JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 18, 825-836 (2002)

Short Paper

Performance Modeling and Evaluation of MPI-I/O on a Cluster*

JACOBO BARRO, JUAN TOURIÑO, RAMON DOALLO
AND VICTOR M. GULIAS+
Department of Electronics and Systems
+*Department of Computer Science*
University of A Coruña
15071 A Coruña, Spain
E-mail: juan@udc.es

Cluster computing is an area of growing interest in search to support parallel and distributed applications. Many of these applications are I/O intensive, and the limited bandwidth of the I/O subsystem of the cluster is an important bottleneck that is usually ignored. Thus, the performance of parallel I/O primitives is critical for overall cluster performance. In this work, we characterize the performance of basic ROMIO MPI-I/O routines on a PC cluster using the NFS and PVFS file systems. Our goal is to detect weak spots in the use of these routines and to predict their impact on the application's performance.

Keywords: cluster computing, performance analysis, parallel I/O, MPI-I/O, ROMIO, NFS, PVFS

1. INTRODUCTION

Many parallel applications require huge data sets that have to be stored on disk. For instance, the authors have developed parallel scientific and engineering applications in diverse areas such as fluid mechanics [1], image synthesis [2] and environmental chemistry [3] that require efficient I/O to ensure acceptable performance. Out-of-core computation is another typical example of intensive I/O.

MPI-I/O [4] provides a standard parallel I/O interface. The performance of the I/O primitives depends not only on the disk and network hardware, but also on the underlying file system. It is clear that programming portability does not mean performance portability. Although an exhaustive set of experiments can be done on a cluster to assess quantitatively the performance of the I/O subsystem, we have focused on low-level tests to study basic MPI-I/O primitives. Our aim is to estimate I/O overheads with simple expressions, which can help application developers to design I/O-intensive parallel programs more efficiently. Benchmark suites for MPI-I/O functions were presented in [5, 6].

Received August 31, 2001; accepted April 15, 2002.
Communicated by Jang-Ping Sheu, Makoto Takizawa and Myongsoon Park.
* This research was supported by Xunta de Galicia (Project PGIDT01-PXI10501PR).

Although the authors did not adopt any particular data model, these reports are good starting points for deriving analytic models and developing more in-depth tests.

This work is organized as follows. In the next section, we comment on some related works. Section 2 gives an overview of parallel I/O topics (file systems and MPI-I/O routines). In section 3, the underlying configuration of the cluster used in the experiments is detailed; section 4 presents experimental results and performance models for some MPI-I/O routines, using both the standard NFS file system and PVFS, a parallel file system for clusters. Finally, conclusions are drawn in section 5.

1.1 Related Work

Mache et al. [7] studied the parallel I/O performance of PVFS on a PC cluster using a ray tracing application as a case study. They also compared the influence of different disk types (IDE vs SCSI) and networks (Fast Ethernet vs Gigabit Ethernet) on I/O performance. Taki and Utard [8] presented a straightforward port of ROMIO [9], an MPI-I/O implementation, on PVFS. They compared typical file accesses and data distributions of parallel applications using ROMIO, on both NFS and PVFS. In our work, we used a new version of PVFS with specific interface to ROMIO. Neither paper focused on specific routines from the MPI-I/O library, and they did not model the behavior of these primitives.

In this work, we do not report the raw I/O performance of NFS and PVFS, but rather the performance of specific ROMIO MPI-I/O primitives on both file systems, and we intend to model them analytically. Therefore, the results presented here are user-level oriented in order to give practical help for the development of parallel applications on clusters.

2. BACKGROUND TOPICS

2.1 File Systems: NFS and PVFS

The most widely available remote file system protocol is the Network File System (NFS) [10], designed by Sun Microsystems as a client-server application. It consists of a client part that imports file systems from other machines and a server part that exports local file systems to other machines. NFS is designed to be stateless. As there is no state to maintain or recover, NFS can continue to operate even during periods of client or server failures. Therefore, it is much more robust than a system that operates with a state, although the state requests to the server increase traffic. In addition, as the number of processors and the file size increase, the NFS server and the network are flooded with the client requests, which is an important bottleneck. In fact, NFS was not designed for large parallel I/O applications that require high-performance concurrent file accesses.

The Parallel Virtual File System (PVFS) [11] provides a high-performance and scalable parallel file system, and unlike other proprietary parallel file systems, it was developed for Linux PC clusters. PVFS spreads data out across multiple local disks in cluster nodes. Thus, applications have multiple paths to data through the network (which eliminates single bottlenecks in the I/O path) and multiple disks on which data is

stored. PVFS consists of three components: a metadata server, which maintains information in files and directories stored in the parallel file system; I/O servers that store data on local files; and clients that contact these servers to store and retrieve data. The metadata and I/O servers may be placed on dedicated resources or may be shared for computation purposes, in order to achieve a reasonable tradeoff between I/O and computing performance. PVFS provides multiple interfaces, including an MPI-I/O interface via ROMIO.

2.2 MPI-I/O Routines

MPI-I/O is a standard parallel file I/O interface, part of the MPI-2 specification [4]. An MPI file is an ordered list of MPI datatypes. A view of the file defines what data are visible to each processor. It consists of a displacement (an offset from the beginning of the file), an elementary type (the unit of data access and positioning within a file, which can be predefined or user-defined), and a filetype (a template for accessing the file).

Data access primitives are classified, based on the coordination, as *noncollective* (or independent) and *collective*. Noncollective routines, *MPI_File_{read|write}*, involve only one processor and an I/O request. Collective routines, *MPI_File_{read|write}_all*, involve all the processors that have opened a given file, and they can perform better than noncollective routines, because, as all the processors may coordinate, small requests may be merged (see the discussion of collective I/O optimization given later). In addition, MPI provides three types of positioning and, thus, three categories of data access routines: *individual file pointer* routines that use a private file pointer maintained by each processor and incremented by each read/write (they are the routines listed above); *explicit offset* primitives that take an argument that defines where the data is read or written, that is, *MPI_File_{read|write}_at* and the collective version *{read|write}_at_all*; and *shared file pointer* primitives, which use a shared file pointer, *MPI_File_{read|write}_shared*, and the collective *{read|write}_ordered*. All the enumerated routines are blocking routines; that is, they do not return until data transfer is completed. All of them have nonblocking counterparts, which do not wait for completion, in order to allow overlap of I/O with computation.

ROMIO [9] is a portable implementation of MPI-I/O that works on most parallel computers and networks of PCs/workstations, and supports multiple file systems (such as NFS and PVFS). It is optimized for noncontiguous access patterns (using derived datatypes), which are usually found in parallel applications, in order to reduce the effect of high I/O latency. Specifically, it implements data sieving and collective I/O optimizations [12]. Data sieving makes large I/O contiguous accesses and extracts in memory the data really needed, instead of making several small, noncontiguous accesses. Collective I/O optimization performs I/O in two stages: in the first one (the I/O stage), processors perform I/O for the merged request and, in the second one (the communication stage), processors redistribute data among themselves to achieve the desired distribution (this is for reading data; the order of the stages is reversed for writing). A proposal to improve performance of collective I/O of ROMIO on PVFS is presented in [13].

3. CLUSTER CONFIGURATION

Our PC cluster (see Fig. 1) consists of 24 nodes; one of them acts as a front-end providing services to the rest of the nodes (NFS, for instance). Each node has one AMD K6 processor and two Fast Ethernet interfaces, except for the front-end, which is a dual Pentium-II with an additional network interface attached to the departmental network. Two different networks separate IP administrative traffic from application traffic (MPI programs, for instance), and it is possible to combine both networks into a single virtual network to achieve higher throughput, by bonding both adapters (*channel bonding*). The switches are 24-port 3Com SuperStack II 3300 units stacked in groups of two with a 1Gbit/s link. They can be managed directly from a console or, once they have an IP address assigned by SNMP, from a telnet session or from an HTML client with Java support. From the management point of view, the switches are crucial devices, especially with Ethernet, where they can be integrated with external networks. Switches are suitable points for monitoring because only they can know the real status of physical links, and because having them probe the nodes avoids extra traffic from a management workstation. In our case, switches implement RMON monitoring: instead of directly managing individual nodes from the front-end or an external workstation, most of the work is done by the switch itself, which is then queried by SNMP from the workstation, thus reducing network traffic and complexity. Further information about the cluster configuration can be found in [14].

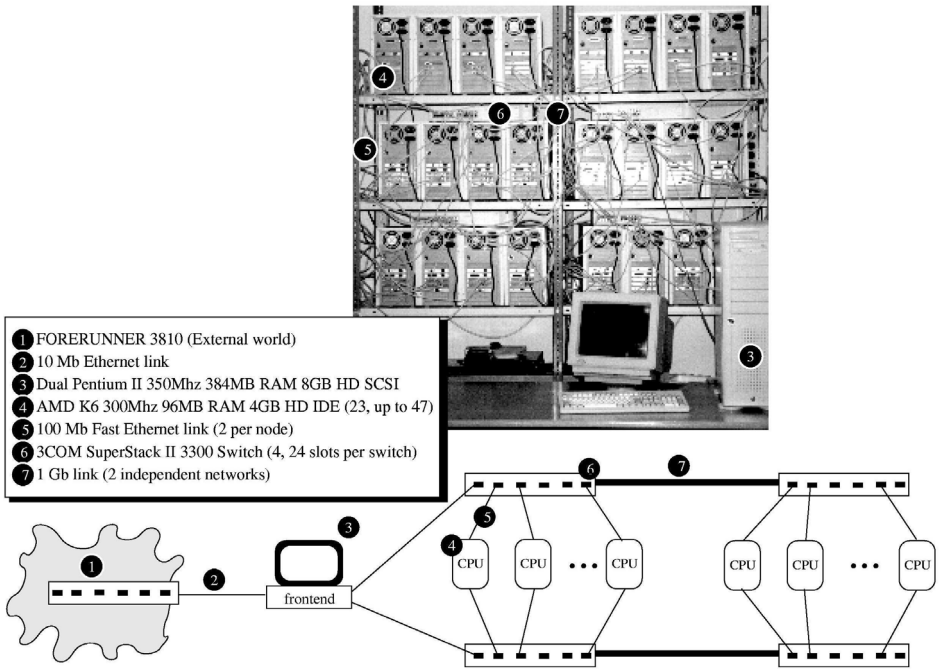

Fig. 1. PC cluster configuration.

For illustrative purposes, we have modeled point-to-point latency (*MPI_Send*) in the cluster as $T(n)=206 + 0.106n\ \mu s$, broadcast latency (*MPI_Bcast*) as $T(n, p) = 206log_2p + (0.105log_2p)n\ \mu s$ and reduction latency (*MPI_Reduce*, specifically sum reduction of doubles) as $T(n,p) = (435log_2p - 103) + (0.185log_2p)n\ \mu s$, where n is the message size in bytes and p the number of processors. As a comparison, in [15] we obtained the following results for the Fujitsu AP3000 multicomputer, composed of UltraSparc-II processors connected via a high-speed communication network (AP-Net): $T(n) = 69 + 0.0162n\ \mu s$ for point-to-point, $T(n,p) = 69log_2p + (0.0162log_2p)n\ \mu s$ for broadcast, and $T(n, p) = (90log_2p - 15) + (0.0222log_2p)n\ \mu s$ for reduction. As the AP-Net is a costly dedicated network, message-passing latencies in the cluster Fast Ethernet are much higher.

4. I/O EXPERIMENTAL RESULTS

4.1 Parallel I/O Performance Model

We have based our work on well-known message-passing communication models [15] with the aim of proposing the following simple model for parallel I/O operations: $T(n, p) = K(p)n$, where $T(n, p)$ is the execution time of the operation (in seconds), p is the number of processors, n is the file size (in MB), and $K(p)$ is the I/O time per data unit (in s/MB). Additional performance metrics (such as bandwidths) can be easily derived from this model. As we will show in sections 4.3 and 4.4, usually $K(p) = k/p$ or $K(p) = k/log_2p$. We have not considered a "startup" time parameter in this model (this would be the time it takes to perform an I/O operation on an empty file) because its cost is negligible in our framework of large files, which are our target for improving performance in real applications.

4.2 Experimental Conditions

We designed our own I/O tests. They were repeated with different file sizes (from 64KB to 32MB) and different numbers of processors. Timing outliers were taken into account to obtain accurate measures. As each test was repeated several times in a loop, a barrier was included to avoid a pipelined effect, where some processors might start the next call to the I/O operation even before all the processors have finished the current operation. The routine *MPI_File_sync*, which performs an I/O flush, was also used at appropriate points in the tests to avoid reads/writes from intermediate memory levels that could distort the performance results. The parameter $K(p)$ of the model was derived from a least-squares fit of T against n and p (from $p = 2$) using the minimum times obtained in the tests.

We installed NFS v3 and PVFS v1.5.0 under Debian Linux (kernel 2.2.18). We used ROMIO v1.0.3 with the MPI implementation MPICH v1.2.1 [16]. In practice, it is more usual to use an implicit file pointer than an explicit offset in I/O operations; thus, we discarded this set of primitives in our experiments. Regarding operations with shared file pointers, they involve serialization ordering (not deterministic for noncollective primitives), which is only desirable in some cases (for instance, to implement a log file of a parallel program), and is inappropriate for exploiting parallelism

in typical I/O-intensive applications. Moreover, the current version of ROMIO does not support shared file pointers on PVFS. Regarding nonblocking I/O primitives, it is difficult to quantify the global performance because it depends on the computation that can be performed concurrently with the I/O operation in a particular application.

In conclusion, we only focused on blocking I/O routines (both collective and noncollective) that use individual file pointers. We also considered in our experiments two file access patterns commonly found in parallel applications: contiguous or block access and interleaved or cyclic access.

4.3 NFS Performance

Fig. 2 shows some experimental results for the MPI-I/O routines obtained by using NFS and fixing $p = 8$ in the first graph and $n = 16$MB in the second one (some graphs presented in this paper use a log scale on the Y axis to improve readability). We experimentally observed that all the primitives under evaluation (except for interleaved access with collective primitives) did not scale using NFS, in the sense that the read/write latencies did not decrease as the number of processors increased. Latencies were even worse when more processors were employed; see, for instance, interleaved access using the noncollective routines shown in the second graph of Fig. 2, where the latency of the interleaved write shown exceeds the limit of the graph from $p = 4$. Thus, it was not worth modeling the routines that did not scale to the number of processors. We obtained the following models for interleaved access using collective I/O primitives: $T_{read\text{-}all} = (1.7045/p)n$ *s*, $T_{write\text{-}all} = (2.0048/\log_2 p)n$ *s*.

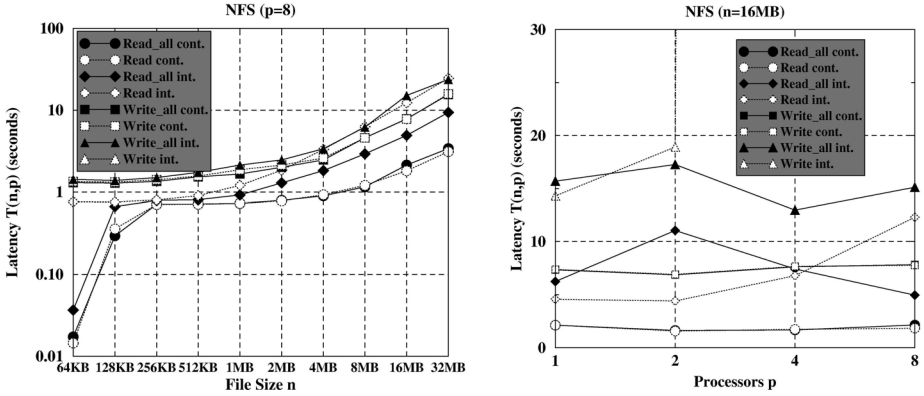

Fig. 2. Measured MPI-I/O latencies on NFS for different file sizes (left) and number of processors (right) (cont.: contiguous access; int.: interleaved access).

Regarding contiguous access, there is not much difference between collective and noncollective primitives. As expected, the collective optimizations described in section 2.2 only affected interleaved access.

4.4 PVFS Performance

In the PVFS tests, 8 nodes in the cluster were configured as I/O servers, and one node was dedicated exclusively as a metadata server. Unlike the NFS results, all the primitives analyzed speeded up contiguous I/O using PVFS. After curve fitting, we obtained the following results: $T_{read} = (0.1524/p)n$ *s*, $T_{read\text{-}all} = (0.1570/p)n$ *s*, $T_{write} = (0.0790/\log_2 p)n$ *s*, $T_{write\text{-}all} = (0.0773/\log_2 p)n$ *s*. As in case of NFS, for a contiguous access, the collective and noncollective routines exhibited practically the same behavior.

Regarding interleaved access, only collective primitives speeded up I/O (from $p = 2$), and they had the same complexity as their contiguous counterparts: $T_{read\text{-}all} = (1.3222/p)n$ *s*, $T_{write\text{-}all} = (0.7327/\log_2 p)n$ *s*. Although collective I/O was optimized for noncontiguous accesses, note that the constant of the models increased by approximately one order of magnitude with respect to contiguous access.

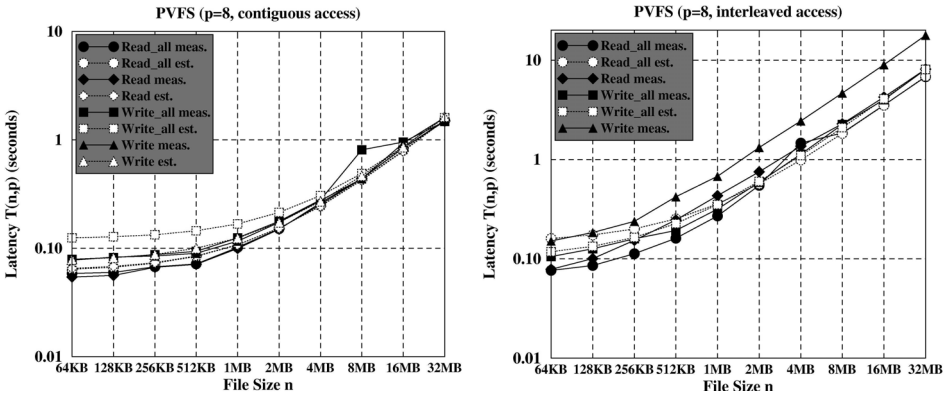

Fig. 3. Measured (meas.) and estimated (est.) MPI-I/O latencies on PVFS for different file sizes, using contiguous access (left) and interleaved access (right).

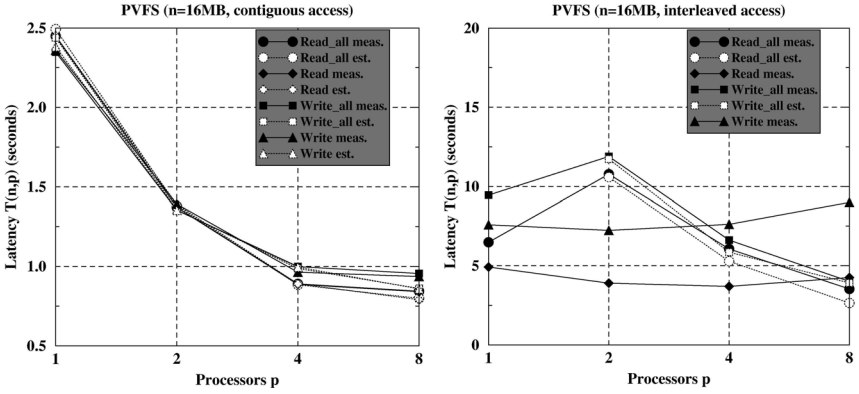

Fig. 4. Measured (meas.) and estimated (est.) MPI-I/O latencies on PVFS for different number of processors, using contiguous access (left) and interleaved access (right).

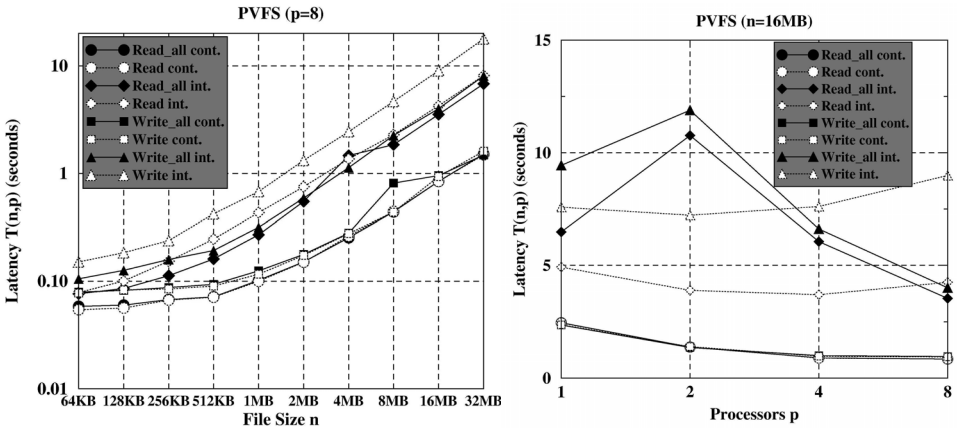

Fig. 5. Measured MPI-I/O latencies on PVFS for different file sizes (left) and number of processors (right) (cont.: contiguous access, int.: interleaved access).

Fig. 3 depicts measured and estimated (where applicable) latencies for $p = 8$, using contiguous access in the first graph and interleaved access in the second one. The same results are presented in Fig. 4, for different numbers of processors and a file size of 16MB. In order to compare contiguous vs interleaved latencies, the two graphs of Fig. 5 show the measured results of both kinds of accesses, for $p = 8$ and $n = 16$MB, respectively. It can be observed from the latter graph that, although the noncollective routines did not scale for an interleaved access, their latencies were lower than those of the corresponding collective primitives for a small number of processors (typically, 2 or 4). However, as p increased, latencies improved through the use of collective primitives (this also happened with NFS; see the interleaved read shown in the second graph of Fig. 2). It seems that the overhead of collective optimization in an interleaved access is greater than the benefit of the own optimization for a small value of p.
Finally, we found that performance tended to degrade for $p > 12$, due to the overlap of I/O servers and clients, which shared the same nodes, as the number of clients increased.

4.5 Putting it All Together

Table 1 summarizes the complexity of the models of each primitive, for each file system and access pattern. The empty entries correspond to the routines that do not scale. Performance was better for the modeled read routines, $O(1/p)$, than for the write routines, $O(1/\log_2 p)$ although the difference was more pronounced in NFS than in PVFS.

Table 1. Model complexity of MPI-I/O routines.

MPI-I/O Routine	NFS		PVFS	
	Contiguous	Interleaved	Contiguous	Interleaved
MPI_File_read	____	____	O(1/p)	____
MPI_File_write	____	____	$O(1/\log_2 p)$	____
MPI_File_read_all	____	O(1/p)	O(1/p)	O(1/p)
MPI_File_write_all	____	$O(1/\log_2 p)$	$O(1/\log_2 p)$	$O(1/\log_2 p)$

Fig. 6 compares the measured read performance using NFS and PVFS, for $p = 8$ in the first graph and $n = 16$MB in the second one. The same results are presented for the write operation in Fig. 7 (some noncollective interleaved write results in NFS do not appear because they exceed the limit of the graph). PVFS clearly outperformed NFS although the improvement was greater for write than for read; see, for instance, the latency curves of NFS/PVFS contiguous access for read (right graph of Fig. 6) and write (right graph of Fig. 7). Improvement can also be easily observed for collective interleaved access by comparing the corresponding models for read/write under NFS and PVFS.

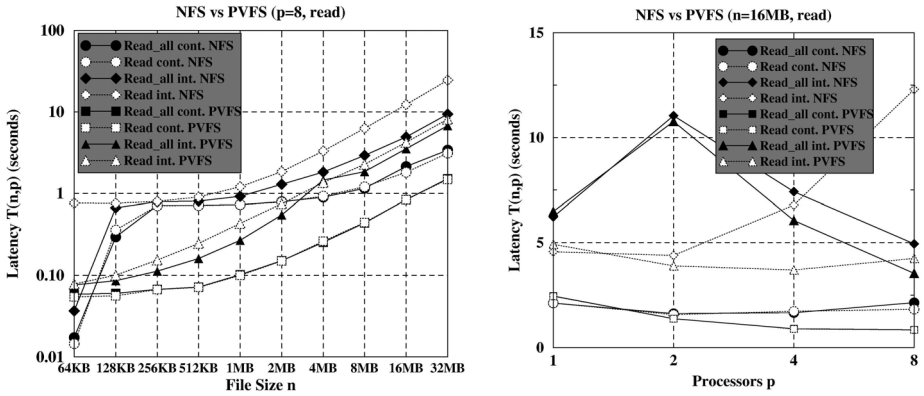

Fig. 6. Measured MPI-I/O read latencies on NFS and PVFS for different file sizes (left) and number of processors (right) (cont.: contiguous access, int.: interleaved access).

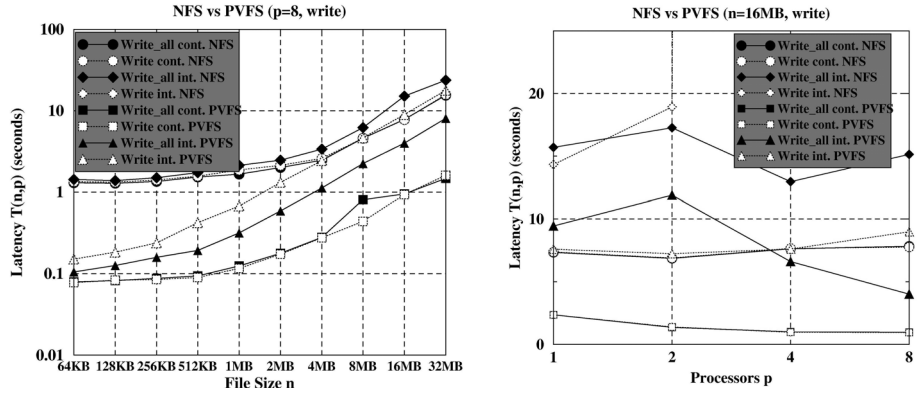

Fig. 7. Measured MPI-I/O write latencies on NFS and PVFS for different file sizes (left) and number of processors (right) (cont.: contiguous access, int.: interleaved access).

5. CONCLUSIONS

Characterization of the I/O overhead is very important for the development of I/O-intensive parallel codes. In this work, we have presented a comprehensive study of basic MPI-I/O primitives on a PC cluster based on the NFS and PVFS file systems. The results reported here can help application developers tune the file system configuration and select the best I/O routine in order to improve I/O performance.

I/O primitives can be more accurately modeled by defining different functions for different file size intervals. Nevertheless, for the sake of generalization, we found it more interesting to show the global functions that have been experimentally proved to have reasonable accuracy. They also provide a clearer overview of the I/O subsystem behavior.

In general, ROMIO MPI-I/O routines do not scale using NFS. It is clear that NFS was not designed for parallel I/O. We found that, in many cases, it was better to use POSIX *read/fread* and *write/fwrite* routines directly (which have an easier interface, and are widely known to programmers) to achieve even better performance than could be achieved using the corresponding MPI-I/O primitives. A file system specifically designed for parallel I/O (such as PVFS) is, therefore, necessary to speed up MPI-I/O primitives, as we have experimentally shown. Although ROMIO is optimized for noncontiguous accesses using collective primitives, the overhead of these optimizations should be reduced.

Network bandwidth is another key parameter in parallel I/O. We have found that our Fast Ethernet network limits I/O performance from a certain number of processors. Thus, faster networks (e.g. Myrinet, Gigabit Ethernet, SCI), should be considered in large cluster configurations when it is critical to achieve good parallel I/O performances for big files.

REFERENCES

1. M. Arenaz, R. Doallo, J. Touriño, and C. Vazquez, “Efficient parallel numerical solver for the elastohydrodynamic Reynolds-Hertz problem,” *Parallel Computing*, Vol. 27, 2001, 1743-1765.
2. E. J. Padron, M. Amor, J. Touriño, and R. Doallo, “Hierarchical radiosity on multicomputers: a load-balanced approach,” in *Proceedings of 10th SIAM Conference on Parallel Processing for Scientific Computing*, 2001.
3. D. E. Singh, M. Arenaz, F. F. Rivera, J. D. Bruguera, J. Touriño, R. Doallo, M. R. Mendez, J. A. Souto, and J. J. Casares, “Some proposals about the vector and parallel implementations of STEM-II,” C. Ibarra-Berastegi, C. A. Brebbia, and P. Zannetti eds., *Development and Application of Computer Techniques to Environmental Studies VIII*, WIT Press, Southampton, 2000, pp. 57-66.
4. Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, 1997; http://www.mpi-forum.org.
5. D. Lancaster, C. Addison, and T. Oliver, “A parallel I/O test suite,” in *Proceedings of 5th European PVM/MPI Users’ Group Meeting, EuroPVM/MPI’98*, LNCS 1497, 1998, pp. 36-43; benchmark suite available at http://www.ecs.soton.ac.uk/~djl/

MPI-IO/mpi-io.html.
6. Pallas GmbH, *Pallas MPI Benchmarks - PMB, Part MPI-2*, 2000; version 2.2 available at http://www.pallas.de.
7. J. Mache, J. Bower-Cooley, R. Broadhurst, J. Cranfill, and C. Kirkman IV, "Parallel I/O performance of PC clusters," in *Proceedings of 10th SIAM Conference on Parallel Processing for Scientific Computing*, 2001.
8. H. Taki and G. Utard, "MPI-IO on a parallel file system for cluster of workstations," in *Proceedings of 1st IEEE International Workshop on Cluster Computing*, 1999, pp. 150-157.
9. R. Thakur, E. Lusk, and W. Gropp, *User's Guide for ROMIO: a High-Performance, Portable MPI-IO Implementation*, 1998; http://www.mcs.anl.gov/romio.
10. Sun Microsystems, *NFS: Network File System v3 Protocol Specification*, 1993.
11. P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: a parallel file system for Linux clusters," in *Proceedings of 4th Annual Linux Showcase and Conference*, 2000, pp. 317-327; http://www.parl.clemson.edu/pvfs.
12. R. Thakur, E. Lusk, and W. Gropp, "Data sieving and collective I/O in ROMIO," in *Proceedings of 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999, pp. 182-189.
13. J. Ilroy, C. Randriamaro, and G. Utard, "Improving MPI-I/O performance on PVFS," in *Proceedings of 7th International Euro-Par Conference*, LNCS 2150, 2001, pp. 911-915.
14. M. Barreiro and V. M. Gulias, "Cluster setup and its administration," R. Buyya ed., *High Performance Cluster Computing, Architectures and Systems*, Prentice-Hall, 1999, pp. 48-67.
15. J. Touriño and R. Doallo, "Characterization of message-passing overhead on the AP3000 multicomputer," in *Proceedings of 30th International Conference on Parallel Processing, ICPP'01*, 2001, pp. 321-328.
16. W. Gropp and E. Lusk, *User's Guide for MPICH, a Portable Implementation of MPI*, 2000; http://www.mcs.anl.gov/mpi/mpich.

Jacobo Barro received the B.S. degree in Computer Science from the University of A Coruña, Spain, in 2001. He is currently a senior software engineer in Softgal S.A., a software firm in A Coruña. His main research area is cluster computing.

Juan Touriño is an associate professor of Computer Engineering at the University of A Coruña, where he earned the B.S., M.S. and Ph.D. degrees in Computer Science. His major research interests include performance evaluation of supercomputers, parallel algorithms and applications, parallelizing compilers and cluster/grid computing. He is a member of the ACM and IEEE Computer Society.

Ramon Doallo is a professor of Computer Engineering in the Department of Electronics and Systems at the University of A Coruña. He received the B.S., M.S. and Ph.D. degrees in Physics from the University of Santiago de Compostela, Spain. He has

extensively published in the areas of computer architecture and parallel and distributed computing. He is a member of the IEEE.

Victor M. Gulias received the B.S., M.S. and Ph.D. degrees in Computer Science from the University of A Coruña, Spain. He has been a lecturer in the Department of Computer Science of this university since 1994. His current research interests are cluster computing and novel techniques for the development of concurrent and distributed applications, such as distributed functional programming and design patterns for distributed systems.