The impact of applications' I/O strategies on the performance of the Lustre parallel file system

Francieli Zanon Boito*, Rodrigo Virote Kassick and Philippe O.A. Navaux

Institute of Informatics,
Federal University of Rio Grande do Sul,
Porto Alegre, Brazil
E-mail: fzboito@inf.ufrgs.br
E-mail: rvkassick@inf.ufrgs.br
E-mail: navaux@inf.ufrgs.br
*Corresponding author

Abstract: Parallel applications present multiple approaches regarding the management of data. Due to specific characteristics of parallel file systems, some approaches will provide better performance than others due to a better matching to the system's internals.

One common situation is when each instance of an application accesses exclusive data stored in the file system. This paper studies some I/O techniques for this situation and evaluates them on the Lustre file system. We provide a guide to help developers tune their application to extract the best performance out of Lustre.

Our results show expressive gains in performance related with the choice of access pattern of the application. We present considerations on operation granularity, intra-node concurrency and temporal behaviour of the application.

Keywords: parallel file system; PFS; Lustre; parallel I/O; access patterns; HPC; cluster; I/O granularity; scalability; I/O strategies; data organisation.

Reference to this paper should be made as follows: Boito, F.Z., Kassick, R.V. and Navaux, P.O.A. (2011) 'The impact of applications' I/O strategies on the performance of the Lustre parallel file system', *Int. J. High Performance Systems Architecture*, Vol. 3, Nos. 2/3, pp.122–136.

Biographical notes: Francieli Zanon Boito is a PHD student. She received her Bachelors in Computer Science from the Federal University of Rio Grande do Sul and has worked with the dNFSp distributed file system from 2005 to 2008. Since 2008, she has been studying Lustre's performance.

Rodrigo Virote Kassick is a PHD student. He received his Bachelor and Master in Computer Science from the Federal University of Rio Grande do Sul and has worked with the dNSFp distributed file system since 2005.

Philippe O.A. Navaux is a Full Professor at the Institute of Informatics of the Federal University of Rio Grande do Sul (UFRGS), Brazil. His research interests include high performance computing and computer architectures. He received his BEE in Electronic Engineering, in 1970 from the UFRGS, MSc in Applied Physics, in 1973 from the UFRGS, and PhD in Computer Science, in 1979 from the INPG/Grenoble, France.

1 Introduction

In 1965, Gordon Moore stated that the computing power of processors would double approximately every two years. This idea became known as 'Moore's Law' and has been confirmed during several years (Schaller, 1997). However, as the growth in processors' speed approached physical limits, the solution to offer more processing capacity was the shift to parallel and distributed architectures, like clusters of computers.

While the processing speed have increased (either by more transistors or by more cores), the improvements in I/O components did not follow this rate. This happens because their performance is limited by mechanical devices like magnetic disks and the bandwidth of their interconnection (Patterson and Hennessy, 2004). Since the amount of data grows with the applications' complexity, I/O operations have become a bottleneck, impairing the system's performance.

A usual solution that aims at hiding this gap between processing and I/O speeds is the use of *parallel file systems* (PFS). In these systems, data is distributed among several

machines in order to allow parallel accesses, offering a better performance.

Although each application has its peculiarities, they usually share several characteristics on their I/O strategies. These common behaviours are called 'access patterns' in this work. In this scenario, it is important to study the performance of a file system under several well-known access patterns in order to allow a good matching between application and PFS and achieve good I/O performance.

There are several PFS being studied in both commercial and academic fields, each of them with a unique set of project choices. These choices influence directly the performance that an application can obtain from the PFS: the design of the system will favour I/O strategies used by some applications, but can cause unexpected overheads to applications with different approaches. Since it is not always possible to adapt the system design to provide good performance in every possible scenario, some kinds of applications will be favoured at the price of sub-optimal performance for others.

This article studies the performance of some common I/O strategies for parallel applications in order to identify the best practices on a PFS. We evaluated a situation that is very common in scientific applications: exclusive access to data by the clients. In this case, each instance of a parallel application writes or reads its own data in some instant. This data may be stored in segments of a shared file or in individual files. When a shared file is used, there are several ways of distributing it among the processes. Each process can operate over a contiguous portion of the file or several sparse portions of it.

We chose to evaluate these strategies on the Lustre file system (CFS, 2002; SUN, 2008; Wang et al., 2009). In order to define Lustre's behaviour under such conditions, we executed tests that mimic the described access patterns under several different parameters. We present a performance study from the client's point of view and provide the characterisation of the patterns favoured by Lustre's design and how better fit applications to obtain good I/O performance when using this system.

The remainder of the paper is organised as follows: Section 2 presents the state of the art and related works. The Lustre file system is described in Section 3. Section 4 introduces the I/O strategies that are evaluated in this work and presents the tests that represent them. The results obtained with the proposed tests are presented in Section 5. Finally, Section 6 discusses our conclusions and future work directions.

2 Related work

This section presents some of the state-of-art in parallel file systems, their performance evaluation and the matching of these evaluations for application I/O strategies. Subsection 2.3 presents a study on Lustre and positions our work in the area.

2.1 Parallel file systems

Quoting Coulouris et al. (2005), "a distributed file system enables programmes to store and access remote files exactly as they do local ones, allowing users to access files from any computer in an intranet". The first distributed file systems – DFS – have been created aiming at sharing storage devices that were an expensive resource then. These first systems, like *sun network file system* – NFS – (Sun Microsystems, 1989), developed in the '80s are usually composed by a centralised server responsible for all the clients' requests.

Quoting Coulouris et al. (2005) again, "a well-designed file service provides access to files stored at a server with performance and reliability similar to (and in some cases better) files stored on local disks". As the number of clients and the amount of data grew, centralised file systems like NFS no longer met these requirements. This happens because the centralised server became a bottleneck during data transfers (Martin and Culler, 1999). As storage devices have ceased to be such an expensive resource, the next step was the distribution of the server functionality among several machines. This idea was introduced by *Vesta file system* (Corbett and Feitelson, 1996), developed by IBM in the first half of the '90s.

Current file systems have several data servers, from which data can be accessed in parallel by the applications. Because of that, they are usually called *PFS*. Some examples of PFS include expand (Carballeira et al., 2003), GPFS (Schmuck and Haskin, 2002), Lustre (CFS, 2002), Panasas (Welch et al., 2008), PVFS (Latham et al., 2004), dNFSp (Ávila et al., 2004) and GFS (Ghemawat et al., 2003).

In these systems, data is usually stored separated from its metadata – information like size, permissions and location. In order to access the data, clients can obtain this information from a *metadata server* (MDS) and then send their requests directly to where the data is stored. This solution is applied in Lustre and PVFS, i.e. Another option is to have the MDSs to intermediate access from clients to the data servers. This way, requests are sent from clients to MDSs and from these to the corresponding data servers. The data servers may then answer directly to the clients or use the MDSs again as intermediate. While the first solution (direct access) has a lower overhead with communication, the second (indirect access), applied in systems like dNFSp and pNFS (Hildebrand and Honeyman, 2007), allows more dynamism in data distribution and the use of legacy protocols like NFS to communicate with the clients.

Because metadata storage is involved in all operations of the file system, the scalability of the accesses to it impacts the scalability of the whole system, mainly when the indirect access to data is used. Some systems allow the distribution of the metadata too, like dNFSp and PVFS. However, the distribution brings some complex issues about consistency among the meta-servers – like having clients to contact all the MDSs in order to locate information on files (Brandt et al., 2003).

The operation that distributes a file among several data servers is called *striping*. It can be made through some static rule, like in dNFSp, or be configurable by the users, like in Lustre and Expand. Therefore, the number of data servers impacts directly the performance of the accesses.

2.2 PFS and application behaviour

We listed some of the several project choices that can be made by PFS developers. Different choices result in systems with different behaviours that have better performance for some access patterns than for others. This fact has two-way implications:

- The application, with knowledge over the behaviour of the file system to be used, chooses to do its I/O operations in a way that will result in the best performance. In other words, it takes advantage of the system's optimisations. Another way to do that would be, instead of adapting the application's behaviour to a specific system, choosing a file system that will provide better performance given the way that the application does I/O.
- The PFS developers, knowing the access patterns that are favoured by the target applications, make project choices that will lead the system to have better performance under those access patterns.

This perfect matching between file system and applications is, however, very hard to achieve. This situation would require the application to know the behaviour of the file system and the system developers to know the access patterns from the applications. In this case, the developers should know what project choices lead to what behaviours, and this is not always clear.

Since the '90s, several works explored access patterns characterisation for parallel scientific applications and evaluated their performance on several systems (Smirni and Reed, 1998; Kotz and Ellis, 1993; Wang et al., 2004; Zhang et al., 2004; Miller and Katz, 1991; Smirni and Reed, 1997).

Kotz and Ellis (1993), aiming to evaluate the performance of PFS cache approaches, presented an organisation of file access patterns in classes. This classification served as inspiration for the one used in this work – see Section 4 for details.

A common conclusion among the works that evaluate access patterns' performance on different systems is that fine grained requests to sparse portions of files causes low I/O performance. This access pattern is, though, very common in scientific applications (Fryxell et al., 2000).

Studies pointed several techniques to solve this problem. One of these is the use of *data sieving* (Thakur et al., 1998). With this technique, instead of making small sparse operations, data is requested from the server in large and contiguous portions of the file that cover the small portions needed by all processes. If the amount of unused requested data is not too big compared with that of useful data, the bigger throughput compensates for the greater amount of transferred data.

Another largely applied technique is *collective* I/O. It consists in merging the portions needed by the clients in order to create larger and continuous requests. The collective operations can be implemented server-side (Seamons et al., 1995) or client-side. The implementations in the client-side are, however, more usually applied, as they promote the portability between different architectures and configurations. The MPI-IO API (Thakur et al., 1999; MPI-IO Committee, 1996) uses a two-phase approach, with a phase of requests to the servers and another of data exchange between the clients (Seamons et al., 1995).

The performance of collective operations usually depends on the architecture (file system, middleware, network interconnection, etc.) and specific optimisation routines can be applied to match the infrastructure. Additionally, the implementation of this method requires expensive synchronisation operations between clients, frequently imposing an overhead that overshadows the gains in throughput.

Zhang et al. (2009) show that performance can be improved if not only the requests are made to contiguous portions of data, but also arrive in the server ordered by their offset – a phenomenon called *resonance*. This same work presented a technique called *resonant I/O* that aims at inducing this ordered request dispatch through a client-side implementation of MPI-IO collective operations. While the technique worked on the client side, the metadata service was used to obtain information on the data distribution and thus guide the request ordering.

2.3 Evaluating Lustre file system's performance

Wang et al. (2004) have utilised traces to characterise scientific applications I/O operations. Through experiments with Lustre, they stated that using exclusive files to the nodes has performance up to four times better than using a shared file.

The situations tested by Wang et al. – multiple files versus shared file – are similar to the ones in this work. However, they did not use MPI-IO, explored less configurations of nodes accesses' distribution and the number of servers receiving portions of each file was substantially larger. Additionally, our results are different from the ones obtained by them.

Several recent works study Lustre's performance, especially when used in conjunction with MPI-IO. Liao et al. (2007) show the performance degradation when I/O operations are not aligned with stripe size if independent MPI-IO operations are used along with Lustre.

Larkin and Fahey (2007) examine the behaviour of Lustre on Cray XT3/XT4. Based on this analysis, they established some desirable characteristics to file system's applications on the platform. Among them is the number of processes that do I/O operations being greater or equal than the number of servers, but not too large. Their work studied only contiguous accesses, unlike ours.

Dickens and Logan (2008) have shown that, unlike the common intuition, accesses to large contiguous areas in the files may not be the pattern with the best performance on

MPI-IO with Lustre. This happens because a single segment can be distributed among all the data servers, imposing a larger communication overhead to the clients. The same authors propose a library that reorganises the accesses in order to minimise the number of servers contacted by each client, obtaining better performance (Dickens and Logan, 2009). They studied only the number of contacted servers, while our work focus on the distribution of the clients' accesses in the files.

A *network request scheduler* – NRS – for Lustre is presented by Qian et al. (2009), using an algorithm similar to the one described by Lebre et al. (2006). The scheduler receives the requests to a data server and orders them by object and offset in order to allow lower level optimisations. The gains in performance are up to 40%.

Piernas et al. (2007) evaluate active storage strategies for Lustre. The idea behind these techniques is to use the processing capacity present in data servers to perform some common routines with data that are stored on them. They show good performance gains with both kernel and user space implementations. However, there are still some issues regarding striped data.

In the work of Zhao et al. (2010), they propose a model to predict the performance of Lustre with different configurations, like number of servers and interconnection velocity. Nonetheless, they obtained estimating errors of about 20% and did not consider the access pattern of the applications in their experiments.

Yu et al. (2008) aimed at evaluating Lustre's performance under different access patterns. They conducted several tests with Lustre in the Jaguar cluster at the Oak Ridge National Laboratory. However, their analysis was strongly coupled to the hierarchical organisation of their architecture, studying parameters like distribution of the file among racks.

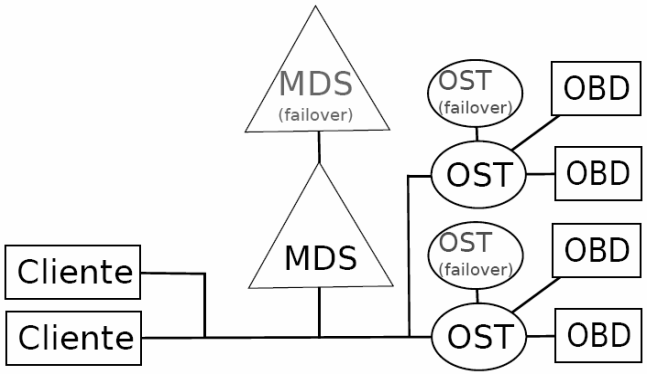
In this work, we aim at identifying the access patterns that better fit Lustre's architecture when a parallel application reads or writes data that are not shared with other instances during a same I/O phase. Our goal is not to investigate the causes of Lustre performance under each of the proposed scenarios in order to propose changes in the system, but to determine the most suitable patterns in different situations, allowing applications to adapt to Lustre and improve I/O performance.

3 Lustre file system

Lustre (CFS, 2002; SUN, 2008; Wang et al., 2009) is a parallel and DFS developed aiming to provide high performance and scalability when serving clusters of tens of thousands of nodes. Its architecture, illustrated on Figure 1, is composed by a centralised MDS and object-based data servers (OSTs, *object storage targets*).

The meta-server supports all the operations on the file system's namespace, like lookups and creations. Data servers are responsible for operations on the data itself and for the interaction with the storage devices, the *object-based disks* (OBDs).

Figure 1 Lustre file system's architecture



OSTs are also responsible for managing locks on the data that they store in order to maintain coherency under concurrent accesses. The way the files are separated in objects and their distribution among the servers are configurable by the user via command line, or during the file creation when using APIs like MPI-IO.

New OSTs or OBDs can be added to the system without interrupting it. However, only data that were written after their addition will be stored in them, i.e., the system does not reorganise the files after inclusions of servers.

Despite being named OBDs, the storage devices do not have to be disks. Their interface with the associated OST is done through a device driver that hides the identity of the OBD in use, enabling the usage of different storage technologies like, i.e., multi-channel external disks and journaling file systems for Linux (ext3, ReiserFS, etc.).

Fail-over replicas can be used by each server to enable fault-tolerance without interruption of the service. The redundant meta-server replaces the original without losses, as they both keep a synchronised transactional record of all the changes on metadata. OST servers and their fail-over instances, on the other hand, must be connected to the same OBD to provide continued access to the data in case of failure. A service named MGS is responsible for providing information about replacement servers, in addition to helping in the configuration and maintenance of the PFS.

When a client can not reach a server, it notifies the problem to the MGS that will provide a copy to take the place of the outage server. If there is no such available copy, the server will be kept out of the future writes, and error messages will only occur in read operations.

OSTs do not have cache, i.e., they write data synchronously. However, cache can be present in the employed OBD. The client has cache (writeback for data and write-through for metadata) and performs *read-ahead*, which means it requests more data from the servers than the application demanded in the last operation.

The clients prefetching algorithm tries to detect strided access patterns. However, it may fail with large stride sizes. When the read-ahead mechanism detects a pattern, it starts the prefetching with a 1 MB window. This window is increased at each request, until one that does not fit the pattern is received or a configurable read-ahead limit is reached.

The writeback data cache keeps modified blocks in a dirty cache. This cache is cleaned (and the data flushed to the servers):

- Every time that there is enough data to fill an entire RPC (1 MB) and the limit of simultaneous RPC is not reached. This limit is configurable.
- When a lock revocation occurs.
- When the dirty cache limit is reached. This limit (per OST) is also configurable.
- When it is necessary to free memory space. In this case, partial RPC may be written to the servers.

The system offers its interface to Linux clients through the standard VFS layer, enabling the usage of standard APIs. Lustre's communication is done through the *Lustre networking* (LNET) that supports several network infrastructures through pluggable drivers, named *Lustre network drivers* (LNDs). This makes Lustre adaptable to different technologies and architectures.

We chose Lustre to our work because it is widely used on production systems: it is the file system of 3 of the top-5 supercomputers (Top500, 2010). Being object of a lot of research, it was not based on a previous system. Therefore, Lustre does not inherit deficiencies of aged systems that were not built aiming at high performance.

4 Application I/O strategies

A common situation in scientific applications is when all nodes involved in some computation need to read or write exclusive data – i.e., segments of data that are utilised only by one instance of the application. For example, this data can be the result of previously executed calculations – in case of some iterative application. This is also usual in the generation of *checkpoints*, where each node will write its state in a given moment so that the execution can be continued in the future or to analyse the results at a given moment of the execution.

When using a DFS shared by all the clients, this can be achieved in several ways. For each strategy used by an application, we define a *class* that represents the corresponding *access pattern*. This section presents such classes and the situations that they represent.

4.1 SFSA – single file, segmented access

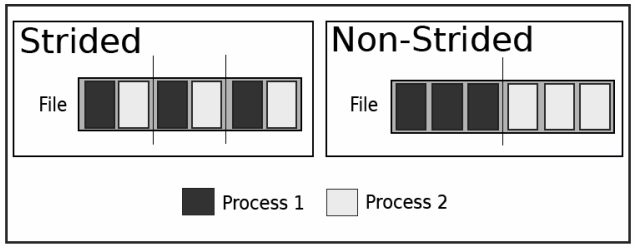
In order to allow each process to access its exclusive data, the application can use a shared file and assign segments of it to each process. In this case, the developers must distribute data in the file and make clients read only from the offsets where the data they need resides. However, there are APIs like MPI-IO that facilitate this task (MPI-IO Committee, 1996).

When using a single shared file, the distribution of data on the file can follow different strategies. Data used by a single instance can be on contiguous regions off the file or mixed with data from other processes. For example, an application where each parallel instance operates over non-consecutive lines of a matrix can store its data on the traditional row-major order. In this case, accesses to the file system will occur in sparse requests. Alternatively, the file can be organised in such manner that all the lines operated by a same instance are stored in a same segment of the file, allowing for larger requests.

mpiBLAST (Darling et al., 2003) is an example of the SFSA strategy. mpiBLAST searches for genomic sequences over a shared database. Each instance of the application is responsible for searching over a range of the shared database. Flash (Fryxell et al., 2000), an application for astrophysical simulations, also uses this approach. Each instance of this application calculates a series of variables that are later written on a shared file. These writes are done in a rather sparse manner, since each segment of the file stores all the values of a single variable.

Considering these variations in the approach, we divide the SFSA class in two: *SFSA strided* and *SFSA non-strided*. Figure 2 illustrates these two alternatives. In *non-strided*, each client operates on a contiguous area. In *strided* mode, all the nodes access a portion of the file, and then seek to the next portion.

Figure 2 Distribution of a file among two processes that access three segments of size n



4.2 MFWA – multiple file, whole access

Another way of accessing exclusive data is to store it in a file per process. In this paper, we call this approach *MFWA – multiple file, whole access*, because there are multiple files in the file system, and they are entirely accessed for reads and writes. While there is no need to manage offsets that are different to each process, using multiple files may be a burden to the file system, given that it will need to manage a multitude of file handles (metadata operations). As a result, I/O performance may suffer.

The MFWA class is represented by applications such as ESCAT (Winstead et al., 1995), MESSKIT and NWChem (Smirni and Reed, 1998). In these applications, results from the execution are stored in exclusive files for each instance. These files are accessed in the next iterations of the execution.

4.3 SMP

With the growth in the utilisation of multi-core architectures, it is usual to have more than one process in

each node. In this scenario, the instances will compete not only for the file system, but for the resources of the local host like network access and disk.

4.4 TMP

Although some applications write/read data only at the beginning or at the end of their executions, is also very common to find applications that make accesses to the file system during their entire lifetime. They can mix I/O and processing phases, reading data and them consuming it or writing preliminary results at a fixed time rate. We call this approach a temporal access pattern.

Figure 3 Classes of tests that simulate access patterns, (a) SFSA non-strided (b) SFSA strided (c) MFWA (d) SFSA-SMP non-strided (e) SFSA-TMP non-strided (f) SFSA-SMP strided (g) SFSA-TMP strided (h) MFWA-SMP (i) MFWA-TMP (see online version for colours)

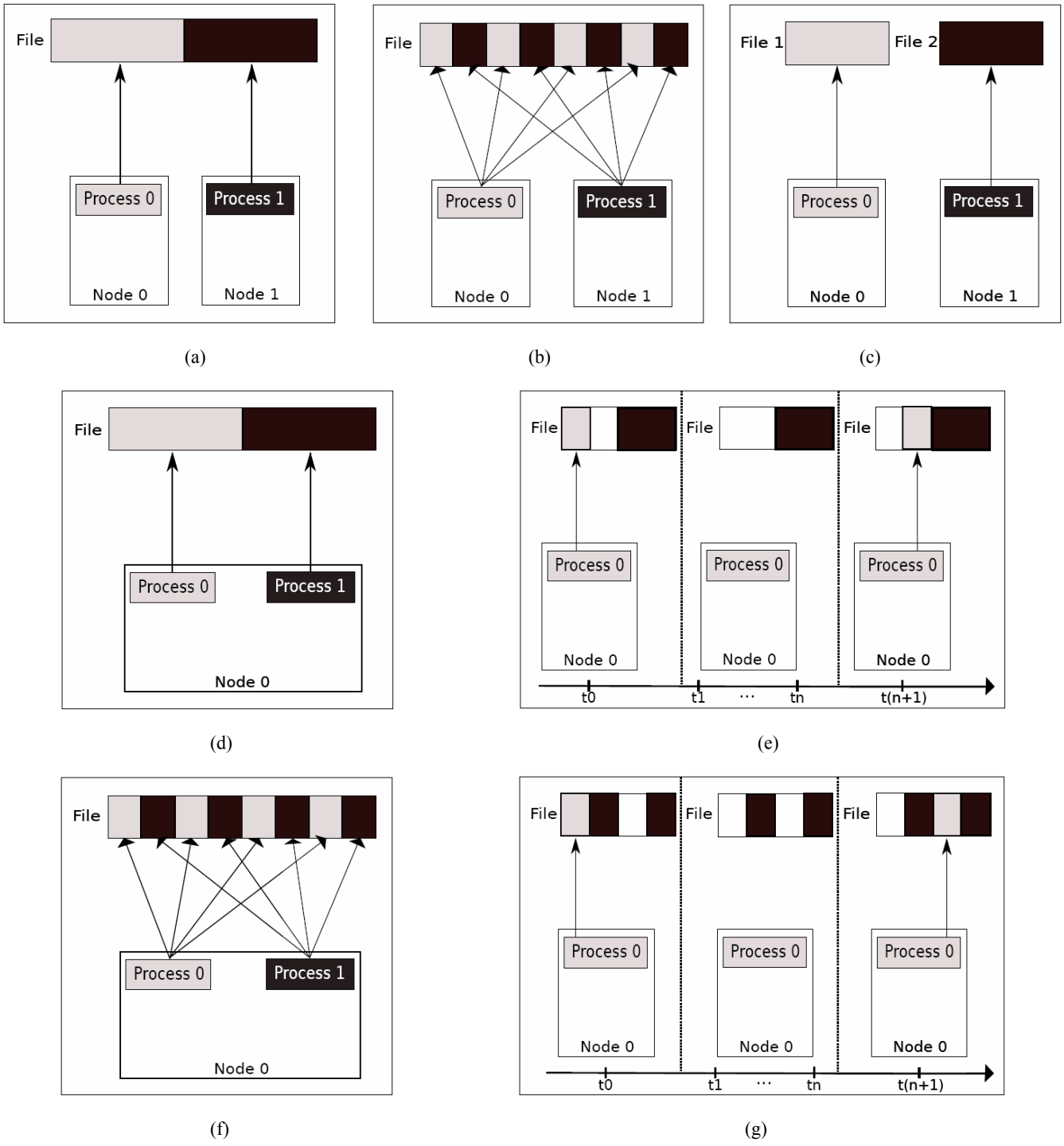
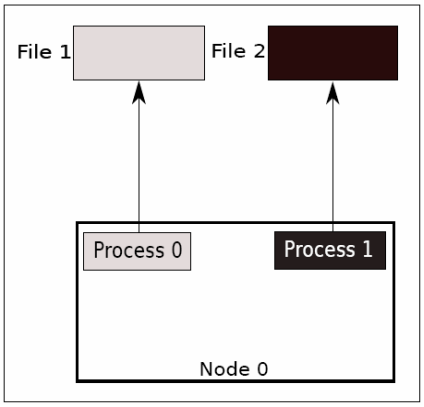
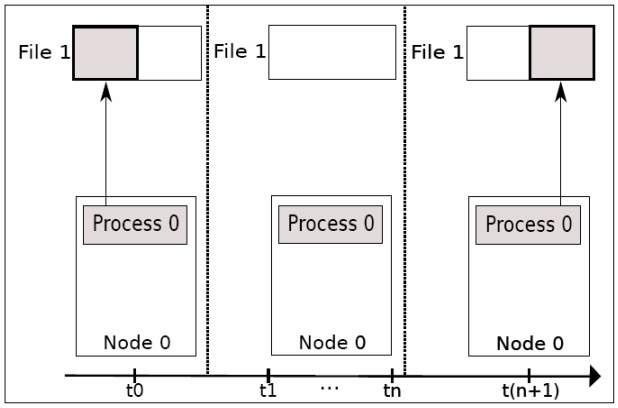


Figure 3 Classes of tests that simulate access patterns, (a) SFSA non-strided (b) SFSA strided (c) MFWA (d) SFSA-SMP non-strided (e) SFSA-TMP non-strided (f) SFSA-SMP strided (g) SFSA-TMP strided (h) MFWA-SMP (i) MFWA-TMP (continued) (see online version for colours)



(h)



(i)

4.5 Classes of tests

In order to simulate all the situations presented in this section, we define the complete set of classes of tests, described in Figure 3, as follows:

1 SFSA
   a SFSA non-strided
   b SFSA strided
   c SFSA-SMP
      - SFSA-SMP non-strided
      - SFSA-SMP strided
   d TMP
      - SFSA-TMP non-strided
      - SFSA-TMP strided

2 MFWA
   a MFWA
   b MFWA-SMP
   c MFWA-TMP.

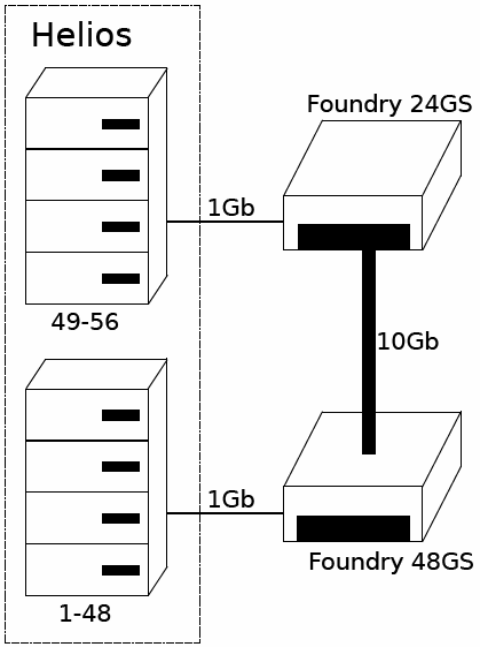
5 Evaluating I/O strategies under Lustre

This section presents the results obtained with the classes of tests proposed in the previous section, along with discussion over the results. We first describe the methodology of the tests. The other subsections explore variations on parameters for the proposed classes of tests – granularity, temporal aspect and scalability.

5.1 Methodology

The tests were executed on the Helios cluster from Grid5000 (Bolze et al., 2006). It has 56 bi-processed Sun Fire X4100 nodes with AMD Opteron 2.2 GHz, 4 GB of RAM and connected in Gigabit Ethernet to 2 Foundry EdgeIron switches (one EdgeIron 48GS and one EdgeIron 24 GS). The switches are interconnected using a 2 × 10 Gbps stack (10 Gbps in each direction). The interconnection of the nodes is presented in Figure 4.

Figure 4 Interconnection of the Helios cluster



Each node has two 73 GB SAS disks set as a single hardware RAID-0 device. All the nodes were prepared with the same image, using Linux 2.6.18.3 and local file system ext3 (with block size of 4 KB).

To implement the tests, we used the MPI-IO test tool (version 21) (LAN, 2008), that uses MPI-IO API calls. A test is described with parameters that determine:

- the operation to be done: write, read or both
- the number of files involved: one shared or one per process
- barriers before or after operations
- how the file must be distributed among the processes: *strided* or *non-strided*

- the number of segments and their size to be accessed in each client.

Instead of using MPI-IO test, we could have chosen another I/O benchmarking tool, like IOzone (Norcott et al., n.d.) or IOR (Shan and Shalf, 2007). The choice was made considering facility to represent the access patterns, use, make modifications in the code, and interpret the results and even the familiarity of the authors with the tool.

Lustre's configuration was four data servers, with circular striping starting at a random server. In the OSTs, the storage device is the local file system ext3. Fail-over servers were not used, so there was no data replication.

The cache configuration on the clients was kept at its default values: at most 40 MB of read-ahead, 8-RPC on the flight and 32 MB of dirty data on cache. Each client has these limits for each OST.

Since a variety of factors of hardware and Lustre configuration impact the I/O performance, the results of our study are strongly coupled with our tests environment configuration. However, this hardware configuration is very typical in clusters around the world, and all (but striping) parameters used for Lustre were the default ones. Therefore, the results are specific, but for a 'very common specific situation'.

The amount of data accessed by each process was 2 GB in all the tests. We believe that 2 GB is enough to study properly the file system behaviour, given the small cache size.

Table 1 lists all the tests that were developed and executed along with the parameters used. They can be divided in three major groups:

1 *Tests varying the number and size of segments (NSEG)*: these tests intend to verify the impact presented by granularity of accesses in their performance. While number and size of segments varies, the total amount of data read/written by each client remains always the same: 2 GB.

2 *Tests varying the wait time between two consecutive reads/writes (TMP)*: these tests verify the impact of temporal access patterns in the performance.

3 *Tests varying the number of clients that access concurrently the PFS (CLIENTS)*: these tests aim at evaluating the scalability of the system. The amount of data accessed by each client is fixed, i.e., the total amount of read/written data grows with the number of clients.

Additionally, we verified the sensibility of the performance to other configuration parameters, like the stripe size, size of segments for the tests varying the number of clients and number of processes by client node.

Although the stripe size is one of the studied parameters, we do not consider situations where the requests size are not aligned with the stripe size. As previously stated, the related work (Liao et al., 2007) presents a discussion on this problem.

More configurations were tested for the classes SFSA non-strided and MFWA than for the SFSA strided one. This choice was made considering that these two classes are alternative solutions for a same problem (contiguous accesses). Therefore, developers of applications with this characteristic need guidance in order to choose the best solution in each case. In the other hand, an application of the SFSA strided class may be limited by file formats, not being able to change its access pattern.

The tests have two phases – read and write – executed separately. Between them, the clients are restarted in order to avoid cache effects. Barriers were placed after open file operations and before closing files. The results are arithmetic averages of series of repetitions (minimum 4) and have a confidence of 95% and maximum error of 5%.

Table 1 Parameters used for the tests

Test	*Number of processes*	*Number of client nodes*	*Stripe size*	*Number of segments*	*Segment size*	*Wait time between reads/writes*
SFSA-non-strided-NSEG	40	40	64 KB, 1 MB, 64 MB	2 to 128 M	16 bytes to 1 GB	-
SFSA-strided-NSEG	40	40	64 KB	2 to 2 K	1 MB to 1 GB	-
MFWA-NSEG	40	40	64 KB, 1 MB, 64 MB	2 to 128 M	16 bytes to 1 GB	-
SFSA-non-strided-NSEG-SMP	40	10	64 KB	2 to 128 M	16 bytes to 1 GB	-
SFSA-strided-NSEG-SMP	40	10	64 KB	2 to 256 K	8 KB to 1 GB	-
MFWA-NSEG-SMP	40	10	64 KB	2 to 128 M	16 bytes to 1 GB	-
SFSA-non-strided-TMP	40	40	64 KB	32	64 MB	1 to 5,000 ms
SFSA-strided-TMP	40	40	64 KB	32	64 MB	1 to 5,000 ms
MFWA-TMP	40	40	64 KB	32	64 MB	1 to 5,000 ms
SFSA-non-strided-CLIENTS	1 to 48	1 to 48	64 KB	32, 32 KB	64 MB, 64 KB	-
SFSA-strided-CLIENTS	1 to 40	1 to 40	64 KB	32, 32 KB	64 MB, 64 KB	-
SFSA-non-strided-CLIENTS	1 to 48	1 to 48	64 KB	32, 32 KB	64 MB, 64 KB	-

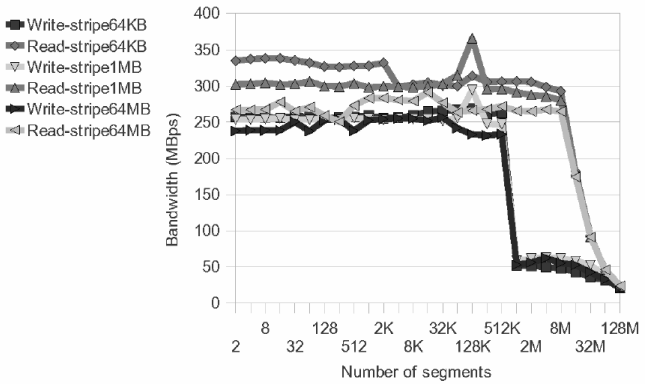
To determine the bandwidth, we measured the total time expended to finish the test, and then calculated number of processes * 2 GB/total time. For the tests that have wait time between operations, the idle time was deducted from the total time. Since the nodes are interconnected through a Gigabit Ethernet and there are four servers, the peak bandwidth expected was always lower than 512 MB/s.

5.2 Access granularity on the application side

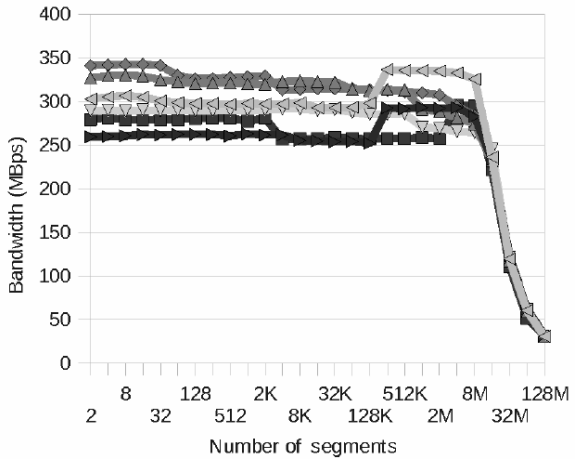
This section evaluates the impact on the I/O performance of the granularity of the requests made by an application. The results presented here correspond to the test with 40 clients and a total amount of data accessed by each client of 2 GB, varying the quantity of segments. The larger the number of segments, the smaller their size.

The results are presented in Figure 5 and Figure 6. Each pair of data-series represents the read and write bandwidth obtained by the test with a given *stripe size* for Lustre – i.e., size of a block of the file stored contiguously in a single OST. The x-axis represents the *number of logical segments* that are operated by an application – in other words, the number of logical units of data that an application reads or writes in a single read or write request. The y-axis represents the I/O bandwidth for the test.

Figure 5 Results of the NSEG test with the classes SFSA non-strided and MFWA, (a) SFSA-non-strided-NSEG (b) MFWA-NSEG

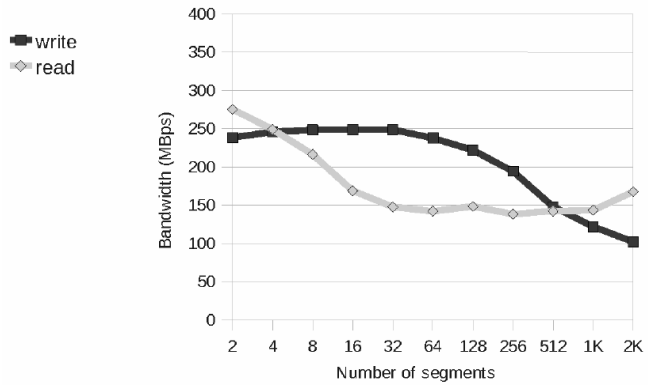


(a)

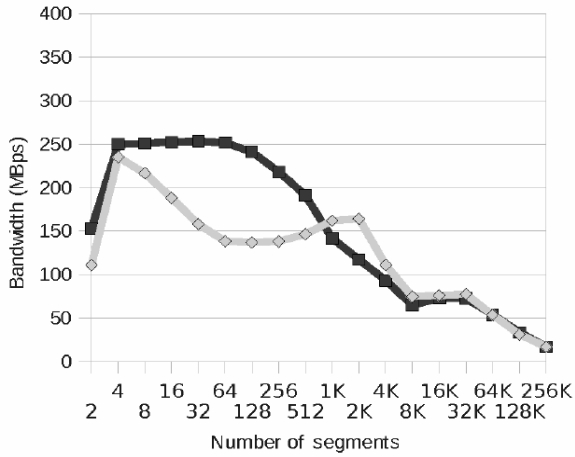


(b)

Figure 6 Results of the NSEG and NSEG-SMP tests with the class SFSA strided, (a) SFSA-strided-NSEG (b) SFSA-strided-NSEG-SMP



(a)



(b)

Considering only the non-SMP tests [Figures 5(a), 5(b) and 6(a)], the worst performance is shown in the SFSA-strided test. The non-strided SFSA approach obtained throughput up to 150% better than the strided one for write and 98% for read operations.

In the SFSA-strided-NSEG [Figure 6(a)], write operations' performance is stable for numbers of segments from 2 to 128 (segment size from 1 GB to 16 MB). In this interval, results are very similar to the ones shown by the other non-SMP tests. With more segments, however, the granularity of accesses causes the test performance to drop. The smaller and more sparse the segments, worse is the bandwidth. Read operations suffer more with the increased granularity because the read-ahead performed by the clients ceases to be advantageous as it requests too much data that are not used by the client.

This behaviour is not observed in the tests where the accesses are done to contiguous portions of the files (Figure 5). These strategies present a nearly-constant performance until there are too many segments and their size becomes too small. In this case, then the bandwidth falls quickly.

For read operations in the SFSA-non-strided-NSEG test and number of segments under 16 M, all the variations are inside the error interval. With more segments, the small

request size (under 128 bytes) causes the bandwidth to decrease about 46% every time the number of segments doubles. The same behaviour is observed in the MFWA-NSEG test for both read and write operations.

However, write operations in the SFSA non-strided approach present a different behaviour. When the number of segments reaches 1 M (requests of 2 KB), the bandwidth decreases approximately 78%. Afterwards, it remains nearly-constant until the number of segments reaches 16 M (as in the read operations). At this point, the performance falls at a rate of 20% every time the number of segments doubles.

5.2.1 Impact of Lustre stripe size

In this section, we evaluate how the stripping unit used by Lustre to distribute data on the OSTs can affect the I/O performance.

In SFSA-non-strided-NSEG and MFWA-NSEG tests, the observed behaviour is similar for the three stripe sizes tested. In the SFSA non-strided approach, the difference between the performances of write operations with the three stripe sizes is not statistically significant until 64 K segments (of size 32 KB). With number of segments from 64 K to 1 M, using stripes of 64 KB and 1 MB presents better performance than using those of 64 MB (in average 16% better). For more segmented accesses, the best choice for stripe size is 1 MB (in average 23% better), the worst performance being shown using a stripe of 64 KB.

The difference between the three stripe sizes is, however, always significant for write operations in the MFWA approach. Stripe sizes of 1 MB and 64 KB present the best performance (about 12% better) until the number of segments reaches 256 K (size reaches 8 KB), when the best choice becomes stripe size of 64 MB (about 12% better) and 64 KB stripes presents the worst performance.

For read operations, before the number of segments reaches 4 K (size 512 KB), stripe size of 64 KB presents the best performance and 64 MB has the worst for both tests (11% in MFWA and 23% in SFSA). After that, for the SFSA non-strided approach, 1 MB has the best bandwidth (17%) from 64 K segments to 8M (size from 32 KB to 256 bytes), where the worst is shown by stripe size of 64 MB. For the MFWA strategy, 64 MB has the best performance (17%) from 1 M segments to 8 M (size from 2 KB to 256 bytes).

5.2.2 Concurrent intra-node access

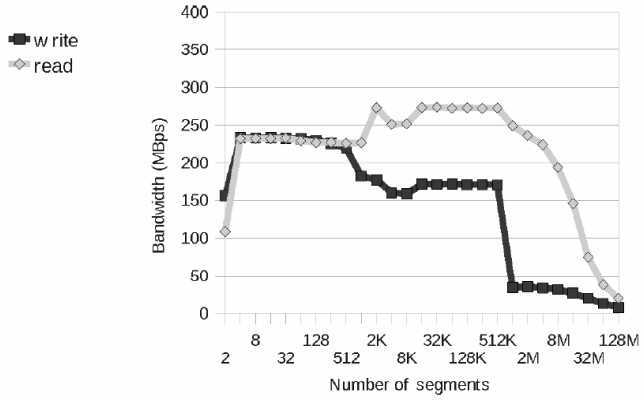
This section presents an evaluation on how the use of multiple processes per client node affects the I/O performance. The results of the SMP tests are shown in Figures 6(b), 7(a), and 7(b).

All the results of these tests present an initial gain of performance (in average 90%) in the change from 2 to 4 in the number of segments. After that, the SFSA-non-strided-NSEG-SMP test has a constant behaviour for accesses using less than 1,024 segments. At this point, write operations' bandwidth is decreased in 17%, while reads bandwidth is increased in 21%. When the number of segments reaches 1 M for write operations and 16 M for reads, the performance starts to fall quickly, as observed in the non-SMP version [Figure 5(a)]. In average, the non-SMP test had bandwidth 52% better for writes and 36% for read operations.

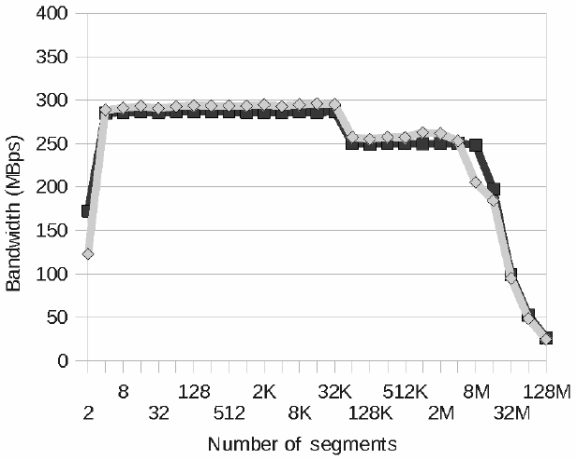
The MFWA-NSEG-SMP test has a constant behaviour for accesses using less than 64 K segments. Then, the bandwidth (for both the operations) decreases 13%. The bandwidth starts to fall again when the number of segments reaches 16 M, also as in the non-SMP version [Figure 5(b)]. In average, write operations had similar bandwidth of the non-SMP test, and reads were 23% worse.

The bandwidth of the SFSA-strided-NSEG-SMP test does not have statistically significant difference to the result of the non-SMP version.

Figure 7 Results of the NSEG-SMP test with the classes SFSA non-strided and MFWA, (a) SFSA-nonstrided-NSEG-SMP (b) MFWA-NSEG-SMP



(a)



(b)

5.2.3 Discussion

The results presented in the previous sections indicate that the granularity of accesses has impact in applications I/O performance on Lustre only when these accesses are make to *sparse portions of a file, or when to contiguous areas but are done in a large amount of small requests*. This 'large

amount of small requests' means 16 M segments of 128 bytes for all the cases except for write operations in the SFSA non-strided approach, where it means 1 M segments of 2 KB.

This 'large amount of small requests' value does not depend on the stripe size used, since all the three stripe sizes tested presented the same behaviour. It does not depend on the number of processes executing in each node either, since the behaviour was observed in the SMP results too.

For the MFWA strategy, better performance is achieved with larger stripe sizes (up to 17% better), except for read operations with segments larger than 512 KB. Using the SFSA non-strided approach, 64 KB is the best stripe size (up to 23% better) for large segments and 1 MB is the best for smaller segments (23%).

Decreasing the number of client nodes from 40 to 10 (executing four processes in each node), the performance was not degraded with the SFSA strided approach and for write operations with the MFWA approach. It was decreased in 23% for reads with MFWA, 36% for reads with the SFSA non-strided strategy and 52% for writes with SFSA non-strided.

These results show that, dividing the number of machines by 4, the performance was at most divided by 2. With 40 nodes we are then under-utilising the resources in the clients. Therefore, the limiting in the bandwidth in our tests was the number of data servers (4). In real employments of Lustre, it is usually applied a greater amount of servers.

5.3 Temporal behaviour and I/O performance

This section presents the results for the case where applications have interleaved I/O and processing phases. This way, the servers will observe variations on the I/O loads incoming from the clients. The test changes the time spent between two consecutive operations. The number of clients is fixed in 40 with 32 segments of 64 MB.

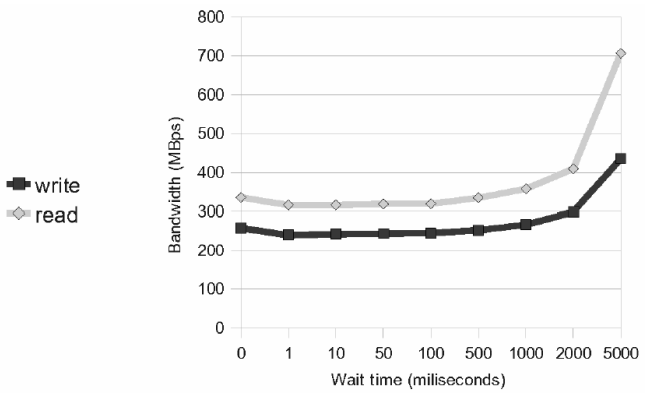
The results can be seen in Figure 8. In this test, the x-axis represents the amount of time spent between each I/O operation (in milliseconds).

In all the tests, the insertion of an interval longer than 2 seconds between consecutive operations caused an increase in the I/O performance. In the SFSA-strided test [Figure 8(b)], this increase was of 41% for write and 27% for read operations. With the non-strided SFSA approach [Figure 8(a)], the improvement was of 46% for writes and 72% for reads. Using the MFWA strategy [Figure 8(c)], write operations' performance was increased in 59% and 58% for reads.

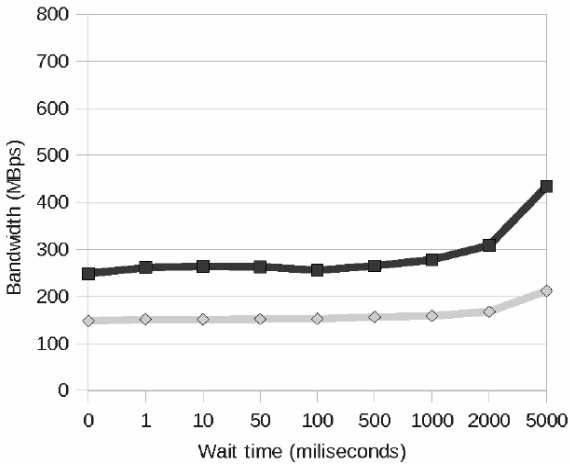
As there were no barriers to guarantee synchronisation between the operations, the results indicate that some processes were doing I/O while others were in the processing phase. This way, during some periods, there were less than 40 processes concurrently accessing the file system, producing a better individual bandwidth. This indicates that Lustre correctly handles variation in the load from individual's clients.

The SFSA-strided-TMP test presented the lowest increase in performance when compared to the other approaches. This situation is coherent with the results presented in Section 5.2, since the strided SFSA presented lower throughput in the standard test. For long intervals (5,000 ms), though, the performance of SFSA-strided-TMP was very similar to the one of the non-strided SFSA.

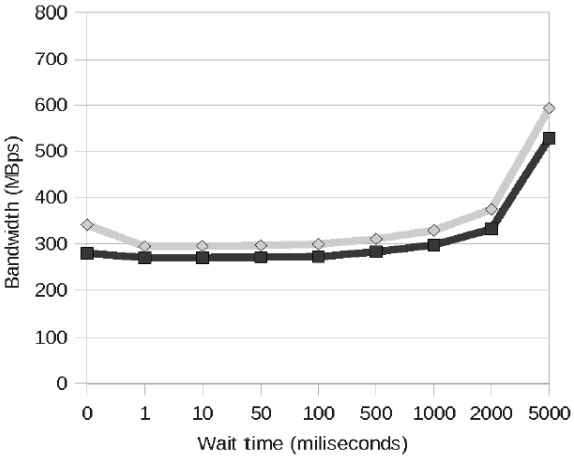
Figure 8 Results of the TMP tests, (a) SFSA-non-strided-TMP (b) SFSA-strided-TMP (c) MFWA-TMP



(a)



(b)



(c)

For the non-strided tests, the increase was superior (and so was the bandwidth) in MFWA for writes, which is consistent with the tests of Section 5.2 since MFWA provided higher throughput than SFSA. On the other hand, the read operations of MFWA and SFSA presented equivalent bandwidths given the error margin, only diverging on intervals longer than 1,000 ms. The comparison between these two strategies will be better explored in the next section.

These results indicate that applications that present interleaved I/O and processing phases and do not need synchronisation between each of them can achieve better performance with Lustre by adjusting their processing phases to be long enough to allow the pending operations to complete. In these tests, a processing phase of at least 2 seconds was the minimum necessary to force such scenario. It's important to note that, for applications that perform communication during their I/O phases, there may be interference and further slowing down of the I/O performance.

5.4 Scalability on the application size

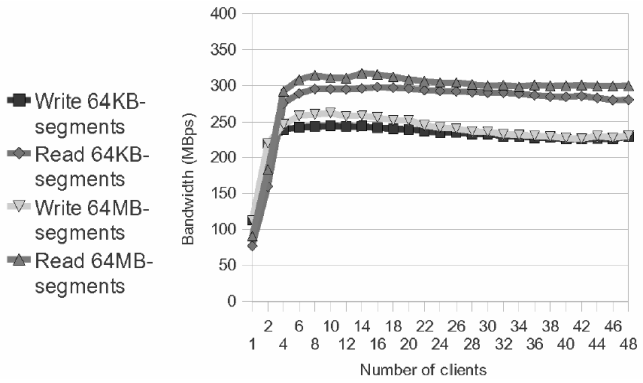
In this section, we will discuss the evaluation of the scalability that an application can expect of the file system. The results presented here vary the number of clients accessing concurrently the file system. Each client accesses 2GB of data, so the total amount of data increases with the number of clients. Since the number of servers is fixed, the ideal result should not present degradation in the throughput as more clients are added. We executed tests with two segments sizes for each test: 32 segments of 64 MB and 32 K segments of 64 KB. The results are presented in Figure 9.

In the results for the multiple files approach [Figure 9(c)], none of the differences between the bandwidths for the two sizes was significant, and no performance degradation was observed as the amount of data grew. The differences between the two sizes of segments are not significant in the SFSA non-strided test too. However, this test had a performance degradation for read operations with the growth in the number of clients (11% for 64 MB segments), indicating that this approach does not scale well.

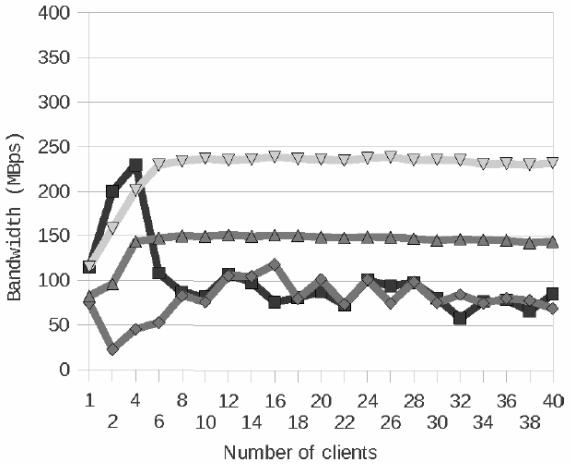
A different behaviour is shown in the results of the test SFSA-strided-CLIENTS [Figure 9(b)]. With large segments, the performance quickly stabilises and does not suffer degradation with the growth in the number of clients. With 64KB segments, however, the bandwidth is substantially lower.

With 64KB segments, up to four clients, since the segment size is the same as the stripe size, each client is attended exclusively by one server. The performance is lower because not all the servers are accessed. After the performance stabilises, both the phases (read and write) have peaks in numbers of processes multiple of 4. This happens because, with multiples of 4, each node accesses always the same server. In other cases, clients alternate their requests between two servers. As shown in the related work (Dickens and Logan, 2008), accessing fewer servers provides better performance. However, as the number of clients grows, this effect loses its importance, and the peaks vanish.

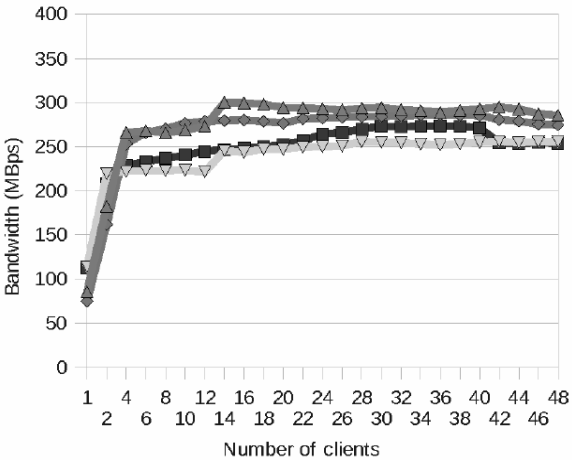
Figure 9 Results of the CLIENTS tests,
(a) SFSA-non-strided-CLIENTS
(b) SFSA-strided-CLIENTS (c) MFWA-CLIENTS



(a)



(b)



(c)

Table 2 to Table 4 compare some results from SFSA non-strided and MFWA approaches varying the number of clients with the two tested segments size. They show only the results which had difference between the classes over

10% (tolerated error). Therefore, the results for read operations with 64 KB segments are not shown as they do not match this criteria.

Table 2 and Table 3 show that, when accesses are made with big segments (64 MB), the single file approach performs on average 13% better for fewer clients (up to 12, i.e., three times the number of data servers), for read and write operations. For read operations (Table 3), there are not other significant differences between the two approaches.

Table 2 Bandwidth (MBps) of read operations with 64 MB segments

Clients	*SFSA non-strided*	*MFWA*	*Difference*
4	246.83	221.95	≈ 10%
6	258.66	222.88	≈ 13%
8	260.87	222.61	≈ 14%
10	262.61	223.04	≈ 15%
12	258.03	221.30	≈ 14%
38	229.19	253.08	≈ 10%
40	227.09	254.79	≈ 12%
42	226.30	256.08	≈ 13%
44	229.74	254.42	≈ 10%
46	227.30	255.59	≈ 12%
48	230.35	255.41	≈ 10%

Table 3 Bandwidth (MBps) of read operations with 64 MB segments

Clients	*SFSA non-strided*	*MFWA*	*Difference*
6	308.12	268.02	≈ 13%
8	314.09	265.52	≈ 15%
10	310.49	268.55	≈ 14%
12	310.12	272.88	≈ 12%

Table 4 Bandwidth (MBps) of write operations with 64 KB segments

Clients	*SFSA non-strided*	*MFWA*	*Difference*
24	234.22	264.12	≈ 13%
26	234.71	265.84	≈ 13%
28	232.46	269.81	≈ 16%
30	232.39	272.75	≈ 17%
32	229.44	272.31	≈ 19%
34	229.48	273.00	≈ 19%
36	227.17	273.44	≈ 20%
38	228.12	273.13	≈ 20%
40	225.31	270.74	≈ 20%
42	225.61	254.23	≈ 13%
44	226.53	253.19	≈ 12%
46	225.62	254.65	≈ 13%
48	229.41	253.56	≈ 11%

For write operations (Table 2 and Table 4) using more clients, the multiple files approach performs on average 14% better than the contiguous portions on a shared file one. For 64 KB segments, this gap started to be significant with a lower number of clients (24, against 38 for 64 MB segments).

6 Conclusions and future works

The performance of applications when executed using a PFS will strongly depend on the strategies applied to its data organisation and requests characteristics. Additionally, the performance will be dependent on the behaviour that these strategies present on a real PFS. This way, studying the performance of a range of such strategies is essential to develop applications with good I/O performance.

A common situation in scientific application handles data exclusive to the clients – i.e., in a same step of the computation, two clients do not need access to the same data. Two ways of implementing these accesses are through an exclusive file per instance of the application, or a single file shared by all of them, where processes have exclusive segments in it. Using the single file approach, each client can have several sparse portions of the file, or just one contiguous region.

This work presented a study on I/O strategies that applications may apply in this situation and identified the ones that obtained better performance on Lustre file system. The shared file and multiple files approaches, additionally to other parameters like temporal behaviour and presence of concurrency intra-node, were represented by classes of access patterns and evaluated under Lustre. The results provide some conclusions on these strategies:

- The SFSA strided approach, where each process has several sparse segments on a shared file, has performance up to *98%* worse than the non-strided approach for read and *150%* for write operations. This strategy should be avoided and, if possible, data should be reorganised in order to create large contiguous segments in the file. Other alternative would be to organise data in independent files and reorganise them during the processing phases.
- If the accesses are to sparse areas of a shared file, the granularity has a big impact in the performance: up to *77%* for writes and *57%* for read operations. The more sparse and smaller the segments, the worse are the bandwidth.
- For accesses to contiguous portions of data, the granularity of the requests does not have impact in the performance unless the segments – and thus, the requests to the servers – become too small. In this case, performance decreases up to *46%* every time the segment size is cut in half. This threshold in the tests was 128 bytes for MFWA tests and for the read operations of non-strided SFSA. For write operations in

the single file approach, the threshold was 2 KB. This value does not depend on the used stripe size or on the number of processes in each node.

- When using the multiple files strategy, larger stripe sizes should be used (1MB or 64 MB), except for read operations of big segments (larger than 512 KB), since these strategies provide a performance up to *12%* better. For the non-strided single file strategy, data should be striped on 1 MB blocks if the logical segments are small and 64 KB for large logical units. This provides an increase in bandwidth of up to *23%*.

- Placing four processes in each node degraded in *23%* the performance of read operations in MFWA, *36%* for read and *52%* for writes in non-strided SFSA.

  Still, the decrease in performance is small comparing to the decrease in the number of nodes (40 to 10), indicating that the servers were the bottleneck. When possible, a bigger number of data servers should be used for the benefit of applications.

- When the application presents temporal behaviour and no need for barriers to synchronise clients after each processing phase, the individual nodes take advantage of others' processing phases and obtain better individual bandwidths. With intervals longer then 2 seconds, the observed gains are up to 72% in read operations and 59% in writes.

- If an application presents long processing phases (5 seconds or more), the performance differences in strided and non-strided SFSA vanish. In that case, data reorganisation to improve I/O performance may be unnecessary.

- For fewer clients (up to three times more than the number of servers) and requests to large segments, the shared file approach performs up to *15%* better than the multiple files one. In other situations – about ten times greater than the number of servers – they are both equivalent for read operations. In this case, for write operations, the multiple files approach has a better performance (up to *20%*).

As future work, we plan to investigate strategies on client-side and server-side to reorganise accesses to the file system based on the conclusions of this work. These strategies should intentionally cause the situations that, as we have shown, result in better performance.

References

Ávila, R.B., Navaux, P.O.A., Lombard, P., Lebre, A. and Denneulin, Y. (2004) 'Performance evaluation of a prototype distributed NFS server', in *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2004)*, Foz do Iguaçu, Brazil.

Bolze, R., Cappello, F., Caron, E., Dayde, M., Desprez, F., Jeannot, E., Jegou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E-G. and Touche, I. (2006) 'Grid'5000: a large scale and highly reconfigurable experimental grid testbed', *International Journal of High Performance Computing Applications*, Vol. 20, No. 4, pp.481–494.

Brandt, S.A., Miller, E.L., Long, D.D.E. and Xue, L. (2003) 'Efficient metadata management in large distributed storage systems', in *MSS '03: Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, IEEE Computer Society, p.290, Washington, DC, USA.

Carballeira, F.G., Calderón, A., Carretero, J., Fernández, J. and Perez, J.M. (2003) 'The design of the expand parallel file system', *IJHPCA*, Vol. 17, No. 1, pp.21–37.

CFS (2002) 'Lustre: a scalable, high-performance file system', Cluster File Systems Inc., White paper, version 1.0, available at http://www.lustre.org/docs/whitepaper.pdf.

Corbett, P.F. and Feitelson, D.G. (1996) 'The Vesta parallel file system', *ACM Transactions on Computer Systems*, Vol. 14, No. 3, pp.225–264.

Coulouris, G., Dollimore, J. and Kindberg, T. (2005) *Distributed Systems, Concepts and Design*, 4th ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Darling, A.E., Carey, L. and Feng, W-C. (2003) 'The design, implementation, and evaluation of MPIblast', in *Proceedings of Cluster World 2003*.

Dickens, P. and Logan, J. (2008) 'Towards a high performance implementation of mpi-io on the lustre file system', in *OTM '08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on the Move to Meaningful Internet Systems*, pp.870–885, Springer-Verlag, Berlin, Heidelberg.

Dickens, P.M. and Logan, J. (2009) 'Y-lib: a user level library to increase the performance of mpi-io in a lustre file system environment', in *HPDC '09: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, pp.31–38, ACM, New York, NY, USA.

Fryxell, B., et al. (2000) 'Flash: an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes', *The Astrophysical Journal Supplement Series*, Vol. 131, pp.273–334.

Ghemawat, S., Gobioff, H. and Leung, S-T. (2003) 'The Google file system', *ACM SIGOPS Operating Systems Review*, Vol. 37, No. 5, pp.29–43.

Hildebrand, D. and Honeyman, P. (2007) 'Direct-PNFS: scalable, transparent, and versatile access to parallel file systems', in *Proceedings of the 16th International Symposium on High Performance Distributed Computing – HPDC 07*, pp.199–208, ACM, New York, NY, USA.

Kotz, D. and Ellis, C.S. (1993) 'Practical prefetching techniques for multiprocessor file systems', *Distributed and Parallel Databases*, Vol. 1, No. 1, pp.33–51.

LAN (2008) 'Los Alamos National Laboratory, MPI-IO test user's guide', available at http://institutes.lanl.gov/data/software/src/mpi-io/README_21.pdf.

Larkin, J. and Fahey, M. (2007) 'Guidelines for efficient parallel I/O on the Cray XT3/XT4', in *Cray Users Group Meeting (CUG) 2007*, Seattle, Washington.

Latham, R., Miller, N., Ross, R. and Carns, P. (2004) 'A next-generation parallel file system for Linux clusters', *LinuxWorld*, Vol. 2, No. 1, pp.56–59.

Lebre, A., Denneulin, Y., Huard, G. and Sowa, P. (2006) 'I/O scheduling service for multi-application clusters', in *Proceedings of IEEE Cluster 2006, Conference on Cluster Computing*.

Liao, W., Ching, A., Coloma, K., Choudhary, A. and Kandemir, M. (2007) 'Improving MPI independent write performance using a two-stage write-behind buffering method', in *Parallel and Distributed Processing Symposium, IEEE International. IPDPS 2007*, p.295, IEEE Computer Society, Los Alamitos, CA, USA.

Martin, R.P. and Culler, D.E. (1999) 'NFS sensitivity to high performance networks', *ACM SIGMETRICS Performance Evaluation Review*, Vol. 27, No. 1, pp.71–82.

Miller, E.L. and Katz, R.H. (1991) 'Input/output behaviour of supercomputing applications', in *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pp.567–576, ACM, New York, NY, USA.

MPI-IO Committee (1996) *MPI-IO: A Parallel File I/O Interface for MPI Version 0.5*.

Norcott, W., et al. (n.d.) 'Iozone benchmark', available at http://www.iozone.org.

Patterson, D.A. and Hennessy, J.L. (2004) *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA.

Piernas, J., Nieplocha, J. and Felix, E.J. (2007) 'Evaluation of active storage strategies for the lustre parallel file system', in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pp.1–10, ACM, New York, NY, USA.

Qian, Y., Barton, E., Wang, T., Puntambekar, N. and Dilger, A. (2009) 'A novel network request scheduler for a large scale storage system', *Computer Science – R&D*, Vol. 23, Nos. 3–4, pp.143–148.

Schaller, R.R. (1997) 'Moore's law: past, present, and future', *IEEE Spectrum*, Vol. 34, No. 6, pp.52–59.

Schmuck, F. and Haskin, R. (2002) 'GPFS: a shared-disk file system for large computing clusters', in *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, p.19, USENIX Association, Berkeley, CA, USA.

Seamons, K.E., Chen, Y., Jones, P., Jozwiak, J. and Winslett, M. (1995) 'Server-directed collective I/O in panda', in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM)*, p.57, ACM, New York, NY, USA.

Shan, H. and Shalf, J. (2007) 'Using IOR to analyze the I/O performance for HPC platforms', Technical report, Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US).

Smirni, E. and Reed, D. (1998) 'Lessons from characterizing the input/output behavior of parallel scientific applications', *Performance Evaluation*, Vol. 33, No. 1, pp.27–44.

Smirni, E. and Reed, D. A. (1997) 'Workload characterization of input/output intensive parallel applications', in *Proceedings of the 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pp.169–180, Springer-Verlag.

SUN (2008) 'Lustre networking: high-performance features and flexible support for a wide array of networks', Sun Microsystems White paper, version 1.0.

Sun Microsystems (1989) *NFS: Network File System Protocol Specification*.

Thakur, R., Gropp, W. and Lusk, E. (1998) 'Data sieving and collective I/O in romio, in *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pp.182–189, IEEE Computer Society Press.

Thakur, R., Gropp, W. and Lusk, E. (1999) 'On implementing MPI-IO portably and with high performance', in *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pp.23–32, ACM Press.

Top500 (2010) 'Top500 supercomputer sites', available at http://www.top500.org/.

Wang, F., Oral, S., Shipman, G., Drokin, O., Wang, T. and Huang, I. (2009) 'Understanding lustre file system internals', Technical Report ORNL/TM-2009/117, Oak Ridge National Lab., National Center for Computational Sciences.

Wang, F., Xin, Q., Hong, B., Brandt, S.A., Miller, E.L., Long, D.D.E. and Mclarty, T.T. (2004) 'File system workload analysis for large scale scientific computing applications', in *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp.139–152.

Welch, B., Unangst, M., Abbasi, Z., Gibson, G., Mueller, B., Small, J., Zelenka, J. and Zhou, B. (2008) 'Scalable performance of the Panasas parallel file system', in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pp.1–17, USENIX Association, Berkeley, CA, USA.

Winstead, C., Pritchard, H. and McKoy, V. (1995) 'Parallel computation of electron-molecule collisions', *IEEE Comput. Sci. Eng.*, Vol. 2, No. 3, pp.34–42.

Yu, W., Oral, H.S., Canon, R.S., Vetter, J.S. and Sankaran, R. (2008) 'Empirical analysis of a large-scale hierarchical storage system', in *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pp.130–140, Springer-Verlag, Berlin, Heidelberg.

Zhang, J., Sivasubramaniam, A., Franke, H., Gautam, N., Zhang, Y. and Nagar, S. (2004) 'Synthesizing representative i/o workloads for TPC-H', in *Proceedings on 10th International Symposium on High Performance Computer Architecture, 2004. HPCA-10*, pp.142–142.

Zhang, X., Jiang, S. and Davis, K. (2009) 'Making resonance a common case: a high-performance implementation of collective I/O on parallel file systems', in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IEEE Computer Society, pp.1–12, Washington, DC, USA.

Zhao, T., March, V., Dong, S. and See, S. (2010) 'Evaluation of a performance model of lustre file system', in *ChinaGrid Conference (ChinaGrid), 2010 Fifth Annual*, IEEE, pp.191–196.