

# Latency hiding file I/O for Blue Gene systems

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

Robert Latham, Samuel Lang, Robert Ross - *Argonne National Laboratory*  
{robl, slang, rross}@mcs.anl.gov

## Abstract

*This paper presents the design and implementation of a novel file I/O solution for Blue Gene systems. We propose a hierarchical I/O cache architecture based on open source software. Our solution is based on an asynchronous data staging strategy, which hides the latency of file system access from compute nodes. The performance results demonstrate the high scalability and significant performance improvements of our architecture over existing solutions.*

## 1 Introduction

In the last years the exponential increase in the processing power of large computing systems has continued. A significant number of systems from Top 500 [5] is represented by large supercomputers with hundreds of thousands of processors such as Blue Gene and Cray.

In order to make full benefit of the processing scalability, the parallel applications need also a scalable parallel I/O system. This paper addresses this challenge by proposing a novel scalable parallel I/O solution for Blue Gene systems. This solution consists of a scalable multi-tier caching architecture and a I/O architecture implementation hiding the latency of file accesses to the applications.

The parallel file system is a basic component of any scalable parallel I/O solution, which stripe file data and metadata over several independent disks managed by I/O nodes in order to allow parallel file access from several compute nodes. The parallel file systems employed in the majority of the scalable parallel architectures include GPFS [18], PVFS [13] and Lustre [6]. The solution presented in this paper uses the PVFS parallel file system, but any other parallel file system can be included with minimal changes.

A limited number of papers have proposed novel solutions for scalable parallel I/O systems of large supercomputers. Nevertheless, the supercomputers such as Blue Gene have a complex architecture consisting of several networks,

several tiers (computing, I/O, storage) and, consequently a potential deep cache hierarchy. This architecture provides a rich set of opportunities for optimizations.

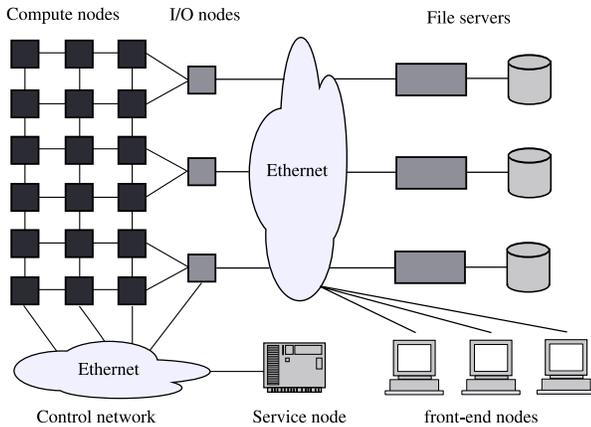
The contributions of this paper can be summarized as follows:

- We present the design and implementation of a novel hierarchical I/O cache architecture for Blue Gene systems based on open source software.
- We show how asynchronous data staging can hide the latency of file system access from compute nodes.
- The performance results demonstrate the high scalability and superior performance of our architecture over existing solutions.
- We discuss how the limitations of the Blue Gene/L architecture affect our solution and describe the way in which our architecture can benefit from design features of the successor Blue Gene/P supercomputer.

The remainder of the paper is structured as follows. Section 2 reviews related work. The hardware and operating system architectures of Blue Gene/L is presented in Section 3. We discuss our latency hiding file I/O solution in Section 4. The experimental results are presented in Section 5. Finally, we summarize and discuss future work in Section 6.

## 2 Related work

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* [12, 19]. If the merging occurs at intermediary nodes or at compute nodes, the method is called *two-phase I/O* [3, 2]. Data shipping [16] is a collective optimization that uniquely binds each file block in a round-robin manner to



**Figure 1. Blue Gene/L architecture overview.**

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.

MPI-IO [14] has imposed as a portable and high performance interface for parallel applications. The MPI-IO standard and its distributions (such as the popular ROMIO [22]) offer various file access optimizations on top of existing file systems. Several researchers have contributed with optimizations of MPI-IO data operations: data sieving [21], non-contiguous access [23], collective caching [10], cooperating write-behind buffering [11], integrated collective I/O and cooperative caching [8].

A limited number of recent studies have proposed and evaluated parallel I/O solutions for supercomputers. Yu et al. [17] present a GPFS-based three-tiered architecture for Blue Gene/L. The tiers are represented by I/O nodes as GPFS clients, network shared disks and storage area network. Our solution extends this hierarchy to include the memory of the compute nodes and proposes an asynchronous data staging strategy that hides the latency of file accesses from the compute nodes. An implementation of MPI-IO for Cray architecture and Lustre file system is described in [26]. In [25] the authors propose a collective I/O technique, in which processes are grouped together for collective I/O according to the Cray XT architecture.

### 3 Blue Gene/L

This section presents the hardware and operating system architectures of Blue Gene/L.

#### 3.1 Blue Gene/L architecture

Figure 1 shows a high-level view of a Blue Gene/L system. Applications run on compute nodes. Compute nodes

are grouped into processing sets, or “pset”. Each pset has an associated I/O node, which performs I/O operations on behalf of the compute nodes from the pset. The supported sizes of a Blue Gene/L pset are 8, 16, 32, 64 or 128 compute nodes (determined when machine is built). The compute and I/O nodes are controlled by service nodes. The file system components run on dedicated file servers connected to storage nodes.

Compute and I/O nodes use the same ASIC with two PowerPC 440 cores, with non-coherent L1 caches, coherent 2KB L2 caches, and a 4MB shared DRAM L3 cache. The RAM may have 512 MBytes or 1 GBytes.

Compute and I/O nodes are interconnected by five networks: 3D torus, collective, global barrier, Ethernet and control. The 3D torus is typically used for point-to-point communication between compute nodes. The collective network has a tree topology and serves collective communication operations and I/O traffic. The global barrier network offers an efficient barrier implementation. The Ethernet network interconnects I/O nodes and file servers. The service nodes control the whole machine through the control network.

A Blue Gene/L system can be divided in partitions. At a given time each partition executes only one job. The compute nodes of a partition may run in two modes: coprocessor and virtual. In coprocessor mode one processor is used for computation and has access to the whole memory, while the other performs communication operations. In virtual mode both processors perform communication and computation.

#### 3.2 Operating system architecture

In the IBM solution [15], the compute node kernel (CNK) is a light-weight operating system offering basic services: creation of one or two address spaces (depending if the running mode is coprocessor or virtual), simple system calls such as setting an alarm, and forwarding I/O-related system calls to the I/O nodes. CNK does not offer local support for multi-threading, TCP/IP, file systems, or system calls such as fork/exec or mmap.

The I/O system calls are forwarded to and served by the I/O node associated to the pset. The I/O nodes run a simplified Linux OS kernel (IOK) with a small memory footprint, an in-memory root file system, TCP/IP and file system support, no swapping, and lacking the majority of classical daemons. The I/O nodes do not run applications. Because the L1 caches are not coherent, all the threads of a daemon are running on the same core.

The I/O forwarding from compute to I/O node is similar to Remote Procedure Calls. As shown in Figure 2(a), the file system calls of compute nodes are shipped through the tree collective network to the control and I/O daemon (CIOD) on the I/O node. CIOD executes the requested sys-

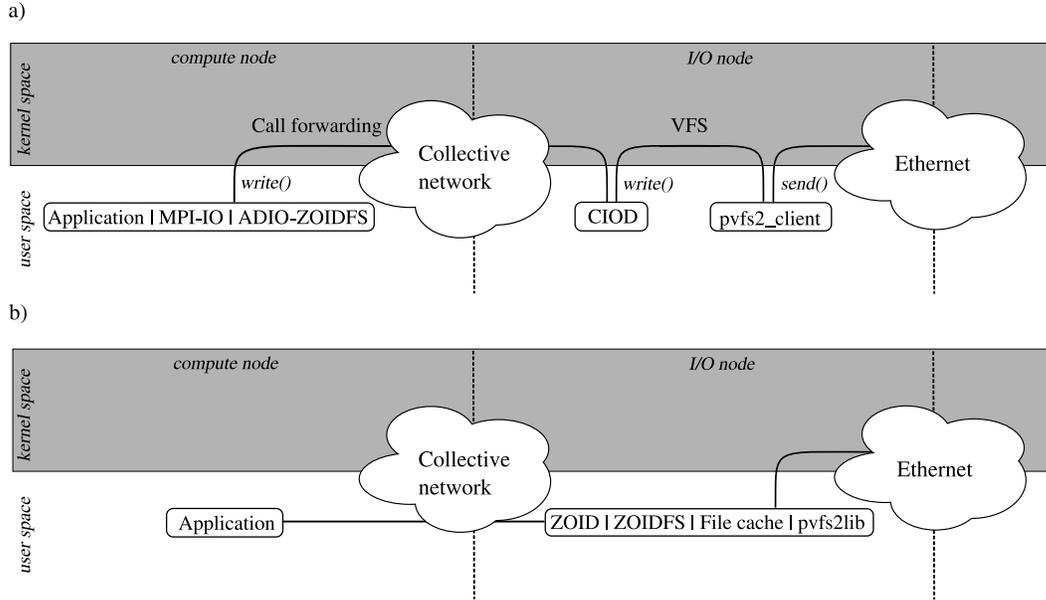


Figure 2. I/O forwarding for IBM and ZeptoOs solutions.

tem calls on locally mounted file systems and returns the results to the I/O nodes. On the I/O node the communication proceeds in kernel space, while CIOD is a user-level daemon serving a mounted file system. Consequently, several context switches are required for each file access.

## 4 Latency hiding file I/O

In this section we present our parallel I/O solution, which is based on ZOID and ROMIO. ZOID [9] is an I/O forwarding software developed under the ZeptoOS project [4]. ROMIO [22] is the most popular MPI-IO implementation and is part of the majority of MPI distributions.

### 4.1 ZOID and ZeptoOS

The ZeptoOS project is developing an open-source alternative to the proprietary software stacks available on massively parallel architectures such as Blue Gene and Cray.

The I/O forwarding is implemented in a ZeptoOS component called ZOID [9]. ZOID is designed to improve the performance and flexibility of existing I/O forwarding proprietary solutions. ZOID infrastructure consists of a multi-threaded I/O daemon, a tool for automatic generation of forwarded I/O routines, and an optimized network protocol.

As shown in Figure 2(b), the multi-threaded I/O ZOID daemon is the equivalent of CIOD: it runs on each I/O node and serves the I/O requests of the compute node. Each compute node is served by a distinct ZOID thread.

The ZOID daemon can be easily extended with new functionality in a flexible way by plug-ins. In order to implement new forwarded calls as a plug-in, a programmer declares the functions to be forwarded from a compute node (client) to an I/O node (server) in a header file. Subsequently, a ZOID tool generates client and server stubs. Finally, the programmer implements the server functions and deploys them to the I/O node, where they are plugged into the ZOID daemon as a dynamic library.

ZOID daemon and compute nodes communicate through an optimized network protocol running in user space. The network communication for I/O forwarded calls is automatically generated by the ZOID tool.

Unlike CIOD, the communication proceeds in user space and the extensibility of ZOID allows to easily add user-level plug-ins such as ZOIFDS.

ZOIFDS [9] is a ZOID plug-in for I/O forwarding of file system calls. ZOIFDS abstracts away the details of a file system API under a stateless interface consisting of generic functions for file create, open, write, read, close, etc. As seen in the right hand side of Figure 2 (b), the ZOIFDS solution does not require any context switch. The solution presented in this paper is based on ZOID-FS backend over the PVFS parallel file system.

### 4.2 ADIO

The most wide-spread implementation of MPI-IO standard is ROMIO [22]. In ROMIO, the MPI-IO interface is implemented portably on top of an abstract device interface called ADIO [20]. ADIO consists of general-purpose op-

timizations and a file system specific implementation. The general optimizations include among others file views and collective two-phase I/O [21].

The files system specific part has to be implemented for providing MPI-IO support for a novel file system, as in the case of ZOIDFS. The ZOIDFS-ADIO implementation calls the client stubs generated by ZOID generation tools. The client stubs forward the calls to the ZOID on the I/O node, where they are served.

### 4.3 Collective I/O

Our collective I/O solution is based on view-based collective I/O [1]. In view-based I/O each file block is uniquely mapped to one process called aggregator, which is responsible to perform the file access on behalf of all processes of the application. When an MPI-IO application defines a collective view (MPI views are collective operations), the views of all processes are transferred to all aggregators, where they are cached. At access time, contiguous view data can be transferred between compute nodes and aggregators: using the view parameters, the aggregator can perform locally the scatter/gather operations between view data and file blocks.

The aggregators cache the data in collective buffer cache. In the original view-based solution this buffer cache is asynchronously flushed to the final file system by an I/O thread. Because the Blue Gene/L does not offer support for threads on the compute nodes, the asynchronous flushing has not been used in the Blue Gene/L solution. However, the full solution has been just ported on Blue Gene/P, in which multi-threaded applications can run on compute nodes. The results presented in this paper can be further improved by this technique.

In the existing solutions for Blue Gene systems, the tree network is used for file system traffic. In our solution the torus network is leveraged for collecting the file system requests and data to aggregators. Subsequently, the aggregators transfer data to the I/O nodes through the tree network. This approach reduces the contention on the tree network by avoiding the transfer of small I/O requests.

There are several differences between view-based I/O and two-phase I/O. First, in view based I/O files are mapped onto aggregator at file block granularity, while in two-phase I/O the file access range is evenly split among aggregators. Caching file blocks at I/O nodes is complicated for two-phase I/O because different accesses will produce data blocks with different sizes, complicating the consistency protocol. Second, in view-based I/O, the MPI-IO view is transferred once and reused several times. Third, in two-phase I/O lists of file offsets and lengths have to be send for each file access, whereas in view-based I/O the views are used for scatter/gather. Fourth, unlike two-phase I/O, view-

based I/O offers an compute node cache, which scales with the number of compute nodes.

### 4.4 Asynchronous data staging

In order to hide the latency of file accesses we have implemented a file cache on the I/O node. As depicted in Figure 2 (b), the file cache module lies between ZOIDFS server stub and the PVFS2 library. It is managed by a dedicated data staging ZOID thread, which asynchronously stages the data between the local cache and the file system servers from the storage node.

Incoming write requests do not wait for the data to be transferred to the file servers through the Ethernet network: the data is copied into the I/O node cache and a successful acknowledgment is returned to the compute node. Subsequently, the data is asynchronously flushed to the file servers by the data staging thread. The write requests have to wait only if the cache is full. A prefetching policy can be implemented in a similar way: the data staging thread may prefetch blocks sequentially or depending on user hints. In this paper we present an evaluation of the flush policy. Current work concentrates on prefetching.

### 4.5 Consistency issues

In the presented solution, in order to insure consistency, each file system block can be cached only at one I/O node. Therefore, each file block assigned to one I/O node must be gathered/scattered at aggregators located at the pset corresponding to the same I/O node. This constraint is imposed by the fact that a compute node can transfer I/O related traffic only to its master I/O node. View-based I/O naturally addresses this constraint, as the file blocks can be mapped in a user-defined way to aggregators (the default mapping is round-robin). The blocks assigned to aggregators are mapped to local I/O nodes, where they can be cached.

An alternative solution is based on cooperative caches of the I/O nodes. As in the previous solution, each file block is assigned to exactly one I/O node. However, if one compute node writes data to a block assigned to an I/O node from a different pset, the data is sent first to the own I/O node, who forwards it to the destination I/O node. Ongoing work concentrates in the implementation of this solution under the AHPIOS parallel I/O system [7].

## 5 Experimental results

The experiments presented in this paper have been performed on the Blue Gene/L system from Argonne National Lab. The system has 1024 dual-core 700 MHz PowerPC 440 processors with 512 MB of RAM. Data is transferred between compute nodes and I/O nodes over the a global

tree network with a bandwidth of 2.8Gb / link. Each pset of 32 compute nodes is served by one I/O node. All 32 I/O nodes are interconnected to 14 storage nodes through a Gigabit Ethernet interface. The storage nodes provide mass storage for the BlueGene/L system. Each storage node contains a ServeRAID 6i+ SCSI RAID Controller, which connects to six internal 146.8 GB 10K SCSI HDDs for a total of 880 GB raw storage per server or 14 TB total raw storage (11.7 TB usable). Requests to storage nodes are served by 4 xSeries 346 servers with dual 3.4 GHz Xeon processors, 4 GB RAM. All the experiments were run in coprocessor mode (one MPI process per node).

### 5.1 Contiguous access

We have implemented a benchmark that simulates the behavior of parallel applications. The benchmark consists of alternating compute phases and I/O phases. The compute phases are simulated by idle spinning. In the I/O phase all the processors write non-overlapping contiguous records to a file. The number of alternating phases was 20. The maximum file size produced by 512 processes and record size of 1MB was 10GB. The compute nodes do not use any caching. We compare four cases: IBM's CIOD-based solution, ZOID without cache, ZOID with cache and zero-time compute phase and ZOID with cache with a 2 seconds compute phase. The I/O node thread starts flushing data to the file system, when more than half of the cache contains dirty blocks.

In one setup we have fixed the record size (we report two cases: 128KB and 1MB) and varied the number of processors from 32 (one pset) to 512 (16 pssets). In the second setup we use a fixed number of processors (we report two cases: 64 and 512 processors) and vary the record size from 1KB to 1MB. Figures 3, 4, 5, and 6 display the results: in the upper row the aggregate throughput of all nodes and in the lower row the file close times. The file close times are relevant because the remaining dirty blocks of I/O node cache are flushed to the file system at close time.

As expected, we can notice that ZOID solutions caching file data at I/O nodes significantly outperform the CIOD and ZOID without cache. The close times of caching solutions is larger than for non-caching solutions. However, the close time pays off when compared with the benefit in terms of aggregate throughput. The close time is smaller for the 2-seconds compute phase, as the cache is asynchronously flushed to the file system while the computing proceeds.

The aggregate throughput is especially large for small records for both 64 processors and 512 processors. For small records the file system latencies are large, therefore, the effects of latency hiding have a substantial impact.

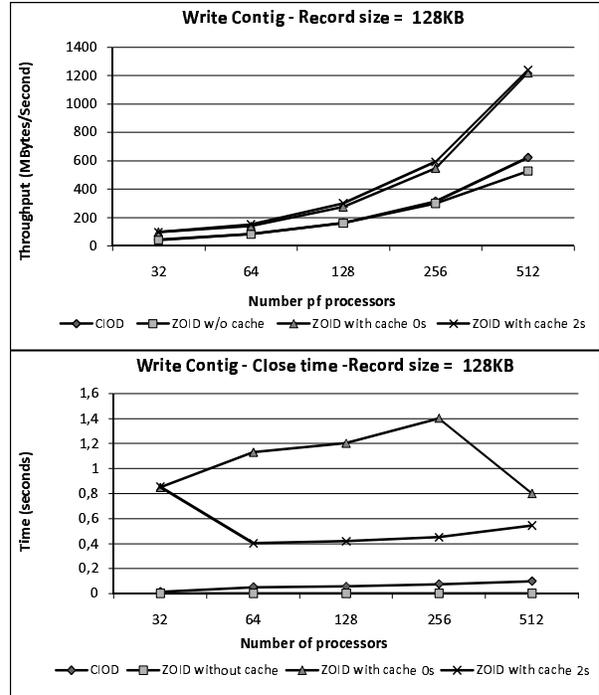


Figure 3. Performance of contiguous access for a fixed record size of 128KB.

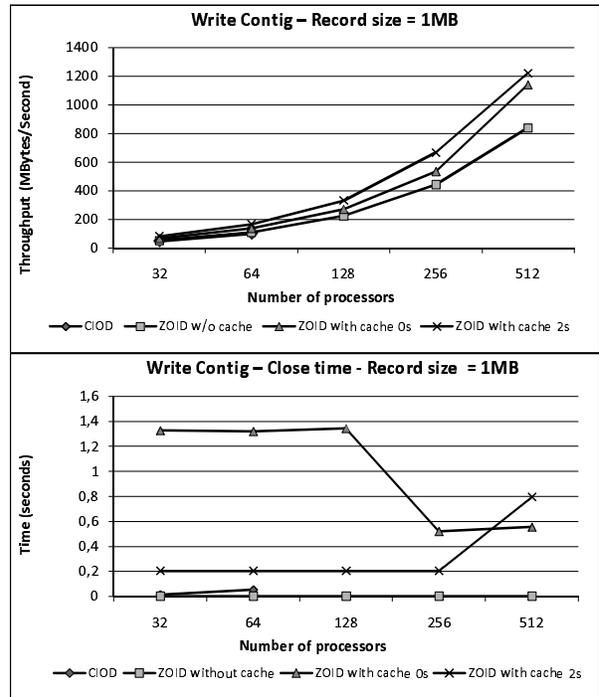
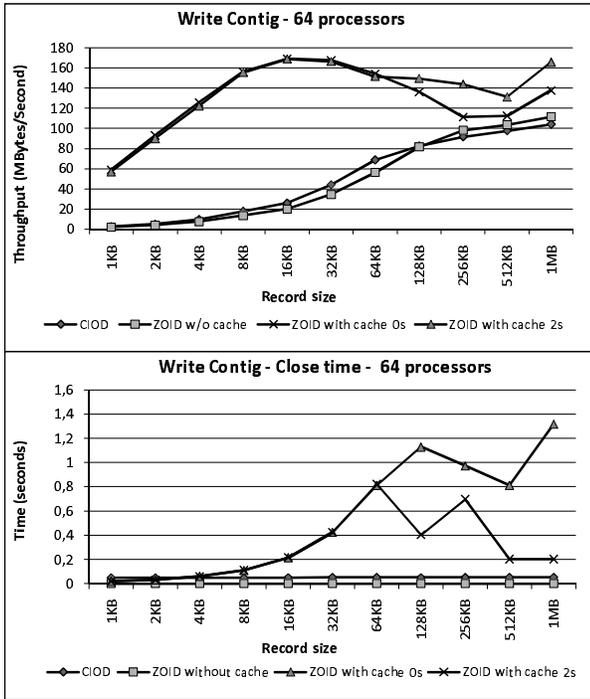
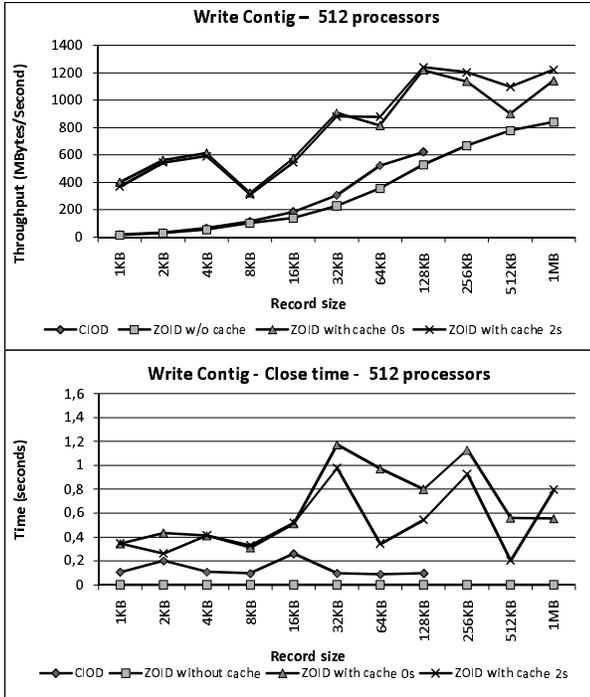


Figure 4. Performance of contiguous access for a fixed record size of 1MB.



**Figure 5. Performance of contiguous access for different record sizes and 64 processors.**



**Figure 6. Performance of contiguous access for different record sizes and 512 processors.**

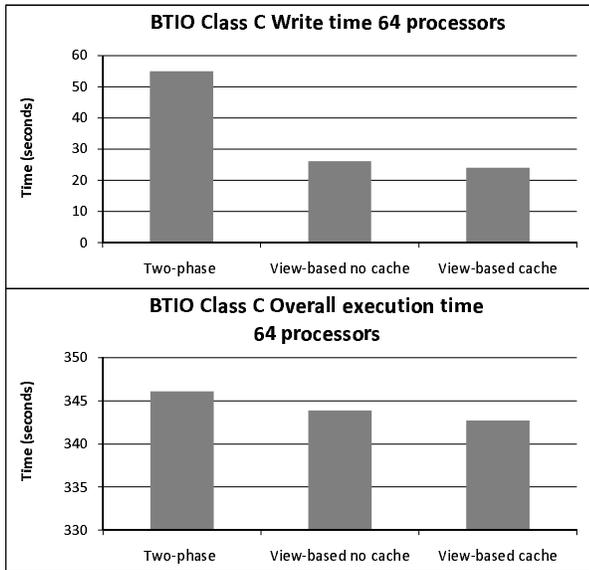
## 5.2 BTIO benchmark

NASA's BTIO benchmark [24] 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 verified for correctness. In this paper we report the results for the MPI implementation of the benchmark, which uses MPI-IO's collective I/O routines.

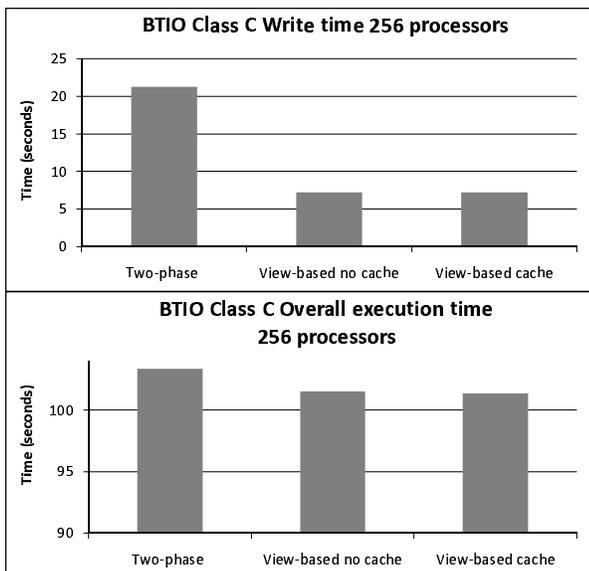
We have run the BTIO benchmark with three different variants of collective I/O techniques: two-phase I/O (the original collective I/O implementation from ROMIO), view-based I/O with no cache on the I/O node and view-based I/O with a cache of 128 MBytes on the I/O nodes. All collective I/O implementation were using the ZOIDFS module. All nodes acted as aggregators. Figures 7 and 8 show the total write time and total application time for the class C (writing around 4 GBytes of data). The upper row depicts the total write time and the lower row the overall application times. Figure 9 displays the write times of all of the 40 I/O steps. The file write time seen by the application is significantly lower for both cached and uncached cases of view-based I/O than for two-phase I/O. This is due to the caching at the compute nodes. However, the improvement in total application time is lower, because this compute node cache can not be flushed asynchronously due to the lack of threads support on Blue Gene/L architecture. For 64 processors the graph shows an increase in access times when the compute node cache becomes full. In the case of 256 processors, the file fits completely in the cache of the compute nodes and no change in performance was noted. It can be noticed also that two phase I/O has a higher cost in the first access and that, subsequently, flushes the collective buffer at each file access.

## 5.3 Discussion

Blue Gene/L presents two limitations that affect the efficiency of our solution. First, the compute nodes do not support multi-threading. Therefore, the data flushing from the collective buffers cannot be flushed asynchronously to the I/O nodes. Second, because the L1 caches are not coherent, the I/O nodes run all the ZOID threads on the same processor. Therefore, it is not possible to overlap communication and file transfer. However, our solution allows to overlap computation and file transfer: while the applications are



**Figure 7. BTIO class C times for 64 processors.**



**Figure 8. BTIO class C times for 256 processors.**

running, the file cache on the I/O node is asynchronously flushed to the file system over the Gigabit Ethernet network.

The limitations discussed above have been removed from the Blue Gene/P architecture: multi-threading is supported on the compute nodes and the L1 caches of the four cores are cache coherent. Therefore, we expect that our solution

will offer an additional performance benefit on Blue Gene/P systems.

## 6 Conclusion and future work

In this paper we have proposed a novel file I/O solution for Blue Gene systems based on MPI-IO and ZeptoOS. Our solution is based on a multi-tier cache strategy and an asynchronous data staging strategy that hides the latency of data transfer between cache tiers. We have shown that file system latency hiding may provide parallel applications a significant performance benefit and scalability.

We have ported the solution presented on this paper to Blue Gene/P systems and we are currently performing experimental evaluations. We are also implementing prefetching techniques for both I/O and compute node caches. Initial evaluation demonstrate significant performance improvements.

## References

- [1] J. G. Blas, F. Isaila, D. E. Singh, and J. Carretero. View-based collective I/O for MPI-IO. In *CCGRID '08: Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid*, 2008.
- [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] 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.
- [4] <http://www.unix.mcs.anl.gov/zeptoos/>. *ZeptoOs Project.*, 2008.
- [5] <http://www.top500.org>. *Top 500 list*.
- [6] 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>.
- [7] F. Isaila, J. G. Blas, J. Carretero, W.-K. Liao, and A. Choudhary. AHPIOS: An MPI-based ad-hoc parallel I/O system. In *Proceedings of IEEE ICPADS*, 2008.
- [8] F. Isaila, G. Malpohl, V. Oлару, 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.
- [9] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman. ZOID: I/O-forwarding infrastructure for petascale architectures. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 153–162, New York, NY, USA, 2008. ACM.

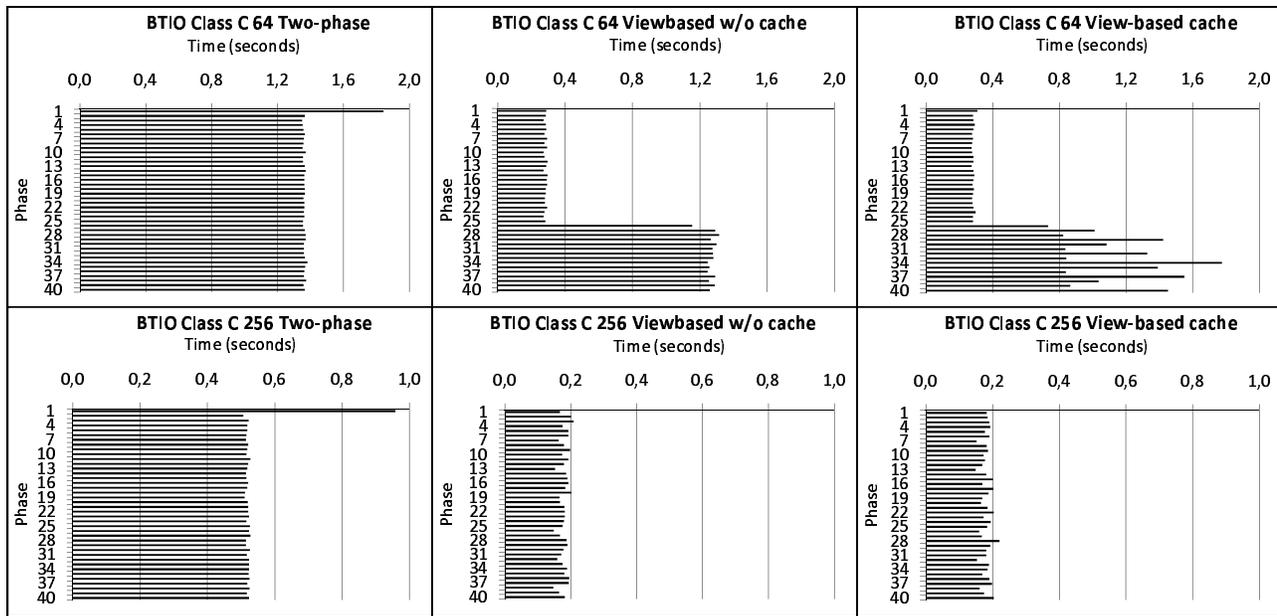


Figure 9. Times for all steps of BT-IO class C.

- [10] W. keng Liao, K. Coloma, A. Choudhary, L. Ward, E. Ruszel, 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.
- [11] 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.
- [12] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation*, 1994.
- [13] W. Ligon and R. Ross. An Overview of the Parallel Virtual File System. In *Proceedings of the Extreme Linux Workshop*, June 1999.
- [14] Message Passing Interface Forum. *MPI2: Extensions to the Message Passing Interface*, 1997.
- [15] J. Moreira, M. Brutman, n. José Casta T. Engelsiepen, M. Giampapa, T. Gooding, R. Haskin, T. Inglett, D. Lieber, P. McCarthy, M. Mundy, J. Parker, and B. Wallenfelt. Designing a highly-scalable operating system: the blue gene/l story. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 118, New York, NY, USA, 2006. ACM.
- [16] 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.
- [17] Y. H. Sahoo, R. Howson, and et all. High performance file i/o for the blue gene/l supercomputer. *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 187–196, 11-15 Feb. 2006.
- [18] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of FAST*, 2002.
- [19] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*.
- [20] R. Thakur, W. Gropp, and E. Lusk. An abstract device interface for implementing portable parallel-I/O interfaces.
- [21] 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.
- [22] 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.
- [23] R. Thakur, W. Gropp, and E. Lusk. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, Jan. 2002.
- [24] 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.
- [25] W. Yu and J. Vetter. Parcoll: Partitioned collective i/o on the cray xt. *Parallel Processing, International Conference on*, 0:562–569, 2008.
- [26] W. Yu, J. S. Vetter, and R. S. Canon. Opal: An open-source mpi-io library over cray xt. In *SNAPI '07: Proceedings of*

*the Fourth International Workshop on Storage Network Architecture and Parallel I/Os*, pages 41–46, Washington, DC, USA, 2007. IEEE Computer Society.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.