Dynamic Protocol Tuning Algorithms for High Performance Data Transfers

Engin Arslan, Brandon Ross, and Tevfik Kosar

Department of Computer Science & Engineering
University at Buffalo (SUNY), Buffalo NY 14260, USA
{enginars,bwross,tkosar}@buffalo.edu

Abstract. Obtaining optimal data transfer performance is of utmost importance to today's data-intensive distributed applications and wide-area data replication services. Doing so necessitates effectively utilizing available network bandwidth and resources, yet in practice transfers seldom reach the levels of utilization they potentially could. Tuning protocol parameters such as pipelining, parallelism, and concurrency can significantly increase utilization and performance, however determining the best settings for these parameters is a difficult problem, as network conditions can vary greatly between sites and over time. In this paper, we present four application-level algorithms for heuristically tuning protocol parameters for data transfers in wide-area networks. Our algorithms dynamically tune the number of parallel data streams per file, the level of control channel pipelining, and the number of concurrent file transfers to fill network pipes. The presented algorithms are implemented as a standalone service as well as being used in interaction with external data scheduling tools such as Stork. The experimental results are very promising, and our algorithms outperform existing solutions in this area.

Keywords: Application-level protocol tuning, throughput optimization, wide-area networks, data-intensive applications, data replication.

1 Introduction

Despite the increasing availability of high-speed wide-area networks and the use of modern data transfer protocols designed for high performance, file transfers in practice often only attain fractions of theoretical maximum throughputs, leaving networks underutilized and users unsatisfied. This fact is often due to a number of confounding factors, such as underutilization of end-system CPU cores, low disk I/O speeds, server implementations not taking advantage of parallel I/O opportunities, traffic at inter-system routing nodes, and unsuitable system-level tuning of networking protocols.

The effects of some of these factors can be mitigated to varying degrees through the use of techniques such as command pipelining, transfer-level parallelism, and concurrent transfers using multiple control channels. The degree to which these techniques are utilized, however, has the potential to negatively impact the performance of the transfer and the network as a whole. Too little use of one technique, and the network might be underutilized; too much, and the network might be overburdened to the detriment of the transfer and other users. Furthermore, the optimal level of usage for each technique varies depending on network and end-system conditions, meaning no combination of parameters is optimal for every scenario.

Dynamic optimization techniques provide a method for determining which combination of parameters is "just right" for a given transfer. This paper proposes optimization techniques that try to maximize transfer throughput by choosing optimal parallelism, concurrency, and pipelining levels through file set analysis and clustering. Our algorithms also re-provision idle control channels dynamically to improve the performance of "slower" file clusters, ensuring that resources are effectively utilized.

In this paper, we present four application-level algorithms for heuristically tuning protocol parameters for data transfers in wide-area networks. Our algorithms can tune the number of parallel data streams per file (for large file optimization), the level of control and data channel pipelining (for small file optimization), and the number of concurrent file transfers to fill network pipes (a technique useful for all types of files) in an efficient manner. The developed algorithms are implemented as a standalone service as well as being used in interaction with external data scheduling tools such as Stork [10,12]. The experimental results are very promising, and our algorithms outperform other existing solutions in this area.

2 Related Work

Liu et al. [14] developed a tool which optimizes multi-file transfers by opening multiple GridFTP control channels. The tool increases the number of concurrent flows up to the point where transfer performance degrades. Their work only focuses on concurrent file transfers, and other transfer parameters are not considered.

Globus Online [1] offers fire-and-forget GridFTP file transfers as a service. The developers mention that they set the pipelining, parallelism, and concurrency parameters to fixed values for three different file sizes (i.e. less than 50MB, larger than 250MB, and in between). However, the tuning Globus Online performs is non-adaptive; it does not change depending on network conditions and transfer performance.

Other approaches aim to improve throughput by opening flows over multiple paths between end-systems [17,8], however there are cases where individual data flows fail to achieve optimal throughput because of end-system bottlenecks. Several others propose solutions that improve utilization of a single path by means of parallel streams [2,6,16,23], pipelining [5,4,3], and concurrent transfers [13,11,14]. Although using parallelism, pipelining, and concurrency may improve throughput in certain cases, an optimization algorithm should also consider system configuration, since end-systems may present factors (e.g., low disk I/O speeds or over-tasked CPUs) which can introduce bottlenecks.

In our previous work [21], we proposed network-aware transfer optimization by automatically detecting bottlenecks and improving throughput by utilizing network and end-system parallelism.

We developed three highly-accurate models [24,22,9] which would require as few as three sampling points to provide accurate predictions for the optimal parallel stream number. These models have proved to be more accurate than existing similar models [7,16] which lack in predicting the parallel stream number that gives the peak throughput. We have developed algorithms to determine the best sampling size and the best sampling points for data transfers by using bandwidth, Round-Trip Time (RTT), or Bandwidth-Delay Product (BDP) [20].

3 Dynamic Protocol Tuning

Different transfer parameters such as pipelining, parallelism, and concurrency play a significant role in affecting achievable transfer throughput. However, setting the optimal levels for these parameters is a challenging problem, and poorly-tuned parameters can either cause underutilization of the network or overburden the network and degrade the performance due to increased packet loss, end-system overhead, and other factors.

Among these parameters, **pipelining** specifically targets the problem of transferring a large numbers of small files. In most control channel-based transfer protocols, an entire transfer must complete and be acknowledged before the next transfer command is sent by the client. This may cause a delay of more than one RTT between individual transfers. With pipelining, multiple transfer commands can be queued up at the server, greatly reducing the delay between transfer completion and the receipt of the next command. **Parallelism** sends different portions of the same file over parallel data streams (typically TCP connections), and can achieve high throughput by aggregating multiple streams and getting an unfair share of the available bandwidth. **Concurrency** refers to sending multiple files simultaneously using parallel control channels, and is especially useful for taking advantage of I/O concurrency in parallel disk systems.

The models developed in our previous work [21,24,22,9] lay the foundations of the dynamic protocol tuning algorithms presented in this paper, where we utilize all three parameters in combination to heuristically determine near-optimal network throughput.

In this paper, we present four dynamic protocol tuning algorithms:

1. The "Single-Chunk (SC)" approach, which divides the set of files into chunks based on file size, and then transfers each chunk with its optimal parameters;
2. the "Multi-Chunk (MC)" approach which likewise creates chunks based on the file size, but, rather than scheduling each chunk separately, it co-schedules and runs small-file chunks and large-file chunks together in order to balance and minimize the effect of poor performance of small file transfers;
3. the "Pro-Active Multi-Chunk (ProMC)" approach, which, instead of allocating channels equally among chunks, considers chunk size and type, and improves the performance if small chunks dominate the dataset; and

4. the "Max Fair MC (FairMC)" approach, which aims to make use of simultaneous chunk transfers but also tries to be fair in terms of network resource usage by limiting the maximum number of simultaneous chunk transfers.

3.1 Single-Chunk (SC) Algorithm

Files with different sizes need different transfer parameters to obtain optimal throughput. For example, pipelining and data channel reuse would mostly improve the performance of small file transfers, whereas per-file parallelism would be beneficial if the files are large. Optimal concurrency levels for different file sizes would be different as well. Instead of using the same parameter combinations for all files in a mixed dataset, we partition the dataset into chunks based on file size and Bandwidth Delay Product (BDP), and use different parameter combinations for each chunk.

As shown in Algorithm 1, we initially partition files into different chunks, then we check if each chunk has a sufficient number of files using the `mergePartitions` subroutine. We merge a chunk with another if it is deemed to be too small to be treated separately. After partitioning files, we calculate the optimal parameter combination for each chunk in `findOptimalParameters`. When calculating the density of a chunk, we take the average file size of the chunk and find its density in a similar way we do in `mergePartitions`.

Pipelining and concurrency are the most effective parameters at overcoming poor network utilization for small file transfers, so it is especially important to choose the best pipelining and concurrency values for such transfers. We set the pipelining values by considering the BDP and average file size of each chunk (lines 23, 27, 31 and 35); set the parallelism values by considering the BDP, average file size, and the TCP buffer size (lines 24, 28, 32, and 36); and set the concurrency values by considering the BDP, average file size, number of files in each chunk, and the maximum concurrency level (lines 25, 29, 33, and 37) in Algorithm 1.

As the average file size of a chunk increases, we decrease the pipelining value since it does not further improve performance, and can even cause performance degradation by poorly utilizing concurrent control channels. The method of selecting parallelism prevents using unnecessarily large parallelism levels for small files and insufficiently small parallelism levels for large files. Concurrency is set to larger values for small files, whereas for large files it is limited to smaller values, as higher concurrency values might cause unfair usage of end-system and network resources.

We tested our algorithms with concurrency levels up to 10. Although higher concurrency levels could possibly further increase throughput, the testing was performed on a shared testbed where it was against policy to open more than 10 file transfer connections at a time. Presumably many other shared network environments implement similar policies. Furthermore, throughput gains were found to experience diminishing returns as the concurrency level was increased. These factors led us to limit the maximum concurrency level our algorithms could reach to some safe fixed value – specifically 10.

Algorithm 1. — Partitioning Dataset and Setting Parameter Values

```
function PARTITIONFILES(allFiles,BDP)
    Chunk Small, Middle, Large, Huge
    while $allFiles.count() > 0$ do
        $File f = allFiles.pop()$
        if $f.size < \frac{BDP}{10}$ then
            Small.add(f)
        else if $f.size < \frac{BDP}{2}$ then
            Middle.add(f)
        else if $f.size < BDP * 20$ then
            Large.add(f)
        else
            Huge.add(f)
        end if
    end while
    allChunks.add(Small,Middle,Large,Huge)
    mergePartitions(allChunks)
return allChunks
end function

function FINDOPTIMALPARAMETERS(chunk,BDP,bufferSize,concurrency)
    Density d = findDensityofPartition(chunk)
    $avgFileSize = findAverage(chunk)$
    if $d == SMALL$ then
        $pipelining = \left\lceil \frac{BDP}{avgFileSize} \right\rceil - 1$
        $parallelism = Min(\left\lceil \frac{BDP}{bufferSize} \right\rceil, \left\lceil \frac{avgFileSize}{bufferSize} \right\rceil) + 1$
        $concurrency = Min(\frac{BDP}{avgFileSize}, chunk.count(), concurrency)$
    else if $d == MIDDLE$ then
        $pipelining = \left\lceil \frac{BDP}{avgFileSize} \right\rceil$
        $parallelism = Min(\left\lceil \frac{BDP}{bufferSize} \right\rceil, \left\lceil \frac{avgFileSize}{bufferSize} \right\rceil) + 1$
        $concurrency = Min(\frac{BDP}{avgFileSize}, chunk.count(), concurrency)$
    else if $d == LARGE$ then
        $pipelining = \left\lceil \frac{BDP}{avgFileSize} \right\rceil + 1$        ▷ This chunk should have pipelining
        $parallelism = Min(\left\lceil \frac{BDP}{bufferSize} \right\rceil, \left\lceil \frac{avgFileSize}{bufferSize} \right\rceil) + 2$
        $concurrency = Min(2, chunk.count(), concurrency)$
    else if $d == HUGE$ then
        $pipelining = \left\lceil \frac{BDP}{avgFileSize} \right\rceil - 1$        ▷ Pipelining will be zero in most cases
        $parallelism = Min(\left\lceil \frac{BDP}{bufferSize} \right\rceil, \left\lceil \frac{avgFileSize}{bufferSize} \right\rceil) + 2$
        $concurrency = Min(2, chunk.count(), concurrency)$
    end ifreturn pipelining,parallelism,concurrency
end function
```

After deciding the best parameter combination for each chunk, the Single-Chunk (SC) algorithm transfers each chunk one-by-one.

3.2 Multi-Chunk (MC) Algorithm

In the Multi-Chunk (MC) method, the focus is mainly on minimizing the effect of small file chunks on the overall throughput. Based on the results obtained from the SC approach, we deduced that even after choosing the best parameter combination for each chunk, the throughput obtained during the transfer of the small file chunks (called Small and Middle in Algorithm 1) is significantly worse compared to large chunks (Large and Huge in Algorithm 1) due to the high overhead of reading too many files from disk and underutilization of the

Algorithm 2. — Pro-Active Multi-Chunk (ProMC) Algorithm

```
function TRANSFER(source,destination,BW,RTT,concurrency)
    $BDP = BW * RTT$
    $allFiles = fetchFilesFromServer()$
    $chunks = partitionFiles(allFiles, BDP)$
    for $i = 0; i < chunks.length; i++$ do
        if $chunks[i] == SMALL$ then
            $weights[i] = 6 * chunks[i].size$
        else if $chunks[i] == MIDDLE$ then
            $weights[i] = 3 * chunks[i].size$
        else if $chunks[i] == LARGE$ then
            $weights[i] = 2 * chunks[i].size$
        else if $chunks[i] == HUGE$ then
            $weights[i] = 1 * chunks[i].size$
            $totalWeight = totalWeight + weights[i]$
        end if
    end for
    for $i = 0; i < chunks.length; i++$ do
        $weights[i] = weights[i]/totalWeight$        ▷ Calculate proportional weight of each chunk
        $channelAllocation[i] = \lfloor concurrency * weights[i] \rfloor$
    end for
    $transferChunks(chunks)$        ▷ Run chunks concurrently
end function
```

network pipe. Depending on the weight of small files relative to the total dataset size, overall throughput can be much less than the throughput of large file chunk transfers. Thus, we developed the MC method which aims to minimize the effect of poor transfer throughput of a dataset dominated by small files.

The MC method distributes data channels among chunks using round-robin in the order of Huge–Small–Large–Middle. The ordering of chunks provides better chunk distribution if the number of channels is less than the number of chunks. After channel distribution is completed, MC schedules chunks concurrently using the calculated concurrency level for each chunk.

The estimated completion time for each chunk is calculated every five seconds by dividing the remaining data size by the throughput of the chunk (i.e. the sum of the throughput for all channels for a given chunk). When the transfer of all files in a chunk is completed, the channels of the chunk are scheduled for other chunks based on their estimated completion time.

3.3 Pro-Active Multi-Chunk (ProMC) Algorithm

The way the MC approach distributes channels among chunks might be non-optimal if the weights of chunks are different. For example, if we have a dataset dominated by small files, then round-robin scheduling of channels may lead to sub-optimal channel allocation. This can cause sub-optimal transfer throughput since large chunks can be transferred more quickly than smaller chunks. The Pro-Active Multi-Chunk (ProMC) approach concentrates on more effectively distributing chunks among channels to improve the effectiveness of concurrency between the small and large chunks.

Channel allocation in the ProMC approach is demonstrated in Algorithm 2. ProMC also considers the type of a chunk when calculating its weight since the

Algorithm 3. — Max-Fair Multi-Chunk (FairMC) Algorithm

```
1: function TRANSFER(BW,RTT,BufferSize)
2:     BDP = BW*RTT
3:     allFiles = fetchFilesFromServer()
4:     chunks = partitionFiles(allFiles,BDP)
5:     if chunks contains Huge&Large chunk c then
6:         $c.channels++$                    ▷ Allocate a channel for huge&large chunk
7:         $concurrency--$
8:         if chunks contains Small&Middle chunks then
9:             allocateChannels(Small,Middle,concurrency);  ▷ If there exist Small&Middle chunks then allocate rest of channels to them
10:        else
11:            allocateChannel(Large,Huge,1);
12:        end if
13:    else
14:        allocateChannel(Small,Middle,concurrency) ▷ If there is no large chunk, then distribute given channels among Small&Middle chunks
15:    end if
16:    $transferChunks(chunks)$                 ▷ Run chunks concurrently
17: end function
```

transfer time of a chunk heavily depends its file distribution. Another way of achieving fairness among chunks in ProMC is dynamic channel allocation. It calculates the transfer completion time of each chunk periodically (by default, it is set to check every five seconds, similar to MC). If a chunk's completion time is calculated to be significantly less than another chunk's completion time for three consecutive periods, then a channel is taken over by the slow chunk from the faster chunk. Since channel transfer from one chunk to another is a costly operation, the threshold must be chosen carefully when comparing completion time differences. Also, rather than deciding on channel allocation after each period, ProMC waits three periods to make sure the estimated completion time difference is not a temporary condition.

3.4 Max-Fair Multi-Chunk (FairMC) Algorithm

The idea behind the Max-Fair Multi-Chunk (FairMC) approach is to make use of concurrent chunk transfers and to keep network and end-system utilization at a fair level. FairMC first calculates how many channels are needed for each chunk. Then, if small and large chunks exist, it opens only one channel for large chunks and uses the rest of the available channels for small chunks as shown in lines 5-9 of Algorithm 3. Otherwise, the channels are shared between small or large chunks as shown in lines 11 and 14. The goal here is to achieve high performance throughput without violating network fairness policies.

4 Performance Evaluation

We tested our experiments on XSEDE [18] and LONI [15] production-level high-bandwidth networks. Although both of the networks have 10G network bandwidth between sites, XSEDE provides higher throughput in end-to-end (disk-to-disk) transfers despite the high RTT between its sites. This is mainly due to the highly tuned and parallelized disk sub-systems at the XSEDE sites.

Table 1. Network specifications of test environments

Specs	**XSEDE**		**LONI**
	(Lonestar-Gordon)	**(Blacklight-Trestles)**	**(Queenbee-Painter)**
Bandwidth	10 Gbps	10Gbps	10 Gbps
RTT	60 ms	71 ms	10 ms
TCP Buffer Size	32 MB	32 MB	16 MB
BDP	75 MB	90 MB	9 MB

On XSEDE, we tested our dynamic protocol tuning algorithms between two different site pairs – Lonestar-Gordon and Blacklight-Trestles – with specifications given in Table 1. We also tested our algorithms using two different datasets where file sizes range between 3MB and 20GB. The datasets differ in the proportion of small files to the total dataset size. In the first one (referred to as "mixed"), small files are almost 35-40% of the total dataset, whereas they make up 55-65% of the second dataset (referred to as "small"). The purpose of using two different datasets is to demonstrate how our algorithms perform when the dataset is dominated by small or large files.

To analyze the effects of different parameters on the transfer of different file sizes, we initially conducted experiments for each of the parameters separately, as shown in Figure 1. We transferred each dataset, only changing one parameter (i.e., pipelining, parallelism, or concurrency) at a time to observe the individual effect of each parameter. Then we introduced other parameters one-by-one. These results show that concurrency is the most influential parameter for all file sizes on both networks, with parallelism being the second most. For this reason, we use concurrency as the pivot parameter in the comparison of different algorithms in this section. In all of our algorithms, it is assumed that RTT, bandwidth, and TCP buffer capacities are known beforehand. However, one can easily obtain RTT and TCP buffer capacity with negligible overhead. Available bandwidth can also be measured via bandwidth estimator tools (e.g. Iperf) with the cost of a couple of seconds.

We compared the performance of our four dynamic protocol tuning algorithms with Globus Online [1], PCP [19], and optimized globus-url-copy (GUC). Globus Online is a well-known data transfer service which uses a heuristic approach for transfer optimizations. The heuristic they use is similar to our basic Single-Chunk (SC) method in terms of dividing the dataset into chunks and running each chunk sequentially using different parameter sets. However, SC and Globus Online differ in the way they divide the chunks and in choosing the parameter set for each chunk. PCP employs a similar divide-and-transfer approach like SC and Globus Online using its own specific heuristic. For GUC, we set the pipelining to 30, parallelism to 4, and changed concurrency to different values (specifically, 2, 6, and 10) for different runs. We chose the pipelining and parallelism parameters in a way that they give close-to-best results based on our prior observations.

Results for Globus Online transfers are shown for concurrency level two as it always uses two channels for every chunk it creates. Although the PCP algorithm does not use a statically defined concurrency level, we observed that it

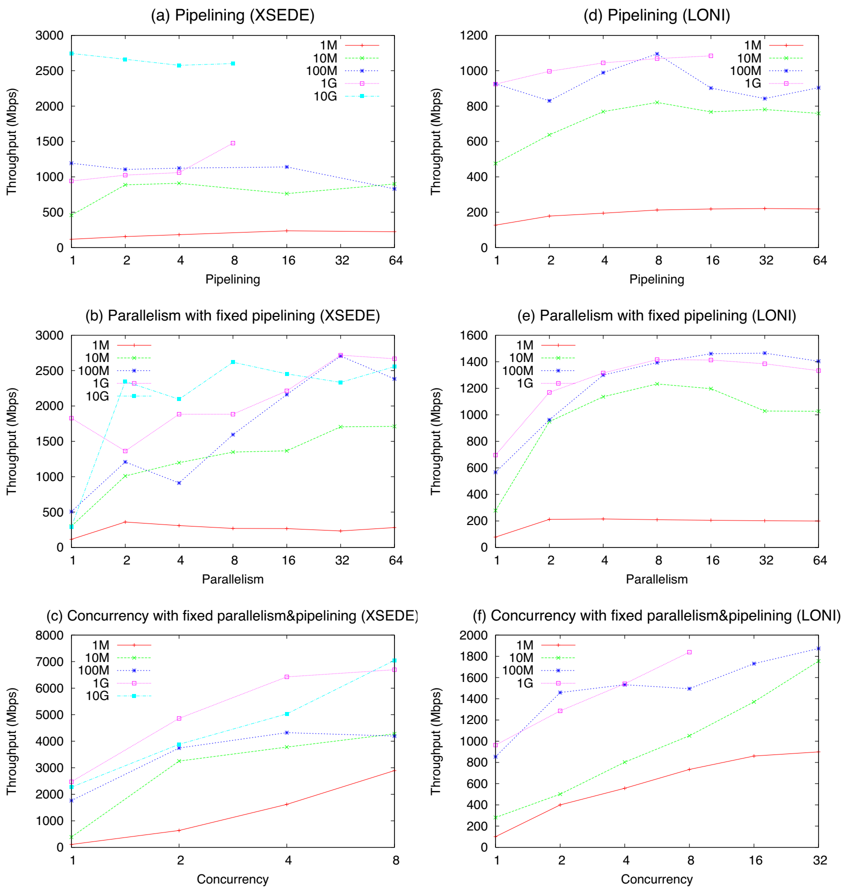


Fig. 1. Effect of combining parameters on throughput

generally choses a concurrency level between one and three; we set its performance histogram on concurrency level two. When concurrency level is set to two for all algorithms, almost all of our dynamic tuning algorithms perform better than PCP, Globus Online, and GUC. As we increase the concurrency level (as our dynamic algorithms do so), we can see significant performance improvement for all algorithms. However, SC is unable to improve the performance after concurrency level six while MC makes use of concurrency more effectively and its performance continues to increase in proportion to the concurrency level.

FairMC also improves in performance as concurrency increases. However, it does not perform as well as MC and ProMC, since it limits concurrency levels for large files and aims for fairness in lieu of maximum performance. ProMC and MC achieve similar performance in this case, since ProMC plays a significant role when small files dominate the data set. ProMC and MC perform better than GUC for all concurrency levels, which is mostly due to the efficient channel management of these two algorithms.

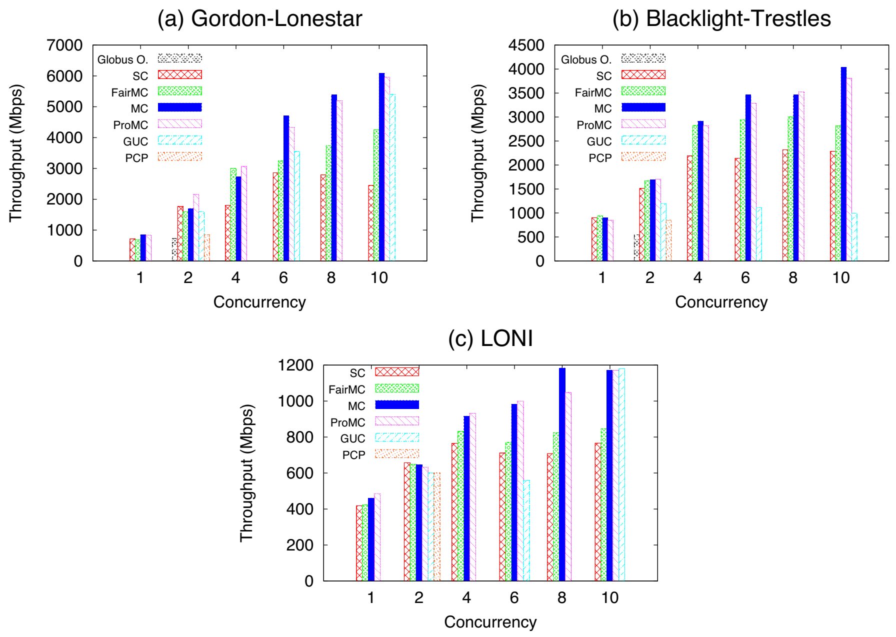
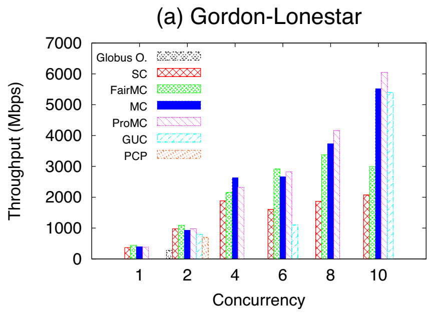


Fig. 2. Disk-to-disk transfer performance comparison with the mixed dataset

Figures 2(a) and 2(b) show the performance of optimization methods when used between different site pairs on XSEDE for the same dataset. Since disk read/write performance on Blacklight-Trestles is less than that on Gordon-Lonestar, throughput values obtained between these hosts are relatively smaller. GUC performance is very low compared to MC, since, when the pipelining is set in the GUC transfer, it statically sets the pipelining level for each channel to the number of transfer tasks. This means that it is possible for files to be assigned to the channels unequally. For example, one channel can be assigned to transfer the set of files contributing to the dominant portion of total dataset size. This will cause inefficient usage of channels, since, even if some channels finish their transfer tasks earlier, they will not be able to help others by sharing the remaining tasks.

We observed that LONI end-system disk performance is much lower than on XSEDE sites. This affected the results we obtained from LONI considerably as shown in Figure 2(c). Although the increased concurrency contributed positively to the throughput on LONI, the improvement is not as noticeable as it is on XSEDE.

In Figure 3, we observe that ProMC performed better than MC for almost all concurrency levels, as it allocates the channels to chunks more efficiently. It also monitors each chunk's performance and acts to re-allocate a channel from one chunk to another to minimize the negative effect of small file transfers on overall transfer performance.



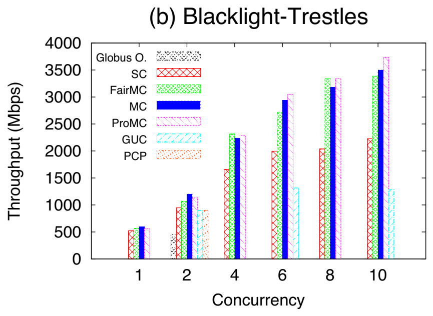


Fig. 3. Disk-to-disk performance comparison with the small file dominant dataset

5 Conclusions

We have presented four application-level algorithms for heuristically tuning protocol parameters for data transfers in wide-area networks. The parameters dynamically tuned by our algorithms (parallelism, pipelining, and concurrency levels) have shown to be very important factors in determining the ultimate throughput and network utilization obtained by many data transfer applications. Though determining the best combination for these parameter values is not a trivial task, we have shown that our algorithms can choose parameter combinations which yield demonstrably higher throughputs than those used in unoptimized transfers or chosen by less sophisticated heuristics.

Our algorithms were designed to be client-side techniques and operate entirely in user space, and thus special configurations at the server side or at the kernel level are not necessary to take advantage of them. The algorithms can be implemented as standalone transfer clients or as part of an optimization library or service. We plan to include these (and future algorithms based thereupon) in the Stork data scheduler [10,12] as well as our new Cloud-hosted transfer optimization suite, StorkCloud.

Acknowledgments. This project is in part sponsored by NSF under award numbers CNS-1131889 (CAREER), OCI-0926701 (STCI-Stork), and CCF-1115805 (CiC-Stork).

References

1. Allen, B., Bresnahan, J., Childers, L., Foster, I., Kandaswamy, G., Kettimuthu, R., Kordas, J., Link, M., Martin, S., Pickett, K., Tuecke, S.: Software as a service for data scientists. Communications of the ACM 55(2), 81–88 (2012)
2. Altman, E., Barman, D.: Parallel TCP sockets: Simple model, throughput and validation. In: Proceedings of IEEE INFOCOM (2006)
3. Bresnahan, J., Link, M., Kettimuthu, R., Fraser, D., Foster, I.: Gridftp pipelining. In: Proceedings of TeraGrid (2007)

4. Farkas, K., Huang, P., Krishnamurthy, B., Zhang, Y., Padhye, J.: Impact of TCP variants on HTTP performance. In: Proceedings of High Speed Networking, vol. 2 (2002)
5. Freed, N.: SMTP service extension for command pipelining, `http://tools.ietf.org/html/rfc2920`
6. Hacker, T.J., Noble, B.D., Athey, B.D.: Adaptive data block scheduling for parallel TCP streams. In: Proceedings of HPDC (2005)
7. Hacker, T.J., Noble, B.D., Atley, B.D.: The end-to-end performance effects of parallel TCP sockets on a lossy wide area network. In: Proc. of IPDPS (2002)
8. Khanna, G., Catalyurek, U., Kurc, T., Kettimuthu, R., Sadayappan, P., Foster, I., Saltz, J.: Using overlays for efficient data transfer over shared wide-area networks. In: Proceedings of SC, Piscataway, NJ, USA (2008)
9. Kim, J., Yildirim, E., Kosar, T.: A highly-accurate and low-overhead prediction model for transfer throughput optimization. In: Proceedings of ACM SC 2012 DISCS Workshop (2012)
10. Kosar, T.: A new paradigm in data intensive computing: Stork and the data-aware schedulers. In: Proceedings of IEEE HPDC 2006 CLADE Workshop (2006)
11. Kosar, T., Balman, M.: A new paradigm: Data-aware scheduling in grid computing. Future Generation Computing Systems 25(4), 406–413 (2009)
12. Kosar, T., Balman, M., Yildirim, E., Kulasekaran, S., Ross, B.: Stork data scheduler: Mitigating the data bottleneck in e-science. The Phil. Transactions of the Royal Society A 369(3254-3267) (2011)
13. Kosar, T., Livny, M.: Stork: Making data placement a first class citizen in the grid. In: Proceedings of ICDCS 2004, pp. 342–349 (March 2004)
14. Liu, W., Tieman, B., Kettimuthu, R., Foster, I.: A data transfer framework for large-scale science experiments. In: Proceedings of DIDC Workshop (2010)
15. LONI: Louisiana optical network initiative (LONI), `http://www.loni.org/`
16. Lu, D., Qiao, Y., Dinda, P.A., Bustamante, F.E.: Modeling and taming parallel TCP on the wide area network. In: Proceedings of IPDPS (2005)
17. Raiciu, C., Pluntke, C., Barre, S., Greenhalgh, A., Wischik, D., Handley, M.: Data center networking with multipath TCP. In: Proceedings of Hotnets-IX (2010)
18. XSEDE: Extreme Science and Engineering Discovery Environment, `http://www.xsede.org/`
19. Yildirim, E., Kim, J., Kosar, T.: How gridftp pipelining, parallelism and concurrency work: A guide for optimizing large dataset transfers. In: Proceedings of Network-Aware Data Management Workshop (NDM 2012) (November 2012)
20. Yildirim, E., Kim, J., Kosar, T.: Optimizing the sample size for a cloud-hosted data scheduling service. In: Proc. of IEEE/ACM CCGrid CCSA Workshop (2012)
21. Yildirim, E., Kosar, T.: Network-aware end-to-end data throughput optimization. In: Proceedings of Network-Aware Data Management Workshop (NDM 2011) (2011)
22. Yildirim, E., Yin, D., Kosar, T.: Prediction of optimal parallelism level in wide area data transfers. IEEE TPDS 22(12) (2011)
23. Yildirim, E., Yin, D., Kosar, T.: Balancing TCP buffer vs parallel streams in application level throughput optimization. In: Proceedings of DADC Workshop (2009)
24. Yin, D., Yildirim, E., Kosar, T.: A data throughput prediction and optimization service for widely distributed many-task computing. IEEE TPDS 22(6) (2011)