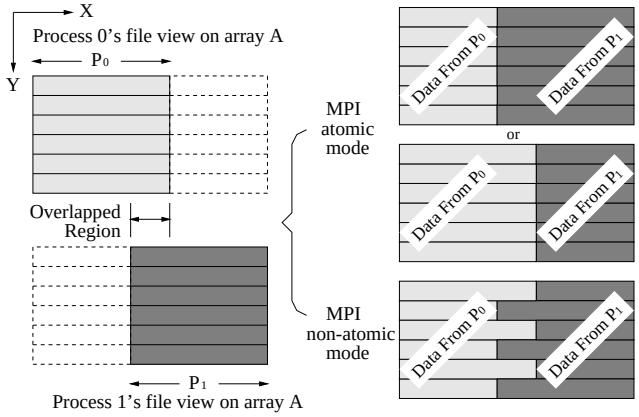


Figure 2. A 2D column-wise partitioning with overlaps on 2 processes. In MPI atomic mode, overlapped data can only come from either P_0 or P_1. Otherwise, the result is undefined, for example, interleaved.

When used in message passing, the MPI derived data type is a powerful mechanism for describing the memory layout of a message buffer. This convenient tool is extended in MPI 2.0 for describing the file layout for process' file view. Since a derived data type can specify a list of non-contiguous file segments, the visible data to a process can also be non-contiguous. In an MPI I/O operation, all visible segments to a requesting process are logically considered as a continuous data stream from/to the file system.

In MPI atomicity semantics, a call to `MPI_File_read_xxx()/ MPI_File_writ_xxx()` is regarded as a single I/O operation. A single collective MPI I/O operation can contain requests from multiple processes. Since each process can defines its own file view with a list of non-contiguous file segments, the overlapped file regions between two processes can also be non-contiguous in file. If the underlying MPI I/O implementation considers the access to each file segment as a single `read()/write()` call, then there will be multiple calls issued simultaneously from a process to the file system. Although the atomicity of accessing to a contiguous overlapped region is guaranteed in the POSIX compliant file systems, the MPI atomicity which demands atomicity across one or more regions of overlap cannot simply rely on the POSIX I/O calls. Additional effort is required to implement a correct MPI atomicity semantics. The fact that MPI derived data types provide more programming flexibility when specifying non-contiguous file layout increases the complexity of enforcing atomicity in MPI.

Figure 2 shows an example of a concurrent write from two processes in MPI atomic and non-atomic modes. The file views of both processes consist of 6 non-contiguous file segments, assuming the two-dimensional array is stored in row major. If writing each of the file segment uses a single call to `write()`, then there will be 12 write calls issued in total Since the processing order of these 12 calls in the file system can be arbitrary, the result in the overlapped columns can contain interleaved data, as illustrated in the MPI non-atomic mode. The same outcome will occur in a POSIX file system since POSIX atomicity only considers the `read()/write()` call individually. Therefore, the MPI implementation cannot simply rely on the file system to provide the correct file atomicity.

3 Implementation Strategies

The design of existing file systems seldom consider concurrent overlapping I/O requests and many optimization strategies can actually hinder the parallelism of overlapping I/O. For example, in most client-server type of file systems, read-ahead and write-behind strategies are adopted in which read-ahead pre-fetches several file blocks following the data actual requested to the client's system cache in anticipation of program's sequential reading pattern and write-behind accumulates several requests in order to better utilize the available I/O bandwidth. The read-ahead and write-behind policies often work against the goals of any file system relying on random-access operations which are used commonly in parallel I/O operations. Under these two policies, two overlapping processes in a concurrent I/O operation can physically cache more overlapping data than logically overlaps in their file views. It is also possible that the overlapping data of two processes is cached by a third process because of the read ahead.

The cache consistency problem has been studied extensively in many client-server based file systems. The most commonly implemented caching scheme is to consult the server's modification time for the data cached on the clients before issuing the I/O requests. Obviously, communication overhead between server and clients for cache validation and refreshing can become significant for a concurrent overlapping I/O request due to the unnecessary data transfers. Although this problem can be alleviated by disabling the use of read-ahead/write-through, the performance gain of the reduced overhead may not offset the performance loss of disabling caching. In this work, our discussion is not limited to specific file systems and we assume the general I/O requests can start at arbitrary file space. We now examine two potential implementation strategies for MPI atomicity and analyze their performance complexity:

1. **Using byte-range file locking** – This approach uses the standard Unix byte-range file locking mechanism to wrap the read/write call in each process such that

the exclusive access permission of the overlapped region can be granted to the requesting process. While a file region is locked, all read/write requests to it will directly go to the file server. Therefore, the written data of a process is visible to other processes after leaving the locking mode and the subsequent read requests will always obtain fresh data from the servers because of the use of the read locks.

2. **Using process handshaking** – This approach uses MPI communication to perform inter-process negotiation for writing to the overlapped file regions. The idea is a preferable alternative to using file locking. However, for file systems that perform read-ahead and write-behind, a file synchronization call immediately following every write call may be required to flush out all information associated with the writes in progress. Cache invalidation may also be needed before reading from the overlapped regions to ensure the fresh data coming from the servers. Under this strategy category, we further discuss two negotiation methods: graph-coloring and process-rank ordering.

In order to help describe the above three approaches in terms of data amount and file layouts, we use two concurrent overlapping I/O cases as examples. These two cases employ commonly seen access patterns in many scientific applications: row-wise and column-wise partitioning on a two-dimensional array.

3.1. Row and Column-wise 2D Array Partitioning

Given P processes participating a concurrent I/O operation, the row-wise partitioning pattern divides a two-dimensional array along its most significant axis while the column-wise divides it along the least significant axis. To simplify the discussion, we assume all I/O requests are write requests and the following assumptions are also made:

- All P processes concurrently write their sub-arrays to a single shared file.
- The layouts of the 2-dimensional array in both memory and disk storage are in row-major order where axis Y is the most significant axis and X is the least.
- The sub-arrays partitioned in every two consecutive processes overlap with each other for a few rows/ columns on the boundary along the partitioning axis.
- The global array is of size $M \times N$ and the number of overlapped rows/columns is R, where $R < M/P$ and $R < N/P$.

Figure 3 illustrates the two partitioning patterns on $P = 4$ processes. In the row-wise case, the file view of process P_i

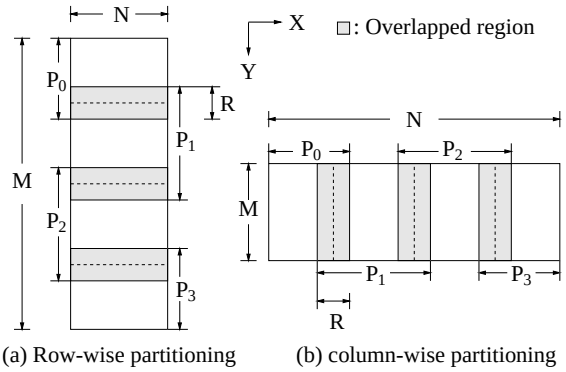


Figure 3. Row-wise and column-wise partitioning on a 2D array. The file views of every two consecutive processes overlap with each other in R rows/columns along YIX axis.

is a sub-array of size $M' \times N$, where $M' = \frac{M}{P} + R$, if $0 < i < P-1$. In the column-wise case, the file view of P_j is of size $M \times N'$, where $N' = \frac{N}{P} + R$ for $0 < j < P-1$. Both P_0 and P_{P-1} contains $\frac{R}{2}$ rows/columns less in row and column-wise cases, respectively.

3.2. Byte-range File Locking

The byte-range file locking is a mechanism provided by a file system within its locking protocol. This mechanism can be used to ensure the exclusive access to a locked file region. If a set of concurrent I/O calls contains only read requests, the locking protocol is usually implemented to allow a shared read lock so that more than one process can read the locked data simultaneously. If at least one of the I/O requests is a write request, the write lock is often granted exclusively to the requesting processes. Most of the existing locking protocols are centrally managed and its scalability is, hence, limited. A distributed locking protocol used in the IBM GPFS file system relieves the bottleneck by having a process manage its granted locked file region for the further requests from other processes [8]. When it comes to the overlapping requests, however, concurrent writes to the overlapped data must be still sequential.

Row-wise Partitioning We now use the row-wise partitioning example shown in Figure 3(a) to describe the atomicity implementation using file locking. In this example, the file view of a process overlaps R rows with its previous and successive processes. Since the file storage layout is assumed to be in a row-major order, i.e. each row of size N is stored consecutively to its previous and successive row, every process' file view actually covers a single

contiguous file space. Therefore, the concurrent overlapping I/O can be implemented using a single `write()` call in each process. On the file system that supports only the atomic mode, atomic file results are automatically guaranteed for the row-wise partitioning case. On file systems that do no support the atomic mode, wrapping the I/O call in each process with byte-range locking of the file region will also generate atomic results. ROMIO, an MPI-IO implementation developed at Argonne National Laboratory, relies on the use of byte-range file locking to implement the correct MPI atomicity in which processes must obtain a exclusive write lock to the overlapped file regions before perform the write [11, 12].

Column-wise Partitioning In the column-wise partitioning case shown in Figure 3(b), the file view of each process is a sub-array of size $M \times N'$ overlapping R columns with its left and right processes. Note that each of the M rows of size N' in the file view is not contiguous with its previous or successive row in the file storage layout. The distance between the first elements of two consecutive rows in each process' file view is N. Therefore, the overlapped file regions of two consecutive processes consist of M non-contiguous rows of size R each. Figure 4 shows an MPI code fragment that creates the file view for each process using a derived data type to specify the column-wise partitioning pattern and uses a collective MPI-IO call to perform the concurrent write.

An intuitive implementation for the column-wise case is to regard each contiguous I/O request as a single `read()/write()` call. This approach results M write calls from each process and PM calls in total. On a POSIX file system, if all PM requests are processed concurrently without any specific order, interleaved results may occur in the overlapped regions. Since processing order of these write requests can be arbitrary, the same scenario can also occur on other file systems even if file locking wraps around each I/O call. Enforcing the atomicity of individual `read()/write()` calls is not sufficient to enforce MPI atomicity. One solution is for each process to obtain all M locks before performing any write calls. However, this approach can easily cause dead lock when waiting for the requesting locks to be granted. An alternative is that the file lock starts at the process's first file offset and ends at the very last file offset the process will write, virtually the entire file. In this way, all M rows of the overlapped region will be accessed atomically.

Though POSIX defines a function, `lio_listio()`, to initiate a list of non-contiguous file accesses in a single call, it does not explicitly indicate if its atomicity semantics are applicable. If POSIX atomicity is extended to `lio_listio()`, the MPI atomicity can be guaranteed by implementing the non-contiguous access on top of

```
1.  MPI_File_open(comm, filename, io_mode, info, &fh);
2.  MPI_File_set_atomicity(fh, 1);
3.  sizes[0] = M;            sizes[1] = N;
4.  sub_sizes[0] = M;    sub_sizes[1] = N / P;
5.  if (rank == 0 || rank == P-1)   sub_sizes[1]  -=  R/2;
6.  starts[0] = 0;           starts[1] = (rank == 0) ? 0 : rank * (N/P - R/2);
7.  MPI_Type_create_subarray(2, sizes, sub_sizes, starts, MPI_ORDER_C,
8.                                MPI_CHAR, &filetype);
9.  MPI_Type_commit(&filetype);
10. MPI_File_set_view(fh, disp, MPI_CHAR, filetype, "native", info);
11. MPI_File_write_all(fh, buf, buffer_size, etype, &status);
12. MPI_File_close(&fh);
```

Figure 4. An MPI code fragment that performs the column-wise access. The shade area illustrates the construction of the derived data type, to define process's file view.

`lio_listio()`. Otherwise, additional effort such as file locking is necessary to ensure the MPI atomicity.

3.3. Processor Handshaking

An alternative approach to avoid using file locking is through process handshaking in which the overlapping processes negotiate with each other to obtain the desirable access sequence to the overlapped regions. In this section, we discuss two possible implementations of process handshaking: graph-coloring and process-rank ordering methods.

3.3.1. Graph-coloring Approach

Given an undirected graph $G = (V, E)$ in which V represents a set of vertices and E represents a set of edges that connect the vertices, a k-coloring is a function $C : V \rightarrow \{1, 2, ...k\}$ such that for all $u, v \in V$, if $C(u) = C(v)$, then $(u, v) \notin E$; that is, no adjacent vertices have the same color. The graph-coloring problem is to find the minimum number of colors, k, to color a given graph. Solving the MPI atomicity problem can be viewed as a graph-coloring problem if the I/O requesting processes are regarded as the vertices and the overlapping between two processes represents the edge. When applying graph coloring to the MPI atomicity implementation, the I/O processes are first divided into k groups (colors) in which no two processes in a group overlap their file views. Then, the concurrent I/O is carried out in k steps. Note that process synchronization between any two steps is necessary to ensure that no process in one group can proceed with its I/O before the previous group's I/O completes. The graph-coloring approach fulfills the requirement of MPI atomicity while maintaining at least a degree of I/O parallelism.

The graph-coloring methodology is a heuristic which has

Given an overlapping P x P matrix, W, where

W [i][j] = { 1 if process i overlaps j and i ≠ j ; 0 otherwise }

R_i : the i^{th} row of W R_i [j] : the j^{th} element of R_i

R' : an array of size P C : an array of size P, initial all -1

```
1.  maxColor ← 0
2.  for each row i = 0 . . . P-1
3.      for j = 0 . . . P-1
4.          if W [ i ][ j ] = 0 and C [ i ] < 0 then
5.              C [ j ] ← maxColor
6.              break
7.      R' ← R_j
8.      for k = j+1 . . . P-1
9.          if R' [ k ] = 0 and C [ k ] < 0 then
10.             C [ k ] ← maxColor
11.             R' ← R' ∨ R_k
12.     maxColor ← maxColor + 1
13. myColor ← C [ self ]
```

Figure 5. A greedy graph-coloring algorithm that finds the color id for each I/O process in variable myColor.

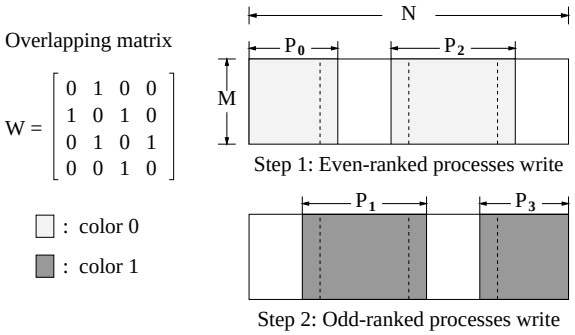


Figure 6. For the 2D column-wise access, the graph-coloring algorithm divides the I/O requests into 2 steps: even-ranked processes write first followed by the odd-ranked.

been studied for a long time and is proved to be NP-hard for general graphs [2]. Because the overlapping I/O patterns present in most of the science applications are hardly arbitrary, a greedy solution may suffice. Figure 5 gives a simple greedy graph-coloring algorithm that first uses a $P \times P$ overlapping matrix, W, to indicate if there is an overlap between two processes and starts coloring the processes by looking for the lowest ranked processes whose file views do not overlap with any process in that color. Let's now consider the column-wise partitioning example. Figure 6 shows the overlapping matrix using this greedy algorithm. It is obvious that two colors are enough to maintain MPI atomicity: the even-ranked processes perform their I/O requests prior to the odd-ranked processes.

3.3.2. Process-rank Ordering

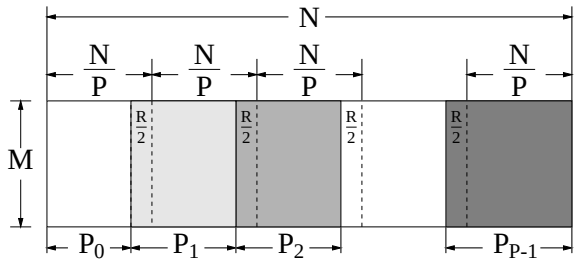
Another process-handshaking approach is to have all processes agree on a certain access priority to the overlapped file regions. An example is to use a policy where the higher ranked process wins the right to access the overlapped regions while others surrender their writes. A couple of immediate advantages of this approach are the elimination of overlapping access so that all I/O requests can proceed concurrently and the reduction of the overall I/O amount. The overhead of this method is the re-calculation of each process's file view by marking down the overlapped regions with all higher-rank processes' file views. Considering the column-wise partitioning example, Figure 7 illustrates the new processes' file views generated from the process-rank ordering approach. The new file view for process P_i, $0 < i < P-1$, is a $M \times \frac{N}{P}$ sub-array while the file views for P_0 and P_{P-1} are $M \times (\frac{N}{P} - \frac{R}{2})$ and $M \times (\frac{N}{P} + \frac{R}{2})$, respectively. Compared to Figure 6, each process surrenders its write for the right-most R columns.

3.4. Scalability Analysis

In the column-wise partition case, the file locking approach results in $MN - (N - N')$ bytes, nearly the entire file, being locked while each process is writing. In fact, once a process is granted its write locking request, no other processes can access to the file. As a result, using byte-range file locking serializes the I/O and dramatically degrades the performance. The purpose of proposing the two process-handshaking approaches is trying to maintain the I/O scalability without the use of file locking. The overhead of the graph-coloring approach is the construction of the overlapping matrix using all processes' file views. In the column-wise partitioning case, the graph-coloring approach maintains half of the I/O parallelism. In the process-rank ordering approach, the exact overlapped byte ranges must be known in order to generate the new local file view. Once the new file views are obtained, I/O requests can proceed with full parallelism. The overhead of both approached is expected to be negligible when compared to the performance improvement resulting from the removal of all overlapping requests. Additionally, the overall I/O amount on the file system is reduced since the lower-rank processes surrender their accesses to the overlapped regions.

4 Experiment Results

We implemented the column-wise partitioning example using standard Unix I/O calls and obtained experimental re-



New process file views

Figure 7. The new process file views for the column-wise overlapping I/O resulted from the process-rank ordering approach.

Table 1. System configurations for the three parallel machines on which the experimental results were obtained.

	Cplant	Origin 2000	IBM SP
File system	ENFS	XFS	GPFS
CPU type	Alpha	R10000	Power3
CPU Speed	500 MHz	195 MHz	375 MHz
Network	Myrinet	Gigabit Ethernet	Colony switch
I/O servers	12	-	12
Peak I/O bandwidth	50 MB/s	4 GB/s	1.5 GB/s

sults from three parallel machines: ASCI Cplant, an Alpha Linux cluster at Sandia National Laboratory; the SGI Origin 2000 at the National Center for Supercomputing Applications (NCSA); and Blue Horizon, the IBM SP at San Diego Supercomputing Center (SDSC). The machine configurations are briefly described in Table 1. Cplant is a Linux cluster running the Extended Network File System (ENFS) in which each compute node is mapped to one of the I/O servers in a round-robin selection scheme at boot time [7]. Basically, ENFS is an NFS file system with a few changes. The most notable is the absence of file locking on Cplant. Accordingly, our performance results on Cplant do not include the experiments that use file locking. ENFS also performs the optimization that NFS usually does, including read-ahead and write-behind.

We ran the experiments with the three array sizes: 4096×8192 (32MB), 4096×32768 (128 MB), and 4096×262144 (1GB). On all three machines, we used 4, 8, and 16 processors and the results are shown in Figure 8. Note the performance of file locking is the worst of the implementations of MPI atomicity. The poor results are also expected as discussed in Section 3.2 that file locking hinders the I/O concurrency. In most of the cases, the process-rank ordering strategy out-performed graph-coloring. The overheads of calculating the overlapping matrix for both graph-coloring and process-rank ordering approaches are less than 1 percent of the execution time in all the experiments.

5 Conclusions

In this paper, we examined the atomicity semantics for both the POSIX and MPI specifications. The difference between them is the number of non-contiguous regions in each I/O requests. While POSIX considers only one contiguous file space I/O, a single MPI I/O request can access non-contiguous file space using MPI's file view facility. We compared a few implementation strategies for enforcing atomic writes in MPI including file locking, graph-coloring, and process-rank ordering. The experimental results showed that using file locking performed the worst when running a two-dimensional column-wise partitioning case. Since file locking is basically a central managed mechanism, the parallelism of concurrent I/O requests, especially for overlapping I/O, can be significantly degraded by using it. The two alternatives proposed in this paper negotiate processes I/O request order of access priority through process handshaking. Without using a centralized locking mechanism, these two approaches greatly improve the I/O performance.

The strategies of graph-coloring and process-rank ordering require every process aware of all the processes participated in a concurrent I/O operation. In the scope of MPI, only collective calls have this property. Note that MPI collective I/O is different from the concurrent I/O in which a concurrent I/O is for more general I/O case. An MPI non-collective I/O operation can also be concurrent. File locking seems to be the only way to ensure atomic results in non-collective I/O calls in MPI, since the concurrent processes are unknown. Otherwise, given the participating processes, I/O optimizations such as the process handshaking approach proposed in this paper can be applied to improve performance.

6 Acknowledgments

This work was supported in part by DOE laboratories, SNL, LANL and LLNL under subcontract No. PO28264 and in part by NSF EIA-0103023. It was also supported in part by NSF cooperative agreement ACI-9619020 through computing resources provided by the National Partnership for Advanced Computational Infrastructure at the San Diego Supercomputer Center. We also acknowledge the use of the SGI Origin2000 at NCSA.

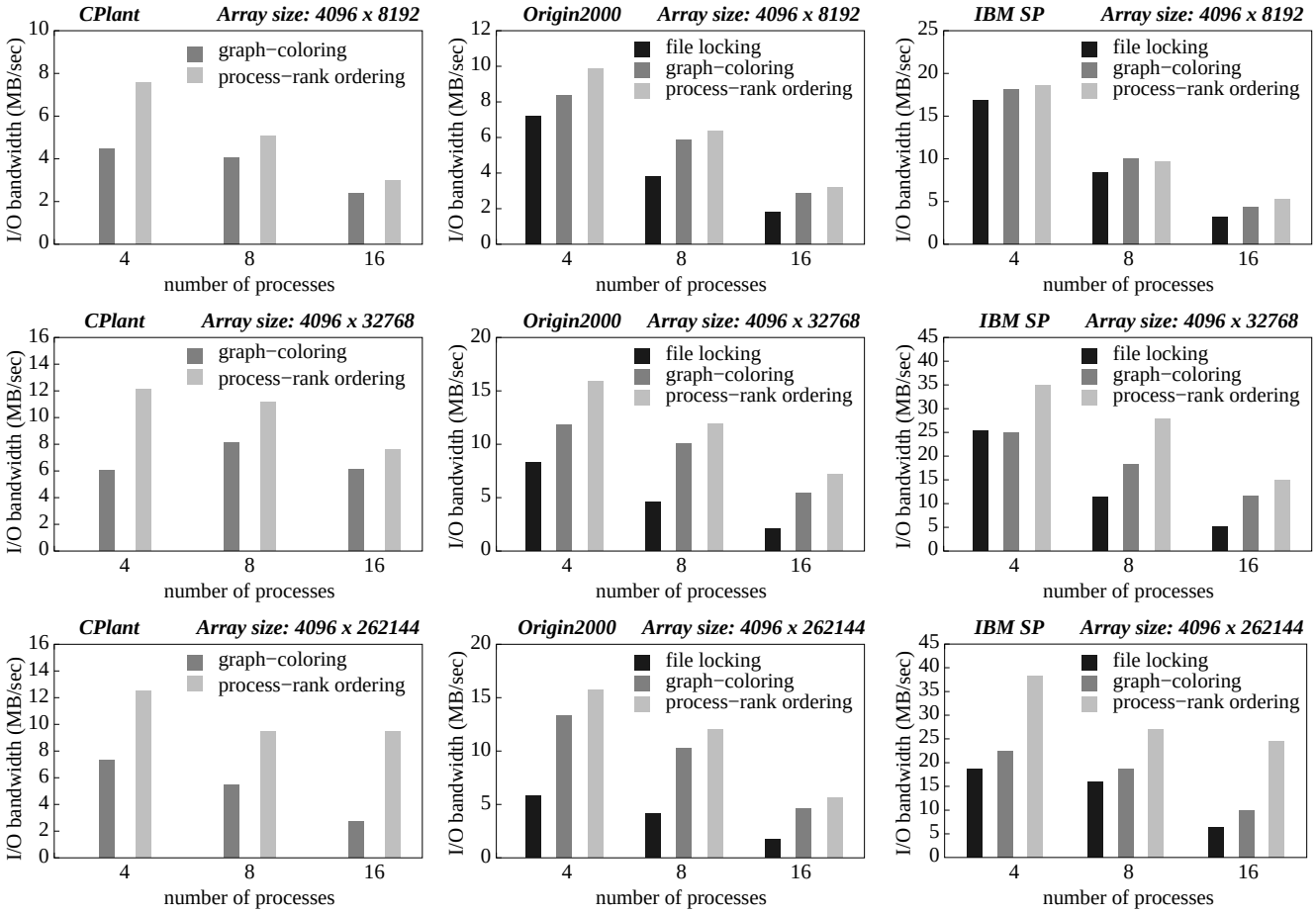


Figure 8. Performance results of running the column-wise partitioning experiments on a Linux Cluster, an IBM SP, and an SGI Origin200. Three file sizes were used: 32 MB, 128 MB, and 1GB.

References

[1] P. Crandall, R. Aydt, A. Chien, and D. Reed. Input-Output Characteristics of Scalable Parallel Applications. In *Supercomputing '95*, Dec 1995.

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

[3] IEEE Std. 1003.1-2001. *System Interfaces*, 2001.

[4] IEEE/ANSI Std. 1003.1. *Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language]*, 1996.

[5] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface.* http://www.mpi-forum.org/docs/docs.html, July 1997.

[6] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File-Access Characteristics of Parallel Scientific Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, Oct 1996.

[7] Sandia National Laboratories. *Computational Plant*. http://www.cs.sandia.gov/ Cplant.

[8] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *the Conference on File and Storage Technologies (FAST'02)*, pages 231–244, Jan 2002.

[9] E. Smirni, R. Aydt, A. Chien, and D. Reed. I/O Requirements of Scientific Applications: An Evolutionary View. In *the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 49–59, 1996.

[10] E. Smirni and D. Reed. Lessons from Characterizing the Input/Output Behavior of Parallel Scientific Applications. *Performance Evaluation: An International Journal*, 33(1):27–44, Jun 1998.

[11] R. Thakur, W. Gropp, and E. Lusk. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation.* Mathematics and Computer Science Division, Argonne National Laboratory, Oct. 1997. Technical Report ANL/MCS-TM-234.

[12] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *the Sixth Workshop on I/O in Parallel and Distributed Systems*, pages 23–32, May 1999.