Integrating Collective I/O and Cooperative Caching into the "Clusterfile" Parallel File System

Florin Isaila, Guido Malpohl, Vlad Olaru, Gabor Szeder, Walter Tichy
Departement of Computer Science, University of Karlsruhe
Fasanengarten 5, 76137 Karlsruhe, Germany
{florin,malpohl,olaru,szeder,tichy}@ipd.uni-karlsruhe.de

ABSTRACT

This paper presents the integration of two collective I/O techniques into the Clusterfile parallel file system : disk-directed I/O and two-phase I/O. We show that global cooperative cache management improves the collective I/O performance. The solution focuses on integrating disk parallelism with other types of parallelism: memory (by buffering and caching on several nodes), network (by parallel I/O scheduling strategies) and processors (by redistributing the I/O related computation over several nodes). The performance results show considerable throughput increases over ROMIO's extended two-phase I/O.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*distributed memories, storage hierarchy*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; D.4.8 [**Operating Systems**]: Performance—*measurements*

General Terms

Design, Measurement, Performance

Keywords

parallel file systems, parallel I/O, collective I/O, cooperative caches, non-contiguous I/O

1. INTRODUCTION

The performance of applications accessing large data sets is often limited by the speed of the I/O subsystems. Studies of I/O intensive parallel scientific applications [21, 24] have shown that an important performance penalty stems from the mismatch of the file striping (physical parallelism) and the access patterns (logical parallelism). In this context, it is important how the software layers between applications and disks, namely I/O libraries like MPI-IO [20] and file systems [6, 11, 22, 12], use the inherent system parallelism. An efficient solution requires an integrated approach at all levels of the system, including system software and applications [16].

The above mentioned studies have found that the processes of a parallel application frequently access a common data set by issuing a large number of small non-contiguous I/O requests. Collective I/O addresses this problem by merging small individual requests into larger global requests in order to optimize the network and disk performance. Depending on the place where the request merging occurs, one can identify two collective I/O methods. If the requests are merged at the nodes with disks the method is called *disk-directed I/O* [17, 23]. If the merging occurs at intermediary nodes or at the compute nodes the method is called *two-phase I/O* [10, 3]. The main advantage of disk-directed is that data travels once through the network. The main drawback is that a large number of small requests may involve a significant computing and copying overhead solely at the nodes with disks. On the other hand, two-phase I/O allows distributing this overhead over several nodes at the cost of a second network transfer.

1.1 Contributions of this paper

Existing collective I/O implementations use either disk-directed or two-phase I/O. In this paper we propose, to the best of our knowledge for the first time, the integration of the two methods into a common design. This approach allows applications to use the most appropriate method according to their needs.

The paper estimates the effect of a global cooperative cache on the collective I/O operations. We are aware of a single study based on simulations that addresses the issue [2]. The global cache is used by the collective I/O implementation for two purposes: for redistribution of scatter-gather costs from I/O servers to compute nodes and for implementing hot collective buffers, i.e. buffers that are not freed after the collective operation finishes.

We introduce a decentralized parallel I/O scheduling heuristic used by the collective I/O implementation. Known parallel I/O scheduling strategies [16, 4] assume a central point of decision. The role of the parallel I/O scheduling policy is to establish an order of request execution that can efficiently exploit the potential parallelism inside the file system.

All results are based on an implementation inside Clusterfile parallel file system.

1.2 Roadmap

The rest of the paper is structured as follows. Preliminar-

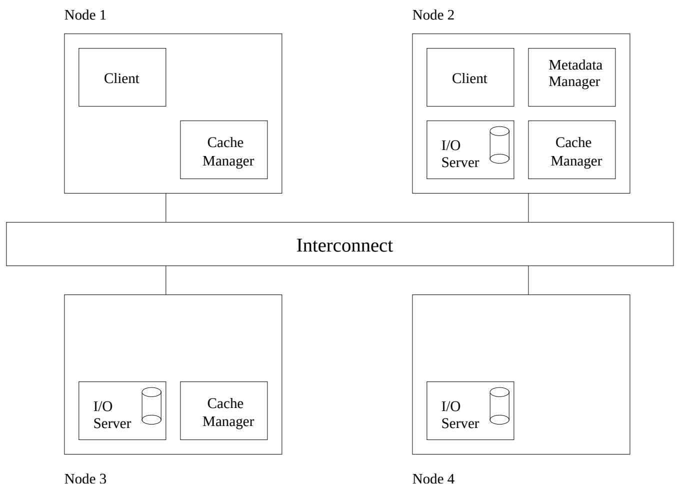


Figure 1: Clusterfile instalation example

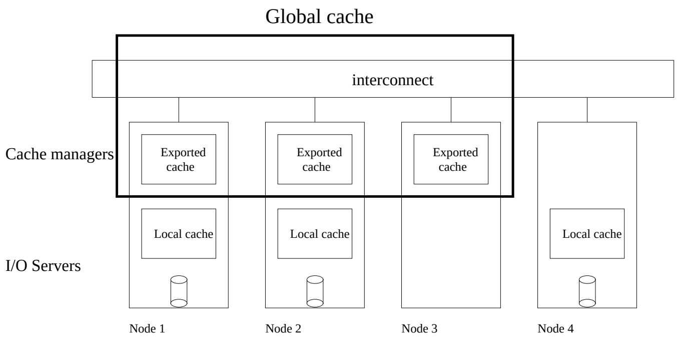


Figure 2: The cache hierarchy of Clusterfile

ies and related work are subject of section 2. An overview of the Clusterfile parallel file system, including the new global cache, is presented in section 3. Section 4 introduces a novel non-centralized parallel I/O scheduling heuristic. Clusterfile's collective I/O operations are discussed in section 5. The experimental results are presented in section 6. Finally, we summarize in section 7.

2. PRELIMINARIES AND RELATED WORK

In the generic, high-level parallel file system architecture, a parallel computer consists of two sets of nodes, which may potentially overlap: *compute nodes* (CN) and *I/O nodes* [13, 9, 11, 6, 12]. The applications run on compute nodes and access disks attached to the I/O nodes. Traditionally, the I/O nodes are managed by an *I/O server* (IOS) that allows sharing of local disks among compute nodes. The files in these systems are striped block-wise across several disks.

xFS [1] uses a log-structured approach. Each compute node writes its files to a log, which is striped over the available disks. This approach allows to maximize the parallelism for compute node writes, but offers no guarantees for read parallelism. Parallel reads are boosted by cooperative caching [8] (joint management of individual caches). Other systems using cooperative caching are PGMS [27], PPFS [11] and PACA [7]. However, none of these research groups have investigated the impact of cooperative caches on collective I/O.

There are several collective I/O implementations. In disk-directed I/O [17], the compute nodes send the requests directly to the I/O nodes. The I/O nodes merge and sort the requests and send them to disk. In server-directed I/O of Panda [23], the I/O nodes sort the requests on file offsets instead of disk addresses. Two-phase I/O [10, 3] consists of an access phase, in which compute nodes exchange data with the file system according to the file layout, and a shuffle phase, in which compute nodes redistribute the data among each other according to the access pattern.

ROMIO [25], a MPI-IO implementation, adds two optimizations to the tho-phase I/O method: data sieving (accessing contiguous intervals and filtering the requested data) and balancing the access size over several processors in the I/O phase. In section 5, we describe in detail ROMIO's extended two-phase method that serves as a basis of comparison with our implementation. ROMIO can be implemented on top of different file systems. This paper evaluates the implementations over PVFS [12] and Clusterfile [14] parallel file systems.

3. PARALLEL FILE SYSTEM OVERVIEW

In the initial design, Clusterfile [14] consisted of a metadata manager, n_{IOS} I/O servers and a parallel I/O library.

Both physical and logical partitioning use the same file model. A file is physically partitioned into *subfiles* stored at I/O servers and may be logically partitioned among several compute nodes by *views* (as described in subsection 3.2). Views are implemented inside the file system, as opposed to MPI-IO views [20], which are implemented on top of file system.

The subfiles or views partitions may be achieved either through MPI data types [19] or through an equivalent, Clusterfile native data representation [14]. File inodes are managed by a central metadata manager. Data and metadata management are separated, therefore data does not travel through the metadata manager. The centralized metadata is of no concern to this paper as we concentrate on large data sets. Ongoing research efforts are investigating ways to distribute and replicate metadata.

For simplicity reasons, throughout the paper we consider that a file consists of a single subfile striped round-robin over all I/O servers, i.e. the x-th block of a file is stored at I/O server x modulo n_{IOS}.

The new design adds n_{CM} cache managers (CM) that cooperate in order to implement a global cache. Figure 1 illustrates a possible installation of Clusterfile. Figure 2 shows the cache hierarchy of Clusterfile. Each I/O server manages a local file cache, storing only local disk blocks. The global cache is the higher level and consists of distributed memories managed in cooperation by the cache managers.

Figure 3 depicts the potential for parallelism inside Clusterfile. A parallel file access (logical parallelism) may translate into a parallel global memory access over parallel network links to several cache managers. Cache managers retrieve their data from I/O servers through parallel network connections that access parallel local caches, and, finally, parallel disks.

3.1 The global cache

The main advantage of a global cache is its potential to scale up to the whole aggregate physical memory of a parallel

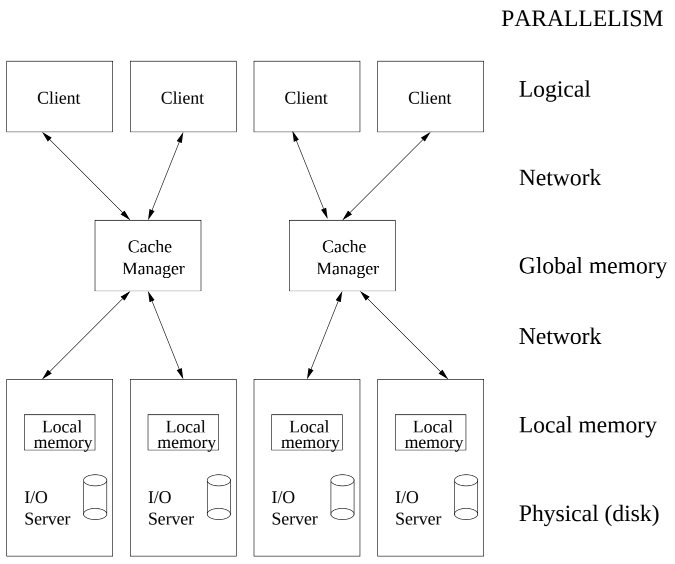


Figure 3: The potential for parallelism inside Clusterfile

computer. The underutilized memory of some nodes can be employed on behalf of other nodes. In our two-phase collective I/O implementation the file blocks are moved from I/O servers to cache managers in order to redistribute the scatter-gather costs.

Each file block is identified inside the global cache by the triplet $(ios, inode, file\ offset)$, where the ios is the I/O server that stores the file block and $inode$ is a unique file system-wide inode number assigned by the metadata manager at file creation.

Given x-th block of a file, the cache manager x modulo n_{CM} either caches the data block or knows where the block resides. When a read request arrives, if the CM has the block it delivers the requested data (not the whole block). If not, the CM brings the block locally and then delivers the data. This approach is tailored for the collective I/O operations, known to show a high temporal and spatial locality across parallel processes [21, 24]. This means that it is probable that in the near future, the same block will be accessed by another compute node and at that time the block will be already available at the CM.

This lookup policy is a variant of Hash Distributed Caching (HDC) as presented by Dahlin et al. [8]. In the original HDC the blocks were hashed based on their disk address. In our case, because the spatial and temporal locality refers to the file positions and not to the disk addresses, the blocks are cached based on their file offset (file offsets are mapped onto disk addresses at I/O servers). Additionally, the likelihood of true parallel access is increased by distributing consecutive file blocks (instead of consecutive disk blocks) to different cache managers.

In order to keep the protocol as simple as possible and to avoid cache coherency problems we implemented a "single-replica" global cache as in PACA [7]. At any time, there is at most one copy of the block in the global cache. Replication could be implemented on top of this global cache.

HDC is suitable for parallel workloads. The compute nodes can guess the likely place of a block without asking a central server. That helps in decongesting the disk nodes

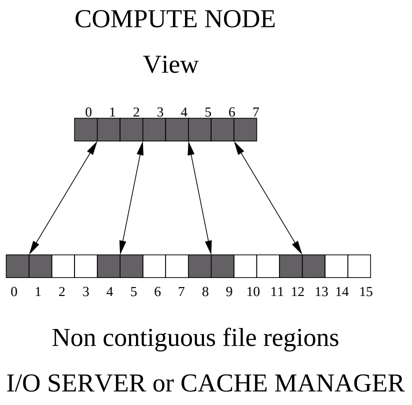


Figure 4: View I/O

by lowering their I/O server load. Section 5 discusses the consequences for the collective operations.

3.2 Non-contiguous I/O

Our non-contiguous I/O method is called view I/O and is presented in another paper [15]. We shortly describe it here, because it is a part of the collective I/O implementation as detailed in section 5. A *view* is a contiguous window to potentially non-contiguous regions of a file. By default, when a view is not explicitely set, a process "sees" the whole file. Figure 4 shows how a view maps on a non-contiguous region of a file. When a view is declared, the mapping information is transfered to the I/O node or cache manager, where it is stored for subsequent use. At access time, only the view interval is sent to the I/O node. For instance, for reading the first three non-contiguous segments a compute node sends a read request for $(0, 5)$ and the I/O server returns the data after performing a gather operation by using the previously stored view mapping.

The scatter/gather operations may become costly both in terms of computations and copying. If no cache managers are used, this overhead is paid at the I/O servers. On the other hand, if $n_{CM} > n_{IOS}$, the scatter/gather operations corresponding to a file can be distributed over the nodes where the data is cached, increasing the parallel execution degree with $n_{CM} - n_{IOS}$. If the data is not cached at CM, the transfer from I/O servers pays off if the gain obtained from additional parallelism is large enough to outperform the case when all the requests are processed at I/O servers.

4. PARALLEL I/O SCHEDULING

Collective I/O operations may involve large transfers of data between pairs of resources, such as processors, memories and disks. File striping and parallel caching offer data

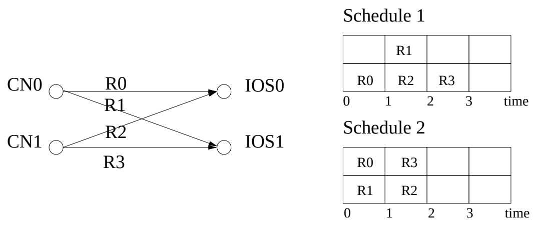


Figure 5: Parallel I/O scheduling example

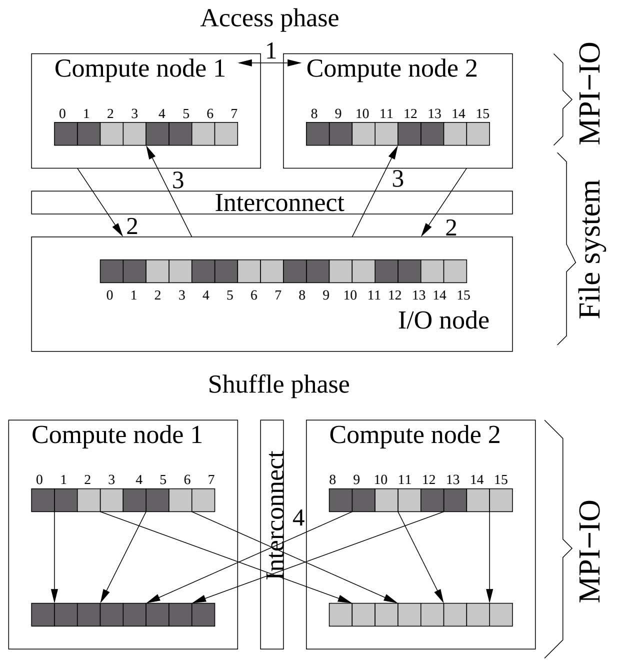


Figure 6: ROMIO's two-phase read example

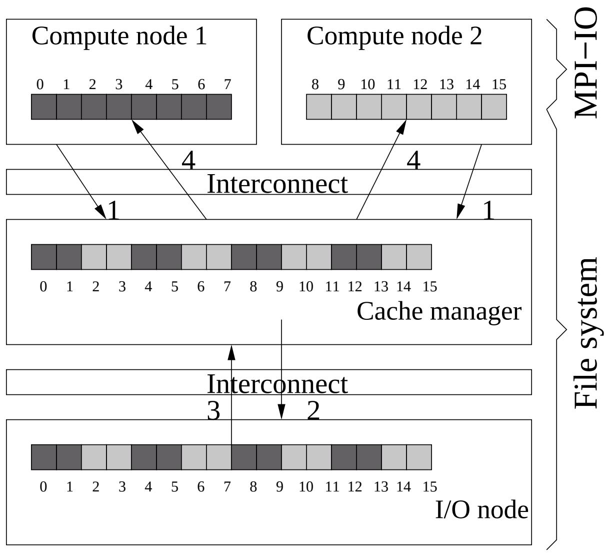


Figure 7: Clusterfile's collective read example

placement solutions that provide *potential* for parallelism. However, the effective use of this parallelism can be given only by the order in which these resources are involved, i.e. by a *parallel I/O schedule*.

The parallel I/O scheduling problem is formulated as follows. Given n_p compute nodes, n_{IOS} I/O servers and a set of requests for transfers of the same length between 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 [16]. As the general scheduling problem is shown to be NP-complete, Jain et al. [16] and Chen and Majumdar [4] proposed several heuristics. Their heuristics are all centralized. However, due to the complex interactions within a parallel computer, it may be difficult or unpractical to gather all the information at a central point, choose the strategy and then redistribute the decision. This approach may introduce costly synchronization points and cause additional network transfers.

Our new parallel scheduling heuristic specifically targets the collective I/O operations. We assume that, at a certain point in time, n_p compute nodes simultaneously issue large data requests for n_{IOS} I/O servers. We find this to be a reasonable assumption for two reasons. First, collective I/O operations frequently involve all the compute nodes on which the application runs. Second, files are typically striped over all the available disks for performance reasons.

For writing, large requests are split by each compute node into smaller requests of size b. 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}. For instance, in figure 5, $n_p = 2$ compute nodes simultaneously issue 4 requests for $n_{IOS} = 2$ I/O servers. If both compute nodes decide to send the request to the IOS0 first, and then to IOS1, a schedule of length 3 results (for instance "Schedule 1"). On the other side our heuristic produces a schedule of length 2 ("Schedule 2"), as the I/O servers run in parallel.

For reading, the compute nodes send all the requests to the I/O servers which, in turn, split the data into blocks of size b. Then, at time step $t_j, j = 0, 1, ...$, the i-th I/O server sends a block to the compute node with the number $(i + j)$ modulo n_p.

Notice that there is no central point of decision, each process acts independently. The heuristic tries to involve all the I/O servers in the system, at a given time t. The heuristic is automatically reseted at the beginning of each collective operation.

The heuristic can be used also for other pairs of resources: compute nodes - cache managers, cache managers - I/O servers. In section 5 we will show how we use it for collective I/O.

5. COLLECTIVE I/O

Let us now assume that the processes of a parallel program (written for instance in MPI) issue parallel write or read operations by using a corresponding collective call (`MPI_File_read_all` or `MPI_File_write_all` in MPI-IO).

ROMIO's two-phase I/O of ROMIO performs two steps as illustrated in the figure 6 for the collective read case. In the access phase, the compute nodes divide the access interval into equal parts after a negotiation (1) and each one reads contiguously its share from the file system into a local collective buffer (2 and 3). In the shuffle phase (4), the compute nodes exchange the data according to the requested access pattern. The access phase is always fast, as only contiguous requests are sent to the file system. The scatter-gather operations take place at compute node, whereas the data travels twice through the network. However, as we discussed in subsection 3.2, the second network access is worth only if the parallel cost of shuffle at compute nodes is smaller. In ROMIO it is not possible to avoid a second transfer.

Figure 7 shows an example of Clusterfile's read collective operation, which consists of the following steps. Each compute node sets a view on the file as shown in subsection 3.2. The compute nodes send the requests to the cache managers (1). If the data is not available, it is retrieved from the I/O servers (2 and 3) into a collective buffer. The steps 2 and 3 are performed solely once at the arrival of the first request from the collective I/O participants at the cache managers. Subsequent requests either wait for the data to arrive or find the data already cached. Finally, the data is sent to the compute nodes(4). The global order of request service is guided by the parallel I/O scheduling heuristic from section 4. The heuristic is used between compute nodes and cache managers as well as between cache managers and I/O servers.

5.1 Parallelism considerations

In figure 3 we illustrated the parallelism potential of Clusterfile. Here we show how the collective I/O operations relate to the different hierarchy levels.

The file striping hashing (x modulo n_{IOS}) allows to maximize local cache and disk parallelism, whereas the global cache function (x modulo n_{CM}) targets the maximal memory parallelism. If $n_{IOS} = n_{CM} = n$, the hashing functions are the same (x modulo n) and the global cache parallelism directly translates into local memory and disk parallelism. If $n_{IOS} < n_{CM}$, the degree of parallelism is increased with $n_{CM} - n_{IOS}$ when the global cache is used. These types of parallelism refer to the *spatial* characteristics of files, i.e. the data placement for potentially parallel access.

The *temporal* order of servicing the requests is given by the parallel I/O scheduling heuristic from section 4 that targets to optimize the network parallelism.

In the experimental section, we will show the impact of the variation of disk, local and global memory parallelism on the performance of the collective I/O operations.

5.2 Two-phase or disk-directed?

Clusterfile's collective I/O method integrates the disk-directed and two-phase approach in a single design.

If $n_{IOS} = n_{CM}$ and the i-th cache manager runs on the same node as the i-th I/O server , the collective buffering takes place at the I/O server's node, which stores the block. In the example from figure 7, the steps 2 and 3 do not occur. The cache managers and the I/O servers share the collective buffers. The data travels over the network only once. In this case, we can say that Clusterfile's collective I/O method is *disk-directed*.

If $n_{IOS} < n_{CM}$, the collective buffers of n_{IOS} disks can be cached at n_{CM} nodes. In this case, Clusterfile's collective I/O is a *two-phase* method: the first phase is the transfer from I/O nodes to the cache managers, while the second phase is the data redistribution from cache managers to the compute nodes. In the two-phase I/O method, data travels twice over the network. However, collective accesses can benefit from a larger degree of parallelism at the cache managers.

5.3 Comparison with ROMIO's two-phase I/O

In the ROMIO's two-phase I/O the collective buffer content is dropped after the collective I/O operation is ended. A subsequent collective I/O operation accessing the same data (as for pipelining or result redistribution) will have to read again the data into the collective buffer. On the other hand, the collective buffers of Clusterfile can be reused as long as they are in the global cache (hot collective buffers).

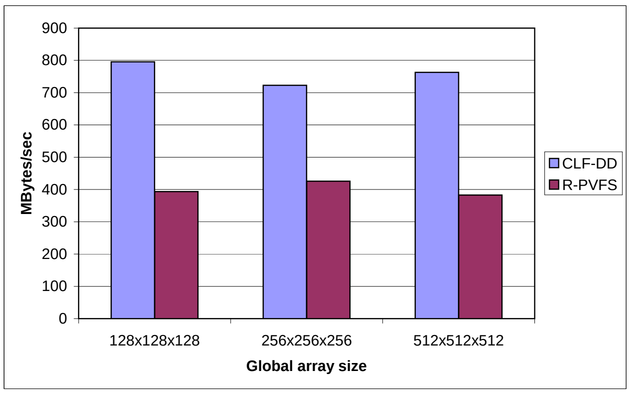


Figure 8: Local caches write aggregate throughput for ROMIO 3D benchmark

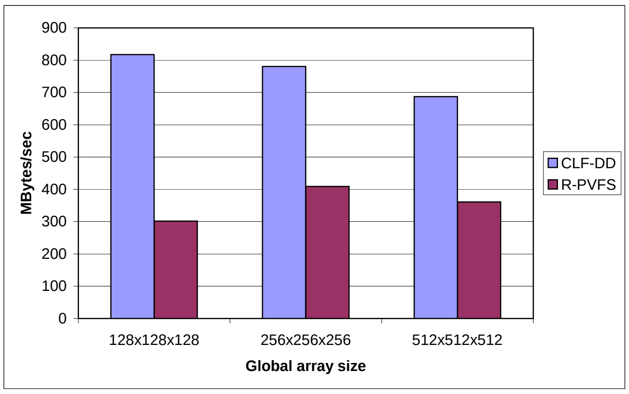


Figure 9: Local caches read aggregate throughput for ROMIO 3D benchmark

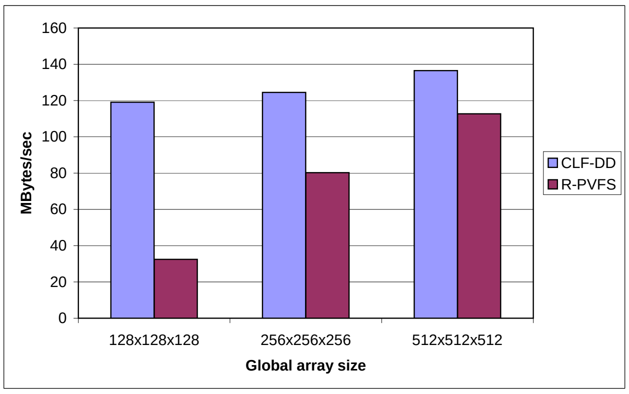


Figure 10: Disk write aggregate throughput for ROMIO 3D benchmark

Clusterfile's design offers flexible choices. The collective buffers can be managed at different places in a cluster. As seen before, the disk-directed I/O approach avoids one network transfer. For two-phase I/O, costly scatter-gather operations may be distributed over several nodes. ROMIO's two phase I/O performs two network transfers in most of the cases, because collective buffers reside always at compute nodes.

An important advantage of ROMIO is portability, as it

Class	Total				Write operations				Read operations		
	CLF (sec.)	R-PVFS (sec.)	Improv. (sec.)	Speedup	CLF (sec.)	R-PVFS (sec.)	Improv. (sec.)	Speedup	CLF (sec.)	R-PVFS (sec.)	Speedup
A 0.4 GB	58.8	63.8	5.0	1.09	7.7	12.9	5.2	1.67	0.9	6.4	7.34
B 1.6 GB	216.0	241.4	25.4	1.12	14.5	39.0	25.5	2.69	8.6	12.2	1.43
C 6.8 GB	866.6	1078.1	211.5	1.24	53.8	255.9	210.9	4.75	51.9	43.4	0.83

Table 1: BTIO benchmark results

can be used with different file systems. ROMIO's collective I/O and views are file system independent, as it can be noticed in the right hand side of figure 6. ROMIO's method can also be used together with the individual I/O operations of Clusterfile through its MPI-IO interface. On the other hand Clusterfile's collective I/O and views are implemented inside the file system (see figure 7). This allows a tight integration of file system policies with different parallelism types.

6. EXPERIMENTAL RESULTS

Parallel scientific applications frequently use regular data distributions like High Performance Fortran [18] BLOCK and CYCLIC distributions. In order to evaluate the performance of Clusterfile's collective I/O operations, we use NASA's BTIO benchmark, a benchmark from the ROMIO suite and a workload that allows variation of granularity and data access sizes. Similar experiments were performed for demonstrating the performance of Panda I/O library for fine granular distributions [5].

We performed our experiments on a cluster of 16 dual processor Pentium III 800MHz, having 256kB 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 2.4.19 with *ext2* local file system. We used TCP/IP on top of the 2.0 version of the GM [26] communication library. The ttcp benchmark delivered a TCP/IP node-to-node throughput of 120 MB/sec. The TCP/IP overhead is known to be large due to multiple copies, flow-control, reliable delivery etc. We believe these to be the reason for the 120 MBytes/s performance of ttcp for a 2Gb network interconnect. For comparison, on our cluster VIA over Myrinet achieves 220 MBytes/sec out of the theoretical 250 MBytes/s. A Clusterfile over VIA implementation is current work. The network buffer size was $b = 64K$. All measurements were repeated five times and the mean value is reported.

6.1 BTIO benchmark

NASA's BTIO benchmark [28] 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. 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. At the end, the file is collectively read and the solution is verified for correctness. In this paper we report the results for the MPI implementation of the benchmark, which uses MPI-IO's collective I/O routines. Two collective I/O implementations are compared: Clusterfile's, denoted as CLF and ROMIO over PVFS, denoted as R-PVFS. The collective buffers of Clusterfile, as well as those of ROMIO were cold. Clusterfile used 16 cache managers and 16 I/O nodes. ROMIO's two-phase I/O employed 16 compute nodes for collective buffering and 16 PVFS I/O servers. We use 16 processes and three classes of data set sizes: A (419.43 MBytes), B (1697.93 MBytes) and C (6802.44 MBytes). The benchmark reports the total time including the time spent to write the solution to the file. However, the verification phase time containing the reading of data from files is not included in the reported total time. Table 1 displays the results on four groups of columns: class (A, B or C), total timing, collective write timing and collective read timing. The improvement is computed as the difference between the measured total time for R-PVFS and CLF. The speedup is the rate between the measured total time for CLF and R-PVFS.

BTIO using Clusterfile's collective I/O runs faster with 9 %, 12 % and 24 % for the classes A, B and C respectively. The improvement comes from the collective I/O methods, as the difference between total times and the difference between collective write times for the two methods show. The total time is smaller with 5.0 sec., 25.4 sec. and 211.5 sec. for the classes A, B and C respectively, while the collective write time (included in the total time) is is smaller with 5.2 sec., 25.5 sec. and 210.9 sec. respectively.

The total times of collective write operations of Clusterfile are 67 % (A), 169 % (B) and 375 % (C) faster that those of ROMIO over PVFS. Clusterfile's collective read operations are faster with 634 %, 43 % for A and B classes respectively. For the C class they are 17 % slower. As these measurements were performed short before the deadline of the camera-ready paper, a better understanding of these particular performance results is to be sought in the future.

6.2 ROMIO three dimensional block benchmark

In this subsection we report the aggregate file read and write throughput of a collective I/O benchmark from ROMIO test suite. A three dimensional array is distributed in three dimensional blocks among compute nodes. All compute nodes simultaneously write and then read their corresponding subarrays by using a collective call. We repeated the experiment for three array sizes: 128x128x128, 256x256x256, 512x512x512. The size of each element was 16 bytes, amounting to matrix sizes of 32 MBytes, 256 MBytes and 2 GBytes, respectively.

In this test, Clusterfile used 16 compute nodes, 16 I/O nodes and 16 cache managers. The CM_i ran at the same node as IOS_i, i.e. the collective I/O method was disk-directed. ROMIO used PVFS parallel file system and em-

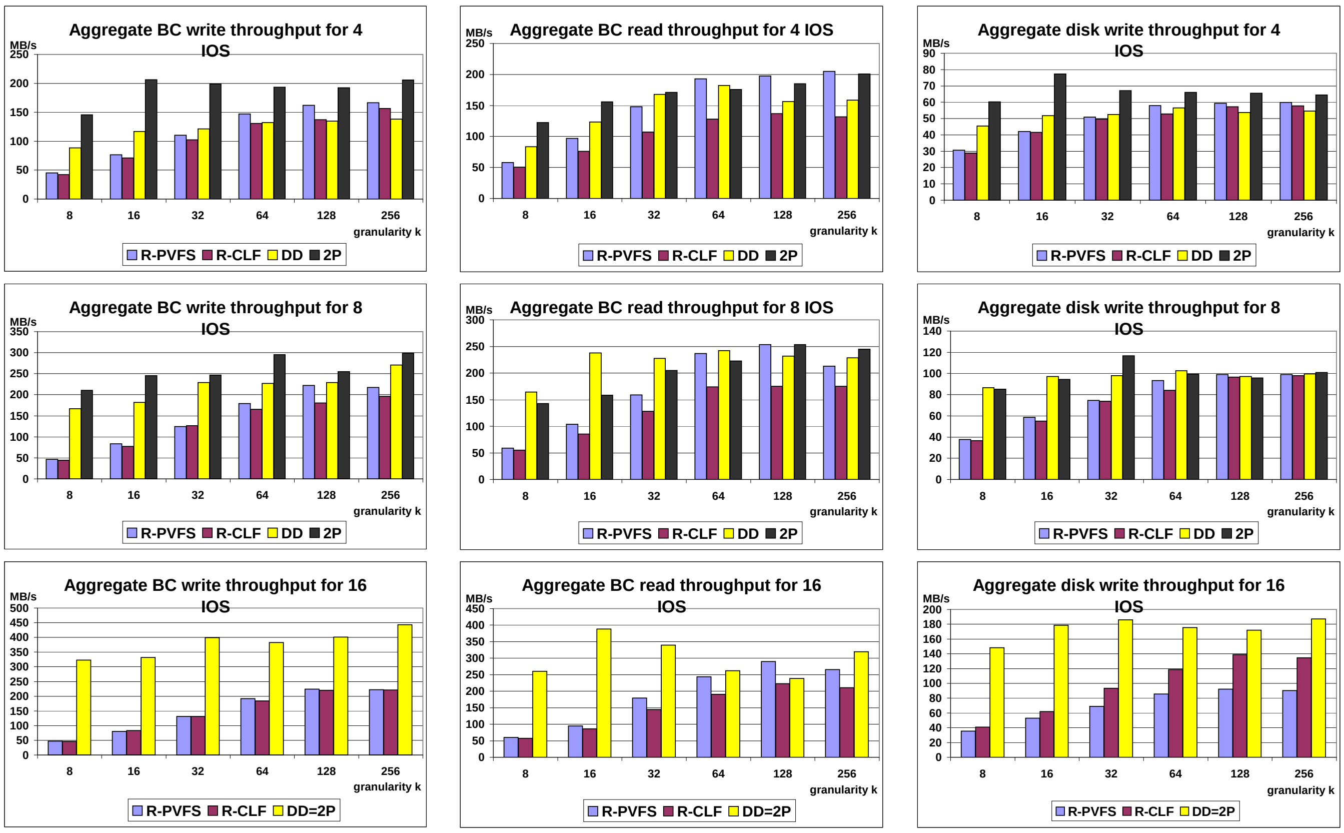


Figure 11: Aggregate throughput for different granularities.

ployed 16 compute nodes for collective buffering. The collective buffers of both Clusterfile and ROMIO were cold. PVFS files were striped over 16 I/O nodes. Figures 8, 9, 10 show the results.

We notice that Clusterfile's disk-directed method significantly outperformed ROMIO's two-phase I/O in all cases. Clusterfile performed a single network transfer, while two-phase I/O performed two. Additionally, the scheduling I/O strategy yielded a good network and disk utilization, while in ROMIO 's two phase I/O the file system access did not overlap the shuffle phase as we will show in subsection *6.3.2* and figure 15.

6.3 Two dimensional matrix synthetic benchmark

In the next experiments we compare 4 collective I/O implementations: ROMIO over PVFS [12], denoted as "R-PVFS" in the graphics, and ROMIO, disk-directed and two-phase I/O over Clusterfile, denoted as "R-CLF", "DD" and "2P" respectively. For collective buffering, our two-phase I/O used 16 cache managers while extended two-phase I/O used 16 compute nodes. For 16 I/O servers and 16 cache managers, our two-phase I/O converges to disk-directed I/O (steps 2 and 3 from figure 7 are not necessary) and therefore we report only one value.

In the next two experiments, the global cache is cold, in order to be fair in the comparison with the ROMIO implementation, which uses cold collective buffers, as explained in section 5. These experiments show the impact of the variation of the degree of local cache and disk parallelism from

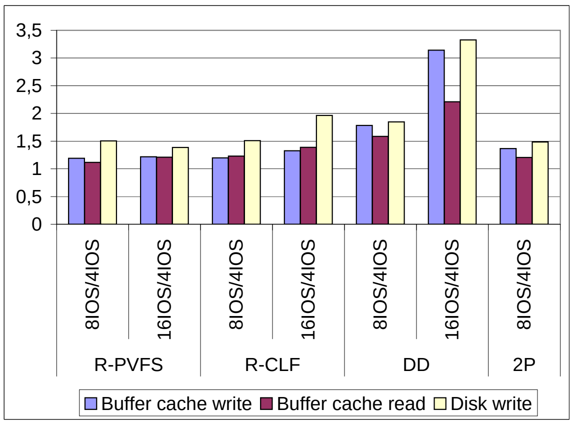


Figure 12: Average speedup for different granularities

the figure 3 on the performance for different granularities and sizes. The global cache degree of parallelism is kept constant.

Our performance metrics are aggregate throughput (for file read and write) and speedup. Computing the speedup is particular to each experiment, as explained in the following subsections. In order to make sure that the compute node accesses files simultaneously, the processes synchronized before and after the file access by using MPI barriers. The reported results include the barrier times.

6.3.1 Different granularities

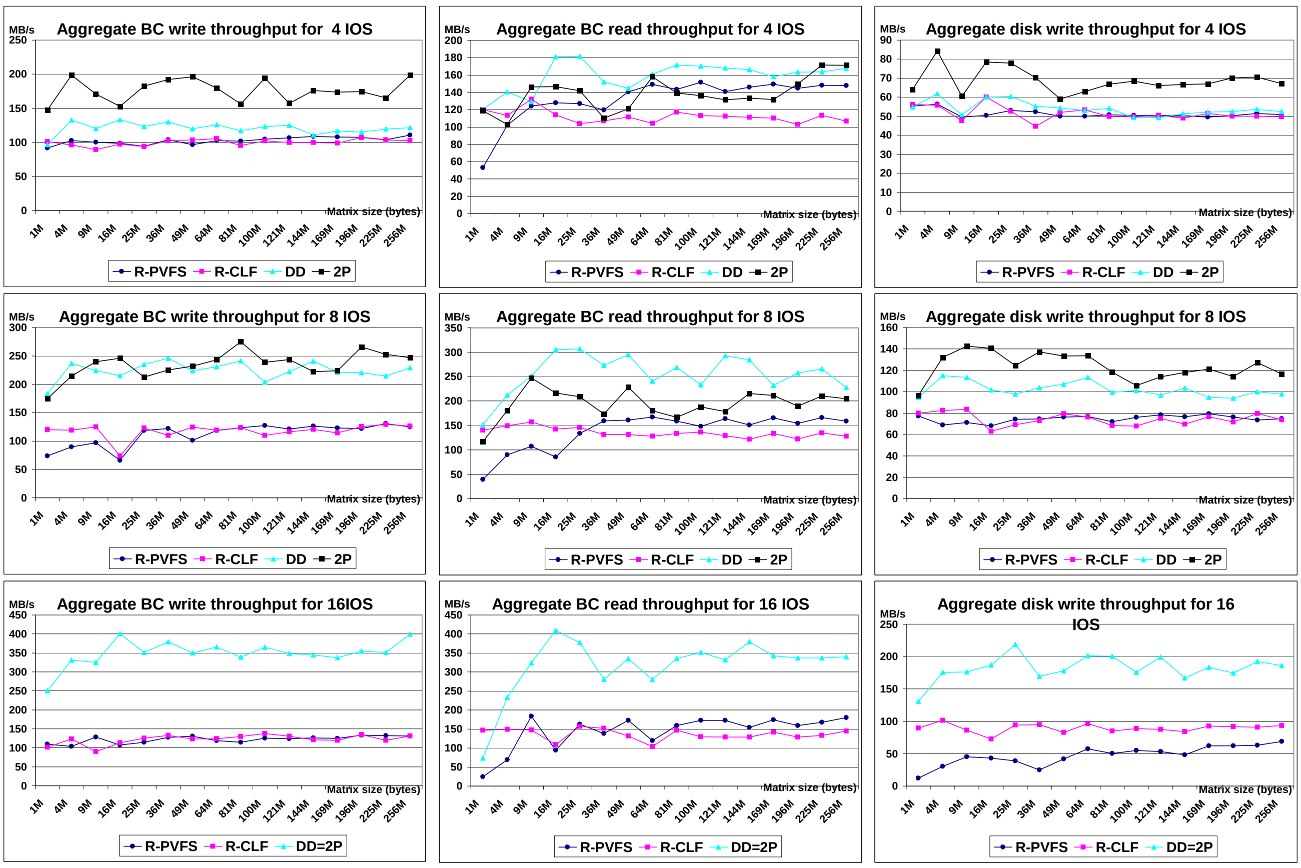


Figure 13: Aggregate throughput variation with size.

The goal of this experiment is to investigate the influence of access granularity on the performance of collective I/O. We wrote a parallel MPI benchmark that reads from and writes to a file a two-dimensional matrix. In each run, p compute nodes, arranged in a $\sqrt{p} \times \sqrt{p}$ grid declare a view on the file by using $CYCLIC(k), CYCLIC(k)$ distributions , for $k = 8, 16, 32, 64, 128, 256$. In figure 11, there are three rows of graphs for 4, 8 and 16 I/O servers. In the first column, the I/O servers write to the local buffer caches(BC), in the second they read from their buffer caches and in the third they write the data to the disks. The size of the matrix was fixed to 256 MB. We define the speedup for a given granularity as the relative aggregate throughput gain, when increasing the number of I/O servers from x to y.

$$S(x, y) = \frac{AggrThroughput_{yIOS}}{AggrThroughput_{xIOS}}$$

The speedups plotted in figure 12 are means of speedups for all granularities from figure 11. The speedup can be interpreted as the performance gain, when the degrees of parallelism of disks and local caches are increased.

Figure 11 shows that,in terms of aggregate throughput, our two-phase I/O method performs better than the others in most of the cases. Our two phase I/O uses all the cache managers in order to compute the access indices and to perform the scatter-gather operations. Compute nodes do not communicate among each other and there is no explicit synchronization point. In this case, the scalability depends on the I/O servers. In turn, ROMIO makes similar operations at the compute nodes that synchronize for exchanging meta-data information. Our measurements confirm that, when varying the number of I/O nodes, the scalable execution part is the communication with the I/O servers. First of all, in figure 12, disk-directed scales up when using 16 instead of 4 I/O servers with disks by a factor of 3.1 for buffer cache writing, 2.2 for buffer cache reading and 3.3 for disk writing, while ROMIO achieves 1.3, 1.3 and 1.9 respectively, for the same operations.

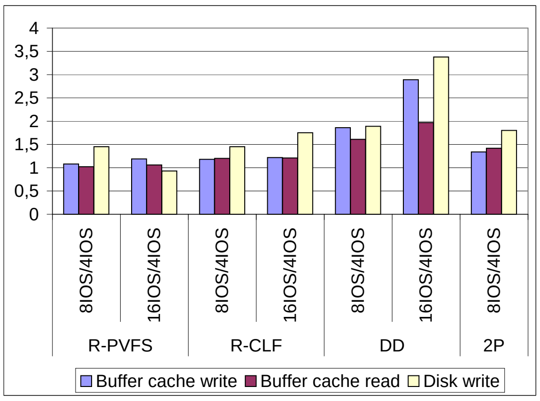
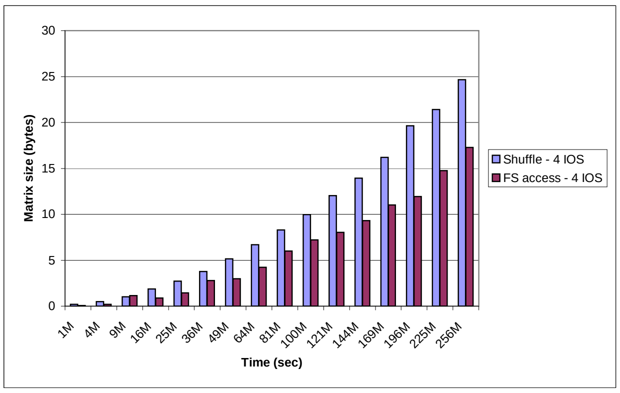


Figure 14: Average speedup for different matrix sizes



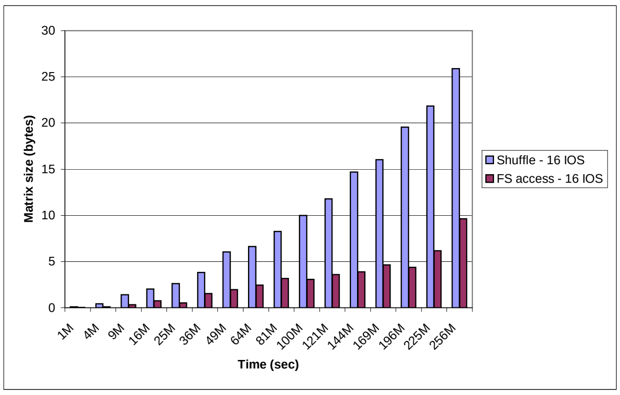
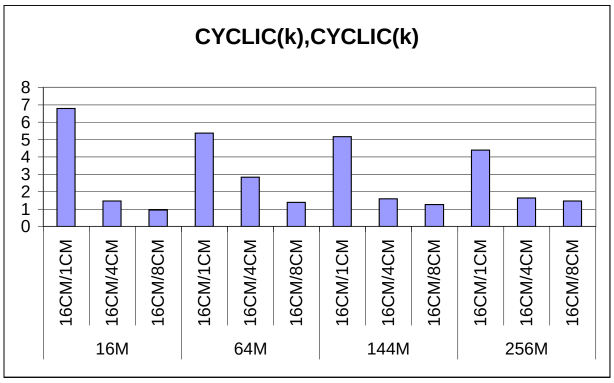


Figure 15: ROMIO's collective read breakdown for 4 and 16 I/O servers



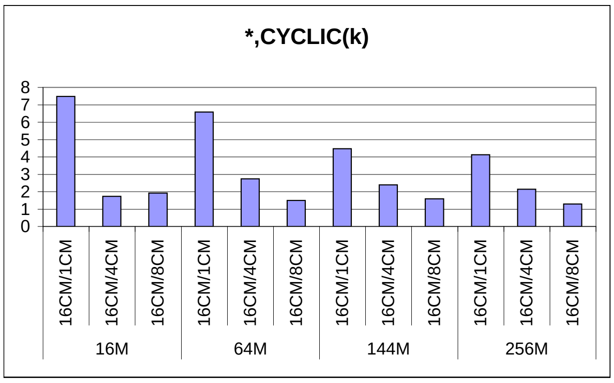


Figure 16: Cooperative caching speedup

6.3.2 *Different sizes*

Further, we assessed the influence of varying the data size over the aggregate throughput. We use the same $CYCLIC(k), CYCLIC(k)$ distribution from the previous subsection for $k = 32$. The matrix size is varied from 1Kx1K bytes (1 MB) to 16Kx16K bytes (256 MB). The X-axis of the graphs in figure 13 represents the matrix size in MBytes.

Figure 13 shows the results for 4, 8 and 16 I/O servers. The speedup is defined like in previous subsection. The speedups plotted in figure 14 are speedup means for all the sizes reported in figure 13.

Notice that our implementation outperforms the other ones except for one case, namely when reading a 1MB matrix from the buffer cache. Again, our two-phase I/O implementation scales better with the number of I/O servers than extended two-phase I/O for ROMIO.

Figure 15 gives more insight about ROMIO two-phase I/O implementation over Clusterfile, by showing the breakdowns of the total time spent in the shuffle and file system access phases of collective read operations, for both 4 and 16 I/O servers, i.e. for the first and last row of graphs from figure 13. As expected, because the I/O servers are not involved, the shuffle time does not change significantly when increasing the number of I/O servers from 4 to 16. The increase in bandwidth is obtained from file system access. However, for small granularities, the shuffle phase is computationally intensive due to index computation and memory copy operations and therefore limits the scalability.

6.3.3 *Global cache scalability*

In this subsection we were interested in evaluating the collective I/O read performance speedup, when the size of the global cache is increased. Two data distributions are used: $CYCLIC(k), CYCLIC(k)$ and $*, CYCLIC(k)$ for k=128. For each distribution we report results for our two-phase I/O and 4 matrix sizes: 16, 64, 144 and 256 MB. We define the speedup for a given size as the relative aggregate throughput gain, when increasing the number of cache managers from x to y.

$$S(x, y) = \frac{AggrThroughput_{yCM}}{AggrThroughput_{xCM}}$$

All accesses were performed from a warm global cache. This experiment shows how the variation the degree of global cache parallelism from the figure 3 impacts the performance. The local cache and disk parallelism are not involved, as the I/O servers are not contacted.

When increasing the number of CM from 1 to 16 the speedup is upto 6.8 for $CYCLIC(k), CYCLIC(k)$ and upto 7.3 for $*, CYCLIC(k)$. The speedup is achieved from both the parallel access to several cache managers and from the distribution of scatter/gather costs over several computers. This represents a significant performance improvement. However, further assessments are necessary in order to better understand the difference between the potential speedup of 16 and the measured speedup. One reason is that the reported results include two global barriers in order to assure true parallel accesses. Therefore, they include the idle times of processes that arrive early at the barriers. Another reason is that the parallel scheduling I/O strategy assumes uniform service time, which is hardly achievable in practice.

7. SUMMARY

In this paper we integrate two collective I/O techniques into a common design. The approach allows combining the advantages of disk-directed I/O (e.g. one transfer over the network, reduced copy operations) with those of two-phase I/O (distribution of I/O related computation over all compute nodes). Additionally, to the best of our knowledge, we present the first implementation that integrates collective I/O and cooperative caching. The performance results without cooperative caching show substantial improvements over ROMIO two-phase I/O. Cooperative caching further speeds up the file access when the collective I/O buffers are reused.

8. REFERENCES

[1] T. Anderson, M. Dahlin, J. M. Neefe, D. Patterson, D. Rosseli, and R. Y. Wang. Serverless Network File Systems. In *The 15th Symposium on Operating System Principles*, Dec. 1995.

[2] R. Bagrodia, S. Docy, and A. Kahn. Parallel simulation of parallel file systems and i/o programs. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–17. ACM Press, 1997.

[3] R. Bordawekar. Implementation of Collective I/O in the Intel Paragon Parallel File System: Initial Experiences. In *Proc. 11th International Conference on Supercomputing*, July 1997. To appear.

[4] F. Chen and S. Majumdar. Performance of parallel I/O scheduling strategies on a network of workstations. In *Proceedings of ICPADS 2001*, pages 157–164, April 2001.

[5] Y. Cho, M. Winslett, Y. Chen, and S. wen Kuo. Parallel I/O performance of fine grained data distributions. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, 1998.

[6] P. Corbett and D. Feitelson. The Vesta Parallel File System. *ACM Transactions on Computer Systems*, 1996.

[7] T. Cortes, S. Girona, and L. Labarta. PACA: A Distributed File System Cache for Parallel Machines. Performance under Unix-like workload. Technical Report UPC-DAC-RR-95/20 or UPC-CEPBA-RR-95/13, Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya, 1995.

[8] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *The First Symp. on Operating Systems Design and Implementation*, Nov. 1994.

[9] E. DeBenedictis and J. D. Rosario. nCUBE Parallel I/O Software. In *Proceedings of 11th International Phoenix Conference on Computers and Communication*, 1992.

[10] J. del Rosario, R. Bordawekar, and A. Choudhary. 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.

[11] J. Huber, C. Elford, D. Reed, A. Chien, and D. Blumenthal. Ppfs: A high performance portable file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, 1995.

[12] W. L. III and R. Ross. An Overview of the Parallel Virtual File System. In *Proceedings of the Extreme Linux Workshop*, June 1999.

[13] Intel Corporation. *Paragon System User's Guide*, April 1996.

[14] F. Isaila and W. Tichy. Clusterfile: A flexible physical layout parallel file system. In *First IEEE International Conference on Cluster Computing*, Oct. 2001.

[15] F. Isaila and W. Tichy. View I/O: Improving the performance of non-contiguous I/O. In *Third IEEE International Conference on Cluster Computing*, Dec. 2003.

[16] R. Jain, K. Somalwar, J. Werth, and J. C. Browne. Heuristics for scheduling I/O operations. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):310–320, March 1997.

[17] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation*, 1994.

[18] D. B. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1993.

[19] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1995.

[20] Message Passing Interface Forum. *MPI2: Extensions to the Message Passing Interface*, 1997.

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

[22] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of FAST*, 2002.

[23] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95.*

[24] H. Simitici and D. Reed. A Comparison of Logical and Physical Parallel I/O Patterns. In *International Journal of High Performance Computing Applications, special issue (I/O in Parallel Applications), 12(3)*, 1998.

[25] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, February 1999.

[26] M. I. G. the low-level message-passing system for Myrinet networks. *http://www.myri.com/scs/index.html.*

[27] G. M. Voelker, E. J. Anderson, T. Kimbrel, M. J. Feeley, J. S. Chase, A. R. Karlin, and H. M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 33–43. ACM Press, 1998.

[28] P. Wong and R. der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, NASA Ames Research Center, Moffet Field, CA, Jan. 2003.