2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

Filesystem Aware Scalable I/O Framework for Data-Intensive Parallel Applications

Rengan Xu
Department of Computer Science
University of Houston
Houston, USA
uhxrg@cs.uh.edu

Mauricio Araya-Polo
Geophysics Development
Repsol
Houston, USA
araya.mauricio@repsol.com

Barbara Chapman
Department of Computer Science
University of Houston
Houston, USA
chapman@cs.uh.edu

***Abstract*—The growing speed gap between CPU and memory makes I/O the main bottleneck of many industrial applications. Some applications need to perform I/O operations for very large volume of data frequently, which will harm the performance seriously. This work's motivation are geophysical applications used for oil and gas exploration. These applications process Terabyte size datasets in HPC facilities [6]. The datasets represent subsurface models and field recorded data. In general term, these applications read as inputs and write as intermediate/final results huge amount of data, where the underlying algorithms implement seismic imaging techniques. The traditional sequential I/O, even when couple with advance storage systems, cannot complete all I/O operations for so large volumes of data in an acceptable time range. Parallel I/O is the general strategy to solve such problems. However, because of the dynamic property of many of these applications, each parallel process does not know the data size it needs to write until its computation is done, and it also cannot identify the position in the file to write. In order to write correctly and efficiently, communication and synchronization are required among all processes to fully exploit the parallel I/O paradigm. To tackle these issues, we use a dynamic load balancing framework that is general enough for most of these applications. And to reduce the expensive synchronization and communication overhead, we introduced a I/O node that only handles I/O request and let compute nodes perform I/O operations in parallel. By using both POSIX I/O and memory-mapping interfaces, the experiment indicates that our approach is scalable. For instance, with 16 processes, the bandwidth of parallel reading can reach the theoretical peak performance (2.5 GB/s) of the storage infrastructure. Also, the parallel writing can be up to 4.68x (speedup, POSIX I/O) and 7.23x (speedup, memory-mapping) more efficient than the serial I/O implementation. Since, most geophysical applications are I/O bounded, these results positively impact the overall performance of the application, and confirm the chosen strategy as path to follow.**

***Keywords*-Parallel I/O, Parallel File System, Dynamic Load Balancing**

I. Introduction

Most industrial applications found that their performance bottleneck is I/O rather than computing. The reason is at least twofold: these applications are data intensive and/or the widening performance gap between the processor and memory or secondary storage. In our case, both mentioned issues are present, we are dealing with Terabytes like datasets, which not even modern HPC data storage systems can handle proper. Our target applications are already optimized in terms of computing (including parallelization), but not regarding I/O. Actually, further we enhanced the computing performance wider the gap with I/O performance becomes, which implies that most of the performance gains are shadowed by the storage stack. Our research focused on I/O improvements that close the aforementioned gap. The parallel nature of the target applications is one of the obvious path to follow. I/O operations in parallel applications can be performed by either threads or MPI processes. In this context, we will only consider MPI processes, but our results and conclusions are general, the same ideas can be exploited by other parallel constructs.

The traditional way to perform I/O operations is serial, which means multiple processes perform I/O operations on the same file alternatively. Also, an I/O coordinator process scheme can be deploy, where only the coordinator process is allowed to access a file after other processes have transferred the data to it. This has negative effect since when one process is accessing a file, other processes are waiting to complete all their I/O operations. An effective way to solve this issue is to use parallel I/O. The idea is simple, multiple processes access the same file simultaneously. However, parallel I/O implementations are not trivial, since relies on HW capabilities that it is preferable to hid to the application. Therefore, a parallel I/O system is comprised of several components, including the high-level application, parallel I/O interface, parallel file system and the underlying RAID-capable [9] storage system. Our approach targets geophysical applications, to which we introduce POSIX I/O and memory-mapping interfaces, where Panasas parallel file system and RAID 1/5 storage hardware are the underlying storage platform. We develop a general dynamic load balancing framework, and introduce an I/O coordinator that handles I/O requests from compute nodes, in order to reduce the communication and synchronization overhead among compute nodes. Before to go into details of our approach, in Section II we review the state of the art related to the problem at hand. In Section III we describe our approach in details. In Section IV the experimental results are introduced and discussed. Finally, we close this work in Section V with our conclusions.

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.196

II. Related Work

PLFS [7] is a interposition layer inserted between parallel application and underlying parallel file system. It convertes a N-1 write access pattern into a N-N write access pattern to improve write bandwidth. Each write appends the data into corresponding data file and appends a record into the appropriate index file. This model would generate lots of directories, data files and indices if thousands of processes are used in an application, which leads to complex files management work, while our model only generate one data file and one index. ADIOS [13] provides a flexible approach for I/O within scientific codes. The end user can choose different I/O routines in external XML metadata file without touching the source code. The richly annotated XML file provides the entry of several well-tuned transport methods (POSIX, MPI I/O, pnetCDF [4] and HDF5 [1], etc.) for different platforms. ADIOS provides an abstraction of different I/O APIs, but unlike us it didn't consider the execution model of parallel applications. Darshan [8] is an characterization tool that can capture the applicatino I/O behavior, e.g. patterns of access within files, with minimum possible overhead. By characterizing four scientific applications, it demonstrated its effective scalability and viability for 24/7 characterization of petascale I/O workloads. IISche et al. [10] integrated the I/O Forwarding Scalability Layer (IOFSL) [5] with the VampirTrace/OTF toolset [11]. They added a distributed atomic file append capability into the I/O forwarding layer. IOFSL provides a client-server communication model and the clients simply need to deliver the data to to be written into the file to the server. The server returns to the client the file offset where the data was written. In this model servers can write files simultaneously, but it is limited by the number of servers. Lusk et al. [14] implemented an asynchronous dynamic load balancing (ALDB) library in which there are several servers and many application processes. This library solves load balancing problem but does not consider parallel I/O issue. Latham et al. [12] implemented MPI I/O shared file pointers within ROMIO [17]. Their algorithm defined two shared data structures N-byte *waitflag* and MPI_Offset-sized *sharedfp* and store them on a single process. The communication among all processes was done by one-sided communication.

Benchmarking plays an important role in our work, and so in this paragraph we present a couple benchmark suites. IOZone [3] benchmark can be used to test the throughput of a file system, and IOR [2] can benchmark the I/O bandwidth of a parallel file system with POSIX, MPI I/O and HDF5 interfaces.

III. Methodology

In this section we introduce our approach, which is based on two main aspects: scheduling (subsections III-A) and parallelism (III-B). Then we review implementation/performance ideas (subsection III-C and III-D). Finally, we discuss relevant considerations regarding our storage system (subsection III-E).

A. Task Scheduling Strategy

One of most important step during application parallelization is to determine how to schedule the tasks into processes. The scheduling can be both static and dynamic. Static scheduling include block distribution, cyclic distribution and block-cyclic distribution and so on. This scheduling determines the number of workloads for each process at the beginning of execution and it will not change in runtime.

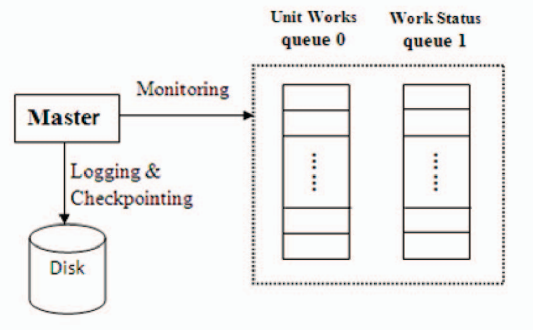

Figure 1. Dynamic Load Balancing Framework, master process view

In theory, this is suitable for the cluster in which the configuration of all nodes are exactly the same. In reality, however, we found that even if all nodes in a cluster are identical, many other factors will disturb the execution of application. As a result, even when the assigned workloads for all nodes are the same, the nodes will not finish their work at the same time. The gap between their finish time is huge when the execution time of application is long enough and all nodes use shared resources (such as storage systems and networking). The nodes who finish earlier need to wait for the nodes who finish later, which will waste the system resource. The dynamic scheduling is required for large application so that the scheduling is dynamic in runtime and the nodes do not need to wait for each other.

Figure 1 shows the dynamic load balancing framework which uses Master/Worker model. The master process is responsible to keep the workload balanced among all workers. At the beginning of the application, the whole workload is partitioned into many unit works and put into a unit works queue. Then the master assigns one unit of work from the work queue to each worker. Whenever a worker finishes its computation and writes the corresponding data, it will send a message to the master to notify it that its unit work is done. As a result, the worker whose is processing faster will get more work, whereas those processes who are slower will get less work. This approach ensures a good load balancing over heterogeneous nodes, especially when the computational demands for each unit work are different, which actually the case with the geophysical applications

at hand. The master keeps monitoring the status of the work queue and put their status into another queue. The work status queue is written into storage periodically by the master for restarting the application in case of failure (checkpointing), an important feature on a large distributed system where one node may be down anytime during the execution.

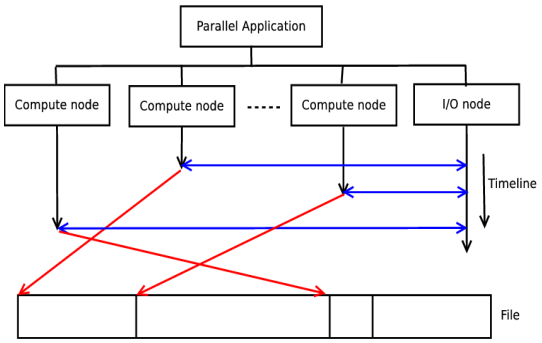

Figure 2. Parallel I/O Design in Target Applications

B. Our Parallel I/O Approach

In most parallel applications, the general steps are reading data, computing and then writing the result data. In our case, the input data is usually composed by many files (seismic data) with an aggregate size of under 500 TeraBytes (TB), the resulting data is usually a limited number of huge files, aggregate size of under 50 TB (seismic models and images). Since multiple processes are working (up to one process per computational node core, and up to 500 computational nodes, where every process writes GigaBytes (GB) of data), the writing to the share output files is usually serialized to avoid contention, or the data is collected into one process and written down by that process. In both situations, the writing is sequential which greatly limits the performance of the application. Therefore, parallel I/O is a must to reduce this bottleneck. With parallel I/O, different processes write to different portions of a common file in parallel. The key point is to know the position to write for each process. However, in some of our target applications, each process does not know how much data it needs to write until the computation is finished. Therefore, each process is not able to locate the position to write its own data within the share files. We can assume that the maximum size of data each process will write is known, so one solution is to create a file with maximum size for all processes. Then each process can write data to its own region. This solution, however, requires huge space in storage, and in average the processes will not fully occupy their portion. Also, as a result, the actual written data is a not contiguous GB to TB file. To make the file compact, each process should write at the end position of another process. In this way, the written data is contiguous in the file. And to maintain such contiguous property, communication and synchronization are required for all processes to exchange their data information before writing.

The solution needs to reduce the communication and synchronization overhead. Therefore, our approach is to add another process (in a specific node) that only coordinates I/O operations, thus only one node handles I/O requests. Figure 2 illustrates this idea. This diagram shows that once a process has finished its computation, it will submit a writing request to I/O node informing it how much data it needs to write, then the I/O node replies back with the writing position and then update the new global position. Thus, essentially implementing a I/O FIFO mechanism, where all compute nodes write data directly and in parallel. Algorithm 1 shows the detailed implementation. Thanks to this approach, the synchronization overhead among all compute nodes is eliminated and the only communication left is between compute nodes and I/O node. Note that the whole application still uses dynamic load balancing framework, then the master becomes I/O node and workers become compute nodes. Since the writing order is random, another auxiliary file is needed to to record the writing order and position of each worker writing. When the application needs to read this file, only trivial preprocessing is needed.

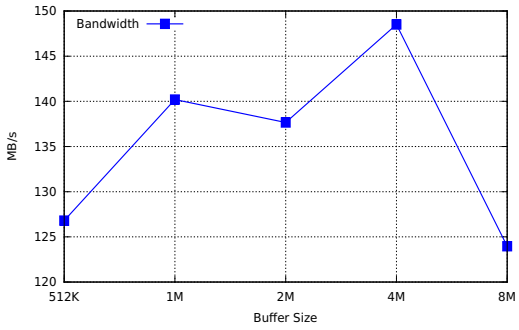

Figure 3. Bandwidth with Different Buffer Size

C. Buffered I/O

To reduce I/O system calls and increase efficiency, we use buffered I/O. The buffer size should neither be too small nor too large, so we need to explore the best buffer size that fits well within the system. Since we use parallel file system, the buffer size should be both page-aligned (multiple of 4 KB in our system) and stripe-aligned (multiple of parity stripe width). From subsection III-E, we know that the parity strip width size is 512 KB. The reason to be stripe-aligned is also explained in subsection III-E, but boiled down to avoid the stripe lock contention. The typical buffer sizes are 512 KB, 1 MB, 2 MB, 4 MB and 8 MB. To choose the one that gives best performance, we write 1 GB of data by one process with different buffer sizes. Figure 3 shows the result of this

Algorithm 1: Framework of Parallel I/O Approach

Input: Input data of an application
Output: Output result file of the application
I/O Node (Master):
Determine total number of unit works;
forall *workers of compute nodes* **do**
  Send one unit work to the worker;
  Get next unit work;
end
Initialization of data structures for the auxiliary file;
while *unit work <total works* **do**
  Receive the local I/O size of a worker from any source;
  Determine the worker from MPI status;
  Record the order and local I/O size of that worker into auxiliary file;
  Get current global position and send message to that worker with I/O tag;
  Update the new global position in the result file;
  Send a new unit work to that worker;
  Get next unit work;
end
`/* No availale work anymore, receive results from workers */`
forall *workers of compute nodes* **do**
  Receive the local I/O size of a worker from any source;
  Determine the worker from MPI status;
  Record the order and local I/O size of that worker into auxiliary file;
  Get current global position and send message to that worker with I/O tag;
  Update the new gloabl position in the result file;
end
forall *workers of compute nodes* **do**
  Send message with EXIT tag to the worker;
end

Compute Nodes (Workers):
`/* Keep receiving messages from master until informed to exit */`
while *true* **do**
  Receive message from master and determine the message tag;
  if *tag == WORK* **then**
    Read input data, do computation and get partial result data;
    Send message with I/O tag to master telling it how much data it needs to write;
    Receive message with I/O tag from master to get the global file position to write;
    Start to write data with caching optimization;
  else
    `/* The tag is EXIT */`
    break;
  end
end

experiment and it indicates that 4 MB is the best buffer size as it allows the highest bandwidth.

D. POSIX I/O and Memory-mapped file

POSIX I/O is the most basic I/O interface which uses *lseek()* to locate the position in a file, and uses *read()* and *write()* to read data from and write to a file. Memory-mapped file uses memory mapping technique to map a file on disk, byte-to-byte to the address space of a process. This technique does not need to use *lseek()*, *read()* or *write()* system calls. After mapping the file, we can access this file as simple as accessing memory. In addition, multiple processes can map the same file into memory so that the data is shared among all processes. However, the memory mapping size must always be multiple of page size, or the difference between the memory mapped size and the actual file size would be wasted. It also has page fault overhead when first accessing the file. This is because after mapping the file, the mappping exists only in the process' virtual memory and system has not allocated main memory for that file. When the file is first accessed, it produces a page fault error then the system load the file from disk to memory. We will use both of these two interface (POSIX I/O and memory-mapping) to test parallel I/O performance.

E. Storage System Considerations

Since our storage system is based on Panasas products, we look for optimize its utilization. During I/O operations, the data on a client (compute node) can be classified in two categories: clean data and dirty data. Clean data means that the data which is in the cache of a client is the same as the data currently stored on storage, while dirty data means the data in the cache of a client is different than the data currently stored on storage. Panasas supports two writing modes, Read/Write (RW) and Concurrent Write (CW) mode. In RW mode, multiple clients have opened the file and at

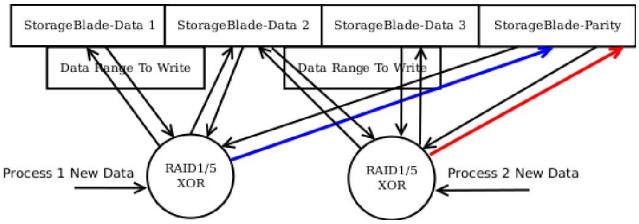

Figure 6. Stripe Lock Contention Example

least one of them is writing to the file. It is not able to cache the clean data since the file is changing, and the write client is not able to cache its dirty data since other clients need to read from this file. To relax RW mode, Panasas also supports another advanced caching mode called CW mode, in which every client does not need to see the latest data that other clients have written. To test the performance

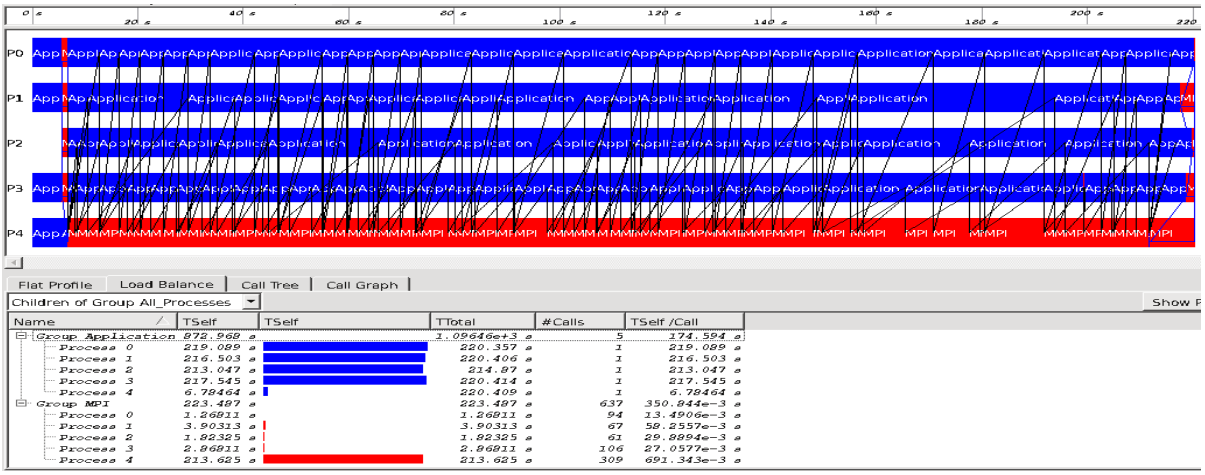

Figure 4. Load Balanced Profiling Result

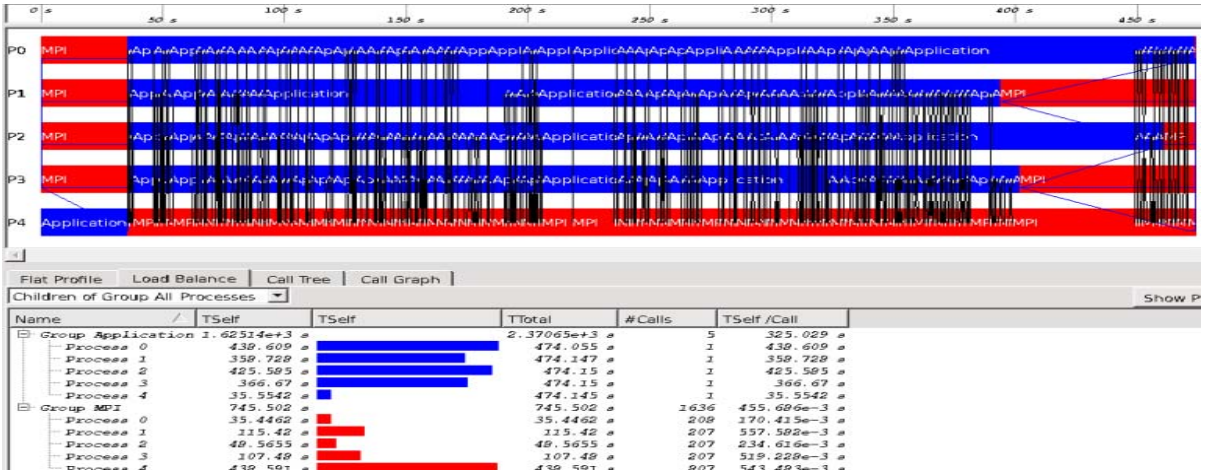

Figure 5. Load Unbalanced Profiling Result

difference between RW and CW mode, 4G data is written by two processes under the above described modes. As we expected, the performance of CW mode (147.86 MB/s) is much better than RW mode (11.54 MB/s). The reason is that it does not need consistent view among all clients, every client just needs to keep the correct view of its own portion of data. We use CW mode in all of our parallel writing experiments.

Even within CW mode, when two processes write close to each other, performance still may be degraded. Figure 6 shows such an example. In this example, the parity stripe width is 4 and one of blades stores parity. Process 1 writes its data into part of storage blade 1 and 2, and process 2 writes its data into part of storage blade 2 and 3. Because parity needs to be updated, process 1 reads old data 1, 2, old parity and its new data to update the new parity, in the mean while process 2 also needs to read corresponding old data, old parity and its new data to update the same parity. The conflict appears because they want to update the same parity simultaneously. To guarantee the correctness of new parity, the parallel writing would be serialized internally by a ”stripe lock”.

To avoid such lock contention, writes should be stripe aligned which means the size of each writing should be multiple of parity stripe width size. Notice that parity strip is only logical view which depends on the total number of available storage blades in the hardware. In our case, after some modulo operation, we will get stripe width size as 512 KB. In Panasas, large file is stored in two-level RAID 1/5 groups [15], each RAID 1/5 is a shelf which has 1 director blade and 10 storage blades. We have three shelves, so there are 30 storage blades in total. In our hardware, two logical spare-equivalent blades are configured for rebuild purpose, therefore 28 storage blades could actually be used for parallel writing in programmer’s view. The typical parity stripe width are 8, 9, 10 and 11. We module 28 against

these stripe widths and choose the one that has the smallest remainder. Based on this modulo operation , we choose 9 as the stripe width because it has the least remainder. Because one storage blade of the parity strip needs to store parity and each strip unit size is 64K, we got stripe width size as $64K * (9-1) = 512K$.

IV. Experimental Set-up and Results

A. Experimental Set-up

For all carried out experiments described Panasas parallel file system has been used. This system is confugred as a RAID 1/5 which is the combination of RAID 1 and RAID 5. In Panasas, each shelf is RAID 1/5 which has 1 DirectorBlade and 10 StorageBlades [18] and each blade has two disks. The shelves are connected to the network with a 10 Gbit/s Ethernet switch which also connect to the cluster. The compute nodes are connected by Infiniband (bandwidth is 40 Gbit/s). All parallel I/O experiments use the dynamic load balancing parallelization framework described in Section III-B. Multiple processes read from or write to non-overlapping locations in one file. We always use one MPI process per node because the applications use OpenMP or pthreads [16] to parallelize the computing within node. All experiment results are averaged value after removing the outliers.

Table I
Configuration of Each Node in the Cluster

Item	Description
Machine Type	x86_64
CPU Model	Intel Xeon X5675
CPU Cores	12 (6 x 2 sockets)
CPU Speed	3.07GHz
Memory Total	48G

We tested the parallel I/O performance for 100 GB data file with 1, 2, 4, 8 and 16 nodes. The nodes number in the result figures means compute nodes, excluding I/O node since it does not perform any I/O operations. When memory-mapping technique is used, since there is a limit for memory-mapping size and we use dynamic load balancing framework, for each iteration we map only 256 MB into main memory.

B. Experimental Results

Figure 4 shows an example profiling result of our approach. The test case uses 4 compute nodes and 1 I/O node with memory-mapping interface. The application has an initial synch point because at first the I/O node needs to create a file for other compute nodes to open, then a synchronization is needed before accessing this file. The I/O node then keeps waiting for messages from compute nodes and replies to them. It can be observed that there is no synchronization in the middle of writing, and compute nodes does not need to communicate to each other. Finally, they almost finish at the same time point. We can see that the workload is distributed to all compute nodes in balance. To compare our scheduling with static scheduling strategy, Figure 5 shows an example profiling result of using static scheduling in which the tasks for each compute node are determined at the beginning. The profiling result indicates that some processes are idle and waiting for other processes after they finish their own work, although they could do the rest work. It is obvious to conclude that our scheduling is better than static scheduling.

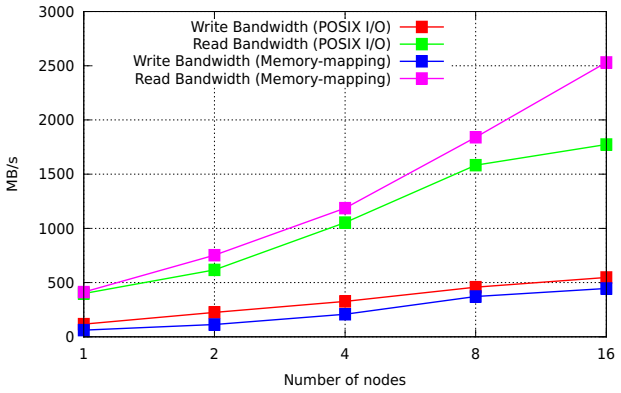

Figure 7. Bandwidth of Parallel I/O

Figure 7 shows the bandwidth of parallel I/O experiments when using both POSIX I/O and memory-mapping. The result indicates that with both interfaces, the performance of both read and write operations increase with more nodes. The read bandwidth reaches the theoretical peak performance (for our platform) starting from 8 nodes with POSIX I/O and 4 nodes with memory-mapping. The read operation performance is around 3-4 times the write operation performance. The reason why writing is slower than reading is that writing pushes the data into storage, while reading pulls the data out of storage, and the cost of pushing is much higher than pulling. Further, writing operation has 11% (1/9) writing overhead, because we need to store parity in 1 out of 9 storage blades, this is a panasas configuration constraint.

Figure 8 shows the speedup of using multiple nodes with POSIX I/O and memory-mapping. With both interfaces, the speedup of both read and write scales when the number of nodes increases. With POSIX I/O, the speedup of write is slightly better than read although the bandwidth of write is lower than read. The reason why both read and write have not reached the ideal speedup is that we are executing a real production level application with our framework, then the execution time includes the computing time of the application rather than pure I/O operations. Also, time measurements include the overhead of communication between compute nodes and I/O node, the status queue management and the memory allocation and de-allocation for data buffer, etc. Further, the data must go through the

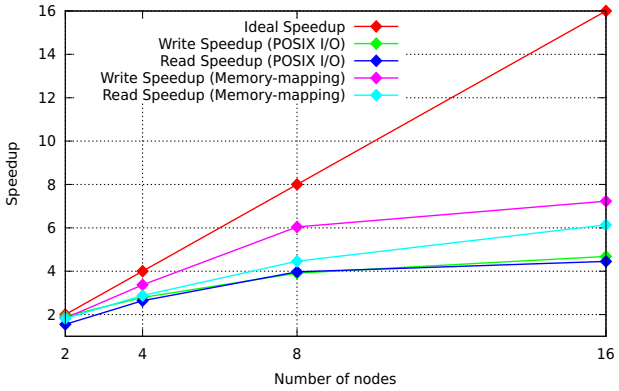

Figure 8. Speedup of Parallel I/O

network between the cluster and storage system, and this network is shared by all users of the cluster. Thus, we carried out the experiments in shared situation which means other users were also performing I/O operations at the same time.

We noticed that with the same number of nodes, the read bandwidth with memory-mapping is higher than that with POSIX I/O. The reason may be that the paging operation of the memory-mapping is much more efficient than *read()* system call. However, memory-mapping write bandwidth is lower than that with POSIX I/O, the reason is that the high overhead of page fault error while loading the file from disk to memory. In our experiments, we map only one portion of file into memory and write only once. The memory-mapping is the way to go when one portion of file is mapped to memory and then that portion is frequently accessed in memory. Also, it can be observed that with memory-mapping, write speedup is higher than read in almost all cases.

Table II
ELAPSED TIME COMPARISON BETWEEN DIFFERENT APPROACHES (TIME IN SECONDS), DATASET SIZE 100 GB, ONLY WRITING OPERATIONS NO READ OPERATIONS INCLUDED. IN [1] (DISK-BASED STRATEGY) EVERY WORKER WRITES ITS OUTPUT TO THE STORAGE SYSTEM, THEN THE MASTER READS ONE-BY-ONE THOSE OUTPUTS FROM THE STORAGE SYSTEM AND PROCESS THEM, IN [2] (NETWORK-BASED STRATEGY) EVERY WORKER SENDS ITS OUTPUT TO THE MASTER, WHICH RECEIVE AND PROCESS THOSE OUTPUTS IN SERIAL FASHION

I/O Approach	Nodes			
	1	4	8	16
Serial[1] (POSIX I/O)	2007.19	2777.92	3805.56	5860.84
Serial[2] (POSIX I/O)	1770.28	1830.28	1910.28	2070.28
Parallel (POSIX I/O)	875.14	313.35	223.94	187.06
Serial[1] (Memory-mapping)	3574.04	4318.88	5312.01	7298.24
Parallel (Memory-mapping)	1662.88	493.09	275.45	229.94

Finally, in order to quantify the impact of our I/O approach into the application performance, we introduce Table II and Table III. It can be observed that overall execution time of the parallel write version is 31.3x (POSIX I/O) and 31.7x (Memory-mapping) faster than the serial write version, and parallel read version is 251.36x (POSIX I/O) and 669.86x (Memory-mapping) faster than the serial read version. This means a reduction of the execution time from hours to few minutes. It is noticed that the serial memory-mapping read is extremely time consuming which is because the data needs to be swapped into disk every time the master send data to each worker.

Table III
ELAPSED READ TIME COMPARISON BETWEEN DIFFERENT APPROACHES (TIME IN SECONDS), DATASET SIZE 100 GB, ONLY READ OPERATIONS NO WRITE OPERATIONS INCLUDED. IN [1] (DISK-BASED STRATEGY), THE MASTER READS THE INPUT DATA AND THEN SEND THE DATA SERIALLY TO EACH WORKER THROUGH THE DISK, IN [2] (NETWORK-BASED STRATEGY), THE MASTER READS THE INPUT DATA AND SENDS TO EACH WORKER THROUGH THE NETWORK.

I/O Approach	Nodes			
	1	4	8	16
Serial[1] (POSIX I/O)	1388.96	4014.38	7514.94	14516.06
Serial[2] (POSIX I/O)	276.91	336.91	416.91	576.91
Parallel (POSIX I/O)	256.91	97.18	64.71	57.75
Serial[1] (Memory-mapping)	2159.44	7148.08	13799.60	27102.64
Parallel (Memory-mapping)	248.28	86.33	55.64	40.46

V. CONCLUSION

This paper proposed a parallel I/O solution to parallel applications which manage large datasets. Our solution reduces the global synchronization and communication overhead among all processes significantly. We also adopted a dynamic load balancing framework which uses master/worker model to ensure load balancing among all processes, especially in a heterogeneous network. This framework can be widely applied in most parallel applications. Although we used a specific experiment environment, our approach is independent of any parallel file system and hardware. Adapting it to other platforms will only require to set the proper parameters, such as stripe unit size, parity stripe width size and buffer size. By using POSIX I/O and memory-mapping interfaces along with our framework, we have achieved impressing bandwidth and speedup results, in some cases close to peak platform performance. All this result in up 30x write improvement and at least 250x read improvement than the worst serial scenario on the overall execution time of the applications at hand, which greatly impact the projects turnaround when these applications are deployed. Facing the reality of *B*ig Data and widening gap between I/O storage and computing performance, simple but robust frameworks -like the one we proposed- can mitigate the I/O related bottlenecks, which will be a step forward towards a better balance computing platforms. Regarding future work, we will explore using multiple I/O nodes if the I/O requests to one I/O node is too intensive, this is the case when the total number of nodes goes beyond thousands or the test cases produces many work units. Also, we will extend our approach to the multi-core level, as there are some research focusing on message passing between threads,

which will enable finer level granularity, therefore open the framework for a wider set of scenarios in terms of testcases and algorithms.

ACKNOWLEDGMENT

The authors would like to thank Repsol for allow us to present this work, and University of Houston for constant support.

REFERENCES

[1] HDF5. http://www.hdfgroup.org/HDF5.

[2] IOR. http://sourceforge.net/project/ior-sio.

[3] IOZone. http://www.iozone.org.

[4] PnetCDF. http://trac.mcs.anl.gov/projects/parallel-netcdf.

[5] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable I/O Forwarding Framework for High-performance Computing Systems. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[6] Araya-Polo M., Cabezas J., Hanzich M., Pericas M., Rubio F., Gelado I., Shafiq M., Morancho E., Cela J.M., Valero M. Assessing Accelerator-Based HPC Reverse Time Migration. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):147 – 162, January 2011.

[7] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 21. ACM, 2009.

[8] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 Characterization of Petascale I/O Workloads. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[9] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[10] T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knüpfer, K. Iskra, R. Ross, W.E. Nagel, and S. Poole. Enabling Event Tracing at Leadership-Class Scale through I/O Forwarding Middleware. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 49–60. ACM, 2012.

[11] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M.S. Müller, and W.E. Nagel. The Vampir Performance Analysis Tool-set. *Tools for High Performance Computing*, pages 139–155, 2008.

[12] R. Latham, R. Ross, R. Thakur, and B. Toonen. Implementing MPI-IO Shared File Pointers without File System Support. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 84–93, 2005.

[13] 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*, pages 15–24. ACM, 2008.

[14] E.L. Lusk, S.C. Pieper, R.M. Butler, et al. More Scalability, Less Pain: A Simple Programming Model and its Implementation for Extreme Computing. *SciDAC Rev*, 17:30–37, 2010.

[15] D. Nagle, D. Serenyi, and A. Matthews. The Panasas Activescale Storage Cluster: Delivering Scalable High Bandwidth Storage. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53. IEEE Computer Society, 2004.

[16] B. Nichols, D. Buttlar, and J.P. Farrell. *Pthreads Programming*. O'Reilly Media, Incorporated, 1996.

[17] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32. ACM, 1999.

[18] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.