Application-Level Optimization of Big Data Transfers Through Pipelining, Parallelism and Concurrency

Esma Yildirim, Engin Arslan, Jangyoung Kim, and Tevfik Kosar, *Member, IEEE*

Abstract—In end-to-end data transfers, there are several factors affecting the data transfer throughput, such as the network characteristics (e.g. network bandwidth, round-trip-time, background traffic); end-system characteristics (e.g. NIC capacity, number of CPU cores and their clock rate, number of disk drives and their I/O rate); and the dataset characteristics (e.g. average file size, dataset size, file size distribution). Optimization of big data transfers over inter-cloud and intra-cloud networks is a challenging task that requires joint-consideration of all of these parameters. This optimization task becomes even more challenging when transferring datasets comprised of heterogeneous file sizes (i.e. large files and small files mixed). Previous work in this area only focuses on the end-system and network characteristics however does not provide models regarding the dataset characteristics. In this study, we analyze the effects of the three most important transfer parameters that are used to enhance data transfer throughput: *pipelining, parallelism* and *concurrency*. We provide models and guidelines to set the best values for these parameters and present two different transfer optimization algorithms that use the models developed.The tests conducted over high-speed networking and cloud testbeds show that our algorithms outperform the most popular data transfer tools like Globus Online and UDT in majority of the cases.

Index Terms—Pipelining, parallelism, concurrency, TCP, GridFTP, data transfer optimization, high-bandwidth networks, cloud networks

1 Introduction

Most scientific cloud applications require movement of large datasets either inside a data center, or between multiple data centers. Transferring large datasets especially with heterogeneous file sizes (i.e. many small and large files together) causes inefficient utilization of the available network bandwidth. Small file transfers may cause the underlying transfer protocol not reaching the full network utilization due to short-duration transfers and connection start up/tear down overhead; and large file transfers may suffer from protocol inefficiency and end-system limitations. Application-level TCP tuning parameters such as pipelining, parallelism and concurrency are very affective in removing these bottlenecks, especially when used together and in correct combinations. However, predicting the best combination of these parameters requires highly complicated modeling since incorrect combinations can either lead to overloading of the network, inefficient utilization of the resources, or unacceptable prediction overheads.

Among application level transfer tuning parameters, *pipelining* specifically targets the problem of transferring large numbers of small files [1]. It has two major goals: first, to prevent the data channel idleness and to eliminate the idle time due to control channel conversations in between the consecutive transfers. Secondly, pipelining prevents TCP window size from shrinking to zero due to idle data channel time if it is more than one Round Trip Time (RTT). In this sense, the client can have many outstanding transfer commands without waiting for the *"226 Transfer Successful"* message. For example, if the pipelining level is set to four in GridFTP [1], five outstanding commands are issued and the transfers are lined up back-to-back in the same data channel. Whenever a transfer finishes, a new command is issued to keep the pipelining queue full. In the latest version of GridFTP, this value is set to 20 statically by default and does not allow the user to change it. In Globus Online [2], this value is set to 20 for more than 100 files of average 50MB size, 5 for files larger than 250MB and in all other cases it is set to 10. Unfortunately, setting static parameters based on the number of files and file sizes is not affective in most cases, since the optimal pipelining level also depends on the network characteristics such as bandwidth, RTT, and background traffic.

Using *parallel streams* is a very popular method for overcoming the inadequacies of TCP in terms of utilizing the high-bandwidth networks and has proven itself over socket buffer size tuning techniques [3], [4], [5], [6], [7], [8]. With parallel streams, portions of a file are sent through multiple TCP streams and it is

- *E. Yildirim is with the Department of Computer Engineering, Fatih University, Istanbul, Turkey. E-mail: esma.yildirim@fatih.edu.tr*
- *Engin Arslan and Tevfik Kosar are with University at Buffalo, New York, USA*
- *Jangyoung Kim is with University of Suwon, Korea*

possible to achieve multiples of the throughput of a single stream.

Setting the optimal parallelism level is a very challenging task and several models have been proposed in the past [9], [10], [11], [12], [13], [14], [15], [16]. The Mathis equation[17] states that the throughput of a TCP stream(BW) depends on the Maximum Segment Size(MSS), Round Trip Time(RTT), a constant(C) and packet loss rate(p).

$$BW = \frac{MSS \times C}{RTT \times \sqrt{p}} \quad (1)$$

As the packet loss rate increases, the throughput of the stream decreases. The packet loss rate can be random in under-utilised networks however when there is congestion, it increases dramatically. In [9], a parallel stream model based on the Mathis equation is given.

$$BW_{agg} \leq \frac{MSS \times C}{RTT}[\frac{1}{\sqrt{p_1}}...\frac{1}{\sqrt{p_n}}] = n\frac{MSS \times C}{RTT \times \sqrt{p}} \quad (2)$$

According to that, use of n parallel streams can produce n times the throughput of a single stream. However excessive use of parallel streams can increase the packet loss rate dramatically, causing the congestion avoidance algorithm of TCP to decrease the sending rate based on the losses encountered. Therefore, the packet loss happening in our case occurs due to congestion. In our previous study[11], we presented a model to find the optimal level of parallelism based on the Mathis throughput equation. However these models can be applied only for very large files where TCP can reach to its maximum sending rate (Maximum window size).

Most studies target the optimisation of large files and do not discuss the effect of parallelism in transferring a set of small files which is harder to base on a theoretical model. Therefore, in Globus Online[2], the parallelism level is set to 2, 8 and 4 respectively for the cases mentioned in the second paragraph above.

In the context of transfer optimization, *concurrency* refers to sending multiple files simultaneously through the network channel. In [18], the effects of concurrency and parallelism were compared for large file transfers. Concurrency is especially good for small file transfers, and overcoming end system bottlenecks such as CPU utilisation, NIC bandwidth, parallel file system characteristics[19](e.g. Lustre file system distributes files evenly and can provide more throughput in multi-file transfers). The Stork data scheduler [20], [21] has the ability to issue concurrent transfer jobs and in most of the cases, concurrency has proven itself over parallelism. Another study [22], adapts the concurrency level based on the changes in the network traffic, does not take into account the other bottlenecks that can occur on the end systems. Globus Online sets this parameter to 2 along with the other settings

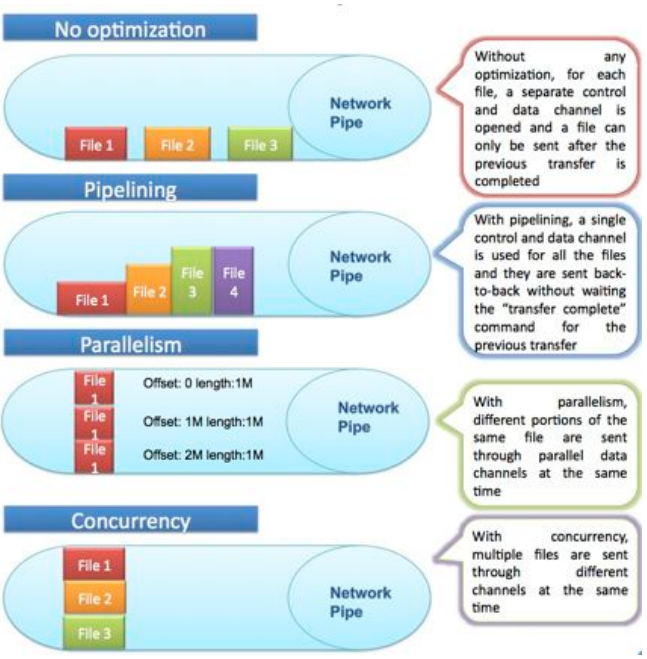

Fig. 1. Visual presentation and comparison of pipelining, parallelism and concurrency to non-optimized transfers.

for pipelining and parallelism. A full comparison of pipelining, parallelism and concurrency is presented in Figure 1, to indicate the differences of each method from each other and from a non-optimized transfer.

In this study, we provide a step-by-step solution to the problem of big data transfer bottleneck for scientific cloud applications. First, we provide insight into the working semantics of application-level transfer tuning parameters such as pipelining, parallelism and concurrency. We show how to best utilize these parameters in combination to optimize the transfer of a large dataset. In contrast to other approaches, we present the solution to the optimal settings of these parameters in a question-and-answer fashion by using experiment results from actual and emulation testbeds. As a result, the foundations of dynamic optimization models for these parameters are outlined, and several rules of thumbs are identified. Next, two heuristics algorithms are presented that apply these models. The experiments and validation of the developed models are performed on high-speed networking testbeds and cloud networks. The results are compared to the most successful and highly adopted data transfer tools such as Globus Online and UDT [23]. It has been observed that our algorithms can outperform them in majority of the cases.

2 Answers to Formidable Questions

There are various factors affecting the performance of pipelining, parallelism and concurrency; and we list some questions to provide a guide for setting

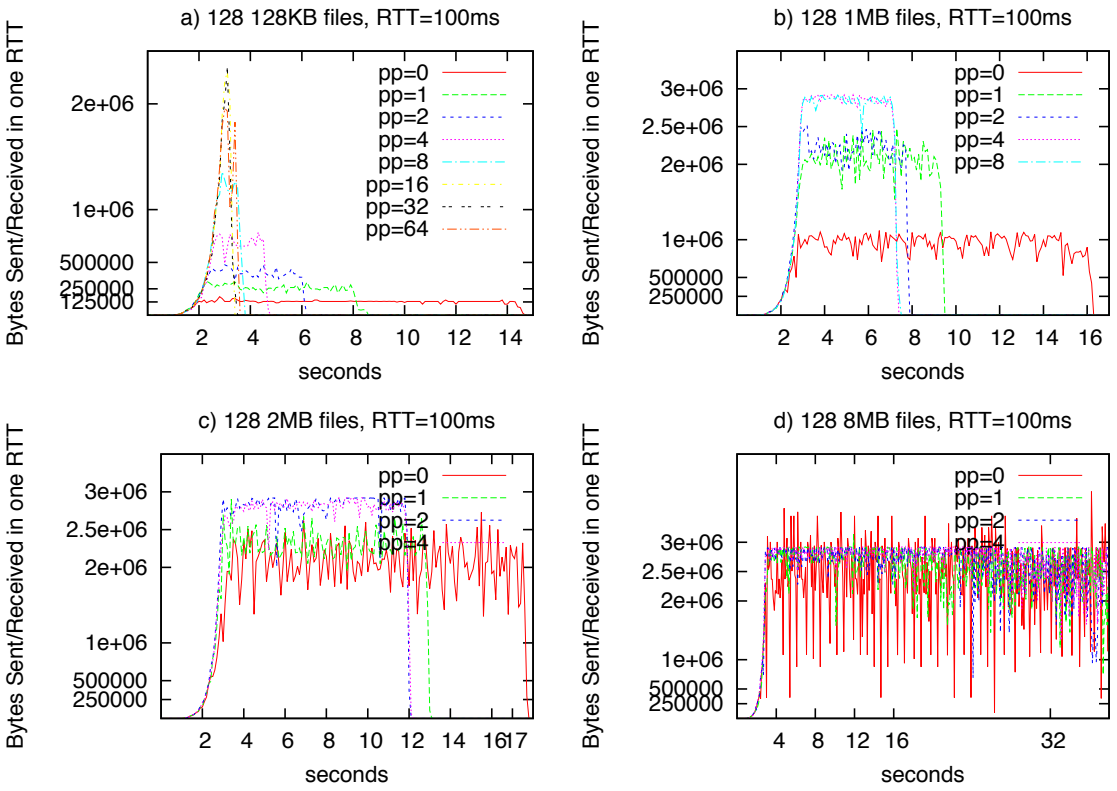

Fig. 2. Emulab [1 Gbps network interface, 100 msec RTT] – Effect of file size and pipelining level on Bytes Sent/Recv in one RTT.(Question 2)

optimal numbers for these parameters. The answers to these questions are given with experimental results from high-speed networks with short and long RTTs (LONI), an emulation testbed (Emulab [24]) that allows the user to set the topology, bandwidth and RTT and AWS EC2 instances with various performance categories. The GridFTP version 5.2.X and UDT version 4 are used in the experiments. The questions posed are listed in Table 1. Throughout the paper, pipelining parameter will be represented with pp, parallelism will be represented with p and concurrency will be represented with cc. Also a *Dataset* will refer to a set, consisting of large number of numerous-size-files and a *Chunk* will refer to a portion of a dataset.

TABLE 1
Questions Posed

1. Is pipelining necessary for every transfer?
2. How does file size affect the optimal pipelining level?
3. Does the number of files affect the optimal pipelining level?
4. What is the minimum amount of data in a chunk for setting different pipelining levels to be effective?
5. Is there a difference in setting optimal pipelining level between long-short RTTs?
6. Is there a difference between performances of data channel caching and pipelining ?
7. Is it fine to set parallelism and concurrency levels after optimizing pipelining?
8. When is parallelism advantageous?
9. How much parallelism is too much?
10. Is concurrency sufficient by itself without parallelism or pipelining?
11. What advantages does concurrency have over parallelism?
12. How does network capacity affect the optimal parallelism and concurrency levels?
13. When to use UDT over TCP?

2.1 Is pipelining necessary for every transfer?

Obviously pipelining is useful when transferring large numbers of small files, but there is a certain breakpoint where the average file size becomes greater than the bandwidth delay product (BDP). After that point, there is no need to use a high level of pipelining. So if we have a data set of files with varying sizes, it is important to divide the dataset into two and focus on the part (file size $<$ BDP) where setting different pipelining levels may affect the throughput. BDP is calculated by taking bulk TCP disk-to-disk throughput for a single TCP stream for bandwidth and average RTT for the delay. In the following subsection, a model is presented for setting the optimal pipelining level.

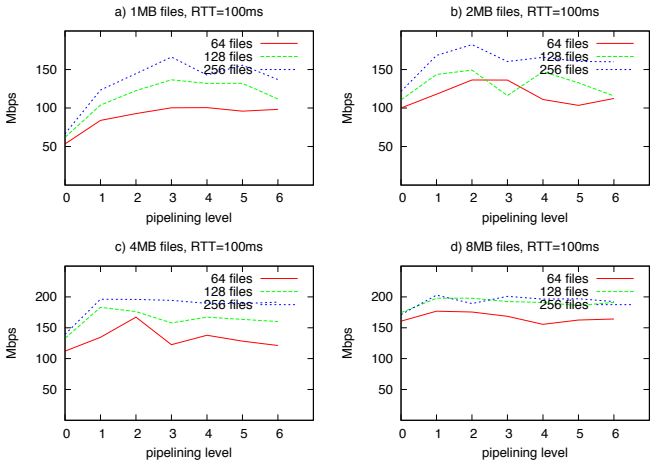

Fig. 3. Emulab [1 Gbps network interface, 100 msec RTT] – Effect of number of files on optimal pipelining level.(Question 3)

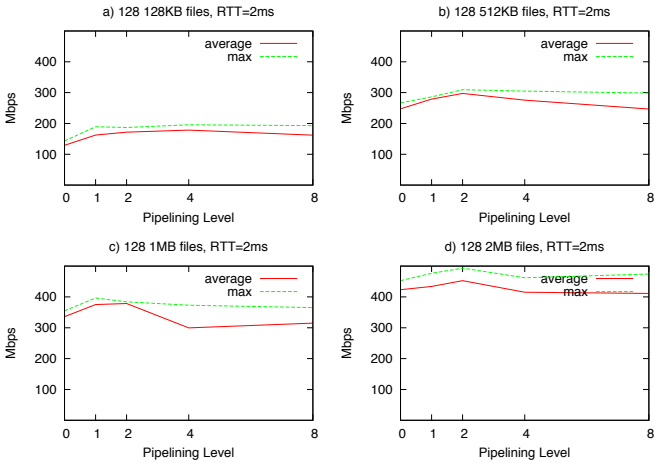

Fig. 4. Emulab [1 Gbps network interface, 2ms RTT] – Effect of pipelining in short RTT networks.(Question 5)

2.1.1 How does file size affect the optimal pipelining level?

File size is the dominating factor in setting the optimal pipelining level, especially for long RTT networks. Figure 2 presents the results of disk-to-disk GridFTP transfers of data sets with 128 different-size files in an Emulab setting of 1 Gbps network bandwidth and 100 ms RTT. A network analyzer tool called *tshark* is used to calculate the statistics about the number of bytes sent/received in each RTT. Data channel conversation is isolated from the control channel conversation and the graphics only present the data channel traffic. One important observation is that different pipelining level transfers go through similar slow start phases regardless of the file size. The crucial point is the highest number of bytes reached by a specific file size. In all of the cases, this is equal to:

$$B = FS \times (pp + 1) \tag{3}$$

where B is the number of bytes sent/received in one RTT, FS is the file size and pp is the pipelining level. Of course this linear increase in the number of bytes with the pipelining level only lasts when it reaches BDP. After that, the increase becomes logarithmic. Therefore the optimal pipelining level could be calculated as:

$$pp_{opt} \simeq \lceil BDP/FS \rceil - 1 \tag{4}$$

2.1.2 Does the number of files affect the optimal pipelining level?

The number of files only affects the total throughput but does not have an effect on the optimal pipelining level. It is obvious from Figure 2, that the slow start curves show the same characteristics for different pipelining levels. Therefore, increasing the number of files only lengthens the proportion where data is transferred at full capacity of the network. In Figure 3, a comparison of disk-to-disk transfers with different number of files in the range [64-256] is presented for different file sizes in the same testbed settings (1 Gbps bandwidth, 100 ms RTT) as in the previous section. In almost all of the cases the optimal pipelining level is the same for different number of files. Also in all of the cases, the increase in the number of files causes an increase in the total throughput which proves our hypothesis about the number of files. If it was instant throughput at one time, the throughput results would have been the same after the saturation point. However since the average throughput is shown in this case, although all the transfers might reach to the same maximum transfer speed (in our case it is when they send as much data as the BDP in one RTT), the average throughput of a transfer will be less for small number of files since the percentage of data transfer in maximum capacity will be less comparing to the whole data transfer . Similar results are obtained for short RTT networks (LONI); however, they are not presented here due to space considerations. At that point, another important question is brought to clarify whether or not we should apply different pipelining levels on different file chunks.

2.1.3 What is the minimum amount of data in a chunk for setting different pipelining levels to be effective?

Since different file sizes have the same slow start phase characteristics, only differ in the highest number of bytes sent/received in one RTT for different pipelining levels, there must be at least a sufficient number of files in a chunk that will pass the transfer through the slow start phase. Otherwise even if we set the optimal pipelining level for the chunk of files, too small chunk sizes with different file size averages will not improve the throughput. To calculate the min-

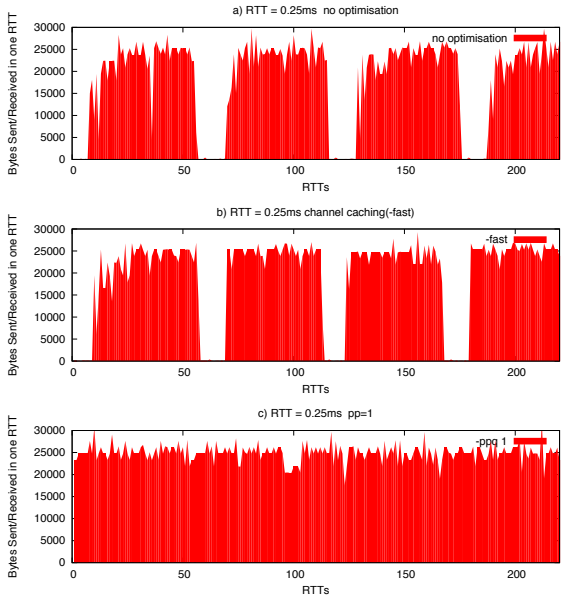

Fig. 5. [1 Gbps network interface, 0.25 ms RTT] – Pipelining vs Data Channel Caching (Question 6)

imum chunk size, a model from a previous study [25] is used in the optimization algorithm presented in Section 4. This model uses a mining and statistics method to derive the smallest data size to transfer to predict the transfer throughput accurately.

2.1.4 Is there a difference in setting optimal pipelining level between long-short RTTs?

Section 2.1.1 mentions that file size, along with the pipelining level, affects the maximum number of bytes sent/received in one RTT especially if BDP is much higher than file size. Therefore RTT has an indirect effect since it is used in the calculation of BDP. For short-RTT networks such as local-area or metropolitan-area networks, pipelining loses its effect considerably. The size of a file becomes larger than the BDP value and pipelining is only used to close the processing overhead and control channel conversation gaps between the transfers. In this sense, we can no longer talk about the linear increase in throughput with increasing pipelining levels and setting a pipelining level of 1-2 would be enough to reach the maximum achievable throughput. Figure 4 presents the throughput results of the disk-to-disk transfer of 128 files for different file sizes in an Emulab setting of 1 Gbps network interface and 2 ms RTT. The optimal pipelining level does not depend on the file size and 1 or 2 pipelining level is enough to achieve the highest throughput.

2.1.5 Is there a difference between performances of data channel caching and pipelining?

When the average file size is greater than the BDP, there is no need to use large pipelining numbers and a pipelining level of 1 is sufficient to fully utilise the BDP. With pipelining, the control messages and data transfers overlap. The pipelined data transfer of multiple small files can act like a large file data transfer (Figure 2). One would argue that for files with sizes larger than BDP, data channel caching(Using the same channel for multiple file transfers) can achieve the same effect. It is true but only to a certain extent. Data channel caching does not let control channel messages to overlap with data transfers. It still waits for one transfer to finish to start the other. Figure 5 presents a comparison of pipelining and data channel caching and a case with no optimisation. 128 1MB files are transferred in a local area 1Gbps network whose BDP is 25K approximately. In Figure 5.a , where no optimisation is applied, each 1MB file is transferred back to back with a few RTTs in between (processing overhead and control channel messages). The effect of a growing window size can be seen for every file in the figure. When data channel caching is applied (Figure 5.b), the distance between consecutive transfers is still there, however the effect of a growing window size can only be observed in the first file transfer. The other file transfers start with the largest possible window size which is BDP in this case. Looking at pipelining level of 1(Figure 5.c), the distance disappears due to data transfers overlapping with processing overhead and control channel messages. The transfer continues at the highest possible window size which is 25K. The same experiment with larger files(128 8MB files) also presented the same distance remain between consecutive transfers. The results indicate that it is better to use pipelining rather than data channel caching although the performance differences might be very small. For large bandwidth, long RTT networks the large window size provided by data channel caching will do the trick for consecutive transfers but this small distance will still be there. However with pipelining even this small distance will disappear.

2.1.6 Is it fine to set parallelism and concurrency levels after optimizing pipelining?

An optimized transfer of a data set with pipelining actually looks like the transfer of a big file transfer without optimization (Figure 2). Therefore setting the optimal pipelining level actually makes things easier to find and set optimal parallelism and concurrency levels.

2.2 When is parallelism advantageous?

Parallelism is advantageous when the system buffer size is set smaller compared with the BDP. This occurs mostly in large bandwidth-long RTT networks. It is also advisable to use parallelism in large file transfers. In the case of small files, parallelism may not give a good performance by itself, however when used

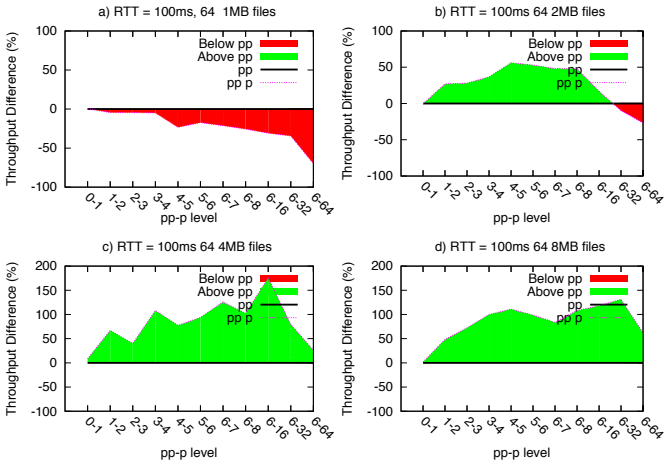

Fig. 6. Emulab [1 Gbps network interface, 100 ms RTT] – Effect of file size and parallelism on throughput (64 files transfer with pipelining and parallelism).(Question 8)

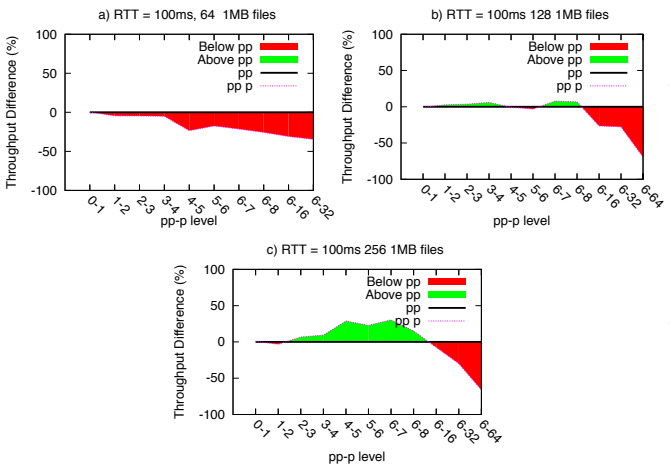

Fig. 7. Emulab [1 Gbps network interface , 100ms RTT] – Effect of number of files and parallelism on throughput (1MB file transfer with pipelining and parallelism).(Question 8)

with pipelining its effects on the performance could be significant as long as it does not cause pipelining to lose its effect due to division of small files into chunks by parallelism. This happens when the number of files and average file size in a chunk are small. In Figure 6, for the same Emulab settings of 1 Gbps network interface and 100ms RTT, parallelism has a negative effect when added to pipelining for the smallest file size case (Figure 6.a), however this effect is changed to positive as the file size increases. Similar effects could be seen with 128- and 256-file transfers. In Figure 7, the effect of parallelism is measured when the number of files is increased for 1MB files transferred with pipelining. Again the negative effect of parallelism is diminished and gradually turns to positive. However,

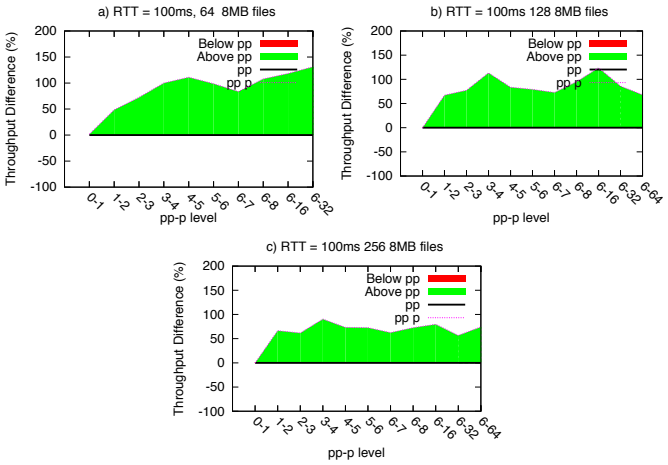

Fig. 8. Emulab [1 Gbps network interface, 100ms RTT] – Effect of number of files and parallelism on throughput (8 MB file transfer with pipelining and parallelism).(Question 8)

as the number of parallel streams increases further, data size is divided into more streams and having big data size again loses its positive effect. Therefore the improvement is more significant for large file sizes (Figure 8). Bigger file sizes and large number of files are good for parallelism.

2.3 How much parallelism is too much?

This is a difficult question to answer. If it were possible to predict when the packet loss rate would start to increase exponentially, it would also be possible how much parallelism would be too much. There are two cases to consider in terms of dataset characteristics. First, when the transfer is of a large file, the point the network or disk bandwidth capacity is reached and the number of retransmissions start to increase is the point where the parallelism level becomes too much. In our previous work [11], we managed to predict the optimal level by looking into throughput measurements of three past transfers with exponentially increasing parallelism levels. There is a knee point in the throughput curve as we increase the parallel stream number.

In the second case, when the transfer is of a dataset consisting of large number of small files, parallelism has a negative effect, because the data size becomes smaller as the file is divided into multiple streams and the window sizes of each stream can not reach to maximum because there is not enough data to send. With the help of pipelining this bottleneck can be overcome to an extent. In Figure 9, where multiple parallel streams are used on pipelined transfers of 64 1MB files, the parallel streams spend most of their time in the slow start phase. The more the parallelism, the larger the overhead and maximum window size can not be reached(Figure 10). Unfortunately, there is

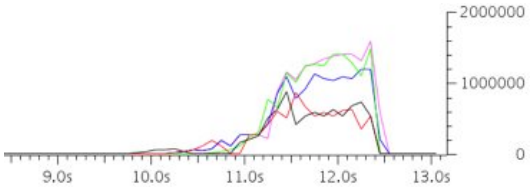

Fig. 9. AWS- [m3.xlarge instances-100ms RTT] - Parallelism effect on small datasets consisting of small files (64 1 MB files(default pipelining level applied)).(Question 9)

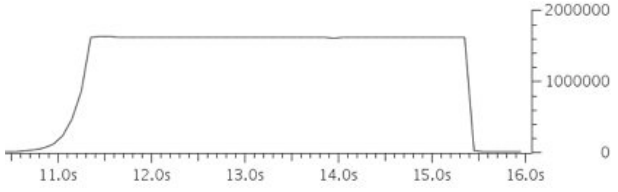

Fig. 10. AWS- [m3.xlarge instances-100ms RTT] - Pipelined transfer of a small dataset consisting of small files (64 1 MB files(default pipelining level applied)).(Question 9)

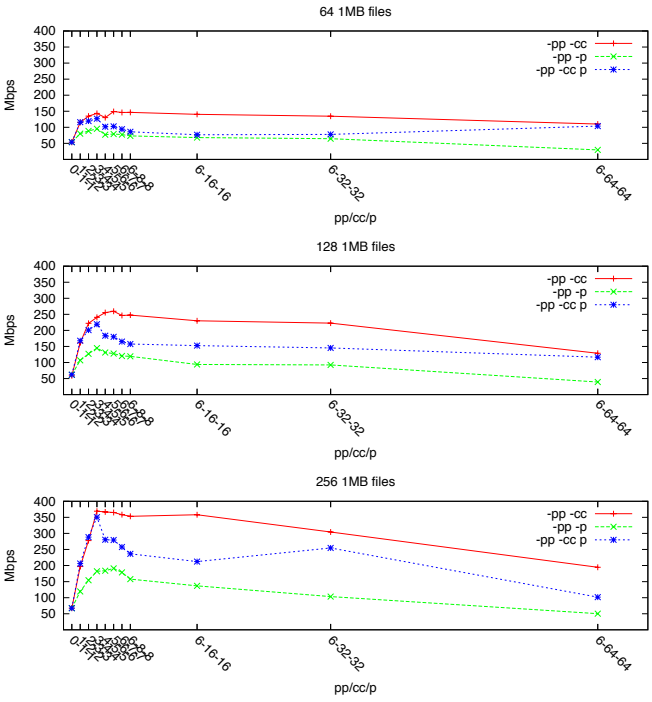

Fig. 11. Emulab [1 Gbps network interface, 100 ms RTT] – Comparison of parallelism and concurrency (1 MB files).(Question 11)

no way to tell the breaking point where individual parallel streams will spend their time in the slow-start phase of TCP.

2.4 Is concurrency sufficient by itself without parallelism or pipelining?

In cases where the bandwidth is fully utilized with small number of concurrent transfers and the number of files is large enough, concurrency, used with data channel caching can achieve similar performances with parallelism + pipelining + concurrency. However, the optimal concurrency level could be much higher when it is used alone. High number of concurrent transfers means many processes, which can degrade the performance. In this case, using concurrency with pipelining and parallelism is a good choice.

2.4.1 What advantages does concurrency have over parallelism?

In cases where parallelism deteriorates the performance improvements of pipelining, it is better to use concurrency. In Figure 11, concurrency with pipelining has better performance than using all three functions together for the same settings of 1 Gbps network interface and 2ms RTT in Emulab. This is due to the negative effect of parallelism on pipelining. For bigger file sizes, the negative effect of parallelism is degraded and when all three functions are used together they can perform as well as concurrency + pipelining case. For small RTT networks(LONI - 10 Gbps network interface - 2 ms RTT), where pipelining has little effect, we achieve a quicker ascend to the peak throughput and see better performance with parallelism + concurrency + pipelining since parallelism does not have a negative effect on pipelining(Figure 12). As the file size and number of files increases, using extra parallelism loses its effect, because concurrency can also provide a quick ascend to the maximum average throughput.

2.4.2 How does network capacity affect the optimal parallelism and concurrency levels?

It is necessary to remove the bottlenecks related to end-systems and dataset characteristics to measure the effect of network capacity over parallelism and concurrency. The most suitable testbed for this experiment is a collection of instances of AWS with 4 different network performance categories: Low, moderate, High and 10Gbps. 32 128MB files (8GB in total) were transferred with a 100ms artificial RTT in between the instances. The size of the files is chosen to be large especially to remove the effects of dataset characteristics. Figure 13 presents the effect of network capacity on parallelism (a), concurrency(b) and concurrency with data channel caching(c). The performance benefits of parallelism and concurrency is best observed in wide area data transfers. As a result of exponentially increasing parallel stream numbers and concurrency, the total throughput shows a linear increase first. However as the number goes high it comes closer to the network capacity and the increase becomes exponential and then starts to decrease or takes the form of a steady-state transfer.

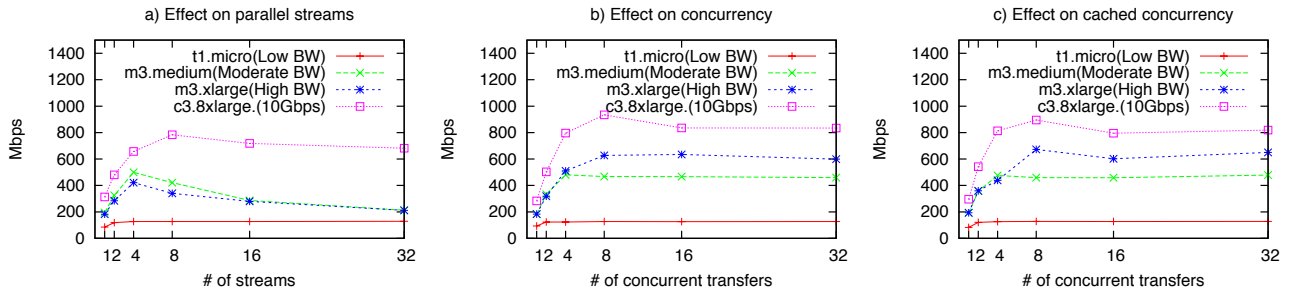

Fig. 13. Effect of Network Capacity on parallelism and concurrency- AWS instances(t1.micro, m3.medium, m3.xlarge, c3.8xlarge) - RTT = 100ms(Question 12)

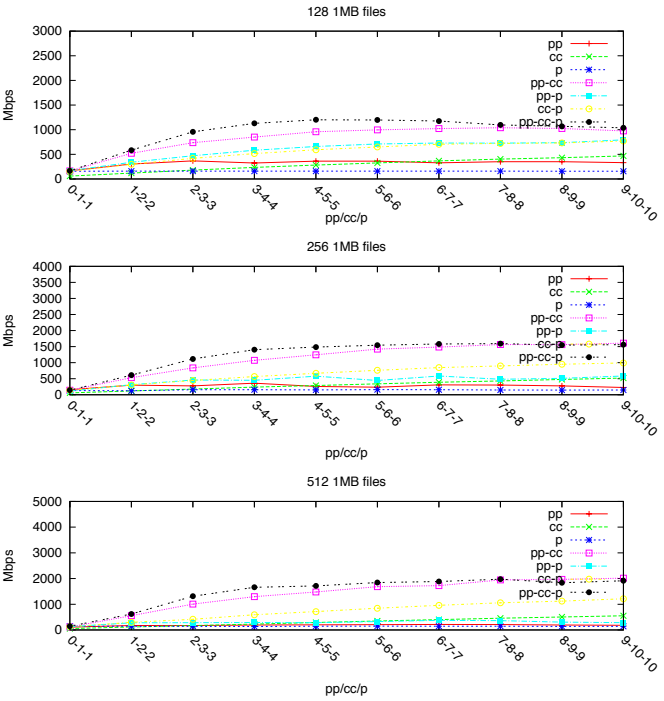

Fig. 12. LONI [10 Gbps network interface, 2ms RTT] – Comparison of pipelining, parallelism and concurrency (1MB files).(Question 11)

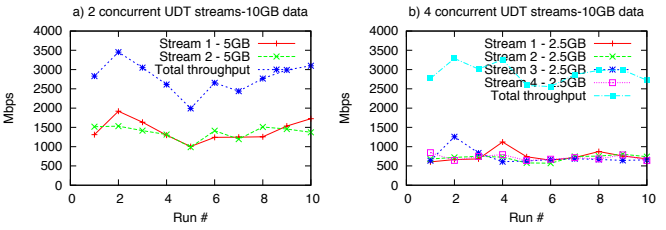

Fig. 14. Effect of Concurrency over UDT transfers, LONI - 1Gbps-RTT=2ms (Question 13)

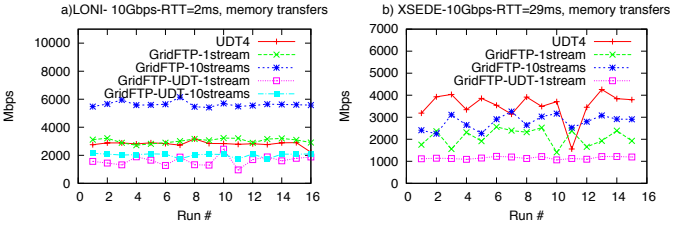

Fig. 15. UDT vs GridFTP, memory transfers (Question 13)

The most apparent outcome that can be deduced from these results is *that the optimal parallelism and concurrency levels increase as the network capacity increases.* With a few minor differences, the optimal parallel stream and concurrency number for t1.micro instances is 2, for m3.medium instances 4, for m3.xlarge instances 4-8 and for c3.8xlarge instances 8. These numbers are close to each other because there is not much of a difference in their optimal network bandwidth(100Mbps, 500Mpbs, 650Mbps, 900Mpbs respectively) and they are hardly multiples of each other except for t1. micro instances. Therefore the distinct parallelism and concurrency numbers are harder to see among m3.medium, m3.xlarge and c3.8xlarge instances. There can also be the underlying restrictions on the highest bandwidth usage for normal instances set by AWS.

2.5 When to use UDT over TCP?

There are many studies that compare UDT and TCP(GridFTP) in the literature ([26], [27], [28], [29]). However all the results belong to under 1Gbps networks and usually wide area transfers. According to their observations, parallel TCP performs better than UDT, however UDT is much better than single stream TCP. The definite outcome is that UDT performs better at long RTTs. None of these results have a 10Gbps network example.

In this section, performance comparison of UDT vs GridFTP is presented in long-short RTT 10Gbps production networks(LONI, XSEDE). It is harder to reach the network's full capacity in 10Gbps networks and even if the network is idle and the protocol inefficiencies are overcome, there may be other types

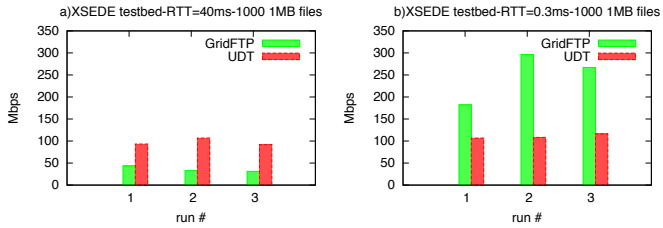

Fig. 16. UDT vs GridFTP, Transfer of large number of small files (Question 13)

of bottlenecks (e.g. CPU capacity, NIC capacity, Disk bandwidth). We first measured the effect of concurrency on UDT flows in a 10Gbps networks with an RTT of 2ms(Figure 14). UDT's single stream throughput performance was around 3Gbps. When we increased the concurrency level to 2 and 4 later, we have seen that there was not a single change in the level of total throughput. The streams simply shared 3Gbps bandwidth among themselves. In the next experiment(Figure 15), we used GridFTP with both TCP and UDT options. In the LONI network with 2ms RTT, single stream GridFTP and UDT4 showed similar performances. There was an additional overhead when we used GridFTP with UDT option. However parallel TCP of 10 streams outperformed all others and was able to reach 6 Gbps throughput. In XSEDE tests where RTT is around 30ms, the results changed and UDT4 outperformed all others reaching 4Gbps throughput followed by GridFTP-10streams. In Figure 16, the results of transfer of 1000 1MB files on XSEDE network with long and short RTTs are presented. According to the results, UDT performed poorly in short RTTs, however it can outperform single stream TCP in long RTTs.

These results confirmed that it is better to use UDT in long RTT networks without additional parallelism but it performs worse in metropolitan or state-wide networks such as LONI. Parallel TCP can compete with UDT in both cases however it is important to set the correct parallelism level without overwhelming the network.

3 Rules of Thumb

This section presents some rules that should be applied when modelling algorithms for transfer of large datasets with GridFTP pipelining, parallelism and concurrency:

- Always use pipelining, even if it has very little effect on throughput. it allows the usage of a single data channel for sending multiple files, resulting a continuous increase in number of bytes sent/received in one RTT. It also overlaps control channel messages and processing overhead with data channel transfers resulting in removal of idle time between consecutive transfers.
- Set different pipelining levels by dividing the data set into chunks where mean file size is less than BDP. The number of bytes sent/received in one RTT cannot pass the average file size multiplied by the pipelining level. Pipelining can have a huge effect as long as this value is less than BDP.
- Keep the chunks as big as possible. It is important to have enough data in a chunk for pipelining to be effective because different pipelining level transfers go through the same slow-start phase.
- Use only concurrency with pipelining for small file sizes and small number of files. Dividing a small file further with parallelism affects throughput adversely.
- Add parallelism to concurrency and pipelining for bigger file sizes where parallelism does not affect pipelining.
- Use parallelism when the number of files is insufficient to apply concurrency.
- Use UDT for wide area transfers only, preferably with only a single stream. In cases where you are allowed parallel stream transfers, TCP with optimal stream number can compete with UDT and sometimes outperform it.

4 Algorithms

Considering the rules described above, two complementary algorithms are presented to set optimal values for pipelining, parallelism and concurrency.

The first algorithm uses an adaptive approach and tries to reach the maximum network bandwidth gradually. The second algorithm follows a more aggressive approach in using concurrency. The details of the algorithms are presented in the following sections.

4.1 Adaptive PCP Algorithm

This algorithm sorts the dataset based on the file size and divides it into 2 sets; the first set (Set_1) containing files with sizes less than BDP and the second set (Set_2) containing files with sizes greater than BDP. Since setting different pipelining level is effective for file sizes less than BDP (Rule 2), we apply a recursive chunk division algorithm to the first set which is outlined in the following subsection. For the second set we set a static pipelining level of 2.

4.1.1 Recursive Chunk Division for Optimal Pipelining

This algorithm is mean-based to construct clusters of files, with each cluster (chunk) having a different optimal pipelining value. The optimal pipelining level is calculated by dividing BDP to the mean file size and the data set is recursively divided by the mean file size index while several conditions are met. The first condition is that a chunk can only be divided further if its optimal pipelining is not the same as its

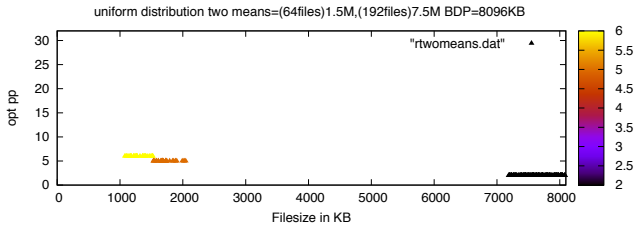

Fig. 17. Example behavior of Recursive Chunk Size Division Algorithm.

parent chunk. Secondly, a chunk cannot be less than a preset minimum chunk size and the last rule is that the optimal pipelining level set for a chunk cannot be greater than the preset maximum pipelining level. The outline is presented in Algorithm 1.

Algorithm 1 Recursive_Chunk_Division(RCD)

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 $RCD\_dividing\_the\_chunk\_by\_mean\_index$ $(start\_index- > mean\_index)$
call $RCD\_dividing\_the\_chunk\_by\_mean\_index$ $(mean\_index + 1- > start\_index)$
else
$opt\_pp = parent\_pp$
end if

Figure 17 shows an example case of clustering of dataset into chunks of different optimal pipelining levels. In this example, the dataset consists of files which are randomly generated with two different mean values of 1.5MB and 7.5MB. The y axis shows the optimal pipelining level assigned to each file while the x axis shows the file size in KB. The algorithm constructs 3 clusters to set different pipelining levels for a BDP of 8MB.

4.1.2 Adaptive Parallelism and Concurrency Levels

The PCP algorithm only uses pipelining with concurrency for files in Set_1(files smaller than BDP) and includes additional parallelism for files in Set_2(files larger than BDP) by considering rules 3 & 4 mentioned previously. File list, number of files, minimum chunk size, BDP and minimum number of chunks are given as input parameters. For Set_1, the divided chunks by the recursive chunk division algorithm go through a revision phase in which smaller chunks are combined and larger chunks are further divided so that we could apply an adaptive concurrency setting to the chunk transfers. Starting with 1 concurrency level, each chunk is transferred with exponentially increasing levels until throughput drops down and the optimal level is found. The rest of the chunks are transferred with the optimal level. 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.

Selection of an exponentially increasing parallelism level strategy is a good choice [11]. First of all, the behaviour of the throughput curve is logarithmic as we increase the parallelism level through parallel streams or concurrency. The throughput changes are much more significant when the number is small and less significant as it gets high. Exponential increase help provide a quicker ascend to the maximum throughput. Linearly increasing this number would cause further division of the data set and result in bringing extra overhead. The algorithm aims to quickly find that breaking point when the throughput drops down. We implemented more than 5% drop down in throughput to stop increasing the parallelism level to avoid for slight fluctuations in the network and end-system load and greedily reach for the maximum level of throughput.

For Set_2, previously set static pipelining level of 2, the optimal parallelism level is found before the optimal concurrency level. Chunks of files are constructed considering the following conditions: First, a chunk should be greater than the minimum chunk size multiplied by the concurrency level and second, the number of files in a chunk should be greater than the concurrency level. Starting with single parallelism level, with each chunk transfer, the parallelism level is increased exponentially until throughput drops down. After setting the optimal parallelism level, the concurrency level is increased in the subsequent chunk transfers the same way. When the optimal levels are found for pipelining-parallelism-concurrency, the rest of the dataset is transferred in one big chunk with the optimal values.

4.2 Multi-Chunk (MC) Algorithm

The Multi-Chunk (MC) [30] algorithm basically tries to improve transfer throughput of mixed datasets which consist of both small and large files (small and large sizes are defined based on network BDP). It divides the dataset into chunks based on file sizes (Small, Middle, Large and, Huge) and find optimal parameter configuration (pipelining, parallelism and concurrency) for each chunk separately. Then, it transfers multiple chunks simultaneously with their optimal parameters.

Algorithm 2 Optimal Parallelism-Concurrency-Pipelining(PCP)

Require: $list\_of\_files \vee total\_number\_of\_files \vee min\_chunk\_size \vee BDP \vee min\_no\_of\_chunks$
Sort the files
Divide the files into $Set1$ and $Set2$
FOR SET1
Create chunks by applying RCD 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 $> 0.95\times$ 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

FOR SET2
Set optimal pp as 2
$curr\_p \leftarrow 1$
$prev\_thr \leftarrow 0$
Create a chunk with the minimum chunk size
perform transfer for the consequent chunk
while Current throughtput $> 0.95\times$ Previous throughput **do**
  perform transfer for the consequent chunk
  $curr\_p = curr\_p * 2$
end while
opt p = prev p
Repeat the same steps for finding Optimal CC
Transfer the rest of the data with Optimal PP, P and CC values

In the Multi-Chunk (MC) algorithm, the focus is mainly on minimizing the effect of low transfer throughput of small files in mixed datasets. This is because the throughput obtained during the transfer of the small file chunks (even after choosing the best parameter combination) is significantly worse compared to large chunks due to the high overhead of reading too many files from disk and underutilization of the network pipe. Depending on the weight of small files' size over the total dataset size, overall throughput can be much less than the throughput of large file chunk transfers. Thus, MC transfers small chunks along with large chunks in order to minimize time period in which only small chunks are transferred.

In terms of finding optimal parameter combination, as opposed to PCP, the MC algorithm calculates value of pipelining and parallelism for a chunk at the beginning of the transfer. For the concurrency level of chunks, it treats all chunks equally and evenly distributes available channels to chunks. As concurrency means opening multiple channels (c=4 stands for 4 active TCP channels), the MC distributes data channels among chunks using round-robin on set 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. Finally, the MC algorithm does not decide the value of concurrency but determines how available channels will be assigned to chunks.

After channel distribution is completed, the MC schedules chunks concurrently using the calculated concurrency level for each 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.

Differences between Adaptive PCP and MC Algorithms:

- MC Algorithm sets a static concurrency level for the whole dataset transfer and a static parallel stream number per chunk. These numbers never change during the transfer. PCP Algorithm starts with a minimum parallel stream and concurrency level and these numbers dynamically go up until they reach the optimum.
- MC Algorithm can do multiple chunk transfers at the same time, while PCP Algorithm transfers chunks one by one.
- MC Algorithm starts with an aggressive predefined concurrency and parallelism value, while PCP adaptively converges to the optimum.

TABLE 2
BASELINE AVERAGE DISK WRITE THROUGHPUT RESULTS

Machine	Site	NIC	Mbps
eric	LONI	10G	1538.4
eric (2 proc.)	LONI	10G	2263.6
eric (4 proc.)	LONI	10G	2675.0
oliver	LONI	10G	1397.1
Hotel	FutureGrid	1G	7824
Sierra	FutureGrid	1G	315.8
Gordon	Xsede/SDSC	10G	3260.8
Stampede	Xsede/TACC	10G	3980.8
nodeX	Emulab	1G	575.2
c3.8xlarge node	AWS	10G	927.84

5 Experimental Results

The algorithms were tested on real high-speed networking testbeds FutureGrid and Xsede and also cloud networks by using Amazon Web Services

EC2 instances. Three different datasets were used in the tests classified as small(between 512KB-8MB), medium(25MB-100MB) and large(512MB-2GB) file sizes. Total dataset size was around 10GB-20GB. The baseline disk write throughput results measured with Bonnie++ and dd for every testbed used in the experiments are presented in Table 2 for comparison purposes.

FutureGrid is a wide area cloud computing testbed with a 10 Gbps network backbone. However it supports only 1 Gbps network interfaces on end-systems so the throughput is bounded by the network interface. Xsede previously known as TeraGrid is also a 10Gbps wide area testbed which is bounded by disk and CPU performance of data transfer nodes. Amazon Web Services (AWS) is a commercial cloud computing services platform that provides a variety of machine instances with different interconnects. On Futuregrid, two clusters Sierra and Hotel which are connected with 76ms RTT were used. On Xsede, Stampede and Gordon were used which are connected with a 50ms RTT. For AWS two compute optimised Linux Ubuntu instances with 10G interconnects were used in the experiments.

5.1 Real Testbed Results

Figure 18.a,b,c shows the results of the algorithms running on FutureGrid testbed. For the small dataset (Figure 18.a), Globus Online(GO) and UDT perform very poorly. This shows that these tools are not designed for datasets consisting of many small files. PCP algorithm throughput divides the dataset in to 5 chunks and then sets different pipelining levels for each and gradually increases the concurrency level. The final chunk size which is about 8GB results in the highest transfer speed. On the other hand the MC algorithm which is more aggressive in setting concurrency levels outperforms the others in terms of the average throughput.

For the medium dataset (Figure 18.b), GO keeps up to the MC algorithm throughput by setting static *pp*, *p* and *cc* values. The average throughput of the PCP algorithm follows them and UDT performs the worst. The final chunk of the PCP algorithm which is around 10GB is transferred at the same speed as the MC and GO speeds. The PCP algorithm gradually increases the parallelism level until it no longer increases the throughput. Then it starts increasing the concurrency level with each chunk transfer.

For the large dataset (Figure 18.c), GO and MC average throughput saturates the network bandwidth and UDT performs better. PCPs last two chunks (12GB)reaches the network bandwidth limit. The maximum throughput that can be achieved is bound by the network interface rather than the disk throughput(Table 2). These results show that our algorithms can reach maximum limit regardless of the dataset characteristics while GO and UDT are only good for relatively large files.

Figure 18.d,e,f presents the experimental results of the algorithms on Xsede network. The same dataset characteristics are used for the tests which are run between SDSC's Gordon and TACC's Stampede clusters. For the small data set (Figure 18.d) of which file size range is between 512KB and 8MB, MC and PCP algorithms perform the best. The worst results are seen with GO while UDT overperforms it. The last chunk transferred with PCP can adaptively reach 3500Mbps throughput. For the middle dataset the dataset (Figure 18.e) is divided into two. The first set increases the concurrency level while the second set adds parallelism. Again MC algorithm which uses concurrency aggressively performs the best while PCP adaptively learns which concurrency level is best. UDT and GO performs worse. The last chunk transferred with PCP can go beyond 4000Mbps throughput. For the large dataset (Figure 18.f), PCP sets the pipelining level to 2 and applies an adaptive parallelism and concurrency. The last chunk throughput can reach 4500 Mbps. Again MC and PCP algorithms are the best and can reach maximum disk throughput of Stampede. GO outperforms UDT in this case.

5.2 Cloud Testbed Results

The cloud experiments were conducted using Amazon's EC2 service. Two cpu-optimized c3.8xlarge type nodes with 10G interconnects were launched with an artificial delay of 100ms. Although the interconnects provide 10G bandwidth, the SSD disk volumes bind the maximum achievable throughput to around 1Gbps (Table 2). For the small dataset transfers (Figure 18.g), UDT performs the worst. GO follows UDT with 390Mbps throughput. MC algorithm with a concurrency level of 32 outperforms all others. PCP adaptively reaches 850Mbps throughput with a data chunk transfer of 7GB but the average throughput of all chunks is around 500Mbps.

In the medium dataset(Figure 18.h) GO performs better than PCP average throughput. MC average throughput outperforms all others again. PCP chunk throughput gradually surpasses the others. UDT again performs the worst. For the large dataset(Figure 18.i) GO performance is worse than PCP and MC. It is interesting to see that but since we do not have any control over GO parameters, we do not know why the medium dataset GO results were better. It can be due to different set of pp,p,cc values used for different dataset sizes.

Overall the algorithms that apply our models perform better than GO and UDT in majority of the cases. While PCP algorithm adaptively tries to reach the end-to-end bandwidth, MC algorithms behaves more aggressively based on the initially set concurrency level and both are able to reach maximum achievable throughput.

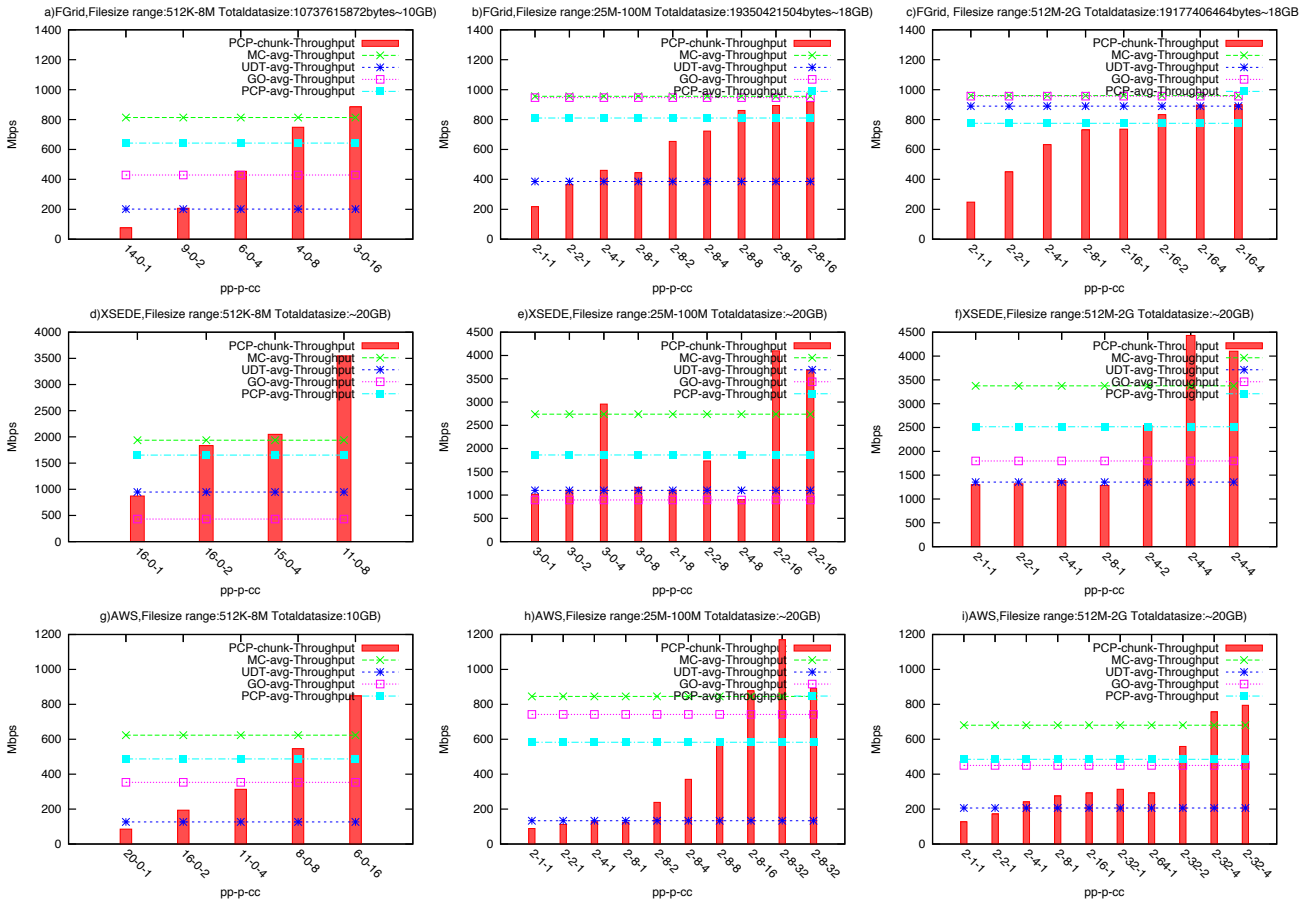

Fig. 18. Experimental Results of Algorithms on FutureGrid, XSEDE and AWS.

6 Conclusions and Future Work

Application-level transfer tuning parameters such as pipelining, parallelism and concurrency are very powerful mechanisms for overcoming data transfer bottlenecks for scientific cloud applications, however their optimal values depend on the environment in which the transfers are conducted (e.g.available bandwidth, RTT, CPU and disk speed) as well as the transfer characteristics (e.g. number of files and file size distribution). With proper models and algorithms, these parameters can be optimized automatically to gain maximum transfer speed. This study analyzes in detail the effects of these parameters on throughput of large dataset transfers with heterogenous file sizes and provides several models and guidelines. The optimization algorithms using these rules and models can provide a gradual increase to the highest throughput on inter-cloud and intra-cloud transfers. In future work, we intend to write an overhead-free implementation of a GridFTP client to reduce the overhead regarding connection start up/tear down processes for different chunk transfers.

Acknowledgment

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

References

[1] J. Bresnahan, M. Link, R. Kettimuthu, D. Fraser, and I. Foster, "Gridftp pipelining," in *TeraGrid 2007*, 2007.

[2] 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. 55:2, pp. 81–88, 2012.

[3] R. S. Prasad, M. Jain, and C. Davrolis, "Socket buffer autosizing for high-performance data transfers," *Journal of Grid Computing*, vol. 1(4), pp. 361–376, Aug. 2004.

[4] G. Hasegawa, T. Terai, T. Okamoto, and M. M, "Scalable socket buffer tuning for high-performance web servers," in *International Conference on Network Protocols(ICNP01)*, 2001, p. 281.

[5] A. Morajko, "Dynamic tuning of parallel/distributed applications," Ph.D. dissertation, Universitat Autonoma de Barcelona, 2004.

[6] T. Ito, H. Ohsaki, and M. Imase, "On parameter tuning of data transfer protocol gridftp for wide-area networks," *International Journal of Computer Science and Engineering*, vol. 2(4), pp. 177–183, Sep. 2008.

[7] K. M. Choi, E. Huh, and H. Choo, "Efficient resource management scheme of tcp buffer tuned parallel stream to optimize system performance," in *Proc. Embedded and ubiquitous computing*, Nagasaki, Japan, Dec. 2005.

[8] E. Yildirim, M. Balman, and T. Kosar, *Data-intensive Distributed Computing: Challenges and Solutions for Large-scale Information Management*. IGI-Global, 2012, ch. Data-aware Distributed Computing.

[9] T. J. Hacker, B. D. Noble, and B. D. Atley, "The end-to-end performance effects of parallel tcp sockets on a lossy wide area network," in *Proc. IEEE International Symposium on Parallel and Distributed Processing(IPDPS'02)*, 2002, pp. 434–443.

[10] D. Lu, Y. Qiao, P. A. Dinda, and F. E. Bustamante, "Modeling and taming parallel tcp on the wide area network," in *Proc. IEEE International Symposium on Parallel and Distributed Processing (IPDPS'05)*, Apr. 2005, p. 68b.

[11] 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(12), 2011.

[12] D. Yin, E. Yildirim, and T. Kosar, "A data thoughput prediction and optimization service for widely distributed many-task computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22(6), 2011.

[13] E. Altman, D. Barman, B. Tuffin, and M. Vojnovic, "Parallel tcp sockets: Simple model, throughput and validation," in *Proc. IEEE Conference on Computer Communications (INFOCOM'06)*, Apr. 2006, pp. 1–12.

[14] J. Crowcroft and P. Oechslin, "Differentiated end-to-end internet services using a weighted proportional fair sharing tcp," *ACM SIGCOMM Computer Communication Review*, vol. 28(3), pp. 53–69, Jul. 1998.

[15] L. Eggert, J. Heideman, and J. Touch, "Effects of ensemble tcp," *ACM Computer Communication Review*, vol. 30(1), pp. 15–29, 2000.

[16] H. Sivakumar, S. Bailey, and R. L. Grossman, "Psockets: The case for application-level network striping for data intensive applications using high speed wide area networks," in *Proc. IEEE Super Computing Conference (SC00)*, Texas, USA, Nov. 2000, pp. 63–63.

[17] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the tcp congestion avoidance algorithm," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 3, pp. 67–82, 1997.

[18] 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(3254-3267), 2011.

[19] E. Yildirim and T. Kosar, "End-to-end data-flow parallelism for throughput optimization in high-speed networks," *Journal of Grid Computing*, vol. 10, no. 3, pp. 395–418, 2012.

[20] T. Kosar and M. Livny, "Stork: Making data placement a first class citizen in the grid," in *Proceedings of ICDCS'04*, March 2004, pp. 342–349.

[21] T. Kosar and M. Balman, "A new paradigm: Data-aware scheduling in grid computing," *Future Generation Computing Systems*, vol. 25, no. 4, pp. 406–413, 2009.

[22] 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.

[23] (2015) Udt, udp-based data transfer. [Online]. Available: http://udt.sourceforge.net/

[24] (2015) Emulab-network emulation testbed. [Online]. Available: http://www.emulab.net/

[25] E. Yildirim, J. Kim, and T. Kosar, "Modeling throughput sampling size for a cloud-hosted data scheduling and optimization service," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1795–1807, 2013.

[26] J. Bresnahan, M. Link, R. Kettimuthu, and I. Foster, "Udt as an alternative transport protocol for gridftp," in *International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT)*. Citeseer, 2009, pp. 21–22.

[27] E. Kissel, M. Swany, and A. Brown, "Improving gridftp performance using the phoebus session layer," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 34.

[28] R. Guillier, S. Soudan, P. Primet *et al.*, "Udt and tcp without congestion control for profile pursuit," Laboratoire de Informatique du Parallelisme, Tech. Rep. inria-00367160, 2009.

[29] (2015) Bio-mirror: Biosequence and bioinformatics data. [Online]. Available: http://bio-mirror.jp.apan.net

[30] E. Arslan, B. Ross, and T. Kosar, "Dynamic protocol tuning algorithms for high performance data transfers," in *Proceedings of the 19th International Conference on Parallel Processing*, ser. Euro-Par'13, 2013, pp. 725–736.

Esma Yildirim received her B.S. degree from Fatih University and M.S. degree from Marmara University Computer Engineering Departments in Istanbul, Turkey. She worked for one year in Avrupa Software Company for the Development of ERP Software. She also worked as a Lecturer in Fatih University Vocational School until 2006. She received her Ph.D. from the Louisiana State University Computer Science Department in 2010. She has worked at the University at Buffalo (SUNY) as a researcher. She is currently an Assistant Professor at Fatih University, Istanbul, Turkey. Her research interests are data-intensive distributed computing, high performance computing, and cloud computing.

Engin Arslan received his BS degree of Computer Engineering from Bogazici University and MS degree from University at Nevada, Reno. Currently, he is pursuing his PhD of Computer Science at University at Buffalo, SUNY. He is also working as a research assistant at UB, SUNY. His research interests include data intensive distributed computing, cloud computing, and high performance networks.

Jangyoung Kim received his B.S. degree in Computer Science from Yonsei university in Seoul, Korea and M.S. degree in Computer Science and Engineering from Pennsylvania State university in University Park. He worked as a Teaching Assistant in Pennsylvania State university. Earlier, he also participated in the Programming Internship in Samsung. He received his Ph.D. in Computer Science and Engineering from the University at Buffalo (SUNY). He is currently an Assistant Professor of Computer Science in University of Suwon. His research interests are data-intensive distributed computing, cloud computing, and throughput optimization in high-speed networks.

Tevfik Kosar is an Associate Professor in the Department of Computer Science and Engineering, University at Buffalo. Prior to joining UB, Kosar was with the Center for Computation and Technology (CCT) and the Department of Computer Science at Louisiana State University. He holds a B.S. degree in Computer Engineering from Bogazici University, Istanbul, Turkey and an M.S. degree in Computer Science from Rensselaer Polytechnic Institute, Troy, NY. Dr. Kosar has received his Ph.D. in Computer Science from the University of Wisconsin-Madison. Dr. Kosar's main research interests lie in the cross-section of petascale distributed systems, eScience, Grids, Clouds, and collaborative computing with a focus on large-scale data-intensive distributed applications. He is the primary designer and developer of the Stork distributed data scheduling system, and the lead investigator of the state-wide PetaShare distributed storage network in Louisiana. Some of the awards received by Dr. Kosar include NSF CAREER Award, LSU Rainmaker Award, LSU Flagship Faculty Award, Baton Rouge Business Report's Top 40 Under 40 Award, and 1012 Corridor's Young Scientist Award.