A Scalable Message Passing Interface Implementation of an Ad-Hoc Parallel I/o system

Florin Isaila, Francisco Javier Garcia Blas, Jesús Carretero, Wei-keng Liao and Alok Choudhary

International Journal of High Performance Computing Applications 2010 24: 164 originally published online 5 October 2009
DOI: 10.1177/1094342009347890

A SCALABLE MESSAGE PASSING INTERFACE IMPLEMENTATION OF AN AD-HOC PARALLEL I/O SYSTEM

Florin Isaila[1]
Francisco Javier Garcia Blas[1]
Jesús Carretero[1]
Wei-keng Liao[2]
Alok Choudhary[2]

Abstract

In this paper we present the novel design, implementation, and evaluation of an ad-hoc parallel I/O system (AHPIOS). AHPIOS is the first scalable parallel I/O system completely implemented in the Message Passing Interface (MPI). The MPI implementation brings the advantages of portability, scalability and high performance. AHPIOS allows MPI applications to dynamically manage and scale distributed partitions in a convenient way. The configuration of both the MPI-IO and the storage management system is unified and allows for a tight integration of the optimizations of these layers. AHPIOS partitions are elastic: they conveniently scale up and down with the number of resources. We develop two collective I/O strategies, which leverage a two-tiered cooperative cache in order to exploit the spatial locality of data-intensive parallel applications. The file access latency is hidden from the applications through an asynchronous data staging strategy. The two-tiered cooperative cache scales with both the number of processors and storage resources. Our experimental section demonstrates that, with various optimizations, integrated AHPIOS offers a substantial performance benefit over the traditional MPI-IO solutions on both PVFS or Lustre parallel file systems.

Key words: parallel I/O, parallel systems, distributed file systems, parallelism and concurrency

1 Introduction

The ever increasing gap between the performance of processors and persistent storage devices has focused the attention of several research projects on providing scalable, high-performance storage solutions. In addition, the scale of modern supercomputers has recently augmented to performance surpassing the PetaFLOP milestone. A recent survey of High-Performance Computing (HPC) open science applications (Geist, 2008) has shown that the vast majority of applications are implemented in the Message Passing Interface (MPI), which has become the *de facto* standard of scalable parallel programming. Despite increasing interest in alternative parallel programming paradigms, there are at least two factors that make the MPI indispensable: the limited scalability of shared memory machines (hence the need for the message passing paradigm on distributed memory machines), and the legacy of a huge amount of robust scientific libraries developed in the MPI.

A large subset of parallel scientific applications is data intensive. Parallel file systems represent traditional scalable high-performance solutions to the storage bottleneck problem. A typical parallel file system stripes files across several independent I/O servers in order to allow parallel file access from many compute nodes simultaneously. Examples of popular file systems include General Parallel File System (GPFS) (Schmuck and Haskin, 2002), Parallel Virtual File System (PVFS) (Ligon and Ross, 1999) and Lustre (Cluster File Systems Inc., 2002). These parallel file systems manage the storage of the vast majority of clusters and supercomputers from the Top500 list (Top 500 list, 2009). They are also used by small and medium clusters of computers, typically for parallel data processing and visualizations. These applications could benefit from a simple transparent solution offering scalable parallel I/O with practically no cost for installing and maintaining a parallel file system.

Many parallel applications access the final storage through parallel I/O libraries, including the MPI-IO (Message Passing Interface Forum, 1997), Hierarchical Data Format (HDF) (HDF5 home page, 2009), and parallel NetCDF (Li et al., 2003). Both HDF and parallel NetCDF are implemented on top of the MPI-IO, as the MPI-IO handles the low-level file access to the file system. Being part of the MPI-2 standard, the MPI-IO defines a set of application programming interfaces (API) for file access in parallel. It also specifies the rules for data consistency.

The International Journal of High Performance Computing Applications, Volume 24, No. 2, Summer 2010, pp. 164–184
DOI: 10.1177/1094342009347890

Figure 9 appears in color online: http://hpc.sagepub.com

[1]UNIVERSITY CARLOS III OF MADRID, AVDA DE LA UNIVERSIDAD 30, LEGANES 28911, SPAIN. (FLORIN@ARCOS.INF.UC3M.ES)

[2]NORTHWESTERN UNIVERSITY, EVANSTON, IL, USA.

Recently, there has been growing interest in High-Productivity Computer Systems (High Productivity Computer Systems, 2008). Besides the performance, the productivity is taken into consideration, which includes the costs of programming, debugging, testing, optimization, and administration. In the case of parallel I/O, it becomes more difficult to obtain optimal performance from the underlying parallel file system, given the different I/O requirements. The default file system configuration cannot always provide an optimal throughput for different data intensive applications. File system reconfiguration may be a solution, but an expensive one, which would inevitably involve administrative overheads, data relocation, and system down time. In addition, the design complexity of a distributed parallel file system, such as GPFS, makes it difficult to address the requirements from different I/O patterns at the file system level. In other words, implementing the solutions for these problems in the file systems would come at the cost of additional complexity. However, the solutions addressing specific I/O requirements can be done at a higher level, closer to the applications.

This paper presents AHPIOS, a light-weight ad-hoc parallel I/O system that addresses some of the issues mentioned above. AHPIOS can be used as a middleware located between the MPI-IO and distributed storage resources, providing high-performance, scalable access to files. AHPIOS can be used as a light-weight low-cost alternative to any parallel file system. The main goals of AHPIOS design are the following:

High performance. High performance is achieved through the tight integration of the MPI-IO and the storage system, which allows an efficient data access through a two-tiered cooperative cache and an asynchronous data staging strategy.

Scalability. The AHPIOS partitions are elastic: they scale up and down with the number of storage resources. In addition, the system performance scales with both the capacity of memory and storage. The first cooperative cache tier runs along with the application processes and hence scales with the number of application processes. The second cooperative cache tier runs at the I/O servers and, therefore, scales with the numbers of global storage devices.

Portability. The portability is achieved through the complete implementation in the MPI. To the best of our knowledge, this is the first implementation of a parallel I/O system in the MPI.

Simplicity. AHPIOS is simple to use and no modification to the existing applications are needed. The setup for AHPIOS is user-configurable through a plain text file.

In our earlier work (Isaila et al., 2008) we have presented the initial design and evaluation of AHPIOS. This paper presents the system evolution and differs from the initial work in the following aspects:

- The novel design includes a hierarchical two-tiered cooperative cache. In the initial design the data was cached locally only at the AHPIOS servers.
- AHPIOS partitions are elastic: they can be scaled up and down by a simple remount.
- The first cooperative cache level is leveraged by a new client-directed collective I/O implementation.
- The new design includes a transparent asynchronous data staging strategy.
- We add the evaluation of scalability of independent I/O operations.
- A performance comparison with Lustre file system is included.
- The client-directed and server-directed collective I/O implementations are compared with other three collective I/O solutions.
- We discuss and evaluate the usage of the MPI in designing a system.
- Further applications of AHPIOS, including initial experiences and potential utilizations in Blue Gene systems and clouds.

The remainder of the paper is structured as follows. Section 2 presents a system overview. The background necessary for understanding of the system is given in Section 3. Section 4 overviews related work. The system design and implementation are described in Section 5. The experimental results are presented in Section 6. In Section 7 we discuss initial experiences and envision potential applications of our system on Blue Gene systems and clouds. Finally, we conclude in Section 8.

2 System Overview

Given a MPI application accessing files through the MPI-IO interface and a set of distributed storage resources, AHPIOS constructs, on demand, a distributed partition, which can be accessed transparently and efficiently. On each AHPIOS partition the users can create a directory name space in the same way as on any regular file system. Files stored on one AHPIOS partition are transparently striped over storage resources as in any parallel file system. Each partition is managed by a set of storage servers, running together as an independent MPI application. The access to an AHPIOS partition is performed through an MPI-IO interface. A partition can be built and scaled up and down on demand during the application run-time.

The system manages a hierarchy of cooperative caches as depicted in Figure 1. Firstly, client applications cache

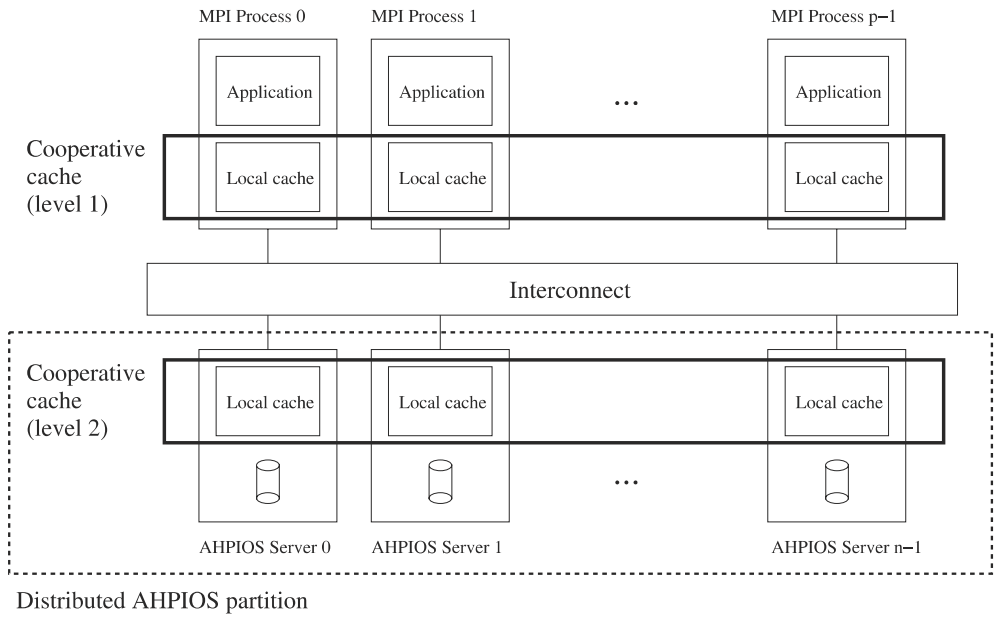

Fig. 1 AHPIOS overview.

collective buffers on the compute nodes. The collective buffers are used by the MPI-IO layer for reordering and gathering I/O requests in order to improve the I/O performance. Secondly, the AHPIOS servers also perform data caching by collectively managing a single-copy cache. The communication within and between these layers is performed through standard MPI communication operations.

A data staging strategy hides the latency of transferring data blocks between the levels of the cache hierarchy. The data transfer between the client cache and AHPIOS server caches, and between AHPIOS server caches and disk storage, is done asynchronously. Therefore, overlapping of computation, communication and I/O is achieved.

The MPI-IO mechanisms and optimizations, such as the file view setting and collective I/O, are strongly integrated into the AHPIOS storage system. The MPI views may be set either at the client or the server, and the MPI collective buffering, the mechanism behind two-phase I/O, can be activated either at the client (close to computing) or at the server (close to storage).

AHPIOS is started by parsing a configuration file, which defines the parameters, such as the number of storage resources, file stripe size, the type of collective I/O to be used, the sizes of the client and server caches, the type of parallel I/O scheduling policy, etc. The user can control both the MPI-IO and the parallel I/O system through the configuration file.

Multiple AHPIOS partitions can coexist, each being managed by different sets of AHPIOS servers with different configurations.

3 Background

ROMIO (Thakur et al., 1999) is the most wide-spread implementation of the MPI-IO standard and is developed at Argonne National Laboratory. ROMIO has been incorporated in MPICH (MPI Forum, 1995), LAM (LAM website, 2009), HP-MPI (HP MPI home page, 2009), NEC-MPI (NEC MPI home page, 2009), and SGI-MPI (SGI MPI home page, 2009) distributions.

The design and implementation of the AHPIOS client is based on the ROMIO software architecture. ROMIO is implemented on top of an abstract device interface called ADIO (Thakur and Lusk, 1996). Figure 2 shows the soft-

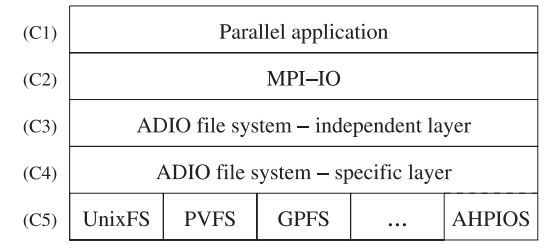

Fig. 2 ROMIO software architecture.

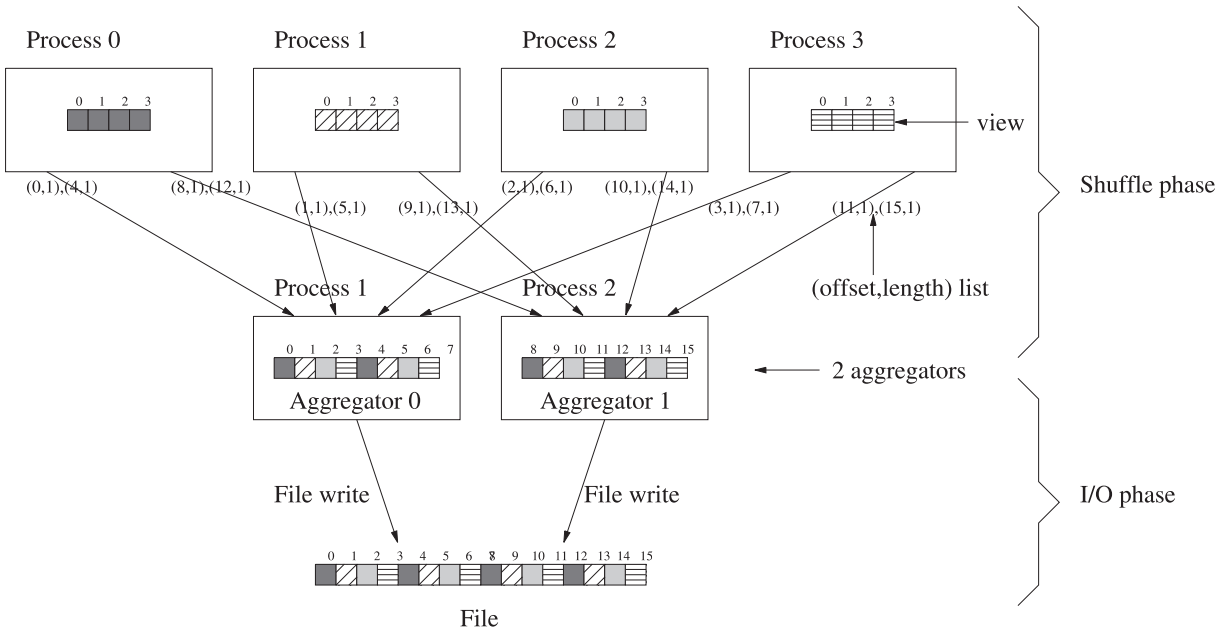

Fig. 3 Two phase I/O write example.

ware architecture of ROMIO on five tiers: (C1) the application layer; (C2) the MPI-IO layer; (C3) the ADIO file system-independent layer; (C4) the ADIO file system-specific layer; and (C5) the file system library. The MPI-IO calls made by 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, non-contiguous file access and the 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. The AHPIOS client was integrated into the ROMIO architecture stack by implementing the C4 and C5 layers.

One of the most interesting MPI-IO file operations is the declaration of a file view. A *file view* defines the current set of data visible and accessible from an open file by a MPI process. Each process defines its own file view. Defining file views offers several advantages: non-contiguous file ranges can be "seen" to be logical as a contiguous range, facilitating the programmer's task and allowing non-contiguous I/O optimizations. At the same time, a process's file view hints at the future access pattern and can be used for optimizing the access. The view is mapped to the linear file space by the MPI-IO and, in turn, the linear file space is mapped on disk blocks by the file system. The two mappings are explicitly performed, even when the view maps are contiguous on a disk. Examples of using file views are given in Figure 3. Four MPI processes shown in the upper part have declared views on the file depicted in the lower part of the figure. Process 0 "sees" only the dark gray bytes of the file, process 1 only the hashed, and so on. In this example, the bytes 0, 1, 2, 3 in process 0 map onto bytes 0, 4, 8, 12 of the file.

In the MPI-IO data is moved between files and process memories by issuing read and write calls. The MPI-IO functions are divided into two categories: independent and collective. Collective I/O functions merge small individual requests from individual processes into larger global contiguous requests in order to better utilize the network and disk performance. Depending on the place where the request merging occurs, there are two well-known implementations for collective I/O. *Disk-directed I/O* (Kotz, 1994; Seamons et al., 1995) merges the requests at the I/O servers, while *two-phase I/O* (del Rosario et al., 1993; Bordawekar, 1997) merges the requests at the compute nodes. AHPIOS incorporates both strategies based on views.

Two-phase I/O implementation in ROMIO performs in two steps, as illustrated in Figure 3, namely the *shuffle phase* and the *I/O access phase*. In the shuffle phase, the data is gathered in contiguous chunks at a subset of MPI processes called *aggregators*. In this example we have two aggregators (processes 1 and 2). The number of aggrega-

tors can be customized by the user through a MPI hint. In the first part of the shuffle phase, the file interval between offsets 0 and 15 is split among the two aggregators into (0,7) and (8,15). Then, the view is mapped to a list of (file offset, file length) tuples of corresponding intervals; e.g. process 0 maps the (0,1) view data onto (0,1), (4,1). These lists correspond to the mapping between each view and the file. Subsequently, the lists are sent from all processes to the aggregators. For instance, process 0 sends (0,1), (4,1) to aggregator 0 and (8,1), (12,1) to aggregator 1. Finally, the view data is transferred to the aggregators and is scattered into contiguous chunks by using the offset length-lists; for example, process 0 sends contiguously bytes 0 and 1 from this view, which corresponds to file offsets 0 and 4. In the access phase single, larger contiguous chunks are transferred from the aggregators to the file system. Combining non-contiguous small requests into fewer large contiguous ones significantly improves the performance of collective I/O.

The ROMIO two-phase I/O needs two network transfers: one corresponds to the shuffle phase, the other to the file access from the aggregators to the I/O servers. As we will show in Section 5.3, AHPIOS needs only one network transfer, when server-directed I/O is used and the AHPIOS server runs on a node with local storage.

4 Related Work

GPFS (Schmuck and Haskin, 2002), PVFS (Ligon and Ross, 1999) and Lustre (Cluster File Systems Inc., 2002) are parallel file systems that are currently installed and used in many production supercomputers. GPFS is based on a virtual shared-disk architecture: logical disks are shared by all the nodes in the cluster/supercomputer. File system clients see the shared disks as if they were locally mounted. Read performance is boosted through client-side file caching and prefetching. Lustre (Cluster File Systems Inc., 2002) aims at providing a file system for clusters consisting of tens of thousands of nodes with petabytes of storage capacity. Lustre consists of the following components: one metadata target (MDT), which stores metadata, such as file names, directories, permissions, and file layout, and one or more object storage targets (OSTs), which store file data on one or more object storage servers (OSSs). Lustre is a POSIX compliant file system. Client-side file caching is enabled in Lustre. File joining (Yu et al., 2007) merges multiple files into one for improving collective I/O over Lustre. PVFS (Ligon and Ross, 1999) is an open source parallel file system that targets the efficient parallel access to large data sets. PVFS consists of several servers and a set of APIs for client processes to access the file system. Unix I/O APIs is also supported through the installation of a Linux kernel module implementing a mountable Virtual File Switch (VFS) interface. A server may be both a data and or a metadata manager. PVFS provides efficient non-contiguous I/O through its list I/O interface.

AHPIOS differs from these systems through its strong integration into the MPI-IO architecture and its dynamic and easily reconfigurable nature. AHPIOS offers several optimizations, such as views, collective I/O operations and transparent asynchronous file data transfers. AHPIOS can be used as a memory-based file system in the same fashion as Memfs (Hermanns, 2006). In this scenario AHPIOS could take advantage of its tight integration with the MPI-IO for better informed optimizations (such as view I/O), before flushing the data to the end file system, which can be any of GPFS, Lustre or PVFS.

pNFS (Hildebrand and Honeyman, 2005) is an extension of the network file system (NFS) protocol (in version 4.1) and provides parallel access to storage systems. pNFS is a continuation of efforts on parallelizing the file access by striping files over multiple NFS servers (Garcia-Carballeira et al., 2003; Kim et al., 1994; Lombard and Denneulin, 2002). pNFS is likely to become the *de facto* standard of high performance parallel storage access and has already been adopted by storage leader companies, such as Panasas and IBM. Panasas is migrating its PanFS parallel file system (PanFS web site, 2008) to pNFS. IBM is also implementing pNFS on top of GPFS. Like AHPIOS, pNFS provides a storage system independent of the operating system and allows client applications to fully utilize the throughput of a shared parallel file system. Unlike pNFS, AHPIOS implementation is fully done in the MPI. In addition, AHPIOS mainly targets MPI applications and enables a tight integration among MPI processes and storage servers, in particular for view-based and collective file accesses.

File systems, such as Sorrento (Tang et al., 2004) and RADOS (Weil et al., 2007) (RADOS is a part of Ceph scalable high-performance distributed file system (Weil et al., 2006)), offer scalable auto-reconfigurable storage solutions for dynamic pools of storage resources. The physical placement of logical data segments in these systems is hidden from the applications. In contrast, AHPIOS targets the optimal mapping of parallel applications' access patterns on the storage layout. However, optimizations implemented in AHPIOS, such as view-based collective and independent operations, can be used on top of these storage systems as well.

Several works have presented implementations and optimizations of the MPI-IO interface. The MPI-IO implementation for GPFS (Prost et al., 2001) contains an optimization called data shipping, which is a collective I/O technique resembling two-phase I/O. As in this paper, this work emphasizes the importance of efficiently data-to-file mapping in the MPI-IO layer. An evaluation of the MPI-IO on PVFS is presented in (Taki and Utard, 1999).

An ADIO implementation that allows the matching of views and the file physical layout is described in (Isaila et al., 2006). The MPI-IO implementation of the VIPIOS parallel I/O run-time system (Stockinger and Schikuta, 2000) maps MPI data types on the internal VIPIOS structures. VIPIOS uses data distribution in two layers: a problem layer analogous to the access pattern and file view, and a data layer, analogous to the physical file distribution. The layout is constructed automatically, which is an approach similar to that of the Panda parallel I/O library (Winslett et al., 1996). Several researchers have contributed with optimizations of the MPI-IO, such as data sieving (Thakur et al., 1999), non-contiguous access (Thakur et al., 2002), collective caching (Liao et al., 2005), and cooperating write-behind buffering (Liao et al., 2005), to name a few. Packing and sending derived data types systematically between clients and servers has been presented in (Ching et al., 2003).

In our earlier work (Isaila and Tichy, 2003) we implemented the view I/O technique in the Clusterfile (Isaila and Tichy, 2001) parallel file system, which uses a data representation equivalent to the MPI data types. Later we integrated collective I/O and cooperative caching (Isaila et al., 2004) in a prototype of Clusterfile. AHPIOS differs from our previous work on several aspects. First, AHPIOS is portable middleware, which can be used independently on top of storage resources or any parallel file system, including Clusterfile. Second, AHPIOS is a completely portable implementation in the MPI, with all the communication using MPI routines, and completely integrated in the MPI-IO implementation. Third, AHPIOS manages two levels of cooperative caches, both at the clients and the servers. Fourth, the data transfer between the client and server caching layers, and between the server-side caching layer and the end storage, is done asynchronously, hiding the file access latency and, therefore, overlapping computation, communication, and I/O.

5 AHPIOS Design and Implementation

As explained in Section 2, a MPI application accesses an AHPIOS partition through the MPI-IO interface. The application processes are linked with an *AHPIOS client*. An AHPIOS partition is managed by a set of *AHPIOS servers*, which are also processes of a MPI program, running independently from the MPI application. Figure 4 shows the software architecture of the AHPIOS system: a client application in the upper part and AHPIOS servers in the lower part.

The AHPIOS servers are interconnected through *a MPI intracommunicator*. A MPI intracommunicator is a MPI mechanism that allows the members of a group of processes to communicate among each other through MPI communication routines. The client processes also communicate among each other through an intracommunicator.

The servers communicate with AHPIOS clients through *a MPI intercommunicator*. A MPI intercommunicator is a MPI mechanism that allows the members of different process groups to communicate. Two different MPI applications communicate also through a MPI intercommunicator.

The client-side of AHPIOS is integrated into the ROMIO architecture stack by implementing the C4 and C5 layers, as seen in Figure 4. The C4 layer can be divided into two sublayers: C4.1 and C4.2. C4.1 maps the ADIO file operations onto I/O 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 C4.2, the I/O tasks are scheduled for transfer by a parallel I/O scheduling module (Isaila et al., 2006). The C5 layer is responsible for communication with the AHPIOS servers.

The AHPIOS servers run as a completely client-independent MPI program. As shown in Figure 4, the server design is structured into four sublayers. Communication with the client is performed through MPI routines in the S4 sublayer. The S3 sublayer is responsible for the parallel I/O scheduling policy, which cooperates with the corresponding modules at the client side. Data and metadata management is performed in the S2 sublayer. Finally, the S1 layer transfers data and metadata to the final storage system.

The partition creation attributes, such as the number of resources, the stripe size and the list of resources used for data and metadata storage, are specified in a configuration file, as in the example shown below:

```
# The default stripe size of the partition
stripe_size = 64k
# Number of IOS
nr_ios = 4
# Storage resources of AHPFS servers
ahpfs_server = n0:/data
ahpfs_server = n1:/data
ahpfs_server = n2:/data
ahpfs_server = n3:/data
# Path of the metadata directory
metadata = n0:/data
```

An AHPIOS partition is created by the first application that uses the partition. The storage servers are spawned through the MPI dynamic process mechanism and the partition is subsequently registered in the global registry identified by a port name. Subsequently, other applications can mount the partitions identified by the port name and employing the client/server functionality of the MPI2.

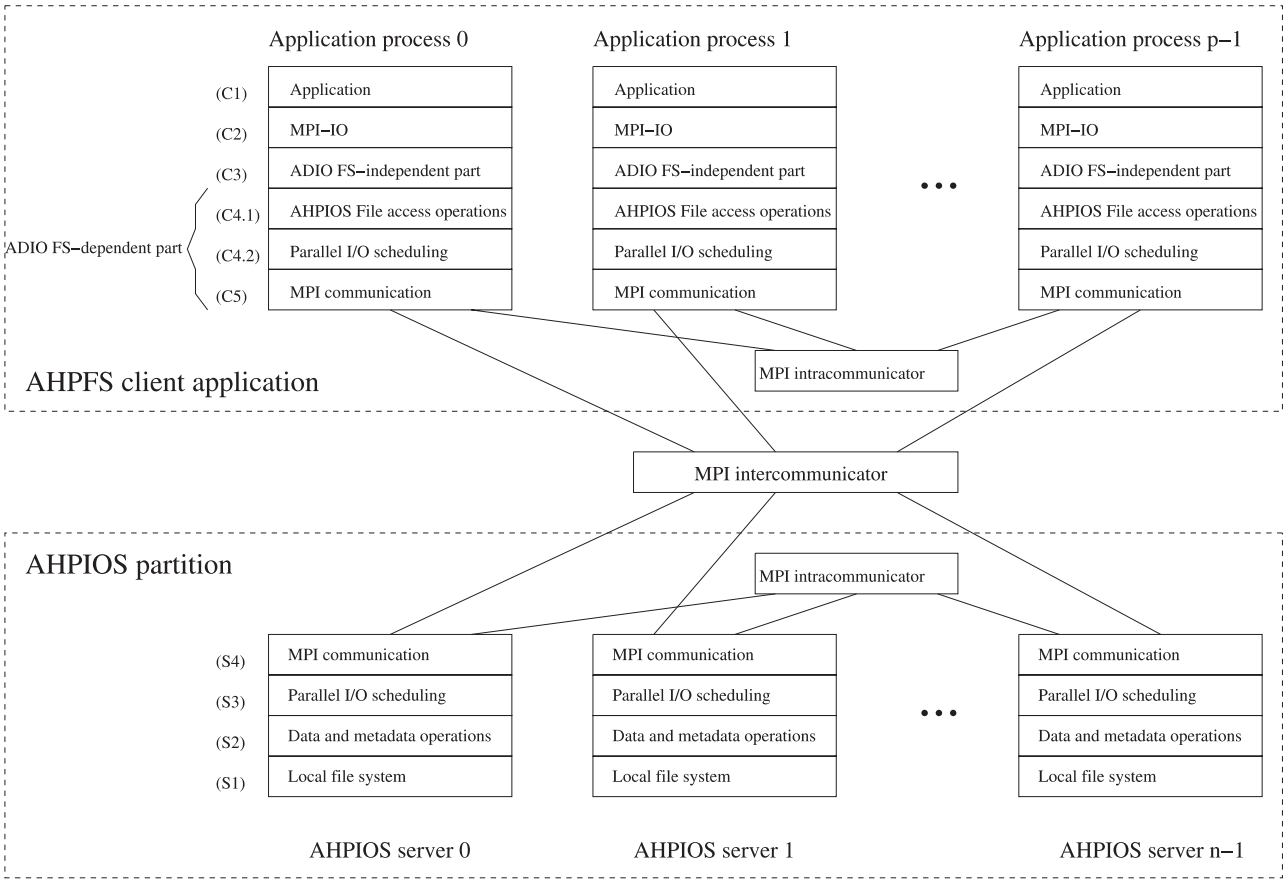

Fig. 4 AHPIOS architecture with one application and one AHPIOS partition.

Several AHPIOS partitions with different configurations may be running in parallel on a cluster of computers, after registering with the *global registry* (more details in Section 5.5). Figure 5 shows an example of two applications sharing two different AHPIOS partitions. For each mounted partition, a dedicated MPI intercommunicator is created, through which the MPI-IO client layer communicates with the AHPIOS servers.

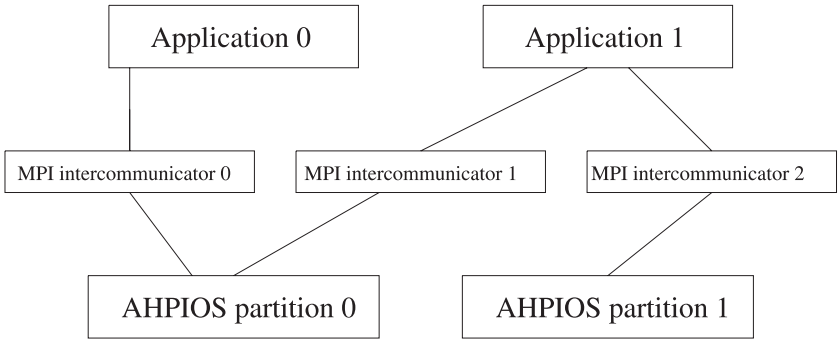

Fig. 5 AHPIOS partitions accessed by two MPI applications.

5.1 Elastic Partitions

AHPIOS partitions are elastic: they can scale up and down by increasing the number of storage resources. Scaling up involves only a system restart with a larger number of AHPIOS servers. In this case, for files that are stored on the old smaller set of storage resources, users can choose either to preserve their initial structure or to redistribute them over the newly available storage resources. Mounting a scaled down partition involves a redistribution of file data from the storage resources that become unavailable to the remaining ones. This is a two-step process. Firstly, all of the old storage resources are mounted by starting the system with the old number of AHPIOS servers and the redistribution is performed. Secondly, the old partition is unmounted and the new partition is mounted.

5.2 Cooperative Caching

Data access is performed through the cooperation of the clients running on several compute nodes and the AHPIOS servers. Data transfer order is controlled by a parallel I/O scheduling strategy, which is described in (Isaila et al., 2006).

An AHPIOS file may be striped over several AHPIOS servers. By default the files are striped over all of the available AHPIOS servers, but the user can control the striping parameters through MPI hints.

An AHPIOS partition is accessed through a two-level hierarchy of cooperative caches as described shortly in Section 2 and shown in Figure 1. The user can choose to disable the first level of the cooperative cache for consistency reasons, when multiple client applications share the same files. However, studies have shown that this is rarely the case with scientific applications (Smirni and Reed, 1997; Wang et al., 2004).

The first level of caching is managed through the cooperation of all application processes (using only a subset of processes is also possible). These processes put in common a fraction of the local memory as cache buffers. Thus, caching can scale with the number of application processes. By analogy with the two-phase collective I/O implementation of ROMIO, we call these nodes *aggregators*, because they also aggregate small pieces of files into larger cache pages. The maximum amount of memory dedicated to the aggregators' cache may depend on each applications and is user-configurable at run time.

The first cooperative caching level works in the following way. File blocks are mapped in a round-robin fashion over all aggregators. The file requests are directed accordingly to the responsible aggregator. The aggregator clusters together several requests before accessing the next level of caching. Communication with the second level of caching is performed asynchronously by an I/O thread, which hides the file access latency from the application. The I/O thread asynchronously writes file blocks to the file system following a high–low watermark policy, where the watermark is the number of dirty pages. When the high watermark is reached, the I/O thread is activated. The I/O thread flushes the last modified pages to the file system until the low watermark is reached. The local replacement policy of each aggregator is the least recently used (LRU).

The second level of cooperative caching is managed by the AHPIOS servers. File blocks are mapped to AHPIOS servers in a round-robin fashion and each server is responsible for transferring its blocks to and from the persistent storage. When an AHPIOS server receives a request for a block assigned to another server, it serves this request in cooperation with the other servers. This approach is useful at least in two scenarios. Firstly, the I/O related computations can be offloaded to the AHPIOS servers. Secondly, a group of application processes is assigned to an I/O server. The I/O server is responsible for all file requests from this group and can serve them in cooperation with other I/O servers, similar to the I/O system of the IBM Blue Gene supercomputers. Similarly to the aggregators, the I/O servers employ a high–low watermark policy for flushing the last recently modified dirty blocks to the disks and a LRU local replacement policy.

5.3 Data Access

As discussed in Section 3, the MPI-IO standard defines two major groups of file access operations: *collective* and *independent*. In AHPIOS, the independent operations are identical to the server-directed I/O operations, in the sense that the data is transferred between the MPI processes and the AHPIOS servers and the AHPIOS servers merge independent small requests into larger collective requests and cache the data in collective buffers. Therefore, the independent I/O operations can perform as efficiently as collective I/O operations.

Collective operations are suitable for parallel workloads because of four basic characteristics that are common in data-intensive parallel scientific applications (Nieuwejaar et al., 1996; Smirni and Reed, 1997; Simitici and Reed, 1998; Crandall et al., 1995; Wong and der Wijngaart, 2003; Fryxell et al., 2000). Firstly, in many cases, all processes of one parallel scientific application perform shared access to the same file. Secondly, each individual compute node accesses the file non-contiguously and with small granularities. Thirdly, there is a high degree of spatial locality: when a node accesses some file regions, the other nodes tend to access neighboring data. Fourthly, write accesses are mostly non-overlapping among processes.

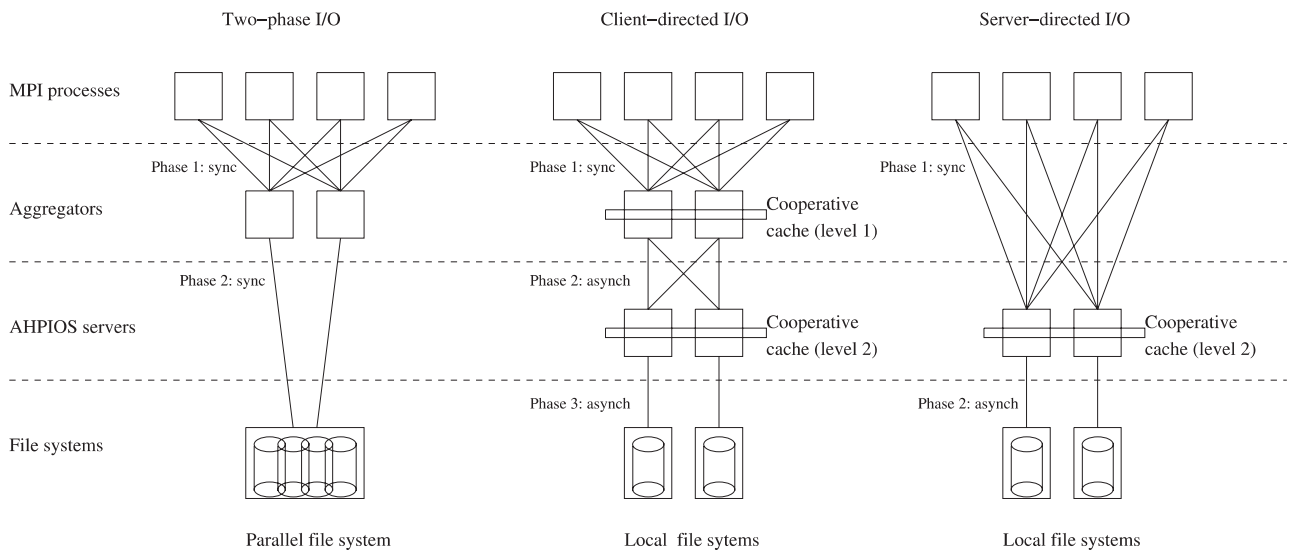

Fig. 6 Comparison of data flow in two-phase I/O, client-directed I/O, and server-directed I/O.

As described in Section 3, ROMIO adopts the two-phase I/O strategy, for which the collective buffers only reside at the aggregators. The novel design of AHPIOS includes two collective I/O methods, both of them based on views: *view-based client-directed* and *view-based server-directed*. Each of them has its own benefit, as we will demonstrate in the experimental section.

Figure 6 depicts two-phase I/O, client-directed I/O, and server-directed I/O. We assume that the view has already been declared. After the view declaration is made, the view description is kept at the client processes in two-phase I/O. For client-directed I/O the view is sent to the aggregators, where it is stored for future use. For server-directed I/O it is transferred to the AHPIOS servers, which also store it for future use.

Storing the view remotely has the potential of significantly reducing the overhead of transmitting non-contiguous file regions, which map contiguously to data locally available at the MPI processes. Firstly, this contiguous data do not have to be processed locally (scattered or gathered) and no offset-length lists have to be sent for each access (as in the case of two-phase I/O). Secondly, the view is sent only once in a compact form and can be reused when the same access pattern appears repeatedly (a frequent behavior of parallel applications).

For both two-phase I/O and client-directed I/O, the aggregators represent a subset of the MPI processes and are used for merging small requests into larger ones. By default, all MPI processes act as aggregators. However, the user may set the number of aggregators by a MPI hint.

Two-phase I/O consists of two synchronous phases, which correspond to the shuffle phase (1) and the I/O phase (2) described in Section 3.

Client-directed I/O consists of three phases. When client-directed I/O starts, the views have been already stored at the aggregators at the view declaration. The first phase (1) is synchronous and consists of shuffling the data between MPI processes and AHPIOS aggregators: small file regions are gathered at the aggregators for writing and are scattered from the aggregators for reading. The small file regions are transferred contiguously between each pair of MPI processes and aggregators and are scattered/gathered by using the view previously stored at the aggregators at the view declaration. In the second phase (2) data are asynchronously staged from the first-level cooperative cache of aggregators to the AHPIOS servers. Only full file blocks are transferred between these two cache levels. Finally, in the third phase (3), data are also asynchronously staged from the second-level cooperative cache of AHPIOS servers to the final storage.

Server-directed I/O consists of two phases. When server-directed I/O starts, the views have also been already stored at the AHPIOS servers. The first phase (1) is synchronous and similar to the client-directed I/O, except the fact that the data are shuffled between the MPI processes and the AHPIOS servers. The second phase (2) is asynchronous and is the same as the third phase in the client-directed I/O.

It can be noticed that there are two main differences between client-directed I/O and server-directed I/O: the

Table 1
Comparison of Three Collective I/O Methods.

Operation	Two-phase (TP)	Client-directed (CD)	Server-directed (SD)
View declaration	Store view at client	Send view to AHPIOS aggregator	Send view to AHPIOS server
File Access (metadata)	Generate file-offset lists from views and send them to TP aggregators	No action	No action
File access (at client)	Non-contiguous	Contiguous access	Contiguous access
File access (aggregator)	Non-contiguous	Non-contiguous	n/a
File access (file server)	Contiguous	Contiguous	Non-contiguous
File cache	No	In the first level cooperative cache	In the second level cooperative cache
Data staging from aggregators to servers/FS	Synchronous	Asynchronous	n/a
Data staging from servers to storage	n/a	Asynchronous	Asynchronous
File close	No action	Ensure AHPIOS aggregators and servers flush all data	Ensure AHPIOS servers flush all data

place where the view is stored and the intermediary level of caching for the client-directed I/O. Consequently, small requests are merged into larger ones at the aggregators for the client-directed I/O and at the AHPIOS servers for the server-directed I/O.

Table 1 compares two-phase I/O with client-directed and server-directed I/O. Unlike in two-phase I/O, for client-directed and server-directed I/O, the views, represented as MPI data types, are not stored at the client application, but decoded at the MPI-IO layer, then serialized and transferred either to AHPIOS aggregators or AHPIOS servers, respectively. Upon receiving the view, the aggregators or servers unserialize and reconstruct the original view data type. The advantage of this approach is that no metadata has to be sent over the network at access time, because the view representing the file access pattern is already stored remotely. For two-phase I/O the access pattern generated by the view must be sent as lists of (file offset, length) tuples. In the case of client-directed and server-directed I/O the data can be transferred contiguously between the client and the aggregator/server.

In two-phase I/O, data are not cached at the aggregators, but only temporarily buffered and synchronously transferred to the file system. In client-directed I/O, the file data may be cached at both levels of the cooperative caching hierarchy and they are transferred asynchronously to the final storage. In server-directed I/O, the data are cached in the second-level cooperative caching hierarchy and they are asynchronously sent to the storage. The asynchronous transfers allow a transparent overlapping of computing, I/O related communication and storage access.

5.4 Cache Coherency and File Consistency

According to the MPI standard, the MPI provides three levels of consistency: sequential consistency among all accesses using a single file handle; sequential consistency among all accesses using file handles created from a single collective open with atomic mode enabled; and user-imposed consistency among accesses other than the above.

AHPIOS implementation provides partial MPI consistency. The data are flushed to the end storage for both cache levels, either upon calling `MPI_FILE_SYNC` or closing the file. The atomic mode for independent I/O file accesses has not been implemented, i.e. sequential consistency is not guaranteed for concurrent, independent I/O with overlapped access regions. This approach is similar to the one taken in PVFS and it is motivated by the fact that overlapping accesses are not frequent for paral-

lel applications. Nevertheless, the atomic mode can be enforced as user-defined consistency semantics by using `MPI_FILE_SYNC` as described in the MPI standard.

Cache coherency is enforced at both cache levels by not allowing more than one copy of the data blocks at each level. This decision is motivated by the frequent access patterns of the parallel applications: individual processes write non-overlapping file regions and there is a high interprocess spatial locality. Data are transferred between cache levels always at block granularity. For writing to a file block less than its size in the first- or second-level cache, a read-modify-write operation is needed, where the final write operation can be performed asynchronously.

For client-directed I/O, the modifications of single-copy file blocks are always performed in the first-level cache by one aggregator. Server-directed I/O does not use the first-level cache; therefore, the coherency is enforced only at the servers by allowing one copy of a file block. A server-directed I/O operation triggers the eviction of the accessed file blocks from the first-level cache to the server, before performing its own operations.

5.5 Metadata Management

In AHPIOS there are two levels of metadata management: global metadata management of AHPIOS partitions and local metadata management of individual AHPIOS partitions.

5.5.1 Global Metadata Management In AHPIOS there is no server that performs global metadata management. The global metadata are minimal, and contain only information about the particular partitions of the file system. This information is stored in a file shared by all partitions and called the *registry*. Each set of AHPIOS servers managing a partition accesses atomically this file in order to read or modify it. This access should not cause a bottleneck in a large system, because the registry is accessed only when a partition is created, mounted or unmounted. All of these operations are infrequent.

The global registry stores structural and dynamic configuration parameters of the AHPIOS partition. The structural parameters are the partition name, the storage resources assigned to the AHPIOS servers, the number of AHPIOS servers, the default stripe size, and metadata file disk location. The dynamic parameters include the network buffer size, the parallel I/O scheduling policy, the buffer cache size of the AHPIOS server, etc. The dynamic parameters can be changed by the user each time a partition is mounted.

5.5.2 Local Metadata Management For each partition, one of the AHPIOS servers also plays the role of a partition-local metadata manager. This server manages a local name space and an inode list, stores and retrieves the file metadata and updates the metadata in coordination with the other servers. The global name of a file is given by appending the local path of a file to the global unique partition 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. An inode stores typical file metadata, including values for the stripe size and the number of stripes. By default, the number of stripes is the same as the number of AHPIOS servers and the user can modify this value through a hint.

6 Experimental Results

The evaluation presented in this paper was performed on the "Lonestar" parallel computer at the Texas Advanced Computing Center (TACC) (Lonestar home page, 2009), which is part of the Teragrid framework (Teragrid home page, 2009). A node consists of a Dell PowerEdge 1955 blade running a 2.6 x86_64 Linux kernel. Each node contains two Xeon Intel Duo-Core 64-bit processors on a single board. The Core frequency is 2.66 GHz and supports four floating-point operations per clock period with a peak performance of 10.6 GFLOPS/core or 42.6 GFLOPS/node. There is an 8 GB memory in each node. The interconnect is an Infiniband with a fat-tree topology. The employed MPI library is MPICH2 (MPI Forum, 1995) version 1.0.5, with the communication running over TCP/IP sockets. The Lonestar Storage includes a 73 GB SATA drive (60 GB usable by user) on each node (the I/O servers of AHPIOS and PVFS2 used this storage). The work file system, also accessible from all nodes, is a Lustre parallel file system with 68 TB of DataDirect Storage.

We compared AHPIOS with Lustre, the parallel file system installed on Lonestar, and with PVFS2, which we launched through the batch system, before the application is started. AHPIOS and PVFS2 used eight I/O nodes. For AHPIOS and PVFS2, the I/O servers and the application processes are running on disjoint nodes. Lustre is installed over 32 OSTs and stripes by default files over eight consecutive OSTs, chosen through an algorithm that combines the randomness with load–balance awareness. Lustre uses the buffer cache of the compute nodes up to a maximum of 6,912 MB per node. In addition, Lustre has an aggressive prefetching policy, which reads ahead up to 40 MB. AHPIOS employs a cooperative cache managed by the application processes for client-directed I/O. PVFS2 and AHPIOS, with server-directed I/O, do not cache data at the clients.

On Lonestar, the internal communication of Lustre is performed directly over the Infiniband network. The

PVFS2 setup could only run TCP/IP over Infiniband. The MPI communication employed by AHPIOS was performed with MPICH2 and not with the Infiniband MVPICH, and consequently, also over TCP/IP. Therefore, in all of the performed measurements, Lustre has an advantage over PVFS2 and AHPIOS, due to its considerably lower communication costs (TCP/IP sockets are known for high overheads).

6.1 Scalability

We evaluated the scalability of independent I/O operations on AHPIOS. The processes of a MPI program write and read in parallel disjoint contiguous regions of a file stored over an AHPIOS system for different numbers of AHPIOS servers. In this experiment the first level cooperative cache is disabled. We evaluate two scenarios: one in which the compute nodes also have local storage and the AHPIOS servers run on the same nodes as the MPI processes and another in which the MPI processes and AHPIOS servers run on disjoint sets of nodes.

Figure 7 shows the aggregate I/O throughput for n MPI processes writing and reading to/from an AHPIOS partition with n AHPIOS servers. The figure represents the throughput to the AHPIOS servers. We can see that the file access performance scales well with the partition size. This happens independently of the location of the AHPIOS servers, both when the AHPIOS servers run on the same nodes as the MPI application and on disjoint nodes.

In Figure 8 the number of MPI processes running on distinct nodes is $n = 64$ and the number of AHPIOS servers is varied between 1 to 64. We notice that the aggregate throughput of both write and read operations scales smoothly with the number of AHPIOS servers, as in the previous case, independently of whether the AHPIOS servers share or do not share the compute node with the MPI application.

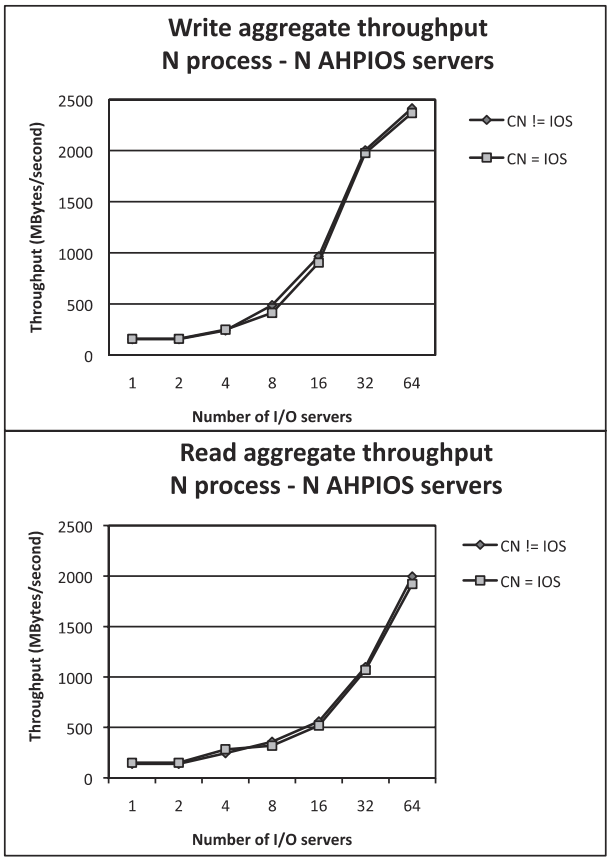

Fig. 7 File independent I/O for *N* processes writing to *N* AHPIOS servers.

6.2 Collective I/O Evaluation

In the following two benchmarks we compare five different solutions for parallel I/O access: ROMIO two-phase I/O over PVFS2 (2P-PVFS2); ROMIO two-phase I/O over Lustre (2P-Lustre); ROMIO two-phase I/O over AHPIOS; and the two AHPIOS-based solutions: server-directed I/O and client-directed I/O. Our goal is to demonstrate that, by the tight integration between application and library offered by the full-AHPIOS solution, a significant performance improvement can be obtained.

In the evaluations for all solutions all compute nodes act as aggregators and the collective buffer size was 4 MB.

6.2.1 BTIO Benchmark NASA's BTIO benchmark (Wong and der Wijngaart, 2003) solves the Block-Tridiagonal (BT) problem, which employs a complex domain decomposition across a square number of compute nodes as shown in Figure 9 for nine processes. 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 (the subcubes along the diagonal line 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 verified for correctness. In this paper we report the results for the MPI implementation of the benchmark, which uses the MPI-IO's collective I/O routines. The collective I/O routines provide significantly better results than the independent ones, due to the coalescing of small requests and a more efficient usage of the network and disk transfers. On all runs we set the benchmark to execute 25 compute steps, which correspond to five I/O steps (five collective

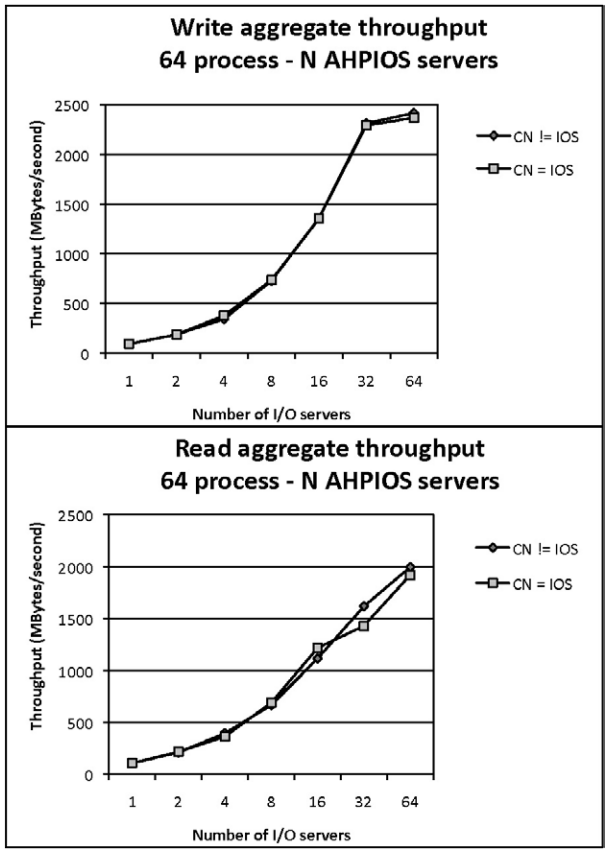

Fig. 8 File independent I/O scaling for 64 processes and *N* AHPIOS servers.

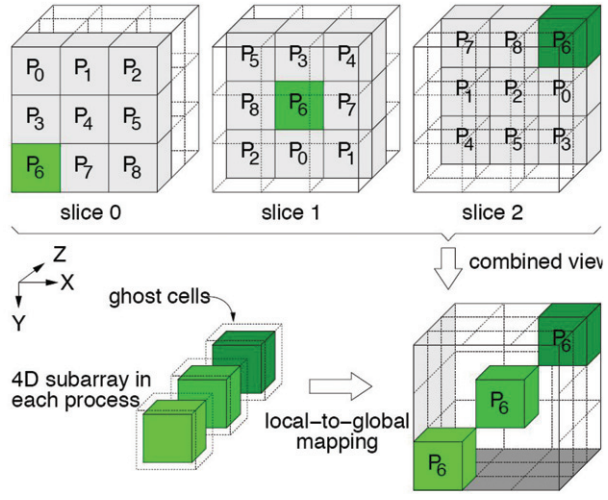

Fig. 9 BTIO data decomposition for nine processes.

Table 2
Granularity of BTIO File Accesses.

Number of processes	Access granularity in bytes (class B)	Access granularity in bytes (class C)
9	1,360	2,160
16	1,000	1,640
25	800	1,280
36	680	1,080
49	600	920
64	480	800

writes followed by five collective reads). The benchmark does not explicitly commit the file writes to disks.

The access pattern of BTIO is nested-strided with a nesting depth of two, with the file access granularity given in the Table 2.

Figures 10 and 11 show the results for the BTIO classes B and C. Because the latency is hidden by AHPIOS at file access time, we show the file write time on the first row, the file read time on the second , and finally the total time as reported by the benchmark (includes file open, set view, write, and close).

For each collective write operation for the client-directed I/O the data are written to the first-level cache on the aggregators, while the server-directed I/O transfers the data to the AHPIOS server in the second-level cache. As a consequence, in most cases the client-directed I/O hides better the write latency to the applications. However, when the final flushing is done the server-directed I/O outperforms the client-directed I/O due to the fact that the file close is done right after the last write. Therefore, there is no overlapping with communication or computation for this last step.

BTIO closes the file after writing and then reads the data for verification. For the client-directed I/O the write data are propagated from the first-level cache to the second-level cache, but a copy of the data is kept in the first-level cache. Therefore, the client-directed collective I/O routines read the data from the first-level cache. This explains the better results for collective reads when using client-directed I/O in most cases.

We note that the AHPIOS client-directed I/O and the server-directed I/O significantly outperform the write and read operations of the other three methods in all cases. In general, the client-directed I/O hides the latency better in most cases, because data are written to the clients and then staged through both cache layers. However, when the file close time is included, the server-directed I/O performs the best in all cases, because the data were already transferred to the AHPIOS servers asynchro-

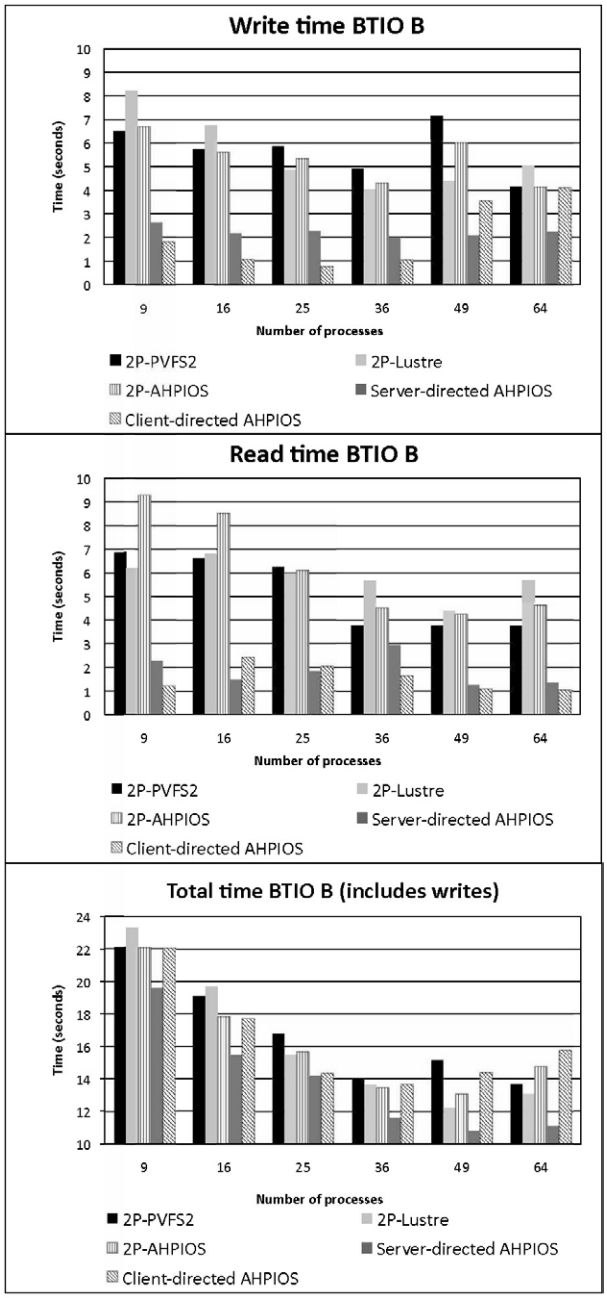

Fig. 10 BTIO class B measurements.

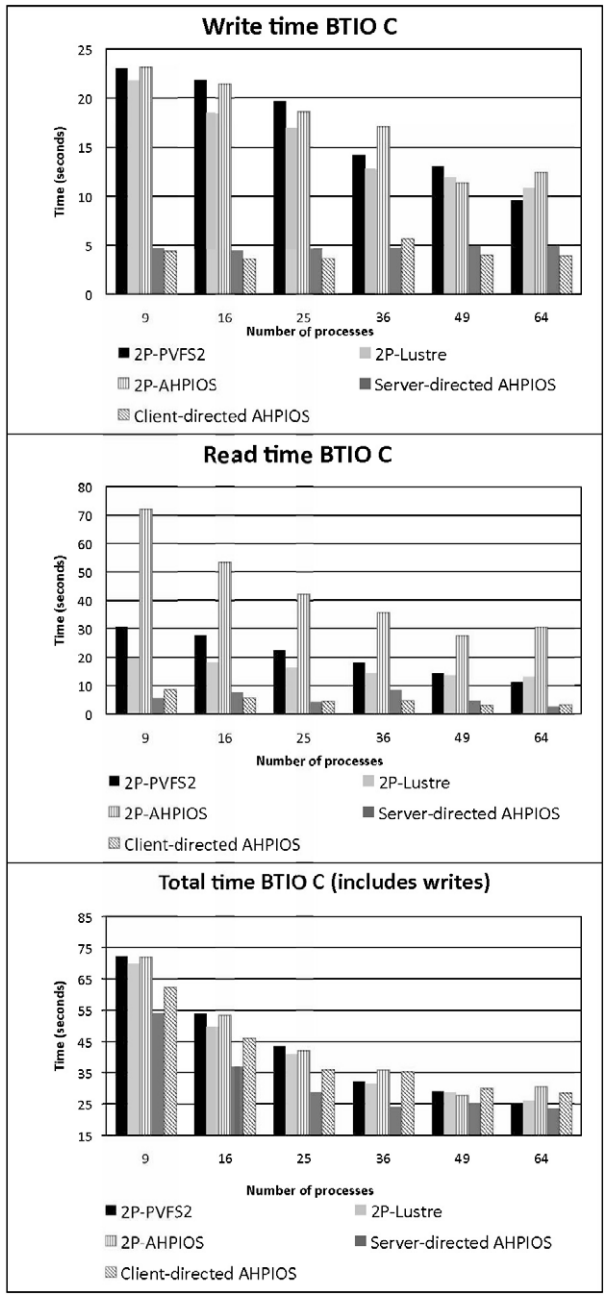

Fig. 11 BTIO class C measurements.

nously. In particular for class B, the client-directed I/O hides the latency of file writes better up to 36 processes, but does not scale well due to the fact that aggregators, as communication hubs, become overloaded: each aggregator receives data from all of the MPI processes, shuffles them and transfers them to all of the AHPIOS servers. In turn, the server-directed I/O performs better for increasing contention, as the communication is reduced. For class C, as the communication volume is larger, the better performance of the client-directed I/O over the server-

directed I/O is marginal for a small number of processes. As for class B, file writes of the server-directed I/O scale better. The file reads perform substantially better for both the server-directed I/O and the client-directed I/O than for the two-phase I/O, as the data is read from the collective cache levels: level 1 for the client-directed I/O and level 2 for the server-directed I/O. The performance is similar for both methods, as the communication volume and contention levels are roughly the same. In terms of total time reported by the application, which includes the write time and the time to completely flush the data to the AHPIOS servers (read time is not included), the server-directed I/O performs best in all cases, as the client-directed I/O incurs the cost of the flushing of the remainder of the data from the level 1 cache.

There are additional reasons explaining these results. In the two-phase I/O, data are in general transferred twice over the network: firstly, for data aggregation at the compute nodes, and secondly, to access the file system. In the server-directed I/O, 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. This is also the case for file-read operations in the client-directed I/O if the data is accessed in the first-level cache. This is the main factor for the better results of the BTIO file reads for the server-directed I/O and the client directed I/O, when compared to the two phase I/O.

In addition, the view I/O technique significantly reduces the size of the metadata sent over the network. The MPI data types are sent in a compact form to the AHPIOS servers at view declaration. This data type transfer is done only once and the data types can be reused by subsequent I/O operations. In contrast, the two-phase I/O requires the lists of (file offset, length) tuples to be sent to the aggregators at each file operation.

6.2.2 MPI Tile I/O Benchmark The MPI Tile I/O benchmark (MPI tile I/O, 2009) evaluates the performance of the MPI-IO library and the 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 the 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 the input parameters. We have chosen these values such that for any number of processors the total amount of data accesses is 1 GByte and the access granularity is 4 kB. Table 3 lists the values of these parameters. The size of an element is 1 byte.

The performance results are plotted in Figure 12. Client-directed I/O scales very well with the problem size. In this case the cooperative cache scales with the number of processes, because the clients put in common parts of their memory. The file-write performance seen by the application is much higher for the client-directed I/O as the data is synchronously transferred only to the first-level cooperative cache and then written back in the background to the second level. 2P-Lustre performs comparably to the others for small numbers of processes, but

Table 3
Parameters of the MPI Tile I/O Benchmark.

Number of processes	sz_x (kB)	sz_y (kB)	nr_x	nr_y
4	4	64	4	4
8	4	32	4	4
16	4	16	4	4
32	4	8	8	4
64	4	4	8	8

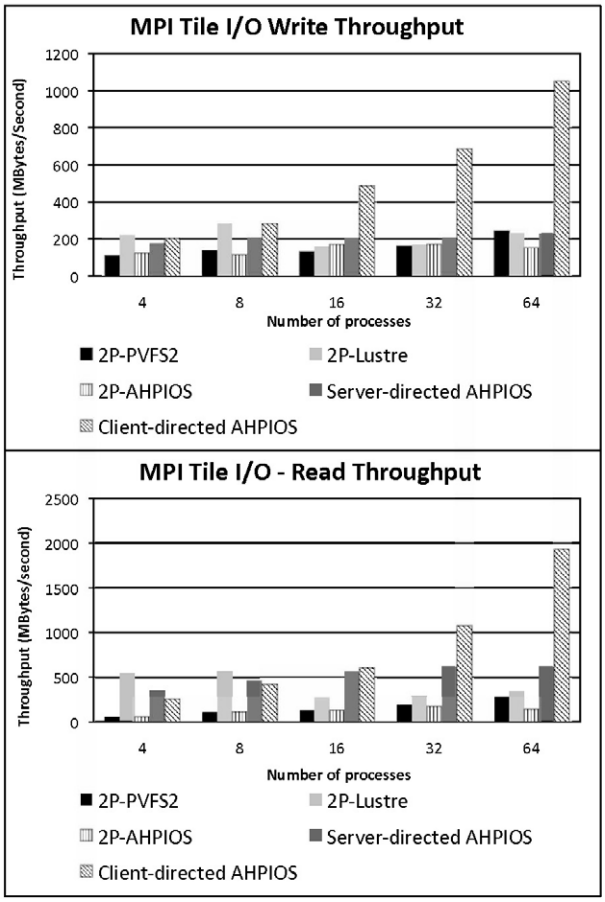

Fig. 12 MPI Tile I/O throughput.

does not scale, even though the two-phase aggregators cache the file blocks in their local memory (client-side caching). The performance seems to be affected by the POSIX semantics, even in this particular case of non-overlapping accesses. The server-directed I/O does not scale beyond eight processes, which represents the same number as the AHPIOS servers employed in this experiment. A larger number of AHPIOS servers would contribute to the scalability of the server-directed I/O, as seen in Section 6.1. The MPI Tile I/O benchmark does not invalidate file caches between write and read operations. Therefore, the expectation is that the performance of the solutions employing client-side caching (client-directed I/O and 2P-Lustre) will be superior to the others. Indeed, 2P-Lustre performs best for a small number of processes (four and eight), but does not scale with the number of processes. The client-directed I/O benefits the most from the client-side caching and significantly outperforms the other solutions. The server-directed I/O performs second-best for a large number of processors, by taking advantage of the second-level cooperative cache. The 2P solutions seem to suffer from the strict alternation of metadata (offset-length lists) and data transfers and conservative implementation with collective communication operations, as will be seen in the next section.

6.3 MPI Implementation Evaluation

AHPIOS is fully implemented in the MPI. The MPI offers a powerful paradigm for programming on distributed memory systems in terms of programming facility, portability, and performance. We found the employment of both point-to-point and collective operations very convenient. Both blocking and non-blocking point-to-point routines are straightforward to use and with a minimal management overhead (when compared with TCP/IP sockets, for instance). The collective synchronization routines, such as `MPI_Barrier` offer the possibility of a rapid (conservative) implementation and facilitate the debugging.

However, one of the most important advantages is portability; the AHPIOS system can run unmodified on all the systems that have a MPI library installed and they support the dynamic process mechanisms from MPI2.

One problematic aspect we found was the error mechanisms implemented in the MPI. In some cases, a MPI implementation must provide an explicit handling of errors, for instance for propagating an error from a faulting process. A global automatic exception mechanism for the MPI would significantly increase the productivity of implementing in the MPI.

In order to better understand the employment of the MPI in the solutions investigated in this paper, we traced the BTIO application by using the performance profiling library MPE (Gropp et al., 1995). We profiled the BTIO execution for nine processes and all evaluated I/O methods. In this case the size of traces was around 4MB. Tables 4 and 5 show the number of communication and synchronization calls performed by the BTIO for I/O purposes. Table 4 shows the number of MPI point-to-point calls for all processes, while Table 5 gives the number of collective calls. Each collective call is counted once for the group of processes that perform it.

Note that the AHPIOS server-directed I/O implementation employs only blocking point-to-point communication (`MPI_Send`, `MPI_Recv`), while the other implementations use both blocking and non-blocking point-to-point operations (`MPI_Isend`, `MPI_Irecv`, `MPI_Waitall`). Therefore, the performance of both server-directed and client-directed AHPIOS can be improved by a subsequent implementation with non-blocking operations.

For AHPIOS, the numbers from the tables include communication to the distributed storage, i.e. the communication overhead for providing a transparent parallel I/O access to files. In the case of Lustre and PVFS these operations are performed by using internal communication protocols and cannot be directly compared with the MPI routines. This fact explains why the client-directed I/O and 2P-AHPIOS generate a significantly higher number

Table 4
The Count of MPI Point-to-Point Communication Operations for *NP* = 9 Processes.

MPI call	2P-PVFS2	2P-Lustre	2P-AHPIOS	Server-directed AHPIOS	Client-directed AHPIOS
MPI_Send	0	0	4,660	4,660	4,660
MPI_Recv	0	0	4,660	4,660	5,830
MPI_Isend	5,670	5,760	5,760	0	5,553
MPI_Irecv	5,670	5,760	5,760	0	4,383
MPI_Waitall	855	855	855	0	396

Table 5
The Count of MPI Collective Communication and Synchronization Operations for *NP* = 9 Processes.

MPI call	2P-PVFS2	2P-Lustre	2P-AHPIOS	Server-directed AHPIOS	Client-directed AHPIOS
MPI_Bcast	60	60	60	0	0
MPI_Barrier	22	25	25	7	154
MPI_Allreduce	22	25	25	7	154
MPI_Alltoall	60	60	60	0	0
MPI_Allgather	20	20	20	0	0

of MPI messages. However, even under these conditions, the communication is lower for the server-directed I/O than for 2P-PVFS2 and 2P-Lustre.

The number of collective communication and synchronization operations performed by the two-phase I/O implementation is considerably higher. The two-phase I/O collective buffering is done at the compute nodes, which need to perform expensive all-to-all operations in the shuffle phase in order to get the list of file offset-length pairs and the data. The 2P-PVFS2, 2P-Lustre and 2P-AHPIOS solutions performed a similar number of operations (2P-AHPIOS and 2P-Lustre used three more barrier operations).

7 Further Application Domains

Besides large clusters of computers, two other potential application domains of AHPIOS are large-scale supercomputers and clouds.

7.1 Supercomputers

The architecture of large-scale supercomputers, such as IBM Blue Gene/L, IBM Blue Gene/P, and Cray XT3 systems, is organized by specializing the system into disjoint sets of compute and I/O nodes. The compute nodes are assigned to an application through a batch scheduler. Each set of compute nodes is served by dedicated I/O nodes. Therefore, the I/O nodes corresponding to the assigned compute nodes are known after the job is scheduled. AHPIOS servers can be spawned after the scheduling is performed. Here we describe our initial experiences of AHPIOS on Blue Gene systems.

On Blue Gene systems, the communication among compute nodes is performed through a torus network. The I/O calls, such as file accesses, are forwarded from the compute nodes to the I/O nodes through a dedicated tree network. On the I/O nodes these requests are served by I/O daemons. In our experiments we employ the ZOID daemon (Iskra et al., 2008), which is an open software package developed at ANL that can be used alternatively to IBM's CIOD daemon.

Blue Gene systems currently do not offer MPI dynamic process management. In our initial setup, I/O daemons are started on the I/O nodes as MPI processes. Each AHPIOS server is the thread of an I/O daemon running on the I/O node. The MPI-IO calls are forwarded through the tree network to the AHPIOS servers, where they are processed in cooperation. The communication among the AHPIOS servers is performed through MPI calls and goes over the switched network interconnecting the I/O nodes. We have managed to deploy the AHPIOS on the Blue Gene/L system in the Argonne National Lab (ANL) and we are currently experimenting with the new Blue Gene/P system installed there.

7.2 Clouds

Many researchers agree that, in the future, companies will rely less on their own infrastructures and more on remote clouds. A cloud is defined as an Infrastructure as a Service (IaaS). Amazon Amazon web services site (2008) provides pay-per-use computing and storage resources through Web Services. With Amazon Elastic Computing Cloud (EC2), users may run instances of virtual machines consisting of a CPU, memory and local disks. The cloud is elastic, the users can increase or decrease their computation needs by acquiring or releasing VM instances. The local disks store the information only during the life of an instance. For permanent storage, the users should use either the Elastic Block Store (EBS) or Simple Storage System (S3). EBS is an elastic storage partition (that can be increased or decreased), on top of which a file system can be installed. EBS can be mounted concurrently inside distinct VM instances. S3 is a persistent storage service on which users can store objects. Storing data on the local storage of each instance comes at no extra cost, while there is a charge for storing both on EBS and S3. Other academic or research cloud projects, such as Nimbus Nimbus (Nimbus Cloud Project, 2008)

and Stratus (Stratus Cloud Project, 2008), offer EC2-like functionality. However, none of these projects offers a parallel I/O distributed system, such as AHPIOS.

AHPIOS could be used in two ways with EC2-like services. Firstly, AHPIOS could offer a MPI-IO integrated, shared, distributed partition, based on the local storage of several available instances. This provides an efficient on-demand parallel I/O system to MPI applications running on the available instances. In addition, AHPIOS can be simply scaled up or down by a simple restart. However, in this case the data from the local storage has to be backed up on either EBS or S3, for instance through an asynchronous data staging running in the background. Secondly, AHPIOS could be used for efficiently store the data in parallel over several EBS partitions. Currently, we are investigating both possibilities and we plan to implement and evaluate both of them in the near future.

8 Conclusions and Future Work

We have presented the AHPIOS parallel I/O system, an ad-hoc parallel system that allows on-demand virtualization of distributed resources, provides a high-performance parallel I/O and that can be used as a cost-efficient alternative to traditional parallel file systems. AHPIOS is completely implemented in the MPI and offers a scalable, efficient platform for parallel I/O. The two-level cooperative cache scales with the number of processors at the first level and with the number of storage resources at the second level. The strategy of asynchronous data staging between the caching levels hides the latency of file accesses from the applications.

The performance results show that the tight integration of the application and storage system, together with the asynchronous data-staging strategy and the cooperative caching, bring a substantial benefit over traditional solutions.

AHPIOS can be used as an alternative to traditional parallel file systems where the applications require a limited set of parallel I/O functionality. Firstly, as in the case of PVFS, AHPIOS requires an external locking mechanism in order to offer atomic file access semantics for overlapping accesses. Secondly, the current prototype offers a MPI-IO interface. Nevertheless, we are currently implementing a POSIX-like interface by using FUSE (Filesystems in Userspace, FUSE). Thirdly, AHPIOS requires the dynamic process mechanism of the MPI2, which is not yet directly integrated with the current schedulers. However, dynamic processes can be spawn on partitions pre-reserved at application scheduling time. Fourthly, AHPIOS can be virtualized on demand to locally available disks. However, AHPIOS is not limited to local disks and can be used with any available storage resource.

Future work will concentrate on expanding the applicability domain of AHPIOS. We believe that AHPIOS is a suitable solution not only for clusters of computers, but also for supercomputers, grids, and clouds. Recently, we have deployed the AHPIOS on a Blue Gene/L system at the Argonne National Laboratory. Currently, we are working toward installing and evaluating the system on the new Blue Gene/P. In addition, we envision AHPIOS as a high-performance parallel I/O solution for computing clouds. We plan to install and evaluate AHPIOS on existing cloud solutions.

Acknowledgements

This work is supported in part by the Spanish Ministry of Education under project TEALES (TIN2007-63092). This work was also supported in part by the U.S. Department of Energy (DOE) SCIDAC-2: Scientific Data Management Center for Enabling Technologies grant DE-FC02-07ER25808, DOE FASTOS award number DE-FG02-08ER25848. This work was also supported in part by the National Science Foundation (NSF) under HECURA CCF-0621443, NSF SDCI OCI-0724599, and NSF ST-HEC CCF-0444405. This research was supported in part by the NSF through TeraGrid resources provided by the TACC, under TeraGrid Projects TG-CCR060017T, TG-CCR080019T, and TGASC080050N.

Authors' Biographies

Florin Isaila has been an Assistant Professor of the University Carlos III of Madrid since 2005. Previously, he was teaching and research assistant in the Departments of Computer Science of Rutgers University and the University of Karlsruhe. His primary research interests are parallel computing and distributed systems. He is currently involved in various projects on topics including parallel I/O, parallel architectures, peer-to-peer systems, and the Semantic Web. He received a PhD in Computer Science from the University of Karlsruhe in 2004 and a MS from Rutgers, The State University of New Jersey in 2000.

Javier Garcia Blas has been a Teaching Assistant at the University Carlos III of Madrid since 2005. He is currently a PhD student in Computer Science. He received the MS degree in Computer Science in 2007 at the University Carlos III of Madrid. He was a visiting scholar at HLRS in 2006–2007 and at Argonne National Lab in 2007–2008. His primary research interests are parallel computing and distributed systems. He is currently involved in various projects on topics including parallel I/O and parallel architectures.

Jesús Carretero is Full Professor of Computer Architecture and Technology at the Universidad Carlos III de Madrid (Spain), where he has been responsible for that knowledge area since 2000. He is also Director of the Master in Administration and Management of Computer Systems, which he founded in 2004. He serves as a Technology Advisor in several companies. His major research interests are in parallel and distributed systems, real-time systems and computing systems architecture. He is Senior Member of the IEEE.

Wei-keng Liao is a Research Associate Professor in the Electrical Engineering and Computer Science Department at Northwestern University. He received his PhD in Computer and Information Science from Syracuse University. His research interests are in the areas of high-performance computing, parallel I/O, parallel data mining, and data management for scientific applications.

Alok Choudhary is the chair and professor of the Electrical Engineering and Computer Science Department in the McCormick School of Engineering and Applied Science at Northwestern University. He also holds an appointment in the Technology Industry Management at Kellogg School of Management. He is the founder and director of the Center for Ultra-scale Computing and Information Security (CUCIS). He joined Northwestern in 1996. Prior to that he was a faculty member of the ECE department at Syracuse University. He received his PhD in Electrical and Computer Engineering in 1989 from the University of Illinois, Urbana-Champaign. He received his MS from the University of Massachusetts, Amherst in 1986. He received the National Science Foundation's Young Investigator Award in 1993 (1993–1999). He has also received an IEEE Engineering Foundation award, an Intel research council award (1993–1997, 2003–2005), and an IBM Faculty Development award. In 2006 he received the first award for "Excellence in Research, Teaching and Service" from the McCormick School of Engineering. He is a Fellow of the IEEE. He has published more than 300 papers in various journals and conferences. He has also written a book and several book chapters. His research has been sponsored by (past and present) DARPA, NSF, NASA, AFOSR, ONR, DOE, Intel, IBM, and TI.

References

Bordawekar, R. (1997). Implementation of collective I/O in the Intel paragon parallel file system: initial experiences. In Proceedings of the 11th International Conference on Supercomputing, July, Vienna, Austria. New York: ACM. pp. 20–27.

Ching, A., Choudhary, A., Liao, W. K., Ross, R. and Gropp, W. (2003). Efficient structured data access in parallel file systems. In Proceedings of the IEEE International Conference on Cluster Computing, December.

Crandall, P., Aydt, R., Chien, A. and Reed, D. (1995). Input/output characteristics of scalable parallel applications. In Proceedings of Supercomputing '95.

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

Fryxell, B., Olson, K., Ricker, P., Timmes, F. X., Zingale, M., Lamb, D. Q., MacNeice, P., Rosner, R. and Tufo, H. (2000). FLASH: an adaptive mesh hydrodynamics code for modelling astrophysical thermonuclear flashes. *Astrophys. J. Suppl.* **131**: S273–S334.

Garcia-Carballeira, F., Calderon, A., Carretero, J., Fernandez, J. and Perez, J. M. (2003). The design of the Expand parallel file system. *Int. J. High Perform. Comput.*, **17**(1): 21–38.

Geist, A. (2008). MPI must evolve or die. In EuroPVM/MPI. http://dx.doi.org/10.1007/978-3-540-87475-1_4.

Gropp, W., Karrels, E. and Lusk, E. (1995). MPE graphics: scalable X11 graphics in MPI. In Proceedings of the 1994 Scalable Parallel Libraries Conference, October 12–14, Mississippi State University, Mississippi, pp. 49–54, IEEE Computer Society Press.

HDF5 home page (2009). http://hdf.ncsa.uiuc.edu/HDF5.

Hermanns, M.-A., Berrendorf, R., Birkner, M. and Seidel, J. (2006). Flexible I/O support for reconfigurable grid environments. In Proceedings of Euro-Par, pp. 415–424.

Hildebrand, D. and Honeyman, P. (2005). Exporting storage systems in a scalable manner with pnfs. Proceedings of the IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies, pp. 0:18–27.

HP MPI home page (2009). http://www.hp.com.

Amazon web services site (2008). http://aws.amazon.com/.

Filesystems in Userspace (FUSE) (2009). http://fuse.sourceforge.net/.

Nimbus Cloud Project (2008). http://workspace.globus.org/clouds/nimbus.html.

Stratus Cloud Project (2008). http://www.acis.ufl.edu/vws/.

High Productivity Computer Systems (2008). http://www.highproductivity.org/.

PanFS web site (2008). http://www.panasas.com/panfs.html.

Top 500 list (2009). http://www.top500.org.

Cluster File Systems Inc. (2002). Lustre: A scalable, high-performance file system. White Paper, version 1.0, Cluster File Systems Inc., November. http://www.lustre.org.

LAM website (2009) Indiana University, http://www.lam-mpi.org/.

Isaila, F., Blas, J. G., Carretero, J., Liao, W. K. and Choudhary, A. (2008). AHPIOS: an MPI-based ad-hoc parallel I/O system. In Proceedings of IEEE ICPADS.

Isaila, F., Malpohl, G., Olaru, V., Szeder, G. and Tichy, W. (2004). Integrating collective I/O and cooperative caching into the "Clusterfile" parallel file system. In Proceedings of the ACM International Conference on Supercomputing (ICS), pp. 315–324, ACM Press.

Isaila, F., Singh, D., Carretero, J. and Garcia, F. (2006). On evaluating decentralized parallel I/O scheduling strategies for parallel file systems. In Proceedings of VECPAR 2006.

Isaila, F., Singh, D., Carretero, J., Garcia, F., Szeder, G. and Moschny, T. (2006). Integrating logical and physical file models in the MPI-IO implementation for Clusterfile. In Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID), IEEE Computer Society.

Isaila, F. and Tichy, W. (2001). Clusterfile: a flexible physical layout parallel file system. In Proceedings of the First IEEE International Conference on Cluster Computing, October.

Isaila, F. and Tichy, W. (2003). View I/O: improving the performance of non-contiguous I/O. In Proceedings of the Third IEEE International Conference on Cluster Computing, December, pp. 336–343.

Iskra, K., Romein, J. W., Yoshii, K. and Beckman, P. (2008). ZOID: I/O-forwarding infrastructure for petascale architectures. In Proceedings of PPoPP '08, pp. 153–162.

Kim, G. H., Minnich, R. G. and Mcvoy, L. (1994). Bigfoot-nfs: A parallel file-striping nfs server (extended abstract).

Kotz, D. (1994). Disk-directed I/O for MIMD multiprocessors. In Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation.

Li, J., Liao, W. K., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B. and Zingale, M. (2003). Parallel netCDF: a high-performance scientific I/O interface. In Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC '03), Washington, DC, USA, IEEE Computer Society p. 39.

Liao, W. K., Coloma, K., Choudhary, A., Ward, L., Russel, E. and Tideman, S. (2005). Collective caching: application-aware client-side file caching. In Proceedings of the 14th International Symposium on High Performance Distributed Computing (HPDC), July.

Liao, W. K., Coloma, K., Choudhary, A. N. and Ward, L. (2005). Cooperative write-behind data buffering for MPI I/O. In Proceedings of PVM/MPI, pp. 102–109.

Ligon, W. and Ross, R. (1999). An overview of the parallel virtual file system. In Proceedings of the Extreme Linux Workshop, June.

Lombard, P. and Denneulin, Y. (2002). nfsp: a distributed nfs server for clusters of workstations. In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2002), Abstracts and CD-ROM, pp. 35–40.

Lonestar home page (2009) http://www.tacc.utexas.edu.

Message Passing Interface Forum (1997) MPI2: Extensions to the Message Passing Interface. http://www.mpi-forum.org

MPI Forum (1995). MPICH website. http://www-unix.mcs.anl.gov/mpi/mpich/.

MPI tile I/O (2009). http://www-unix.mcs.anl.gov/pio-benchmark/.

NEC MPI home page (2009). http://www.nec.com.

Nieuwejaar, N., Kotz, D., Purakayastha, A., Ellis, C. and Best, M. (1996). File access characteristics of parallel scientific workloads. *IEEE Trans. Parallel Distr. Syst.* textbf7(10): 1075–1089.

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

Schmuck, F. and Haskin, R. (2002). GPFS: a shared-disk file system for large computing clusters. In Proceedings of FAST.

Seamons, K., Chen, Y., Jones, P., Jozwiak, J. and Winslett, M. (1995). Server-directed collective I/O in Panda. In Proceedings of Supercomputing '95.

SGI MPI home page (2009). http://www.sgi.com.

Simitici, H. and Reed, D. (1998). A comparison of logical and physical parallel I/O patterns. *Int. J. High Perform. C.* **12**(3): 364–380.

Smirni, E. and Reed, D. (1997). Workload characterization of I/O intensive parallel applications. In Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation, June.

Stockinger, K. and Schikuta, E. (2000). ViMPIOS, a "truly" portable MPI-IO implementation. In Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing (PDP'2000). IEEE Computer Society Press.

Taki, H. and Utard, G. (1999). MPI-IO on a parallel file system for cluster of workstations. In Proceedings of the 1st IEEE Computer Society International Workshop on Cluster Computing (IWCC '99), Washington, DC, USA, IEEE Computer Society, p. 150.

Tang, H., Gulbeden, A., Zhou, J., Strathearn, W., Yang, T. and Chu, L. (2004). A self-organizing storage cluster for parallel data-intensive applications. In Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC '04), Washington, DC, USA, IEEE Computer Society, p. 52.

Teragrid home page (2009). http://www.teragrid.org/.

Thakur, R., Gropp, W. and Lusk, E. (1999). Data sieving and collective I/O in ROMIO. In Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, February, pp. 182–189.

Thakur, R., Gropp, W. and Lusk, E. (1999). On implementing MPI-IO portably and with high performance. In Proceedings of IOPADS, May, pp. 23–32.

Thakur, R., Gropp, W. and Lusk, E. (2002). Optimizing noncontiguous accesses in MPI-IO. *Parallel Comput.* **28**(1): 83–105.

Thakur, R. and Lusk, E. (1996). An abstract-device interface for implementing portable parallel-I/O interfaces. In Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation, pp. 180–187.

Wang, F., Xin, Q., Hong, B., Br, 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.

Yu, W., Vetter, J., Canon, R. S. and Jiang,S. (2007). Exploiting Lustre file joining for effective collective IO. In Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID '07), Washington, DC, USA, IEEE Computer Societypages, pp. 267–274.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E. and Maltzahn, C. (2006). Ceph: a scalable, high-performance

distributed file system. In Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06), Berkeley, CA, USA, USENIX Association, New York: ACM. pp. 307–320.

Weil, S. A., Leung, A. W., Brandt, S. A. and Maltzahn, C. (2007). Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In PDSW, edited by G. A. Gibson. ACM Press, pp. 35–44.

Winslett, M., Seamons, K., Chen, Y., Cho, Y., Kuo, S. and Subramaniam, M. (1996). The Panda library for parallel I/O of large multidimensional arrays. In Proceedings of the Scalable Parallel Libraries Conference III, October.

Wong, P. and der Wijngaart, R. (2003). NAS parallel benchmarks I/O version 2.4. Technical report, NASA Ames Research Center.