ParColl: Partitioned Collective I/O on the Cray XT

Weikuan Yu, Jeffrey Vetter

Computer Science and Mathematics
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6173
{wyu,vetter}@ornl.gov

Abstract

Collective I/O orchestrates I/O from parallel processes by aggregating fine-grained requests into large ones. However, its performance is typically a fraction of the potential I/O bandwidth on large scale platforms such as Cray XT. Based on our analysis, the time spent in global process synchronization dominates the actual time in file reads/writes, which imposes a 'collective wall' on the performance of collective I/O. In this paper, we introduce a novel technique called partitioned collective I/O (ParColl). ParColl augments the original two-phase collective I/O protocol with new mechanisms for file area partitioning, I/O aggregator distribution and intermediate file views. Through these mechanisms, a group of processes and their targeted file are consistently divided into a collection of small subgroups, each performing I/O aggregation in a disjoint manner. File consistency is maintained through intermediate file views when necessary. Together, these mechanisms greatly reduce the cost of global synchronization. Our experimental results demonstrate that ParColl significantly improves the performance and the scalability of collective I/O. In one case, we show a 416% improvement on 1024 processes for a visualization I/O benchmark. We also show that the I/O patterns in scientific applications can benefit significantly from this technique, e.g. BT-I/O and Flash I/O.

1 Introduction

Today's Massively Parallel Processing (MPP) platforms are deployed with 100s of TeraFlops (10^{15}) [4]. To meet the needs of data-intensive scientific applications, these MPP systems such as BlueGene/L [6] and Cray XT [29] are often deployed with a scalable I/O subsystem. For example, Tera-10 at CEA in Europe has reported aggregated I/O throughput reaching 100GBps [9]. However, our experiences with scientific applications show that they use higher-level, hierarchical data structures, such as multi-dimensional arrays with complex data decompositions among parallel processes. These higher-level abstractions often translate into fine-grained, hierarchical, typically non-contiguous data accesses [13, 23]. Such I/O patterns result in I/O throughputs that are orders of magnitude lower than the potential performance of the physical storage hardware [26]. Collective I/O is a strategy developed to cope with such patterns [23, 25]. It is carried out as interleaved phases of *data exchange* and *file I/O* (reads/writes) on the linearly partitioned file domains, thereby aggregating small I/O requests into larger ones for better performance.

However, this two-phase protocol inherently imposes scalability bottlenecks for large-scale platforms, due to its requirement of frequent synchronization between data exchange and file I/O. We have profiled a parallel visualization benchmark, MPI-Tile-IO [19] on the Jaguar Cray XT platform [17]. Figure 1 shows the trend of synchronization cost. For a small number of processes, the actual synchronization cost for a small number of processes, does not pose a limit to the I/O performance. However, we found that, with 512 processes MPI-tile-IO spends 72% of its total time in global synchronization, which dominates the time spent on file reads/writes. Amdahl's law tells us that this essentially imposes a wall for scalable and efficient collective-I/O. We refer to such dominance of synchronization cost as the ***collective wall*** to denote the fact that it imposes a global barrier to the I/O performance of all participating processes. As this is a noticeable problem at 512 processors on current architectures, it can lead to a colossal scalability challenge for upcoming ultra-scale computing platforms that have hundreds of thousands of processors and beyond. Thus it is critical to address this problem in order to ensure that future ultra-scale systems are successful in delivering scalable I/O performance to real scientific applications I/O patterns.

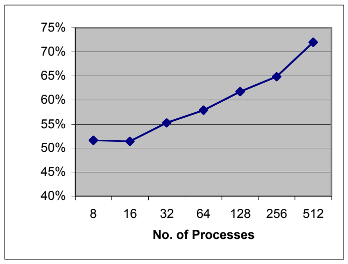


Figure 1 The Collective Wall in Collective IO

Through an open-source MPI-IO implementation [32] with comparable performance, we have examined the processing of collective I/O, and uncovered the collective wall problem as shown in Figure 1. The synchronization that causes the collective wall is actually implemented in the original MPI-IO protocol using global collective operations include allgather, alltoall and allreduce. It may be tempting to address this synchronization problem via a simple replacement of these collective operations with many point-to-point operations. However, the real issue here is the inherent need of synchronization inside the original two-phase protocol. Using point-to-point operations for the synchronization purpose would lead to a similar problem, let alone the collective operations are typically conglomerates of point-to-point operations.

We have designed a new collective I/O protocol called Partitioned Collective I/O (ParColl) to address this problem. ParColl augments the two-phase collective I/O protocol with new mechanisms for file area partitioning, I/O aggregator distribution and intermediate file views. I/O aggregators are the processes that gather the I/O requests and perform reads/writes on behalf of the entire group. Through these mechanisms, a group of processes as well as their targeted file are divided in a consistent way into separated subgroups. These subgroups in turn perform their own aggregation of I/O requests in a disjoint manner, thus avoiding collective I/O across a big global group. Together, they reduce the cost of global synchronization. The file consistency is still maintained, through intermediate file views when necessary. ParColl retains the original benefits of I/O aggregation in two-phase collective I/O, while reducing the synchronization costs and cascading effects among processes. Our experimental results show that ParColl provides significant benefits on the Cray XT. The performance of collective I/O can be improved by as much as 416% for MPI-Tile-IO with 1024 processes. Scientific application such as Flash can also significantly benefit from these techniques. In summary, we make the following contributions in this paper.

- Through detailed dissection, we pinpoint and quantify the scalability constraints imposed by collective communication to collective I/O, i.e. the *collective wall*.
- To optimize collective I/O, we have designed and implemented a new protocol: partitioned collective I/O.
- We demonstrate that ParColl brings significant benefits to various benchmarks and scientific applications over the Cray XT

The rest of the paper is organized as follows. In the next section, we discuss the motivation. In Section 3, we provide an overview of related work. In Section 4, we discuss our design of ParColl in detail, followed by a performance evaluation in Section 5. Section 6 concludes the paper.

2 Motivation

2.1 An Overview of the Cray XT

The Cray XT is a line of massively parallel processor (MPP) systems from Cray. It inherits the system software from a sequence of systems developed at Sandia National Laboratories and University of New Mexico: ASCI Red [28], the Cplant [18], and Red Storm [7]. Computing nodes on the Cray XT run a lightweight operating system called Catamount. The Catamount kernel runs only one single-threaded process and does not support demand-paged virtual memory. On Catamount, data in I/O requests are delivered from one process' user space to the other process' user space without kernel buffering.

Cray XT uses Lustre [10] for a scalable I/O subsystem, supporting I/O services to thousands of concurrent clients. Some of the Cray XT nodes are configured as Lustre servers that provide I/O and Metadata services, being attached with high performance storage targets, such as DDN S2A9550 [12]. Applications perform I/O through an interface called *liblustre* that converted clients' I/O requests to the SYSIO and liblustre libraries [20], and then to the Lustre servers. Parallel processes in an application can perform I/O either by directly invoking POSIX read/write, or by calling through a MPI-IO library. On the Cray XT systems, a proprietary MPI-IO implementation is provided, which offers parallel I/O through a SYSIO-specific ADIO implementation [24], denoted as AD_Sysio.

2.2 The Collective Wall on the Cray XT

We have developed an open-source MPI-IO implementation that provides comparable performance to that of the Cray implementation [32]. This open-source MPI-IO implementation on Cray XT constitutes a comparable baseline implementation for further optimizations of parallel I/O on Cray XT. Using this open source MPI-IO package, we dissect the code path of collective I/O and also provide the breakdown timings of different processing tasks.

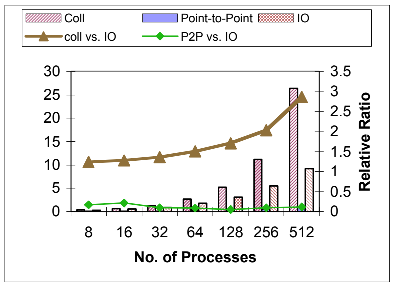


Figure 2 Collective I/O Time Breakdown

In the canonical extended two-phase (data exchange and file I/O) protocol, the processing of collective I/O can be divided into several phases: file range gathering, file domain partitioning, request dissemination and interleaved phases of data exchange and file I/O. There are three main components of processing in these steps: data exchange with point-to-point communication, file I/O, and the required process synchronization. In particular, the synchronization consists of several rounds of collective operations during the interleaved phases of data exchange and file I/O. These collective operations are needed for the coordination among processes, so that a long stream of I/O requests can be exchanged in a staged fashion without overflowing the intermediate system buffer. To gain insights into these processing tasks, we profiled these processing tasks at run-time. When a file is closed, a summary is reported.

Figure 2 shows the time breakdown of collective I/O in MPI-Tile-IO [19]) The profile of MPI-Tile-IO was taken with a tile size of 1024*768. As shown in the figure, the processing time spent in synchronization grows much faster compared to the time spent on point-to-point communication and file I/O. With 512 processes, the time spent in synchronization starts to dominate over the other portions. As mentioned in Section 1, we refer to this as the problem of *collective wall* because Amdahl's law tells that this increasing dominance of collective communication is likely to present a wall on the I/O performance.

2.3 Improving Collective I/O on the Cray XT

One needs to take into account various system issues on the Cray XT when considering different collective I/O strategies. The lack of support for application threads on Cray XT imposes limitations on the utilization of other collective I/O techniques, such as split-phase collective I/O [13] and client-side file caching [16]. A new operating system with threading support (called Compute Node Linux) is in progress to replace the original catamount operating system on the Cray XT. However, even when threading becomes available to help the overlap of I/O and data exchange, the I/O cost can only be hidden within other costs, to the ideal extreme. It does not do away with the need of synchronization and the bound scalability problem in collective I/O. Instead, the relative dominance of synchronization cost could become even more pronounced with the diminishing I/O time. While future features on Cray XT are still evolving, in this paper, we focus on the Cray XT with the catamount operating system in our investigation of collective I/O.

The collective wall problem is essentially caused by the inherent need of global synchronization in the extended two-phase *(ext2ph)* collective I/O protocol. Without addressing the need of synchronization, this problem cannot be solved through the simple replacement of collective operations with point-to-point operations. There are two alternative strategies for solving this problem. One way to do that is to completely re-design the collective I/O protocol so that it no longer involves interleaved phases of data exchange and I/O, therefore no more phases of global synchronization between them. The other way is to take an evolutionary approach and alleviate the scalability problem caused by global synchronization across big groups. In this paper, we take on the second approach. We have designed a new collective I/O protocol called Partitioned Collective I/O (ParColl). ParColl integrates the original two-phase protocol as a built-in component for I/O aggregation. Our results have confirmed that ParColl can significantly reduce the synchronization cost for thousands of processes. It is possible that the same scalability problem can happen again at even larger scale, e.g. more than 100,000 processes. We consider a redesign of ext2ph maybe needed at that point. As ext2ph is essentially the core of an MPI-IO implementation, redesigning ext2ph would imply drastic instrumentation to the existing code and also the MPI-IO implementation itself. We plan to examine the needs of a new design and its trade-off with ParColl in our future study.

3 Related Work

The performance of parallel I/O has been an active topic for many previous studies. ROMIO [6] provides the most popular implementation of the parallel I/O interface, MPI-IO. It implements a variety of techniques including extended two-phase I/O [15, 23], split-phase collective I/O [13] and disk-directed I/O [11]. The two-phase protocol aggregates small, often non-contiguous I/O requests into large contiguous requests for effective collective-IO. Split-phase collective I/O [13] optimizes collective I/O by utilizing additional threads to achieve overlapped communication with I/O. Disk-directed I/O [11] aggregates all I/O requests to a few I/O servers. In doing so, it tries to exploit the servers' proximity to disks and organize the requests in a way suitable for better disk performance.

Some recent studies have been done to improve parallel I/O via optimizations at the MPI-IO layer. For example, Liao *et al.* [10, 11] have carried out a series of studies on improving collective I/O by caching application data at the user level. This strategy has also been shown as beneficial without putting significant pressure on memory requirements. Others have attempted to improve parallel I/O by introducing file system specific optimizations. For example, MPI-IO/GPFS [13] and MPI-IO/BlueGene [31] have introduced MPI-IO optimizations that are specifically designed to take advantage of the specific features of General Parallel File System (GPFS) and BlueGene [6]. Tatebe *et al.* [22] have exploited the concepts of local file view to maximize the use of local I/O bandwidth in the design of a distributed file system for the Grid environment.

Recent efforts on BlueGene/L have reported to achieve high performance parallel I/O [31] through an architecture-specific design. In [31], the two-phase collective I/O protocol has been significantly customized to adapt to the organization of *processing set (pset)*, in which the compute nodes aggregate the data and the I/O node does file I/O. In addition, many of the design choices are based on the underlying file system - GPFS and the availability of fast collective network over BlueGene. Thus, this solution is not generally applicable to other large-scale machines because of its tight integration with BlueGene-specific system architecture.

Little research has been done on optimizing the performance of collective I/O for the Cray XT platforms. Brightwell *et al.* have recently introduced NIC-based optimizations for MPI message passing on Cray SeaStar interconnect [8]. This NIC-based MPI communication is conducive to the performance improvement of collective I/O because it benefits both the point-to-point communication and collective communications, which are two of processing components for collective I/O. In this work, we investigate the performance issue of collective I/O over the massively parallel Cray XT platform, Jaguar. We have designed partitioned collective I/O to cope with the scalability problem imposed by the internal global synchronization in the original protocol. Our design is able to significantly alleviate this key bottleneck.

4 Partitioned Collective I/O

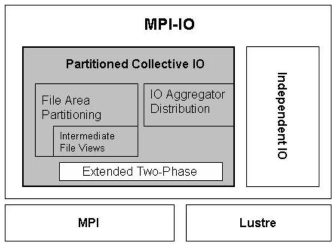


Figure 3 The Architecture of Partitioned Collective I/O

The main theme of ParColl is to reduce the side-effects of global synchronization inside collective I/O. Figure 3 shows the architecture of ParColl. Parallel processes are divided into several groups, each performing its I/O in a smaller, yet collective fashion. The original ext2ph protocol is still retained as a part of ParColl, providing the basic data aggregation and file I/O for all subgroups of processes. There are two main issues related to the associated resources for collective I/O in order to achieve appropriate partitioning. These include: (a) how to partition the file among the process groups; and (b) how to distribute I/O aggregators amongst all process groups. To address these issues, ParColl extends the original two-phase collective I/O protocol with new mechanisms for file area partitioning, I/O aggregator distribution and intermediate file views. There is a tradeoff between synchronization cost and the I/O aggregation when choosing an optimal group size for ParColl. Provided that the size of subgroups is not too small, ParColl retains the benefits of I/O aggregation while achieving more scalable I/O without the detrimental effects of global synchronization. In this paper, we empirically evaluate the impact of the group size to the effectiveness of ParColl, leaving the examination of an optimal group size to a future study as it is closely correlated with the I/O pattern of a particular application. Note that ParColl instruments the internal implementation of Collective I/O. It does not alter the semantics of MPI-IO. The file consistency is still maintained through the use of intermediate file views as needed.

4.1 File Area Partitioning

Because ParColl divides processes into separate subgroups, the entire file is also partitioned among these groups. In the original ext2ph protocol, a file is partitioned into file domains amongst the I/O aggregators. With processes grouped into subsets in ParColl, a file needs to be first divided among the subgroups. We refer to the file region that one subgroup needs to read/write as its *File Area* (FA). In fact, the partitioning of a file into FAs is the premier issue for ParColl because it affects both the I/O consistency and the performance of resulting collective I/O operations. On one hand, a file should be evenly (or close to) divided into FAs for balanced I/O load among subgroups. On the other hand, there should be non-overlapping FAs. Otherwise, the consistency of file accesses cannot be ensured when multiple un-coordinated subgroups perform their own collective I/O to the overlapped FAs.

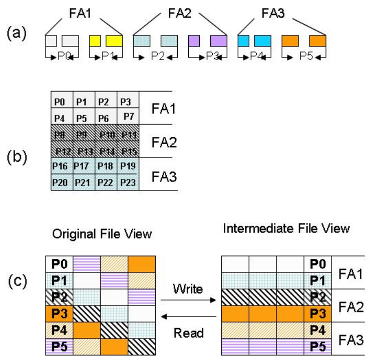
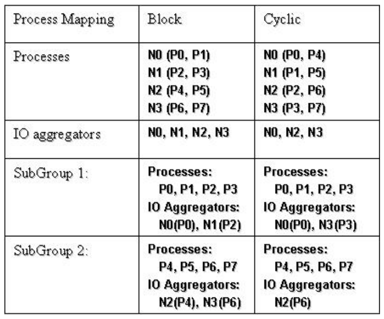


Figure 4 File Area Partitioning

Figure 4 shows the diagram of three different file access patterns for collective I/O. Figure 4(a) represents the simplest pattern in which the file segments for all six processes are serially distributed and there are no intersections among processes. A simple offset calculation would partition the file into non-overlapping FAs, each of which is owned by the corresponding subgroup. In Figure 4 (b), each process reads (or writes) a tile of the file with a two-dimensional global array. The beginning and ending offsets of some tiles are intersecting each other. Such pattern is very common with visualization applications. For such patterns, the tiles from some processes (8 in this case) can be grouped together and form distinct, non-intersecting FAs, which again can be owned by its corresponding subgroup.

Intermediate File Views -- Figure 4 (c) shows the third, more complicated pattern. In this pattern, each process, e.g. P0, reads or writes to 4 tiles of the file. However, the tiles from any process spread widely across the entire file. Any direct file partitioning is improbable to cover the tiles of a single process. Such pattern is exhibited by scientific applications with complex structured datatypes, such as BT-IO [30]. In MPI-IO, such file patterns are established by setting up a layout for the file, called file view [5], before the actual collective I/O. We have designed a file view switching mechanism that detects such pattern at the file view initiation time, and then converts the original file view into an intermediate file view as shown in Figure 6(c). The intermediate file view can be viewed as logical file representation in which different I/O segments for any individual process are consecutively joined together in a virtual manner, before reading/writing the file. With the new file view, file partitioning is then a special case of pattern (b). The original file view is still needed to provide the physical layout and distribution of I/O segments. The correspondence between the intermediate file view and the original file view makes it convenient in partitioning processes and file areas. Data are read or written correctly using the same representation via an intermediate file view to the original file view. In our prototype, the switching of the file views is enabled dynamically by detecting intersections among partitioned FAs.

4.2 I/O Aggregator Distribution



Process Mapping	Block	Cyclic
Processes	N0 (P0, P1) N1 (P2, P3) N2 (P4, P5) N3 (P6, P7)	N0 (P0, P4) N1 (P1, P5) N2 (P2, P6) N3 (P3, P7)
IO aggregators	N0, N1, N2, N3	N0, N2, N3
SubGroup 1:	Processes: P0, P1, P2, P3 IO Aggregators: N0(P0), N1(P2)	Processes: P0, P1, P2, P3 IO Aggregators: N0(P0), N3(P3)
SubGroup 2:	Processes: P4, P5, P6, P7 IO Aggregators: N2(P4), N3(P6)	Processes: P4, P5, P6, P7 IO Aggregators: N2(P6)

Figure 5 Distribution of I/O Aggregators

As discussed in the previous section, file partitioning determines how the processes should be grouped together to form balanced and non-overlapping FAs. However, only a selected number of processes perform file I/O as I/O aggregators. Applications can also provide one of two different hints: (a) the number of I/O aggregators to use from the default list; or (b) a list of physical nodes to use as I/O aggregators. These choices are enabled as user-manageable hints in MPI-IO. While ParColl does not require changes on the existing specification and the usage of such hints, it does introduce more complexities to the selection of I/O aggregators. For example, the I/O aggregators selected by default may fall into the first one (or few) I/O subgroups; multiple processes from the same physical node can be partitioned into different subgroups. To be compatible with the provided list of I/O aggregators, it is necessary to ensure that ParColl meets these requirements: (a) each subgroup of processes shall have at least one I/O aggregators; (b) no processes from the same physical node can be I/O aggregators for different subgroups; (c) I/O aggregators are as evenly distributed as permitted by the groups of processes.

To make ParColl compatible with the original semantics of I/O aggregator specification, we have implemented a distribution algorithm to meet these requirements. It traverses all processes in a subgroup to choose an I/O aggregator from the list of available aggregators. The partitioning is done in a round-robin manner for each subgroup until all I/O aggregators are assigned. Figure 5 shows some sample distributions of I/O aggregators under the common block- and cyclic-based process mapping schemes. For the block mapping case, four I/O aggregators are properly distributed to two subgroups; for the cyclic case with three I/O aggregators, each group first gets one I/O aggregator, the third one is then left to Subgroup 1.

5 Performance Evaluation

We have conducted our experiments on the Jaguar supercomputer [17] at Oak Ridge National Laboratory. There are three different Lustre file systems on this platform with different number of storage targets [14, 33]. We used one file system with 72 Lustre object storage targets (OSTs), each with a 4Gbps Fibre Channel interface. The files for our tests are all striped across 64 targets with a stripe size of 4MB. All our tests are also conducted using both cores on the compute PEs. We have evaluated with the performance of ParColl using both micro-benchmarks such as IOR, and the I/O kernel of scientific applications such as BT-IO and Flash I/O. Our experiment results were collected with repeated measurements to eliminate any significant interference from other loads.

5.1 IOR

IOR [2] is a benchmark that measures the performance of various parallel I/O patterns. IOR also can test the performance of parallel I/O through different I/O interfaces, including POSIX read/write, MPI-IO independent or collective read/write, as well as higher level libraries such as HDF5 [27].

IOR Collective I/O with ParColl – We have tested IOR collective I/O with a varying number of processes. In our IOR experiments, all processes are collectively writing a contiguous buffer of 512MB, in units of 4MB, into a shared file. While this type of contiguous I/O would not benefit with the aggregation from collective I/O, it typically leads to a significant low I/O throughput on Cray XT. Note that we carried out this test to see how ParColl can benefit IOR in such scenarios, not to recommend the use of collective I/O for this type of contiguous I/O patterns. Figure 6 shows the performance of IOR with 128- and 512- processes, with a least group size of 8. *ParColl-N* denotes that a run of Flash I/O using ParColl with N subgroups. Compared to the Cray implementation, ParColl improves the aggregated bandwidth up to 5,301MB/sec. This leads to a performance improvement of 12.8 times, compared to the original 380MB/sec across 512 processes. It demonstrates that ParColl is very beneficial to collective I/O in IOR by breaking processes into groups and reducing the synchronization amongst processes.

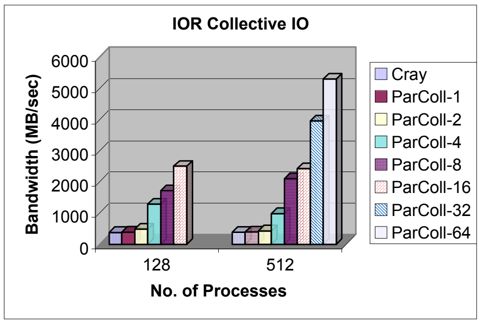


Figure 6 Benefits of ParColl to IOR collective I/O

5.2 MPI-Tile-IO

MPI-Tile-IO [19] is an MPI-IO benchmark testing the performance of tiled data accesses. In this application, data I/O is non-contiguous and issued in a single step using collective I/O. It tests the performance of tiled access to a two-dimensional dense dataset, simulating the type of workload that exists in some visualization and scientific applications. In our experiments, each process renders a 1x1 tile with 1024x768 pixels. The size of each element is 64 bytes, leading to a file size of 48**N* MB, where *N* is the number of processes.

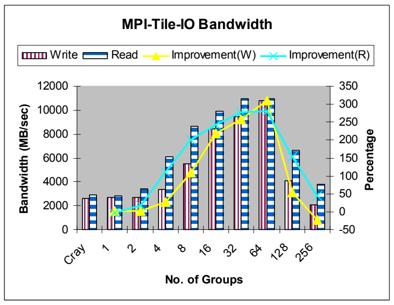


Figure 7 Performance of MPI-Tile-IO

To measure the benefits of ParColl with different number of groups, we have divided the file into a varying number of FAs, i.e. the processes being divided into in the same number of subgroups. Figure 7 shows the performance of MPI-Tile-IO with a varying number of subgroups. Compared to the Cray implementation, ParColl provides comparable performance with one and two subgroups. As shown in Figure 10, the best performance is achieved with 64 subgroups for MPI-Tile-IO. The performance is improved by 210% and 180% for writes and reads, respectively. While ParColl is partitioning processes into even more subgroups, the aggregated I/O performance drops significantly. This represents a balance point between the size of I/O aggregation and the synchronization cost.

Synchronization Cost Reduction -- As discussed in Section 2, the extended two-phase protocol requires frequent synchronization. We have also evaluated the benefits of ParColl in reducing synchronization cost. Figure 8 shows that the synchronization cost is significantly reduced by both absolute value and relative ratio. These results suggest that ParColl is effective in breaking the collective wall imposed by synchronization, unless a group is over partitioned into an extreme number of subgroups, in which case, fine-grained I/O relinquishes the benefits of aggregation in collective I/O.

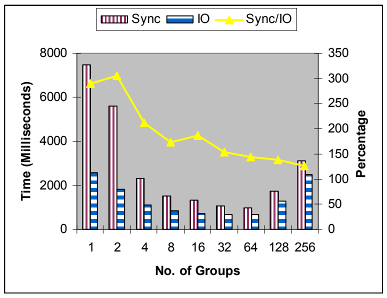


Figure 8 Reduction of Synchronization Cost

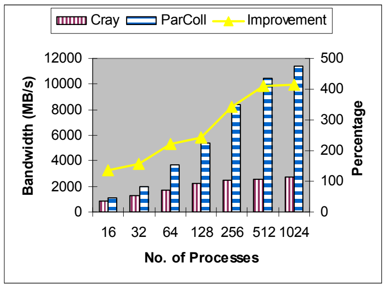


Figure 9 The Improved Scalability of MPI-Tile-IO

Scalability -- Figure 9 shows the best performance of MPI-Tile-IO collective write using ParColl with a varying number of processes. As shown in the figure, compared to the Cray implementation, ParColl provides much better scalability with an increasing number of processes. This improvement is nearly proportional to the number of processes. With 1024 processes, ParColl can significantly improve the performance of MPI-Tile-IO up to 11.4GB/sec, which is 416% of the reachable bandwidth 2.7GB/sec using the Cray implementation. A similar trend has been observed for collective read. These results indicate that ParColl is beneficial to the scalability of collective I/O operations for the very large scale of Cray XT platforms such as Jaguar.

5.3 NAS BT-IO

NAS BT-IO [30] is an I/O benchmark that tests the output capability of NAS BT (Block-Tridiagonal) parallel benchmark. It is developed at NASA Ames Research Center. Data sets in BT-IO undergo diagonal multipartitioning and then distributed among MPI-processes. The data structures are represented as structured MPI datatypes and written to a file periodically. There are several different BT-IO implementations, which vary on the way the file I/O is performed. In our experiments, we used an implementation that performs I/O using MPI-IO collective I/O routines, so called *full mode* BT-IO.

Figure 10 shows the performance of BT-IO, Class C with different number of processes. Compared to the Cray MPI-IO implementation, ParColl significantly improves the performance of collective I/O. For the class C program of BT-IO, the best I/O performance is achieved with 576 processes, which represents a good point of tradeoff between the number of processes and the granularity of the I/O requests. These results suggest that ParColl is beneficial to the scientific I/O pattern as exhibited by BT-IO, for any number of processes. Note that BT-IO represents the type of complicated I/O patterns that require the use of intermediate file views in ParColl. So our BT-IO experiments suggest that ParColl brings significant benefits to such patterns as shown in Figure 4 (c).

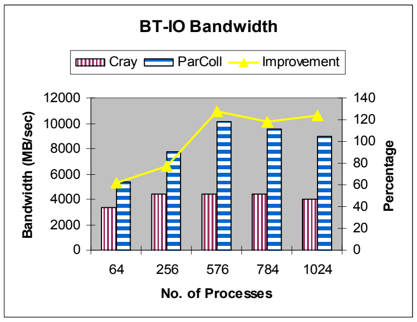


Figure 10 The Performance of BT-IO with ParColl

5.4 Flash

Flash is an application that simulates astrophysical thermonuclear flashes. It is developed in part at the University of Chicago by the DOE-supported ASC Alliance Center for Astrophysical Thermonuclear. Flash I/O [1] benchmark is the I/O kernel of the Flash program and measures the performance of its parallel HDF5 [27] output. MPI-IO is used internally in the HDF5 library. Three different output files are produced in Flash IO: a checkpoint file, a plotfile with centered data, and a plotfile with corner data. The checkpoint file is written through in the HDF5 [27] data format. It consumes the bulk of the I/O time. We evaluated Flash I/O with its memory structure being a 3D array of size 32x32x32. This results in a checkpoint file of 60.8GB for 128 processes, and 486GB for 1024 processes.

Figure 11 shows the I/O Bandwidth in writing its checkpoint file from a 1024-process Flash I/O program over Jaguar. The series denoted with *default* shows the performance of flash I/O under the default selection of I/O processes. Compared to the Cray implementation, ParColl with 64 subgroups can improve the I/O bandwidth by 38.5% for the default selection of I/O aggregators. The performance benefits of ParColl for Flash I/O is relatively smaller compared to MPI-Tile-IO and BT-IO. This is because the I/O requests in Flash I/O are of larger sizes, fewer segments compared to those in the other two programs. Thus the cost of synchronization in Flash I/O has relatively less impact. On Cray XT, there have been studies suggesting I/O for very large scale applications needs to be carried out with fewer I/O aggregators [33]. We have measured the performance of ParColl with only 64 I/O processes selected as I/O aggregators. Figure 11 shows that ParColl can also improve Flash I/O with a different number of I/O aggregators.

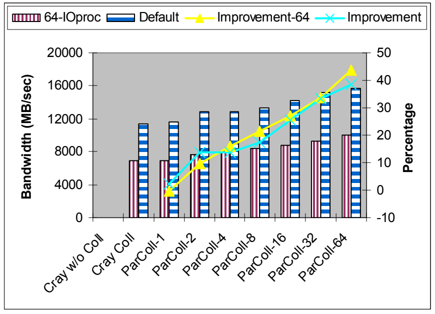


Figure 11 The Performance of Flash IO

Note that we have also measured the bandwidth of writing Flash I/O checkpoint file without enabling collective I/O, which is denoted in the figure as "Cray w/o Coll". The resulting bandwidth of Flash I/O is only around 60MB/sec. This suggests that it is very important to use collective I/O for writing the Flash I/O checkpoint file.

6 Conclusions

In this paper, we take on the challenge of optimizing collective I/O on Cray XT. We have examined the processing costs of collective I/O, and uncovered the collective wall that is caused by the required synchronization in the extended two-phase protocol of MPI-IO. We have introduced a novel technique called partitioned collective I/O (ParColl) to augment and optimize the original two-phase collective I/O protocol. Accordingly, we have introduced new mechanisms for file area partitioning, I/O aggregator distribution and intermediate file views. Our experimental results indicate that ParColl significantly improves the I/O performance and scalability of various benchmarks and scientific applications, such as BT-IO and Flash I/O [1].

In the future, we plan to carry out a comprehensive study on the collective wall problem over other massively parallel platforms with different underlying file systems, such as GPFS [21] and PVFS [3]. We also plan to study how to better adapt collective I/O for the new era of multi- or many-core processing. For example, we will study how to consolidate I/O requests from different cores to maximize the utilization of in-core bandwidth for collective I/O, and how to adaptive choosing the best group size for ParColl.

Acknowledgments

This research is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725. We are also very thankful to the anonymous reviewers for their detailed constructive comments.

References

[1] FLASH I/O Benchmark Routine -- Parallel HDF 5,
[2] IOR Benchmark, http://www.llnl.gov/asci/purple/benchmarks/limited/ior.
[3] The Parallel Virtual File System, Version 2, http://www.pvfs.org/pvfs2.
[4] TOP 500 Supercomputers, http://www.top500.org/.
[5] MPI-2: Extensions to the Message-Passing Interface, 1997.
[6] Argonne National Laboratory, ROMIO: A High-Performance, Portable MPI-IO Implementation,
[7] R. Brightwell, W. Camp, B. Cole, E. DeBenedictis, R. Leland, J. Tomkins, and A. B. MacCabe, Architectural specification for massively parallel computers: an experience and measurement-based approach: Research Articles, *Concurr. Comput. : Pract. Exper.,* vol. 17, 2005.
[8] R. Brightwell, K. T. Pedretti, K. D. Underwood, and T. Hudson, SeaStar Interconnect: Balanced Bandwidth for Scalable Performance, *IEEE Micro,* vol. 26, pp. 41-57, 2006.
[9] Bull Direct, French Atomic Energy Authority (CEA) takes delivery of Tera-10, http://www.bull.com/bulldirect/0601/hot.html
[10] Cluster File System, Lustre: A Scalable, High Performance File System,
[11] K. David, Disk-directed I/O for MIMD multiprocessors, *ACM Trans. Comput. Syst.,* vol. 15, pp. 41-74, 1997.
[12] DDN, Products: S2A9550,
[13] P. M. Dickens and R. Thakur, Improving Collective I/O Performance Using Threads, in Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing: IEEE Computer Society, 1999.
[14] M. Fahey, J. Larkin, and J. Adams, I/O Performance on a Massively Parallel Cray XT3/XT4, in *22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS'08)*, Miami, FL, 2008.
[15] R. Juan Miguel del, B. Rajesh, and C. Alok, Improved parallel I/O via a two-phase run-time access strategy, *SIGARCH Comput. Archit. News,* vol. 21, pp. 31-38, 1993.
[16] W.-k. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and S. Tideman, Collective caching: application-aware client-side file caching, in *High Performance Distributed Computing (HPDC-14)*, 2005.
[17] National Center for Computational Sciences, http://nccs.gov/computing-resouces/jaguar/,
[18] K. Pedretti, R. Brightwell, and J. Williams, Cplant" Runtime System Support for Multi-Processor and Heterogeneous Compute Nodes, in Proceedings of the IEEE International Conference on Cluster Computing: IEEE Computer Society, 2002.
[19] R. B. Ross, Parallel I/O Benchmarking Consortium,
[20] Sandia National Laboratories, Scalable IO,
[21] F. Schmuck and R. Haskin, GPFS: A Shared-Disk File System for Large Computing Clusters, in *FAST '02*, 2002, pp. 231-244.
[22] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi, Grid Datafarm Architecture for Petascale Data Intensive Computing, in Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid: IEEE Computer Society, 2002.
[23] R. Thakur and A. Choudhary, An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays, *Scientific Programming,* vol. 5, pp. 301-317, Winter 1996.
[24] R. Thakur, W. Gropp, and E. Lusk, An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces, in *Proceedings of Frontiers '96: The Sixth Symposium on the Frontiers of Massively Parallel Computation*, 1996.
[25] R. Thakur, W. Gropp, and E. Lusk, Data Sieving and Collective I/O in ROMIO, in *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, 1999, pp. 182-189.
[26] R. Thakur, W. Gropp, and E. Lusk, Optimizing noncontiguous accesses in MPI– IO, *Parallel Computing,* vol. 28, pp. 83-105, 2002.
[27] The National Center for SuperComputing, HDF5 Home Page,
[28] G. M. Timothy, S. David, and R. W. Stephen, A TeraFLOP Supercomputer in 1996: The ASCI TFLOP System, in Proceedings of the 10th International Parallel Processing Symposium: IEEE Computer Society, 1996.
[29] J. S. Vetter, S. R. Alam, T. H. Dunigan, Jr.,, M. R. Fahey, P. C. Roth, and P. H. Worley, Early Evaluation of the Cray XT3, in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Greece, 2006.
[30] P. Wong and R. F. Van der Wijngaart, NAS Parallel Benchmarks I/O Version 2.4, NASA Advanced Supercomputing (NAS) Division NAS-03-002, 2002.
[31] H. Yu, R. K. Sahoo, C. Howson, et al., High performance file I/O for the Blue Gene/L supercomputer, in *High-Performance Computer Architecture (HPCA-12)*, Austin, Texas, 2006.
[32] W. Yu, J. S. Vetter, and R. S. Canon, OPAL: An Open-Source MPI-IO Library over Cray XT, in *International Workshop on Storage Network Architecture and Parallel I/O (SNAPI'07)*, San Diego, CA, 2007.
[33] W. Yu, J. S. Vetter, and H. S. Oral, Performance Characterization and Optimization of Parallel I/O on the Cray XT, in *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, Miami, FL, 2008.