Parallel Processing Letters
© World Scientific Publishing Company

Massively Parallel I/O for Partitioned Solver Systems

NING LIU, JING FU, CHRISTOPHER D. CAROTHERS
*Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street
Troy, New York 12180, USA*

and

ONKAR SAHNI, KENNETH E. JANSEN, MARK S. SHEPHARD
*Scientific Computation Research Center, Rensselaer Polytechnic Institute, 110 8th Street
Troy, New York 12180, USA*

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

This paper investigates approaches for massively parallel partitioned solver systems. Typically, such systems have synchronized "loops" and will write data in a well defined block I/O format consisting of a header and data portion. Our target use for such an parallel I/O subsystem is *checkpoint-restart* where writing is by far the most common operation and reading typically only happens during either initialization or during a restart operation because of a system failure. We compare two parallel I/O strategies: a synchronized parallel IO library (*syncIO*), and a "reduced blocking" strategy (*rbIO*). Performance tests executed on the Blue Gene/P at Argonne National Laboratory using real CFD solver data from PHASTA (an unstructured grid finite element Navier-Stokes solver) show that the *syncIO* strategy can achieve a read bandwidth of 47.4 GB/sec and 27.5 GB/sec write bandwidth using 128K processors. The "reduced-blocking" *rbIO* strategy achieves an actual writing performance of 17.8 GB/sec and the *perceived* writing performance is 166 TB/sec using 128K Blue Gene/P processors.

Keywords: massively parallel processing, parallel I/O, synchronized, reduced blocking, Blue Gene/P

1. Introduction

Solver systems are an important software tool across a wide range of scientific, medical and engineering domains. To date, great strides have been made in terms of enabling complex solvers to scale to massively parallel platforms [25, 26]. In particular, the PHASTA computational fluid dynamics (CFD) solver has demonstrated strong scaling (85% processor utilization) on 294,912 Blue Gene/P processors [25] compared to a base case using 16,384 processors. Thus, for PHASTA and other petascale applications, the central bottleneck to scaling is parallel I/O performance, especially at high processor counts (i.e., >16K processors).

The computational work involved in implicit solver systems, using PHASTA as a driving example, consists of two components: (a) formation of the linearized algebraic system of equations and (b) computation of the implicit, iterative solution to the linear system of equations. In the context of implicit CFD solver systems, entity-level evaluations over the mesh are performed to construct a system of equations ($Ax = b$) as part of the finite element method. This results in an implicit solution matrix that is sparse but involves a large number of unknowns. The underlying mesh is partitioned into balanced *parts* and distributed among processors such that the workload imbalance is minimized for parallel processing. In the second component, the pre-conditioned iterative solver computes $q = Ap$ products and generates an update vector, x. Since q is distributed among processors, complete values in q would be assembled through a two-pass communication step.

In terms of I/O, PHASTA, like many partitioned solver systems, such as the Blue Waters PACT Turbulence system [9, 1], GROMACS [13], and LAMMPS [14], reads an initial dataset across all processors prior to starting the main computational components of the solver. This initialization data is divided into *parts*, where a *part* of the mesh or grid is assigned to a specific processor. In such an approach, each processor can read a dedicated *part* mesh/grid file coupled with an additional file for solution fields. For writing part data, especially for the purposes of system checkpoint-restart, the previously mentioned update vector x for each entity (i.e., vertex, element, particle etc.) needs to be written out to stable storage. The frequency of system wide entity writes to disk can vary. For PHASTA, it is sufficient to only write the solution state about every 10 time steps which is approximately one massively parallel write every 10 seconds of compute time where checkpoint data sizes range between two to 60 gigabytes per system wide write depending on mesh size and only a single checkpoint is typically maintained (e.g. the data is overwritten each checkpoint epoch).

Based on the above, efficient, massively parallel I/O for partitioned solver systems should meet the following set of objectives:

(1) The parallel I/O system needs to provide scalable parallel I/O performance and maintain that peak scaling if saturated as the processor counts continue to increase.
(2) The parallel I/O interface needs to be sufficiently flexible to allow the number of files used to not be directly dependent on the number of processors used. In fact, the number of files should be a tunable parameter to the parallel I/O system.
(3) The parallel I/O system should be able to “hide” I/O latency and not keep compute tasks waiting while I/O is being committed to disk.

The key contribution of this paper is a performance study of I/O alternatives to meet these objectives and dramatically decrease the time required for both the reading and writing of checkpoint-restart data files for massively parallel partitioned solver systems over our previous approach based on 1-POSIX-File-Per-Processor

(1PFPP). This is a significant extension of our previous study [11] which limits results to a 32K processor Blue Gene/L system. This study goes much deeper and provides scaling data at significantly higher processor counts using the 163,840 core Blue Gene/P system “Intrepid” at Argonne National Laboratory (ANL) for the two best performing I/O alternatives, *syncIO* and *rbIO*, from our previous study.

The remainder of the paper is organized as follows: Section 2 provides a detailed description for the I/O file format and system design. Section 3 presents an overview of Blue Gene/P and their I/O subsystem and detailed performance results and analysis of the two approaches using actual data files from PHASTA which serves as our example system. Section 4 contrasts the approaches in this paper with prior related work in parallel I/O and finally in Section 5 conclusions and future work are discussed.

2. Partitioned Solver Parallel I/O File Format and Interface

The central assumption of our partitioned solver I/O interface is that data is written and read across all processors in a coordinated manner. That is, each processor will participate in an IO operation, either reading or writing. With this assumption, the partitioned solver file format is defined as follows: for every file, there will be a master header and data blocks. The master header contains file information, such as number of data blocks, number of fields, and more importantly, an offset table that specifies the starting address of each data block in the file. The data blocks are divided into two sections: data header and data. The header specifies information about the data block (e.g., data block size, field string, etc) and the actual binary data generated from solver computations is contained in the data section.

```
Call Open File Function: (create one file)
Setup(#fields, #files, # parts/file)
Loop over Fields:  (0-2, solution,
                    time derivative solution &
                    discretization error)
    Parallel Loop over Parts:
        Proc 0: Part[0] & Part[1];
        Proc 1: Part[2] & Part[3];
           Compute GPID for this part field data;
           Call write_header(...);
           Call write_datablock(...);
    End Parallel loop
End loop
Close file
```

Fig. 1. Example parallel I/O using solver parallel I/O interface for a 2 processor (`Proc 0, 1`), 2 part (`Part[0], Part[1]`) and 3 field (`Field[0], Field[1], Field[2]`) scenario. The precise details of the arguments provided to the `write_header` and `write_datablock` functions are beyond the scope of this paper.

An example using the parallel I/O interface is shown in Figure 1. Here, a two processor case is given where each processor is assigned two parts and each part contains three fields (e.g., solution, time derivative and discretization error). These two processors write *in parallel* to one file. That is, the I/O is done in parallel using part-specific data assigned to each processor. In the multi-file case, each file has the same general format except that a different processor will have a different rank. In this case, the global part ID `GPid` is used instead of MPI rank when writing/reading parts. It is observed that by providing multiple parts per file, the number of different processor configurations that can be supported for a given set of input data increases.

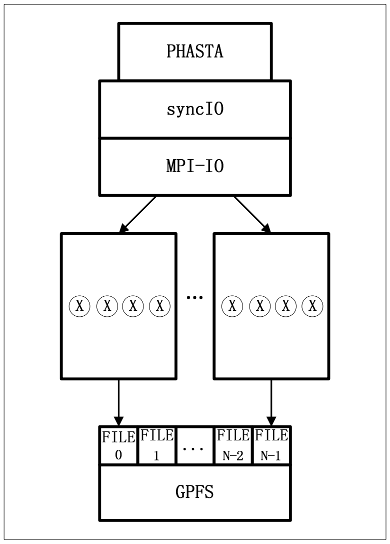


Fig. 2. Architecture for *syncIO*. *X* in each of the panels denotes which of the processors are *writers*.

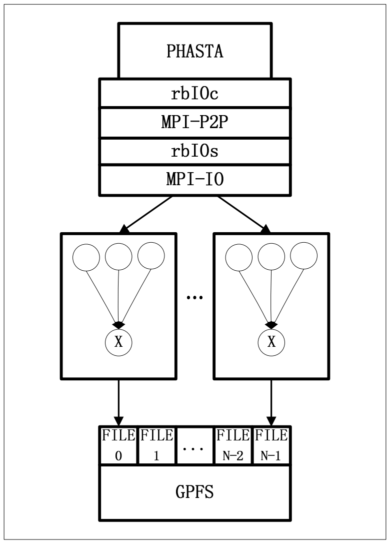


Fig. 3. Architecture for *rbIO*. *X* in each of the panels denotes which of the processors are *writers*.

2.1. syncIO*: Synchronized parallel I/O*

The synchronized parallel IO library (*syncIO*) is based directly on the MPI-IO library as shown in Figure 2. Here, all processors are divided evenly into groups before writing, which determine the file format based on the previous discussion. For reading, the number of groups is driven by the number of processors in the configuration and number of parts written in each file as well as the total number of files in the mesh/element dataset. However, when writing, the number of groups is a flexible user controllable parameter but the number of mesh *parts* does come into play when setting this parameter. Here, an MPI communicator enables a group of processors to communicate with each other. In our case, each group of processors shares a local MPI communicator and writes to one file in parallel. In order to avoid file system lock contention, each group of processors has one directory for their file.

2.2. rbIO*: Reduced-blocking Worker/Writer Parallel IO*

Reduced-blocking Worker/Writer parallel I/O (*rbIO*) approach, as shown in Figure 3, divides compute processors into two categories: workers and writers. A writer is dedicated to a specific, fixed group of workers and takes charge of the I/O operations that the workers require. When the workers want to write data to disk (i.e., rbIOc "client" layer in Figure 3), they send their data to their dedicated writer with `MPI_Isend()` and return from this non-blocking call very quickly without any interruption for I/O. The writer (rbIOs "server" layer in Figure 3) would aggregate the data from all workers, reorder data blocks and write to disk using a single call per writer to the `MPI_File_write_at()` function . The number of writers is tunable parameter in the I/O system.

3. Performance Results

3.1. *I/O Experiment Setup*

All experiments were executed on a 163,840 processor Blue Gene/P system called "Intrepid" located at the Argonne Leadership Computing Facility (ALCF). The Blue Gene system is a well known large-scale supercomputer system that balances the computing power of the processor against the data delivery speed and capacity of the interconnect, which is a 3D torus along with auxiliary networks for global communications, I/O and management. The Blue Gene divides compute and I/O work between dedicate groups of processors: compute and I/O. Intrepid has 64 compute nodes for each I/O node (each node has 4 processors) but the compute-I/O ratio can vary. Because compute processors are diskless, the OS forwards all I/O service calls from the compute processors to the dedicated I/O processors which executes the real I/O system call on behalf of the compute processor and processes the I/O request in-conjunction with a farm of fileservers which are running the GPFS file system [12, 28].

The ALCF storage system is composed of 16 racks of Data Direct Network (DDN) 9900 storage arrays. Here, the disk block is configure to be 4MB. Each disk logical unit (LUN) can be accessed by 8 fileservers, thus a total of 128 file servers are connected to the DDN devices. Two file systems share the identical resource where GPFS is configured to use all 128 file servers and PVFS is configured to use 112 file servers (7 file servers per DDN). This provides redundant paths to storage and the actual hardware IO limit could be achieved at lower processor count. The hardware limit for reading is 60 GB/sec and writing is 47 GB/sec[18]. Finally the 128 file servers are connected to 640 IO nodes through a 10 Gb Myricom switch complex. Two machines, Intrepid and Eureka, shares the file systems. Therefore, any activity from Eureka would affect the IO performance of Intrepid as well as other jobs running on Intrepid during our I/O experiments.

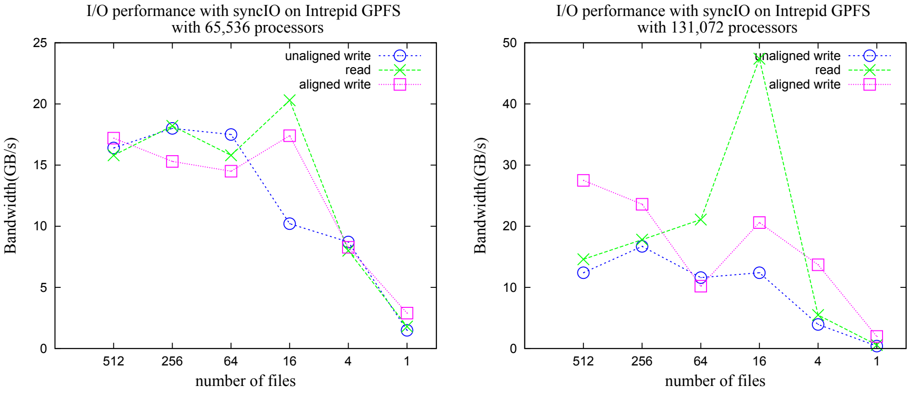


Fig. 4. *syncIO* performance on Blue Gene/P as a function of number of files. 64K processors is left panel and 128K processors is right panel

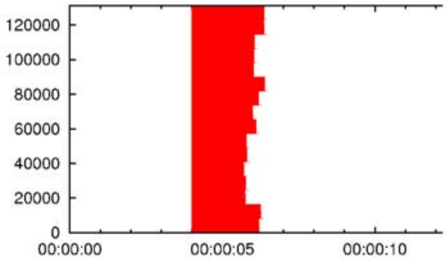


Fig. 5. *syncIO* reading on Blue Gene/P, 128K processors

The experiment test data comes from an 8 billion element mesh partitioned to 16K, 32K, 64K, 96K and 128K parts.The total size of this mesh is 60 GB. Recall, there is a one to one correspondence between the number of parts and the number of processors. Thus, as the number of processors increases, the computation required and data per processor decreases. The total computation and total data written in one step is nearly constant (slight increase due to boundary duplication). For example, with one billion mesh 2K parts, each processor will write 3.5MB of data while on 8 billion mesh 128K parts the size of data for each processor is about 500K.

3.2. syncIO *Results*

SyncIO is implemented using the `MPI_File_write_at_all_begin()/end()` function pair. This decision was based on our own performance tests which compared the performance of this function pair to using `MPI_File_write_at_all()`, and `MPI_File_write_at()`. While the implementation of `MPI_File_write_at_all_begin()` uses `MPI_File_write_at_all()` [24], we found "begin/end" to be slightly faster on a Blue Gene/L system and have kept that implementation for the Blue Gene/P

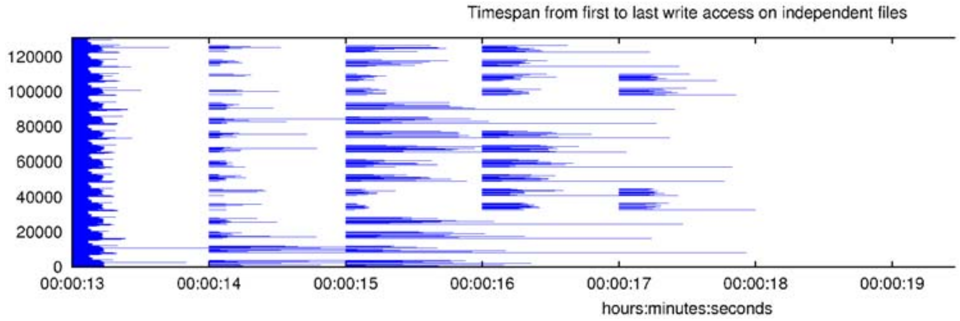


Fig. 6. *syncIO* unaligned writing on Blue Gene/P, 128K processors

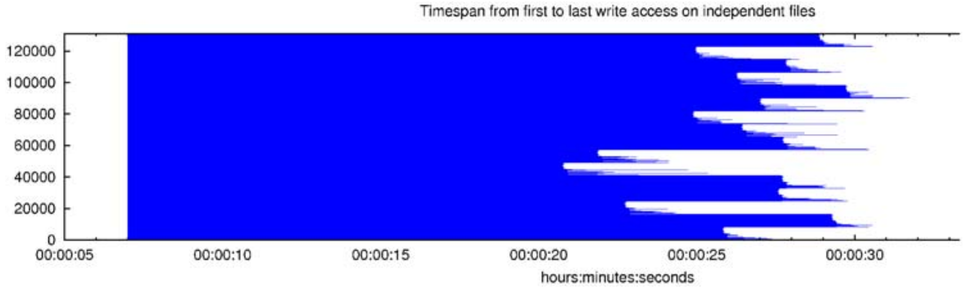


Fig. 7. *syncIO* aligned writing on Blue Gene/P, 128K processors

experiments presented here.

Our *syncIO* experiments were divided into two test cases: *aligned* and *unaligned* similar to the performance study done by [18]. For the aligned test cases, each processor will first check the size of data upon each I/O request and aggregate until it reaches the 4 MB GPFS "block" boundary or the file is closed. Then the data is committed to disk and the internal buffer is cleaned. For the unaligned case, each processor will write any size of data upon each I/O request. Specifically, the reading performance is based on the unaligned data which comes from the 8 billion element mesh. Here, each processor reads in about 500KB data when 128K cores are used. During the aligned writing test, each processor writes an exact 4MB data chunk regardless how much of the buffer holds real data. The total file size for 128K processors is about 512GB. While in the unaligned writing test, each processor also has about 500KB data and the total file size is 60GB.

It was initially thought that committing the whole 4MB data block to disk for the aligned cases as opposed to flushing partial data blocks to disk results in a performance penalty, e.g, more data equals more time. However, our experiments indicate otherwise. For example, using 1024 processors, the collective write times for 4MB vs. 500KB data chunks per processor are the same. At 8K processors, writing a 500KB buffer completes in one-half the time as writing a 4MB block. So, at 8K processors, the obtained bandwidth writing a 500KB buffer is only one-quarter of the bandwidth obtained using the full 4MB block. Finally, at 128K processors, we observed that 500KB write per processor takes 35 seconds, while the 4MB write per processor takes only 17 seconds. Thus, at 128K processors, we are observing a 16x bandwidth improvement for writing the whole 4MB data block in the aligned case. We attribute these results to optimizations which aggregate file I/O that is less than the GPFS disk block size. However, if the data chunk coming from each processor is a 4MB block (i.e., GPFS disk block size), then there is no reason to perform the aggregation and data moves significantly faster through the parallel I/O hardware infrastructure (e.g., from compute-node to io-node, io-node to fileserver and finally fileserver to disk). Consequently, we suggest application developers considering using both strategies but switch from aligned to unaligned when aggregate data becomes much smaller than the GPFS block size of the parallel system.

Now, the I/O performance on 64K processors and 128K processors are reported as a function of number of files in Figure 4. The peak performance for reading on 128K processors is 47.4 GB/sec, which is 75% the hardware peak rate. The writing peak performance is 27.5 GB/sec which is 46% of the peak rate. One reason for the off-peak write bandwidth is because the DDN write back cache is turned off during normal production runs for increased reliability but the mode of operation significantly degrades write performance. That said, Intrepid yields the highest I/O performance we have experienced to date compared with our previous parallel I/O performance study conducted on a Blue Gene/L system [11]. We attribute this to the large increase in number of fileservers (48 to 128 on Intrepid), significantly faster storage arrays (IBM DS4200 vs DDN 9900 on Intrepid) and the vastly improved I/O network (Force10 switch vs. Myricom on Intrepid).

We also observe that I/O performance degrades when total number of files is less than 16 both for read and write operations. Disk block and metadata contentions within each file is the reason for the performance drop. We also observe some anomalies which may be caused by other users from both Intrepid and Eureka. This observation can be further substantiated using the Darshan I/O performance characterization tool [4] on Intrepid which logs almost all I/O activities. Figure 5 shows the time for independent read and collective read operations of 60GB data using 128K processors. In this figure, the horizontal axis represents the time-span from the first to the last write access (in hour : minute : second), and the vertical axis represents all processors labeled by their respective MPI rank. The collective reading starts at 00:00:04 and completes nearly simultaneously across all processors at 00:00:06. Figure 6 shows the time for unaligned write operations where many small unordered write accesses take longer to commit to disk. Figure 7 shows the processor time-span for aligned writing. Here, most processors have finished writing synchronously and are able to take advantage of the available IO bandwidth.

3.3. rbIO *Results*

The *rbIO* writing performance results are shown in Figure 8. In terms of the write implementation, unlike *syncIO*, *rbIO* uses the `MPI_File_write_at()` function because writing is not done across all processors. *rbIO* worker processors only transmit data in a non-blocking manner via `MPI_Isend()` to writer processors and writer processors are the only tasks to communicate to the disk. The notion of *perceived* writing speed is defined as: *the speed at which worker processors could transfer their data*. This measure is calculated as total amount of data sent across all workers over the average time that `MPI_Isend()` takes to complete. To be clear, the writers can flush their IO requests in roughly the time between writes (recall, the typical delay between steps that write solution states is $O(10)$ seconds).

Figure 9 shows the average "perceived" writing performance from the worker's view. *A bandwidth of more than 166 terabytes per second using 128K processors is obtained on Intrepid.* This level of performance coincides the total bandwidth for `memcpy()` function coupled with some overhead for MPI non-blocking message processing. In other words, *rbIO* leverages the Blue Gene's internal memory bandwidth and torus network to transfer I/O data to the writer. So, from the worker's view, the writing speed is on par with the total memory system bandwidth, which is 13.6 GB/s peak per node, 445.6 TB/sec peak for a 128K partition. Because 3% or less processors are dedicated as writers, the majority of processors (the workers) incur little overhead for write I/O operations and return to next computation step immediately. Since the write time (3 to 4 seconds) on the I/O side is typically shorter than the computational time required to execute 10 time steps, the writers would be ready to receive worker's data before the workers actually require a new write and initiate I/O write requests. This trade-off can dramatically reduce the application execution time. The caveat to this approach is that the application has to be written such that it can "give-up" 3% or less of physical compute processors (512 and 1024 on 128K processors) and re-map computational work among the remaining 97% of compute processors.

Figure 8 reports the actual writing speed from the file system's perspective. The actual writing speed is measured as the total amount of data across all writers divided by the total wall-clock time it takes for the slowest writer to finish. This improvement over other approaches including *syncIO*(Figure 4) is attributed to lower file system overheads because a writer task will commit data chunk sizes that are inline with the GPFS file system 4 MB block size.

For the writers, there are several parameters we could explore here: number of writers, number of files and the manner (mode) of writing. Number of writers relies on the total size of data and the available RAM on each processors. In our case, the total data size is 60GB and a BG/P processors has 512MB RAM under *virtual node* mode. So if 128MB buffer was allocated for each writer, 512 or 1024 writers would be enough to hold all the data from workers in their memory before writing into disk. Additionally, the compute-node to io-node ratio should also be taken into

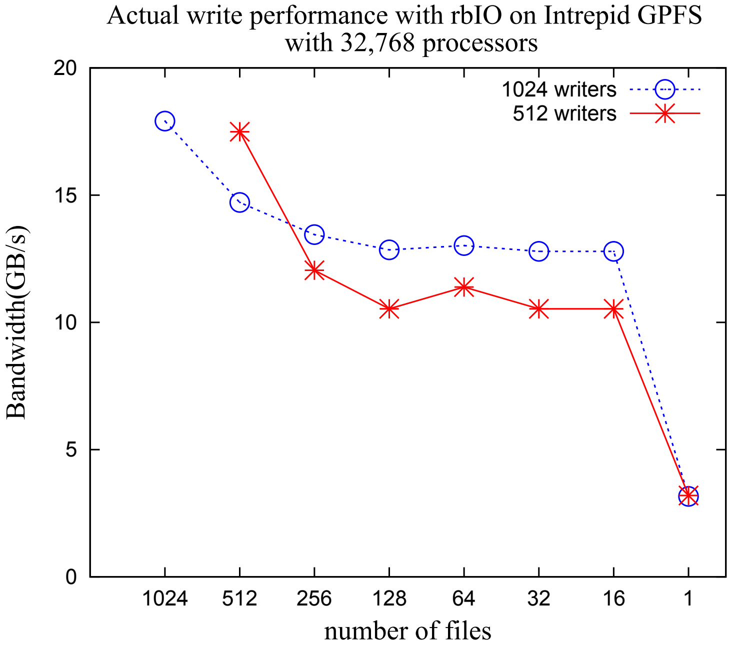


Fig. 8. Actual write performance with *rbIO* as a function of number of files and processors on Blue Gene/P.

Architecture	#procs	Bandwidth(TB/sec)
ALCF	32,768	20.9 TB/sec
Blue Gene/P	65,536	83.3 TB/sec
	131,072	166.7 TB/sec

Fig. 9. Perceived write performance with *rbIO* under 32K, 64K and 128K processors.

consideration when deciding on how many writers to use. That is, the worker/writer ratio should be set so that writers could be distributed equally over the physical Blue Gene `psets`, thus leveraging all available io-nodes in a given partition.

In this final series of experiments shown in Figure 10, three writing modes were tested among writers across a different number of files. `Mode 0` refers to non-collective/independent writes using the `MPI_COMM_SELF` communicator and `MPI_File_write_at()`. Here, each writer has its own file. `Mode 1` refers to a collective write over split MPI communicators and `MPI_File_write_at_all_begin()/end()`. `Mode 1` is essentially a *syncIO* for writers. In `Mode 2`, a collective open is issued among a sub-communicator of writers but writers would write to their own explicit (non-overlapping) offsets with `MPI_File_write_at()` routine simultaneously.

Figure 10 shows the actual write performance with *rbIO* as a function of write modes and number of files, under 64K and 128K processors with 1024 writers. In

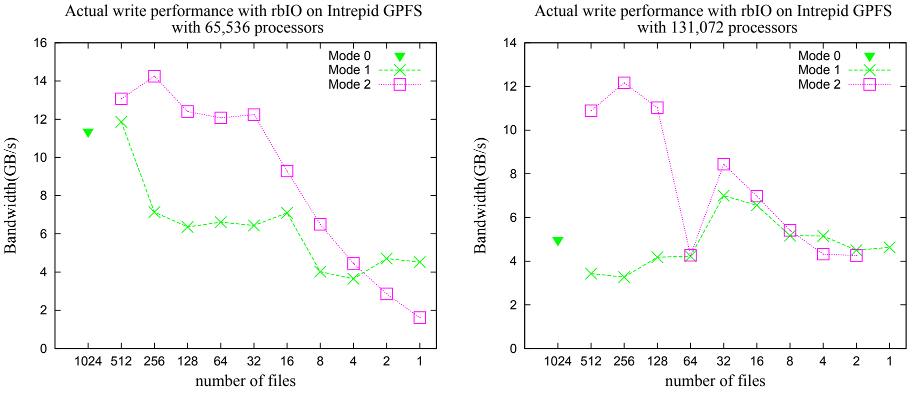
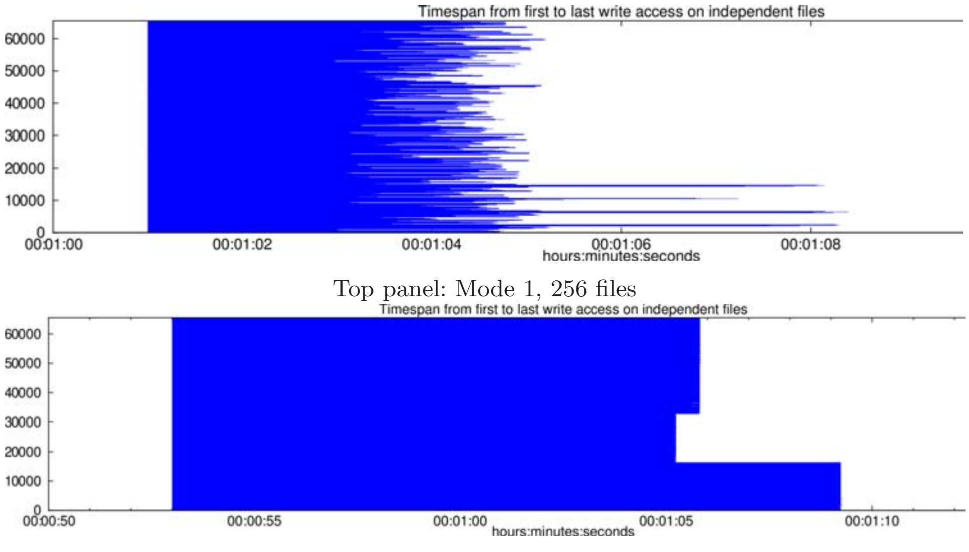


Fig. 10. Actual write performance with *rbIO* as a function of write modes and number of files, under 64K (left) and 128K (right) processors on Intrepid

Figure 10 (left), the general trend is similar to that observed in Figure 8 where a larger number of files yields better write performance. For Mode 1 and Mode 2, as the number of files increases, the overall performance would decrease due to file sharing contention. Another observation is that Mode 2 (independent writes to shared files) performs better than Mode 1 (collective writes to shared files). Finally, Mode 2 with 64 files experience a sudden loss in performance and it is observed that the peak I/O performance of Mode 2 using 256 files (12 GB/sec) at 128K cores is lower than the 18 GB/sec reported in Figure 8 which was strictly a Mode 0 experiment using 32K cores and 1024 files. This could have been caused by I/O noise from other online users.

To better understand the performance trend in Figure 10, we examined the Darshan file system performance logs available on Intrepid [4]. Figure 11 shows the write activity of Mode 1 under 256 files and 4 files in the 64K processor case. In Mode 1, several writers form a group and write their data in a collective manner. From Figure 11 (top), we can see that when the number of groups (e.g., file number) is large, there is some degree of variance in the time spent writing on a per-processor basis which implies there is some amount of contention in the file system. However, as shown in Figure 11 (bottom), when the group number becomes small, the major write phase across processors is more “aligned” due to the nature of collective write routines, but the major write itself takes a much longer time (16.4 seconds) than the 256 file case which completes in 8.4 seconds. This demonstrates that overhead of large collective write routines could be more significant in some cases compared to the overhead of managing more files.

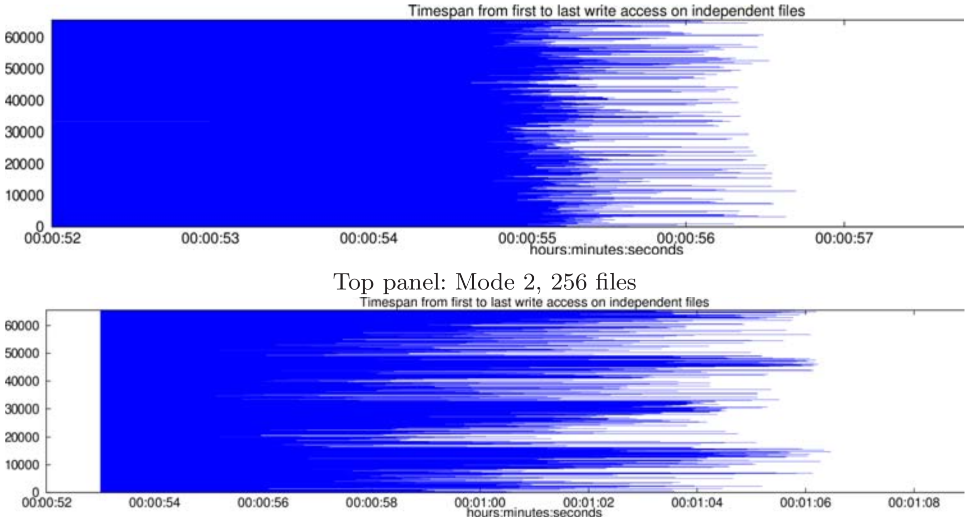
Figure 12 shows the write activity of Mode 2 under 256 files and 4 files in 64k processor case. In Mode 2, several writers form a group and write their data to a shared file in an independent manner. Figure 12 (top) shows how the major write phase finishes quickly across most of the processors and the overall performance is good in that it consumes only 4.2 seconds. In Figure 12 (bottom), less shared



Top panel: Mode 1, 256 files

Bottom panel: Mode 1, 4 files

Fig. 11. The write activity of Mode 1 under 256 files (top) and 4 files (bottom) in 64K processor case



Top panel: Mode 2, 256 files

Bottom panel: Mode 2, 4 files

Fig. 12. The write activity of Mode 2 under 256 files (top) and 4 files (bottom) in 64k processor case

files are used and thus more writers independently write into a shared file. So, here we observe a much higher degree of variance in per-processor write times, indicating there is some contention underneath happening even when writers are writing independently.

4. Related Work

The research mostly closely related to and inspired our *rbIO* system is by Nisar et al [22]. Here, an I/O delegate and caching system (IODC) is created that aggregates data by combining many small I/O requests into fewer but larger I/O request before initiating requests to file system while keeping a small number of processors dedicated to perform I/O. By using 10% extra compute processors as I/O processors, they improved performance of several I/O benchmarks using upto 400 processors. This work differs from the results presented here in several ways. First, IODC operates below the MPI-IO layer. The consequence of this implementation is that it has to be general and requires a complex cache/replica consistency mechanism. In contrast, *rbIO* is implemented at the user-level and leverages knowledge of the application in terms of how to partition writers and workers. Finally, *rbIO* only requires MPI/MPI-IO functionality whereas IODC requires OS support for threading which the Blue Gene/L does not support and Blue Gene/P supports threading only in a limited fashion. Finally, *rbIO* targets parallel I/O at scales greater than >16K processors.

Moreover, Lofstead et al [19, 20] designed an adaptable IO approach called "ADIOS" aiming at providing a portable and metadata rich high performance IO architecture. ADIOS is a high level API under which the user can choose from POSIX, MPI-IO, HDF-5, pnetCDF, DART, etc. at run time with the external metadata. The external metadata is a XML file which describes user IO configuration, including data structure, attributes, etc. So far, their work has reported performance data on Jaguar/Cray XT5 system configured with the Lustre file system using upto 8k processors. Our work resembles theirs in a way that our PHASTA-IO library can switch between parallel I/O strategies.

Seelam et al [29] have devised an application level I/O caching and prefetching approach to improve I/O latency on Blue Gene systems. In their work they use part of the memory on compute node as I/O buffer and use pthread/async I/O to hide latency. Our *rbIO* differs in that we leverage the torus network to hide I/O latency and use separate compute nodes to commit data so that the majority processors are not blocked at all thus achieving memory speed I/O bandwidths. Next, Alok et al [5] improve the collective I/O performance of MPI by providing three new methods for file domain partitioning. The first method aligns the partitioning with the file system's lock boundaries. The second method partitions a file into fixed blocks guided by file system lock granularity and then round-robin assigns the blocks to the I/O aggregators. The third approach divides the I/O aggregators into equalized groups. Here, the size of the group is equal to the number of I/O servers. These methods are then evaluated across the GPFS and Lustre file systems using the FLASH and S3D I/O benchmarks on the Cray XT4 and a Linux cluster (Mercury at NCSA). Using the group-cyclic approach on the CrayXT4 configured with the Lustre file sytem, a peak of 14 GB/sec is obtained using 2048 processors. Increasing processor count does not improve performance (configuration with up to 16K processors are reported). Because this approach operates within the MPI library software layer, our *syncIO* approach could leverage these improvements. The performance of *rbIO* may improve but the writer tasks would need to be re-implemented to make use of the collective I/O operations. All *rbIO* writers are currently implemented using the `MPI_COMM_SELF` communicator and write to their own file.

H. Yu et al [34] designed a parallel file I/O architecture for Blue Gene/L to exploit the scalability of GPFS. They used MPI-IO as the interface between application and file system. Several I/O benchmarks and a I/O intensive application were evaluated with their solution upto 1k processors with reported bandwidth 2 GB/s. Saini et al [27] ran several I/O benchmarks and synthetic compact application benchmarks on Columbia and NEC SX-8 using upto 512 processors. They conclude that MPI-IO performance depends on access patterns and that I/O is not scalable when all processors access a shared file. Dickens et al [8] report poor MPI-IO performance on Lustre and presented a user-level library that redistributes data such that overall I/O performance is improved. W. Yu et al [35] characterized the performance of several I/O benchmarks on Jaguar using the Lustre file system. Here, they demonstrate the efficacy of their I/O tuning approaches such as the two-phase collective I/O. Shan et al [30, 31] analyzed the disk access patterns for I/O intensive applications at the National Energy Research Supercomputing Center (NERSC) and selected parameters for IOR benchmark [17] to emulate the application behavior and overall workloads. They tested I/O performance with different programming interfaces (POSIX, MPI-IO, HDF5 and pNetCDF) upto 1k processors and they show that the best performance is achieved at relatively low processor counts on both ORNL Jaguar and NERSC Bassi. In contrast to these past results, this paper presents scaling data at higher processor counts.

Larkin et al [15] ran the IOR benchmark on Cray XT3/XT4 and report a tremendous performance drop at higher processor counts (around 2k). When a subset of processors (1K out of 9K processors) for I/O is used, which is similar to our *rbIO* approach, they achieved much better performance. A key difference is that for their approach, the I/O between workers, writers and the file system is synchronous based on the performance data provided. Additionally, Fahey et al [10] performed subsetting experiments on the Cray XT4 using four different I/O approaches (mpiio, agg, ser and swp) and upto 12,288 processors. Here, they achieved a write performance that is about 40% of peak. However, in their sub-setting benchmark, all subset master processors write data into a single file. The *rbIO* and *syncIO* approaches allow the number of files to be a tunable parameter.

Borrill et al [2, 3] develop the MADbench2 benchmark to examine the file I/O performance on several systems including: Lustre on Cray XT3; GPFS and PVFS2 [23] on Blue Gene/L; and CXFS [6] on SGI Altix3700. They extensively compared I/O performance across several different parameters including: concurrency, POSIX vs. MPI-IO, unique vs. shared-file access and I/O tuning parameters using upto 1K processors. They report there is no difference in performance between

POSIX and MPI-IO for large contiguous I/O access patterns of MADbench2. They also found that GPFS and PVFS2 provide nearly identical performance between shared and unique files under default configurations. The Blue Gene/L achieved slow I/O performance even at low processor counts (256 processors). In contrast, the results reported here targets parallel I/O performance at much higher processor counts than discussed in this previous work.

Lang et al [18] has recently conducted intensive IO tests on Intrepid PVFS file system. Their experiments were performed during an acceptance period before the machine and file system is available to other users. On IOR benchmark, they performed aligned/unaligned shared/unique file tests. The aligned tests used 4MB accesses for a total of 64 MB per process which inspired us to adjust the PHASTA-IO writing at 4MB boundary. They demonstrate IO scalability at over 128k processors and obtain 85% writing peak performance and 79% reading peak performance.

5. Conclusions

SyncIO and *rbIO* provide a new perspective on combining portable parallel I/O techniques for massively parallel partitioned solver systems. Performance tests using real partitioned CFD solver data from the PHASTA CFD solver shows that it is possible to use a small portion (less than 3%) of the processors as the dedicated writers and achieving an actual writing performance of 17.8 GB/sec and *perceived/latency hiding* writing performance of more than 166 TB/sec (nearly 37% of aggregate peak memory bandwidth) on the Blue Gene/P. It is additionally reported that the *syncIO* strategy can achieve a read bandwidth of 47.4 GB/sec while the writing performance is observed to be 27.5 GB/sec for the aligned case and 20 GB/sec for the unaligned case. Because of the bandwidth penalty for writing less than a 4MB block in the unaligned case, HPC application developers should be flexible in supporting both the aligned an unaligned data formats depending the per-processor data size that will be written to disk.

In the future, parallel I/O performance studies at even higher processors counts and other system architectures are planned by leveraging the JSC (294,912 processors) and Kraken (99,072 processors) the CrayXT5 system which uses Lustre file system. A thorough investigation between different file systems , including GPFS, PVFS and Lustre is also scheduled. In addition, we plan to investigate how the distribution of the designated I/O processors impacts the performance of *rbIO* across these systems for a variety of data sets.

Acknowledgments

We acknowledge funding support from the NSF PetaApps Program, OCI-0749152, and computing resources support from DOE INCITE Program. We thank Rob Latham, Ray Loy and Rob Ross at Argonne National Labs for a number of discussions that lead to this research and Rob Ross for an early review this manuscript.

We also acknowledge that the results presented in this article made use of software components provided by ACUSIM Software Inc. and Simmetrix.

References

[1] Blue Waters PACT Team Collaborations, http://www.ncsa.illinois.edu/BlueWaters/prac.html.

[2] J. Borrill, L. Oliker, J. Shalf and H. Shan, Investigation of leading HPC I/O performance using a scientific-application derived benchmark, in *Proceedings of the ACM/IEEE Supercomputing Conference (SC)*, 2007.

[3] J. Borrill, L. Oliker, J. Shalf, H. Shan and A. Uselton, HPC global file system performance analysis using a scientific-application derived benchmark, *Parallel Computing*. **35** (2009) 358–373.

[4] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale I/O workloads. in *Proceedings of 2009 Workshop on Interfaces and Architectures for Scientific Data Storage*, September 2009.

[5] A. Choudhary, W. K. Liao, K. Gao, A. Nisar, R. Ross, R. Thakur and R. Latham, Scalable I/O and analytics, *Journal of Physics: Conference Series*. **180** (2009).

[6] Cluster XFS, http://www.sgi.com/products/storage/software/cxfs.html.

[7] Computational center for nanotechnology innovations (ccni), Rensselaer technology park, north greenbush, n.y., http://www.rpi.edu/research/ccni.

[8] P. M. Dickens and J. Logan, Y-lib: a user level library to increase the performance of MPI-IO in a lustre file system environment, in *Proceedings of the IEEE Conference on High-Performance Distributed Computing (HPDC)*, 2009.

[9] D. A. Donzis, P. Yeung and K. Sreenivasan, Dissipation and enstrophy in isotropic turbulence: Resolution effects and scaling in direct numerical simulations, *Physics of Fluids*. **20** (2008).

[10] M. R. Fahey, J. M. Larkin and J. Adams, I/O performance on a Massively parallel cray XT3/XT4, in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing(IPDPS)* , 2008, pp. 1–12.

[11] J. Fu, N. Liu, O. Sahni, K. E. Jansen, M. S. Shephard and C. D. Carothers, Scalable Parallel I/O Alternatives for Massively Parallel Partitioned Solver Systems, in *Proceeding of the IEEE Large-Scale Parallel Processing Workshop (LLSP), at the IEEE International Symposium on Parallel and Distributed Processing*, 2010.

[12] General Parallel File System, http://www-03.ibm.com/systems/clusters/software/gpfs/index.html.

[13] GROMACS, http://www.gromacs.org.

[14] LAMMPS, http://www.cs.sandia.gov/sjplimp/lammps.html.

[15] J. M. Larkin and M. R. Fahey, Guidelines for Efficient Parallel I/O on the Cray XT3/XT4, in *Cray Users Group Meeting (CUG)* 2007.

[16] HDF5, http://www.hdfgroup.org/HDF5.

[17] IOR Benchmark, https://asc.llnl.gov/sequoia/benchmarks/#ior.

[18] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms and W. Allcock, I/O performance challenges at leadership scale, in *Proceedings of the ACM/IEEE Supercomputing Conference (SC)*, November 2009.

[19] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki and C. Jin, Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS), in *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, 2008.

[20] J. Lofstead, Z. Fang, S. Klasky and K. Schwan, Adaptable, metadata rich IO methods

for portable high performance IO, in *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009.

[21] Message Passing Interface, http://www.mcs.anl.gov/research/projects/mpi.

[22] A. Nisar, W. Liao and A. Choudhary, Scaling parallel I/O performance through I/O delegate and caching system, in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[23] Parallel virtual file system 2, http://www.pvfs.org.

[24] R. Ross, Personal communication via email, December 12, 2009.

[25] O. Sahni, M. Zhou, M. S. Shephard and K. Jansen, Scalable implicit finite element solver for massively parallel processing with demonstration to 160k cores, in *Proceedings of the ACM/IEEE Supercomputing Conference (SC)*, 2009, Portland, Oregon.

[26] O. Sahni, C. D. Carothers, M. S. Shephard and K. E. Jansen, Strong scaling analysis of a parallel, unstructured, implicit solver and the influence of the operating system interference, *Scientific Programming*. **17** (2009) 261–274.

[27] S. Saini, D. Talcott, R. Thakur, P. A. Adamidis, R. Rabenseifner and R. Ciotti, Parallel I/O performance characterization of Columbia and NEC SX-8 superclusters, in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2007.

[28] F. B. Schmuck and R. L. Haskin, GPFS: A shared-disk file system for large computing clusters, in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[29] S. Seelam, Is Chung, J. Bauer and H. Wen, Masking I/O Latency using Application Level I/O Caching and Prefetching on Blue Gene systems, in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing IPDPS*, 2010

[30] H. Shan and J. Shalf, Using IOR to analyze the I/O performance for HPC platforms, in *Cray Users Group Meeting (CUG) 2007*.

[31] H. Shan, K. Antypas and J. Shalf, Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark, in *Proceedings of the ACM Supercomputing Conference (SC)*, 2008.

[32] Silo: A mesh and field I/O library and scientific database, http://wci.llnl.gov/codes/silo.

[33] R. Thakur, E. Lusk and W. Gropp, Users guide for ROMIO: A high-performance, portable MPI-IO implementation, 2004, Technical Report Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory.

[34] H. Yu, R. K. Sahoo, C. Howson, G. Almasi, J. G. Castaños, M. Gupta, J. E. Moreira, J. J. Parker, T. Engelsiepen, R. B. Ross, R. Thakur, R. Latham and W. D. Group, High performance file I/O for the Blue Gene/L supercomputer, in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, 2006.

[35] W. Yu, J. S. Vetter and S. Oral, Performance characterization and optimization of parallel I/O on the cray XT, in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008.