Parallel Computing 82 (2019) 3–18

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Client-side straggler-aware I/O scheduler for object-based parallel file systems

Neda Tavakoli, Dong Dai, Yong Chen*

Computer Science Department, Texas Tech University, Lubbock, TX, USA

ARTICLE INFO

Article history:
Received 21 September 2016
Revised 15 June 2018
Accepted 10 July 2018
Available online 22 August 2018

Keywords:
Parallel file systems
I/O scheduler
Straggler
High performance computing

ABSTRACT

Object-based parallel file systems have emerged as promising storage solutions for high-performance computing (HPC) systems. Despite the fact that object storage provides a flexible interface, scheduling highly concurrent I/O requests that access a large number of objects still remains as a challenging problem, especially in the case when stragglers (storage servers that are significantly slower than others) exist in the system. An efficient I/O scheduler needs to avoid possible stragglers to achieve low latency and high throughput. In this paper, we introduce a log-assisted straggler-aware I/O scheduling to mitigate the impact of storage server stragglers. The contribution of this study is threefold. First, we introduce a client-side, log-assisted, straggler-aware I/O scheduler architecture to tackle the storage straggler issue in HPC systems. Second, we present three scheduling algorithms that can make efficient decision for scheduling I/Os while avoiding stragglers based on such an architecture. Third, we evaluate the proposed I/O scheduler using simulations, and the simulation results have confirmed the promise of the newly introduced straggler-aware I/O scheduler.

1. Introduction

The shift toward data-driven scientific discovery and innovation has made many high-performance computing (HPC) applications more highly data intensive than ever before. The I/O performance is considered an increasingly critical factor that determines the overall HPC system performance. In the meantime, object-based parallel file systems [1], in which files are represented as a set of objects stored on object-based storage devices (OSDs) [2,3], managed by object storage servers (OSSs), have been increasingly deployed on large-scale high-end/high-performance computing systems due to their merits of improved scalability, manageability, and performance [4,5], especially when highly concurrent I/O requests occur. Developing a highly efficient I/O scheduler in the object-based storage systems is arguably critical and a well-acknowledged challenge. A considerable amount of work has been done in this space [2,6–11]. Among them, the *straggler* problem [12–16], which occurs when some of OSSs take a much longer time in responding to I/O requests than other servers, has drawn lots of attention recently. The occurrence of stragglers has significant effects on I/O performance of object storage systems. Since in HPC applications, clients normally need to synchronize after each I/O phase [3,17], the overall I/O performance will be determined by the longest one, which in turn is determined by the *slowest* object storage server. In general, the slow storage servers (i.e., stragglers) can be divided into two categories: long-term stragglers and short-term stragglers. Long-term stragglers can be slow for hours, days and even in the worst-case forever. The main reasons of occurring this type of straggler can be outdated

* Corresponding author.
E-mail addresses: neda.tavakoli@ttu.edu (N. Tavakoli), dong.dai@ttu.edu (D. Dai), yong.chen@ttu.edu (Y. Chen).

https://doi.org/10.1016/j.parco.2018.07.001

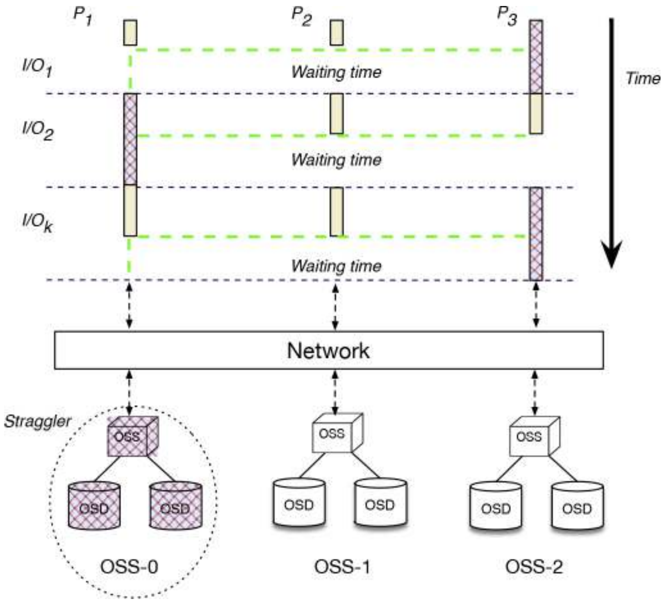


Fig. 1. An illustration of how a straggler in object storage servers can affect the I/O performance in HPC.

hardware, hardware failures and even software bugs. On the other hand, short-term stragglers last only for minutes or less. The main reasons of this type of straggler are interference or resource contention between applications. Fig. 1 illustrates an example of how a straggler in object storage servers can affect the I/O performance. Assuming application processes issue three I/O requests, and each line on the top of the figure represents part of each I/O to access OSSs. The bottom part of the figure illustrates three OSSs in this example, assuming that OSS-0, in shaded pattern, is a straggler. If any part of an I/O hits the straggler server OSS-0, the entire I/O suffers from the straggler problem because the I/O cannot be completed until the slowest server OSS-0 finish servicing its part of I/O access. The existence of storage server stragglers can significantly reduce the productivity of the HPC system. Even though an asynchronous I/O can decouple the computation and I/O and is helpful in some cases, but it is not always possible. For instance, if the computation depends on the I/O reads, or the simulation output is so large that these data have to be flushed to storage systems before the computation can carry on, the application processes have to be blocked until the I/O is completed. The storage server straggler problem can be catastrophic in projected extreme-scale systems, as the large-scale storage system significantly increases the possibility of the existence of a straggler in storage servers.

In order to mitigate the impact of storage server stragglers, a straggler-aware I/O scheduling mechanism is imperatively desired for increasingly popular object-based parallel file systems. Numerous recent studies have focused on this problem, including our prior research of a two-choice randomized I/O scheduling strategy [18]. However, these strategies suffer from expensive probing messages and communication overhead, as they need to probe storage servers' status and outstanding workloads to detect the existence of stragglers and adopt a randomized scheduling strategy to avoid the straggler. In this research study, to overcome the limitation of probing message based straggler-aware I/O scheduling, we extend our client-side, log-*assisted straggler-aware I/O scheduling* [19]. The fundamental idea is that, the I/O scheduler at the client side maintains a log of I/O requests, server's status, and past scheduling decisions, so that the I/O scheduler client can make an optimized scheduling decision when stragglers exist, without incurring expensive probing messages and communication cost. Based on this new idea, we also introduce three different scheduling policies that leverage this log information to make the scheduling decision. To validate the newly introduced log-assisted straggler-aware I/O scheduling strategy and to verify the effectiveness of different scheduling policies, we have conducted simulation evaluation, and these simulation results confirm that the newly proposed scheduler is effective in improving I/O performance with avoiding storage server stragglers and significantly reducing probing messages and communication cost. The contribution of this research study is three-fold:

- Introduce a client-side, log-assisted straggler-aware I/O scheduling for object-based parallel file systems with the log design and the mechanism of maintaining the logs at the scheduler client;
- Introduce three scheduling algorithms based on the log-assisted straggler-aware scheduling mechanism that can be used in parallel file systems;
- Conduct simulation evaluations to validate the concept and analyze the effectiveness of a log-assisted straggler-aware scheduler and two different scheduling algorithms.

The rest of this paper is organized as follows. Section 2 gives the background and motivation of this study. Section 3 gives a detailed description of the proposed client-side log-assisted I/O scheduler and proposed scheduling al-

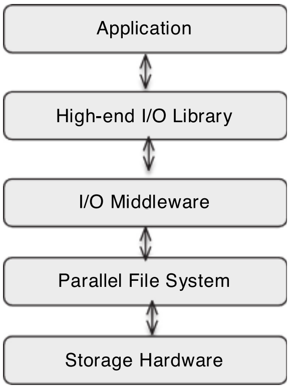


Fig. 2. HPC I/O software stack.

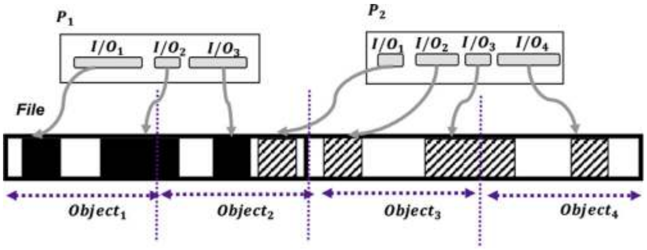


Fig. 3. Example of object file.

gorithms. Section 4 reports the experimental results. Section 5 discusses relevant work and compares with this study. We conclude this research and describe possible future work in Section 6.

2. Background and motivation

2.1. HPC I/O stack

In Fig. 2, we illustrate a typical HPC I/O software stack. The hierarchical view starts from the *application* level, which issues the I/O requests through underlying libraries. There are various *I/O libraries* that can be used for applications to easily and efficiently describe their data requests, for examples HDF5 [20], parallel NetCDF [21] and ADIOS [22]. In most cases, these libraries will translate users' requests into function calls to the *I/O middleware* such as MPI-IO [23]. The I/O middleware normally provides performance optimizations like collective I/O and data sieving [13]. All the requests are finally translated into POSIX function calls to underlying *parallel file system* such as GPFS [24], Luster [25], PVFS, etc. Together with the rapid adoption of object-based storage systems (storage hardware, e.g. object storage devices, OSDs), most parallel file systems in the I/O software stack also become object-based. Along with those I/O software stack layers, the scheduler can run on several places. It can run in high-end I/O libraries [22], in I/O middleware [26], or even in the client-side library of parallel file systems [27]. In this paper, we focus on the last case, i.e., the client-side library of parallel file systems. In object-based parallel file systems, it is actually the client library to divide a file into multiple objects and issue requests on those objects. Fig. 3 illustrates an example of how an object-based parallel file system schedules I/O requests. Normally, there will be multiple applications running with multiple processes to issue object access requests. In this figure, we only show two processes as an example. Here, $p1$ and $p2$ running on different compute nodes. Each of them issues multiple I/O requests on different objects. For example, I/O_1 of p_1 accesses $object_1$ and I/O_4 of p_2 accessing $object_4$. I/O requests may hit the boundary of objects, so it will be split into several requests. For example, I/O_2 of p_1 accessing both $object_1$ and $object_2$. Each object will be scheduled separately. In the following discussion, we no longer explicitly differentiate this scenario for simplicity. After scheduling, each I/O request will be assigned to an appropriate OSD, which is the decision made by the I/O scheduler of object-based parallel file systems. This OSD will provide service for accessing the object.

2.2. Dynamic I/O scheduler

In previous work [18], we have proposed a dynamic client-side I/O scheduler for object storage systems in HPC environment. The core idea of such a scheduler is to dynamically schedule I/O requests to the best appropriate server-based on their real-time performance instead of always scheduling certain object to a fixed object server. Fig. 4 shows how the

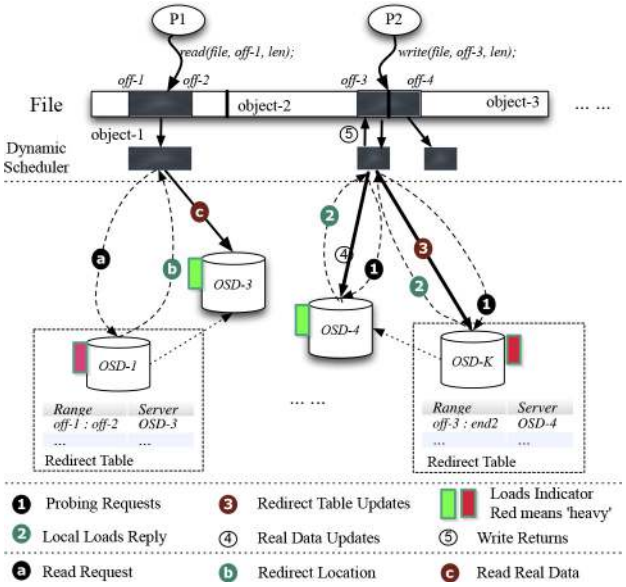


Fig. 4. The procedure of dynamic I/O schedulers [18].

proposed I/O scheduler works for read and write requests as an example. Each time when a new write request is issued on a certain object, the scheduler will first randomly probe two servers (one of them will be original server) and choose the best performing one as the candidate for serving the real I/O requests. This approach can effectively avoid possible stragglers since it compares two object servers each time, hence has a high possibility to avoid stragglers. But, as the figure shows, it may also create overhead on the future read operations, which need to first find the location and then issue the read. Although there are many advantages in avoiding stragglers, dynamic scheduling does introduce other problems. First, it will introduce extra overhead due to the probe phase before I/O scheduling each time. We have proposed a collaborative probe strategy to collaboratively collect performance of storage servers to reduce such overhead. Second, if multiple clients try to probe at the same time, they will get the same best performing server and lead to a possible new bottleneck (i.e., straggler) because of the scheduling action. We have introduced a pre-assigned strategy to tackle this. By pre-assigning, each server will predict whether it will be chosen or not based on historical data and current load. If a server predicts itself to be highly possibly chosen, later probes from other clients will receive updated loads.

Although the two-choice randomized scheduler tends to achieve more balanced workload distribution and better performance when stragglers exist, it still can be largely improved. One significant area is to improve the accuracy and efficiency of client-side probing. It is obvious that client-side schedulers actually are able to learn OSD statuses by recording the historical information about previous scheduling decisions and their performance. In this research, we follow the dynamic I/O scheduling approach, but introduce a new client-side logging mechanism to avoid the probing.

3. Architecture and design

3.1. System architecture

The overall architecture of the proposed log-assisted straggler-aware I/O scheduler is illustrated in Fig. 5. The proposed scheduler runs on the client-side (compute nodes), as an I/O scheduler serving each I/O request. A key component of the proposed scheduler is a *client-side server statistic* log, which will be described in detail in the next subsection. In the object storage server side, we reuse the components designed and implemented in our previous work [18]. Specifically, we have a *redirect table* and *metadata maintainer thread* running on the object storage servers. Among them, the redirect table is used to remember the default location of data objects. This is necessary since we will dynamically place I/O requests to different object storage servers based on the scheduling decision. Obtaining a distributed redirect table in each object storage server is more scalable compared with storing every redirection into metadata service of the parallel file systems. Another component, the metadata maintainer thread runs in the background to move the redirected objects back to their default location when the file systems are idle. This helps keep the data location consistent with metadata in parallel file systems and hence improves the performance of further reads. Fig. 6 further illustrates an example of how the redirect table and metadata maintainer thread work. It shows a data fragment (in black), which should be placed on OSD_0 but is scheduled to be written to OSD_2 as OSD_0 is found to be a straggler. After the scheduler arranges the data fragment to be written to OSD_2, a new entry will be created in the redirect table of OSD_0 indicating that the current location of the segment is

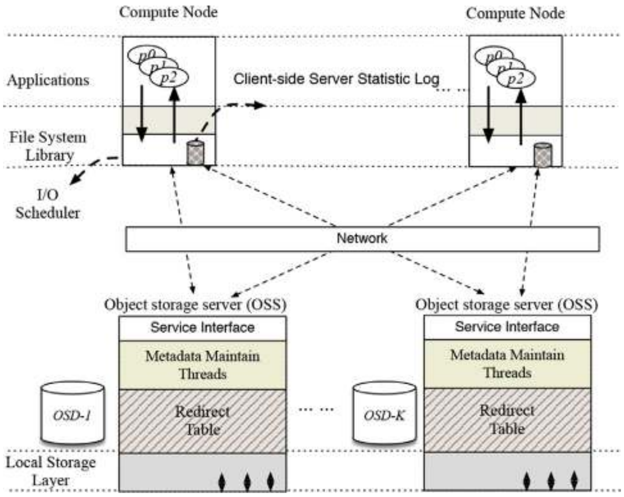


Fig. 5. Overall system architecture of a log-assisted straggler-aware I/O scheduler.

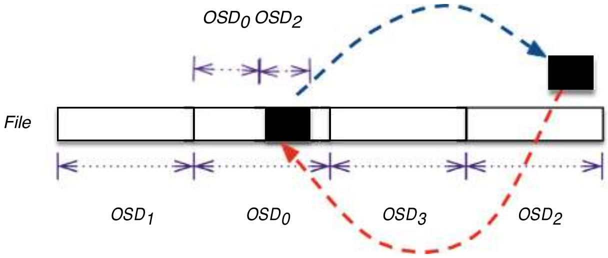


Fig. 6. Redirect table for straggler-aware I/O scheduler.

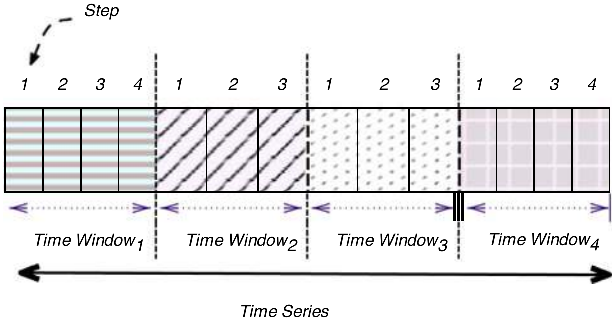
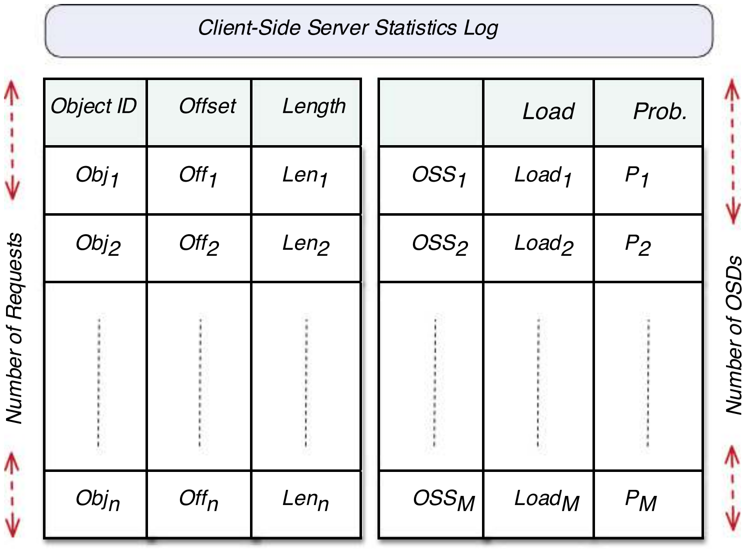


Fig. 7. Time series, time window, and step number in scheduling.

on OSD_2. The dashed red line indicates the metadata maintainer thread periodically runs to retrieve this segment back and deletes the entry in the redirect table. More details can be found in our previous work [18].

3.2. Scheduling model

In the proposed architecture, for each application run with multiple processes, the I/O requests from each process are combined and scheduled to access object storage servers together in a fashion similar to the collective I/O. These I/O requests are not necessarily from the same process and can be from a group of processes, similar as in collective I/O. Specifically, the concurrent I/O requests issued in one client are queued temporarily and served in the group. We define these multiple I/O requests served together as a *time series*, similar as in [27]. We show the concept of time series in Fig. 7. Time series is divided into consecutive segments called *time window*, which contains a fixed time interval. Each time, the proposed I/O scheduler will serve all buffered requests before moving to the next time window. Inside one time window, a number of I/O requests can be queued. We divide queued I/O requests in each time window into multiple steps. Each step will contain the I/O requests on the same object to reduce unnecessary network communications.



Object ID	Offset	Length
Obj_1	Off_1	Len_1
Obj_2	Off_2	Len_2
⋮	⋮	⋮
Obj_n	Off_n	Len_n

	Load	Prob.
OSS_1	$Load_1$	P_1
OSS_2	$Load_2$	P_2
⋮	⋮	⋮
OSS_M	$Load_M$	P_M

Fig. 8. Client-side server statistic log.

3.3. Client-side server statistic log

A client-side server statistic log is introduced as a core component of a new log-assisted straggler-aware I/O scheduler. As shown in Fig. 5, it works inside the I/O scheduler on the client-side.

3.3.1. Data structure of client-side server statistic log

We show the detailed structure of the client-side server statistic log in Fig. 8. It consists of two tables: an *I/O request table*, shown as the left part of Fig. 8, and a *server statistic table*, shown in the right part. In the I/O request table, each request forms a row, which consists of three fields: 1) *Object ID*, which is a unique identifier (a very large index number) of each object; 2) *Offset*, which denotes the starting offset of the I/O request from the beginning of the object; and 3) *Length*, which is the size of the I/O request (in bytes). On the other hand, the server statistic table consists of statistics of each object storage server. It is generated based on scheduling decisions from previous time windows. The *Load* field is an expectation of the amount of I/O requests (in terms of bytes) in each object storage server. Different from our previous work [18], in this research, we do not probe the servers to retrieve their real-time loads; instead, we use the server statistic table to update and record current load of each storage server based on previous scheduling decisions. We use such load to calculate the probability of selecting a corresponding storage server for a given I/O request, which is maintained and updated in the *Prob* field. In the next subsection, we introduce how to update the server statistic table and also calculate the probability.

3.3.2. Maintaining servers statistic table

First, after serving each I/O request in a single step, the server loads in the server statistic table will be updated based on the following formula:

$$l = l' + Len \tag{1}$$

where l is the expected server load at the end of each step, l' represents the server load from the previous step, and Len is the length of scheduled I/O requests. At the beginning of the next step, we will calculate the probability of selecting server i and use such probability to choose the best storage server to schedule the I/O request. The probability is calculated according to the following formula:

$$p_i = p'_i * e^{-l_i} \tag{2}$$

where $p^{j}{}_{i}$ indicates the current probability of selecting server i, and l equals to the updated load of server i. The initial $p^{j}{}_{i}$ is calculated based on the default round-robin scheduling strategy: given there are M object storage servers, each server is equally assigned a $p^{j}{}_{i} = \frac{1}{M}$. Through calculating such a probability for each object storage server, we are able to choose one as target (the detailed selection algorithms are introduced in the next subsection). After making a scheduling decision, we will need to update the server statistic table based on Eq. (1). In addition to updating the load of the selected server, we also need to update the probability of choosing another server to prepare for the next scheduling. In fact, the summation of probabilities of choosing all storage servers should equal to 1. This also requires us to update their probabilities too. The

Algorithm 1 Max Length-Min Load algorithm (MLML).

```
procedure MLML(servers,requests,threshold)
    sortedServers = sort(servers);
    sotRequests = sort(requests);
index = 0;
while need_schedule() do
    default_oss = requests[index] mod M;
    target_oss = sotRequests[index] mod M;
    benefit = load(default_oss) - load(target_oss);
    if benefit ≤ threshold then
        return target_oss;
    else
        return default_oss;
    index++;
```

following equation shows how this is done:

$$p_{\ j} = p_j^{\mathrm{j}} + \frac{(p^{\mathrm{j}} - \ p_i^{\mathrm{j}} * e^{-l_i})}{M - 1}, j \text{ Ç } i \tag{3}$$

where M indicates the total number of storage servers, i is the server that is chosen to serve the current I/O request, and j refers to all other object storage servers. This equation indicates that we evenly re-distribute the decreased probability of server i to all other object storage servers (j). We use this strategy to maintain the probability. Note that, in Eq. (2), we use exponential distribution to calculate the probability of a server being chosen according to their current loads. We use the exponential distribution [28] to calculate it because this distribution describes the behavior of balanced scheduling. Specifically, it indicates that servers with high loads (which could be very high due to imbalanced requests) should be considered with lower probability to serve incoming I/O requests to avoid the straggler. On the other hand, the server with lighter loads (which could approach to 0) should be considered with a much higher probability of being chosen. If a server does not contain any load (i.e., 0), it should be selected with the highest priority. Other distributions can possibly describe similar behaviors; however, since that is not our focus in this research, we leave the investigation of distributions as a future work.

3.4. Scheduling algorithms

Based on the information stored in the server statistic log, we propose three new scheduling algorithms to leverage this statistic log to develop an efficient straggler-aware I/O scheduler. Before introducing the proposed scheduling algorithms, we first introduce the base-line algorithm, a round-robin (RR) algorithm. This strategy is widely used in modern parallel file systems like PVFS, GPFS [24], and Lustre [29]. For an I/O request on object i, RR schedules it to the storage server i mod M, where M indicates the total number of object storage servers to be fair among multiple I/O requests and servers, as well as to avoid starvation. It is not trivial to see that the RR strategy does not work well with stragglers as it does not consider current performance of storage servers. Hence, it may still choose the overloaded servers even there are lightly-loaded servers that can be chosen.

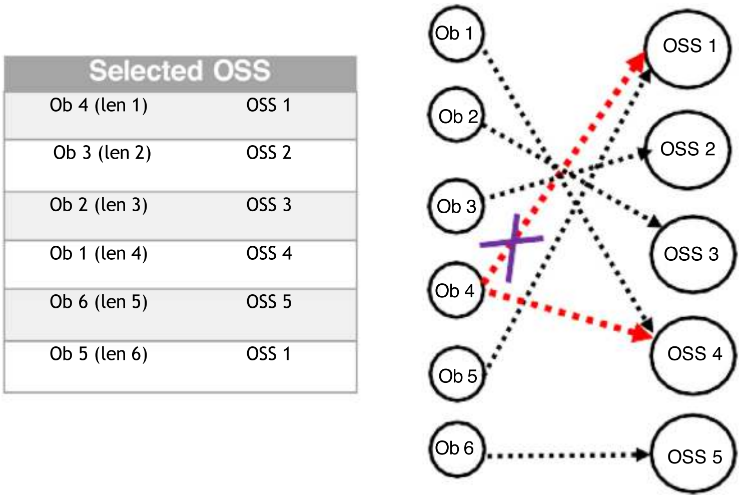
We introduce three straggler-aware scheduling algorithms based on the proposed client-side server statistic log. The first one, called *Max Length–Min Load (MLML)* scheduling algorithm, directly uses logged server statistics to direct scheduling; i.e., it will select the server with lighter workloads with a higher probability. The second scheduling algorithm, called *Two Random from Top Half (TRH)*, takes advantage of the random strategy in addition to server statistic log. The third one, called *n-Level Two Random (nLTR)* scheduling algorithm, which not only takes advantage of the random strategy but also classifies the storage servers (i.e, by using server static log) to different sections, which helps to spread the requests to be more balanced.

Our evaluations confirm that these three scheduling algorithms can reduce the impact of existing stragglers and avoid generating new stragglers. More details of their comparisons can be found in the evaluation section. We discuss each of these three algorithms in detail below.

3.4.1. Max Length–Min Load (MLML) scheduling algorithm

This scheduling strategy uses the statistic server log to sort the storage servers based on their probabilities from the highest to the lowest. At the same time, the I/O requests are sorted from the maximum length to the minimum one. These two sorted lists are processed, from the top to the bottom of lists then starting again from the top of the lists in a circular manner. As a result, the maximum length I/O request will be scheduled to the server with lighter load and the minimum length I/O request will be distributed to the server with heavier load.

We describe the pseudocode of the MLML scheduling strategy in Algorithm 1. In this algorithm, two sorted lists *sortedServers* and *sortedRequests* store a sorted list of servers and I/O requests, respectively. The default_oss describes the default location of requested objects. As we have described, such a server is selected through the default RR strategy. The target_oss,



Selected OSS	
Ob 4 (len 1)	OSS 1
Ob 3 (len 2)	OSS 2
Ob 2 (len 3)	OSS 3
Ob 1 (len 4)	OSS 4
Ob 6 (len 5)	OSS 5
Ob 5 (len 6)	OSS 1

Fig. 9. Example of MLML algorithm.

Algorithm 2 Two Random from top Half (TRH) Algorithm.

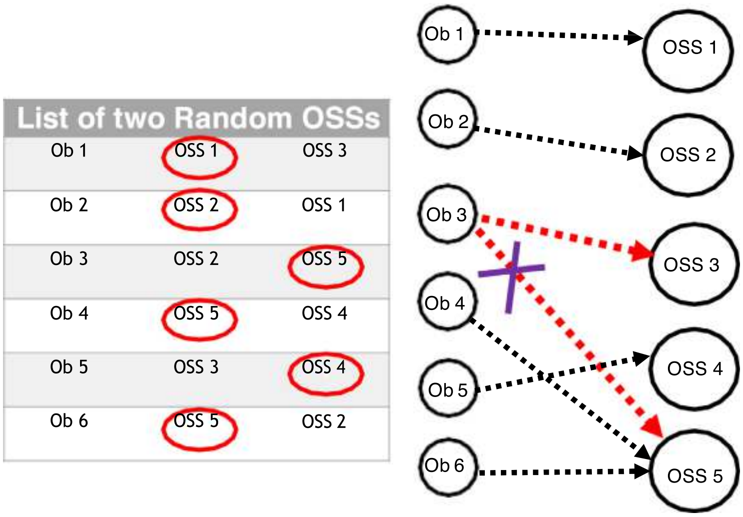
```
procedure TRH(servers,requests,threshold)
  sortedServers = sort(servers);
  index = 0;
  while need_schedule() do
    default_oss = requests[index] mod M;
    ross_1 = random_oss();
    ross_2 = random_oss();
    target_oss = find_min_load(ross_1, ross_2);
    benefit = load(default_oss) - load(target_oss);
    if benefit $\leq$ threshold then
      return target_oss;
    else
      return default_oss;
    index++;
```

on the other hand, represents the storage server calculated based on this MLML scheduling algorithm. If the target_oss and the default_oss are not the same server, we then need to consider whether it is worth of scheduling this request to the target_oss as this might increase read overhead due to the redirection. We introduce a user-defined threshold to indicate how much overhead is acceptable. Specifically, we compare loads of those two servers and if the benefit of choosing the target_oss over the default_oss is larger than the threshold, the selection is acceptable; otherwise the default server will still be selected. The rationale behind this algorithm is that a server with lighter load has a lower chance of being a straggler. By scheduling larger requests on the lighter server, we can reduce or minimize the impact of existing stragglers while the chance of generating new stragglers can be significantly decreased. Fig. 9 illustrates an example of how this MLML algorithm works. In this example, there are six objects and five object storage servers. As shown in the figure, I/O requests are sorted based on their length, and OSSs are sorted based on their probability of being stragglers from the highest to the lowest. Based on the MLML strategy, these I/O requests are assigned to OSSs based on the sorted list. In this example, all selected OSSs by the scheduling algorithm provide benefits greater than the threshold, except for the ob 4, which will be scheduled to the default server (i.e. distributed to OSS4 instead of OSS1). Note that the redirect table is used to keep track these writes and maintains metadata consistency.

3.4.2. Two Random from Top Half (TRH) algorithm

This algorithm considers randomized choices based on the server statistic log to further mitigate the impact of stragglers. The rationale of using randomized choices in this algorithm is that it provides more possible options for selecting servers and helps to spread the requests to be more balanced. In this scheduling algorithm, the sorted list of object storage servers is also used, same as in the MLML algorithm. Specifically, for a given I/O request, two object storage servers will be randomly chosen from the top half of all object storage servers. The top half indicates $\frac{M}{2}$ servers that have lighter loads. We also use the same threshold as in the previous algorithm to decide whether choosing the calculated target_oss or the default_oss. Algorithm 2 shows the pseudocode of the TRH algorithm.

Fig. 10 further illustrates an example showing how this algorithm works. In this example, for each object, two OSSs are randomly selected as potential targets and the one with the lighter load will be chosen according to the Algorithm 2. In



List of two Random OSSs		
Ob 1	OSS 1	OSS 3
Ob 2	OSS 2	OSS 1
Ob 3	OSS 2	OSS 5
Ob 4	OSS 5	OSS 4
Ob 5	OSS 3	OSS 4
Ob 6	OSS 5	OSS 2

Fig. 10. Example of TRH algorithm.

this specific example, we show that all objects are scheduled to the lighter load server based on their two random choices except for ob 3 that is scheduled to its default_oss because its default_oss (i.e., $OS S_3$) has a similar load compared to $OS S_5$ (the algorithm considers it is not worth of scheduling it to another location).

3.4.3. n-Level Two Random (nLTR) algorithm

In this scheduling algorithm, we first sort all storage servers in a descending order based on their probabilities in the statistic server log. Similar to TRH, this algorithm also considers randomized choices based on the server statistic log to mitigate the impact of stragglers. But, different from TRH, the randomized choices are no longer chosen from the top half of the sorted list of object storage servers; instead, a n-Level strategy is used. The n-Level strategy first divides the sorted storage servers list to the K separate sections (where $K = 2^n$). At the same time, it also orders all pending I/O requests based on their lengths to form a descending order and divides them into the same number K sections. To schedule, it will assign I/O requests in section i into a randomly selected server in the ith section of the storage servers. The way to divide storage servers and I/O requests is described as follows. Note that there is a core difference: we use the middle to divide storage servers to guarantee each section has the same number of storage servers; but we use the average to divide I/O requests to better utilize the size factor. Let's consider S as the number of separate sections that we have so far, therefore:

- If $S = K$, no further action is required.
- If $S < K$, the list of storage servers and I/O requests are divided to 2^l sections as follows, where l represents level number:
  - Middle/average element of each available section is chosen.
  - Each section is divided in two sections by using the middle/average element.

We show an example of dividing storage server list in Fig. 11. Here the chosen number of sections is $K = 16$, and the number of levels is $n = \lfloor \log_2 K \rfloor = \lfloor \log_2 16 \rfloor = 4$. All storage servers are divided in 16 sections. As the figure shows, at the Level 1, the middle element of the list (i.e, M1) is used to divide the storage servers list in two sections. At the Level 2, the middle elements of two sections are chosen (i.e, M2, M3) for dividing their corresponding sections. Therefore, at each level, for every section the middle element is chosen to divide it in two sections. This will continue until $K = 16$ sections have been created. The rationale of using K sections in this algorithm is that it helps to spread the requests to be more balanced. Specifically, for a given I/O request, if its length is very small, it is more efficient to choose a storage server that might not be the lightest one but still has enough capability to process the I/O request.

The detailed scheduling algorithm based on such sorted list is described in Algorithm 3. We use the same threshold as in previous algorithms to decide whether choosing the calculated target_oss or the default_oss. The nLTR algorithm allows users to decide an appropriate value for n. In this research, we evaluate two initial cases $n = 1$ and $n = 2$. Large n clearly indicates higher computation complexity. But our evaluations did not show clear performance benefit of increasing n. For example, even the evaluation results for $n = 3$ did not lead to significant performance benefit compared to $n = 2$. However, the computation overhead to generate the sections for $n = 3$ is greater than that of the $n = 2$. In general, the average computation overhead in nLTR algorithm is equal to the overhead required to generate sections for I/O requests and storage servers. Since generating sections ends up in a complete binary search tree, this means that the nLTR algorithm can have the same big-O time/space complexity as the complete binary search tree. Therefore, the average computation overhead in nLTR algorithm is calculated as follows:

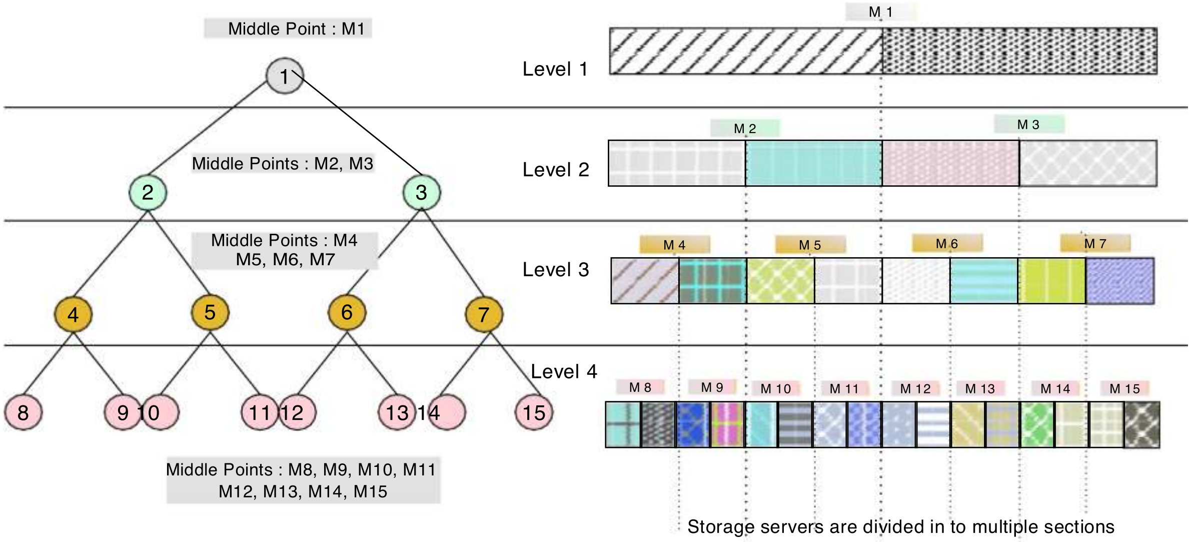


Fig. 11. Levels for dividing the storage servers.

Algorithm 3 *n*-Level Two Random (*n*LTR) Algorithm.

```
procedure nLTR(servers,requests,threshold)
  sortedServers = sort(servers);
  requests = sort(requests);
  index = 0, i = 0;
  level = 1;
  K = NumberofSections;
  $n = \lfloor \log_2 K \rfloor$ ▷ $n$ = Number of levels;
  while (need_schedule() do
    default_oss = requests[index] mod M;
    while ($level \neq n$) do
      for existSection() do
        M = findMiddleElement();
        MiddlePoints[$i$] = M;
        divideSection();
        i++;
      level++;
    T = compare(IOLength, MiddlePoints[]);
    sectionNum = FindSection(T);
    ross_1 = randomOssFromSectionNum(sectionNum);
    ross_2 = randomOssFromSectionNum(sectionNum);
    target_oss = MinLoad(ross_1, ross_2);
    benefit = load(default_oss)- load(target_oss);
    if benefit ≤ threshold then
      return target_oss;
    else
      return default_oss;
    index++;
```

- Space complexity: The space occupied to maintain middle and average points which is equal to $O(k)+O(k)=2O(k)$ for both average and worst case, where $k=2^n$
- Time complexity: Is the time required to find (i.e., search) an appropriate section for a given I/O request which is equal to $O(k \log_2 k)$ for average case, and $O(n)$ in worst case.

Hence, we suggest these two values (i.e., $n=1$ and $n=2$) should be used in most cases.

4. Evaluation and analysis

We have evaluated the proposed I/O scheduler and scheduling algorithms using simulation. In the simulation, we used synthetic workloads generated based on real-world traces. We constructed the synthetic workload based on combining three different types of I/O requests: large I/O (each request is greater than $O(10\,\text{MB})$), medium I/O (each request is between 4 MB and 10 MB) and the small one (less than 4 MB). These three categories of requests are chosen based on our observation of the

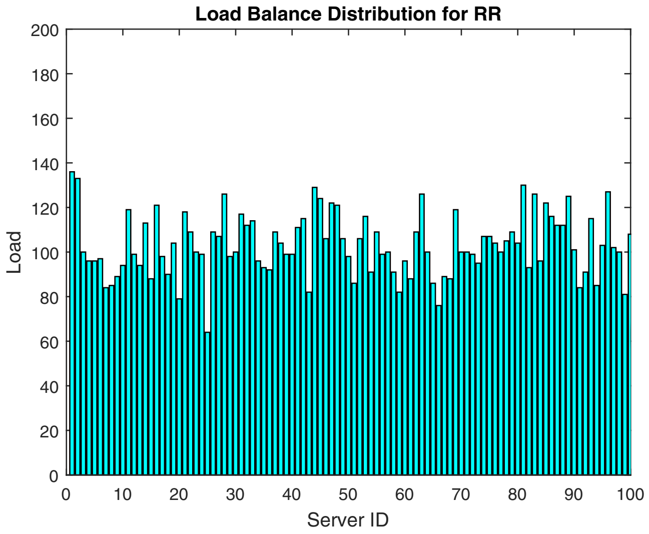


Fig. 12. Load distribution with the RR scheduling algorithm.

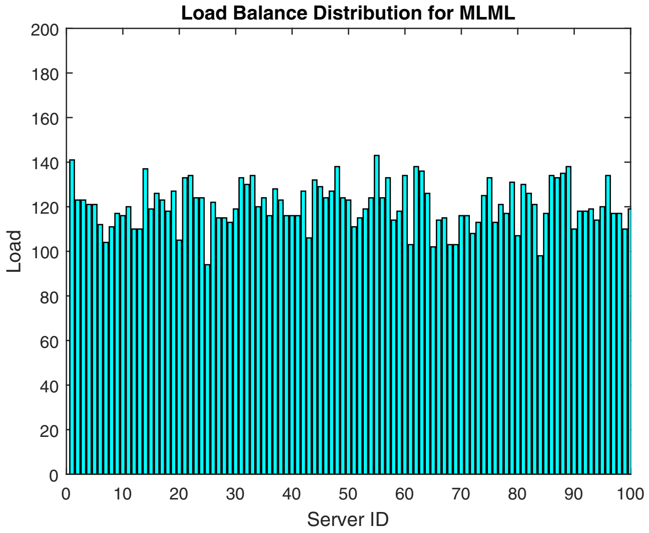


Fig. 13. Load distribution with the MLML scheduling algorithm.

typical scientific applications running on HPC systems. In addition, the initial I/O loads of all OSSs are generated based on a normal distribution with a small standard deviation, which simulates a roughly even workload distribution at the beginning. The simulated environment is an HPC cluster with 100 object storage servers and 200 compute nodes. In all test cases, we run the simulation 100 times and calculate the average load of each OSS with the proposed log-assisted straggler-aware scheduler and with the MLML, TRH, 1LTR, and 2LTR scheduling algorithms, respectively. The total number of I/O requests simulated was 2,000, with various sizes for each request; for instance, the total data written for all medium I/O case can be up to 20GB. The total data written for all large I/O case can be between $O(20GB)$ and $O(2TB)$ depending on the size of each request. We consider the round-robin (RR) algorithm as the base-line case and compare it with the proposed straggler-aware scheduler with three algorithms. In Fig. 12, we first illustrate the load of each server after scheduling all I/O requests using the round-robin policy. The x-axis shows the storage servers, and the y-axis represents the load of each storage server (i.e, write data size in MB). Fig. 13 plots the load distribution of storage servers with the log-assisted straggler-aware scheduler after scheduling all requests using the MLML algorithm. As the figure shows, since the MLML algorithm rotates the requests to servers based on their loads, this algorithm is able to achieve a better load balance across different storage servers compared with Fig. 12. We further show the result of load distribution with the TRH algorithm in Fig. 14, and in Fig. 15 [19]. Since this algorithm randomly chooses two servers from the top half of the lighter loaded storage servers and selects the better one, it can effectively avoid stragglers to achieve better balance compared with the round-robin strategy. However, due to the randomness behavior it may have larger/smaller variances than that of the MLML algorithm. Fig. 14 illustrates

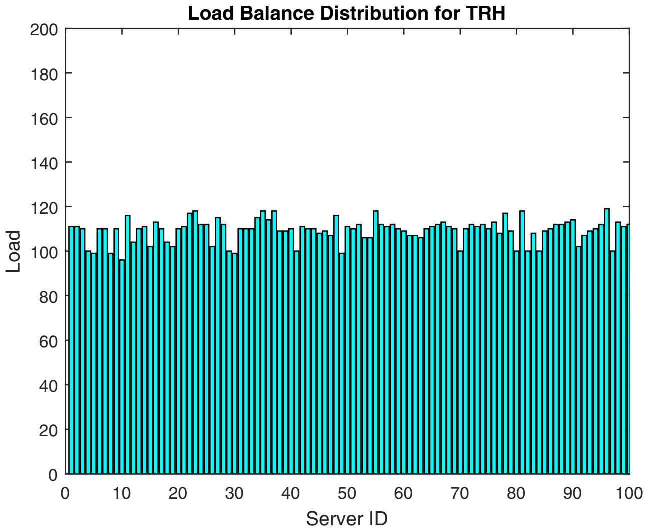


Fig. 14. Load distribution with the TRH scheduling algorithm (first run).

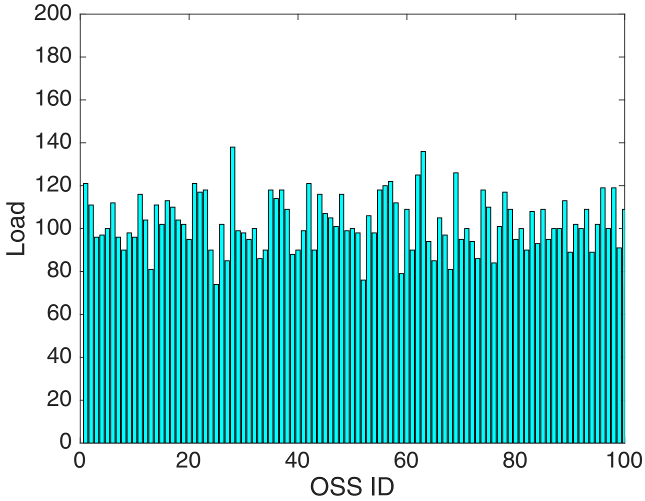


Fig. 15. Load distribution with the TRH scheduling algorithm (second run) [19].

the case where TRH performs better than MLML but Fig. 15 represents the case that TRH achieves worse balance compared with the MLML. We further show the result of load distribution with the 1LTR and 2LTR algorithm in Figs. 16 and 17. Since these algorithms randomly choose two servers from multiple sections, they can effectively avoid stragglers to achieve better balance compared with the round-robin strategy. To further validate and verify the effectiveness of the proposed log-assisted straggler-aware I/O scheduler, in this series of evaluations, we manually inject a number of storage servers as stragglers. We expect the straggler-aware scheduler to be able to avoid hitting existing stragglers and also avoid generating new stragglers. In these tests, we assigned 10% of the total object storage servers as stragglers by adding extra loads on those selected servers. Specifically, we injected 5 times more load compared with the average loads assigned on other storage servers. We report the result in Fig. 18. In this figure, the *x*-axis indicates all the possible loads of storage servers after scheduling; the *y*-axis shows the maximal IO requests scheduled onto storage servers that have the corresponding load as the *x*-axis shows. We only show the maximal I/O requests since there might be multiple servers having the same load after scheduling, and the one receiving the most I/O requests actually determine the overall performance as we have described. An effective straggler-aware scheduler should be capable of avoiding stragglers; in other words, the number of I/O requests falling into the slow storage servers (stragglers) should be close to zero. Based on the simulation results reported in Fig. 18, the log-assisted straggler-aware scheduler with MLML, TRH, 1LTR, and 2LTR algorithms are capable of achieving that since they do not schedule any I/O request onto servers with load greater than 200 in our test cases. However, the RR strategy still schedule I/O requests to highly loaded servers (over 300). Between the MLML and TRH algorithm, we can notice that the TRH scheduling algorithm tends to place loads on lighter loaded servers (the curve is on the left side of the MLML

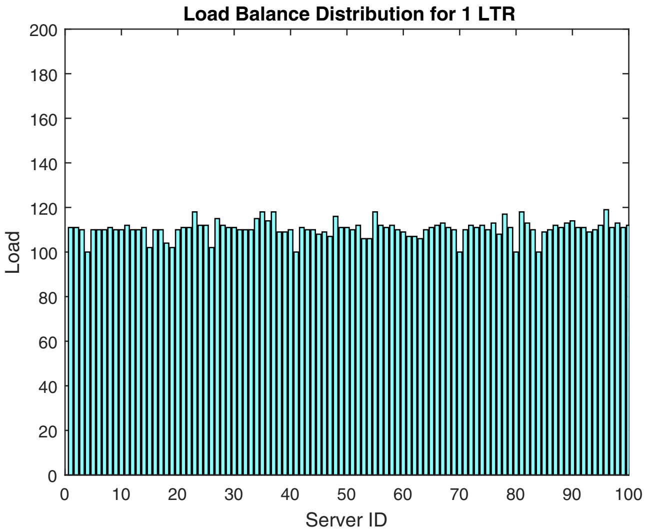


Fig. 16. Load distribution with the 1LTR scheduling algorithm.

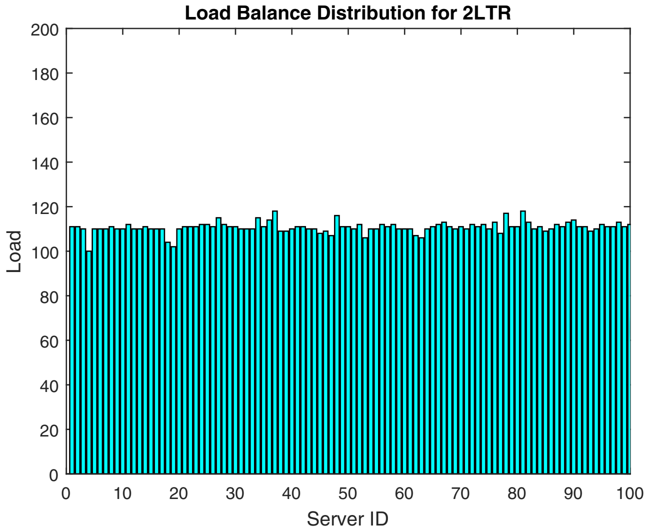


Fig. 17. Load distribution with the 2LTR scheduling algorithm.

curve). This behavior is due to the fact that randomized scheduling choices introduce more options for selection and better promote to spread the requests. Also, between 1LTR and 2LTR, we can notice that they are largely overlapped indicating they provide the similar capability of achieving the balanced I/O. Together with the results plotted in Figs. 16 and 17, we show that, selecting $n = 2$ is good enough for mitigating stragglers and providing a load balanced storage server. These simulation results show that using the client-side server statistic log and the proposed scheduling algorithms can avoid storage server stragglers and improve the scheduled workload balance among storage servers.

5. Related work and comparison

I/O scheduling has been extensively studied in earlier research efforts. The straggler problem has been well acknowledged as one of critical issues in I/O scheduling. Current research is heavily focused on trying to avoid stragglers, including our previous work of an I/O scheduler based on a two-choice randomized scheduling that dynamically places write operations to mitigate the straggler problem in high-end/high-performance computing systems [18]. This paper is different from two-choice approaches as we avoid the probing latency through using client-side logs. ADIOS [30] introduced an adaptive I/O method to reduce the I/O variability within a single application. It gathers all processes inside the application in a group and each group writes in a particular storage server. CALCioM [26] mitigates I/O interferences in HPC systems through a cross-application coordination. It allows different applications to communicate and coordinate their I/O strategies to avoid

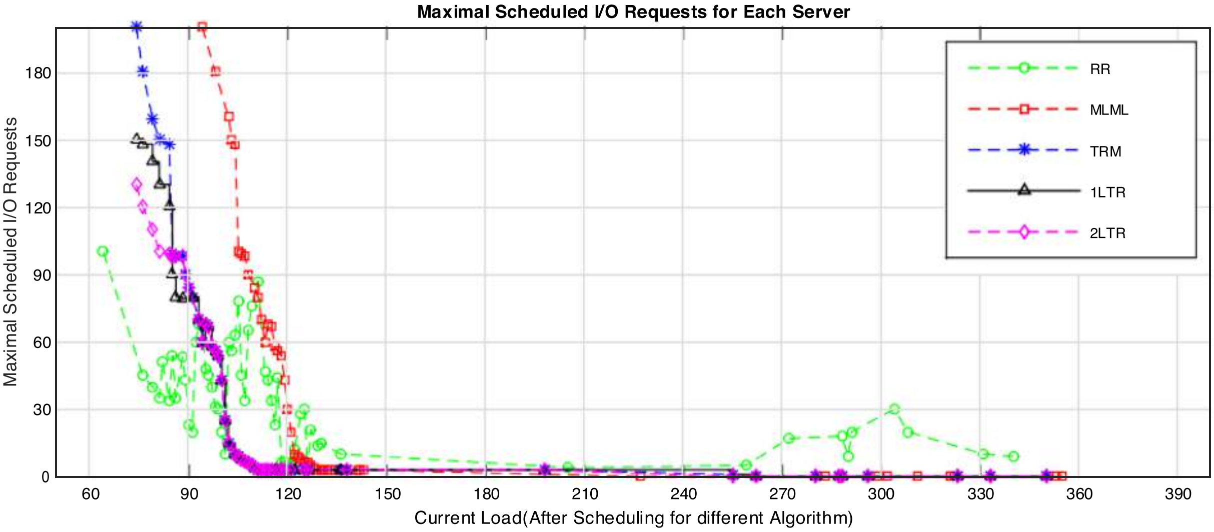


Fig. 18. Comparison of RR, MLML, TRH, 1LTR, and 2LTR scheduling algorithms.

congestion. Our study is different from it as we introduce a client-side scheduler that can transparently identify those patterns and coordinate to mitigate stragglers and hence alleviate the interferences. There are also a series of research focusing on identifying I/O access patterns of applications and using such information to mitigate stragglers and improve the performance [31–33]. Our research is different from them as we do not need to know the details about each application; instead, by checking the requests and previous served I/O logs, we are able to dynamically schedule the I/O requests to mitigate stragglers. The two-choice algorithm is also used in Sparrow, a distributed task scheduler [34]. It uses multiple distributed schedulers on a cloud-based platform to schedule a large number of short tasks. Sparrow avoids the straggler problem by probing multiple nodes for each task and scheduling that task on the node with the least overhead. The Sparrow scheduler is specifically designed for cloud systems and workloads though. Another straggler mitigation technique is introduced in Dolly with a proactive straggler mitigation approach [35]. It uses a proactive database provisioning scheme and leverages virtual machine cloning in cloud platforms. It accomplishes this by using the power-law distribution of job sizes to launch different clones of each task. Furthermore, it attempts to predict stragglers by waiting and observing the system behavior before scheduling a task.

Yet another straggler mitigation technique is presented in Mantri [16], which moves tasks from straggling resources and assigns them to less overloaded resources. It monitors tasks and attempts to avoid stragglers based on their causes. It accomplishes this goal in three primary steps. First, speculative copies of straggler tasks are executed while being aware of resource constraints and work imbalance. Second, all tasks are placed based on the locations of their data sources. Last, once a task is completed, its output (intermediate data) is replicated on other resources with lighter workloads.

One of the widely used techniques for mitigating the impact of stragglers is a speculative execution technique. Speculative execution means that, for those tasks that are stragglers or are likely to be stragglers, extra copies of tasks are executed. Among these copies, the one that finishes first is chosen [16,36,37]. Many systems and studies use speculative executions to address the straggler problem [16,36,38,39], including a cloning approach [40,38] and a straggler-detection-based approach [41,42]. The primary difference between these two approaches is how/when they launch their speculative task copies. In the cloning approach, the initial task is scheduled in parallel with extra copies, and the task that finishes first is used for the next scheduled task. In the straggler-detection based approach, the extra copies are only launched if stragglers are detected. One speculative task scheduler, LATE [39], implicitly assumes that cluster nodes are homogeneous and linearly make progress. It uses these assumptions to speculatively re-execute tasks that appear to be stragglers. In fact, it focuses on reducing the response time of scheduling by speculating which running task can be overtaken. Any task that may be overtaken will then be re-executed to avoid stragglers. Hopper [43], is another example of a job scheduler that uses speculative execution to mitigate the straggler problem. The key idea of Hopper is that it must predict the speculation requirements of jobs and then dynamically launch extra copies on alternative machines. The downside of speculative execution is that, while it mitigates the impact of stragglers, it consumes valuable system resources due to redundant executions. Furthermore, it does not coordinate resources for concurrent jobs well.

Recently, an interesting work is introduced in [44], in which the authors show that a small portion of storage servers (<20%) are stragglers at any given interval, but these stragglers are usually transient. In other words, any storage server that is currently a straggler may return to normal operation within 2 minutes. On the other hand, any non-straggler server may become a straggler within 10 minutes. Recently, an interesting work is introduced in [44,45] in which authors show that a small proportion of storage servers (<20%) are straggler at any given interval, but that stragglers are transient which means any storage server that is currently a straggler, it will return to normal operation within 2 minutes. Any non-straggler storage will become straggler within 10 minutes. Furthermore, the authors show that while an application is running, all I/Os needed for that application will tend to have a long-term average performance for that sized I/O.

In this research, we introduce a log-assisted straggler-aware I/O scheduling. It is different from these existing studies as it addresses the storage server straggler problems in object-based storage systems. It is specifically designed for HPC systems where high concurrency is the norm and uses client-side logs to reduce probing messages that are required in two-choice or similar scheduling algorithms. It avoids redundant executions as required in the class of speculative techniques. To the best of our knowledge, this study is a first research effort that introduces a log-assisted straggler-aware scheduling approach.

6. Conclusion and future work

Many high-end/high-performance computing applications have become highly data intensive and the needs of highly efficient storage systems to better support scientific discovery substantially grow over years. In this research, motivated by the imperative needs of better dealing with storage server stragglers (servers that take much longer time in responding to I/O requests than other servers due to transient overloaded requests, imbalanced accesses, or even hardware/software transient failures), we introduce a new log-assisted straggler-aware I/O scheduler. We have presented the idea, design, and evaluation of such straggler-aware I/O scheduling strategy. We have introduced a client-side server statistic log to maintain I/O requests and serversâA˘Z´ status so that the I/O scheduler client can make an optimized scheduling decision when stragglers exist, without incurring expensive probing messages as in the existing straggler-aware I/O scheduling methods. We presented three new straggler-aware scheduling algorithms based on the proposed client-side statistic logs: Max Length-Min Load (MLML) algorithm, Two Random from Top Half (TRH) algorithm, and (*n*-Level) Two Random algorithm. Our evaluations confirmed that these three scheduling algorithms can reduce the impact of existing stragglers and avoid generating new stragglers, which shows the benefit of log-assisted straggler-aware scheduling scheme for object-based storage systems.

We plan to investigate and integrate the proposed I/O scheduling scheme into existing real parallel file systems. In addition, while the current simulation shows the solution improves the I/O performance, we believe that more optimization can be made with the statistic logs, and we plan to further investigate the possibilities in the future.

Acknowledgment

We are thankful to the anonymous reviewers for their valuable feedback. This research is supported in part by the National Science Foundation under grant CNS-1338078, CNS-1362134, CCF-1409946, and CCF-1718336.

References

[1] M. Mesnier, G.R. Ganger, E. Riedel, Object-based storage, Commun. Mag., IEEE 41 (8) (2003) 84–90.
[2] Y. Liu, R. Figueiredo, D. Clavijo, Y. Xu, M. Zhao, Towards simulation of parallel file system scheduling algorithms with PFSsim, in: *Proceedings of the 7th IEEE International Workshop on Storage Network Architectures and Parallel I/O*, 2011.
[3] J.M. Del Rosario, R. Bordawekar, A. Choudhary, Improved parallel I/O via a two-phase run-time access strategy, ACM SIGARCH Comput. Arch. News 21 (5) (1993) 31–38.
[4] D. Dai, X. Li, C. Wang, M. Sun, X. Zhou, Sedna: a memory based key-value storage system for realtime processing in cloud, in: CLUSTER Workshops, 2012, pp. 48–56.
[5] M. Factor, K. Meth, D. Naor, O. Rodeh, J. Satran, Object storage: the future building block for storage systems, in: Local to Global Data Interoperability-Challenges and Technologies, 2005, IEEE, 2005, pp. 119–123.
[6] R. Jain, K. Somalwar, J. Werth, J.C. Browne, Heuristics for scheduling I/O operations, Parallel Distrib. Syst., IEEE Trans. 8 (3) (1997) 310–320.
[7] F. Chen, S. Majumdar, Performance of parallel I/O scheduling strategies on a network of workstations, in: Eighth International Conference on Parallel and Distributed Systems, 2001. ICPADS 2001. Proceedings, IEEE, 2001, pp. 157–164.
[8] D. Durand, R. Jain, D. Tseytlin, Parallel I/O scheduling using randomized, distributed edge coloring algorithms, J. Parallel Distrib. Comput. 63 (6) (2003) 611–618.
[9] E. Rosti, G. Serazzi, E. Smirni, M.S. Squillante, The impact of I/O on program behavior and parallel scheduling, in: ACM SIGMET-RICS Performance Evaluation Review, 26, ACM, 1998, pp. 56–65.
[10] Y. Wiseman, D.G. Feitelson, Paired gang scheduling, IEEE Trans. Parallel Distrib. Syst. 14 (6) (2003) 581–592.
[11] E. Rosti, G. Serazzi, E. Smirni, M.S. Squillante, Models of parallel applications with large computation and I/O requirements, Softw. Eng., IEEE Trans. 28 (3) (2002) 286–307.
[12] F. Wang, S.A. Brandt, E.L. Miller, D.D. Long, Obfs: a file system for object-based storage devices, in: MSST, 4, 2004, pp. 283–300.
[13] R. Thakur, W. Gropp, E. Lusk, Data sieving and collective I/O in ROMIO, in: The Seventh Symposium on the Frontiers of Massively Parallel Computation, 1999. Frontiers' 99, IEEE, 1999, pp. 182–189.
[14] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, N. Podhorszki, Characterizing output bottlenecks in a supercomputer, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, 2012, p. 8.
[15] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, I. Stoica, The case for tiny tasks in compute clusters, HotOS, 2013.
[16] G. Ananthanarayanan, S. Kandula, A.G. Greenberg, I. Stoica, Y. Lu, B. Saha, E. Harris, Reining in the outliers in map-reduce clusters using mantri, in: OSDI, 10, 2010, p. 24.
[17] J. Liu, Y. Zhuang, Y. Chen, Hierarchical collective I/O scheduling for high-performance computing, Big Data Res. 2 (3) (2015) 117–126.
[18] D. Dai, Y. Chen, D. Kimpe, R. Ross, Two-choice randomized dynamic I/O scheduler for object storage systems, in: International Conference for High Performance Computing, Networking, Storage and Analysis, SC14, IEEE, 2014, pp. 635–646.
[19] N. Tavakoli, D. Dai, Y. Chen, Log-assisted straggler-aware I/O scheduler for high-end computing, in: 2016 45th International Conference on Parallel Processing Workshops (ICPPW), IEEE, 2016, pp. 181–189.
[20] D. Borthakur, Hdfs architecture guide, p. 39, 2008. *HADOOP APACHE PROJECT* http://hadoop.apache.org/common/docs/current/hdfsdesign.pdf.
[21] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, M. Zingale, Parallel netcdf: a high-performance scientific i/o interface, Supercomputing, 2003 ACMIEEE Conference, IEEE, 2003 39–39.

[22] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J.Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, et al., Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks, Concurr. Comput. 26 (7) (2014) 1453–1473.

[23] R. Thakur, W. Gropp, E. Lusk, On implementing mpi-io portably and with high performance, in: Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems, ACM, 1999, pp. 23–32.

[24] F.B. Schmuck, R.L. Haskin, GPFS: a shared-disk file system for large computing clusters, in: FAST, 2, 2002, pp. 231–244.

[25] P.J. Braam et al., "The lustre storage architecture," 2004.

[26] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, et al., CALCioM: mitigating I/O interference in HPC systems through cross-application coordination, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IEEE, 2014, pp. 155–164.

[27] H. Song, Y. Yin, X.-H. Sun, R. Thakur, S. Lang, Server-side I/O coordination for parallel file systems, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2011, p. 17.

[28] A.W. Marshall, I. Olkin, A multivariate exponential distribution, J. Am. Stat. Assoc. 62 (317) (1967) 30–44.

[29] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous data flow programming language LUSTRE, Proc. IEEE 79 (9) (1991) 1305–1320.

[30] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, M. Wolf, Managing variability in the io performance of petascale storage systems, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society, 2010, pp. 1–12.

[31] Y. Lu, Y. Chen, R. Latham, Y. Zhuang, Revealing applications' access pattern in collective I/O for cache management, The 28th International Conference on Supercomputing (ICS'14), 2014.

[32] D. Dai, Y. Chen, D. Kimpe, R. Ross, Provenance-based object storage prediction scheme for scientific big data applications, The 2014 IEEE International Conference on Big Data (BigData'14), 2014.

[33] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, X.-H. Sun, I/O acceleration with pattern detection, in: Proceedings of the 22nd international symposium on High-Performance Parallel and Distributed Computing, ACM, 2013, pp. 25–36.

[34] K. Ousterhout, P. Wendell, M. Zaharia, I. Stoica, Sparrow: distributed, low latency scheduling, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM, 2013, pp. 69–84.

[35] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Why let resources idle? Aggressive cloning of jobs with Dolly," 2012.

[36] G. Ananthanarayanan, M.C.-C. Hung, X. Ren, I. Stoica, A. Wierman, M. Yu, GRASS: trimming stragglers in approximation analytics, in: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), 2014, pp. 289–302.

[37] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.

[38] G. Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica, Effective straggler mitigation: attack of the clones, in: Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), 2013, pp. 185–198.

[39] M. Zaharia, A. Konwinski, A.D. Joseph, R.H. Katz, I. Stoica, Improving MapReduce performance in heterogeneous environments, in: OSDI, 8, 2008, p. 7.

[40] H. Xu, W.C. Lau, Task-cloning algorithms in a MapReduce cluster with competitive performance bounds, in: 2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2015, pp. 339–348.

[41] Q. Chen, C. Liu, Z. Xiao, Improving MapReduce performance using smart speculative execution strategy, Comput. IEEE Trans. 63 (4) (2014) 954–967.

[42] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in: ACM SIGOPS Operating Systems Review, 41, ACM, 2007, pp. 59–72.

[43] X. Ren, Speculation-aware Resource Allocation for Cluster Schedulers, Ph.D. dissertation, California Institute od Technology, 2015.

[44] B. Xie, J.S. Chase, D. Dillow, S. Klasky, J. Lofstead, S. Oral, N. Podhorszki, Output performance study on a production petascale filesystem, in: International Conference on High Performance Computing, Springer, 2017, pp. 187–200.

[45] B. Xie, Y. Huang, J.S. Chase, J.Y. Choi, S. Klasky, J. Lofstead, S. Oral, Predicting output performance of a petascale supercomputer, in: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, ACM, 2017, pp. 181–192.