A Flexible GridFTP Client For Implementation of Intelligent Cloud Data Scheduling Services

Esma Yildirim
Department of Computer Engineering
Fatih University
Buyukcekmece, Istanbul, Turkey
esma.yildirim@fatih.edu.tr

ABSTRACT

Current Cloud providers strive to provide novel services and tools for management, analysis, access and scheduling of Big Data that is generated in massive amounts by a large variety of sources. These tools and services have to be flexible and scalable enough to be able to manage data in exa-scale with the help of data centers that can hold thousands of compute and storage nodes interconnected with high speed networks. In this study, we target the performance improvement that might have been achieved from scheduling of big data transfers and provide a flexible client based on a very widely adopted and acclaimed protocol GridFTP. The latest client provided by the Globus Toolkit project does not answer to the needs of highly intelligent optimized data transfer algorithms, which are crucial for Cloud data scheduling services. With this flexible client, developers can implement various kinds of scheduling algorithms as well as apply optimization techniques like pipelining, parallelism and concurrency in much less restricted use cases. The ability to enqueue, dequeue, combine, sort and divide data transfers into groups helped apply these techniques easily resulting in performance improvements in terms of throughput in Amazon's Elastic Compute Cloud (EC2) environment. The client was used to implement two different algorithms, which were able to exploit its abilities and provided performance improvements in both cases.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]

General Terms

Algorithms, Performance, Experimentation

Keywords

Big data; data scheduling, GridFTP, client, throughput optimization

1. INTRODUCTION

Data scheduling services have become an important component for the Data Cloud which aims to specialize in management, analysis, access and scheduling of Big Data which can be large, diverse, complex, longitudinal and distributed datasets generated from various sources such as scientific instruments, sensors, internet transactions, email, video, click streams and all other digital sources available today or in the future [1]. The application level protocols, designed to provide high performance, especially in high-speed networks, are an integral part of data scheduling services. GridFTP [3] is one of the most advanced and highly adopted protocols that is used in high-speed networks to move around Big Data and also used in current Cloud data scheduling services such as GlobusOnline[5] and Stork[4].

Transferring large datasets that consist of many small files causes many bottlenecks that are present (e.g. transport protocol not reaching full network utilization due to short-duration transfers, connection start-up/tear down overhead) in addition to those occurring in transferring large files (e.g.protocol inefficiency, end-system limitations). With GridFTP, several data optimization techniques are given to the users and developers to overcome these bottlenecks and increase the total throughput of their data transfers. Three of the most powerful functions provided by GridFTP are pipelining, parallelism and concurrency. With pipelining, a single control and data channel is used for all the files and they are sent back-to-back without waiting the transfer complete command for the previous transfer. With parallelism, different portions of the same file are sent through different data channels while in concurrency, multiple files are sent through different channels at the same time. Setting the optimal numbers for these parameters is extremely hard. Unfortunately the current GridFTP client *globus-url-copy* does not give the flexibility to change them dynamically for a single data set. However it is very important for data scheduling algorithms to be able to reorder, combine, divide these data sets and apply different parameter settings to find the optimal scheduling.

In this study, we analyze the current needs of scalable data scheduling algorithms and develop a client based on the GridFTP client API. This new client allows the division, reordering, combining, enqueuing and dequeuing of dataset transfers and is able to apply different pipelining, parallelism and concurrency settings to different parts of the dataset. Adaptive, optimization-based scheduling algorithms are easily implemented and their performance improvement on transfer throughput is measured. Two data scheduling algorithms are implemented by using the client API and tested on Amazon EC2. Both algorithms dynamically alter GridFTP parameters and achieve performance improvements that will not be gained otherwise without knowing the right combination of parameter settings.

2. RELATED WORK

At the origin of data scheduling services, lies the *File Transfer Scheduling* problem which dates back to 1980s. One of the earliest works published in 1983 [6], proposes *list scheduling* algorithms in which the transfers are ordered in a list from the

SuperComputing'13, Month 11, 2013, Denver, Colorado, USA.

largest task to the smallest one. This idea has been used in many other algorithms with the addition of extra parameters to *sort* the transfers [7], [8], [9], [10]. Bandwidth of the path and size of the file are used to calculate how long a file transfer task will take. This ordering usually gives a near-optimal solution while some other methods, such as integer programming, are applied to find the *optimal* solution [11]. But they are not feasible since the whole data transfer graph and the duration of the data transfers are needed to be known before hand. Other types of *scalable* algorithms also exist in which, datasets are transferred from multiple replicas residing on different sources [11]. In some cases, these datasets are *divided* so that they can actually be sent over different paths to make use of additional network bandwidth [12]. On other cases, file transfers on the same path are *grouped* and sent together [2].

Adaptive algorithms exist [13], [14], where each file is divided and transferred on multiple streams or multiple file transfers are started at the same time by using concurrency and parallelism notions. The level of concurrency and parallelism is changed based on the current achieved transfer throughput continuously. *Optimization* algorithms, on the other hand, do not continuously change these numbers but, by making use of small number of adaptive samplings, applies mathematical models to find the optimal values of parallelism, pipelining and concurrency [15], [16]. Most of the algorithms presented above are evaluated theoretically or implemented specifically based on the needs of the algorithm by using protocols such as GridFTP.

Modern day *Data Schedulers* such as Globus Online[5] and Stork [4], queue data transfers and apply their own algorithms to the file transfers. Globus Online provides data management capabilities to users as hosted Software-as-a- Service (SaaS) and manages fire-and-forget file transfers for Big Data through thin clients over the Internet. However, it does not provide any optimization capabilities. It sets pipelining, parallelism and concurrency statically for three groups of datasets: average file size less than 50MB, greater than 250MB and in between. Stork, on the other hand, applies the optimization model in [15] for single file transfers and it is also possible to set a static concurrency level to tasks submitted to the scheduler. Each transfer task is started immediately based on this level. This concurrency notion is different from GlobusOnline's where, a group of similar transfers are started at the same time.

We understand that by using different scheduling and optimization techniques, the data transfer throughput performance can be increased dramatically if an optimal or near-optimal solution is found. From the methods described above, a practical client API to implement data scheduling algorithms should allow the dataset transfers to be:

A. enqueued,dequeued;
B. sorted based on a property;
C. divided, combined into chunks;
D. grouped by source-destination paths;
E. done from multiple replicas

3. IMPLEMENTATION DETAILS

One of the most important setbacks in the current GridFTP client is that although it allows you to set parallelism and concurrency statically and sets its own default value for pipelining for a directory transfer which consists of a large number of files, it does not allow you to change it during the course of the transfer (pp=pipelining, p=parallelism, cc = concurrency):

globus-url-copy -pp -p 5 -cc 4 src url dest url

A list of source and destination url pairs could be given as a file list parameter, however the benefits of pipelining can not be exploited as their developers indicate.

globus-url-copy -pp -p 5 -cc 4 -f filelist.txt

The client presented in this study, provides flexible data structures and functions designed to allow the developer to achieve the five goals laid out in the previous section. The main data structure is called *globus_file_t*, which is designed to hold the file information.

```
typedef struct{
        long file_size;
        char * source_path;
        char *dest_path;
        char *file_name;
} globus file t;
```

The *file_size* information can be used in constructing data chunks based on the total size, calculation of throughput and time it takes to complete the transfer based on the bandwidth information. The *source* and *destination paths* are necessary in case of combining and dividing the dataset as well as changing the source based on the available replica locations. *file_name* is used to reconstruct the full paths. This data structure is used with *list_files* function:

```
list_files(
        char *src_url,
        char * dst_url,
        globus_file_t *file_array,
        int * no_of_files
)
```

For a given source and destination urls, the *list_files* function contacts the GridFTP server, gathers the file names at the source url and fills the file array structure which is of type globus_file_t . When this function returns the information about all of the files at the source url is constructed along with the number of files. In this way, it is possible to divide, combine, sort, enqueue, dequeue file transfers and alter their path information.

The *perform_transfer* function is the most important function in this client, which carries out the actual dataset transfer with the optimization parameters.

```
perform_transfer(
        globus_file_t *file_array,
        int no_of_files,
        long chunk_size,
        int pp,
        int p,
        int cc,
        double *thr
)
```

This function no longer operates on source and destination url of the directory like globus-url-copy does, but manipulates the file list array based on the given optimization parameters which set pipelining, parallelism and concurrency

levels. Based on the concurrency level, different pipeline queues are set up and their parallelism levels are set and the transfers in each queue are carried out at the same time by a different handle (which corresponds to a GridFTP client structure that is responsible for a connection) as many as the concurrency level. Now pipelining level can be set for each one of the queues. The chunk size is the total size of the file array and used in calculation of the throughput value (thr). With these basic functions, it is really easy to implement *auxiliary functions* that manipulate the file list structure such as combining, dividing, sorting and altering. In the following section, we make use of these abilities through different algorithms.

4. ALGORITHMS

In this section, two different algorithms are presented which are implemented by using the data structures and functions described in Section 3. The first algorithm is a pure adaptive algorithm, which dynamically changes the concurrency level of the dataset transfer, while the second one is a hybrid of adaptive as well as optimization algorithm based on mathematical models.

4.1 Adaptive Concurrency Algorithm

The adaptive algorithm takes a file list structure returned by list files function and the number of files in a chunk as input. It divides the *file list* into chunks each having the specified *number of files in a chunk* parameter. Starting with concurrency level of 1, performs the first chunk transfer and keeps the returned throughput value. Then it increases the concurrency level for the consequent chunk transfers as long as the measured throughput for each chunk transfer is greater than the previous one. If the throughput decreases, the concurrency level is decreased as well. In this way, changing the concurrency level based on the changes in the throughput allows to adapt network variations and also achieve better performance by using concurrent transfers. Our client serves to the ability to divide dataset transfers into chunks and apply adaptively changing concurrency levels.

Algorithm 1 Adaptive_CC

Require: $list\_of\_files \vee no\_of\_files\_in\_a\_chunk \vee total\_number\_of\_files$
  $cc \leftarrow 1$
  while $There\_are\_more\_chunks$ **do**
    $Perform\_transfer$
    if $curr\_thr < prev\_thr \& cc >= 2$ **then**
      $cc \leftarrow cc/2$
    else
      $cc \leftarrow cc \times 2$
    end if
  end while

4.2 PP-CC-P Optimization Algorithm

The second algorithm targets to find the optimal values of pipelining, concurrency and parallelism, first through minimal samplings, then transfers the rest of the dataset with these values. According to Yildirim et al [16], pipelining is good for file transfers of which sizes are less than the Bandwidth-Delay Product(BDP) of the network. Also parallelism is not good for small files. The optimal pipelining level for the small files is calculated approximately:

$$opt_{pp} \approx BDP / mean\_file\_size + 1 \quad (1)$$

For larger files, a static value of pp = 2 is enough. Considering the dataset at hand, consists of files smaller than BDP, setting different pipelining levels to different groups of chunks with different mean file sizes is a promising solution to increase performance. To divide the dataset into chunks and assign optimal pipelining levels to each chunk, a recursive division algorithm is applied (Algorithm 2). The dataset is divided recursively until the optimal pipelining level for the parent chunk becomes equal to the divided chunk. There are also other stopping conditions. First, a chunk cannot be less than the minimum chunk size provided. This ensures that chunks do not become very small because larger chunks tend to achieve better throughput. Also, not to overwhelm the GridFTP server with back-to-back transfer requests a maximum pipelining level is given and any pipelining value calculated cannot be greater than this value. When the chunks are divided and optimal pipelining values are assigned, an adaptive concurrency scheme is applied to each chunk. During the course of the transfers further division/combination actions might be taken.

Algorithm 2 Optimal_PP

Require: $list\_of\_files \vee start\_index \vee end\_index \vee total\_number\_of\_files \vee min\_chunk\_size \vee parent\_pp \vee max\_pp$
  Calculate $mean\_file\_size$
  Calculate $current\_opt\_pp$
  Calculate $mean\_file\_size\_index$
  if $current\_opt\_pp! = 1 \&$
  $current\_opt\_pp \neq parent\_pp$ &
  $current\_opt\_pp <= max\_pp$ &
  $start\_index < end\_index$&
  $mean\_file\_size\_index > start\_index$&
  $mean\_file\_size\_index < end\_index$&
  $current\_chunk\_size > 2 * min\_chunk\_size$ **then**
    call $optimal\_pp\_dividing\_the\_chunk\_by\_mean\_index$ $(start\_index- > mean\_index)$
    call $optimal\_pp\_dividing\_the\_chunk\_by\_mean\_index$ $(mean\_index + 1- > start\_index)$
  else
    $opt\_pp = parent\_pp$
  end if

The next algorithm finds the optimal concurrency level in addition to optimal pipelining level in an adaptive strategy (Algorithm 3). This algorithm takes file list, number of files, minimum chunk size, BDP and minimum number of chunks as input. The first step sorts the files in ascending order based on their sizes. Then OPTIMAL PP algorithm is applied and the file list is recursively divided with each chunk being assigned their pp values. If the number of created chunks is less than minimum number of chunks parameter, the largest chunk is divided until there are enough available chunks to apply an adaptive concurrency strategy. Each chunk is transferred by setting its optimal pipelining and current concurrency value and the subsequent chunks are transferred by exponentially increasing the concurrency values until the measured throughput starts to drop down. In this case, the previous concurrency level is chosen as optimal and the rest of the

chunks are transferred with this value. However before that, if chunks with same pp values exist, they are combined so that chunk transfers with same pp and cc values will not be conducted separately. This algorithm realizes several of the goals mentioned in Section II such as sorting, dividing and combining datasets.

For datasets with file sizes greater than BDP, a similar algorithm is applied to add parallelism. In this case, optimal pipelining level is set to 2 statically and a new chunk is created with a pre-calculated minimum chunk size. The parallelism level is increased first as in Algorithm 3 and after finding the optimal parallelism value, the concurrency value is increased adaptively until an optimal value is found. The rest of the dataset is transferred by setting these optimal values.

Algorithm 3 Optimal_CC

Require: $list\_of\_files \vee total\_number\_of\_files \vee min\_chunk\_size \vee BDP \vee min\_no\_of\_chunks$
Sort the files
Create chunks by applying OPTIMAL_PP algorithm
while Number of chunks is less than min_chunk_no **do**
Divide largest chunk
end while
$curr\_cc \leftarrow 1$
$prev\_thr \leftarrow 0$
perform transfer for the first chunk
while current throughtput greater than previous throughput **do**
perform transfer for the consequent chunk
$curr\_cc = curr\_cc * 2$
end while
opt cc = prev cc
Combine chunks with same pp values
perform transfer for the rest of the chunks with optimal pp and cc

Table 1. Large Node Specification

vCPU	Storage	Memory	Network Performance
2	8GB	7.5GB	Moderate

5. AMAZON EC2 EXPERIMENTS

The implemented algorithms with the introduced flexible client are tested in Amazon EC2 large nodes. The specification related to this type of nodes is given in Table 1. Each node runs Ubuntu 11.04 and a GridFTP server. The nodes are set up using Globus Provision [17] tool. An artificial delay of 50ms is generated by using Linux *netem* tool. The Bandwidth Delay Product of the network is around 5MB. To measure the effects of the dataset characteristics on the achieved throughput, datasets comprise of large number of small files, which are less than BDP. Two different datasets are generated: 5000 1MB files and 1000 randomly generated files of which sizes range between 1Byte and 10MB. Figure 1 shows the results of application of the algorithms for the first dataset. 3 different chunk sizes are used for Adaptive_CC algorithm (Figure 1.a and b). For all of the cases the throughput increases as the concurrency level is increased. The baseline is the throughput measured if the dataset is sent as a whole by using default GridFTP pipelining and data channel caching. For majority of the chunk transfers, the throughput achieved is higher than the baseline value. For the optimization algorithm (Figure 1.c and d), the optimal pipelining value is selected as 6 and since the average file size is less than BDP value, parallelism is not used. Because optimal pipelining level is the same for all files, the data set is further divided to be able to apply an adaptive concurrency scheme. 5 chunks are generated and the throughput gradually increases above the baseline value in each chunk transfer. It is important to note that the baseline value is the average of instant throughput measurements of the whole dataset transfer. In all of our cases the average of the throughput achieved by individual chunk transfers is higher than the baseline value.

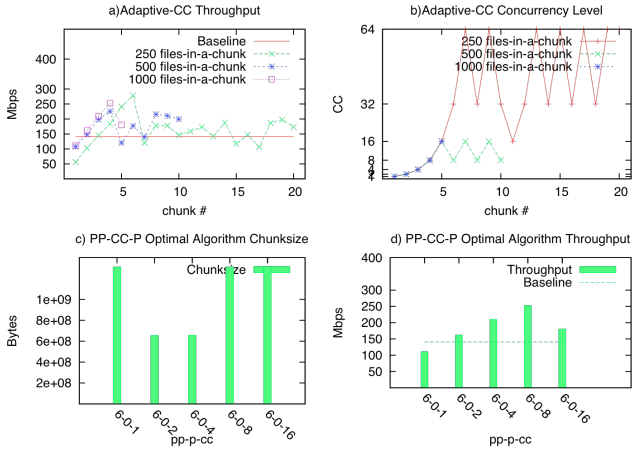

Figure 1. 5000 1MB files, BDP=5MB RTT=100ms-Large Nodes

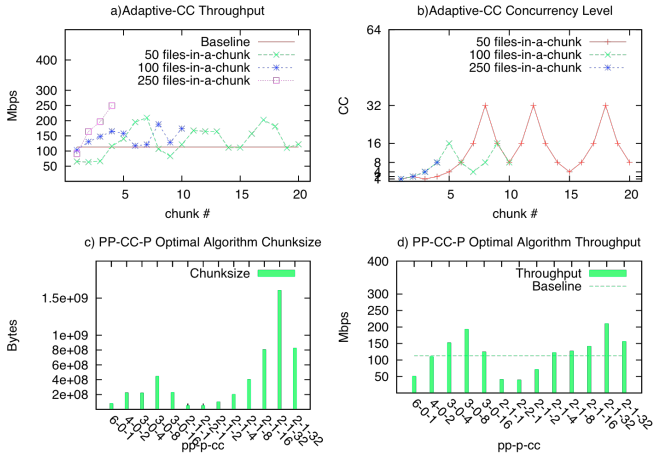

Figure 2. 1000 randomly generated files (Max size = 10MB), BDP=5MB RTT=100ms- Large Nodes

In Figure 2, the results of the application of the algorithms on a dataset transfer which consists of 1000 random-size files is presented. Adaptive-CC algorithm starts with a small throughput and after reaching the peak value, it presents are wavy behavior which results in continuous change in concurrency level (Figure 2.a and b). The majority of the transfer throughput values are above the baseline value. For the optimization algorithm, the dataset is divided into two parts: Files of which sizes are less than and greater than BDP value. For the former set, the OPTIMAL_PP algorithm assigns different pipelining values. For the second set, a static value of pp=2 is given and parallelism and concurrency levels are adaptively changed until an optimum value is found. The throughput achieved gradually increases for both

sets and become higher than baseline performance (Figure 2.c and d).

The highest throughput achieved by the algorithms may change depending on the characteristics of the dataset (e.g. file size and chunk size) but in the end a significant performance improvement is observed in all of the cases. The algorithms are easily implemented due to the flexibility of proposed client functionality. The variety of the algorithms can be extended and would be just as easy to implement as well.

6. CONCLUSION

A flexible GridFTP client is implemented which has the ability to comply with different-natured data scheduling algorithms designed to move around Big Data. The new client can overcome the lacks of the current GridFTP client and allows easy implementation of various kinds of algorithms by allowing optimized values set for pipelining, parallelism and concurrency. The adaptive and optimization algorithms, implemented with the new client; sort, divide and combine datasets easily and improves the achieved throughput. The current structures and functions of the client allow enqueuing and dequeuing of dataset transfers and alteration of source and destination paths. As future work, these auxiliary functions will be implemented in a generic way and to be able to download from multiple replicas at the same, the GridFTP client API will be altered so that it is thread-safe. Reusing the open connections is also another future goal to reduce the overhead of the client.

7. REFERENCES

[1] N. S. Foundation, "Core Techniques and Technologies for Advancing Big Data Science Engineering (BIGDATA," Tech. Report.

[2] S. Tummala and T. Kosar, "Data management challenges in coastal applications," *Journal of Coastal Research,* vol. special Issue No.50, pp. 1188–1193, 2007.

[3] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The globus striped gridftp framework and server," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing.* Washington, DC, USA: IEEE Computer Society, 2005, p. 54.

[4] T. Kosar, M. Balman, E. Yildirim, S. Kulasekaran, and B. Ross, "Stork data scheduler: Mitigating the data bottleneck in e-science," *The Philosophical Transactions of the Royal Society A,* vol. 369, no. 1949, pp. 3254–3267, 2011.

[5] B. Allen, J. Bresnahan, L. Childers, I. Foster, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Link, S. Martin, K. Pickett, and S. Tuecke, "Software as a service for data scientists," *Communications of the ACM,* vol. 55, no. 2, pp. 81–88, 2012.

[6] E. G. C. Jr., M. R. Garey, D. S. Johnson, and A. S. LaPaugh, "Scheduling file transfers in a distributed network." in *PODC,* R. L. Probert, N. A. Lynch, and N. Santoro, Eds. ACM, 1983, pp. 254–266. [Online]. Available: http://dblp.unitrier.de/db/conf/podc/podc83.htmlCoffmanGJL83

[7] A. Giersch, Y. Robert, and F. Vivien, "Scheduling tasks sharing files from distributed repositories." in *Euro-Par,* ser. Lecture Notes in Computer Science, M. Danelutto, M. Vanneschi, and D. Laforenza, Eds., vol. 3149. Springer, 2004, pp. 246–253. [Online]. Available: http://dblp.uni-trier.de/db/conf/europar/europar2004.htmlGierschRV04

[8] G. K. 0002, . atalyrek, T. M. Kur, P. Sadayappan, and J. H. Saltz, "Scheduling file transfers for data- intensive jobs on heterogeneous clusters." in *Euro-Par,* ser. Lecture Notes in Computer Science, A.-M. Kermarrec, L. Boug, and T. Priol, Eds., vol. 4641. Springer, 2007, pp. 214–223. [Online]. Available: http://dblp.uni-trier.de/db/conf/europar/europar2007.html0002CKSS07

[9] M. Hu, W. Guo, and W. Hu, "Dynamic scheduling algorithms for large file transfer on multi-user optical grid network based on efficiency and fairness." in *ICNS,* J. L. Mauri, V. C. Giner, R. Tomas, T. Serra, and O. Dini, Eds. IEEE Computer Society, 2009, pp. 493–498. [Online]. Available: http://dblp.uni-trier.de/db/conf/icns/icns2009.htmlHuGH09

[10] Y. Lin and Q. Wu, "On design of bandwidth scheduling algorithms for multiple data transfers in dedicated networks." in *ANCS,* M. A. Franklin, D. K. Panda, and D. Stiliadis, Eds. ACM, 2008, pp. 151–160. [Online]. Available: http://dblp.uni-trier.de/db/conf/ancs/ancs2008.htmlLinW08

[11] G. Khanna, . V. atalyrek, T. M. Kur, R. Kettimuthu, P. Sadayappan, and J. Saltz, "A dynamic scheduling approach for coordinated wide-area data transfers using gridftp." in *IPDPS.* IEEE, 2008, pp. 1–12. [Online]. Available: http://dblp.uni-trier.de/db/conf/ipps/ipdps2008.htmlKhannaCKKSS08

[12] G. K. 0002, . V. atalyrek, T. M. Kur, R. Kettimuthu, P. Sadayappan, I. T. Foster, and J. H. Saltz, "Using overlays for efficient data transfer over shared wide-area networks." in *SC.* IEEE/ACM, 2008, p. 47. [Online]. Available: http://dblp.uni-trier.de/db/conf/sc/sc2008.htmlKhannaCKKSFS08

[13] W. Liu, B. Tieman, R. Kettimuthu, and I. Foster, "A data transfer framework for large-scale science experiments," in Proc. 3rd International Workshop on Data Intensive Distributed Computing (DIDC '10) in conjunction with 19th International Symposium on High Performance Distributed Computing (HPDC '10), Jun. 2010.

[14] M. Balman and T. Kosar, "Dynamic adaptation of parallelism level in data transfer scheduling." in *CISIS,* L. Barolli, F. Xhafa, and H.-H. Hsu, Eds. IEEE Computer Society, 2009, pp. 872–877. [Online]. Available: http://dblp.uni-trier.de/db/conf/cisis/cisis2009.htmlBalmanK09a

[15] E. Yildirim, D. Yin, and T. Kosar, "Prediction of optimal parallelism level in wide area data transfers," *IEEE Transactions on Parallel and Distributed Systems,* vol. 22, no. 12, pp. 2033–2045, 2011.

[16] E. Yildirim, J. Kim, and T. Kosar, "How gridftp pipelining, parallelism and concurrency work: A guide for optimization of large dataset transfers," in *Proceedings of Network-Aware Data Management Workshop (NDM 2012),* November 2012.

[17] Globus Provision Tool, Available: http://globus.org/provision/