A New Paradigm in Data Intensive Computing: Stork and the Data-Aware Schedulers

Tevfik Kosar*

***Abstract*— The unbounded increase in the computation and data requirements of scientific applications has necessitated the use of widely distributed compute and storage resources to meet the demand. In a widely distributed environment, data is no more locally accessible and has thus to be remotely retrieved and stored. Efficient and reliable access to data sources and archiving destinations in such an environment brings new challenges. Placing data on temporary local storage devices offers many advantages, but such "data placements" also require careful management of storage resources and data movement, i.e. allocating storage space, staging-in of input data, staging-out of generated data, and de-allocation of local storage after the data is safely stored at the destination. Traditional systems closely couple data placement and computation, and consider data placement as a side effect of computation. Data placement is either embedded in the computation and causes the computation to delay, or performed as simple scripts which do not have the privileges of a job. The insufficiency of the traditional systems and existing CPU-oriented schedulers in dealing with the complex data handling problem has yielded a new emerging era: the data-aware schedulers. One of the first examples of such schedulers is the Stork data placement scheduler. In this paper, we will discuss the limitations of the traditional schedulers in handling the challenging data scheduling problem of large scale distributed applications; give our vision for the new paradigm in data-intensive scheduling; and elaborate on our case study: the Stork data placement scheduler.**

***Index Terms*— Scheduling, data-aware, data-intensive, Grid, Stork, staging, storage management, data placement.**

I. Introduction

The computational and data requirements of scientific applications have been increasing drastically over the recent years. In year 2000, the total amount of data to be processed by scientific applications was on the order of a couple hundred terabytes per year. This amount is expected to reach the order of several million terabytes per year by 2010. This exponential increase in the size of scientific data has already outpaced the increase in the computational power and the storage space predicted by the Moore's Law [1] [2].

Figure 1 shows the increase solely in the genomics datasets over the last decade [3] and its comparison with the expected growth according to the Moore's Law. When we include the data from other fields of science such as astronomy, high energy physics, chemistry, earth sciences, and educational technology to this picture, the total amount of data to be processed is hard to estimate and far more than the current computational infrastructure can handle. Table I shows the data requirements of some of the scientific applications in different areas.

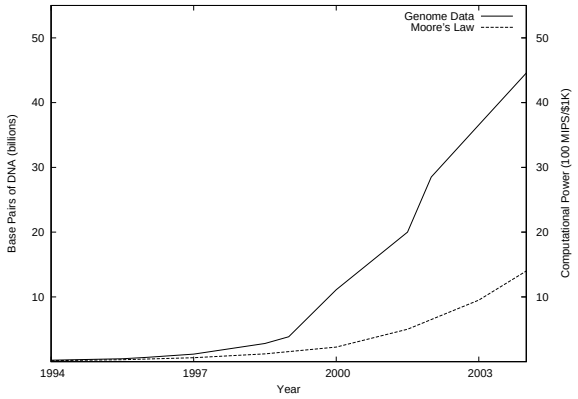

Fig. 1. Growth of Genomics Datasets *[Source: National Center for Biotechnology Information (NCBI)]*

Large amounts of scientific data also require large amounts of computational power in order to be processed in a reasonable time scale. The number of CPUs necessary for ATLAS [9] and CMS [10] applications alone is on the order of hundred thousands. These high computational and data requirements of scientific applications necessitated the use of widely distributed resources owned by collaborating parties to meet the demand. There has been considerable amount of work done on distributed computing [11] [12] [13] [14], batch scheduling [15] [16] [17] [18], and Grid computing [19] [20] [21] [22] to address this problem.

While existing solutions work well for compute-intensive applications, they fail to meet the needs of the data intensive applications which access, create, and move large amounts of data over wide-area networks, due to problems such as:

1. Insufficient storage space when staging-in the input data, generating the output, and staging-out the generated data to a remote storage.
2. Trashing of storage server and subsequent timeout due to too many concurrent read data transfers.
3. Storage server crashes due to too many concurrent write data transfers.
4. Data transfers hanging indefinitely, i.e. loss of acknowledgment during third party transfers.
5. Data corruption during data transfers due to faulty hardware in the data stage and compute hosts.
6. Performance degradation due to unplanned data

*Department of Computer Science & CCT, Louisiana State University, Baton Rouge, LA 70803, USA

Application	Area	Data Volume	Users
VISTA [4]	Astronomy	100 TB/year	100s
LIGO [5]	Astrophysics	250 TB/year	100s
WCER EVP [6]	Educational Technology	500 TB/year	100s
LSST [7]	Astronomy	1000 TB/year	100s
BLAST [8]	Bioinformatics	1000 TB/year	1000s
ATLAS [9]	High Energy Physics	5000 TB/year	1000s
CMS [10]	High Energy Physics	5000 TB/year	1000s

TABLE I
DATA REQUIREMENTS OF SCIENTIFIC APPLICATIONS

transfers.

7. Intermittent wide-area network outages.

All traditional batch schedulers are CPU-centric and are not be able to handle the complications raised due to the need to access to large amounts of remote data during computation. In the next section, we will give a short history of the evolution of the CPU- and data-centric schedulers.

II. BACKGROUND

I/O has been very important throughout the history of computing, and special attention given to it to make it more reliable and efficient both in hardware and software.

In the old days, the CPU was responsible for carrying out all data transfers between I/O devices and the main memory at the hardware level. The overhead of initiating, monitoring and actually moving all data between the device and the main memory was too high to permit efficient utilization of CPU. To alleviate this problem, additional hardware was provided in each device controller to permit data to be directly transferred between the device and main memory, which led to the concepts of DMA (Direct Memory Access) and I/O processors (channels). All of the I/O related tasks are delegated to the specialized I/O processors, which were responsible for managing several I/O devices at the same time and supervising the data transfers between each of these devices and main memory [23].

On the operating systems level, initially the users had to write all of the code necessary to access complicated I/O devices. Later, low level I/O coding needed to implement basic functions was consolidated to an I/O Control System (IOCS). This greatly simplified users' jobs and sped up the coding process [24]. Afterwards, an I/O scheduler was developed on top of IOCS that was responsible for execution of the policy algorithms to allocate channel (I/O processor), control unit and device access patterns in order to serve I/O requests from jobs efficiently [25].

When we consider scheduling of I/O requests at the distributed systems level, we do not see the same recognition given them. They are not considered as tasks that need to be scheduled and monitored independently. I/O and computation is closely coupled at this level, and I/O is simply regarded as a side effect of computation. In many cases, I/O is handled manually or using simple scripts which require baby-sitting throughout the process. There are even cases where the data is dumped to tapes and sent to the destination via postal services [26].

In [27], we have introduced the concept that "data placement" activities in a distributed computing environment must be first class citizens just like the computational jobs. In that work, we have presented a framework in which data placement activities are considered as full-edged jobs which can be queued, scheduled, monitored, and even check-pointed. We have introduced the first batch scheduler specialized in data placement and data movement: Stork. This scheduler implements techniques specific to queuing, scheduling, and optimization of data placement jobs, and provides a level of abstraction between the user applications and the underlying data transfer and storage resources.

Later, Bent et. al. introduced a new distributed file system, the Batch-Aware Distributed File System (BADFS) [28], and a modified data-driven batch scheduling system [29]. Their goal was to achieve data-driven batch scheduling by exporting explicit control of storage decisions from the distributed file system to the batch scheduler. Using some simple data-driven scheduling techniques, they have demonstrated that the new data-driven system can achieve orders of magnitude throughput improvements both over current distributed file systems such as AFS as well as over traditional CPU-centric batch scheduling techniques which are using remote I/O.

We believe that these were only the initial steps taken towards a new paradigm in data-intensive computing: the data-aware batch schedulers. This trend will continue since the batch schedulers are bound to take the data into consideration when making scheduling decisions in order to handle the data-intensive tasks correctly and efficiently.

III. A MOTIVATING EXAMPLE: BLAST

In this section, we provide a motivating example illustrating how data placement is handled by scientific applications using the traditional systems. This example is a well known bioinformatics application: Blast [8]. Blast aims to decode genetic information and map genomes of different species including humankind. Blast uses comparative analysis techniques while doing this and searches for sequence similarities in protein and DNA databases by comparing unknown genetic sequences (on the order of billions) to the known ones.

Figure 2 shows the Blast process, the inputs it takes and the output file it generates, as observed by the traditional CPU-centric batch schedulers [30]. This is a very simplistic view of

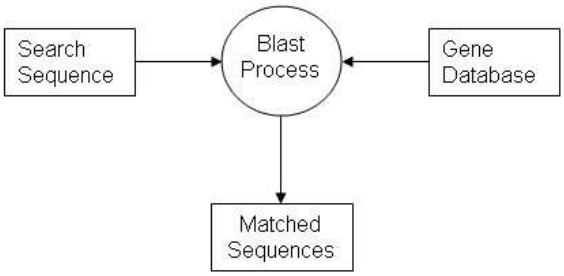

Fig. 2. Blast Process (*blastp*)

what is actually happening in a real distributed environment when we think about the end-to-end process. The diagram in Figure 2 does not capture how the input data is actually transferred to the execution site, and how the output data is utilized.

If we consider the end-to-end process, we see how actually the data is moved and processed by the Blast application in a distributed environment, shown in Figure 3. Data movement definitely complicates the end-to-end process. In Figure 3a, we see the script used to fetch all the files required by the Blast application, such as the executables, the gene database, and the input sequences. After all of the files are transferred to the execution site and preprocessed, a Directed Acyclic Graph (DAG) is generated, where jobs are represented as nodes and dependencies between jobs are represented as directed arcs. This DAG, shown in Figure 3b, can have up to *n* Blast processes (*blastp*) in it, all of which can be executed concurrently. After completion of each *blastp* process *i*, a parser process *i'* is executed which extracts the useful information from the output files generated by *blastp* and reformats them. If these two processes get executed on different nodes, the transfer of the output file from the first node to the second one is performed by the file transfer mechanism of the used batch scheduling system. When all of the processes in the DAG complete successfully, another script is executed, which is shown in Figure 3c. This script double-checks the generated output files for consistency and then transfers them back to the home storage.

During Blast end-to-end processing, most of the data movements are handled using some scripts before and after the execution of the actual compute jobs. The remaining intermediate data movements between jobs are performed by the file transfer mechanism of the batch scheduling system used for computation. The compute jobs are scheduled by the batch scheduler for execution. On the other side, the data transfer scripts are run as "fork" jobs, generally at the head node, which do not get scheduled at all. There are no concurrency controls on the data transfers and no optimizations. Too many concurrent transfers can overload network links, trash storage servers, or even crash some of the source or destination nodes. They can fill in all of the disk space available before even a single transfer gets completed, since no space allocation is performed, which in turn can cause the failure of all of the jobs. More than one computational job on the same execution host or on the same shared file system can ask for the same files, and there can be many redundant transfers.

A message sent from one of the Blast site administrators to the rest of the Blast users illustrates this very well:

> *"... I see a lot of gridftp processes on ouhep0, and about 10 GB of new stuff in the $DATA area. That's taxing our NFS servers quite a bit (load averages of up to 40), and you're about to fill up $DATA, since that's only a 60 GB disk. Keep in mind that this is a small test cluster with many really old machines. So please don't submit any jobs which need a lot of resources, since you're likely to crash those old nodes, which has happened before..."*

IV. Data-aware Scheduling (Case Study: Stork)

We have presented the fundamental features of our data placement scheduler Stork in [27] and [31]. In this paper, we will focus on the general data-aware scheduling techniques that Stork employs to provide efficient scheduling of data-intensive applications.

In order to be able to perform data-aware scheduling, first the scheduler needs to understand the characteristics and semantics of data placement tasks. For this purpose, we differentiate between different types of data placement jobs.

A. Data Placement Job Types

We categorize data placement jobs into seven types. These are:

transfer: This job type is for transferring a complete or partial file from one physical location to another one. This can include a get or put operation or a third party transfer.

allocate: This job type is used for allocating storage space at the destination site, allocating network bandwidth, or establishing a light-path on the route from source to destination. Basically, it deals with all necessary resource allocations prerequired for the placement of the data.

release: This job type is used for releasing the corresponding resource which is allocated before.

remove: This job is used for physically removing a file from a remote or local storage server, tape or disk.

locate: Given a logical file name, this job consults a meta data catalog service such as MCAT [32] or RLS [33] and returns the physical location of the file.

register: This type is used to register the file name to a meta data catalog service.

unregister: This job unregisters a file from a meta data catalog service.

The reason that we categorize the data placement jobs into different types is that all of these types can have different priorities and different optimization mechanisms.

B. Efficient Resource Utilization

The data-aware batch scheduler (Stork) can control the number of concurrent requests coming to any storage system it has access to, and makes sure that neither that storage system nor the network link to that storage system get overloaded.

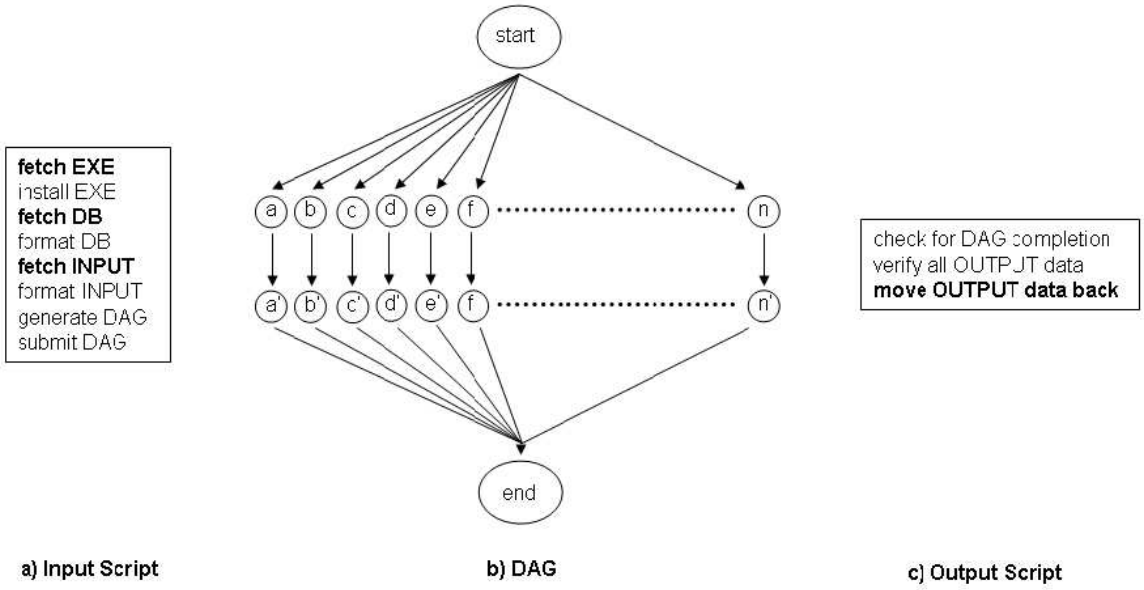

Fig. 3. End-to-end Processing Performed by Blast

It can also perform space allocation and deallocations to make sure that the required storage space is available on the corresponding storage system. The space allocations are supported by Stork as long as the corresponding storage systems have support for it.

Figure 4 shows the effect of increased parallelism and concurrency levels on the transfer rate. With the level of parallelism, we refer to the number of parallel streams used during the transfer of a single file; and with the level of concurrency, we refer to the number of files being transferred concurrently.

When the level parallelism and concurrency increases, the transfer rate incurred in the wide area transfers increases as expected. But for the local area transfers, the case is different. We observe that increased parallelism and concurrency levels help with increasing the transfer rate in local area transfers up to a certain point, but after that, they have a negative impact on the transfer rate. The transfer rate comes to a threshold, and after this point the overhead of using multiple streams and issuing multiple transfers starts causing a decrease in the transfer rate.

These observations show us that increasing parallelism and concurrency levels do not always increase the transfer rate. The effect on the wide and local area can be different. Increased concurrency has a more positive impact on the transfer rate compared with increased parallelism.

Figure 5 shows the effect of increased parallelism and concurrency levels on the CPU utilization. While the number of parallel streams and the concurrency level increases, the CPU utilization at the client side increases as expected. On the server side, same thing happens as the level of concurrency increases. But, we observe a completely opposite effect on the server side as the level of parallelism increases. With the increased parallelism level, the server CPU utilization starts dropping and keeps this behavior as long as the parallelism level is increased.

The most interesting observation here is that concurrency and parallelism have completely opposite impacts on CPU utilization at the server side. As stated by the developers of GridFTP, a reason for this can be the amortization of the select() system call overhead. The more parallel streams that are monitored in a select() call, the more likely it will return with ready le descriptors before GridFTP needs to do another select() call. Also, as the more le descriptors that are ready, the more time is spent actually reading or writing data before the next select() call.

These results show that some of the assumptions we take for granted may not always hold. We need a more complicated mechanism to decide the correct concurrency or parallelism level in order to achieve high transfer rate and low CPU utilization at the same time.

C. Job Scheduling Techniques

We have applied some of the traditional job scheduling techniques common in computational job scheduling to the scheduling of data placement jobs:

First Come First Served (FCFS) Scheduling: Regardless of the type of the data placement job and other criteria, the job that has entered into the queue of the data placement scheduler first is executed first. This technique, being the simplest one, does not perform any optimizations at all.

Shortest Job First (SJF) Scheduling: The data placement job which is expected to take least amount of time to complete will be executed first. All data placement jobs except the transfer jobs have job completion time in the order of seconds, or minutes in the worst case. On the other hand, the execution time for the transfer jobs can vary from couple of seconds to couple of hours even days. Accepting this policy would mean

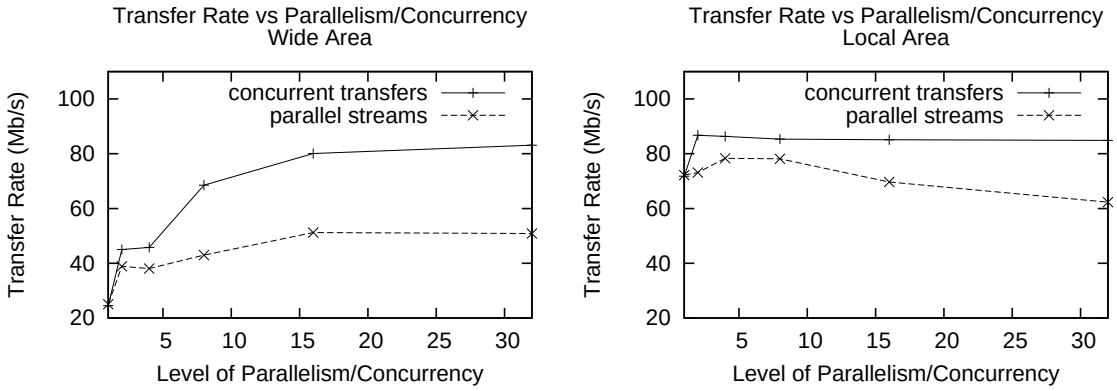

Fig. 4. Controlling the Throughput

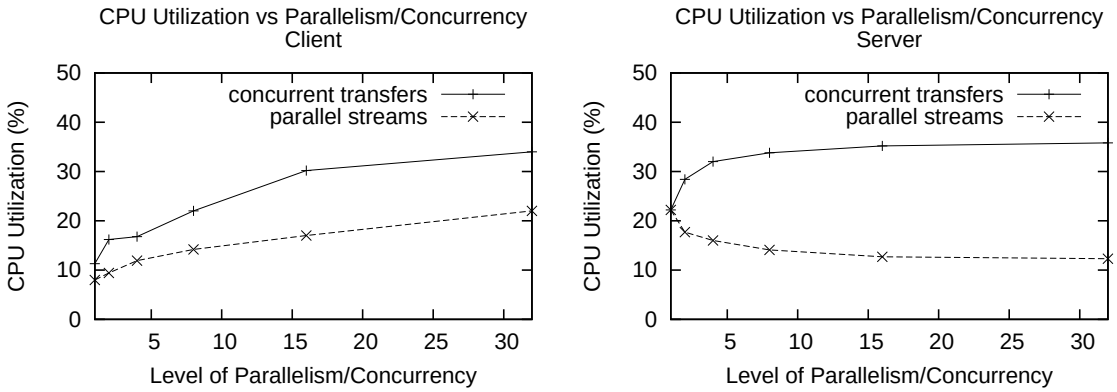

Fig. 5. Controlling the CPU Utilization

non-transfer jobs would be executed always before transfer jobs. This may cause big delays in executing the actual transfer jobs, which defeats the whole purpose of scheduling data placement.

Multilevel Queue Priority Scheduling: In this case, each type of data placement job is sent to separate queues. A priority is assigned to each job queue, and the jobs in the highest priority queue are executed first. To prevent starvation of the low priority jobs, the traditional aging technique is applied. The hardest problem here is determining the priorities of each data placement job type.

Random Scheduling: A data placement job in the queue is randomly picked and executed without considering any criteria.

1) Auxiliary Scheduling of Data Transfer Jobs: The above techniques are applied to all data placement jobs regardless of the type. After this ordering, some job types require additional scheduling for further optimization. One such type is the data transfer jobs. The transfer jobs are the most resource consuming ones. They consume much more storage space, network bandwidth, and CPU cycles than any other data placement job. If not planned well, they can fill up all storage space, trash and even crash servers, or congest all of the network links between the source and the destination.

Storage Space Management. One of the important resources that need to be taken into consideration when making scheduling decisions is the available storage space at the destination. The ideal case would be the destination storage system support space allocations, as in the case of NeST [34], and before submitting a data transfer job, a space allocation job is submitted in the workflow. This way, it is assured that the destination storage system will have sufficient available space for this transfer.

Unfortunately, not all storage systems support space allocations. For such systems, the data placement scheduler needs to make the best effort in order not to over-commit the storage space. This is performed by keeping track of the size of the data transferred to, and removed from each storage system which does not support space allocation. When ordering the transfer requests to that particular storage system, the remaining amount of available space, to the best of the scheduler's knowledge, is taken into consideration. This method does not assure availability of storage space during the transfer of a file, since there can be external effects, such as users which access the same storage system via other interfaces without using the data placement scheduler. In this case, the data placement scheduler at least assures that it does not over-commit the available storage space, and it will manage the space efficiently

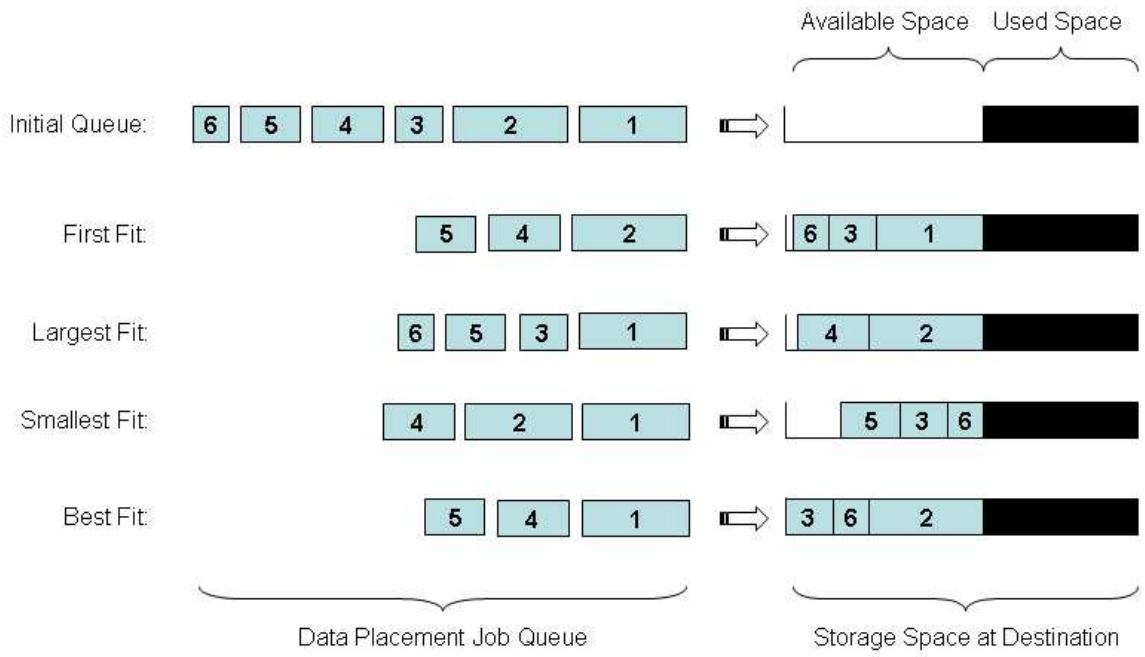

Fig. 6. Storage Space Management: Different Techniques

if there are no external effects.

Figure 6 shows how the scheduler changes the order of the previously scheduled jobs to meet the space requirements at the destination storage system. In this example, four different techniques are used to determine in which order to execute the data transfer request without over-committing the available storage space at the destination: *first fit*, *largest fit*, *smallest fit*, and *best fit*.

First Fit: In this technique, if the next transfer job in the queue is for data which will not fit in the available space, it is skipped for that scheduling cycle and the next available transfer job with data size less than or equal to the available space is executed instead. It is important to point that a complete reordering is not performed according to the space requirements. The initial scheduling order is preserved, but only requests which will not satisfy the storage space requirements are skipped, since they would fail anyway and also would prevent other jobs in the queue from being executed.

Largest Fit and *Smallest Fit:* These techniques reorder all of the transfer requests in the queue and then select and execute the transfer request for the file with the largest, or the smallest, file size. Both techniques have a higher complexity compared with the *first fit* technique, although they do not guarantee better utilization of the remote storage space.

Best Fit: This technique involves a greedy algorithm which searches all possible combinations of the data transfer requests in the queue and finds the combination which utilizes the remote storage space best. Of course, it comes with a cost, which is a very high complexity and long search time. Especially in the cases where there are thousands of requests in the queue, this technique would perform very poorly.

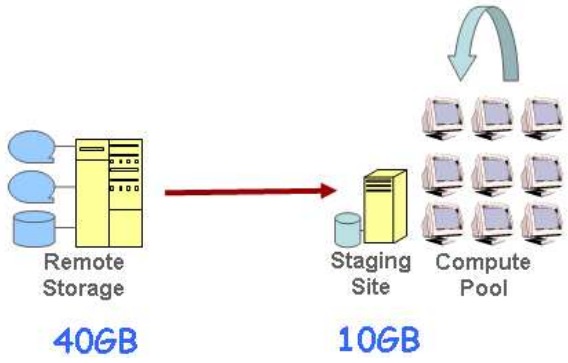

Fig. 7. Storage Space Management: Experiment Setup

Using a simple experiment setting, we will display how the built-in storage management capability of the data placement scheduler can help improving both overall performance and reliability of the system. The setting of this experiment is shown in Figure 7.

In this experiment, we want to process 40 gigabytes of data, which consists of 60 files each between 500 megabytes and 1 gigabytes. First, the files need to be transferred from the remote storage site to the staging site near the compute pool where the processing will be done. Each file will be used as an input to a separate process, which means there will be 60 computational jobs followed by the 60 transfers. The staging site has only 10 gigabytes of storage capacity, which puts a limitation on the number of files that can be transferred and processed at any time.

A traditional scheduler would simply start all of the 60 transfers concurrently since it is not aware of the storage space

limitations at the destination. After a while, each file would have around 150 megabytes transferred to the destination. But suddenly, the storage space at the destination would get filled, and all of the file transfers would fail. This would follow with the failure of all of the computational jobs dependent on these files.

On the other hand, Stork completes all transfers successfully by smartly managing the storage space at the staging site. At any time, no more than the available storage space is committed, and as soon as the processing of a file is completed, it is removed from the staging area to allow transfer of new files. The number of transfer jobs running concurrently at any time and the amount of storage space committed at the staging area during the experiment are shown in Figure 8 on the left side.

In a traditional batch scheduler system, the user could intervene, and manually set some virtual limits to the level of concurrency the scheduler can achieve during these transfers. For example, a safe concurrency limit would be the total amount of storage space available at the staging area divided by the size of the largest file that is in the request queue. This would assure the scheduler does not over-commit remote storage. Any concurrency level higher than this would have the risk of getting out of disk space anytime, and may cause failure of at least some of the jobs. The performance of the traditional scheduler with concurrency level set to 10 manually by the user in the same experiment is shown in Figure 8 on the right side.

Manually setting the concurrency level in a traditional batch scheduling system has three main disadvantages. First, it is not automatic, it requires user intervention and depends on the decision made by the user. Second, the set concurrency is constant and does not fully utilize the available storage unless the sizes of all the files in the request queue are equal. Finally, if the available storage increases or decreases during the transfers, the traditional scheduler cannot re-adjust the concurrency level in order to prevent overcommitment of the decreased storage space or fully utilize the increased storage space.

Storage Server Connection Management. Another important resource which needs to be managed carefully is the number of concurrent connections made to specific storage servers. Storage servers being thrashed or getting crashed due to too many concurrent file transfer connections has been a common problem in data intensive distributed computing.

In our framework, the data storage resources are considered first class citizens just like the computational resources. Similar to computational resources advertising themselves, their attributes and their access policies, the data storage resources advertise themselves, their attributes, and their access policies as well. The advertisement sent by the storage resource includes the number of maximum concurrent connections it wants to take anytime. It can also include a detailed breakdown of how many connections will be accepted from which client, such as "maximum *n* GridFTP connections, and "maximum *m* HTTP connections".

This throttling is in addition to the global throttling performed by the scheduler. The scheduler will not execute more than lets say *x* amount of data placement requests at any time, but it will also not send more than *y* requests to server *a*, and more than *z* requests to server *b*, *y+z* being less than or equal to *x*.

Other Scheduling Optimizations. In some cases, two different jobs request the transfer of the same file to the same destination. Obviously, all of these request except one are redundant and wasting computational and network resources. The data placement scheduler catches such requests in its queue, performs only one of them, but returns success (or failure depending on the return code) to all of such requests. We want to highlight that the redundant jobs are not canceled or simply removed from the queue. They still get honored and the return value of the actual transfer is returned to them. But, no redundant transfers are performed.

In some other cases, different requests are made to transfer different parts of the same file to the same destination. These type of requests are merged into one request, and only one transfer command is issued. But again, all of the requests get honored and the appropriate return value is returned to all of them.

V. Conclusion

The insufficiency of the traditional distributed computing systems and existing cpu-oriented batch schedulers in dealing with the complex data handling problem has yielded a new emerging era: the data-aware batch schedulers. One of the first examples of such schedulers is the Stork data placement scheduler. In this paper, we have discussed the limitations of the traditional schedulers in handling the challenging data scheduling problem of large scale distributed applications; gave our vision for the new paradigm in data intensive scheduling; and elaborated on our case study: the Stork data placement scheduler.

We believe that Stork and its successors are only the initial steps taken towards a new paradigm in data intensive computing: the data-aware batch schedulers. This trend will continue since the batch schedulers are bound to take the data dependencies and data movement into consideration when making scheduling decisions in order to handle the data-intensive tasks correctly and efficiently.

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, 1965.

[2] J. Gray and P. Shenoy, "Rules of thumb in data engineering," in *Proceedings of the IEEE International Conference on Data Engineering*, San Diego, CA, February 2000.

[3] "NCBI: Growth of genbank," http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.

[4] "The Visible and Infrared Survey Telescope for Astronomy," http://www.vista.ac.uk/.

[5] "Laser Interferometer Gravitational Wave Observatory," http://www.ligo.caltech.edu/.

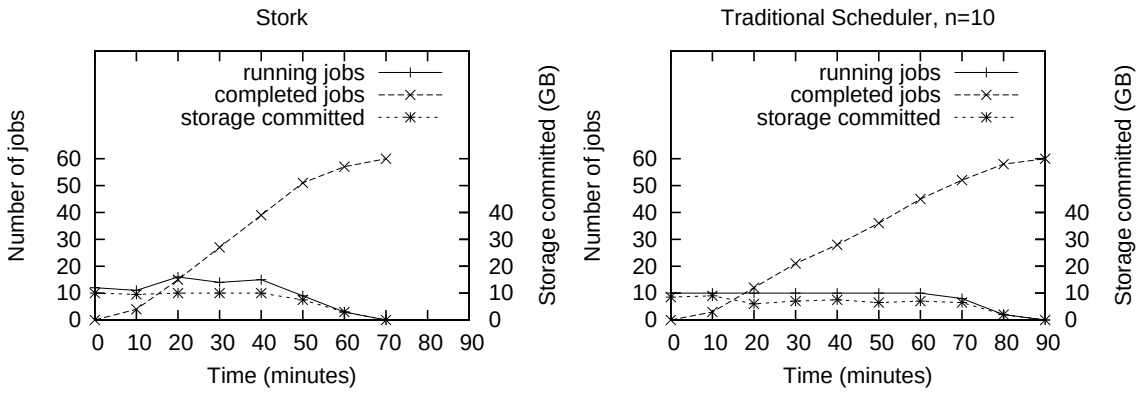

Fig. 8. Storage Space Management: Stork vs Traditional Scheduler

[6] G. Kola, T. Kosar, and M. Livny, "A fully automated fault-tolerant system for distributed video processing and off-site replication," in *Proceedings of the 14th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2004)*, Kinsale, Ireland, June 2004.

[7] "The Large Synoptic Survey Telescope (LSST)," http://www.lsst.org/.

[8] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology*, vol. 3, no. 215, pp. 403–410, October 1990.

[9] "A Toroidal LHC ApparatuS Project (ATLAS)," http://atlas.web.cern.ch/.

[10] "The Compact Muon Solenoid Project (CMS)," http://cmsinfo.cern.ch/.

[11] L. Lamport and N. Lynch, "Distributed computing: Models and methods," *Handbook of Theoretical Computer Science*, pp. 1158–1199, Elsevier Science Publishers, 1990.

[12] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141–154, February 1998.

[13] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A note on distributed computing," Sun Microsystems Laboratories, Tech. Rep. TR-94-29, November 1994.

[14] E. Gabriel, M. Resch, T. Beisel, and R. Keller, "Distributed computing in a heterogeneous computing environment," *Lecture Notes in Computer Science*, vol. 1497, p. 180, January 1998.

[15] R. Henderson and D. Tweten, "Portable Batch System: External reference specification," 1996.

[16] "Using and administering IBM LoadLeveler," IBM Corporation SC23-3989, 1996.

[17] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - A hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, 1988, pp. 104–111.

[18] S. Zhou, "LSF: Load sharing in large-scale heterogeneous distributed systems," in *Proc. of Workshop on Cluster Computing*, 1992.

[19] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the Grid: Enabling scalable virtual organizations," *International Journal of Supercomputing Applications*, 2001.

[20] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger, "Data management in an international DataGrid project," in *First IEEE/ACM Int'l Workshop on Grid Computing*, Bangalore, India, December 2000.

[21] "The Grid2003 production grid," http://www.ivdgl.org/grid2003/.

[22] B. Sagal, "Grid Computing: The European DataGrid Project," in *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, France, October 2000.

[23] L. Bic and A. C. Shaw, "The Organization of Computer Systems," in *The Logical Design of Operating Systems.*, Prentice Hall., 1974.

[24] H. M. Deitel, "I/O Control System," in *An Introduction to Operating Systems.*, Addison-Wesley Longman Publishing Co., Inc., 1990.

[25] S. E. Madnick and J. J. Donovan, "I/O Scheduler," in *Operating Systems.*, McGraw-Hill, Inc., 1974.

[26] W. Feng, "High performance transport protocols," Los Alamos National Laboratory, 2003.

[27] T. Kosar and M. Livny, "Stork: Making data placement a first class citizen in the Grid," in *Proceedings of the 24th Int. Conference on Distributed Computing Systems (ICDCS 2004)*, Tokyo, Japan, March 2004.

[28] J. Bent, D. Thain, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Explicit control in a batch-aware distributed file system," in *Proceedings of the First USENIX/ACM Conference on Networked Systems Design and Implementation*, March 2004.

[29] J. Bent, "Data-driven batch scheduling," Ph.D. dissertation, University of Wisconsin-Madison, 2005.

[30] D. Thain, J. Bent, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Pipeline and batch sharing in grid workloads," in *Proceedings of the Twelfth IEEE Symposium on High Performance Distributed Computing*, June 2003.

[31] T. Kosar and M. Livny, "A framework for reliable and efficient data placement in distributed computing systems," *Journal of Parallel and Distributed Computing*, 2005.

[32] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC Storage Resource Broker," in *Proceedings of CASCON*, Toronto, Canada, 1998.

[33] L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf, "Performance and scalability of a Replica Location Service," in *Proceedings of the International Symposium on High Performance Distributed Computing Conference (HPDC-13)*, Honolulu, Hawaii, June 2004.

[34] "NeST: Network Storage Technology," http://www.cs.wisc.edu/condor/nest/.