# End-to-End Dynamic Pipelining Tuning Strategy for Small Files Transfer

Shimin Wu[1], Dawei Sun[1(✉)], Shang Gao[2], and Guangyan Zhang[3]

[1] School of Information Engineering, China University of Geosciences, Beijing 100083, People's Republic of China
{wushimin,sundaweicn}@cugb.edu.cn
[2] School of Information Technology, Deakin University, Victoria 3216, Australia
shang.gao@deakin.edu.au
[3] Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
gyzh@tsinghua.edu.cn

**Abstract.** Improving the transmission efficiency for small files over a wide area network is always challenging. Time may be wasted when waiting for transmission commands due to the design of transfer protocols, which in turn increases the Round-trip time (RTT). GridFTP is widely deployed as a transfer protocol in the grid era, where a concept of pipelining is proposed to improve the transmission efficiency for small files. Based on the GridFTP protocol, we design a smart data structure to classify files and propose a corresponding scheduling algorithm to tune the pipelining parameters, making them more reasonable and adaptive to different transmission scenarios. Bandwidth usage is optimized when a large number of small files are transferred with our strategy by combining the optimal pipelining and concurrency parameters. A method to optimizing the throughput for high-priority file transfer is also proposed. By adjusting the pipelining parameter dynamically, the throughput is increased by almost 10% compared with other methods. Moreover, our method achieves better performance even with a smaller concurrency setting. The favorable throughput is maintained when transferring high-priority files.

**Keywords:** GridFTP · Throughput optimization · Pipelining and concurrency · Lots of small files · Network protocols

## 1 Introduction

In this fast-developing era, a variety of new concepts and technologies are emerging. Massive volume of data ranging from GB to PB is generated every day, such as scientific experimental and e-commerce data. Most of the data are small files consisting of pictures and text information. For example, in BLAST [1], the bioinformatic data generated and required by experiments is usually string


W. Xiang et al. (Eds.): BROADNETS 2021, LNICST 413, pp. 20–36, 2022.
https://doi.org/10.1007/978-3-030-93479-8_2

data. The data needs to be transferred between sites before the experiments are conducted. Moreover, a research report from the Pacific Northwest National Laboratory in the United States showed that there were 12 million files in their system, 94% of which were smaller than 64 MB, and 58% were smaller than 64 KB [2]. Similar situations also exist in other systems and become more and more common. E.g. the meteorological data in a meteorological communication system [3] has the characteristics of large amount of output, diverse value types and small file sizes. With such data characteristics, the system needs to transfer them to the National Weather Information Center for further processing. There are priority or dependency relationships among these small files in some scenarios. For example, the model parameters of certain scientific experiments should be transmitted as soon as possible to start the next experiments.

When transferring different types of files, insufficient utilization of bandwidth on WAN has always been a problem under discussion. This phenomenon is more obvious when transferring data sets composed of small files like the aforementioned meteorological data. The low utilization of bandwidth might be caused by many reasons, such as the overhead of channel connection/disconnection, the transfer protocols which cannot make full use of the network bandwidth due to the time spent on waiting for the confirmation of transmission commands between two parties. Furthermore, the total idle time of a data channel will increase with the RTT (Round-Trip Time) and the number of files. For such scenarios, GridFTP supports a pipelining [4] parameter to ease the bandwidth under-utilization problem when transferring a large number of small files. However, how to find the best pipelining parameter is still a challenging problem to solve. Pipelining can neither make full use of bandwidth when the parameter value is too small, nor improve the utilization further when it is too large. Sometimes it may even consume more system resources (e.g. more storage and CPU to analyze the additional transmission commands). Moreover, the optimization of pipelining parameter often vary with the characteristics of file sets and network conditions.

The paper designs a smart data structure and an algorithm to find an optimal pipelining parameter value for different types of file sets by rearranging a transfer queue, while keeping the consumption of system resources as low as possible. The proposed algorithm not only suits for the transmission of static file sets, but also works well for the dynamic ones. The rest of the paper is organized as follows: Sect. 2 briefly introduces the background of GridFTP and related work. Section 3 describes the design and implementation of our structure and algorithm. Section 4 presents the experimental results. Section 5 concludes the paper and discusses future work.

## 2 Background and Related Work

In this section, we introduce the background knowledge of GridFTP. Furthermore, we review the typical existing methods for improving the GridFTP protocol and analyze their advantages and disadvantages.

### 2.1 GridFTP

GridFTP [5–7], as a part of the Globus Toolkit [8] project, is a high-performance, secure and reliable data transfer protocol optimized for high-bandwidth wide area networks. It is an extension to FTP [9] protocol and defined for high-performance operation and security purposes. GridFTP contains many new features such as automatic negotiation of TCP socket buffer size, third-party control of file transfer and partial file transfer, etc. GridFTP's parallelism, concurrency and pipelining [10] are three powerful parameters to improve bandwidth utilization for different scenarios.

Parallel streams have a positive impact on the transmission of a single large file. The provision of parallel streams is mainly through the establishment of multiple data channels for single file transmission, where a file is divided into multiple parts, and each data channel transmits one part. This approach can quickly transit through the slow start-up phase and make full use of network bandwidth. Parallel streams are controlled by the parallelism parameter in GridFTP. Many scholars use socket buffer size adjustment technology together with parallelism to resolve the deficiencies of TCP [11–13]. There is an optimal value for the parallelism parameter in a given scenario. An excessively large parallelism value cannot increase the transmission throughput, and sometimes has negative effects.

Concurrency supports transmission of multiple files simultaneously through the establishment of multiple control channels. Therefore, the overall utilization of bandwidth is improved. In most cases, the effect of concurrency parameter is better than that of parallelism [14]. Increasing the value of concurrency parameter can significantly improve the throughput, but the use of concurrency parameters may be constrained by resources such as CPU and file system processes.

Pipelining is mainly used to solve the problem of transferring a large number of small files. By allowing the client to have multiple unconfirmed transfer commands at once, the idle time between transmission is reduced as much as possible. Before reaching the formulated number of unconfirmed commands, the client can send transmission commands at any time, and the server processes the requests in the arrival order of the commands. Similar to the other two parameters, how to find the best pipelining parameter for different file sets and network conditions is challenging.

### 2.2 Related Work

There are many research aiming at finding an optimal value for the GridFTP parallelism parameter to improve the throughput of large file transmission over WAN. A formula was proposed in [15] to get the throughput of multiple parallel streams, followed by a series of transmission experiments conducted on WAN to evaluate how parallel streams could improve the throughput. [16] further studied the relationship between packet loss rate, RTT and parallel flow. It proposed a prediction formula for parallel flow. Based on the research of [15,16], a new

model was proposed in [17] to approximate the optimal number of parallelism with the least historical information and the lowest prediction overhead.

Although there have been some optimization studies on GridFTP parameters, most of them aimed at large file transmission scenarios. With the development of technologies, the problem of lots of small files (LOSF) transmission should also be taken into consideration. The pipelining of GridFTP is a powerful tool to improve the efficiency of small file transmission, but it has the same problem as the parallelism parameter, i.e. how to determine an optimal pipelining parameter value.

In Globus Online [18], a static method was proposed for setting pipelining parameter according to file size and quantity. If there are more than 100 files for transmission in a file set, and the average file size is less than 50 MB, the pipelining parameter is set to 20. If all files are larger than 250 MB, the pipelining parameter is 5. Otherwise, the default value is 10. This algorithm does not consider the factor that the pipelining parameter is not only related to file characteristics, but also related to network characteristics. Too large a pipelining parameter value may have a negative impact on throughput, and even waste system resources.

Many scholars have conducted further research to explore the combined use of the three parameters. [19] proposed a model that adjusted the values of the three parameters based on historical information. [20] found that when transferring LOSF, pipelining did not work well with parallelism, because the implementation of parallelism transmission is based on the establishment of multiple TCP transmission channels, which will further reduce the size of the transmitted file and exacerbate the problem of bandwidth under-utilization. Therefore, for LOSF transmission scenarios, better to use the pipelining parameter and concurrency parameter together. Based on this finding, a Recursive Chunk Division (RCD) algorithm was proposed in [20] to calculate the best pipelining parameter value. The advantage of this algorithm is that the pipelining parameter value can be obtained according to different network conditions and file set characteristics. However, this algorithm could easily reach the maximum pipelining parameter value (e.g. 20) when processing KB-level files, and it will take up a lot of resources when the number of server requests increases. In some cases, even setting with the maximum pipelining parameter value, the file clusters could not take up most of the link bandwidth, resulting in a waste of bandwidth.

## 3 Optimal Pipelining Adjustment

When transmitting LOSF, the file composition of different file sets might be different. For example, some may have files of similar sizes, while others might have high variance in file size. So how to deal with file sets with different characteristics and determine the best pipelining parameter value for them is difficult. In this section, we design a smart data structure to address this problem, and propose an algorithm to determine an optimal pipelining parameter value based on this structure.

### 3.1 Pipelining Calculation

File size is the main factor to consider when setting an optimal pipelining value, especially for long RTT networks. Regardless of the file size, transmission at different pipeline levels will experience a similar slow start phase. The key point is whether each transmission after the slow start phase can reach the maximum number of bytes under current network condition. If the file size is larger than the BDP [21] (Bandwidth-Delay Product), the bandwidth of each transmission can be fully utilized. If the file size is smaller than the BDP, the link might not be fully used, or there will be under-utilization of bandwidth. With an optimal pipelining parameter value, each transmission will more likely reach the maximum link bandwidth to reduce any waste and increase the throughput. However, under special circumstances, if there are a large number of extremely small files, even if the maximum pipelining value is adopted, the link still cannot be fully utilized. For this kind of scenarios, this paper designs a new data structure, called bucket structure, to store files with different characteristics. On top of it, a Bucket Divide (BD) algorithm is proposed to tune the optimal pipelining parameter value and rearrange the file transmission queue according to that value.

The original intention of bucket design is to reduce the time complexity of sorting files. Similar to the RCD algorithm, accurate sorting based on file size is required for getting an optimal pipelining parameter. The buckets are used to hold different sized files. If the file sorting is not desirable, the files combined from the buckets for transmission may not match the target BDP well, resulting low bandwidth utilization. To reduce the influence of imprecise sorting, we increase the denominator of the pipelining parameter calculation. The pp (pipelining parameter) value calculated in formula (1) is an initial value. It will be fine tuned when the files in the bucket structure meet certain conditions.

The pp calculation formula is listed as formula (1):

$$pp = 3 * BDP/MF - 1 \tag{1}$$

where the BDP is Bandwidth-Delay Product and MF is the mean file size of the files smaller than BDP. One reason why the denominator is set to triple BDP is to reduce the effect of phenomenon when the size of file combination cannot match the BDP well. Secondly, we hope that the transmission can be stabilized over the slow start phase, and the TCP stream needs to transmit about three times [22] BDP to fill the channel. The difference between our formulation and RCD algorithm is the RCD algorithm divides the files of similar sizes into a file cluster and calculates the best pp value based on this cluster, while our formulation uses all the files smaller than BDP in the bucket structure (to be introduced later in detail) to calculate the pp. If a file set contains some files whose sizes are close to the BDP, the MF can be increased significantly. The calculated pp is smaller than that of the RCD algorithm, giving us an opportunity to increase the denominator and improve transmission performance.

### 3.2 File Classification

The design of the new data structure mainly adopts the idea of bucketing used to store files by size. When the number of buckets increases, the file sizes in each bucket become more closer. Allocating files to these buckets is like sorting the file set. To improve the bandwidth utilization, we can deliberately select files from these buckets and make the combined file size as close as possible to the BDP. But the drawback is using a large number of buckets for sorting all files of the file set will increase time complexity. We need to use less buckets to decrease time complexity. The algorithms discussed in Subsect. 3.3 are proposed to address this problem: first allocating files into different sized buckets, then selecting files from proper buckets and creating file combination with the right size.

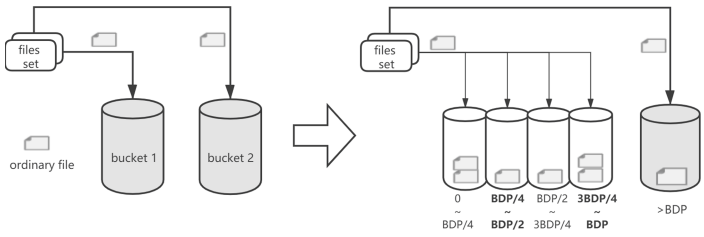


**Fig. 1.** The process of ordinary file classification

As shown in Fig. 1, bucket 2 is used to load files larger than BDP, while bucket 1 keeps the files smaller than BDP. All the files stored in bucket 1 are small files, which are the main factor to derive the best pp value. We further divide bucket 1 into multiple smaller buckets. One thing to note is that the number of smaller buckets is to be carefully selected as it impacts on the size accuracy of the file combination. The larger number of buckets is adopted, the more accurate size of file combination we have. In extreme cases, all the same sized files are allocated to one bucket. The fewer the number of buckets, the more files placed in each bucket. This results in greater size difference in a bucket and larger size deviation to BDP for the file combination being created by selecting files from different buckets. Since the files in each bucket are randomly placed and not ordered purposely, this randomness lowers the probability of selecting multiple smallest files for one pp transmission. Moreover, the performance improvement brought by a large number of buckets is not very outstanding. We tested the performance with different bucket numbers and found that when the number of buckets exceed 10, there was almost no improvement. We recommend that 4–10 buckets is normally a good choice. The following analysis and description will use 4 buckets as an example.

As shown in Fig. 1, bucket 1 is equally divided into 4 small buckets, represented by No.1–No.4. The size ranges of files stored in each bucket are 0–BDP/4, BDP/4–BDP/2, BDP/2–3BDP/4 and 3BDP/4–BDP, respectively. Bucket 2 (No. 5) is used

to store files larger than BDP. The files in the 4 small buckets (No. 1–4) are mainly used to calculate the pp value. The files in the bucket 2 (No. 5) is to supplement the file combination, offsetting the gap between the pipelining files and the BDP, and increasing the pipelining transmission bandwidth.

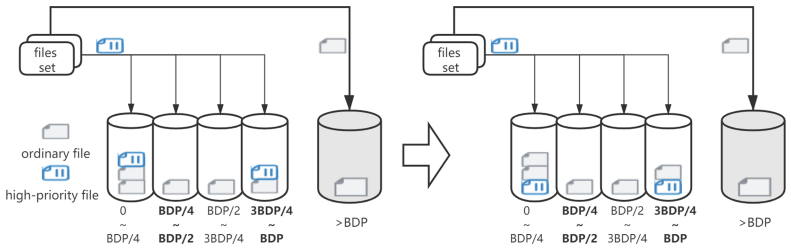


**Fig. 2.** The process of high-priority file classification

For a file set including high-priority files, we rearrange the file transfer queue to improve the throughput and deliver them to the destination as fast as possible. As files in each bucket are not sorted, it is possible that the high-priority files are placed at the end of the bucket file transmission queue. Although the transmission of a small file does not take too much time, given a large number of small files are to be transmitted, the files placed at the end of transmission queue may still take a considerately long time to be delivered. As such, the transmission order is important for the high-priority files. If we simply insert the high-priority files to the front of the queue without specific ordering, their transmission might not consume the ideal bandwidth, which in turn reduces the total throughput.

To reduce the potential bandwidth waste when transferring large number of high-priority files, we cannot simply put them at the front of the queue. The best arrangement should be transferring them as soon as possible without significant decrease of bandwidth utilization. To do that, we arrange the high-priority files to the No. 1, No. 4 and No. 5 buckets because the files transmission priority of these buckets are set higher. Furthermore, we put the files to the front of these bucket queues. As shown in Fig. 2, the file replacement operation is performed after the files are sorted into buckets. If there is a non-prioritized ordinary file in front of the high-priority file sorted into No.1 bucket, that ordinary file will be swapped with the high-priority file. The right side of Fig. 2 shows the status after the replacement.

To ensure the pipelining transmission size as close to the BDP as possible, each transmission chunk may mix several ordinary files. The mixing of several non-prioritized ordinary files will not consume too much time. The transmission speed of high-priority files is still improved without notable total throughput drop.

### 3.3 File Queue Rearrangement

Our Bucket Divide (BD) has 2 steps: File Bucket Divide (Algorithm 1 FBD) and Transmission Queue (Algorithm 2 TQ). The proposed bucket structure is used to support file classification from line 1 to 13, as shown in Algorithm 1. After the file classification, the files are put into corresponding buckets by size.

The OPTIMALPP function is used to calculate the pp value based on the status of $bucketNum[0]$ to $bucketNum[4]$ (e.g. No. 1–5 bucket), as shown from lines 16 to 23 of OPTIMALPP function. The variable *times* in OPTIMALPP represents the number of rounds that the files in $bucketNum[4]$ (e.g. No. 5 bucket) can be transferred for a *pp* value. $N$ is the total file number in $bucketNum[4]$. The *pp* value will be fine tuned if $N$ and *time* meet certain conditions, which will be discussed in Algorithm 2.

**Algorithm 1.** File Bucket Divide(FBD)

**Input:** *file* list, $bucketNum[0] \rightarrow bucketNum[4]$, $BDP$
**Output:** $bucketNum[0] \rightarrow bucketNum[4]$, $N$, *pp*, *times*

1: **while** *file* list **do**
2: **if** $0 < file[i].size <= BDP/4$ **then**
3: $bucketNum[0] \leftarrow file[i]$
4: **else if** $BDP/4 < file[i].size <= BDP/2$ **then**
5: $bucketNum[1] \leftarrow file[i]$
6: **else if** $BDP/2 < file[i].size <= 3BDP/4$ **then**
7: $bucketNum[2] \leftarrow file[i]$
8: **else if** $3BDP/4 < file[i].size <= BDP$ **then**
9: $bucketNum[3] \leftarrow file[i]$
10: **else** $file[i].size > BDP$
11: $bucketNum[4] \leftarrow file[i]$
12: **end if**
13: **end while**
14: OPTIMALPP ($bucketNum[0] \rightarrow bucketNum[4]$)
15:
16: **function** OPTIMALPP($bucketNum[0] \rightarrow bucketNum[4]$)
17: Calculate $Nums \leftarrow$the number of files except for $bucketNum[4]$
18: Calculate $meanFileSize \leftarrow$ mean file size except for $bucketNum[4]$
19: Calculate $N \leftarrow$ the number of files in $bucketNum[4]$
20: Calculate $pp \leftarrow [3BDP/meanFileSize] - 1$
21: Calculate $times \leftarrow Nums/(pp + 1)$
22: **return** $N$, *pp*, *times*
23: **end function**

The Algorithm 2 Transmission Queue (TQ) aims to establish a file transmission queue by taking out files from buckets selectively, so that the pp transmission can match the BDP as close as possible. The rule to take out files is determined by *times*, *pp* and $N$ from Algorithm 1.

As shown in Algorithm 2, the value of maxPP is set to 20 (an empirical setting). Given $N$ is the number of files contained in $bucketNum[4]$, when the value of "times" is greater than $N$, it means that the files in the $bucketNum[4]$ are not enough to make the pipelining transmission match BDP optimally. In this case, we perform the function EQUALTRANSFER in Algorithm 2, which uses the large files in $bucketNum[4]$ to consume small files as much as possible for decreasing the effect of small files in subsequent transmission and increase the probability of the remaining file combination matching the BDP.

**Algorithm 2.** Transmission Queue(TQ)

**Input:** $bucketNum[0] \rightarrow bucketNum[4]$, $maxPP$, $N$, $pp$, $times$
**Output:** $queue$

**if** $times > N$ **then**
    EQUALTRANSFER($bucketNum[0] \rightarrow bucketNum[4], pp$)
**else**
    $newPP \leftarrow pp + [N/times]$
    **if** $newPP < maxPP$ **then**
        INEQUALTRANSFER($bucketNum[0] \rightarrow bucketNum[4], newPP, pp$)
    **else**
        $newPP \leftarrow maxPP$
        INEQUALTRANSFER($bucketNum[0] \rightarrow bucketNum[4], newPP, pp$)
    **end if**
**end if**
**function** EQUALTRANSFER($bucketNum[0] \rightarrow bucketNum[4], pp$)
    initialize an empty $queue$
    $left = 0$, $right = 3$ and $big = 4$
    **while** $bucketNum[0] \rightarrow bucketNum[4]$ is not empty **do**
        **if** the file number is zero in bucketNum[left] or bucketNum[right] **then**
            $left++$ or $right--$
        **end if**
        **if** $bucketNum[big]$ is not empty **then**
            $queue \leftarrow pp$ number of files from bucketNum[left]
            $queue \leftarrow$ one file from bucketNum[big]
        **else if** $(pp + 1)$ is even **then**
            $queue \leftarrow (pp + 1)/2$ number of files from bucketNum[left]
            $queue \leftarrow (pp + 1)/2$ number of files from bucketNum[right]
        **else if** $(pp + 2) <= maxPP$ **then**
            $queue \leftarrow (pp + 2)/2$ number of files from bucketNum[left]
            $queue \leftarrow (pp + 2)/2$ number of files from bucketNum[right]
        **else**
            $queue \leftarrow (pp + 1)/2$ number of files from bucketNum[left]
            $queue \leftarrow [(pp + 1)/2] + 1$ number of files from bucketNum[right]
        **end if**

remove the files in queue from bucketNum[left-big]
**end while**
**return** *queue*
**end function**
**function** INEQUALTRANSFER($bucketNum[0] \rightarrow bucketNum[4], newPP, pp$)
initialize an empty *queue*
$left = 0$, $right = 3$, $big = 4$
**while** $bucketNum[0] \rightarrow bucketNum[4]$ is not empty **do**
**if** the file number is zero in bucketNum[left] or bucketNum[right] **then**
$left++$ or $right--$
**end if**
**if** $(pp + 1)$ is even **then**
$queue \leftarrow (pp + 1)/2$ number of files from bucketNum[left]
$queue \leftarrow (pp + 1)/2$ number of files from bucketNum[right]
$queue \leftarrow newPP - pp$ number of files from bucketNum[big]
**else**
$queue \leftarrow [(pp + 1)/2] + 1$ number of files from bucketNum[left]
$queue \leftarrow (pp + 1)/2$ number of files from bucketNum[right]
$queue \leftarrow newPP - pp$ number of files from bucketNum[big]
**end if**
remove the files in queue from bucketNum[left-big]
**end while**
**return** *queue*
**end function**

If the value of *times* is less than or equal to $N$, the function INEQUALTRANSFER in Algorithm 2 is called. It indicates that the number of files in $bucketNum[4]$ is sufficient to support all the pipelining transmissions and the files in $bucketNum[4]$ can be used as supplementary to help the transmission size match BDP. In order to ensure the transmission reaches the maximum link bandwidth, the *pp* should be changed to $newPP$ in this situation.

The detailed arrangement of transmission queue is shown in line 12–35 of EQUALTRANSFER function and line 36–55 of INEQUALTRANSFER function. The line 16 and 40 check if there are no more files in current bucket, the adjacent non-empty bucket's files are transferred. In EQUALTRANSFER function, if the file number in $bucketNum[4]$ is not large enough to participate pp transfer from beginning to end, the pp is not changed, and the queue arrangement is as shown from line 19–31. In INEQUALTRANSFER function, if $bucketNum[4]$ contains many files larger than BDP, the transmission queue can make full use of the files in $bucketNum[4]$ from beginning to end. The queue arrangement is shown from line 43 to 51. With the TQ algorithm, we get a transmission queue that makes each pp transfer match BDP as much as possible, which in turn improves the throughput.

For high-priority file transmission, the first thing is to ensure they are transmitted as soon as possible. The second thing is the throughput should not

decrease drastically. The first answer can be found based on high-priority file's size. If the file size is less than or equals to BDP/2, it should be put into the $bucketNum[0]$ (e.g. No. 1 bucket). If the file size is in range BDP/2 and BDP, it should be put into the $bucketNum[3]$ (e.g. No. 4 bucket), the other files are put into $bucketNum[4]$ (e.g. No. 5 bucket). The purpose of this classification is to place the high-priority files to the front of the transmission queue in the buckets. Another purpose is to ensure the size of the pipelining file combination is as close as possible to BDP. When the number of high-priority files in buckets No. 1, No. 4 and No. 5 are similar, it can be guaranteed that the high-priority files will be transmitted before all the other files. In some extreme cases, the files in No.1 bucket are all high-priority files, and there are no high-priority files in No. 4 or No. 5 bucket. In this case, the transmission time of the last high-priority file may be delayed. But when the high-priority files are evenly distributed, our algorithm achieves good performance.

## 4 Experiment

### 4.1 Experiment Environment

The experimental environment consists of three servers running CentOS7 operating system and xfs file system. One server works as the GridFTP server and the other two work as client1 and client2. The bandwidth bottleneck is 100 Mbps from the server to the clients. The RTT is 50 ms from client1 to the server and 100 ms from client2 to the server. The buffer size is set to BDP in all experiments.

The throughput with different file sets are tested on this two-clients/server network. We also test the performance when transferring small file sets with simple pipelining settings. The pp is set to static value 2 as the baseline experiment, called DEFAULT when referred to. Furthermore, the RCD algorithm mentioned above is implemented and its throughput is tested in the same environment as our BD algorithm and DEFAULT. Simultaneously, we test the performance of transferring high-priority files and the impact brought by the combination of pipelining parameter and concurrency parameter, followed by the comparison of the mentioned algorithms and analysis of the outcomes produced by different algorithms.

### 4.2 Performance Analysis

As shown in Fig. 3, the experiments are carried out under different RTT conditions (50 ms and 100 ms). Given the max bandwidth is 100 Mbps, the test file set consists of 500 0 KB–100 KB files and 500 1 MB–2 MB files that are generated randomly. It can be found that the throughput of different methods (Default, BD and RCD) when the RTT is 50 ms are all higher than those when the RTT is 100 ms in Fig. 3. The reason behind is simple: the BDP increases accordingly with the increase of RTT. The probability of bandwidth loss will increase when the KB files are transmitted with the DEFAULT method and the RCD method.

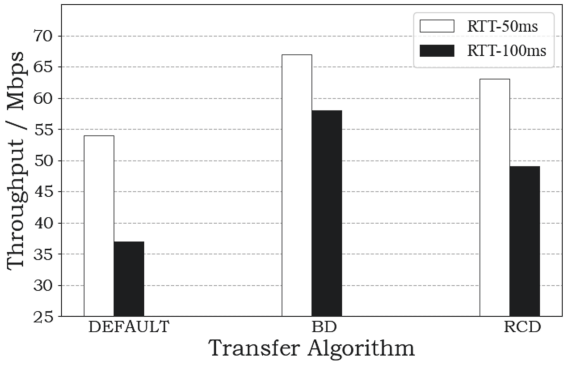


**Fig. 3.** The impact of different RTT

This leads to a decrease of the overall throughput. The BD algorithm, through the rearrangement of file orders, makes the transferred file size under a pp value match the BDP as close as possible. However, the throughput of BD decreases because the difficulty of making the pp transmission better match the BDP size also rises when the BDP increases. But compared to other methods, it only reduces the total throughput at about 8%, while the DEFAULT and RCD reduce about 17% and 14%, respectively. When the RTT is 50 ms, the throughput gap between BD and RCD is not too large. The reason is that RCD can fill up the link easily by using the maximum pp value at the early stage under 50 ms RTT. However, the ability of RCD to process extremely small files is weakened with the increase of BDP, so the throughput of RCD drops significantly when RTT increases to 100 ms.

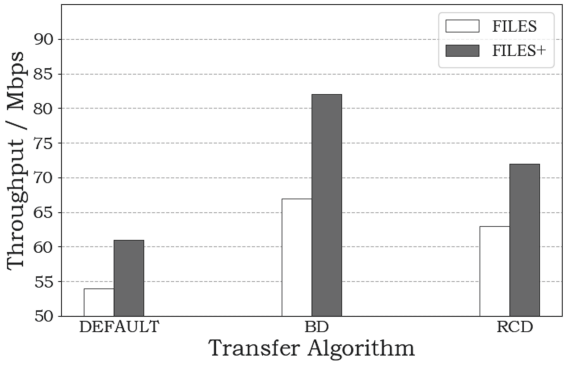


**Fig. 4.** The impact on throughput by different file sizes

On the basis of the above experiments, we explore the impact on transmission throughput when file sizes are larger than BDP. The condition FILES set in Fig. 4 includes 500 0 KB–100 KB files and 500 1 MB–2 MB files that are generated randomly. FILES+ set adds additional randomly generated 500 2 MB–4 MB files on top of FILES. The RTT is 50 ms. From the result shown in Fig. 4, it can be seen that the throughput of FILES+ all perform better than those under condition FILES set. This is because the additional 500 files are all larger than BDP, which increases the maximum bandwidth transmission time and improves the overall throughput. It can be seen that the BD algorithm improves the throughput more significantly than the other methods. It is because the BD algorithm uses all the additional files larger than BDP to supplement the transfer of small files, reducing the impact of pipelining transmission not matching the BDP, thereby increasing the overall throughput. RCD and DEFAULT algorithm only extend the maximum bandwidth transfer time under FILES+ compared to those under FILES transfer.

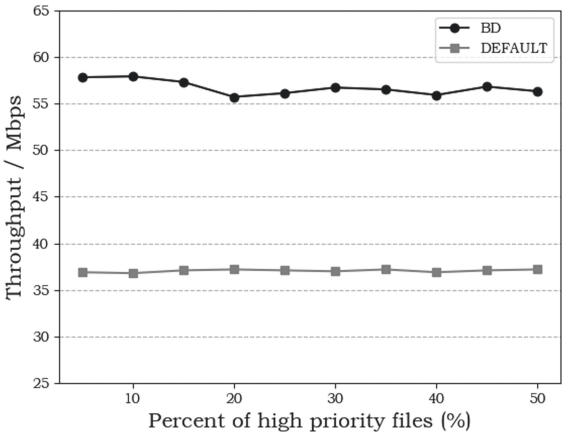


**Fig. 5.** The impact on throughput when transferring high-priority files

To explore the performance of the BD algorithm when transferring high-priority files, we randomly increase the number of high-priority files based on the condition FILES until it reaches half of the total file number. The RTT is 100 ms. From Fig. 5, it can be found that with the increase of number of high-priority files, the overall throughput fluctuates within 3% range. Even if the number continues to increase, the overall throughput will not drop significantly. The main reason for this phenomenon is that we force the order arrangement for high-priority files and put them into the side buckets (e.g. bucket No.1, No. 4 and No. 5) to make them first transferred before other bucket files. Moving these high-priority files from the middle buckets (e.g. bucket No. 2 and No. 3) to the side buckets makes the size of the high-priority file combination match BDP easily, but this file movement will also increase the size variance in the side buckets,

making it harder to match BDP for the immediate following combinations, which in turn causes a slight throughput loss.

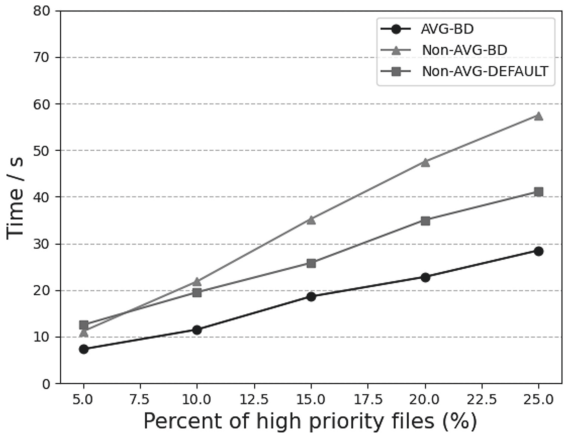


**Fig. 6.** The transmission time of high-priority files

As shown in Fig. 6, we increase the percentage of high-priority files based on the condition FILES when RTT is 100 ms. AVG-BD represents the case where the sizes of high-priority files are evenly distributed among the overall 1000 files, and we use BD algorithm to transfer the file set. Non-AVG-BD represents that the sizes of high-priority files are only within the range of 500 0 KB–100 KB. When the size distribution of high-priority files is even, AVG-BD performs best. For Non-AVG-BD, its high-priority file transmission time is longer than that of the Non-AVG-DEFAULT. The reason is with an uneven file size distribution, the BD algorithm will have to select a large number of ordinary files to meet the BDP requirement, leaving the high-priority files waiting in the queue for preferred file combination, and affecting their transmission order. Under an even file size distribution, the BD algorithm is able to improve the throughput and transmit the high-priority files promptly.

No comparison to RCD is provided here is because RCD does not consider the high-priority files in a file set. After the file set is split by RCD, the high-priority files are distributed to each file cluster and their delivery priority demands are no longer supported.

As the implementation of parallelism has a negative impact on the throughput when processing small file transmission, we test the performance of combining the concurrency parameter (cc) and pipelining parameter together. As shown in Fig. 7, when RTT is 50 ms, it can be seen that the cc has a very obvious impact on the throughput, but compared to the DEFAULT and RCD algorithm, the BD algorithm has a stable throughput when the concurrency parameter is set to 2. When the cc increases to 4 and 6, the improvement on throughput is minimal. The DEFAULT method with concurrency value 6 can only reach almost identical

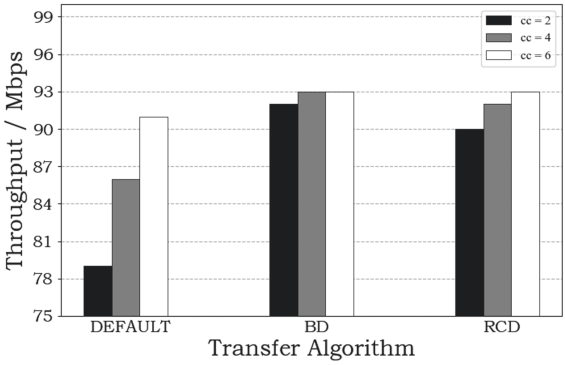


**Fig. 7.** The state of throughput under different cc values

throughput to the BD algorithm with concurrency value of 2. The BD algorithm requires a smaller value of cc than the other two algorithms in most situations and saves system resources because of this characteristic.

From the above experiments, it can be found that the throughput of the BD algorithm is better than those of the DEFAULT and RCD when transmitting small files, regardless of whether the pipelining parameter is used alone or in combination with the concurrent parameter. With the increase of transfer delay time and number of large files, the performance of BD algorithm is further improved without notable increase of resource assumption.

## 5 Conclusion and Future Work

As a high-performance transmission protocol, GridFTP provides three parameters for transmission optimization. In different file transmission scenarios, the optimal parameter values are not the same. If the parameters are not set properly, the improvement on the throughput will be compromised, and sometimes may even have negative effects. In this paper, a smart bucket structure is designed for GridFTP. It is mainly used to perform file classification operation based on their size. On top of the bucket structure, a BD algorithm is proposed to conduct corresponding bucket division and queue arrangement operations on files. With the bucket structure and the BD algorithm, we can not only find optimal pipelining parameter values, but also optimize the transmission throughput for LOSF with low resource consumption. Furthermore, more than 90% of the total throughput can be easily achieved when our BD algorithm is used with the concurrency parameter.

In terms of future work, the following points are worth further exploring: the first is to refine the model to make the rearrangement of various special files more accurate; the second is to update the parameters for continuous influx of files in real time.

**Acknowledgement.** This work is supported by the National key R&D Program of China under Grant 2018YFB0203902, the National Natural Science Foundation of China under Grant No. 61972364; and the Fundamental Research Funds for the Central Universities under Grant No. 2652021001.

## References

1. Altschul, S.F., Gish, W., Miller, W., et al.: Basic local alignment search tool. Journal of molecular biology **215**(3), 403–410 (1990)
2. Mackey, G., Sehrish, S., Wang, J.: Improving metadata management for small files in HDFS. In: IEEE International Conference on Cluster Computing and Workshops, IEEE, pp. 1–4 (2009)
3. Wang, F.: WMO information system: Beijing global information system center. Bull. Am. Meteorol. Soc. **94**(7), 991–994 (2013)
4. Bresnahan, J., Link, M., Kettimuthu, R., et al.: Gridftp pipelining. In: Proceedings of the 2007 TeraGrid Conference (2007)
5. Allcock, W.: GridFTP: protocol extensions to FTP for the Grid. http://www.ggf.org/documents/GFD.20.pdf(2003)
6. Bresnahan, J., Link, M., Khanna, G., et al.: Globus GridFTP: what's new. In: Proceedings of the First International Conference on Networks for Grid Applications, pp. 1–5 (2007)
7. Allcock, W., Bresnahan, J., Kettimuthu, R., et al.: The globus striped GridFTP framework and server. In: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, SC 2005. IEEE, pp. 54–54 (2005)
8. Foster, I.: Globus toolkit version 4: software for service-oriented systems. J. Comput. Sci. Technol. **21**(4), 513–520 (2006)
9. Postel, J., Reynolds, J.: File transfer protocol (1985)
10. Liu, Y., Liu, Z., Kettimuthu, R., et al.: Data transfer between scientific facilities–bottleneck analysis, insights and optimizations. In: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 122–131. IEEE (2019)
11. Ito, T., Ohsaki, H., Imase, M.: On parameter tuning of data transfer protocol GridFTP for wide-area networks. Connections **3**, 9 (2008)
12. Choi, K.M., Huh, E.-N., Choo, H.: Efficient resource management scheme of TCP buffer tuned parallel stream to optimize system performance. In: Enokido, T., Yan, L., Xiao, B., Kim, D., Dai, Y., Yang, L.T. (eds.) EUC 2005. LNCS, vol. 3823, pp. 683–692. Springer, Heidelberg (2005). https://doi.org/10.1007/11596042_71
13. Data Intensive Distributed Computing: Challenges and Solutions for Large-scale Information Management: Challenges and Solutions for Large-scale Information Management. IGI Global, Hershey (2012)
14. Kosar, T., Balman, M., Yildirim, E., et al.: Stork data scheduler: mitigating the data bottleneck in e-science. Phil. Trans. R. Soc. Math. Phys. Eng. Sci. **2011**(369), 3254–3267 (1949)
15. Hacker, T.J., Athey, B.D., Noble, B.: The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network. In: Proceedings 16th International Parallel and Distributed Processing Symposium, 10p. IEEE (2002)
16. Lu, D., Qiao, Y., Dinda, P.A., et al.: Modeling and taming parallel TCP on the wide area network. In: 19th IEEE International Parallel and Distributed Processing Symposium, 10 p. IEEE (2005)

17. Yildirim, E., Balman, M., Kosar, T.: Dynamically tuning level of parallelism in wide area data transfers. In: Proceedings of the 2008 International Workshop on Data-Aware Distributed Computing, pp. 39–48 (2008)
18. Allen, B., Bresnahan, J., Childers, L., et al.: Software as a service for data scientists. Commun. ACM **55**(2), 81–88 (2012)
19. Kim, J.: Tuning GridFTP pipelining, concurrency and parallelism based on historical data. IEICE Trans. Inf. Syst. **97**(11), 2963–2966 (2014)
20. Yildirim, E., Arslan, E., Kim, J., et al.: Application-level optimization of big data transfers through pipelining, parallelism and concurrency. IEEE Tran. Cloud Comput. **4**(1), 63–75 (2015)
21. Yildirim, E., Kim, J., Kosar, T.: Optimizing the sample size for a cloud-hosted data scheduling service. In: Proceedings of the 2nd International Workshop on Cloud Computing Science Application (2012)
22. Cardwell, N., Savage, S., Anderson, T.: Modeling the performance of short TCP connections. Techical Report (1998)