On evaluating decentralized parallel I/O scheduling strategies for parallel file systems

Florin Isailă, David Singh, Jesús Carretero, and Félix Garcia

Department of Compute Science,
University Carlos III de Madrid, Spain
{florin,desingh,jcarrete,fgarcia}@arcos.inf.uc3m.es

Abstract. This paper evaluates the impact of the parallel I/O scheduling strategy on the performance of the file access in a parallel file system for clusters of commodity computers (Clusterfile). We argue that the parallel I/O scheduling strategy should be seen as a complement to other file access optimizations like striping over several I/O servers, non-contiguous I/O and collective I/O. Our study is based on three simple decentralized parallel I/O heuristics implemented inside Clusterfile. The measurements in a real environment show that the performance of parallel file access may vary with as much as 86% for writing and 804% for reading with the employed heuristic and with the schedule block granularity.

1 Introduction

The performance of applications accessing large data sets is often limited by the speed of I/O subsystems. On one hand, this limitation comes from the ever increasing discrepancy between processor, memory speed and magnetic disks. On the other hand, the potential for parallelism existent in clusters of commodity computers and supercomputers is not always fully exploited by the I/O system software, like the parallel file systems [1–11] and libraries [12, 13]. These systems employ mechanisms such as striping a file over several independent disks managed by I/O nodes and allowing parallel file access from several compute nodes.

For a better utilization of network and storage resources several point-to-point non-contiguous I/O methods have been proposed: data sieving [14], list I/O [15], view I/O [16]. These methods greedily optimize the communication between exactly one pair compute node - I/O node without regard at the global system performance. The collective I/O methods two-phase I/O [17] and disk-directed I/O [18] use collective buffers in order to gather the requests from compute nodes before sending them to disks. For disk-directed I/O the collective buffers reside at I/O nodes, whereas for two-phase I/O at intermediary compute nodes. Both of these methods describe how the data flows through the system between compute nodes and I/O nodes, but do not say anything about the order in which requests are sent between parallel running compute nodes and I/O nodes. However, an improper request ordering may cause idleness, load imbalance or resource contention, which may have a tremendous impact on performance.

The parallel I/O scheduling strategy may be seen as a complement to the above mentioned I/O optimizations. File striping describes a parallel spatial data placement, whereas the parallel I/O strategy decides the temporal order of parallel requests. Non-contiguous I/O methods gather small messages into larger ones, while the parallel I/O strategy targets to schedule requests with sizes and in an order that optimize the resource usage. For collective I/O methods, the scheduling strategy may intervene both in the process of gathering the data into collective buffers, as well as in the sending the collective buffers to the I/O servers.

In previous work [19] we have developed a decentralized parallel I/O scheduling strategy for collective I/O operations. However, this strategy is specialized for collective I/O operations, and the impact of this strategy on the global system performance was not evaluated.

In this paper we evaluate three simple decentralized parallel I/O scheduling strategies implemented in the Clusterfile [20] parallel file system. We show that optimizations like non-contiguous I/O and collective I/O can not achieve a high resource utilization without a proper parallel I/O scheduling strategy. Additionally, the choice of the proper strategy and proper schedule block size may have an important influence on the overall file system performance.

2 Parallel I/O scheduling problem

The parallel I/O scheduling problem is not new. It was formulated by Jain and et al. [21] as follows. Given n_p compute nodes, n_{IOS} I/O servers and a set of requests for transfers of the same length among compute nodes and I/O servers and assuming that a compute node and an I/O server can perform exactly one transfer at any given time, find a service order that minimizes the schedule length [21].

Figure 1 shows an example, in which $n_p = 2$ compute nodes simultaneously issue in order four requests T1, T2, T3, T4 for $n_{IOS} = 2$ I/O servers. For this set of requests, several schedules are possible under the assumption that for each pair compute node - I/O node, only one request can be serviced at a time. Two of them are shown in the figure. In "Schedule 1", T1 and T2 are serviced at time 0; subsequently, T3 and T4 can not be scheduled simultaneously, because they have the same destination. The resulting schedule has the length 3. If T4 and T1 are scheduled in the first phase, T2 and T3 can be executed in parallel in the second phase and the schedule length is 2 ("Schedule 2").

The general scheduling problem is shown to be NP-complete, which makes it impractical for the real medium size parallel systems. Consequently, all solutions presented in the related work section are based on heuristics that try to minimize the schedule length, but without guaranteeing that the optimal value is used.

3 Related work

The proposed solutions to the parallel I/O scheduling problem can be divided at least by five criteria. First, the proposed algorithms may be centralized or distributed. The centralized algorithms assume that there is a place in the system where all information about requests are gathered, before the schedule is computed. In the distributed algorithms the global schedule is computed in parallel by different nodes having only partial information about the requests. Second, some algorithms solve the parallel I/O schedule

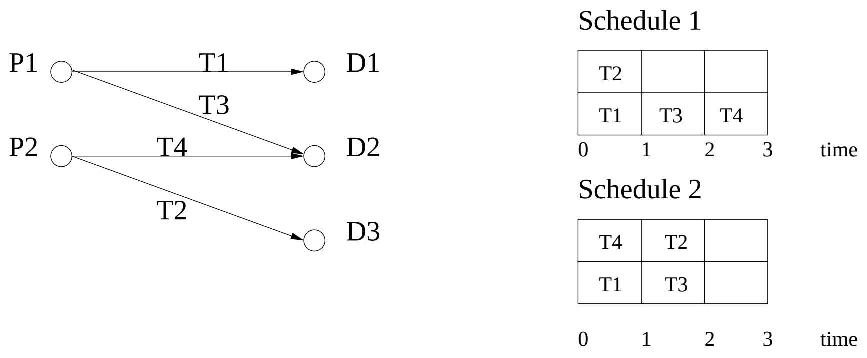

Fig. 1. Parallel I/O scheduling problem.

problem in the presence of replication and others consider that there is only one copy of the data in the system. Third, the algorithms may be off-line or on-line. In the off-line algorithms the schedule is computed based on the fact that all the request information is available and is executed as such. In the on-line algorithms requests generated during the execution trigger a re-computation of the schedule. Fourth, the algorithms may differentiate between data with and with-out real-time constraints. Fifth, the evaluation can be based on simulations or on real implementations and systems.

The strategies incorporated in Clusterfile and discussed in this paper are decentralized, without replication, off-line, without real-time constraints, implemented and evaluated in a real environment.

Jain et al. [21] were among the first that formalized the parallel I/O scheduling problem in absence of replication and proposed three centralized off-line heuristics based on graph coloring algorithms. In First-Come First-Serve (FCFS), in each phase, as many as possible requests are served in parallel (colored with the same color) in the order of their arrival. For Figure 1, if the order of request arrival is T1, T2, T3, T4, “Schedule 1” is produced. Highest Degree First (HDF) considers first the graph nodes with the higher degrees in order to schedule parallel transfers. Both schedules from Figure 1 may be produced. Highest Common Degree First (HCDF) processes first the graph edges with the higher sum of the node degrees in order to schedule parallel transfers. Only the optimal “Schedule 2” can be produced in the example from Figure 1. The evaluation is based on a simulation and shows as expected a superior performance for the “more-informed” HCDF heuristic.

Chen and Majumdar [22] evaluate five centralized parallel I/O scheduling strategies for clusters in the presence of replication. On one hand they add replication support to FCFS and HCDF. On the other hand they propose Lowest Destination Degree First (LDDF), Shortest Job First (SJF) and Shortest Outstanding I/O Demand Job First (SOJF). The strategies are evaluated on a real system for single job and multi-programmed workloads. An other real evaluation of five replication-based centralized parallel I/O scheduling strategies including those from [22] is presented in [23].

Durand et al. [24] propose distributed randomized parallel I/O scheduling algorithms based on edge colorings. In Uniformly at Random (UAR) each client selects randomly a request and sends it as a bid to an I/O server. Then each I/O server selects one received request at random and colors it with the current color. The algorithm repeats until all the graph is colored. Our implemented randomized strategy is a simplified version of this algorithm. MPASSES gives several (M) opportunities to color an edge to the clients whose proposal were rejected in the first pass of UAR. HDF is a distributed variant of the centralized HDF from [21] in which the clients send their degree together with the bid and the I/O servers picks up the client with the highest number of pending requests. The evaluation is based on a simulation.

In [25], the authors propose a decentralized update-based parallel I/O algorithm (D-SPTF) targeting load balance, efficient global cache exploitation and reducing disk positioning times for writing. The data may be replicated over several disks, which allows for an efficient read from the client which can serve the request fastest and a slow write due to the update of all replicas. Locality Aware Request Distribution (LARD) [26] requires a front-end which distributes the requests among the I/O servers according to the locality. Simulation results show that D-SPTF outperforms LARD and hash-based request distribution in terms of throughput and response time.

Lebre et al. [27] present the implementation and real system evaluation of two centralized parallel I/O strategies targeting global performance optimization and fairness in a multi-application environment. Their solution is a file system independent application-level library, whereas ours is done at file system level.

4 Parallel file system overview

Clusterfile(CLF) [20] is a parallel file system for clusters of commodity computers. The architecture is based on the classical parallel file system model, in which the files are declustered over several I/O nodes managed by I/O servers. The applications run on compute nodes and access the file system through a POSIX-like proprietary interface or a classical UNIX interface after mounting the file system. Each individual process may declare a file *view*, i.e. a *logical contiguous* window mapped onto a non-contiguous file region. After declaration, each view can be accessed like a regular file. Clusterfile performs efficient non-contiguous I/O through a method called view I/O, described in detail in [16].

Clusterfile integrates two well-known collective I/O techniques, disk-directed [18] and two-phase I/O [17], into a common design [19]. The collective buffers are stored into a global cache, managed in cooperation by several cache managers running a version of the decentralized hash-distributed cooperative caching algorithm presented in [29].

5 Goals

The parallel I/O scheduling presented in this paper are implemented inside a real parallel file system. The parallel file system consists of four types of components: several parallel acting clients, several I/O servers, several cache managers, one metadata manager. The interaction between these components even in a relatively small cluster is highly complex. Experience with the xFS file system [28] has shown that complex protocols may make the development of a parallel system very difficult. In fact, the initial proposal of xFS was never fully implemented in part due to the exponential explosion of protocol complexity, which made bug detection very challenging even with formal verification tools. Consequently, when adding adding a parallel I/O scheduling strategy to an existing complex system we have in mind *simplicity*.

The scheduling strategy should have a small overhead. On one hand, this overhead is proportional with the number of messages exchanged for taking a scheduling decision. Even though the latency of the network is low, the communication may cause side-effects like context switches or evictions affecting data locality. On the other hand, data replication would perform poor for file writing. We have chosen not to replicate the data inside Clusterfile. Eventual replication schemes could be implemented on top of the file system.

Some scheduling strategies presented in the related work section are centralized. However, gathering the scheduling information at a central point may be difficult. First, this involves communication that adds additional complexity to the existing distributed protocols. Second, the additional communication for gathering the requests from all nodes and distributing the decision makes the solution costly and non-scalable. Therefore, the scheduling I/O strategies we chose are decentralized.

6 Parallel scheduling I/O heuristics

For all parallel scheduling heuristics, we assume that, at a certain point in time, n_p compute nodes simultaneously issue large data requests for n_{IOS} I/O servers. The decision of the order of data service is taken by the compute node for writing and by the I/O for reading in a similar way. For this reason we describe here only the write scheduling strategy. For writing, large requests are split by each compute node into smaller requests of size b.

In the first scheduling strategy, *first-IOS* (I/O server), each compute node sends the data to the I/O nodes in the order of file offsets. This is a natural approach, but may pose the potential risk that all the compute nodes send the data to the same I/O node at the same instant. However, the load balance problem may be compensated by high data locality in the case of non-contiguous interleaved access, as will be shown in the evaluation section.

In the second write scheduling strategy, *random-IOS*, each compute node first builds a list of requests targeted to each I/O node. Then the compute node chooses randomly the I/O server to which the data will be send until all the data is sent.

The third scheduling strategy, *hash-IOS*, is the one employed for the collective I/O operations of Clusterfile [19]. Conforming to the theoretical problem definition, for which each compute node can perform exactly one transfer at any given time, at time step $t_j, j = 0, 1, ...$, the compute node i sends a block to the I/O server $(i + j)$ modulo n_{IOS}.

Figure 1 shows an example, in which $n_p = 2$ compute nodes simultaneously issue 4 requests for $n_{IOS} = 2$ I/O servers. For the first-IOS method, CN0 decides to send the request to the IOS0 first, and then to IOS1, and a schedule of length 3 is produced

("Schedule 1"). On the other hand, hash I/O produces a schedule of length 2 ("Schedule 2"), as the I/O servers may run in parallel. Random IOS may produce any possible schedule, depending on the generated random numbers.

Notice that, for all strategies, there is no central point of decision, each process acts independently.

7 Evaluation

We performed our experiments on a cluster of 16 dual processor Pentium III 800MHz, having 256 KBytes L2 cache and 1024 MB RAM, interconnected by Myrinet LANai 9 cards at 133 MHz, capable of sustaining a throughput of 2 GB/s in each direction. The machines are equipped with IDE disks and were running LINUX kernels version 2.6.13 with the *ext2* local file system. We used TCP/IP on top of the 2.0.24 version of the GM [31] communication library. The ttcp benchmark delivered a TCP/IP node-to-node throughput of 120 MB/sec.

The I/O scheduling heuristics are all implemented inside the Clusterfile parallel file system.

In the following two subsections, we present the results for two different workloads: a synthetic parallel benchmark accessing contiguously a file and BTIO [32], a NASA parallel benchmark, in which several processes write non-contiguously to a file and then read back the result.

7.1 Synthetic benchmark

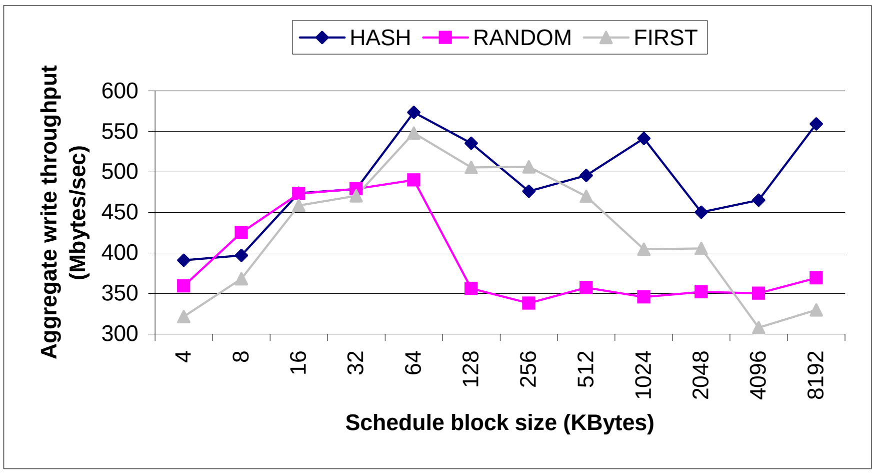

Fig. 2. Synthetic benchmark file write throughput.

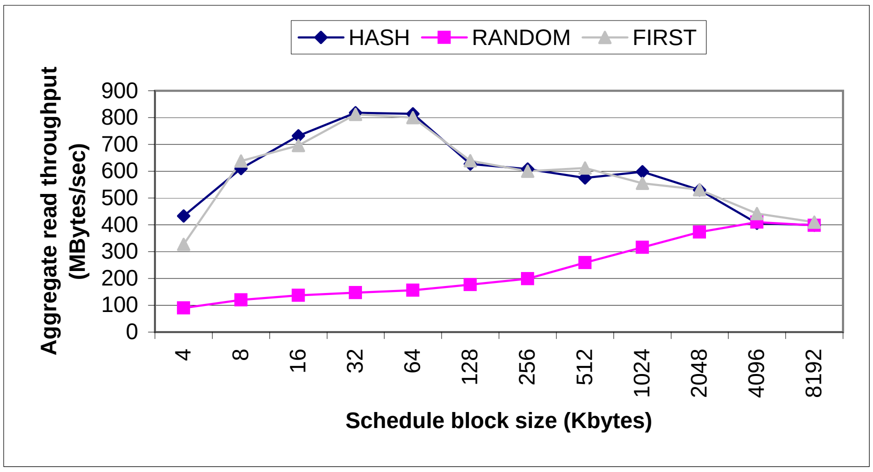

Fig. 3. Synthetic benchmark file read throughput.

We have written a synthetic benchmark in Message Passing Interface [30], in which all processes write and read in parallel different regions of the same file. The writes and reads are performed contiguously, as we first want to investigate the effect of parallel I/O heuristics on the performance, unaffected from the gather-scatter operations that are necessary for non-contiguous I/O.

Clusterfile uses 8 I/O server running on 8 I/O nodes. The file block size is 64 KBytes. In the benchmark each of the 8 compute nodes writes 32 MBytes, resulting in a total of 256 MBytes. Each measurement was repeated 5 times and the mean value is reported.

Figures 2 and 3 show the aggregate throughput in MBytes/second obtained employing the three parallel I/O scheduling heuristics for writing and reading, respectively. The x-axis values represent the length of schedule block b (as introduced in the previous section).

First of all, note that for diverse parameters the performance of the same application may vary by as much as 86% for writing and 804% for reading. For writing, the highest value is obtained for hash-IOS for $b = 64KBytes$ (573 MBytes/second) and the lowest for first-IOS for $b = 4096KBytes$ (308 MBytes/second). For reading, the highest value is obtained for hash-IOS for $b = 32KBytes$ (817 MBytes/second) and the lowest for first-IOS for $b = 4KBytes$ (90 MBytes/second).

As expected, for first-IOS strategy, the aggregate write throughput decreased with schedule block granularity. The reason is that all the I/O servers try to send the data in the same order to all the I/O servers, which creates contention at I/O servers. The contention prevents the compute nodes from advancing and employing the other available I/O servers.

We have expected that the random-IOS write performance results lie somewhat in the middle between the results of hash-IOS and first-IOS. Surprisingly, the random IOS heuristic outperformed first-IOS only for the smallest four and largest two values.

We believe that the reason lies in the fact that first-IOS generates a critical bottleneck only when accessing the first I/O server. The first compute node that “escapes” this bottleneck continues sending the data to the second I/O server and so on, generating a pipeline behavior. On the other hand, it appears that the randomly generated bottlenecks cause a higher overhead, as they can appear non-deterministically throughout the whole run of the application.

For large schedule block sizes, hash-IOS clearly outperforms the other two methods. For this heuristic each compute node starts by contacting a different I/O server which provides a good initial load balance, which is then preserved throughout the whole run by a cyclic access to the I/O servers.

The aggregate read throughput was similar for hash-IOS and first-IOS. This is due to the fact that in the present Clusterfile implementation an I/O server that receives the first request starts serving it. It appears that the initial potential bottleneck can not be overcome by the hash-IOS. This is unlike the write case, where the performance does not degrade with the the size of the schedule block.

The first-IOS and hash-IOS managed to exploit 85% of the theoretical point-to-point bandwidth of 8x120MBytes/second (as measured by the ttcp benchmark) for reading with $b = 32KBytes$ and $b = 64KBytes$. A further performance analysis is necessary in order to try to improve the write aggregate throughput.

7.2 BTIO benchmark

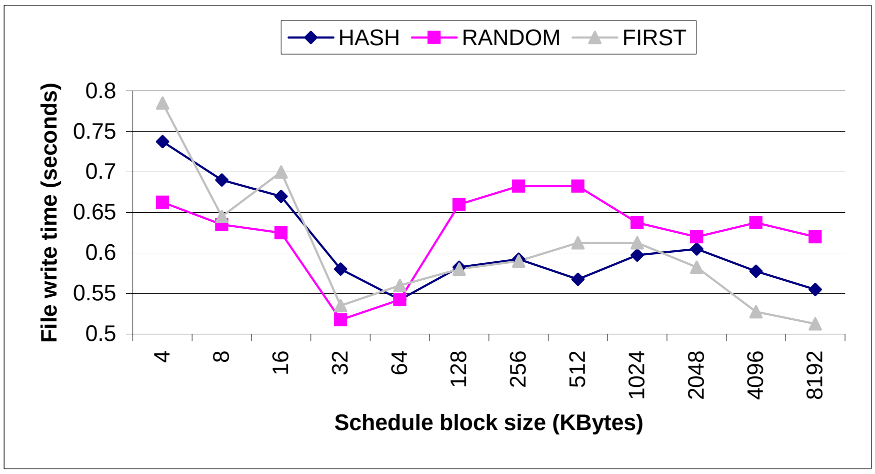

Fig. 4. BTIO file write times.

NASA’s BTIO benchmark [32] solves the Block-Tridiagonal (BT) problem, which employs a complex domain decomposition across a square number of compute nodes. Each compute node is responsible for multiple Cartesian subsets of the entire data set.

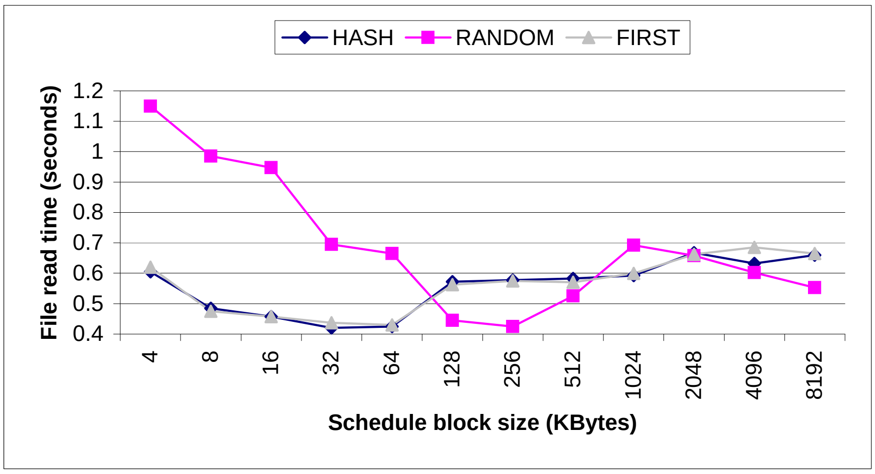

Fig. 5. BTIO file read times.

The execution alternates computation and I/O phases. Initially, all compute nodes collectively open a file and declare views on the relevant file regions. After each five computing steps the compute nodes write the solution to a file through a collective operation. There are three sizes of the data sets: A (419.43 MBytes), B (1697.93 MBytes) and C (6802.44 MBytes). For these classes the benchmark performs 200 compute steps and 40 I/O steps. We are interested only in the results for a single I/O phase writing 10.5 MBytes (A), 42.2 MBytes (B) and 170 MBytes (C). The parallel I/O scheduling policies are relevant for large amounts of data, therefore, we report in this paper the I/O access times of the C class data set. The access pattern of C class is nested-strided with a nesting depth of 2 with an access granularity of 3240 bytes. We report the results for 9 compute nodes and 9 I/O nodes in Figures 4 and 5 for writing and reading, respectively.

At the beginning of the BTIO benchmark, each process opens a file and declares a view on the file regions of interests. The individual file regions of the processes corresponding to the views do not overlap. Later, during each I/O phase, each process writes to the file through the view I/O method of Clusterfile [16]. Each process uses the view in order to contiguously send the data from each compute node to the I/O nodes. At I/O node, the data is scattered into the file blocks, kept in collective buffers. The reverse process takes place for reading. In a previous paper we have [19], we have shown that the combined view I/O and collective I/O method of Clusterfile significantly outperform two-phase I/O method of ROMIO [14], the most popular MPI-IO implementation. In the paper cited above the parallel scheduling strategy was fixed.

However, Figures 4 and 5 show that, depending on the parallel I/O scheduling strategy employed, the time to write a file in BTIO may vary with as much as 53% for writing (the ratio of 0.78 seconds for first-IOS with $b = 4KBytes$ to 0.51 seconds for first-IOS with $b = 8MBytes$) and 173% for reading (the ratio of 1.15 seconds for random-IOS with $b = 4KBytes$ to 0.42 seconds of hash-IOS with $b = 8MBytes$). As it can be noticed the performance span is not as large as in the case of the contiguous access. This is mainly due to the fact that the non-contiguous access generates a relatively constant overhead for scattering or gathering the data at the compute and I/O nodes.

8 Conclusions and current work

This paper presents and contrasts three parallel I/O scheduling heuristics implemented in Clusterfile parallel file system. The performance results show that the performance of parallel file access strongly depends on the choice of the parallel I/O scheduling strategy, as a combination of the employed heuristic and the granularity of the schedule. An improper scheduling strategy may result in inefficient utilization of the parallel network paths, poor load balance and high contention at I/O nodes.

The classical parallel I/O optimizations like data striping, non-contiguous I/O, collective I/O should be seen as a complement to a parallel I/O scheduling strategy. Our experiments have demonstrated that various simple parallel I/O scheduling strategies may produce a performance difference of as much as 53% for file writing and 173% for file reading over the above mentioned optimizations.

The decentralized strategies presented in this paper address I/O workloads of well-balanced parallel applications. For irregular applications, some form of centralization or communication between the application library and I/O servers would be needed. Our current work includes the design and analysis of strategies for this type of applications.

Acknowledgments

The authors want to thank the anonymous reviewers for the very useful comments and suggestions.

This work has been funded in part by the project TÉCNICAS DE OPTIMIZACIÓN Y FIABILIDAD PARA SISTEMAS DE ENTRADA/SALIDA ESCALABLES DE ALTAS PRESTACIONES (COMUNIDAD DE MADRID-UC3M)

References

1. DeBenedictis, E., Rosario, J.D.: nCUBE Parallel I/O Software. In: Proceedings of 11th International Phoenix Conference on Computers and Communication. (1992)
2. LoVerso, S., Isman, M., Nanopoulos, A., Nesheim, W., Milne, E., Wheeler, R.: sfs: A Parallel File System for the CM-5. In: Proceedings of the Summer 1993 USENIX Conference. (1993) 291–305
3. Huber, J., Elford, C., Reed, D., Chien, A., Blumenthal, D.: PPFS: A High Performance Portable File System. In: Proceedings of the 9th ACM International Conference on Supercomputing. (1995)
4. Corbett, P., Feitelson, D.: The Vesta Parallel File System. ACM Transactions on Computer Systems (1996)
5. Carretero, J., Serez, F., Miguel, P., Garca, F., Alonso, L.: ParFiSys: A Parallel File System for MPP. ACM SIGOPS **30** (1996)

6. Freedman, C., Burger, J., DeWitt, D.: SPIFFI-A Scalable Parallel File System for the Intel Paragon. IEEE Transactions on Parallel and Distributed Systems (1996)
7. Nieuwejaar, N., Kotz, D.: The Galley Parallel File System. Parallel Computing (1997)
8. O'Keefe, M.: Shared file systems and fibre channel. In: In the Proceedings of the Sixth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies. (1998)
9. Ligon, W., Ross, R.: An Overview of the Parallel Virtual File System. In: Proceedings of the Extreme Linux Workshop. (1999)
10. Schmuck, F., Haskin, R.: GPFS: A Shared-Disk File System for Large Computing Clusters. In: Proceedings of FAST. (2002)
11. Garcia-Carballeira, F., Calderon, A., Carretero, J., Fernandez, J., Perez, J.M.: The Design of the Expand Parallel File System. The International Journal of High Performance Computing Applications **17** (2003) 21–38
12. Winslett, M., Seamons, K., Chen, Y., Cho, Y., Kuo, S., Subramaniam, M.: The Panda library for parallel I/O of large multidimensional arrays. In: Proceedings of Scalable Parallel Libraries Conference III. (1996)
13. Message Passing Interface Forum: MPI2: Extensions to the Message Passing Interface. (1997)
14. Thakur, R., Gropp, W., Lusk, E.: Data Sieving and Collective I/O in ROMIO. In: Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation. (1999) 182–189
15. Thakur, R., Gropp, W., Lusk, E.: On Implementing MPI-IO Portably and with High Performance. In: Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems. (1999) 23–32
16. Isaila, F., Tichy, W.: View I/O:improving the performance of non-contiguous I/O. In: Third IEEE International Conference on Cluster Computing. (2003) 336–343
17. del Rosario, J., Bordawekar, R., Choudhary, A.: Improved parallel I/O via a two-phase run-time access strategy. In: Proc. of IPPS Workshop on Input/Output in Parallel Computer Systems. (1993)
18. Kotz, D.: Disk-directed I/O for MIMD Multiprocessors. In: Proc. of the First USENIX Symp. on Operating Systems Design and Implementation. (1994)
19. Isaila, F., Malpohl, G., Olaru, V., Szeder, G., Tichy, W.: Integrating Collective I/O and Cooperative Caching into the "Clusterfile" Parallel File System. In: Proceedings of ACM International Conference on Supercomputing (ICS), ACM Press (2004) 315–324
20. Isaila, F., Tichy, W.: Clusterfile: A flexible physical layout parallel file system. Concurrency and Computation: Practice and Experience **15** (2003) 653–679
21. Jain, R., Somalwar, K., Werth, J., Browne, J.C.: Heuristics for scheduling I/O operations. IEEE Transactions on Parallel and Distributed Systems **8** (1997) 310–320
22. Chen, F., Majumdar, S.: Performance of parallel I/O scheduling strategies on a network of workstations. In: Proceedings of ICPADS 2001. (2001) 157–164
23. Abawajy, J.H.: Performance Analysis of Parallel I/O Scheduling Approaches on Cluster Computing Systems. In: CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid, Washington, DC, USA, IEEE Computer Society (2003) 724
24. Durand, D., Jain, R., Tseytlin, D.: Parallel I/O scheduling using randomized, distributed edge coloring algorithms. J. Parallel Distrib. Comput. **63** (2003) 611–618
25. Lumb, C.R., Golding, R.A., Ganger, G.R.: D-SPTF: decentralized request distribution in brick-based storage systems. In: ASPLOS. (2004) 37–47
26. Pai, V., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., Nahum, E.: Locality-Aware Request Distribution in Cluster-based Network Servers. In: Proceedings of the ACM Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII) . (1998)

27. Lebre, A., Denneulin, Y., Van, T.T.: Controlling and Scheduling Parallel I/O in Multi-application Environments. Technical report, INRIA (2005)
28. Wang, R.Y., Anderson, T.E., Dahlin, M.D.: Experience with a distributed file system implementation with adaptive. Technical report (1998)
29. Dahlin, M., Yang, R., Anderson, T., Patterson, D.: Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In: The First Symp. on Operating Systems Design and Implementation. (1994)
30. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. (1995)
31. Myricom. GM: the low-level message-passing system for Myrinet networks: http://www.myri.com/. (2000)
32. Wong, P., der Wijngaart, R.: NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, NASA Ames Research Center, Moffet Field, CA (2003)