High Performance File I/O for The Blue Gene/L Supercomputer

H. Yu, R. K. Sahoo, C. Howson, G. Almási, J. G. Castaños, M. Gupta
IBM T. J. Watson Research Ctr, Yorktown Hts, NY
{*yuh,rsahoo,chowson,gheorghe,castanos,mgupta*}*@us.ibm.com*

J. E. Moreira, J. J. Parker
IBM System & Tech. Group, Rochester, MN
{*jmoreira,jjparker*}*@us.ibm.com*

T. E. Engelsiepen
IBM Almaden Research Ctr, San Jose, CA
engelspn@almaden.ibm.com

R. B. Ross, R. Thakur, R. Latham, W. D. Gropp
MCS, Argonne Nat'l Lab., Argonne, IL
{*rross,thakur,robl,gropp*}*@mcs.anl.gov*

Abstract

Parallel I/O plays a crucial role for most data-intensive applications running on massively parallel systems like Blue Gene/L that provides the promise of delivering enormous computational capability. We designed and implemented a highly scalable parallel file I/O architecture for Blue Gene/L, which leverages the benefit of the hierarchical and functional partitioning design of the system software with separate computational and I/O cores. The architecture exploits the scalability aspect of GPFS (General Parallel File System) at the backend, while using MPI I/O as an interface between the application I/O and the file system. We demonstrate the impact of our high performance I/O solution for Blue Gene/L with a comprehensive evaluation that consists of a number of widely used parallel I/O benchmarks and I/O intensive applications. Our design and implementation is not only able to deliver at least one order of magnitude speed up in terms of I/O bandwidth for a real-scale application HOMME [7] (achieving aggregate bandwidth of 1.8 *GB/Sec and* 2.3 *GB/Sec for write and read accesses, respectively), but also supports high-level parallel I/O data interfaces such as parallel HDF5 and parallel NetCDF scaling up to a large number of processors.*

1. Introduction

In recent years, one of the major challenges for computational scientists is to deal with very large datasets while running massively parallel applications on supercomputers. While most computationally intensive challenges are handled by emerging massively parallel systems with thousands of processors (e.g. Blue Gene/L), data-intensive computing with scientific and non-scientific applications still continues to be a major area of interest due to the gap between computation and I/O speed. Crucial for useful performance in a high-performance computing environment is the seamless transfer of data between memory and a file system for large-scale parallel programs.

Blue Gene/L [8] is a new massively parallel computer developed by IBM in partnership with Lawrence Livermore National Laboratory (LLNL). The Blue Gene/L (BG/L) system exploits low power processors, system-on-a-chip integration and a highly scalable architecture putting together up to 65536 embedded dual-processor PowerPC nodes (700 MHz) with high speed interconnects. LLNL's 64K-node BG/L system will deliver up to 360 Teraflops of peak computing power upon completion. BG/L is currently ranked as the world's fastest supercomputer in Top500 list of supercomputers, and represents five of the top ten entities in the list. While impressive scaling results have been obtained up to 32768 nodes [11], there has been relatively few results published on I/O performance for data-intensive applications on BG/L or any other massively parallel system.

A scalable parallel I/O support mainly consists of high-performance file systems and effective parallel I/O application programming interfaces. There have been many efforts developing parallel file systems for supercomputers, such as GPFS (General Parallel File System) [20] for IBM SP systems as well as Linux clusters, PVFS [5] and Lustre [2] for Linux-based platforms. In this work, we leverage GPFS, which is widely used on many large-scale commercial and supercomputing systems for its scalability, stability as well as reliability. Representative parallel file I/O programming interfaces include POSIX I/O interface [16], MPI I/O [19], and high-level abstraction layers such as parallel

Hierarchical Data Format (HDF) [1] and parallel NetCDF (PnetCDF) [17]. Among them, MPI I/O is synonymous with parallel file I/O for scientific computing, because of its wide use and its base on MPI. In addition, MPI I/O supports for relatively rich file access patterns and operations, which allows aggressive optimizations to be integrated.

The primary goal of the paper is to demonstrate the design, integration, and implementation of a hierarchical scalable file I/O architecture for BG/L. Our parallel file I/O solution consists of GPFS at the backend, while the I/O interface is handled through an optimized implementation of MPI I/O. We came across a number of BG/L specific hardware and software requirements while implementing MPI I/O (an optimized port of ROMIO [22]). In particular we evaluate the role of optimization of collective I/O primitives in communication phase, and file-domain partitioning within our implementation. We also characterize the scalability and performance of our BG/L file I/O solution not only for a list of I/O intensive benchmarks, but also real-scale applications as well as high-level libraries.

This paper makes the following contributions:

- We present the design and integration of a hierarchical parallel file I/O architecture for BG/L supercomputer, which delivers I/O performance scaling beyond conventional cluster-based parallel systems. We describe optimizations that contribute to the scalability.
- Quantitatively, we show the need to use MPI I/O collective operations for the scalability of parallel file I/O encountered frequently in real-world applications, usually with non-contiguous and irregular access patterns.
- We report the best bandwidth speedups ever achieved (to the best of our knowledge) for a number of commonly used I/O benchmarks, including an important I/O intensive application, HOMME from NCAR [7].
- Our file I/O solution provides efficient support for well-known high-level parallel file I/O interfaces, parallel HDF5 and parallel NetCDF. We present the first set of results demonstrating the scaling of programs written on these interfaces to a large number of processors.

The rest of the paper is organized as follows. Sec. 2 gives an overview of BG/L architecture from a parallel file I/O perspective. Sec. 3 presents the design of a parallel I/O solution followed by the optimizations carried out for scaling application-level parallel I/O. Sec. 4 describes issues related to our implementation of MPI I/O. Sec. 5 presents a thorough evaluation for the scaling performance of our solution against widely used parallel I/O benchmarks and a real-world application. Finally, we conclude the paper in Sec. 6.

2. Blue Gene/L: A Parallel I/O Perspective

BG/L uses a scalable architecture based on low power embedded processor, and integration of powerful networks, using system-on-a-chip technology [8]. It uses a hierarchical system software architecture to achieve unprecedented levels of scalability [10]. In this section we describe some of these features we exploit to achieve highly scalable parallel I/O and create an attractive platform for data-intensive computation.

The BG/L core system consists of compute and I/O nodes with compute nodes viewed as computational engines attached to I/O nodes. The compute nodes and I/O nodes are organized into *processing sets* (*pset*), each of which contains one I/O node and a fixed number of compute nodes. Running on simplified embedded Linux, the I/O nodes represent the core system to the outside world as a cluster. In each pset, program control and I/O are accomplished via messages passed among its I/O node and compute nodes over a *collective* network. The I/O related communication among compute nodes and the I/O node in a pset is with little disturbance. BG/L compute nodes and I/O nodes are built out of the same chips, with only differences in terms of packaging and enabled networks. While, the compute node runs a unique, light-weight compute node kernel (CNK), the I/O node provides file service via Linux VFS. I/O related system calls trapped in CNK are shipped to the corresponding I/O node and processed by a console daemon CIOD, thus a POSIX-like I/O interface is supported for user-level compute processes. The separation and cooperation accomplished via the simple function shipping off-loads non-computation related services to the I/O nodes to keep a noise-free state for the compute nodes while achieving superior scalability [11]. In addition, the separation of I/O nodes and compute nodes, together with organizing them into balanced psets and providing almost dedicated communication channels for I/O operations, essentially provide the capability for scalable I/O.

Nevertheless, the scalable I/O capability does not necessarily lead to scalable application-level file I/O. Usually, such applications involve accessing a single file concurrently and with non-contiguous and/or irregular access patterns. While, the rest of the paper describes our effort on meeting the challenge, we first discuss a couple of BG/L features that we have particularly utilized.

The BG/L torus network, connecting the compute nodes, is the primary network for inter-processes communications for its high speed. Specifically, the bandwidth of a single link is close to its designed peak, 175 MB/sec in each direction. For the LLNL machine, the 65,536 compute nodes are organized into a $64 \times 32 \times 32$ three-dimensional torus. At 1.4 Gb/s per torus link, the unidirectional bisection bandwidth of the system will be 360 GB/s. The high bandwidth and high speed of the torus network provides the capability to move large amount of data across the compute nodes.

BG/L-MPI [9] has successfully exploited the rich features of BG/L in terms of the network topology, special purpose network hardware, and architectural compromises. While BG/L-MPI is originally ported from MPICH2 [3], its collective routines have demonstrated superior performance comparing to the original implementation and is close to the peak capabilities of the networks and processors. From an I/O perspective, a scalable implementation of BG/L-MPI provides an effective mean to optimize for concurrent and coordinated file accesses from a large number of processes.

In the following section, we will describe how we leverage BG/L's features such as its capability for scalable I/O, its high-performance interconnects, and its MPI implementation to deliver application-level scalable I/O.

3. A Scalable Parallel I/O Design for BG/L

In this section, we first present our parallel file system solution for BG/L: GPFS. Then, we describe our design steps and consideration to provide MPI I/O support to complement the GPFS-based solution and eventually to meet the I/O demands of applications running on BG/L. Additional implementation details will be discussed in Sec. 4.

3.1. GPFS-Based Solution

GPFS is IBM's parallel, shared-disk file system supporting both AIX- and Linux-based systems [20]. It allows parallel applications' concurrent access to the same files or different files, from nodes that mount the file system. In its latest release, GPFS 2.3 allows users to share files across clusters, which improves the system capacity as well as simplifies system administration. So far, the largest tested GPFS cluster contains 1170 Linux nodes [6], which is more than the I/O nodes of the largest BG/L systems installed so far (e.g., the LLNL 64-rack system has 1024 I/O nodes and the ASTRON [4] 6-rack system has 768 I/O nodes).

Our solution for integrating GPFS and BG/L consists of a three tier architecture (Fig. 1). The first tier of the architecture consists of I/O nodes as GPFS clients, whereas the second tier is an array of NSD (network shared disks) servers. The third tier consists of the storage area network (SAN) fabric and the actual disks. The interconnect between the I/O nodes and the NSD servers are Ethernet, while the connection between the second and third tier can be either fiber channel, fiber-channel switch or Ethernet (iSCSI). The choice of NSD servers, SAN fabric and storage devices depend on the customer requirements. This solution has been fully implemented and the performance evaluation shows that it had successfully explored and utilized the enormous potential I/O bandwidth of BG/L. Later on, we will show that the aggregated read/write at BG/L compute nodes reach 80% and 60% of that the underneath file system can deliver.

GPFS achieves its high throughput based on techniques such as large-block based disk striping, client-side caching,

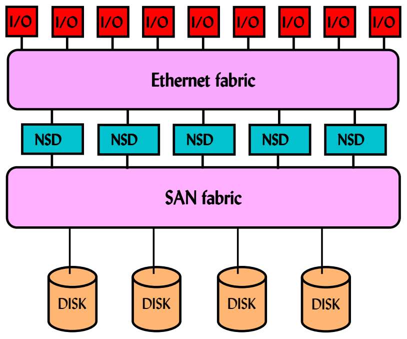

Figure 1. GPFS layout for BG/L

prefetching, and write-behind [20]. In addition, it supports file consistency using a sophisticated distributed byte-range file-locking technique. GPFS has managed to limit the performance side-effect of the file locking operation with an aggressive optimization for block-level data accesses.

Overall, GPFS is highly optimized for large-chunk I/O operations with regular access patterns (contiguous or regularly strided). However, its performance for small-chunk, non-contiguous I/O operations with irregular access patterns (non-constant strided) is less optimized. Particularly, concurrent accesses (from distinct processes) to different file regions in the same GPFS block introduce additional file system activities associated to its file locking mechanism, which can hurt performance. To complement GPFS, we use MPI I/O for I/O operations in scientific applications.

3.2. MPI I/O Design Goals

MPI I/O is the parallel I/O interface specified in the MPI-2 standard [19]. Because of its support for a much richer file access patterns and operations, MPI I/O is better suited for MPI based parallel programs than POSIX I/O interface. Particularly, MPI I/O includes a class of collective I/O operations (enabling a group of processes to access a common file in a coordinated fashion), which provide better flexibility for MPI I/O implementations to optimize I/O, by aggregating/distributing I/O operations transparently.

The general technique for implementing MPI I/O collective operations is by means of providing *collective buffering* [21]. Collective buffering is based on a common experience that the aggregated inter-processor communication speeds are significantly higher than I/O speeds. It rearranges and aggregates data in memory prior to writing to files to reduce the number of disk accesses. For scientific applications (typically with non-contiguous and irregular file access patterns), collective buffering is effective for achieving scalable I/O, particularly for systems without efficient support of asynchronous communication primitive, and exploiting massive parallelism like BG/L. Therefore, we set our design goal of MPI I/O for BG/L as to maximize the performance of MPI I/O collective operations.

To facilitate the design goals, we started our work on MPI I/O from ROMIO [21], an implementation of MPI I/O from Argonne National Laboratory. The primary reason for choosing ROMIO is for it's fully functional MPI I/O implementation (except *data representation*). Further, ROMIO integrated the collective buffering technique into its implementation of collective I/O operations, which allows us to concentrate on performance optimizations. In the following sub-sections, we elaborate our design decisions made in providing a scalable implementation of MPI I/O collective operations for BG/L configured with GPFS.

3.3. The *pset* Organization

I/O systems by nature are much more efficient for contiguous disk accesses than for non-contiguous, or irregular accesses. Particularly, GPFS is highly optimized for large-chunk, regular (contiguous or regularly strided) I/O accesses [20]. Therefore, it is better for the I/O requests from compute nodes to be contiguous.

As stated previously, pset organization of compute and I/O nodes plays a key role for BG/L I/O performance. Exploiting the collective buffering technique, MPI I/O collective operations provide opportunities for the pset structure of BG/L to be communicated, and an optimized file access pattern can be reached. The specific motivations for using pset and collective buffering approach are two folds. First, the best observed I/O performance of a BG/L partition containing multiple psets is often obtained when the I/O load is balanced across all the I/O nodes. Second, we have observed that for the case of a relatively large compute node to I/O node ratio (e.g. 64:1 on LLNL system), the I/O performance of a pset reaches the peak when 8-16 compute nodes perform I/O concurrently, not all the compute nodes.

The implementation of collective buffering in ROMIO (referred as *two-phase I/O*) distinguishes two interleaved phases: an inter-process data exchange phase and an I/O phase [21]. The two-phase I/O first selects a set of processes as I/O aggregators, which partitions the I/O responsibilities for a file. Based on this file partitioning, the in-memory data exchange phase routes the data among all participating processes and the I/O aggregators. In the I/O phase, the I/O aggregators issue read or write system calls to access file data.

In our design, the I/O aggregators are chosen in such a way that they are evenly distributed across the participating psets. Each BG/L node has a personality structure which keeps its run-time configuration data [10]. We utilize the pset configuration information. First, each compute node collects its pset ID, MPI rank. Then, the information of all compute nodes is gathered onto each compute node. Finally, each compute node generates a list of I/O aggregators so that they are distributed evenly across the psets, and the I/O aggregators from the same pset are contiguous in the list. When one MPI I/O collective routine is called, the collective file access region is computed and the I/O responsibility is distributed across the selected I/O aggregators. For a collective I/O operation with small data amount, a subset of the pre-selected I/O aggregators are used.

Here, the BG/L I/O nodes are not used for I/O aggregation. First, with the same ASIC as the compute nodes, I/O nodes have limited processing power for relative complicated aggregation tasks. Second, the possible communication channels among the I/O nodes are Ethernet and BG/L collective network. Although the collective network can be configured to contain multiple I/O nodes, the bisection bandwidth of the collective network is bounded by 350 MB/sec. The I/O nodes can potentially talk to each other over Ethernet. However, additional communication among I/O nodes through Ethernet may interfere with the file system client-side operations running on the I/O nodes. A comparison of file access operations on compute nodes against that from I/O nodes indicated little performance difference. Hence, having the compute nodes as I/O aggregators potentially provides more flexibility for dealing different BG/L configurations that have different compute nodes to I/O nodes ratio, avoiding a potential bottleneck at the I/O nodes.

3.4. File Domain Partitioning

As mentioned, GPFS block-level file accesses are highly optimized and scalable. Therefore, if a GPFS client node (i.e. BG/L I/O node in our design) only issues data-access requests having large size and aligning to GPFS block boundaries, the scalability of GPFS will be preserved, while successfully meeting the MPI I/O collective operation requirements. ROMIO assigns the I/O responsibilities for a file across the I/O aggregators with a balanced partitioning of the file region defined by the first and last file offsets of the collective operation. It has been reported in [14] that the simple file-partitioning method has various drawbacks and there are multiple ways for its optimization. For the case of GPFS, the main problem of the default file-partitioning method is that it can generate I/O operations only accessing part of GPFS blocks, which will trigger additional file system activities related to file locking.

To investigate the effect of associating the I/O aggregators with file regions align to the GPFS blocks, we wrote a synthetic program that partitions the file domain across processes in an absolutely balanced manner or with additional consideration to align with GPFS blocks. We call the two file partitioning methods as *balanced* and *aligned* file partitioning, respectively. After establish a mapping from file regions to processes, each process performs one contiguous write operation to its designated file region. Here, when ap-

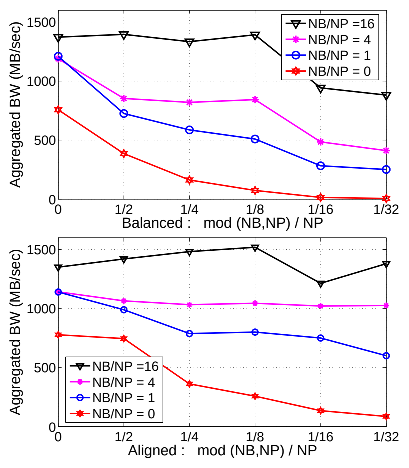

Figure 2. Balanced vs Aligned file partitioning

plying the balanced file partitioning, the neighboring I/O aggregators will compete for the lock of a single GPFS block. On the other hand, when applying aligned file partitioning, load imbalance could be a problem. This way, we will find which hurts performance more between concurrent accesses of a GPFS block and the imbalanced load.

We ran the experiment on an I/O rich BG/L system using 256 compute nodes and 32 I/O nodes (details of the platform is given in Sec. 5.1). Fig. 2 shows the results of the two mapping methods. Here, *NB* represents the total number of GPFS blocks and *NP* represents the number of processes. A non-zero value of $X = mod(NB,NP)/NP$ indicates that $R = X \times NP$ file blocks will be shared by multiple processes for the balanced partitioning case; for the aligned partitioning case, there will be R processes each of which accesses one additional file block comparing to the other processes. The results indicate that aligned file partitioning outperforms balanced file partitioning and its I/O performance stays flat for most of the cases, except when some of the I/O nodes have no work to do.

Based on the observation, we augmented ROMIO's default file partitioning method to have each I/O aggregator's file domain aligned to GPFS block in both size and offset. The GPFS block size can be reset using an environment variable at run-time, with a pre-set value as 1MB (the maximal file block size currently supported in GPFS). Later on (Sec. 5), we will show that the file-domain partitioning optimization (referred as *Aligned*) has contributed to significant performance improvement.

3.5. Communication Phase Optimizations

For the two phases of MPI I/O collective operations, the performance of file accesses issued by the I/O aggregators are bounded by the POSIX I/O performance on BG/L. However, for certain I/O access patterns (e.g. all processes reading the same data from a file), the inter-process data exchange phase may dominate the overall performance. To address the issue of the communication phase of MPI I/O collective operations, we rely on the BG/L MPI implementation [9] as it has successfully explored and utilized the rich network features of BG/L machine. We tuned the communication phase of MPI I/O collective operations to choose the best performing communication method among BG/L MPI routines. Here, we highlight two optimizations whose benefits are clearly demonstrated in our experiments.

3.5.1. Inter-Process Data Exchange ROMIO has implemented the communication phase of MPI I/O collective operations by using MPI send/receive functions, which can be further optimized on BG/L. As reported in [9], on BG/L, the optimized collective communication functions usually achieve much higher performance than the point-to-point communication functions. Particularly, the optimized MPI_Alltoall and MPI_Alltoallv can utilize up to 98% of the peak bandwidth of the underneath torus/mesh network for long messages, and their performance scales with the number of compute nodes. Therefore, we have replaced the use of the point-to-point functions in this phase with MPI_Alltoallv. In our experiments, we refer to this specific optimization as *Alltoall*.

3.5.2. Access Range Information Exchange Prior to the interleaved communication and I/O phases, the two-phase I/O implementation schedules the communication among the I/O aggregators and the processors issuing I/O operations. Here, all processes exchange the information about their I/O requests. This was originally implemented via MPI_Allgather operations so that each process will have the file range information of all the I/O requests. We found that these operations take up to 30% of the time of the collective write operations in a few runs of the BTIO benchmark (see Sec. 5.1 for details). On BG/L, the MPI_Allgather operation is implemented by performing P MPI_Bcast operations, where P is the number of processes in the communicator. On BlueGen/L, for very short messages, MPI_Bcast has high overhead. We replaced the MPI_Allgather with an MPI_Allreduce, which performs much better for short and medium sized messages to gain the MPI benefits. The optimization is referred as *Allreduce* in our experiments.

4. MPI I/O Implementation Issues

Having been ported to a large number of parallel systems [22], ROMIO's portability mainly relies on two features. First, it uses MPI, which allows ROMIO to be ported to most parallel computing platforms that support MPI. Secondly, ROMIO has an intermediate interface called ADIO (Abstract Device Interface for I/O), which hides the implementation details for different file systems and makes it a

Program	Description	From	Used by
IOR	General I/O benchmarking	LLNL	LLNL
NAS BT-IO	NAS BT benchmark with checkpointing	NASA Ames Research Ctr	[21] [14] [25]
coll_perf	ROMIO collective I/O performance test	ANL	[13]
FLASH io_bench	Benchmark for the I/O requirements of FLASH2	U Chicago	[17] [13] [14]
HOMME	High Order Method Modeling Environment	NCAR	[7]

Table 1. Benchmarking programs and applications

relatively easy task for porting and performance tuning on a specific system. To port ROMIO onto a specific system or file system, one only needs to implement a small set of system-dependent functions under ADIO. Most of our development for BG/L MPI I/O that presented in this paper are under the ROMIO ADIO layer. Concentrating on optimizing MPI I/O collective operations, we have overridden ADIO's default implementation for this class of operations.

To provide a complete MPI I/O support, we have augmented CIOD with *fcntl* byte-range file locking functionality, facilitating MPI I/O atomic mode. On BG/L, fcntl byte-range file locking is different from most of the file I/O system calls, as it cannot be realized by simple function forwarding. Specifically, when a file lock is trapped at the compute node kernel and forwarded to the I/O node, the CIOD process is the one that actually issues fcntl for locking a file region. Since CIOD is a user process, the Linux kernel on the I/O node treats all the file locking requests (though from different compute-nodes) as if they are from a single Linux process. This causes a problem when two compute nodes in the same pset (i.e., their I/O requests go to the same I/O node) compete for a lock over the same file region. As a result, both compute-processes will gain access to the file region and the MPI I/O atomic semantic is violated.

To support correct fcntl file locking functionality, we keep the file locking information in CIOD for all the files opened by the computing-processes in the pset. That is, for each opened file, CIOD keeps two lists of byte-range pairs. One list keeps the regions that are locked by compute nodes in the pset, while another list keeps the pending file locking requests. Recall that the protocol between BG/L compute and I/O nodes is blocking, and the fcntl is no exception. For the blocked call of fcntl (e.g. *F_SETLKW*), we do not want to block the I/O node. In our implementation, CIOD translates all blocked fcntl file locking requests from compute nodes into repeated non-blocking calls. Specifically, in CIOD there is a loop repeatedly querying the devices (tree, Ethernet), and for the denied non-blocking fcntl file locking requests, CIOD re-tries the file-locking requests for every few iterations. This process repeats until a fcntl file locking requests is satisfied. The rest of the implementation is similar to the file locking implementation in Linux kernel [12].

5. Performance Evaluation

In this section we present the evaluation of our scalable parallel I/O solution against a collection of widely used benchmarks and a real application utilizing parallel I/O. We will demonstrate scalable performance resulted from our solution showing that it matches the extraordinary computational power and the corresponding file I/O needs of scientific applications running on BG/L.

5.1. Experiment Setup

We used a two-rack I/O rich BG/L system, containing 2048 compute nodes and 256 I/O nodes. The system has two 512 compute-node partition and one 1024 compute-node partition that mount GPFS on their I/O nodes, and no smaller partitions are configured. To demonstrate scalability, we used mapping files [9] that assigns contiguous MPI ranks to the compute nodes in the same *pset* to keep the compute nodes to I/O nodes ratio as 8:1, when using less than 512 nodes.

The attached GPFS file system contains 32 NSD servers (running on x335 Linux PCs). The system has 8 DS4300 storage manager (aka FAStT600), with each has 4 Fiber Channel connections to 4 NSD servers. The 8 DS4300 together host 448 disks. On the other side, the NSD servers connect to the BG/L I/O nodes via Gigabit Ethernet. Although the aggregated disk bandwidth can be as high as 10 to 20 GB/sec, the performance of the configuration is bounded by the bandwidth from the I/O nodes to the NSD servers (the possible peak bandwidth of the 32 Gb Ethernet links is 4 GB/sec). The GPFS block size is set to 1 MB to maximize the system throughput. The page pool size on I/O nodes is 192 MB (GPFS page pool is used for client-side caching). Currently, the maximum performance for accessing a single file from BG/L compute nodes is about 2.0 GB/sec for write and 2.6 GB/sec for read. The gap between the observed I/O rates from BG/L compute nodes and available bandwidth of the specific configuration (4 GB/sec) are from protocol overheads of TCP, and the data transfer among BG/L compute and I/O nodes.

Table 1 lists five benchmarks and applications we used here for our experiments.

IOR is a parallel file system test code developed at LLNL. It performs parallel writes and reads to/from file(s) using MPI I/O and POSIX operations. It can be configured to access a single file or each process accessing a separate file; use collective/non-collective operations; and perform contiguous or strided accesses, etc. IOR (version 2.8.1) has been used to measure the parallel file system performance for BG/L. Here, we compare the performance of IOR using MPI I/O against using POSIX operations to show that our MPI I/O implementation delivers comparable performance for parallel I/O operations with regular (contiguous or strided) access patterns. We use the rest of the programs to demonstrate the advantage of MPI I/O for I/O with non-contiguous and irregular access patterns. For eliminating possible client-side caching effects at the I/O nodes, we configured each process to do 384 write/read operations, each of which access 1 MB of data.

NAS BT-IO [24], an extension of NAS BT benchmark, simulates the I/O requirement of BT. BT-IO distributes multiple Cartesian subsets of the global dataset to processes using diagonal multi-partitioning domain decompositions. The in-core data structure are three-dimensional arrays and are written to a file periodically. The file access pattern of each process is non-contiguous with non-constant strides. NAS BT-IO's file access pattern is typical among scientific applications, which has been frequently used for performance evaluation of parallel file I/O [21, 14, 25]. BT-IO has three implementations for its file I/O. The *full-mpiio* carries out the I/O using MPI I/O collective routines, while the *simple-mpiio* does the I/O using MPI I/O independent routines, and the *fortran-io* has FORTRAN iterative write statements in a multi-level loop nest. Here, we used NAS input class C (the underneath process topologies are squares). To keep the per-process load constant across runs, we changed the default *problem_size* parameter to have each process access about 3 MB data in each collective I/O call.

coll_perf is a synthetic benchmark in ROMIO [23]. It collectively writes and reads a three-dimensional, block-distributed array to/from a file in a non-contiguous manner. To keep the work (data/file size) of each processor constant, each process accesses 256^3 integer elements, resulting an average I/O of 64 MB per-process.

FLASH I/O benchmark [26] simulates the I/O pattern of the FLASH2 code [15], a parallel hydrodynamics code that simulates astrophysical thermonuclear flashes in two or three dimensions by solving the compressible Euler equations on a block-structured adaptive mesh. The benchmark program is distributed together with the full FLASH2 application and its I/O routines are identical to the routines used by FLASH2. So the performance demonstrated by the benchmark would closely reflect the performance of the full application's I/O performance. The FLASH I/O benchmark recreates the primary data structures in the FLASH2 code and produces a checkpoint file, a plotfile for centered data, and a plotfile for corner data. Same as FLASH2 code,

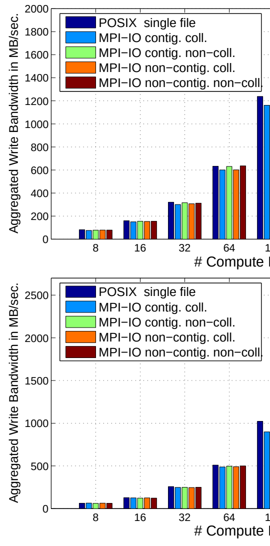

Figure 3. Performance baseline (IOR W/R)

the original version of FLASH I/O benchmark uses parallel HDF5 [1]. In short, the in-core data structure of FLASH I/O benchmark is a block-distributed three-dimensional array, and the file structure is a distribution of the three-dimensional array along the Z-dimension. In its latest distribution, FLASH I/O benchmark includes a version that uses PnetCDF [17]. The benchmark measures the time spent on a collection of I/O and data manipulation operations and then computes the data rate based on these timings. To be consistent with the rest of the performance data shown in this paper, we have instrumented the HDF5 and PnetCDF libraries to time the MPI I/O file access routines (used inside the libraries). Because PnetCDF only uses collective file access routines, we did not collect independent I/O performance for FLASH I/O benchmark. We configured the benchmark for each process to access about 2.5 MB data in each MPI I/O collective operation.

5.2. Performance Baseline

The performance of application-level parallel I/O demonstrated in this paper is bounded by the POSIX I/O performance of our experiment platform, and our first experiment is to establish a performance baseline.

Fig. 3 shows the results from IOR. We ran various configurations of IOR. The *POSIX single file* configuration gives the I/O bandwidth obtained on the system when all processes accessing distinct regions of a common file. The results indicate that the performance scales well (averaging 100 MB per second for each BG/L I/O node) up to about 256 compute nodes (32 I/O nodes), and then flats out. The specific GPFS file system was set up for the purpose of development and the performance shown in Fig. 3 is quite close

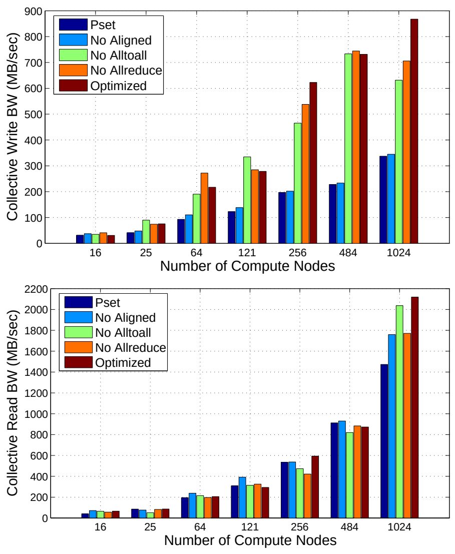

Figure 4. Optimization effects on BT-IO

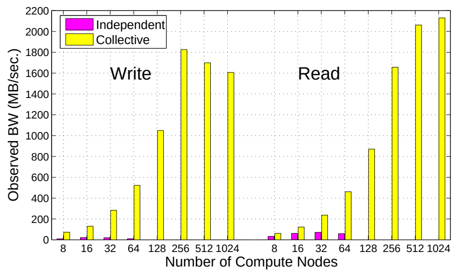

Figure 5. ROMIO coll_perf results

to its capability. We have run most of our experiments up to 1024 compute nodes, and some results show that the performance still grows when 1024 compute nodes are used. This is because MPI I/O performance is different from raw I/O performance. It reflects the MPI communication performance and how the cross-processor communication and the I/O activities interacts. For the MPI I/O related configurations, *contig.* indicates that the accesses from each process are contiguous; *non-contig.* indicates that the accesses are constantly strided; *coll.* indicates that collective I/O operations are used; *non-coll.* indicates that independent I/O operations are used. The corresponding results are fairly close to that of POSIX, which indicate that MPI I/O does not introduce significant overhead (from implementation, byte-range file locking) for simple file access patterns (i.e., having large contiguous chunks as 1 MB). The following sections will show that for irregular file access patterns exposed in benchmarks and applications, MPI I/O becomes the most effective mean for obtaining high-level I/O performance.

5.3. Effects of Optimizations

In this section, we evaluate the effects of different optimizations discussed in Sec. 3 using NAS BT-IO benchmark. We compare five configurations:

- *Optimized* applies all the optimizations from Sec. 3;
- *No Allreduce* applies *Optimized*, except the *Allreduce* optimization (Sec. 3.5.2);
- *No Alltoall* applies *Optimized*, except the *Alltoall* optimization (Sec. 3.5.1);
- *No Aligned* applies *Optimized*, except the *Aligned* optimization (Sec. 3.4).
- *Pset* only applies the optimization that integrates the *pset* structure into the selection of I/O aggregators (Sec. 3.3).

The configurations *No Allreducce*, *No Alltoall*, and *No Aligned* are to show the performance degradations from the best configuration (*Optimized*), when the specific optimizations are not applied.

Fig. 4 gives the results for BT-IO. It shows that using MPI I/O collective routines in general gives the best I/O performance when the number of processes increases. As shown in the graph, among the three optimizations, the aligned file domain partitioning gives the best performance improvement, i.e., while comparing to the best configuration, *No Aligned* gives the most performance degradation for write operations. This is consistent with our synthetic experiment in Sec. 3.4. For the other two optimizations (*Alltoall* and *Allreduce*), it shows that, when they are not applied, the performance degradations scale with the number of processors, which confirms that the optimizations result better scaling. Finally, there is at least three fold speedup for the write operations, when comparing the *Optimized* and the *Pset* configurations.

These optimizations have more significant effects on the write operations than for read, because GPFS' file byte-range locking is not optimized for small-chunk file write operations. In addition, the bandwidth numbers reported in Fig. 4 represent averaged numbers across all the instances of MPI I/O collective operations. We have noticed that there are performance variations across the instances, and we are investigating the issue.

5.4. Benchmark Results

In Fig. 5 and 6, we show performance obtained on two widely used benchmark programs. The results for coll_perf confirm that MPI I/O collective operations on BG/L can deliver scalable performance that is only bounded by the underlying system. Particularly, the poor performance of independent I/O operations states a strong recommendation to use MPI I/O collective routines. For the write results, there is a slight drop of the performance when more than 256 processes are used. We think the performance drop was due to

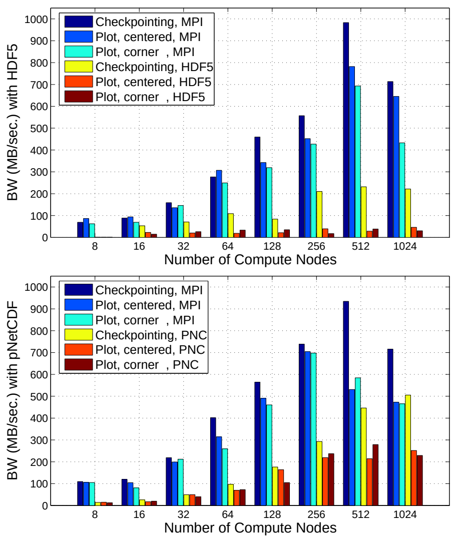

Figure 6. FLASH I/O Benchmark results

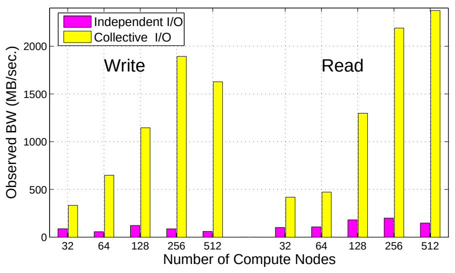

Figure 7. HOMME I/O results

the increasing latency of the inter-process data exchange of the two-phase I/O when the performance of the I/O phase reached the system's capability, and we are working on a thorough investigation of this behavior.

Fig. 6 shows the performance of the FLASH I/O benchmark. We present two sets of results: MPI and HDF5; MPI and PNC (PnetCDF). The legends marked MPI are obtained from our instrumentation inside HDF5 and PnetCDF that measure the wall-time of MPI I/O collective read/write operations. The legends marked with HDF5 or PNC are simply based on the wall-time of the application-level I/O functions. This results show that at the observation level of high-level parallel I/O library routines, our solution delivers up to 240 MB/sec with HDF5, and 500 MB/sec with PnetCDF. The results are about several folds better than similar experiments presented previously for cluster-based parallel file I/O solutions [17]. Here, because the bandwidth numbers are averaged across multiple MPI I/O calls, the results do not match up with those of IOR. Besides the effects similar to coll_perf, the results show that the best performance is reached when using 512 processes, this is because that the data-size was small (averaged 2.5 MB per process per collective call) and hence the I/O phase could not fully explore available bandwidth of the system (e.g., disk accesses are not balanced across a large number of disks) for small scale runs. Nevertheless, the aggregate performance is significant. In addition to scalable MPI I/O performance, the results marked as *Checkpointing, HDF5/PNC* indicate that for the two MPI I/O based high-level interfaces, when observing at the level of their high-level I/O routines, the I/O bandwidth numbers scale up to 256/512 processes.

5.5. Application Results

Finally, we demonstrate the scalability of the file I/O for the HOMME (High-Order Multiscale Modeling Environment) code, a scalable and efficient spectral element based atmospheric dynamical code. The application has been running on BG/L and its scalability and performance (without the I/O part) has been demonstrated with up to 7776 processors [7]. The file I/O part of HOMME deals with checkpointing, restarting from checkpoints and dumping movie files. In addition, each process dumps a movie file where MPI I/O is not used. Since writing to separate files is not interesting from MPI I/O perspective, we did not configure HOMME input file for such operations. The checkpoint/restore part of HOMME is written using MPI I/O independent file access routines in a loop-nest. We have replaced the loop nest containing the MPI I/O independent operations with a MPI I/O collection operation and a file set-view call within the application.

We used an input of simulation on *Aquaplanet* with Emanuel physics [18]. To benchmark its I/O aspect, we have changed the input configuration of checkpoint frequency and simulation steps. The total amount of work was kept constant for all the runs. To evaluate the performance for both write and read, for each data point in the graphs, we invoked the application four times: an initial run with checkpoint only configuration followed by three runs with restart and checkpoint configuration. Each run has 20 time steps with one checkpointing for every 5 time steps. Each checkpoint operation generates a 500 MB file.

Fig. 7 shows the performance of the I/O part of HOMME. Here, we show the performance improvement associated with our modification of changing MPI independent I/O operations to collective operations. The results indicate that for the application, using MPI I/O collective read/write operations delivers I/O performance that is close to the baseline. Overall, we achieved 10-15 folds speedups for the time consuming I/O part of HOMME compared to its original implementation (using MPI I/O independent operations).

Program	BW in MB/sec.	
	Write	**Read**
LLNL IOR	1,970	2,600
ROMIO collective I/O performance test	1,400	1,800
NAS BT benchmark with checkpointing	870	2,100
FLASH2 I/O benchmark with HDF5	963	
FLASH2 I/O benchmark with PnetCDF	940	
HOMME application from NCAR	1,800	2,300

Table 2. Results summary

6. Conclusions and Future Work

Productivity and performance of today's supercomputing systems have been limited by the I/O bottlenecks due to the inability of existing software layers to scale to the desired level. While impressive computational scaling results have been obtained on massively parallel systems like Blue Gene/L, relatively very few scalable I/O performance results exist for data-intensive applications on such systems.

We address the challenge with a highly scalable file I/O design and implementation which would deliver unprecedented levels of I/O performance for BG/L. By leveraging the benefit of functional partitioning and hierarchical structure of Blue Gene/L system software, our parallel file I/O design, is able to provide scalable file I/O bandwidth far beyond the level of any conventional cluster-based supercomputing systems. The design exploits the scalability of GPFS at the backend and provides effective MPI I/O support (collective operations in particular) as the API for application-level I/O requirements.

We evaluated our design and implementation on an 1-rack Blue Gene/L system against a number of popular benchmarks as well as HOMME [7], a real application. Highlights of our results (the best aggregated bandwidth of MPI I/O collective file access operations running on up to 1024 BG/L compute nodes) are summarized in Table 2. It shows the best bandwidth speedups ever achieved (to the best of our knowledge) for these benchmarks and applications. In addition, for the first time, we demonstrate the scaling for popular parallel I/O interfaces such as parallel HDF5 and parallel NetCDF. to a large number of processors.

While the results presented in this paper are obtained on an I/O rich BG/L system with 1024 compute nodes, our solution is not limited to this configuration. We are investigating the solution on bigger systems. As a continuation to our on-going research we would like to expand API solutions beyond the scope of MPI I/O. We are also investigating the possibility of optimizing across-process data exchange patterns, exploring effective file domain distribution, and addressing potential meta-data scalability.

References

[1] HDF5 home page. URL: *http:// hdf.ncsa.uiuc.edu/ HDF5*.

[2] Lustre scalable storage. URL: *http:// www.clusterfs.com*.

[3] The MPICH and MPICH2 homepage. URL: *http:// www-unix.mcs.anl.gov/ mpi/ mpich*.

[4] LOFAR BlueGene/L Workshop. URL: *http:// www.lofar.org/ BlueGene/*, 2004.

[5] Parallel Virtual File System 2 (PVFS2). URL: *http:// www.pvfs.org/ pvfs2*, 2004.

[6] *GPFS for Linux, FAQ*. URL: *http:// publib.boulder.ibm.com/ clresctr/ windows/ public/ gpfsbooks.html*, 2005.

[7] HOMME on the IBM BlueGene/L. URL: *http:// www.homme.ucar.edu*, 2005.

[8] N. R. Adiga et al. An overview of the BlueGene/L supercomputer. In *SC'02*.

[9] G. Almási et al. Optimization of MPI collective communication on BlueGene/L systems. In *ICS'05*.

[10] G. Almási et al. An overview of the BlueGene/L system software organization. In *Euro-Par'03*.

[11] G. Almási et al. Scaling physics and material science applications on a massively parallel BG/L system. In *ICS'05*.

[12] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly and Associated, Inc., 2002.

[13] A. Ching et al. Noncontiguous I/O Accesses Through MPI-IO. In *CCGRID'03*.

[14] K. Coloma et al. Scalable high-level caching for parallel I/O. In *IPDPS'04*.

[15] B. Fryxell et al. FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophysical Journal Supplement*, 2000.

[16] B. Gallmeister. *POSIX.4*. O'Reilly and Assoc., Inc., 1994.

[17] J. Li et al. Parallel netCDF: A High-Performance Scientific I/O Interface. In *SC'03*.

[18] R. D. Loft. Blue Gene/L Experiences at NCAR. in IBM System Scientific User Group meeting (SCICOMP11), URL: *http:// www.spscicomp.org*, 2005.

[19] Message Passing Interface Forum. MPI-2: a message passing interface standand. *High Performance Computing Applications*, 12(1-2), 1998.

[20] F. B. Schmuck and R. L. Haskin. GPFS: a shared-disk file system for large computing clusters. In *FAST'02*.

[21] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *IOPADS'99*.

[22] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *IOPADS'99*.

[23] R. Thakur, R. Ross, E. Lusk, and W. Gropp. ROMIO: A High-Performance, Portable MPI-IO Implementation. URL: *http:// www-unix.mcs.anl.gov/ romio/*, 2002.

[24] P. Wong and R. F. V. der Wijingaart. NAS Parallel benchmark I/O v2.4. Technical Report NAS-03-002, NASA Ames Research Center, 2003.

[25] J. Worringen, J. L. Traeff, and H. Ritzdorf. Fast parallel non-contiguous file access. In *SC'03*.

[26] M. Zingale. FLASH I/O benchmark routine. http:// flash.uchicago.edu/ zingale/flash_benchmark_io, 2002.