Conflict Detection Algorithm to Minimize Locking for MPI-IO Atomicity

Saba Sehrish[1], Jun Wang[1], and Rajeev Thakur[2]

[1]School of Electrical Engineering and Computer Science,
University of Central Florida, Orlando, FL 32816, USA
[2]Mathematics and Computer Science Division,
Argonne National Laboratory, Argonne, IL 60439, USA

Abstract. Many scientific applications require high-performance concurrent I/O accesses to a file by multiple processes. Those applications rely indirectly on atomic I/O capabilities in order to perform updates to structured datasets, such as those stored in HDF5 format files. Current support for atomicity in MPI-IO is provided by locking around the operations, imposing lock overhead in all situations, even though in many cases these operations are non-overlapping in the file. We propose to isolate non-overlapping accesses from overlapping ones in independent I/O cases, allowing the non-overlapping ones to proceed without imposing lock overhead. To enable this, we have implemented an efficient conflict detection algorithm in MPI-IO using MPI file views and datatypes. We show that our conflict detection scheme incurs minimal overhead on I/O operations, making it an effective mechanism for avoiding locks when they are not needed.

1 Introduction

Some of the scientific applications are I/O intensive and demand high-performance I/O access and bandwidth to store and retrieve their simulation results. A relatively slow procedure to store and retrieve the scientific application results to and from the storage system limits the overall performance despite the ever increasing compute power. Parallel file systems and high-level parallel I/O libraries are provided as a solution to mitigate this lag. Parallel file systems provide the semantics of the local file system but face additional challenges regarding atomicity and consistency. Some parallel file systems such as PVFS [6] and PVFS2 [3] do not support POSIX [4] semantics for atomicity and consistency. These semantics are guaranteed only if there is no data sharing among concurrent processes and are inadequate to describe complex requests in scientific computing applications that are non-contiguous in files. Scientific applications do perform these types of operations [11] [13], so alternative mechanisms for guaranteeing atomicity are necessary, e.g. a locking mechanism.

Atomicity semantics define the outcome of multiple concurrent I/O accesses, at least one of which is a write to a shared or overlapping region. With the advent of parallel I/O libraries, data can be accessed in various complex patterns by multiple processes. These access patterns can be contiguous or non-contiguous, overlapping or non-overlapping depending upon the application behavior. Locking mechanisms provided by the file system are used to ensure that during a concurrent I/O access, shared data is not being violated. File locks, byte range locks, list locks, and

datatype locks are various locking mechanisms that are available to guarantee the atomicity semantics. Adapted from the POSIX semantics, parallel file systems such as GPFS [16] and Lustre [2] provide a byte range locking mechanism. Byte range locks provide an option for guaranteeing atomicity of non-contiguous operations. By locking the entire region, changes can be made by using a read-modify-write sequence. However, this approach does not consider the actual non-contiguous access pattern that may occur in a byte range and introduces *false sharing*. This approach also limits the benefits of parallel I/O that can be gained, by unnecessarily serializing the accesses. To address these particular cases, [5] [10] propose to lock the exact non-contiguous regions within a byte range and maximize the concurrent I/O access.

The overhead of a locking mechanism appears in three forms; first is the communication overhead that is generated while acquiring and releasing the locks (sending the requests to lock server(s)), second is the storage space overhead that is caused by storing the locks during their acquire and release time (a data structure is maintained to store the locks, and for fine grained locks like list locks, this structure can grow very large.), and the third is computation overhead to assign new locks (i.e., the tree data structure is scanned to check if the same lock request is being held by a different process). These overheads can be reduced by making sure that unnecessary locking requests are not generated. An observation is that the locks are requested (e.g. file locks, byte-range locks, list locks, and datatype locks) even if there are no overlaps, when locks are not needed. This observation leads us to a very important question, whether it is possible to isolate the cases where atomicity is required and where it is not, and optimize the locking mechanism.

We propose a scheme to identify the conflicts at the application level, by identifying the concurrent access patterns and overlaps within an application. Our conflict detection algorithm, implemented at the MPI-IO level for independent I/O operations uses file views and datatype decoding to determine the overlaps. Our results show that the conflict detection algorithm incurs a minimal overhead and performs within 3.6% of the ideal case (i.e. concurrent access with no locks). When the file view for a process does not overlap with the file views of other processes, locking is not required; there will be no conflicts. Our approach reduces the locking overhead at minimum for the non-overlapping regions but handles the overlapped regions using either file or byte range locks.

2 Design

We propose a conflict detection algorithm that should be performed before any locking mechanism. Our goal is to optimize the lock acquiring process by providing an efficient conflict detection algorithm beforehand to identify the overlapping regions, thereby requesting the locks only if there are overlapping regions. The conflict detection algorithm presented in this paper is based on MPI datatypes and file views. Typically, a file read/write request in any MPI-IO program consists of following steps: 1) Create the Data types, 2) Create the File views, and 3) Read-/Write Request. The conflict detection is performed when a file view is created (`MPI_File_set_view`). Since it is a collective call, each process can exchange their file views and determine the overlapping regions by comparing offset/block length pairs. Each node acts as a conflict detector for itself.

The file view is created using `MPI_File_set_view`, and then each node decodes the supplied MPI datatype. Decoding a datatype is not straight forward and is a two step procedure using MPI-IO functions. The first step is getting the envelope of the datatype using `MPI_Type_get_envelope` which returns information such as number of displacements and block lengths used to create the datatype. The second step is getting the actual contents in the form of offset/block length using `MPI_Type_get_contents`. The decoded datatypes are exchanged using collective communication functions. The overhead of conflict detection is based on the complexity and size of the datatype. For some datatypes this overhead is as small as exchanging two long integer values, while for others it can consist of a long list of offset/length pairs.

2.1 Conflict Detection

We categorize the derived datatypes in to two broader categories based on their structure. The first category is a *regular datatype*; all the processes have the same block size but a different displacement e.g. `MPI_Type_vector`, `MPI_Type_subarray`. In a subarray, each process accesses a subarray that is defined by the number of dimensions and the starting and ending offsets in each dimension. For a regular datatype, we need to exchange the start and end offset in each dimension for each process. The second category is *irregular datatype*; all the processes may access different block sizes, different patterns and different displacements, e.g. `MPI_Type_hindexed`, where each process accesses a non-contiguous region defined by a list of offsets and corresponding block lengths.

Regular Datatypes: In independent write operations, when there are overlaps among different writes, only one process should perform the write at a time. For regular datatypes, we take the example of `MPI_Type_subarray`. A subarray datatype is defined by the number of dimensions, size of array in each dimension, sizes of subarrays in each dimension and the start positions for each subarray. The start of each subarray and the size in each dimension is used to identify the overlaps between any two consecutive tiles or subarrays as shown in the following equations. A conflicting region is specified by $CR(CO, CL)$, where CO is the conflicting offset and CL is the conflicting length. The displacement of the i^{th} process is specified by $disp_i(x, y)$ for a two-dimensional subarray and the corresponding block length is given by $blklen_i(x, y)$.

$$CO = max(disp_i(x, y), disp_j(x, y)) \tag{1}$$

If both the displacements i.e. $disp_i$ and $disp_j$ are the same, the CL is given by eq. 2, otherwise eq. 3 is used.

$$CL = min(blklen_i(x, y), blklen_j(x, y)) \tag{2}$$

$$\begin{aligned} CL = [&min(disp_i(x, y) + blklen_i(x, y), \\ &disp_j(x, y) + blklen_j(x, y))] - CO \end{aligned} \tag{3}$$

Irregular Data types: For irregular datatypes, the offset/length pairs are required because each non-contiguous region will be of a different size. Each process compares its own offset/length list against the others. A conflicting region

$CR(CO, CL)$ is defined by the following equations, where CO is the starting offset, and CL is the length of the conflicting region. *disp(i)* is the displacement of the i^{th} process and *blklen(i)* is the corresponding block length.

$$CO = max(disp(i), disp(j)) \tag{4}$$

If both the displacements i.e. $disp(i)$ and $disp(j)$ are the same, the CL is given by eq. 5, otherwise eq 6 is used.

$$CL = min(blklen(i), blklen(j)) \tag{5}$$

$$CL = [min(disp(i) + blklen(i), \\ disp(j) + blklen(j))] - CO \tag{6}$$

Since, the exact displacement and block length values are used, *false sharing* is eliminated completely. There are more complex datatypes that we categorize as multi-level datatypes, and MPI facilitates the creation of nested datatypes. For example in *noncontig* benchmark, `MPI_Type_contig`, `MPI_Type_vector` and `MPI_Type_struct` are used to create file views. In such cases, we perform multi-level decoding to determine the conflicts. The overhead incurred by conflict detection is evaluated in Section 4 in terms of communication and computation time. The communication overhead is determined by the collective communication calls to exchange datatypes. The computation overhead includes the time to generate and compare the datatypes. It depends on the datatype or the size of the list to be compared. For each process, if the size of the list is N, it will perform a linear compare of order N. The space required is equal to the size of the datatype or the offset/length list.

3 Implementation

We implement the self-detecting locking mechanism in ROMIO, by adding it to `MPI_File_set_view` as shown in the listing 1. Listing 1 also shows how to use the conflict variable in the main program. The decoding process utilizes two function from MPI-IO library; `MPI_Type_get_envelope` and `MPI_Type_get_contents`. Our initial implementation provides conflict detection support for a few selected data types, `MPI_Type_vector, MPI_Type_subarray, MPI_Type_hindexed`. Each process exchanges the view information using the `MPI_Allgather` collective communication function. Once the data is ready at each node, it performs the comparison for the conflicts. Our current implementation is tested with PVFS2, which does not support locking. We have used the algorithms presented in [15] for file locks and [17] [14] for byte range locks implementations with PVFS2. For byte range locks, we determine the start and end offsets of the byte range accessed by each process using an existing function in ROMIO i.e. $ADIOI\_Calc\_my\_off\_len$; it returns the start and end offsets used by $BR\_Lock\_acquire(br\_lock, ..)$.

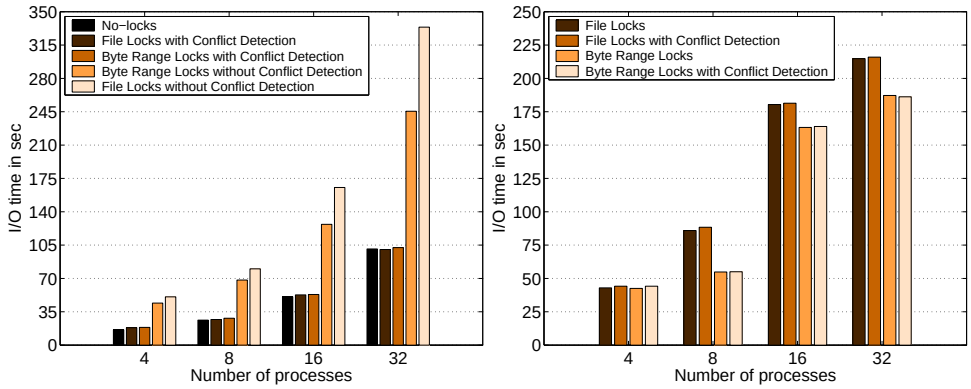
MPI-IO write functions can be performed collectively or independently. The collective write operations do not require conflict detection because conflicts cannot occur in collective operation. The independent write operations do not communicate with each other to optimize the non-contiguous access and require locking to protect the shared data regions. The blocking independent write functions are `MPI_File_write`, `MPI_File_write_at`, and `MPI_File_write_shared`. Our conflict detection implementation can be used with any locking mechanism and independent write function. et.

4 Performance

In this section, we evaluate our conflict detection algorithm. We quantify the overhead of the conflict detection algorithm, and compare the I/O time of various benchmarks. We evaluate the conflict detection using three different benchmarks for both overlaps and no-overlaps in file access patterns. We compare the time to determine conflicts combined with and without locks and also show the overhead of conflict detection in terms of the communication and computation time. In our experiments, we use the best case of a concurrent write access i.e. when there are no locks, and all processes can perform a write operation concurrently. The worst case used in the experiments is the whole file locks [15], when all writes by different processes become serial. Additionally, we have also used the byte range locks as implemented in [17] [14].

The experimental setup consists of a 16 node cluster, with PVFS2. Each node is a Dell PowerEdge 2 CPU, dual core with 4GB memory and two 500 GB SAS hard drives. PVFS2 has been setup on all 16 nodes, so all nodes serve as I/O nodes and the compute nodes. The network connection is Intel Pro/1000 NIC, and the cluster network consist of Nortel BayStack 5510-48T GigaBit switch. We have used PVFS2 version 2.7.0 and MPICH2-1.0.7 in our experiments.

MPI-Tile IO: MPI-Tile IO [1] is used to write non-overlapping and overlapping tiles. *Non-overlapping Access*: The number of dimensions for `MPI_Type_subarray` is



(a) I/O time for non-overlapping regions (b) I/O time for overlapping regions

Fig. 1: MPI-Tile-IO: Comparing I/O time for non-overlapping and overlapping regions with conflict detection and locks.

2 in MPI-Tile-IO, and the array of starts gives the x and y position for each tile. The array of sub-sizes returns the size of each tile in both dimensions. The problem size consist of an array of *4096 X 8192* per process, where each element size is 32 bytes. Each process writes 1GB of data, hence if there are 32 processes then the total amount of data written will be 32GB. The 4, 8, 16 and 32 processes are arranged in 2x2, 4x2, 8x2 and 16x2 panels. The I/O time results are shown in Figure 1. I/O

time includes the time for MPI_File_set_view (also the time for conflict detection), MPI_File_write and waiting time if there are file locks or byte range locks.

There are five bars, the first bar shows the best case of no-locks, whereas the last bar shows the worst case. The second and third bar shows the case when conflict detection is performed, no conflicts are reported and the underlying file and byte range locking is disabled as a result. The fourth bar performs byte range locks without using conflict detection. We can see that file and byte range locks combined with conflict detection performs close to the no-locks, i.e. an ideal case for the non-overlapping I/O accesses. The overhead of conflict detection is minimal, because the datatypes exchanged in MPI-Tile-IO consist of start and end offset of each tile accessed by a process. This overhead increases with the number of processes, and the detailed overhead results are shown in Figure 3. File locks have the worst performance because they introduce sequential access and the I/O time increases with the increase in number of processes.

Overlapping Access: We used *overlap-x* and *overlap-y* options in MPI-Tile-IO to generate the overlapped I/O access patterns. The problem size consists of an array of 4096×8192 per process with an overlap of 512×1024 in x and y direction respectively, where each element size is 32 bytes. Each process writes 1GB of data with an overlapping data of 16MB per process. The atomicity semantics guarantee that the 16MB overlapping data will be defined by one process at a time and not contain any data from more than one processes. We compare the I/O time for file locks and byte range locks with and without conflict detection. No-locks results are not provided here, because the output in overlapping region is not defined without locks. I/O time for the other four cases is shown in the Figure 1. It should be noted that since there are overlaps, locks cannot be avoided, and the purpose of these results is to demonstrate that if a conflict detection is performed before any locking mechanism, the overhead is not significant. The major contribution of our work is for the cases when there are no overlaps and locking is eliminated completely. These results also show that conflict detection can be combined with any other locking mechanism.

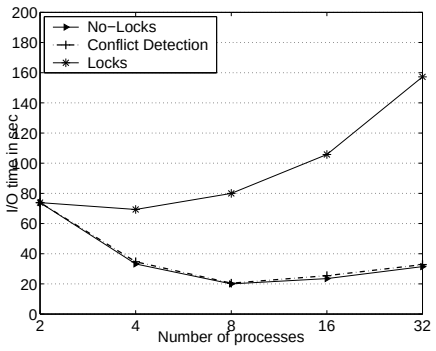


Fig. 2: S3asim: Comparing I/O time when there are no-locks, conflict detection with file locks, and file locks.

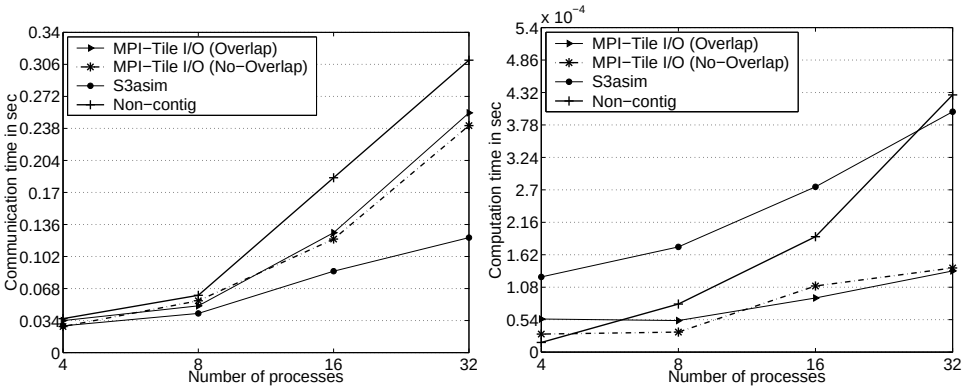
S3asim: S3asim is a sequence similarity search algorithm simulator [7], that uses a variety of parameters to adjust the total fragments in the database, sequence size, query count, etc. After each worker finishes its query processing of its fragment, it sends its ordered scores to the master process. The master process merges the ordered scores to its list and once all fragments of an input query have been processed, it send the locations in the aggregate file to each worker to write the results. Finally, each worker writes the result data to the output file independently when it receives the location

from master. The datatype used by workers is `MPI_Type_hindexed`, and is defined by an array of block lengths and displacements.

Our conflict detection algorithm returns no conflict for the S3asim benchmark, and this result is in accordance with [7]. All the workers work on different segments of the database for query search, and a few of them write the results to a shared file. Figure 2 shows three cases; no-locks i.e. none of the locking mechanism was applied, conflict detection i.e. conflict detection algorithm was run in order to determine overlaps, and finally file locks. The conflict detection is performed only for the processes that actually write, and on average it performs within 3.6% of the no-locks case. The file locks show increase in I/O time with increase in number of processes, since there are fewer writers as compared to the number of processes, and the line is not a steep curve.

Conflict Detection Overhead: In this section, we present the overhead incurred by our conflict detection algorithm. Each process participates in two collective communication calls to gather the file view and the starting offset. We measure the communication overhead as the time spend in communicating the required information. This overhead depends on the datatypes and the number of processes that actually perform the write operation. In Figure 3, we show the communication overhead in different benchmarks. It is noticed that in tile-io, each process writes a tile/subarray that may or may not have overlapping regions, but in S3asim only a few workers that find the match perform the write operation. This explains the less steep curve for S3asim as compared with tile-io. Non-contig shows the maximum overhead because it has a multi-level datatype and, we need two collective calls to communicate the two levels of datatypes.

After communicating the file views, each process computes the conflicts, which are the results of the comparisons of its own file view with the received ones. The time spent in computing the conflicts is quantified as the computation overhead. This overhead depends on the datatypes and also the size of offset/length pairs generated from the file views. In Figure 3, we also show the computation overhead in various benchmarks. It can be seen that the overhead is minimal in tile-io, the reason being that tile-io writes data logically in sub-arrays, and in order to perform the conflict detection we do not generate the offset/length pairs and use the start and end offsets in each dimension to detect conflicts. For S3asim, the overhead is also minimal because the actual number of workers writing the results is less than the number of processes performing the search. For example, in a $32p$ run, for certain queries only 4 or 5 processes write in the end. S3asim has a few writers but its datatype is more complex and with the increase in number of processes it has more writers, so S3asim has greater overhead with increased number of processes. Noncontiguous benchmark performs four different access patterns, i.e. contiguous/non-contiguous in memory and contiguous/non-contiguous in file. We only present the results when accesses are either non-contiguous in memory or in file. A file size of 2GB is used but we keep per process file size constant, the vector length i.e. the number of elements in vector datatype used is set to 32 and element count i.e. the number of elements in a contiguous chunk to 128. The first derived datatype is `MPI_Type_vector`, and the second derived datatype that is comprised of `MPI_Type_vector` is `MPI_Type_struct`. The overhead of conflict detection for non-contig is shown in Figure 3. Non-contig has a steady increase in computation overhead with number of processes, it is a case of multi-level datatype.



(a) Communication Time Overhead (b) Computation Time Overhead

Fig. 3: Communication and Computation Overhead of Conflict Detection for different Benchmarks

Number of Lock Requests: We emphasize that with conflict detection, if there are no overlaps in the application access pattern, locking can be avoided. Otherwise, only the overlapped region should be locked to guarantee the atomicity of concurrent operations. Finally, we investigate the utilization of our scheme with the existing locking mechanisms. Assuming that there are three locking implementations, i.e. whole-file, byte range and list locks, we compare the number of locks per client in each case with and without conflict detection. There is one lock per client for the file locks and the byte range locks, but these are coarse grained locks. The non-contiguous access patterns observe false sharing with coarse grained locks. We want to show that a locking mechanism combined with our approach is effective in reducing the number of lock requests that will be issued for any lock server that needs communication and space on the server to be stored. Many scientific applications have patterns that would require hundred thousands of locks [10]. In Table 1, we show a simple comparison of the number of locks (whole file, byte-range and list) with and without using conflict detection. It should be noticed that the list locks and the datatype locks are very fine grained locks. The locks acquiring process is instigated by a client. The client first calculates which servers to access for the locks; the saving from conflict detection will come in the form of either none or a fewer number of requests. The implementation of conflict detection with list locks [5] is left for future work.

Table 1: Reduction in number of locks

Approach	**Number of Locks/Client**
Whole-File Locks	1
Byte-Range Locks	1
List Locks	64
Locks with Conflict detection	*No-0verlaps:* 0 *Overlaps:* 1 (Whole-file), 1 (Byte-Range), Number of CR (List)

5 Related Work

Researchers have contributed to provide atomicity semantics both at application and file system level. Non-contiguous access patterns and overlapping I/O patterns [8] [9] [12] have been widely studied and the customized locking schemes, process rank ordering and handshaking have been proposed. List locks and datatype locks [5] [10] have maximum concurrency, but they acquire and maintain locks for all regions accessed by a process. We provide conflict detection to find the overlaps before lock requests are issued. The conflict check facilitates the locking mechanism by providing a decision where locks are necessary to guarantee atomicity.

6 Conclusion

We have proposed a scheme to perform conflict detection using file views, and introduce lock free independent write operations if there are no conflicts. We have implemented our algorithm in ROMIO. In MPI-IO applications atomicity guarantees rely on the file system locks. Our Conflict detection algorithm is able to extract overlapping regions from the file views (for independent operations) created by MPI-IO application with a minimal overhead. It paves the way to the lock-free and scalable approaches of MPI-IO atomicity support.

Listing 1 Pseudocode for Conflict Detection

```
//Pseudocode for Conflict Detection
int Conflict_Detection(MPI_Datatype ftype) {
//Get the datatype envelope,
MPI_Type_get_envelope(ftype, &num_ints, &num_adds, &num_dtypes, &combiner);

//Get the actual contents of the datatype
MPI_Type_get_contents(ftype, num_ints, num_adds, num_dtypes,
array_of_ints, array_of_adds, array_of_dtypes);

//Gather datatypes from all other nodes
MPI_Allgather(array_of_ints, num_ints, MPI_INT, ai_all,
num_ints, MPI_INT, MPI_COMM_WORLD);
...
//Compare datatypes
switch(combiner){
case MPI_COMBINER_SUBARRAY:
// Compare the elements of ai_all, that contains array of
   starts, and the block lengths are same for all blocks!
break;
case MPI_COMBINER_INDEXED:
case MPI_COMBINER_HINDEXED:
break;
...
}
return conflict;
}
```

7 Acknowledgments

We would like to thank Rob Ross of Argonne National Laboratory for his guidance and valuable comments on this paper. This work is supported in part by the US National Science Foundation under grants CNS-0646910, CNS-0646911, CCF-0621526, CCF-0811413, and the US Department of Energy Early Career Principal Investigator Award DE-FG02-07ER25747.

References

1. http://www.cs.dartmouth.edu/pario/examples.html.
2. Lustre filesystem. http://www.lustre.org/.
3. Parallel virtual file system version 2. http://www.pvfs.org/.
4. IEEE/ANSI std.1003.1. portable operating system interface (POSIX)-part 1: System application program interface (API)[C language], 1996.
5. Peter M. Aarestad, Avery Ching, George K. Thiruvathukal, and Alok N. Choudhary. Scalable approaches for supporting MPI-IO atomicity. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 35–42, Washington, DC, USA, 2006. IEEE Computer Society.
6. Philip H. Carns, III Walter B. Ligon, Robert B. Ross, and Rajeev Thakur. PVFS: a parallel file system for linux clusters. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.
7. A. Ching, W. Feng, H. Lin, X. Ma, and A. Choudhary. Exploring I/O strategies for parallel sequence-search tools with s3asim. *hpdc*, 0:229–240, 2006.
8. Avery Ching, Alok Choudhary, Kenin Coloma, Wei keng Liao, Robert Ross, and William Gropp. Noncontiguous I/O accesses through MPI-IO. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 104, Washington, DC, USA, 2003. IEEE Computer Society.
9. Avery Ching, Alok Choudhary, Wei keng Liao, Rob Ross, and William Gropp. Noncontiguous I/O through PVFS. In *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*, page 405, Washington, DC, USA, 2002. IEEE Computer Society.
10. Avery Ching, Wei keng Liao, Alok Choudhary, Robert Ross, and Lee Ward. Noncontiguous locking techniques for parallel file systems. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.
11. Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *in Proceedings of Supercomputing '95*. Press, 1995.
12. Wei keng Liao, Alok Choudhary, Kenin Coloma, George K. Thiruvathukal, Lee Ward, Eric Russell, and Neil Pundit. Scalable implementations of MPI atomicity for concurrent overlapping I/O. *Parallel Processing, International Conference on*, 0:239, 2003.
13. Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael L. Best. File-access characteristics of parallel scientific workloads. *IEEE Trans. Parallel Distrib. Syst.*, 7(10):1075–1089, 1996.
14. Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William D. Gropp. Formal verification of programs that use MPI one-sided communication. In *PVM/MPI*, pages 30–39, 2006.
15. Robert B. Ross, Robert Latham, William Gropp, Rajeev Thakur, and Brian R. Toonen. Implementing MPI-IO atomic mode without file system support. In *CCGRID*, pages 1135–1142, 2005.
16. Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 19, Berkeley, CA, USA, 2002. USENIX Association.
17. Rajeev Thakur, Robert B. Ross, and Robert Latham. Implementing byte-range locks using mpi one-sided communication. In *PVM/MPI*, pages 119–128, 2005.