Block-based Concurrent and Storage-aware Data Streaming for Grid Applications with Lots of Small Files

Wen Zhang[1], Junwei Cao[2,3*], Yisheng Zhong[1,3], Lianchen Liu[1,3], and Cheng Wu[1,3]
[1]*Department of Automation, Tsinghua University, Beijing 100084, China*
[2]*Research Institute of Information Technology, Tsinghua University, Beijing 100084, China*
[3]*Tsinghua National Laboratory for Information Science and Technology, Beijing 100084, China*
**Corresponding email: jcao@tsinghua.edu.cn*

Abstract

Data streaming management and scheduling is required by many grid computing applications, especially when the volume of data to be processed is extremely high while available storage is relatively limited. Big bulk of data from scientific experiments is usually partitioned into lots of small files (LOSF), bringing challenges to data streaming supports. Block-based data transferring is proposed in this work and implemented using GridFTP, where the number of blocks or the size of each block must be carefully scheduled, taking makespan and available storage into account simultaneously. To increase processing efficiency, data streaming and processing have to be performed concurrently; data streaming scheduling must be storage-aware to avoid data overflow. Experimental results show that the optimization method for block-based concurrent and storage-aware data streaming proposed in this work is efficient to deal with the LOSF problem with a relatively good performance in terms of makespan and storage usage.

1. Introduction

Data management is one of the most challenging issues in the Grid implementation. While most existing research on data grids prefer to a *bring-program-to-data* approach, some applications such as astronomical observations, large-scale simulation and sensor networks require *bring-data-to-program* supports, especially when there is a shortage of CPU processing capability located at data sources. In those cases, data streaming supports are required in order to utilize remote CPU resources.

For example, LIGO (Laser Interferometer Gravitational-wave Observatory) [1][2] is generating 1TB scientific data for gravitational wave detection per day but LIGO observatories are lack of enough processing capacity on sites. LIGO is trying to benefit from CPU cycles provided by the Open Science Grid (OSG) [3]. Since most OSG sites are CPU-rich and storage-constrained with no LIGO data available, data streaming supports are required in order to utilize OSG CPU resources.

In a data streaming scenario, data are transferred and processed continuously as if data were always available from local storage. Meanwhile, processed data have to be cleaned up for subsequently coming data. A data streaming environment is implemented with Condor [4] for job allocation and Globus [5] for data transfers using GridFTP [6], as described in our previous work [7].

Known as the “lots of small files” (LOSF) problem, a large number of small files bring new challenges to data streaming, since GridFTP is excellent to transfer large volumes of data, but suffers dramatically if the dataset is composed with many small files. For example, LIGO data are stored in time series binary files, each including 16-256 seconds of data from multiple channels.

A block-based data transferring method is proposed in this work to address the LOSF issue and improve data streaming performance, with considerations of three typical scenarios: faster processing, faster streaming and workflow manners. An example from LIGO burst data analysis is utilized to verify our scheduling algorithms. Experimental results show that block-based concurrent data streaming can achieve good performance in both makespan and storage usage with well-scheduled number of data blocks; concurrent data streaming provides better performance than sequential ones in terms of makespan with the same number of blocks; storage-aware data streaming can relax the storage requirement dramatically without decreasing processing efficiency.

Some existing researches on data streaming management are derived from database management systems, e.g. STREAM [8], Aurora [9], NiagaraCQ [10], and Gigascope [11]. Most of data streaming research in context of grid computing [12][13][14] are application specific or just focus on computational resources. The following software implementation is most relevant to the work described in this paper:

- Chirp [15]. Chirp is a distributed file system for grid computing, providing protocols to efficiently access to many small files. It is focused on how to improve small file performance. Our major concern is to establish a data streaming environment where data streaming and processing are considered simultaneously.
- GridFTP Pipelining [16]. GridFTP Pipelining is developed to cope with the LOSF situation to hide the latency of each transfer request by sending requests while a data transfer is in progress. Also, it focuses on the data transfers without data processing consideration. Our method emphases on concurrent and storage-aware data streaming and processing in an integrated environment.

The rest of this paper is organized as follows: Section 2 provides an overall introduction to data streaming for grid applications; Section 3 analyzes three scenarios of data streaming, including their makespan and usage of storage in different scheduling algorithms; experimental results are included in Section 4 to verify our scheduling algorithms, and Section 5 concludes the paper.

2. Grid Data Streaming

In some data streaming scenarios, data should be pulled by end users rather than pushed to computational resources in the form of streams of tuples, and processing is continuous over these streams as if data were always available from local storage. What's more, data arrival rates must be controlled to match corresponding processing speeds to avoid waste of computational capacity or data overflow. Meanwhile, processed data have to be cleaned up to save space for subsequently coming data.

2.1. Concurrent Data Streaming

To make full use of computational capacity, data streaming should be concurrent with the processing, i.e., data should be streamed to local storage while the processing is going on. For data streaming applications, data transfers and processing take place simultaneously to overlap themselves as much as possible to decrease idle time.

2.2. Storage-aware Data Streaming

It is not always the case that data should be transferred as fast as possible. For data streaming applications, data transfers should be storage-aware instead of spontaneous because available storage is relatively small compared with high volumes of data to be processed. This is to say, data transfers should be controlled to make required data available according to actual data processing speeds and usage of storage. If data arrive too fast and cannot be processed in time, accumulated data may require large amount of storage space over time; on the other hand, if data transfer speed is lower than processing, corresponding jobs may become idle and computational power is not fully utilized.

In some cases where data processing takes longer than data streaming per tuple, data transferring should be intermittent rather than continuous to avoid data overflow. The so-called repertory policy is introduced, where an upper limit and a lower limit are set to control data streaming: when the amount of data in storage is less than the lower limit, data streaming will be invoked until the amount arrives the upper limit and then data streaming will be stopped. Due to data streaming and cleanup, such procedure will repeat. Both limits are set according to the size of blocks in which data are transferred as mentioned below and the available storage.

2.3. Block-based Data Streaming

The target data set in LIGO consists of lots of small files, and unfortunately, the data throughput of small file operations on both network and file systems is many orders of magnitude worse than the bulk transfer speeds available with large files, as known as the problem of LOSF.

In our scenario, small files are transferred in the unit of blocks. One block can be a group of data files, whose size can be adjusted as needed. Accordingly, data are processed by blocks, and it is not necessarily that such blocks have the same size with those in data transfers. The size of each block, or put it in another way, the number of blocks the whole data set is divided into, is the scheduling variable to achieve high processing efficiency, taking the available storage into account simultaneously.

3. Scheduling Algorithms

A well-made scheduling scheme will help to achieve high throughput with small storage in an efficient way. The main variable of scheduling here is the number of blocks, denoted as n, to minimize the required storage and makespan, denoted as T.

GridFTP is applied to transfer data once in a block, and every time the needed time to establish a connection, complete authentication, close the connection and so on is denoted as t_a. Suppose that it will cost a short period of time, t_c, to invoke the processing, and furthermore, both t_a and t_c are assumed to be a constant value.

Given the total amount of data to be processed and *n*, size of each block is set; denote the time period to transfer and process one block as t_t and t_p, respectively. Obviously, different *n* correspond to different t_t and t_p, denoted as $t_t(n)$ and $t_p(n)$. For the sake of analysis, suppose $n*t_t(n)$ and $n*t_p(n)$ are constants respectively, i.e., no matter how many blocks the total data set is divided into, the *pure* time to transfer and process all of data are fixed, eliminating the sum of all t_a and t_c.

Let *A* be the total amount of data to be processed and *S* be available storage. Three scenarios are discussed in the following sections, former two are for concurrent data streaming while the latter one is for sequential data streaming and processing in a workflow manner.

3.1. The Concurrent Manner with Faster Data Processing

Here, one block will be processed before the next one is arriving, so required storage is relatively small and no repertory policy is needed.

The total makespan, T_1, during which all the data are streamed and processed, can be calculated as follows:

$$\begin{aligned} T_1 &= n*(t_a + t_t(n)) + t_c + t_p(n) \\ &= n*t_a + t_p(n) + n*t_t(n) + t_c, \\ &= n*t_a + t_p(n) + C_1 \end{aligned}$$

where C_1 is a constant and has nothing to do with *n*:

$$C_1 = n*t_t(n) + t_c.$$

It can be observed that larger *n* leads to more $n*t_a$ but less $t_p(n)$, and vice versa. So there is an optimal *n* leading to the least T_1, which is also demonstrated in experimental results in Section 4.

As far as storage is concerned, it can be inferred that double block size is enough. So, larger *n* means smaller size of each block, or smaller required storage. But as mentioned above, too large *n* leads to longer makespan. So there must be a tradeoff when setting the number of blocks, taking makespan and storage usage into account simultaneously.

3.2. The Concurrent Manner with Faster Data Streaming

In this circumstance, the processing of one block of data takes longer than corresponding data transfer. During the long run, there will be backlogs of data, which may result in data overflow if data transferring is not controlled according to available storage.

In this case, the time span to process all the blocks, T2, can be calculated as follows:

$$\begin{aligned} T_2 &= t_a + t_t(n) + n*(t_c + t_p(n)) \\ &= n*t_c + t_t(n) + t_a + n*t_p(n), \\ &= n*t_c + t_t(n) + C_2 \end{aligned}$$

where

$$C_2 = n*t_p(n) + t_a,$$

and C_2 is another constant irrespective of *n*. The larger *n* makes larger $n*t_c$ but less $t_t(n)$, so there may also be another optimal *n* which leads to the minimal T_2.

In this case, data streaming must be storage-aware, as mention in subsection 2.2. In general, the lower limit, n_L, can be set to the double of a block, which is enough for concurrent data streaming and processing, and the upper limit, n_P, is determined by the available storage. Higher values of lower and upper limits would not help decrease the total makespan, but may improve robustness of processing, especially when network failures occur and no more data can be streamed. In that case, data stored in local storage in advance can be processed. The processing survives at a cost of more local storage usage.

3.3. The Workflow Manner

In this case, data streaming and processing work in a sequential way, i.e., one data block is streamed to local storage and processed, after which another block is transferred and processed until all the blocks are processed.

The total time span, T_3, can be calculated as follows:

$$\begin{aligned} T_3 &= n*(t_a + t_t(n) + t_c + t_p(n)) \\ &= n*(t_a + t_c) + n*(t_t(n) + t_p(n)), \\ &= n*(t_a + t_c) + C_3 \end{aligned}$$

where

$$C_3 = n*(t_t(n) + t_p(n)),$$

and C_3 is also a constant.

It can be observed that larger *n* leads to longer makespan since more authentications and system calls are involved. On the other hand the storage requirement is less as *n* increases, since each block is smaller in size. There is also a tradeoff between makespan and usage of storage.

4. Experimental Results

To verify the scheduling issues described in Section 3, experiments are carried out using our grid data streaming environment with an application from LIGO gravitational wave data analysis.

4.1. Experiment Design

In the work described in this paper, the LIGO application is a data analysis program that reads in two data streams from two LIGO observatories and calculates the correlation coefficients that can be used to characterize similarity of two curves. If two identified signals from two observatories (one in the Washington State and the other in the Louisiana State) occur simultaneously with similar curves, it would be very likely that a gravitational wave burst is detected. Note that this is only a simplified case study since actual LIGO burst analysis pipeline is much more sophisticated because many pre- and post- signal processing steps are required.

LIGO data streams are composed with small data files, each containing observational data acquired in 16 seconds. The data files used in the experiment is the LIGO level 3 data with reduced sizes that only include data from the gravitational wave channel. Each data file is about mega bytes in size. In the experiment, 1188 pairs of data files will be transferred and processed. A local grid is established where the nominal bandwidth among each computer is 10 Mbps and shared. Three computers are used, two as data processors and one as data storage. All of them are equipped with Intel Pentium IV processor, 1GB of memory and 160GB of disk. But the two data processors have different workload, which results in different data processing speeds and scenarios in Sections 3.1 and 3.2 are produced. Data transfer is implemented using GridFTP once a block.

4.2. Makespans

In both scenarios 3.1 and 3.2, makespans show similar characteristics varying with the number of blocks: when *n* is less than a certain number, makespan decreases dramatically with increasing *n*, while the makespan increases slowly with increasing *n* exceeding the certain number. The result justifies our allege in Sections 3.1 and 3.2, and we are confirmed that there must be an optimal *n* to achieve the minimum makespan, although we are not aware of the precise values of t_a, t_c, $t_t(n)$ and $t_p(n)$, and sometimes we need to find the optimal or at least satisfactory number of blocks or in other words, the size of each block, empirically. Figures 1 and 2 illustrate the makespans against different number of blocks in the scenario 3.1 and scenario 3.2 respectively.

From Figure 1, it can be observed that (1) the makespan drops the most fiercely at the front part of the curve, i.e., when the number of blocks increases from 1 to 2, or 2 to 3 and so on because $t_p(n)$ decreases dramatically; (2) in the medium part of the curve, the makespan just vary gently because the decrease of $t_p(n)$ and increase of $n*t_a$ *offset* each other, and the optimal numbers of blocks can be located here; (3) in the tail of the curve, the makespan increases slowly as *n* increases because $n*t_a$ prevails. So the middle part of this curve represents optimized numbers of blocks in both terms of makespan and storage usage. In particular, the optimal number of blocks in this case is about 40, and in other words, each block includes 30 data files.

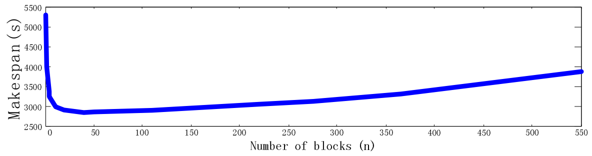

Figure 1. Experimental Results of Scenario 3.1 – Makespan against the Number of Blocks *n*

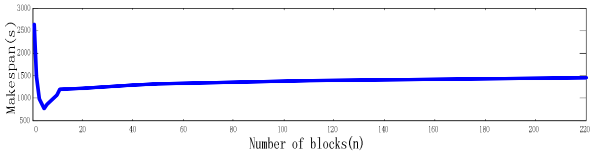

Figure 2. Experimental Results of Scenario 3.2 – Makespan against the Number of Blocks *n*

In the scenario 3.3, makespans show different characteristics from those in 3.1 and 3.2. Makespans increase slowly with increasing number of blocks, due to the more overhead of authentications and system calls for processing programs, as shown in Figure 3. But on the other hand, with increasing number of blocks, the size of each block decreases, or more exactly, the required storage for processing decreases. Just as the two sides of a coin, makespan and storage usage must be scheduled as a whole. It can be observed that makespans in scenarios of concurrent data streaming and processing are much less than those performed in a workflow manner, which verifies the advantage of the concurrent method.

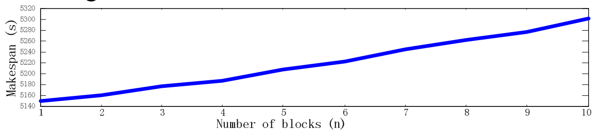

Figure 3. Experimental Results of Scenario 3.3 – Makespan against the Number of Blocks *n*

4.3. Storage Usage

In scenarios 3.1 and 3.3, required storage varies under the size of two data blocks and one data block, respectively, so the more blocks the whole data set is divided into, the less storage is required. Figures 4 and 5 demonstrate the storage usage of scenarios 3.1 and 3.3, respectively, in terms of the number of data files in storage.

In this experiment, the number of blocks is about 20 and each block includes 55 data files. As shown in Figure 4, the total number of data files in local storage vibrates between 40 and 110 periodically. This is an overall effectiveness of concurrent data streaming, processing and cleanups. Since the processing speed is a bit faster than data streaming, total number of local data files cannot exceed two blocks (110) but can be less than one block (55). The whole process keeps the local storage usage within a reasonable scope and scales well over time.

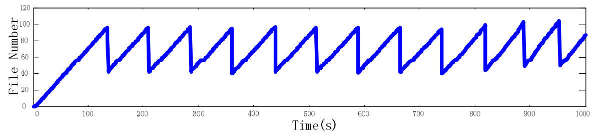

Figure 4. Experimental Results of Scenario 3.1 – Storage Usage over Time (*n*=20)

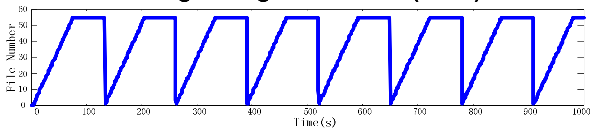

Figure 5. Experimental Results of Scenario 3.3 – Storage Usage over Time (*n*=20)

As shown in Figure 5, the process survives with local storage usage of only one block (55) of data files under the workflow manner. In this scenario, data streaming, processing and cleanups are performed sequentially. While less storage is required, the period is much longer (almost doubled) than that of scenario 3.1. It is obvious that the whole data throughput decreases and system resources are not fully utilized.

As we can notice, the number of data files in storage is not continuous. There are some cliffs in the curves, which is due to the fact that we process and delete data files in the unit of blocks, not one file by another. And it also holds true in scenario 3.2, where the required storage increases persistently due to the backlogs of data and data overflow may occur, so some policy must be adopted to avoid it. Repertory policy is taken and its effect on storage usage and makespan is discussed in the next section.

4.4. Repertory Policies

Experimental results of scenario 3.2 are illustrated in Figure 6 that compares storage usage over time with or without repertory policies. It can be observed that, if no policy is applied, data files are streamed continuously all the way to local storage. Since data processing is slower than data streaming, data files are accumulated in local storage up to 1000 files and consequently the data will be processed in blocks one by one. The processing takes longer, so there is a large amount of backlogs of data, which leads to high volume of storage usage.

The performance is improved if repertory policy is adopted, where the upper and lower limits are set to 5 and 1 blocks, respectively. In this scenario, data streaming is intermittent rather than continuous, which helps confine the storage usage to a certain scope. Also this guarantees that there are always data blocks in storage to keep data processing going on constantly. In this way large amount of data can be processed with just reasonable storage and high processing efficiency.

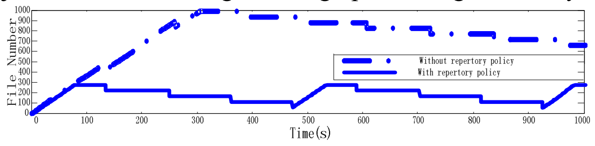

Figure 6. Experimental Results of Scenario 3.2 – Comparison of Storage Usage over Time with/without the Repertory Policy (*n*=20)

Not surprisingly, introduction of repertory policy dose not prolong the makespan, for as can be noticed, the same numbers of processing periods are included in curves in Figure 6, and makespans of the two cases are almost the same. The makespan is not sensitive to the upper and lower limits of repertory policy as long as there are data files available in local storage. While higher limits do not result in higher processing performance, it can help to increase the robustness of data processing.

To achieve high processing performance with reasonable storage usage, it is important to divide the whole data set consisting of lots of small files into appropriate blocks and carry out concurrent data streaming, processing and cleanups. There is a tradeoff between makespan and storage usage, and experimental results show that medium sizes of blocks lead to a relatively better compromise. To be storage-aware, the repertory policy must be introduced, when data streaming is faster than processing, to enable the processing of high volumes of data with relatively small available storage and to achieve similar makespan. It is also observed that the concurrent and storage aware data streaming for lots of small files in blocks prevails over the workflow manner by much in terms of data processing performance.

5. Conclusions

This work is focused on addressing the LOSF problem for grid data streaming applications. A block based method is applied together with consideration of concurrency and storage awareness. The number of blocks, or the size of each block, must be scheduled carefully to achieve high processing performance with reasonable storage usage. Compared with sequential data streaming, processing and cleanups usually performed in a workflow manner, the concurrent approach proposed in this work achieves higher performance in terms of data throughput, since data streaming and processing are overlapped as much as possible to get better utilization of grid resources. Storage-aware data streaming, implemented with the introduction of repertory policy, is used to confine storage usage to a limited scope without delaying processing, so that large amount of data can be processed with reasonable storage requirements.

A simplified application from LIGO gravitational wave data analysis is used in our work to verify our approach, which actually motivates all the work described in this paper. Ongoing work include the application of our data streaming environment for gravitational wave data analysis during the sixth LIGO science run to be started in 2009, in close collaboration with the MIT/LIGO laboratory.

We believe integrated and cooperative management of various grid resources is becoming essential for future grid development, since performance of grid applications are always associated with all aspects of these grid resources. A unified mechanism is critical for integrated management and scheduling of various grid resources.

Acknowledgement

This work is supported by National Science Foundation of China (grant No. 60803017), Ministry of Science and Technology of China under the national 863 high-tech R&D program (grants No. 2006AA10Z237, No. 2007AA01Z179 and No. 2008AA01Z118), Ministry of Education of China under the program for New Century Excellent Talents in University and the Scientific Research Foundation for the Returned Overseas Chinese Scholars, and the FIT foundation of Tsinghua University.

Junwei Cao would like to express his gratitude to Professor Erik Katsavounidis of LIGO (Laser Interferometer Gravitational-wave Observatory) Laboratory at Massachusetts Institute of Technology for their long-term collaboration supports.

References

[1]. D. A. Brown, P. R. Brady, A. Dietz, J. Cao, B. Johnson, and J. McNabb, "A Case Study on the Use of Workflow Technologies for Scientific Analysis: Gravitational Wave Data Analysis", in I. J. Taylor, D. Gannon, E. Deelman, and M. S. Shields (Eds.), *Workflows for eScience: Scientific Workflows for Grids*, Springer Verlag, pp. 39-59, 2007.

[2]. E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, et. al., "GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists", in *Proc. 11th IEEE Int. Symp. on High Performance Distributed Computing*, pp. 225-234, 2002.

[3]. R. Pordes for the Open Science Grid Consortium, "The Open Science Grid", in *Proc. Computing in High Energy and Nuclear Physics Conf.*, Interlaken, Switzerland, 2004.

[4]. M. Litzkow, M. Livny, and Matt Mutka, "Condor – A Hunter of Idle Workstations", in *Proc. of 8th Int. Conf. on Distributed Computing Systems*, pp. 104-111, 1988.

[5]. I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit", *Int. J. Supercomputer Applications*, Vol. 11, No. 2, pp. 115-128, 1997.

[6]. B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, and S. Tuecke, "Data Management and Transfer in High Performance Computational Grid Environments", *Parallel Computing*, Vol. 28, No. 5, pp. 749-771, 2002.

[7]. W. Zhang, J. Cao, Y. Zhong, L. Liu, and C. Wu, "An Integrated Resource Management and Scheduling System for Grid Data Streaming Applications", in *Proc. 9th IEEE/ACM Int. Conf. on Grid Computing*, Tsukuba, Japan, 258-265, 2008.

[8]. A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "STREAM: The Stanford Stream Data Manager", *IEEE Data Eng. Bull.*, Vol. 26, No. 1, pp.19-26, 2003.

[9]. D. Carney, U. Cetinterne, M. Cherniack, et. al., "Monitoring Streams: A New Class of Data Management Applications", in *Proc. Int. Conf. on Very Large Data Bases*, pp. 215-225, 2002.

[10]. J. Chen, D. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases", in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 379-390, 2000.

[11]. C. Cranoe, T. Johnson, V. Shkapenyuk, and O. Spatscheck, "Gigascope: a Stream Database for Network Applications", in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 379-390, 2003.

[12]. B. Agarwalla, N. Ahmed, D. Hilley, and U. Ramachandran, "Streamline: Scheduling Streaming Applications in a Wide Area Environment", *Multimedia Systems*, Vol. 13, No. 1, pp. 69-85, 2007.

[13]. V. Bhat, S. Klasky, S. Atchley, M. Beck, D. McCune, and M. Parashar, "High Performance Threaded Data Streaming for Large Scale Simulations", in *Proc. 5th IEEE/ACM Int. Workshop on Grid Computing*, pp. 243-250, 2004.

[14]. G. Fox, H. Gadgil, S. Pallickara, M. Pierce, R. L. Grossman, et. al., "High Performance Data Streaming in Service Architecture", *Technical Report, Indiana University and University of Illinois at Chicago*, July 2004.

[15]. D. Thain and C. Moretti, "Efficient Access to Many Small Files in a Filesystem for Grid Computing", in *Proc. 8th IEEE/ACM Int. Conf. on Grid Computing*, pp. 243-250, 2007.

[16]. J. Bresnahan, M. Link, R. Kettimuthu, D. Fraser and I. Foster, "GridFTP Pipelining", in *Proc. TeraGrid Conf.*, 2007.