2008 14th IEEE International Conference on Parallel and Distributed Systems

AHPIOS: An MPI-based ad-hoc parallel I/O system

Florin Isailă, Javier Garcia Blas, Jesus Carretero - *University Carlos III of Madrid*
{florin, fjblas, jcarrete}@arcos.inf.uc3m.es

Wei-keng Liao, Alok Choudhary - *Northwestern University*
{wkliao, choudhar}@ece.northwestern.edu

Abstract

This paper presents the design and implementation of a portable ad-hoc parallel I/O system (AHPIOS). AHPIOS virtualizes on-demand available distributed storage resources and allows the files to be striped over several storage devices. Additionally, the design unifies the configuration of the MPI-IO library and the AHPIOS data servers. By a strong integration of the application, MPI-IO library and file system, a significant performance improvement can be achieved. The experimental section shows that the full MPI-IO integrated AHPIOS implementation of file access operations outperforms the existing MPI-IO implementation by as much as 495% for file writes and 522% for file reads.

1 Introduction

In the last years the computing power of high-performance systems has continued to increase at an exponential rate, making even more challenging the access to large data sets. The ever increasing gap between I/O subsystems and processor speeds has driven researchers to look for scalable I/O solutions, including parallel file systems and libraries.

A typical parallel file system stripes the file data and metadata over several independent disks managed by I/O nodes in order to allow parallel file access from several compute nodes. Examples of popular file systems include GPFS [21], PVFS [15] and Lustre [8]. These parallel file systems manage the storage of several clusters and supercomputers from top 500 list [7].

The design complexity of distributed parallel file systems makes difficult to address the various requirements of different classes of applications at file system level. In other words, the inclusion of these optimizations inside the file systems would come at the cost of additional complexity. These optimizations addressing specific application requirements can be done at a higher level in an intermediate layer, whose stability is not critical for the system behavior.

The access to parallel file systems is typically done either through POSIX file interface or MPI-IO [17]. While ongoing efforts try to adapt POSIX to high performance requirements, MPI-IO has imposed as a portable and high performance interface for parallel applications. MPI-IO standard and its distributions (such as the popular ROMIO [27]) offer various file access optimizations (e.g. collective I/O, buffering, caching) on top of existing file systems.

A loose integration between file system and MPI-IO library supposes the existence of specific mechanisms and policies at each layer. In order to achieve an optimal performance an user should search through a high configuration parameter space and try to tune both file system and library.

In this paper we propose AHPIOS, a system that tightly integrates the MPI-IO distribution with an on-demand configurable parallel I/O system and unifies the configuration of both layers. Additionally, AHPIOS offers a high performance I/O system by virtualizing distributed storage and allowing a file to be striped over several storage devices, in the same way as in a parallel file system. This makes AHPIOS an alternative choice to a parallel file system.

The rest of the paper is structured as follows. Section 2 overviews related work. The system architecture and implementation is described in Section 3. The experimental results are presented in section 4. Finally, we summarize in section 5.

2 Related work

GPFS [21] is a parallel file system based on a shared-disk architecture: the disks are shared by all the nodes in the cluster/supercomputer. The Lustre [8] project aims at providing a file system for clusters of tens of thousands of nodes with petabytes of storage capacity. PVFS [15] is an open source parallel file system that targets the efficient access to large data sets. AHPIOS can be used alternatively to these parallel file systems. Additionally, can be launched in any distributed system which allows running an MPI ap-

1521-9097/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPADS.2008.50

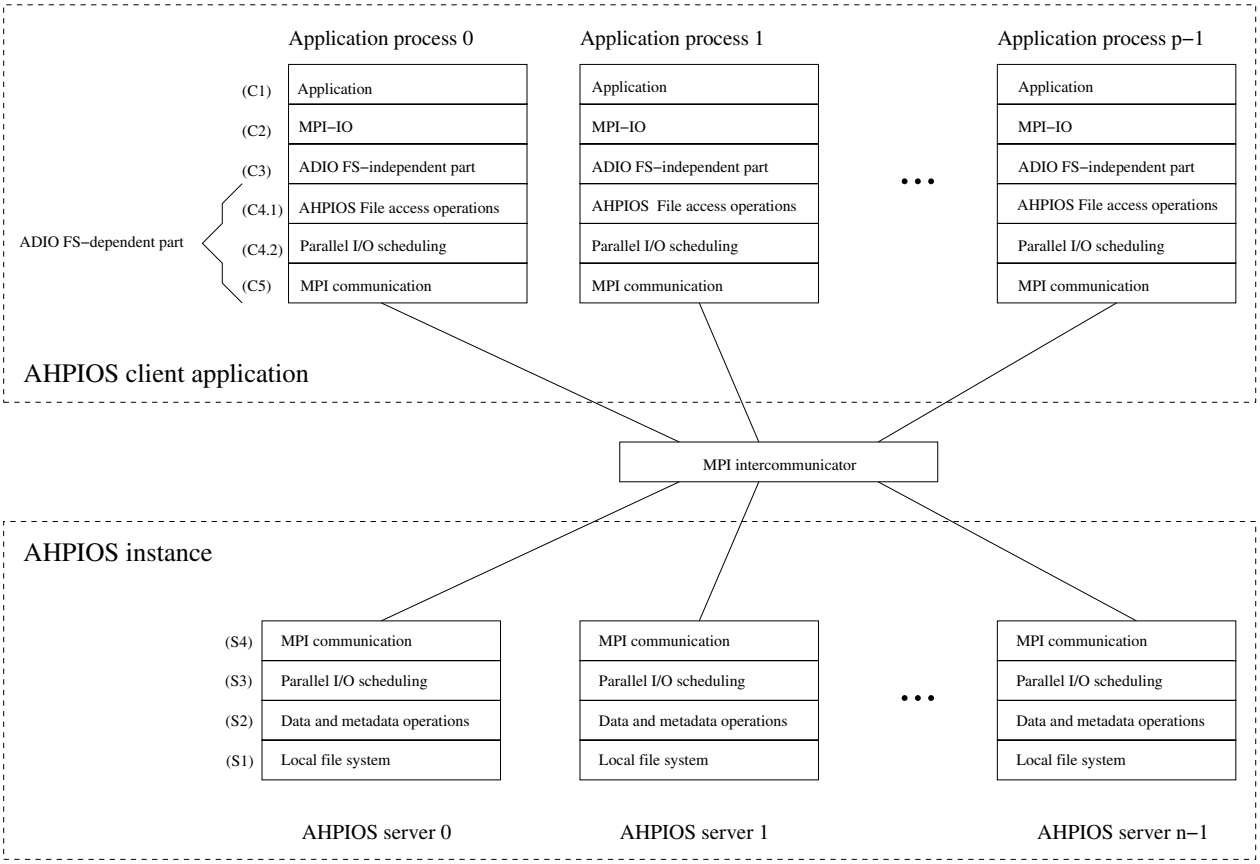

Figure 1. AHPIOS architecture with one application and one AHPIOS instance.

plication, allowing to virtualize on-demand cluster or grid storage resources.

Collective I/O techniques merge small individual requests from compute nodes into larger global requests in order to optimize the network and disk performance. Depending on the place where the request merging occurs, one can identify two collective I/O methods. If the requests are merged at the I/O nodes the method is called *disk-directed I/O* [14, 22]. If the merging occurs at intermediary nodes or at compute nodes, the method is called *two-phase I/O* [5, 2]. Data shipping [20] is a collective optimization that uniquely binds each file block in a round-robin manner to a unique I/O agent. All subsequent read and write operation on the file go through the I/O agents, which ship the requested data between the file system and the appropriate processes.

Several researchers have contributed with optimizations of MPI-IO data operations: data sieving [26], noncontiguous access [28], collective caching [12], cooperating write-behind buffering [13], integrated collective I/O and cooperative caching [9]. Packing and sending MPI data types has been presented in [3]. In our previous work [11] we have implemented the view I/O technique in Clusterfile [10] parallel file system, which uses a data representation equivalent to the MPI data types.

3 AHPIOS design and implementation

AHPIOS system manages several dynamic partitions of a parallel file system. The partitions are independent from each other, two different applications can access them in a concurrent manner.

Each partition is managed by a set of data servers, which run together as an MPI program called an *AHPIOS instance*, as it can be contemplated in the lower part of Figure 1. A partition can be created on-demand by any MPI application, which transparently reads the configuration from a user file and dynamically spawns the AHPIOS servers. Alternatively, an MPI application can mount an existing partition, either by re-launching an AHPIOS instance or by connecting to a running one. An AHPIOS instance virtualizes distributed storage resources and it is equivalent with a par-

allel file system installation. For example, a file may be striped over several independent storage resources.

The clients are unmodified MPI applications, which access the parallel file systems through an MPI-IO interface. One client is shown in the upper part of Figure 1. Each client is connected to an AHPIOS instance through an MPI intercommunicator. An intercommunicator is an MPI mechanism that allows two independent MPI applications to connect to each other and communicate. After connecting, all application processes can communicate individually or collectively with all server processes.

The design and implementation of the AHPIOS client is based on the ROMIO software architecture. Figure 2 shows the software architecture of ROMIO on five tiers: (C1) application layer, (C2) MPI-IO layer, (C3) ADIO file system - independent layer, (C4) ADIO file system-specific layer and (C5) file system library. The MPI-IO calls of the applications are translated in the C2 layer (MPI-IO) into a smaller subset of ADIO calls. The C3 layer, contains implementations of mechanisms and optimizations such as views, noncontiguous file access and collective I/O. The C4 layer maps an even smaller set of file access functions on particular file systems. This layer has to be implemented in order to add ROMIO support for a new parallel file system. Finally, the C5 layer consists of the file system access routines (it can be a user level library or the locally mounted file system). AHPIOS client was integrated into the ROMIO architecture stack by implementing the C4 and C5 layers, as it can be seen in the upper part of Figure 1. The C4 layer can be divided into two sublayers. The upper sublayer maps the ADIO file operations onto tasks to be performed by the individual AHPIOS servers. These can be metadata-related, such as creating or deleting a file or data operations. In the lower sublayer, these tasks are scheduled for transfer by a parallel I/O scheduling module. The C5 layer is responsible for communication with the AHPIOS servers through MPI communication routines.

As shown in the lower part of Figure 1, the server design is structured in four sublayers. The communication with the client application is performed through MPI routines in the S4 sublayer. The S3 sublayer is responsible for the parallel I/O scheduling policy, enforced in cooperation with the corresponding modules of the client side. The data and metadata management is performed in the S2 sublayer. Finally, the S1 layer transfers the data and metadata to the final storage.

3.1 Data access

The data access is performed through the cooperation of the client library running on several compute nodes and the AHPIOS servers. An AHPIOS file may be striped over several AHPIOS servers. An AHPIOS server has the following

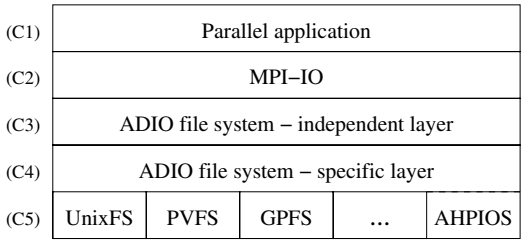

Figure 2. ROMIO software architecture

data-related duties (in subsection 3.2 we describe also the metadata operations):

- data communication with the clients
- cooperation of the parallel I/O scheduling module with the parallel I/O scheduling module of other I/O clients in order to optimize the global transfer performance
- data sieving and view optimizations
- management of a local cache; a local LRU replacement policy is employed for this cache
- end-storage access

3.1.1 Views

The view mechanism is implemented inside AHPIOS parallel I/O system. For each compute node declaring a view, the view file type, provided by the application as an MPI data type is decoded by the file system client. This is achieved by a recursive top-down traversal of the date type tree, which reconstructs the steps employed by the original application for data type creation. This data type structure is serialized and transferred to all AHPIOS servers, over which the file is striped. Each of these AHPIOS servers unserializes and reconstructs the original data type.

Subsequently, at file access time, the view data is directly mapped to the end-storage blocks at the AHPIOS servers. In ROMIO, this mapping is decomposed into two components: view-file mapping inside MPI-IO library and file-end storage mapping in the file system.

Figure 3 shows an example of a compute node that has declared a view by using an MPI vector data type with 4 blocks of 1 byte each and stride 4. The file is striped over 2 AHPIOS servers with a stripe of 8 (only the first two file blocks are shown in the figure, one at each I/O server). The left hand side depicts the view declaration operation, when the data type is sent to all AHPIOS servers and stored there. The right hand side illustrates the direct transfer of the contiguous data view to the AHPIOS servers. The contiguous data of the view is split into two packets at the file block

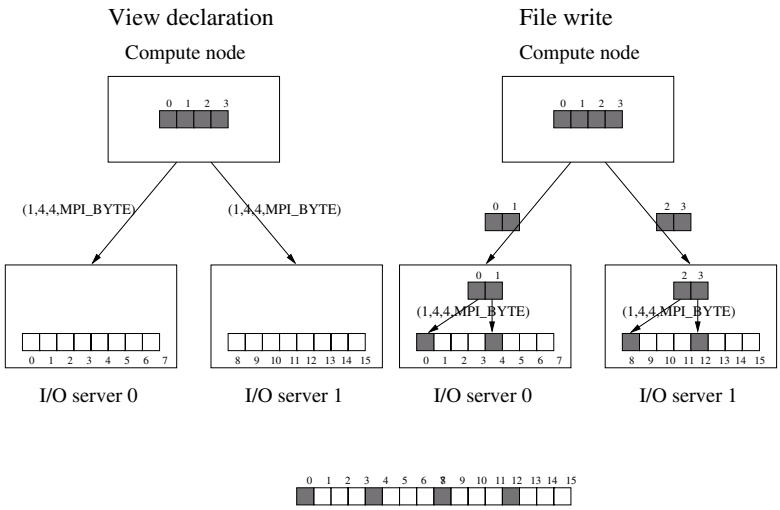

Figure 3. View I/O example in AHPIOS.

boundary. Each packet it is send to the corresponding AHPIOS server, where it is scattered by using the previously stored data type. The bottom part shows the file after the transfer.

The AHPIOS servers use the stored view data types for unpacking and transferring the data to/from the local storage repositories. This approach is different from the one employed in ROMIO. The ROMIO view is implemented in the ADIO file system-independent layer, which performs the mapping to the linear file space at the compute node. Data lying non-contiguous in file has to be re-mapped on a contiguous network buffer for efficient remote transfer to the file system servers.

3.1.2 Collective and independent I/O operations

As discussed in subsection 2, the MPI-IO standard defines two major groups of file access operations: *collective* and *independent*. The collective operations are suitable for the parallel workloads because of four basic characteristics, shown to be common in the data-intensive parallel scientific applications [19, 24, 23, 4, 29, 6]. First, it is frequent that all compute nodes perform access to the same file. Second, each individual compute node accesses the file non-contiguously and with small granularities. Third, there is a high degree of spatial locality: when a node accesses some file regions, the other nodes tend to access neighboring data. Fourth, the write accesses are mostly non-overlapping.

ROMIO distribution contains an implementation of two-phase I/O, for which the collective buffers reside at the aggregators (which are a subset of the compute nodes). AHPIOS collective I/O operations are related with disk-directed [14] and server-directed [22] collective I/O techniques (described in Section 2).

The AHPIOS implementation does not make any explicit distinction between independent and collective I/O operations, but it relies on the spatial locality property of the collective I/O calls. For write operations, when receiving a request from a compute node, the AHPIOS server checks to see if it has already cached the *collective buffer* (a collective buffer is a buffer that stores data on behalf of several compute nodes). If yes, the data is scattered by using the view data type. If not, the buffer is first read from the storage (for an existing file) or acquired from a pre-allocated pool (for a new file region) and then the scatter operation is performed. For read operations, the collective buffer is read from the file system at the first request of a compute node. Subsequent accesses find the data in the cache.

There are four main differences between the implementations of AHPIOS file access operations and ROMIO collective two-phase I/O. First, the view data types of all nodes involved in a collective access are sent at view declaration to all AHPIOS servers and can be reused until the view is changed. In ROMIO the pairs offset-length corresponding to the access pattern have to be transferred at each access. Second, no shuffle is necessary at compute node, because the mapping between application view and file layout is performed at the AHPIOS server. Third, if the AHPIOS servers have locally-attached storage, only one network transfer is needed, while in ROMIO two-phase I/O at least two transfers are performed. Fourth, in ROMIO, the collective buffers are not reused across different collective

operations. Consequently, workloads that show temporal locality cannot take advantage of the data already cached in them. In contrast, in AHPIOS, the collective buffers are stored in the local cache of the AHPIOS servers and can be reused across different collective I/O operations (if they have not been evicted by the replacement policy in the meantime).

Data consistency is enforced by caching the data blocks only once at the AHPIOS servers. This approach is similar to the one of PVFS. For concurrency control of *independent* access operations the user should employ external locking mechanisms. In the case of *collective* operations, the participant processes cooperate in order to access the file. For the collective write operation, it is the user responsibility to assure that the processes do not write overlappingly the same file (for instance by declaring non-overlapping views).

3.2 Metadata management

In AHPIOS there is no server that performs global metadata management. The global metadata is minimal, and contains only information about the particular instances of the file system. This information is stored in a shared registry. Each instance accesses atomically the registry in order to read or modify it. It is improbable that this access could represent a bottleneck in a large system, because it is accessed only when an instance is created, shut down or restarted. All these operations are infrequent.

The global registry stores the initial static configuration parameters of the AHPIOS instance. A rebooting instance of AHPIOS retrieves the original static configuration parameters from the global registry and uses them for restarting the file system created by a previous AHPIOS instance. The dynamic parameters are newly indicated by the user. We discuss more static and dynamic instance parameters in the subsection 3.3.

One of the servers of each AHPIOS instance plays also the role of an instance-local metadata manager. This server performs in the present implementation two functions: manages a local name space, stores and retrieves the file metadata. The global name of a file is given by appending the local path of a file to the global unique instance name. The local name space can be as simple as a directory in the name space of the local file system on the node where the AHPIOS metadata server is running.

For each file, additional metadata information is stored such as: the number of I/O servers over which the file is stored, the path on each local file system of each AHPIOS server, where the file data is permanently stored, and the stripe size.

3.3 AHPIOS configuration

One of the main goal of AHPIOS was avoiding that the specific system parameters are the default parameters of a cluster-wide installation. There may be several instances of AHPIOS running at the same time on the same cluster. Each of these instances may use a different block size, network buffer size, AHPIOS server set, metadata manager, etc. More than that, a particular AHPIOS instance can be shut down and restarted with a different set of dynamic parameters, which would provide better performance results.

An AHPIOS instance may be configured at the start of an application. The user may define the file system parameters in a configuration file, such as the one from below. The static parameters include the default file stripe size and the storage resources over which the files are striped. In the example below file data will be striped over nodes n1, n2, n3, n4 in local directories, while the instance metadata will be stored in the directory /metadata of node n0. The dynamic parameters include the network buffer size, the cache size of the servers and the optimizations to be used (we show here only the generic ROMIO and AHPIOS optimizations). The static parameters could be changed by creating a new instance and then moving the data from old instance to the new instance. The dynamic parameters can be changed when an instance restarts.

```
#AHPIOS configuration file: filesystem.cfg
######### Static parameters
# The default stripe size of the filesystem
stripe_size = 64k
# Number of IOS
nr_ios = 4
# Storage resources of AHPIOS servers
ahpios_server = n0:/data
ahpios_server = n1:/data
ahpios_server = n2:/data
ahpios_server = n3:/data
# Path of the metadata directory
metadata = n0:/metadata

######### Dynamic parameters
# Scheduling block size
net_buf_size = 64k
#Buffer cache size of AHPIOS server
cache_size = 1G
# AHPIOS or ROMIO optimizations
# 0: AHPIOS 1:ROMIO
optimizations = 0
```

4 Experimental results

The evaluation presented in this paper was performed on the dual-core "Lonestar" system at TACC [16] in the frame-

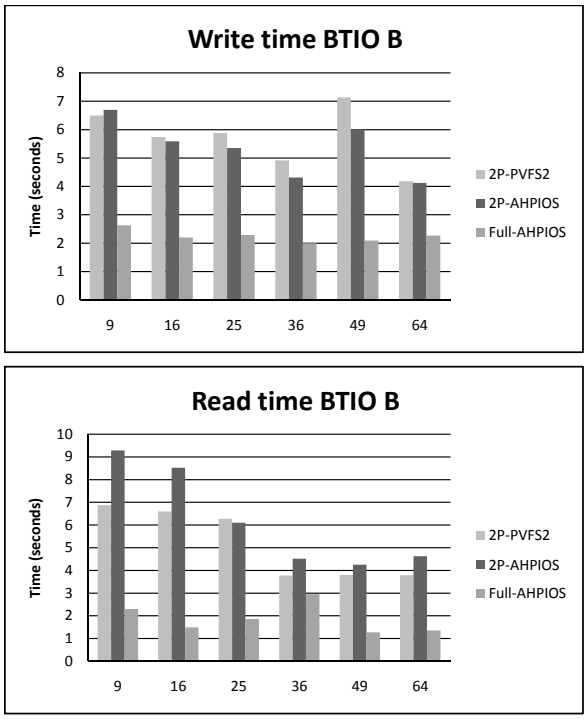

Figure 4. BTIO measurements (class B).

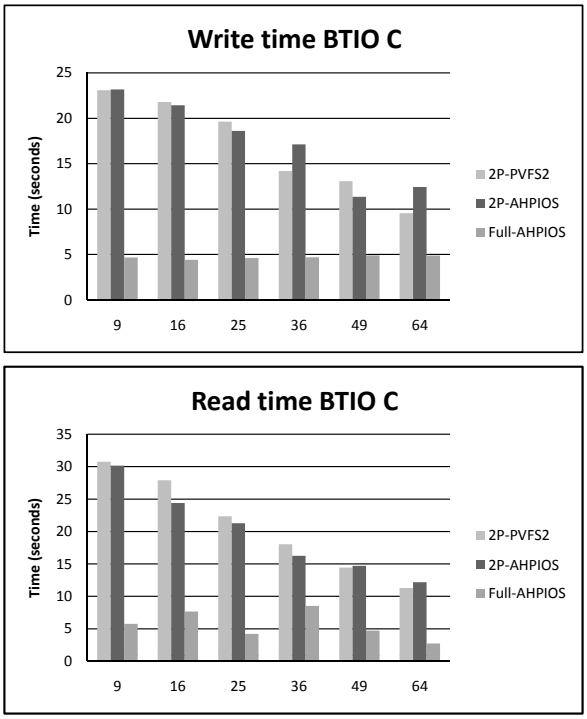

Figure 5. BTIO measurements (class C).

work of the Teragrid project [25]. A node consists of a Dell PowerEdge 1955 blade running a 2.6 x86_64 Linux kernel from kernel.org. Each node contains two Xeon Intel Duo-Core 64-bit processors (4 cores in all) on a single board, as an SMP unit. The Core frequency is 2.66GHz and supports 4 floating-point operations per clock period with a peak performance of 10.6 GFLOPS/core or 42.6GFLOPS/node. Each node contains 8GB of memory. The interconnect is Infiniband. The employed MPI library is MPICH2 [1] version 1.0.5, with the communication running over TCP/IP sockets. The Lonestar Storage includes a 73GB SATA drive (60GB usable by user) on each node (the I/O servers of AHPIOS and PVFS2 used this storage).

In the experiments we have used AHPIOS and PVFS2, which we launched through the batch system, before the application is started. Both AHPIOS and PVFS2 used 8 I/O nodes and a file block of 64KBytes. For both systems, the I/O servers and application processes are running on disjoint nodes. The communication inside the file systems and in the MPICH2 was done with TCP/IP sockets over Infiniband.

The largest number of nodes used by the applications for the computation was 64 (an additional number of 8 nodes were used for I/O servers by PVFS2 and AHPIOS). Due to the large wait times in the batch system queues of Lonestar, we could not perform measurements on a larger number of nodes.

In all benchmarks we compared three solutions: ROMIO two-phase I/O over PVFS2 (2P-PVFS2), ROMIO two-phase I/O over AHPIOS, and an integrated AHPIOS-based solution (full-AHPIOS). In the first two solutions, the view and the collective I/O optimizations are implemented in the file system-independent part of ADIO, while the file system is accessed only for contiguous accesses. In the integrated solution, the view and collective I/O operations are performed in cooperation by the application processes and AHPIOS servers. Our goal is to demonstrate that, by the tight integration between application and library offered by the full-AHPIOS solution, a significant better performance can be obtained.

4.1 BTIO benchmark

NASA's BTIO benchmark [29] solves the Block-Tridiagonal (BT) problem, which employs a complex domain decomposition across a square number of compute nodes. Each compute node is responsible for multiple Cartesian subsets of the entire data set. The execution alternates computation and I/O phases. Initially, all compute nodes collectively open a file and declare views on the relevant file regions (a subcube in the Cartesian domain). After each five computing steps the compute nodes write the solution to a file through a collective operation. At the end, the resulting file is collectively read and the solution ver-

ified for correctness. In this paper we report the results for the MPI implementation of the benchmark, which uses MPI-IO's collective I/O routines. On all runs we set the benchmark to execute 25 compute steps, which correspond to 5 I/O steps (5 collective writes followed by 5 collective reads). The access pattern of BTIO is nested-strided with a nesting depth of 2.

Figures 4 and 5 show the results for the classes B and C, respectively. The upper row shows the total time (in seconds) spent in writing the file, while the lower row the total time spent in reading.

First we note that the two implementations based on two-phase I/O (2P-PVFS2 and 2P-AHPIOS) show similar results.

In all cases the AHPIOS-based reads and writes significantly outperform 2P-PVFS2 and 2P-AHPIOS. When comparing the AHPIOS accesses with the *best* performing implementation, the improvement ranged for class B writes from 1.81 times (64 processes) to 2.87 times (32 processes), for class B reads from 1.28 times (36 processes) to 4.42 times (16 processes), for class C writes from 1.95 times (64 processes) to 4.95 times (16 processes) and for class C reads from 1.9 times (36 processes)to 5.22 times (9 processes).

There are additional reasons, which explain the results. In two-phase I/O the data is in general transferred twice over the fabric: once for the data aggregation at the compute nodes and the second time when accessing the file system. In Full-AHPIOS the aggregation is done at the AHPIOS server, i.e. close to the storage. If the storage is locally available, the second communication operation is spared.

Additionally, the view I/O technique significantly reduces the size of the metadata sent over network. First, the MPI data types are sent compact to the AHPIOS servers at view declaration. Second, this data type transfer is done only once and it can be reused by subsequent operations. In contrast, the lists of offset-length pairs are sent to the aggregators at each two-phase I/O operation.

Finally, in AHPIOS, the collective buffers are cached at the AHPIOS servers across collective I/O operations. Subsequent read operations find them in the cache.

4.2 MPI Tile I/O benchmark

MPI Tile I/O benchmark [18] evaluates the performance of underlying MPI-IO library and file-system implementation under a non-contiguous access workload. The benchmark logically divides a data file into a dense two-dimensional set of tiles. The number of tiles along rows (nr_x) and columns (nr_y) and the size of each tile in the x and y dimensions(sz_x and sz_y) are specified as input parameters. We have chosen this values such that for any number of processors the total amount of data to be accessed is 1 GByte and the access granularity is 4 KBytes.

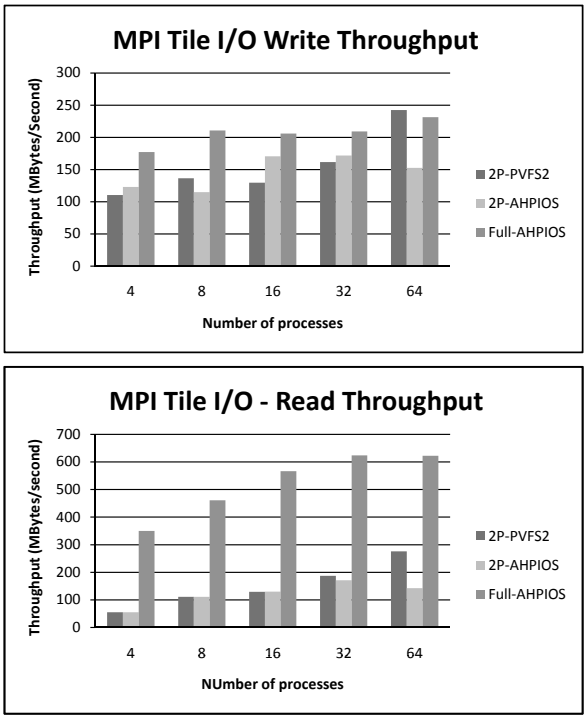

Figure 6. MPI Tile I/O throughput.

The results are plotted in Figure 6. Full-AHPIOS outperforms the other solutions in nine of the ten cases. From these nine cases the write improvement of Full-AHPIOS over the best of 2P-AHPIOS and 2P-PVFS2 ranges between 1.2 times for 16 processes and 1.54 times for 8 processes. Full-AHPIOS shows remarkable results for reading the file. The improvement is between 2.25 for 64 processes to 6.3 for 4 processes. When 64 processes write the file, the 2P-PVFS outperforms Full-AHPIOS by 5%.

5 Conclusion and future work

In this paper we have presented AHPIOS parallel I/O system, a dynamic parallel I/O system, that virtualizes on-demand independent storage resources. AHPIOS can take advantage of its tight integration with MPI-IO, and use the application access patterns in order to increase the performance. AHPIOS employs efficient non-contiguous I/O and collective I/O techniques, which can be used alternatively to the ones in ROMIO. The experimental results show that, even in an incipient development phase, AHPIOS outperforms solutions in which the parallel file systems and the application optimizations are separated, using various access patterns common to parallel scientific applications. The main advantage of AHPIOS solution comes from the tight integration between the application and the end storage.

Acknowledgments

This work was supported in part by Spanish Ministry of Science and Innovation under the project TIN 2007/6309 and by DOE SCIDAC-2: Scientific Data Management Center for Enabling Technologies (CET) grant DE-FC02-07ER25808, DOE SCiDAC award number DE-FC02-01ER25485, NSF HECURA CCF-0621443, NSF SDCIOCI-0724599, and NSF ST-HEC CCF-0444405.

References

[1] *MPICH website*. http://www-unix.mcs.anl.gov/mpi/mpich/.

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

[3] A. Ching, A. Choudhary, W. K. Liao, R. Ross, and W. Gropp. Efficient Structured Data Access in Parallel File Systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, December 2003.

[4] P. Crandall, R. Aydt, A. Chien, and D. Reed. Input/Output Characteristics of Scalable Parallel Applications. In *Proceedings of Supercomputing '95*, 1995.

[5] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proc. of IPPS Workshop on Input/Output in Parallel Computer Systems*, 1993.

[6] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modelling Astrophysical Thermonuclear Flashes. *Astrophysical Journal Suppliment*, pages 131–273, 2000.

[7] http://www.top500.org. *Top 500 list*.

[8] C. F. S. Inc. Lustre: A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0, November 2002. http://www.lustre.org/docs/whitepaper.pdf.

[9] F. Isaila, G. Malpohl, V. Olaru, G. Szeder, and W. Tichy. Integrating Collective I/O and Cooperative Caching into the "Clusterfile" Parallel File System. In *Proceedings of ACM International Conference on Supercomputing (ICS)*, pages 315–324. ACM Press, 2004.

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

[11] F. Isaila and W. Tichy. View I/O:improving the performance of non-contiguous I/O. In *Third IEEE International Conference on Cluster Computing*, pages 336–343, Dec. 2003.

[12] W. keng Liao, K. Coloma, A. Choudhary, L. Ward, E. Russel, and S. Tideman. Collective Caching: Application-Aware Client-Side File Caching. In *Proceedings of the 14th International Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[13] W. keng Liao, K. Coloma, A. N. Choudhary, and L. Ward. Cooperative Write-Behind Data Buffering for MPI I/O. In *PVM/MPI*, pages 102–109, 2005.

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

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

[16] Lonestar home page. *http://www.tacc.utexas.edu/services/-userguides/lonestar/*.

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

[18] MPI tile I/O. *http://www-unix.mcs.anl.gov/pio-benchmark/*.

[19] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File Access Characteristics of Parallel Scientific Workloads. In *IEEE Transactions on Parallel and Distributed Systems, 7(10)*, pages 1075–1089, Oct. 1996.

[20] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 17–17, New York, NY, USA, 2001. ACM Press.

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

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

[23] H. Simitici and D. Reed. A Comparison of Logical and Physical Parallel I/O Patterns. In *International Journal of High Performance Computing Applications, special issue (I/O in Parallel Applications), 12(3)*, pages 364–380, 1998.

[24] E. Smirni and D. Reed. Workload Characterization of I/O Intensive Parallel Applications. In *Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, June 1997.

[25] Teragrid home page. *http://www.teragrid.org/*.

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

[27] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems*, pages 23–32, May 1999.

[28] R. Thakur, W. Gropp, and E. Lusk. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, Jan. 2002.

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