Performance Evaluation of Collective Write Algorithms in MPI I/O

Mohamad Chaarawi, Suneet Chandok, and Edgar Gabriel

Parallel Software Technologies Laboratory,
Department of Computer Science, University of Houston
{mschaara,schandok,gabriel}@cs.uh.edu

Abstract. MPI is the *de-facto* standard for message passing in parallel scientific applications. MPI-IO is a part of the MPI-2 specification defining file I/O operations in the MPI world. MPI-IO enables performance optimizations for collective file I/O operations as it acts as a portability layer between the application and the file system. The goal of this study is to optimize collective file I/O operations. Three different algorithms for performing collective I/O operations have been developed, implemented, and evaluated on a PVFS2 file system and over NFS. The results indicate that different algorithms promise the highest write bandwidth for different number of processors, application settings and file systems, making a one-size-fits-all solution inefficient.

1 Introduction

Many scientific applications utilizing parallel computers have to analyze tremendous amounts of data. The main challenge for such applications is the limited performance of individual magnetic hard drives, respectively of the entire I/O subsystem attached to typical clusters. Compared to the performance of CPU, memory and networking cards, a magnetic hard drive offers multiple orders of magnitude of higher latencies and lower bandwidths. Operating Systems try to hide the latency of file I/O operations by applying buffering and caching techniques, which show significant performance improvements for certain (regular) scenarios, but lead to performance degradation for applications having more irregular I/O patterns. In order to overcome the bandwidth limitations, most systems combine multiple disks to a single logical unit in a RAID configuration [1], such that files can be striped over multiple disks. The I/O performance of an application will thus depend on characteristics of the storage device, the file system utilized, the network interconnect used between compute nodes and the storage as well as in-between the compute nodes, and the I/O pattern of the applications.

The MPI 2 specification [2] includes routines for handling files in a parallel application, leveraging existing concepts from MPI 1, such as process groups and derived data-types. Among the most important features of the MPI I/O specification is the notion of a file view, which defines the portion of a file accessible to a particular process. Declaring a file view allows the MPI I/O implementation

G. Allen et al. (Eds.): ICCS 2009, Part I, LNCS 5544, pp. 185–194, 2009.

to pre-calculate offsets for the subsequent I/O operations from a process, detect overlap or the lack of overlap between individual processes, and potentially prefetch data items for read operations.

The MPI I/O interfaces offer both blocking and non-blocking operations to access a file, as well as the notion of individual vs. collective I/O operations. The latter offers the possibility to concatenate data from multiple processes, potentially avoiding a sequence of small, individual file requests, but post a single, large I/O request instead. ROMIO [3], the most wide-spread implementation of MPI I/O as of today, utilizes so-called two-phase collective I/O operations [4], which combines the data and posts I/O requests similarly to what we described in this paragraph. This approach has also been taken by some file systems to accumulate multiple user-level requests [5,6], and has been extended in various ways, e.g. to reduce the amount of meta-data required to be communicated between the processes by using derived data types [7] instead of lists of offsets. In [8], an adaptive approach for parameters that are passed as file hints is presented to optimize the performance of collective I/O operations on the SX vector computer with the GFS [9] file system. Yu et. al. exploit the file joining feature of Lustre to optimized collective write operations [10].

In this paper we analyze the performance characteristics of three different algorithms to implement collective write operations. Although the algorithms here can easily be transformed and used for read operations as well, we stick with write operations due to space limitations. The first algorithm is based on the two phase collective I/O algorithm described above, having the option to group internally the processes and thus vary the number of processes executing file I/O operations. The second is a modification of the two-phase collective I/O algorithm which does not optimize the file access to the hard drive, but the communication occurring during the shuffle operation. Lastly, the third algorithm avoids any communication between the processes and has each process handle its own I/O requests. We explore the performance behavior of these algorithms over a PVFS2 file system and over NFS, and compare them to the performance numbers achieved using ROMIO.

The remainder of the paper is organized as follows: section 2 gives details to the three algorithms explored in this paper. In section 3 we present the results obtained over PVFS2 and NFS. Finally, in section 4 we summarize this work and outline the future work in this area.

2 Collective Write Algorithms

This section describes the algorithms that have been implemented within this study. Two of the algorithm are derived from the two-phase collective I/O approach used in ROMIO, while the third algorithm acts similarly to individual I/O routines. The routines have been implemented in a stand-alone library utilizing the profiling interface of MPI to intercept the `MPI_Init` and the `MPI_Finalize` functions. The library implements a subset of the MPI I/O routines defined in the standard. In `MPI_Init` the library reads a configuration file which specifies

the algorithm to be used for collective I/O operations. This allows us to test various algorithms without having to recompile the library.

For simplicity of low level I/O operations, a file I/O layer is introduced consisting of four simple interfaces, namely open, read, write and close. Two different implementations for the lower level I/O operations are available as of today. The synchronous low level I/O functions use `pwrite/pread` while the asynchronous version relies on the `aio_write/aio_read` [11] operations. Both sets of routines eliminate the need for seek operations, since the user has to provide explicitly the offset into the file for every operation. Since our results did not show a significant impact on using the synchronous or asynchronous low-level I/O functions for the platforms tested, we stick for the sake of simplicity to the synchronous low-level I/O operations for the rest of the paper.

2.1 Dynamic Segmentation Algorithm with Multiple Writers

This algorithm follows mostly the two-phase collective I/O operations outlined in the introduction. Its main goal is to combine data from multiple processes in order to minimize the number of I/O operations presented to the file system and avoid rewinds on the disk if possible. In a first step, all processes share location information about the data to be written with each other. Using two `MPI_Allgather()` operations, all processes share the list of file offsets and number of elements to be written with each other within the given collective write operation. Thus, every process has the knowledge of the operations to be performed by every other process. All processes can than sort these lists in an ascending order of the file offsets. The lists are divided in cycles of operation, where in each cycle, a fixed number of bytes are written to disk. A process can calculate how many elements it has to contribute for the write operation in that cycle. Using an extended version of the `MPI_Gatherv()` function, each process sends its elements contributing in the current cycle to the writer process assigned to him.

Note, that there can be more than one writer process, depending on the number of writer processes defined in the configuration file. Each writer process would handle the I/O for a certain number of processes. For example, for 24 processes and 4 writers, the processes would be grouped such that processes $0-5$ will have the rank 0 as a writer, processes $6-11$ process 6 as a writer, processes $12-17$ process 12 as a writer and so on. The writers would gather all the information and data needed from the other processes in their group and perform the actual write to disk.

Figure 1 shows a case where three processes are writing collectively with a cycle size of five bytes using a single writer process, namely rank zero. It shows that the operation is performing sequential disk access under optimal conditions with each process contributing varying amount of data in each cycle. Furthermore, it highlights one of the important characteristics of this algorithm, namely the fact, that a process might not be actively involved in every cycle of this algorithm. Thus, while the algorithm optimizes the disk access operations, the communication pattern deployed is suboptimal due to the fact that the data

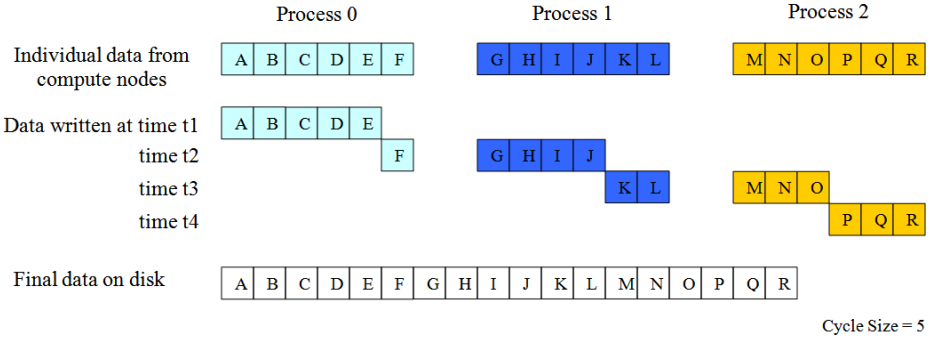

Fig. 1. Sketch of the write dynamic segmentation algorithm with a single writer

contributed by a particular process varies per cycle and thus does not make use of all communication channels in each cycle. The communication costs are affected however positively when using multiple writers in this algorithm, since it leads to the forming of smaller subgroups of process, which each execute internally a gather operation. Thus, the congestion typically occurring at the root process of the gather operation is avoided by having multiple root processes.

Since large write-all operations are typically executed in multiple cycles, one of the open questions is whether the size of the temporary buffer used for concatenating data from multiple processes, in the following referred to as the cycle buffer, shall be fixed and independent of the number of writer processes, or whether it should scale with the number of writers. We will evaluate both options in the subsequent results section.

2.2 Static Segmentation Algorithm

In this algorithm, data is gathered from all processes at a root process which will perform the low-level write operation. Data is written in fixed chunks, with the size of the chunk being a parameter of the configuration file read in `MPI_Init`. In contrary to the previous algorithm, the root process gathers a fixed number of bytes from all processes in each cycle. Thus, the algorithm does not necessarily reduce the overall number of I/O requests presented to the file system, but reduces the number of processes executing these I/O requests. Furthermore, due to the fact that every process contributes in every cycle a constant amount of data, this algorithm makes a better use of the communication resources in the cluster.

Figure 2 shows a scenario where three processes are writing collectively with cycle buffer size of two bytes. It shows that the operation is performed in three cycles. After the 1st cycle, the file pointer needs to be moved back for the next cycle. Note, that the algorithm does not support as of today the notion of multiple writers, i.e. one process is only allowed to execute I/O operations. Similarly to the dynamic segmentation algorithm, the question on whether to scale the cycle buffer with the number of processes or whether to keep it constant independently of the number of process used, arises.

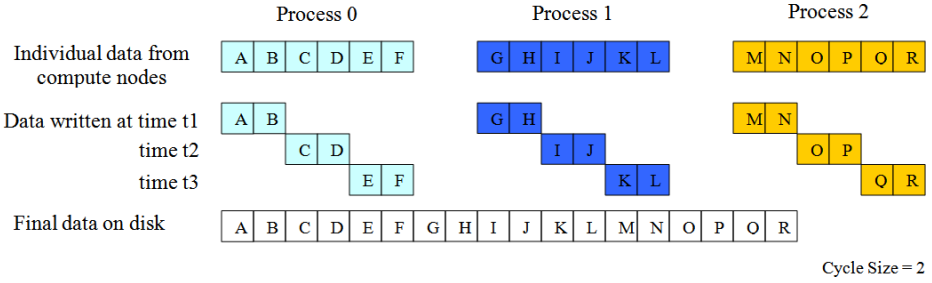

Fig. 2. Sketch of the write static segmentation algorithm

At first sight, this algorithm seems counterintuitive to the common knowledge which states, that file access operations are the most time consuming part of collective I/O operations. However, many large scale installations provide huge caches on the I/O nodes, which effectively decouple the compute cluster form the storage devices, and thus show - from the application perspective - virtually no sensitivity to irregular or strided file access patterns [12]. Furthermore, one of the distinctive features of a technology currently on the rise, solid state hard drives (SSD), is its insensitivity to irregular access in the file. For these two scenarios, the static segmentation algorithm optimizes the second most time consuming operation, namely the communication occurring during the shuffle step.

2.3 Individual Write

This algorithm avoids communication operations entirely and has each process write its data individually to the hard drive. The `MPI_File_write_all` operation exposes therefore the behavior that the application would face when using individual `MPI_File_write` operations instead of the collective version. The main difference compared to the latter approach is that the collective operation internally structures the I/O operations in cycles, similarly to the previous algorithms, in order to overlap the I/O operations with the calculation of the file offsets based on the file view and the merging of different segments.

Note, that this algorithm has also been extended by using a scheduling approach to control the number of processes concurrently performing I/O operations, and thus limit the burden on meta-data servers for some file systems. Due to space limitations we skip however this (fourth) algorithm.

3 Performance Evaluation

This section presents the performance evaluation of the three algorithms presented previously. For the dynamic segmentation algorithm using multiple writers, we explore the performance using 1, 2, 4, 8, 12, and 24 writers. We also compare these algorithms to the performance of ROMIOs `MPI_File_write_all` routine. The ROMIO algorithm has been executed without passing any additional hints to the library, since the main goal of using ROMIOs implementation

within this context is to have a baseline for the comparison of our algorithms. The cluster used for these tests consists of 24 single process dual-core AMD Opteron nodes and 5 dual-processor quad-core AMD Opteron nodes, providing a total of 88 compute cores. The nodes are connected by a 4x InfiniBand network interconnect. It has a parallel file system (PVFS2) mounted as '/pvfs2', which utilizes 22 hard drives, each hard drive is located on a separate compute node. The PVFS2 file system internally uses the Gigabit Ethernet network to communicate between the pvfs2-servers. The home file system on is NFS mounted from the front-end node.

We executed a simple benchmark in which processes collectively write *max_size* bytes of data for a given number of iterations to the file. The file view is determined by another parameter (*segment_size*), which allows to control the size of the data portion 'owned' by a process. To explain the correlation between the two parameters, consider an example, where *max_size* is set to 8 bytes, while *segment_size* is set to 2 bytes. In a four process test case, a call to `MPI_File_write_all` having to write *max_size* bytes of data per process would lead to process 0 having to write two bytes each at offsets (0,8,16,24), process 1 writing two bytes each at the offset (2,10,18,26) and so on. We measure the execution time of the test required to write **all** data to file, and calculate the bandwidth achieved by taking the size of the overall file created and overall execution time. Each test has been executed three times, and we present here the maximum bandwidth achieved through these runs.

3.1 Results Achieved over PVFS2

When writing over PVFS2, each process executes `MPI_File_write_all` operations writing 20MB of data per function call, writing all-in-all 1GB of data to file. Note, that we did perform tests with even larger files, which lead however to the same performance numbers that we present in this paper. Thus, the overall file size for the 24 processes test cases is 24GB and for the 48 processes test cases is 48 GB. Furthermore, we vary the segment size (2MB, 10MB, 20MB) and the cycle buffer size (1MB, 10MB, 20MB) for our tests.

The results of the first set of tests executed over PVFS2 are shown in Fig. 3. For both, the 24 and the 48 processes tests, the dynamic segmentation and the static segmentation algorithm were using scaling cycle buffers, as explained in the according subsections. Both graphs have in common, that the dynamic segmentation algorithm with 24 writers and the individual algorithm achieve the highest bandwidth. More generally, as the number of writers increase in the dynamic segmentation algorithm, so does the bandwidth achieved for those operations. The static-segmentation algorithm shows a bad performance in these tests. ROMIO is achieving a reasonably good performance, although the two top performing algorithms are significantly outperforming the default ROMIO version.

All algorithms show increasing performance with increasing segment sizes. The main reason for this is that the number of data blocks that have to be sorted and potentially merged is decreasing, since the size of each data block is

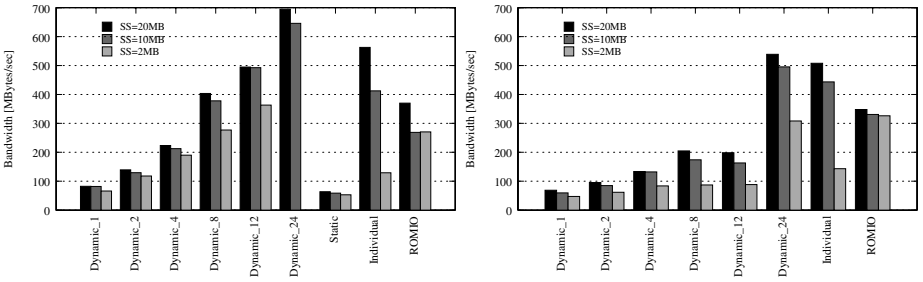

Fig. 3. Performance Comparison for 24 processes (left) and 48 processes (right) with varying the segment size and keeping the cycle buffer size constant at 20MB

Table 1. Performance Breakdown (in seconds)of the Dynamic Segmentation and Individual Algorithms over PVFS2 with 24 and 48 processes with segment size and cycle buffer size being 20 MB

	Gathering		I/O		Total	
	24	48	24	48	24	48
Dynamic_24	0.74	2.97	33.18	99.39	34.91	104
Dynamic_12	1.8	5.24	46.62	232.28	50.08	239.63
Dynamic_1	25.88	49.46	285.14	755.75	325.74	837.83
Individual	N/A	N/A	39.8	86.49	39.8	86.49

increasing. ROMIO shows a significantly lower sensitivity to the segment size than our algorithms, probably due to the usage of derived data types [7] instead of lists of offsets for the according operations.

Comparing the performance of the 24 and 48 process test-cases, it is notable, that the overall bandwidth of all algorithms drops when using two processes per node. Tabel 1 shows the performance breakdown of the dynamic segmentation and individual algorithms. Both cases (24 and 48 processes) show that most of the time is spent in I/O operations, and a smaller fraction is spent on the data gathering operations. However, the data also indicates, that although the data volume doubles between the two cases, both the I/O and the communication costs increase more severely, e.g. by a factor of 2-5. Note that for the 48 processes case, we were not able to get a result for the static segmentation algorithm due to the enormous amount of temporary buffer required for that scenario (960MB). With a fixed cycle buffer size, the algorithm could finished successfully, but performed poorly overall.

In Fig. 4, we analyze the effect of the cycle buffer size on the algorithms. For this, we execute the same write tests keeping the segment size constant at 20MB. The results show that increasing the cycle buffer size improves the performance but not by a huge factor. The individual algorithm is an exception there, where increasing the cycle buffer size does not necessarily yield a better performance.

Finally, table 2 shows the difference between fixed and scaling cycle buffer sizes when using the dynamic segmentation algorithm. The results show that

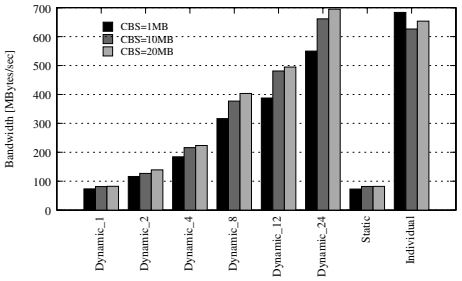

Fig. 4. Comparing the performance of some of the algorithms while varying the cycle buffer size and keeping the segment size constant at 20MB

Table 2. Bandwidth comparison (MB/sec) of the Dynamic Segmentation algorithm with Constant vs. scaling cycle buffer size of 20 MB and a segment size of 20 MB

	24 procs		48 procs	
	constant	scaling	constant	scaling
Dynamic_1	74.47	82.27	55.09	68.7
Dynamic_2	116.52	139.31	86.87	95.35
Dynamic_4	189.37	223.48	112.72	133.59
Dynamic_8	328.27	403.49	110.21	204.81
Dynamic_12	330.99	494.89	71.59	198.53
Dynamic_24	350.41	694.82	280.26	538.81

having each writer write the specified cycle buffer size in a cycle would be better than dividing the cycle buffer size over all the writers in each cycle. The reason would be that with the scaling cycle buffer size, the actual I/O is done with larger sequential chunks of data as compared with small chunks of the fixed cycle buffer size. Another common observation between the two approaches is that increasing the number of writers over PVFS2 still provides better results.

3.2 Results Achieved over NFS

Although NFS is considered to not be well suited for parallel I/O, it is as of today the most wide spread file system on small and medium size clusters. End-users experimenting with MPI I/O over NFS should not see a performance degradation compared to sequential POSIX I/O even on this file system, since this will discourage them from using MPI I/O in their codes.

In order to keep the execution time of our tests within a reasonable time frame, we set each process to write only 100MB of data. Tests have been executed with various cycle buffer size (1MB, 10MB, 20MB), and keep the segment size constant at 2MB.

The results that were gathered over NFS show significant deviations from the PVFS2 results. For the dynamic segmentation algorithm, the lower the number

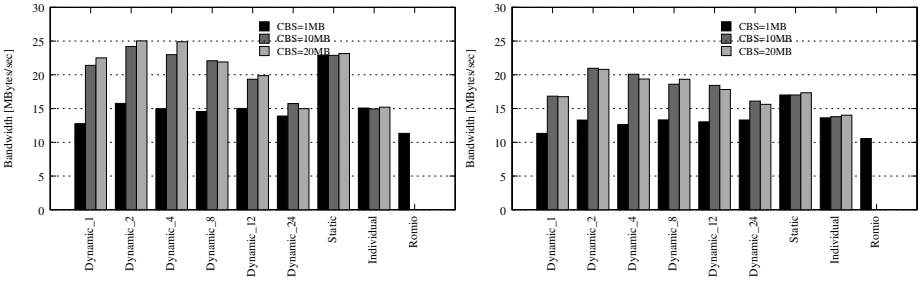

Fig. 5. Performance Comparison for 24 processes (left) and 48 processes (right) with varying the cycle buffer size and keeping the segment size constant at 2MB

of writers, the better the performance. The static segmentation algorithm using scaling cycle buffers performs well in that case, compared to the other algorithms. Most algorithms that we tested outperformed ROMIOs version in this setting. Since the cycle buffer size is not relevant for ROMIO, the graph shows only one bar for ROMIO. Independent of the algorithm used, the user still would observe a performance hit when using MPI I/O over NFS compared to the raw write performance of a single hard drive (35 MB/s sustained).

4 Summary

In this paper, we described three algorithms for MPI-IO collective write operations, the dynamic segmentation, static segmentation, and individual algorithms. The testing was done over two file systems, PVFS2 and NFS. The results show that there is a large room for optimizations within the collective I/O operations. The performance of the algorithms depended on the file system, number of processes and the file view (segment size) utilized by the processes, and lead in fact to different algorithms delivering the best performance.

Future work in this area includes further extending some of the algorithms described above, e.g. including the ability to have multiple writers for the static segmentation algorithms, and extend the individual algorithms by sophisticated scheduling approaches in between the processes to limit the burden on meta-data servers. Furthermore, we plan to evaluate the performance of the algorithms on a wide variety of hardware and software configurations in collaboration with various institutions, including a RAID of SSD disks. Preliminary tests on a Lustre file system have already been performed and revealed again a highly different behavior of the algorithms. The long term goal of the project is to develop a flexible module for collective I/O operations that can easily adjust to various hardware and software configurations and choose the right algorithm dynamically, using dynamic runtime adaption techniques.

Acknowledgments. This research was funded in part by a gift from the Silicon Valley Community Foundation, on behalf of the Cisco Collaborative Research Initiative of Cisco Systems.

References

1. May, J.: Parallel I/O for High Performance Computing. Morgan Kaufmann, San Francisco (2001)
2. Message Passing Interface Forum: MPI-2: Extensions to the Message Passing Interface (July 1997), http://www.mpi-forum.org
3. Thakur, R., Gropp, W., Lusk, E.: On Implementing MPI-IO Portably and with High Performance. In: Proceedings of the sixth workshop on I/O in parallel and distributed systems, pp. 23–32 (1999)
4. Thakur, R., Gropp, W., Lusk, E.: Data Sieving and Collective I/O in ROMIO. In: FRONTIERS 1999: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation, p. 182. IEEE Computer Society, Los Alamitos (1999)
5. Prost, J.P., Treumann, R., Hedges, R., Jia, B., Koniges, A.: MPI-IO/GPFS, an Optimized Implementation of MPI-IO on top of GPFS. In: Proceedings of Supercomputing (November 2001)
6. Isaila, F., Malpohl, G., Olaru, V., Szeder, G., Tichy, W.: Integrating collective I/O and cooperative caching into the "clusterfile" parallel file system. In: Proceedings of the 18th Annual International Conference on Supercomputing, Sain-Malo, France, pp. 58–67. ACM Press, New York (2004)
7. Ching, A., Choudhary, A., Liao, W.K., Ross, R., Gropp, W.: Efficient structured data access in parallel file systems. In: Proceedings of the IEEE International Conference on Cluster Computing (December 2003)
8. Worringen, J.: Self-adaptive Hints for Collective I/O. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 202–211. Springer, Heidelberg (2006)
9. Ohtani, A., Aono, H., Tomaru, H.: A File Sharing Method for Storage Area Network and Its Performance Verification. NEC Res. Dev. 44(1), 85–90 (2003)
10. Yu, W., Vetter, J., Canon, R.S., Jiang, S.: Exploiting lustre file joining for effective collective io. In: CCGRID 2007: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid, Washington, DC, USA, pp. 267–274. IEEE Computer Society Press, Los Alamitos (2007)
11. Wadleigh, K.R., Crawford, I.L.: Software Optimization for High-Performance Computing. Prentice-Hall, Englewood Cliffs (2000)
12. Simms, S.C., Pike, G.G., Balog, D.: Wide Area Filesystem Performance using Lustre on the TeraGrid. In: Teragrid Conference (2007), http://datacapacitor.researchtechnologies.uits.iu.edu/lustre_wan_tg07.pdf