Improving the Average Response Time in Collective I/O

Chen Jin[1], Saba Sehrish[1], Wei-keng Liao[1],
Alok Choudhary[1], and Karen Schuchardt[2]

[1] Northwestern University, Evanston, IL 60202, USA
[2] Pacific Northwest National Laboratory, Richland, WA 99352, USA
{chen.jin,ssehrish,wkliao,choudhar}@eecs.northwestern.edu,
Karen.Schuchardt@pnnl.gov

Abstract. In collective I/O, MPI processes exchange requests so that the rearranged requests can result in the shortest file system access time. Scheduling the exchange sequence determines the response time of participating processes. Existing implementations that simply follow the increasing order of file offsets do not necessary produce the best performance. To minimize the average response time, we propose three scheduling algorithms that consider the number of processes per file stripe and the number of accesses per process. Our experimental results demonstrate improvements of up to 50% in the average response time using two synthetic benchmarks and a high-resolution climate application.

1 Introduction

Parallel I/O systems have always faced challenges to efficiently store and retrieve the ever-growing amount of data. Over the past two decades, researchers have proposed different solutions, such as MPI I/O, to improve the performance. MPI collective I/O requires the participation of all processes that open a shared file. This requirement provides a collective I/O implementation an opportunity to exchange access information and reorganize I/O requests among the processes. Several process-collaboration strategies have been proposed, such as two-phase I/O [1], disk directed I/O [2], and server-directed I/O [3].

Two-phase I/O is a representative collaborative I/O technique that runs in the user space. Its idea is to reorganize the requests among processes, so that the rearranged requests incur the minimal overhead from the underlying file system. The request reorganization is referred to as the communication phase while the read/write system calls constitute the I/O phase. ROMIO, a popular MPI-IO implementation [4], adopts the two-phase I/O strategy [5], which first identifies the aggregate access region and picks a subset of MPI processes as the I/O aggregators, the only processes making I/O calls to the file system. The aggregate access region is a minimum contiguous file region covering all the I/O requests. The region is partitioned into disjointed sub-regions denoted as **file domains**, and each is assigned to a unique I/O aggregator.

Y. Cotronis et al. (Eds.): EuroMPI 2011, LNCS 6960, pp. 71–80, 2011.

MPI collective operations only require the participation of all processes, which should not be confused with the process synchronization. Essentially, once a process participates in a collective operation, it can return from the call without waiting for the completion of other processes. Hence, an optimal request scheduling method for a collective operation should minimize the average response time of all processes. The I/O request scheduling is a key component of the communication phase in the collective I/O implementation. Let us take ROMIO's implementation for the Lustre file system as an example to examine the importance of a scheduling strategy. At the file open, the Lustre driver chooses an equal number of I/O aggregators as the number of the file servers, referred to as Object Storage Targets (OSTs) in Lustre. All stripes of a file are assigned to the aggregators' file domains in a round-robin fashion in order to produce a one-to-one mapping between the aggregators and file servers [6]. Because Lustre adopts an extent-based locking protocol, such assignment can optimally minimize the lock request from each aggregator to the file system [7,8].

In the current ROMIO implementation, each aggregator handles the requests exchange for all the stripes in its file domain. That is, it schedules one stripe at a time in the increasing file offset order of the stripes. We argue that such service scheduling strategy does not necessarily result in the best response time for the non-aggregators. The example in Figure 1(a) shows a collective write from 4 MPI processes to 3 aggregators. The service scheduling using the increasing file offset order is shown in Figure 1(b), where the requests from P_3 are served last as they have the highest offsets. In this case, the average response time is $(4t \times 3 + 5t)/4 = 4.25t$. In Figure 1(c) where P_3's requests are served first, the average response time is reduced to $(t + 5t \times 3)/4 = 4t$. The faster response time means the processes can return earlier from the call and proceed to the successive tasks. This example shows that a different request scheduling order can result in a different average response time and serves as the motivation of our work.

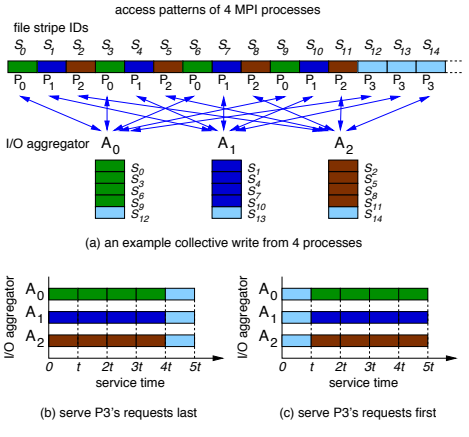


Fig. 1. (a) Collective I/O example with 4 MPI processes and 3 OSTs. (b) Average response time is $4.25t$ when P_3's requests are served last. (c) Average response time is $4t$ when P_3's requests are served first.

We propose three alternative algorithms for request scheduling: Most Degree First (MDF), Locally Weighted MDF (LW-MDF), and Globally Weighted MDF (GW-MDF). These algorithms prioritize the file stripes based on their access degree, the number of accessing processes. The MDF schedules the stripe with the highest degree to be served first. The LW-MDF assigns a weight to each

process using its total number of requests to individual aggregators. The GW-MDF assigns the weight based on all the local weights of a process across the aggregators. The weighted schemes are used to calculate the priority scores for the stripes. Our experiments on the Cray XT4 parallel machine and Lustre file system at the National Energy Research Scientific Computing Center show that the average response time is reduced by 30% for a fixed uneven workload, 50% for a random workload, and 20% for a large-scale climate simulation application.

2 Design and Implementation

Our objective for developing the three alternative scheduling algorithms is to minimize the average service response time, $\overline{T_c}$, of all the MPI processes in a collective I/O. We take into consideration the accessing **degree** of a file stripe, which is defined as the number of processes accessing it. The proposed algorithms do not change the I/O amount on the aggregators and if the cost of I/O phase dominates the collective I/O, then the time on the aggregators will not change significantly. What the proposed algorithms intend to improve is the response time mainly on the non-aggregators. We assume the same cost for the I/O phase irrespective of the stripe permutations carried out to the file system. The request size per aggregator is same as the stripe size, which is between 1MB and 4MB. The stripe size is a multiple of the disk sector size (512 bytes), hence, it will not affect the disk seek time. Based on our experiments on Lustre, the I/O for different stripe permutations costs approximately the same, as long as each aggregator only accesses the same server.

Most Degree First (MDF). Among all the stripes in an aggregator's file domain, the MDF method schedules the stripe with the highest degree first. Intuitively, if the stripes with larger degree stripes are serviced first, then more non-aggregator processes will complete their collective I/O earlier. In ROMIO, at the beginning of a collective I/O, the request information of all processes is made available to all the aggregators. Hence, with MDF each aggregator can calculate and sort the stripes in its file domain independently from other aggregators. Once the scheduling is determined, the two phases are carried out alternatively, one stripe at a time.

Solely utilizing the access degree may not always give the minimal $\overline{T_c}$. For example, the first three stripes of a file are written by P_0 and each of the successive 12 stripes is written by P_1, P_2 and P_3, in an interleaving manner. If there are three aggregators, then each has a file domain consisting of 5 stripes in which the access degree is 1 for the first stripe, and 3 for the remaining four stripes, as illustrated in Figure 2(a). In the MDF algorithm,

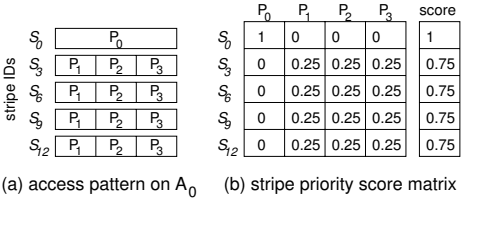


Fig. 2. An example access pattern and the weight score assignment by the LW-MDF method

P_0's requests are served last and $\overline{T_c} = (5t + 3 \times 4t)/4 = 4.25t$. However, if P_0's requests are served first then $\overline{T_c} = (t + 3 \times 5t)/4 = 4t$. Hence, solely depending on the stripe's degree is not sufficient to achieve the best response time. We propose two additional algorithms with weighted schemes.

Locally Weighted Most Degree First (LW-MDF). As the assumption in the MDF method that each process has equal contribution to the stripe scheduling priority may not produce the best result, the LW-MDF method is designed to assign a weight to each process based on its number of requests in an aggregator's file domain. In each aggregator, the weight of a process is set to the inverse of the number of stripes accessed on that aggregator. For example, in Figure 2, the weight is 1 for P_0, and 0.25 for the others. The priority score of a stripe is then calculated as the sum of all process weights on that stripe. Note that the scores only depend on the local data access pattern on the aggregators. As a result, the LW-MDF method assigns the higher priority to stripe S_0 than other stripes.

In the LW-MDF method, the weights are calculated using only the local information on each aggregator. Consider a case that process P_0 accesses only two aggregators, for instance two stripes on aggregator A_0 and six stripes on aggregator A_1. The weights assigned to P_0 on both aggregators will be $\frac{1}{2}$ and $\frac{1}{6}$, respectively. If A_0 schedules P_0's stripes first but A_1 schedules P_0's stripes later, then P_0's collective I/O will not complete until the six stripe requests on A_1 are processed. In order to deal with this problem, we propose another variant of MDF algorithm that considers the weights of a process across all aggregators.

Globally Weighted Most Degree First (GW-MDF). When a process has a higher number of accesses to an aggregator, it makes little sense to schedule its requests first on other aggregators, as the response time of a process is determined by the slowest aggregator that serves its requests. The GW-MDF method selects the minimum of local weights of a process across all the aggregators to calculate the priority scores for stripes.

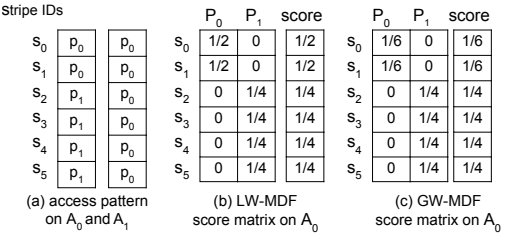


Fig. 3. An example access pattern and the weight score assignment by the GW-MDF method

Consider the example case presented in Figure 3. Process P_0 accesses two stripes on aggregator A_0, and six on aggregator A_1, the weight of P_0 on A_0 is $\frac{1}{2}$ according to LW-MDF and $\frac{1}{6}$ according to GW-MDF. There is another process P_1 that only accesses 4 stripes on aggregator A_0, the weight of P_1 on A_0 is $\frac{1}{4}$. In GW-MDF, the weight of P_1 is higher than P_0's weight ($\frac{1}{6}$), thus, P_1's requests are served first. The average response time $\overline{T_c} = (6t + 4t)/2 = 5t$. However, in LW-MDF, P_0 and P_1 will complete at the same time, and $\overline{T_c} = (6t + 6t)/2 = 6t$.

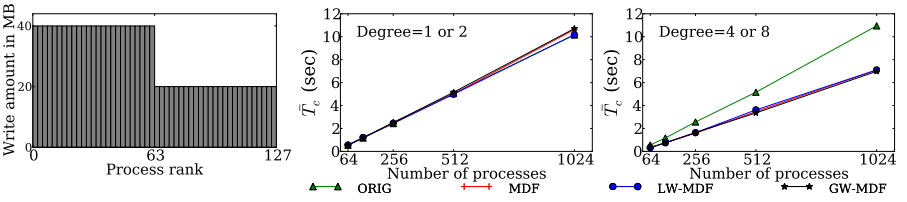
Therefore, GW-MDF achieves the optimal scheduling in this case. One thing to note, if the processor doesn't have any stripe to access, its score is assigned to zero rather than infinity. Our score algorithm only needs to consider the positive values to exclude the processes with zero score. For highly irregular or unbalanced data access patterns, it is anticipated that GW-MDF can outperform the other two MDF methods. However, in order to find the global minimums, there is an extra communication for gathering the local weights of all processes on every aggregator. This global communication among aggregators adds an overhead to the overall performance.

3 Performance Evaluation

All the proposed scheduling algorithms are implemented in the ROMIO library released along with MPICH version 1.3.2p1. By using two artificial benchmarks and a climate simulation application, our evaluation was carried out on Franklin, a Cray XT4 supercomputer at NERSC. Franklin consists of 9572 computer nodes, each of which runs a 2.3 GHz quad-core AMD Opteron processor and 8 GB memory. The parallel file system is Lustre version 2.2.48 with total of 48 OSTs. In our experiments, the stripe size was configured to 1 MB and the numbers of OSTs are set to 8 and 40 for the artificial benchmarks and GCRM evaluation, respectively. The performance results of MPI collective write operations are presented in this work.

Fixed Uneven Workload. We assign the first half of the processes twice the write amount as the other half. The access pattern is illustrated as an example shown in Figure 4(a), where the first half of the 128 processes write 40 MB of data each, and the second half writes 20 MB. The write amount on each process is further partitioned into 160 smaller pieces whose offsets are interleaved among all processes. For each piece in the first 64 processes, the size is $\frac{1}{4}$ MB, and for each piece in the second 64 processes is $\frac{1}{8}$ MB. This setting produces multiple noncontiguous file regions for each process to access and each stripe is accessed by more than one process. In addition, each process accesses the same number of stripes on each aggregator. This pattern implies that all MDF methods have the same weights and should have similar performance. The results presented in Figure 4(b) clearly demonstrate that the proposed scheduling methods outperform the traditional scheduling of the increasing file offset order. With the experiments running on up to 1024 processes, we observe that all four scheduling methods have the same wall time for the slowest processes. The slowest process is one of the aggregators whose file domain has the most stripes. All MDF methods show the similar improvement as the weights of a process, which contribute to the priority are the same for all stripes. In this case, the priorities are determined by the local access degrees. A reduction of up to 30% in average response time is obtained.

Random workload. Given a fixed file size, the random workload partitions the entire file into pieces with arbitrary lengths which are then assigned to processes based on the Gaussian distribution. Figure 5(a) shows the random workload



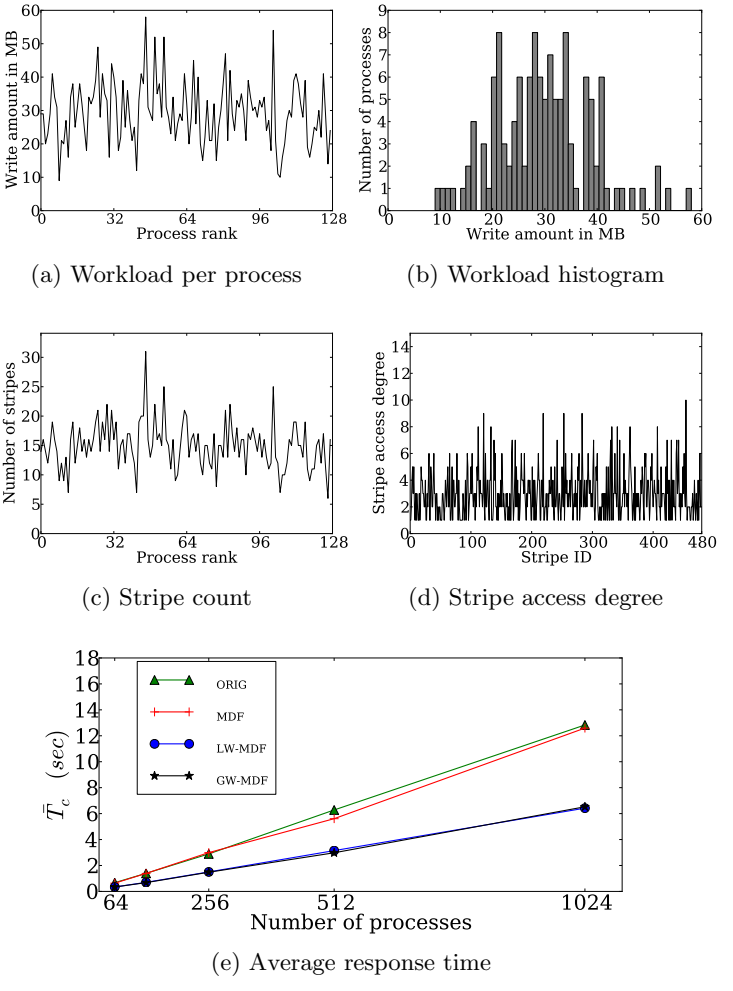
(a) Workload distribution (b) Average response time

Fig. 4. Access pattern and performance results for the fixed uneven workload

distribution among 128 processes, and Figure 5(b) shows the corresponding histogram. Each process has a random amount of data to write and the processes with ranks close to the median are assigned more workload. Figure 5(c) and (d) illustrate the number of stripes per process and stripe access degree per process, respectively. Each piece of write segment is randomly assigned to a process such that the number of stripes accessed by a process obeys the Gaussian distribution. The stripe degree is also random because the size of each piece is arbitrary. The results in Figure 5(e) indicate that both LW-MDF and GW-MDF perform better than MDF by up to 50%. MDF does not have a noticeable improvement over the original scheduling as the number of stripe accesses in this random pattern varies significantly with the number of processes.

Global Cloud Resolving Model (GCRM). The GCRM is a climate application framework designed to simulate the circulations associated with large convective clouds [9]. Its I/O uses Geodesic Parallel I/O (GIO) library [10], which interfaces parallel netCDF (PnetCDF) [11]. In our experiments, we enable the PnetCDF non-blocking I/O option to aggregate multiple grid data variables into large-sized collective writes. Non-grid variables are excluded from our evaluation, as they are written individually, which does generate uneven file access degrees. There are 11 grid variables and each variable is approximately evenly partitioned among all the processes. We collected results for 3 cases: 640 processes with resolution level 9, 1280 processes with level 10, and 2560 processes with level 11. Resolution levels 9, 10, and 11 correspond to the geodesic grid refinement at about 15.6, 7.8, and 3.9 km, respectively. Figure 6(a) and (b) show the I/O pattern for the 1280-process case and indicates about 80% of the processes access 13 file stripes. The majority of the processes have the same request length, and only a few processes have smaller request lengths. There are two peaks of stripe access degree in Figure 6(c) at stripe ID 0 and 1441 because a few small sized grid variables (40 MB each) are written at these file offsets. Since each variable is evenly partitioned among 1280 processes, there are 32 processes accessing the same stripe, as shown by the right-most bar in the histogram chart, Figure 6(d). The histogram also shows that more than half of the stripes are accessed by 6 distinct processes, while others have even higher degrees.

Both MDF and LW-MDF yield similar performance in Figure 6(e) because more than 80% of the processes have the same request count and hence share



(a) Workload per process (b) Workload histogram

(c) Stripe count (d) Stripe access degree

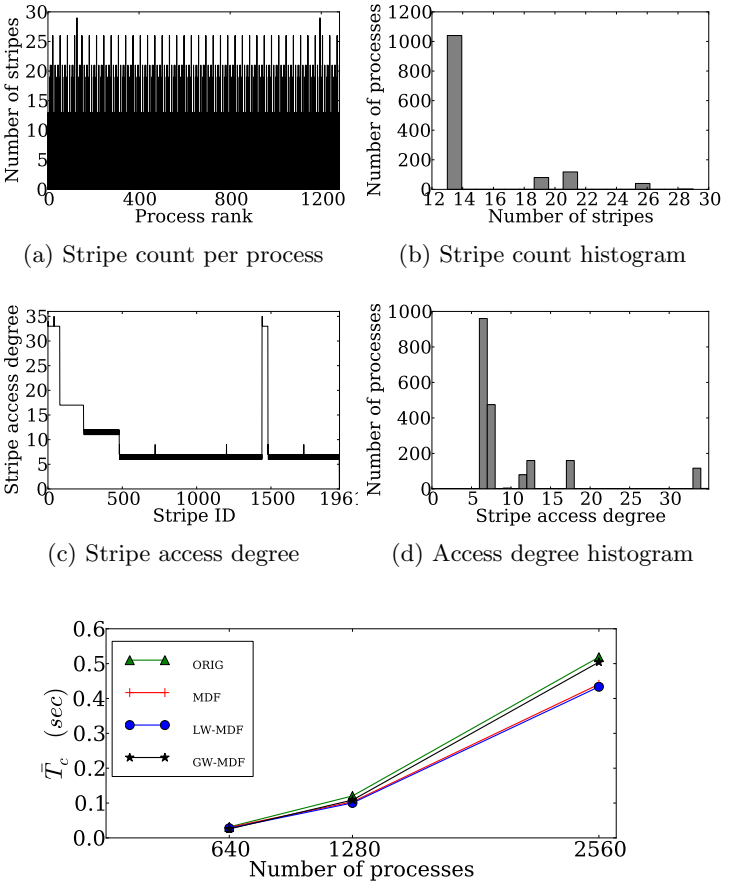
(e) Average response time

Fig. 5. Access pattern and performance results for the random workload

the same weights. Improvement in the average response time is approximately 20%. Compared with the random workload distribution, GCRM's access pattern is relatively regular. As described earlier, GW-MDF requires an additional global communication for finding the minimal weights. However, the benefit of using the global weights is not significant enough to outperform the communication overhead. The similar results between the MDF and the original methods attribute to the balanced I/O workload in the GCRM.

4 Related Work

Scheduling parallel I/O operations has been studied by many researchers to address the speed gap between the CPU and I/O systems. Three off-line heuristics



(a) Stripe count per process (b) Stripe count histogram

(c) Stripe access degree (d) Access degree histogram

(e) Average response time

Fig. 6. Access pattern and performance results of GCRM

based on graph coloring algorithms are developed to formalize the simultaneous resource scheduling problem [12]. Based on this work, a distributed randomized version using edge coloring method is proposed in [13]. Distributed I/O scheduling in the presence of data replication is presented in [14]. Decentralized I/O scheduling strategies between computer nodes and I/O servers for parallel file systems are developed in [15]. [16] has proposed three different techniques to increase the write bandwidth for collective I/O. The first technique is similar to two-phase collective I/O which aggregate I/O requests from participating processes such that the number of I/O operations provided to the underlying parallel file systems can be minimized. The second technique is to use a designated root process gathering data from all the processes, thus, the communication

parallelization can be better utilized in some degree. In the third method, each process writes out data independently. However, there is no I/O scheduling algorithm employed in any of the proposed methods.

5 Conclusion

In collective I/O operations, different I/O request scheduling strategies can give different response time. We use the stripe access degree and request count per process on the I/O aggregators to develop algorithms that improve the average response time of collective I/O operations. Reducing the average response time in collective I/O operations equivalently increases the computational resource utilization in high-performance computing systems. Our performance results show significant improvement in average response time for various data access patterns in collective write operations. In the future, we plan to apply similar approaches for read operations and develop different scheduling methods for different parallel file systems.

Acknowledgment. This work is supported in part by NSF award numbers: OCI-0724599, CNS-0830927, CCF-0621443, CCF-0833131, CCF-0938000, CCF-1029166, and CCF-1043085 and in part by DOE grants DE-FC02-07ER25808, DE-FG02-08ER25848, DE-SC0001283, DE-SC0005309, and DE-SC0005340. A portion of this work was performed under project 57746 funded by DOE's Office of Science under the Scientific Discovery through Advanced Computing program. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

1. del Rosario, J., Brodawekar, R., Choudhary, A.: Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In: The Workshop on I/O in Parallel Computer Systems at IPPS (1993)
2. Kotz, D.: Disk-directed I/O for MIMD Multiprocessors. ACM Transactions on Computer Systems 15(1), 41–74 (1997)
3. Seamons, K., Chen, Y., Jones, P., Jozwiak, J., Winslett, M.: Server-directed Collective I/O in Panda. In: Supercomputing (November 1995)
4. Thakur, R., Gropp, W., Lusk, E.: Users Guide for ROMIO. Technical Report ANL/MCS-TM-234, Argonne National Laboratory (October 1997)
5. Thakur, R., Gropp, W., Lusk, E.: Data Sieving and Collective I/O in ROMIO. In: The Symposium on the Frontiers of Massively Parallel Computation (1999)
6. Ying, L.: Lustre ADIO Collective Write Driver. Lustre Technical White Paper (September 2008)
7. Liao, W., Choudhary, A.: Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols. In: SuperComputing Conference (2008)
8. Liao, W.: Design and Evaluation of MPI File Domain Partitioning Methods under Extent-Based File Locking Protocol. IEEE Transactions on Parallel and Distributed Systems 22(2), 260–272 (2011)

9. Randall, D., Khairoutdinov, M., Arakawa, A., Grabowski, W.: Breaking the Cloud Parameterization Deadlock. Bull. Amer. Meteor. Soc. 84, 1547–1564 (2003)
10. Schuchardt, K., Palmer, B., Daily, J., Elsethagen, T., Koontz, A.: IO Strategies and Data Services for Petascale Data Sets from a Global Cloud Resolving Model. Journal of Physics: Conference Series 78 (2007)
11. Li, J., et al.: Parallel netCDF: A High-Performance Scientific I/O Interface. In: SuperComputing Conference (2003)
12. Jain, R., Somalwar, K., Werth, J., Browne, J.: Scheduling Parallel I/O Operations in Multiple Bus Systems. Journal of Parallel and Distributed Computing 16(4), 352–362 (1992)
13. Durand, D., Jain, A., Tseytlin, D.: Applying Randomized Edge Coloring Algorithms to Distributed Communication: An Experimental Study. In: SPAA (1995)
14. Wu, J., Lin, Y., Liu, P.: Efficient Distributed Algorithms for Parallel I/O Scheduling. In: International Conference on Parallel and Distributed Systems (2005)
15. Isaila, F., Singh, D., Carretero, J., Garcia, F.: On Evaluating Decentralized Parallel I/O Scheduling Strategies for Parallel File Systems. In: VECPAR (2006)
16. Chaarawi, M., Chandok, S., Gabriel, E.: Performance Evaluation of Collective Write Algorithms in MPI I/O. In: Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2009. LNCS, vol. 5544, pp. 185–194. Springer, Heidelberg (2009)