Journal Pre-proof

Moving small files in a networked environment

Chao Jin, David Abramson, Jake Carroll, Zhengchun Liu,
Rajkumar Kettimuthu

PII: S0167-739X(22)00300-4
DOI: https://doi.org/10.1016/j.future.2022.09.016
Reference: FUTURE 6598

To appear in: *Future Generation Computer Systems*

Received date : 19 June 2022
Revised date : 6 September 2022
Accepted date : 17 September 2022

Please cite this article as: C. Jin, D. Abramson, J. Carroll et al., Moving small files in a networked environment, *Future Generation Computer Systems* (2022), doi: https://doi.org/10.1016/j.future.2022.09.016.

Highlights.

None of existing data movement tools handles Lots of Small Files (LOSF) efficiently, because they mainly optimize network usage. In practice, users are forced to pack small files into a large archival file before transferring them. We identify that the major bottleneck of moving small files is caused by storage I/O.

We have performed a number of experiments using a state-of-the-art transfer tool (GridFTP) to demonstrate the root cause of the performance bottleneck. We also examined engineering approaches that can mitigate it.

Specifically, this paper presents the following contributions:

- We examined the storage and file system features that constrain moving LOSF;
- We analyzed small file transfer performance using a data transfer pipeline model;
- We demonstrated appropriate engineering approaches that can mitigate the LOSF bottleneck.

Moving Small Files in a Networked Environment

Chao Jin[a], David Abramson[a,*], Jake Carroll[a], Zhengchun Liu[b], and Rajkumar Kettimuthu[b]

[a]*The University of Queensland, St Lucia QLD 4072, Australia*

[b]*Data Science and Learning Division, Argonne National Laboratory, IL 60439, USA*

Abstract

Globally distributed computing infrastructures, such as clouds and supercomputers, are currently used to manage data that is generated with an unprecedented speed from a variety of resources. Coping with this trend, the volume of data exchanged across distant sites increases substantially. To accelerate data transfer, high-speed networks are provided to connect remote sites. Most existing data movement solutions are optimized for moving large files. However, it is still challenging to transfer a large number of small files across networks. This disadvantage not only lowers data transfer performance, but also decreases overall system utilization. We identify that moving small files is mainly constrained by degraded file system throughput, not just network performance as might be suspected. We have built a data transfer pipeline model to analyze the impact of small network I/O and storage I/O on data movement. Extending one of the widely used open source data movement solutions, GridFTP, we demonstrate several appropriate engineering approaches that mitigate the bottleneck and increase data transfer efficiency. We show optimizations that improve data transfer performance more than 5 times. In comparison to existing solutions, our approaches can save a significant amount of system resources for moving lots of small files.

Keywords: Data movement; File transfer; Small files; High-speed networking; Distributed computing; Data transfer pipeline; GridFTP

1. Introduction

To address the growth of compute and storage demand in the big data era, high-performance computing (HPC) systems and clouds are being deployed worldwide [59] to accelerate data-intensive computing. Coping with this architectural advance, massive amounts of data is distributed geographically, and manipulated beyond organizational boundaries. Typically, manipulating data between geo-distributed sites involves moving large volumes of files across wide area networks (WANs). For example, many scientific workflows transfer a large number of files between distant computing facilities for data acquisition, processing, analysis, interpretation, sharing, and archive [53][66].

Moving data in a WAN has been studied extensively. Most existing tools designed for Grid computing focus on increasing the data transfer rate to saturate high-bandwidth interconnects, especially for large files. Recent studies on international scientific data movement shows the size of files has a substantial impact on the overall performance. Specifically, moving lots of small files (LOSF) slows down the data transfer rate and significantly decreases the utilization of high-speed networks [63][66].

The smallest file size handled efficiently by most existing data movement tools is around the magnitude of network bandwidth delay product (BDP), which is typically larger than 10MB for most existing WAN configurations [33]. However, a study on scientific data movement shows the median file size of datasets moved between data centres is only a few MBs, with the majority of files less than 1 MB [65][66].

* Corresponding author
Email addresses: {c.jin, david.abramson, jake.carroll}@uq.edu.au, {zhengchun.liu, kettimut}@anl.gov

Small files cause performance problems due to the comparatively high meta-data overhead and lower efficiency of handling small payloads. Manipulating LOSF not only becomes a challenge for computations [39][46], but also creates a serious bottleneck [63][64] when they are transported through networks, either within a cluster or across data centers. To date, most existing wide area data movement solutions cannot handle files smaller than 1MB efficiently. In particular, widely used tools, such as GridFTP [63], FDT [11] and others, transfer small files with a rate more than 15X slower than large ones, the details of which are discussed in Section 2.2.

To address this issue, GridFTP scales up the concurrency of storage access and the number of network streams to improve the aggregate data transfer rate. Unfortunately, this method can lead to using a significant fraction of compute resources in a small or medium size cluster just for data movement. For example, in one experiment to transport a 2TB data set with the average file size around 40MB across a 100 Gbps network, 12 data transfer nodes (DTNs) were required to saturate the network bandwidth [63] using multiple GridFTP processes and 500 parallel TCP streams. Each DTN requires a high-performance server with multiple CPU sockets and high speed network adapters. This approach increases data movement complexity and wastes system resources.

Even though solutions like the one posed above can work, the fundamental bottleneck of moving LOSF has not been extensively investigated in literature. We identify the performance is actually constrained by degraded file system throughput, not network bandwidth as might be suspected. Our experiments show most file systems lower performance more than 10 times when decreasing file sizes from 100MB to less than 1MB. We examine the impact of small I/O on transferring files smaller than 1MB using a data transfer pipeline model. We show appropriate engineering approaches that can alleviate the storage bottleneck for moving LOSF. We extended GridFTP to build the pipeline that packs small files into large buffers for transmission and overlaps network transfers, storage reads and writes to maximize performance. We demonstrated the advantage of our extension over existing approaches using both synthetic and real-world workloads. Specifically, this paper presents the following contributions:

- We examined the storage and file system features that constrain moving LOSF;
- We analyzed small file transfer performance using a data transfer pipeline model;
- We demonstrated appropriate engineering approaches that can mitigate the LOSF bottleneck.

The rest of this paper is organized as follows. Section 2 discusses related work and general background. Section 3 reviews the storage bottleneck of accessing small files. Section 4 introduces the data transfer pipeline model and presents our extension of GridFTP. Section 5 examines small file transfer performance. Our conclusions follow in Section 6.

2. Background and motivation

Globally distributed computing infrastructures, such as traditional HPC systems and modern cloud computing platforms, are used to manage data that is generated with an unprecedented speed from a variety of resources. HPC centers are used worldwide to address growing data processing demands for applications such as artificial intelligence (AI) and mission-critical grand challenges. Each major commercial cloud vendor has deployed tens of data centers globally to minimize access time, maximize fault tolerance and manage data sovereignty concerns. As massive amounts of data is manipulated in a geo-distributed manner, moving data across distant sites is inevitable. First, it is often difficult to use one computing platform to serve all stages of the data analysis pipeline [37]. Many scientific applications transfer large volumes of data between distant facilities to handle different stages of data analysis [53][66][12][48][51]. Second, offloading computation onto public clouds normally replicates files across on-premises and cloud datacenters [1][3][60].

To accelerate data transfer across remote sites, high-bandwidth and low-latency networks are provided to connect large computing facilities both within a metro area and across different cities [13][54][57]. For example, the U.S. Department of Energy's Energy Sciences Network (ESnet) provides connections with 100 Gbps bandwidth or more to many science facilities [64]. Various optimizations are also applied to network connections and storage systems to improve the data transfer rate [35].

In spite of the rise of new data storage systems such as object stores [4], most contemporary applications

still access data using files, many of which are smaller than 64KB [7][44]. According to the recent study on data movement for scientific computing, 70% of files being moved across distant facilities are smaller than 1MB [66]. For real-world applications such as bioinformatics, image processing, HPC checkpoint, machine learning training [29] and others, it is common to process large numbers of files around this size. To date, moving a large number of files smaller than 1MB is still a significant challenge [63][64][66].

2.1. An overview of data movement solutions

2.1.1. Solutions for Grid computing

Data movement has been investigated extensively for Grid computing and various tools are available, such as rsync, scp, bbcp [6], Fast Data Transfer (FDT) [11], Globus online [27], XRootD [2][62], and The eXtreme *dd* toolset (XDD) [9]. State-of-the-art solutions for moving large data sets across distant sites, such as GridFTP [61][55], FDT, mdtmFTP [36] and XROOTD, have different strengths of data transfer and small file optimization. XRootD was designed to handle high demand data access for modern High Energy Physics (HEP) experiments, such as the BaBar experiment at Stanford Linear Accelerator Center (SLAC). The XRootD file access system [2] allows users to acces remote data as local files, and supports high speed, fault tolerant, robust, and scalable concurrent data access from multiple clients to petabyte-scale data repositories. GridFTP is an extension of the File Transfer Protocol (FTP) for Grid computing and provides a high-performance and reliable file transfer across a WAN. GridFTP is used within large science projects, such as the Large Hadron Collider, and many supercomputer centers. The most widely used GridFTP implementation is provided by the Globus Toolkit, which is an open-source toolkit for grid computing developed by the Globus Alliance.

Existing approaches mainly focus on increasing network usage for large files as the remote link bandwidth is limited. Specifically, to saturate a high-bandwidth network, these solutions use parallel streams to bridge the data transfer source and sink for moving lots of files. Large files are typically moved using striped and parallel data transmission [61], in which each file is partitioned into smaller chunks and transported simultaneously using multiple data streams. Additionally, GridFTP incorporates automatic renegotiation of TCP buffers and window sizes. FDT supports continuous streaming of a list of files using a managed pool of TCP socket buffers. HARP [17] tunes GridFTP data transfer protocols based on historical data analysis and real-time background traffic probing.

Optimizing LOSF movement over a WAN has been studied to some extent. Similar to moving large files, existing approaches concentrate on using parallel streams to mask the latency of the network and to maximize bandwidth [18][33][64]. Various methods are used to decrease data channel idle time and to tune network transfer protocols for different file lengths. mdtmFTP provides a pipelined I/O-centric architecture on the multi-core platform to transfer data with dedicated threads for both disk and network I/O operations. mdtmFTP also employs a "virtual file" mechanism to reduce the per-file protocol processing overhead. GridFTP provides pipelining [33] to reduce data channel idle times for moving LOSF. GridFTP also supports on-the-fly *tar* [52] to tolerate FTP's low efficiency of handling small files, which packs small files into a large archive file before transferring them.

2.1.2. Solutions for Cloud computing

Existing cloud migration systems face the challenge of transporting directories with millions of small files stored in deep hierarchies from on-premises file systems to the cloud storage [60][63]. A large number of these files are around KBs in size [63]. Cloud migration tools, such as AWS DataSync [15] and Snowball [63][34], mainly focus on moving data from an on-premise site to Amazon S3 [4]. These tools are designed to reduce migration cost for moving small files by shortening data migration time. They actually adopt an approach similar to GridFTP on-the-fly tar feature, in which each small file is batched and archived in a TAR format. A demo of using DataSync to move 10,000 small files [5], around 325MB in total, from NFS to S3, achieved 30MB/s bandwidth. IBM Cloud Migration tool Aspera [28] supports moving directories that contain a large number of 1-10 KB files with more than 62.5MB/s bandwidth over WAN. This low transfer rate cannot efficiently utilize high-speed network connections. Google Storage Transfer Service [24] cannot handle files smaller than 16MB efficiently, and suggests users should pack small files into a single object before moving them into Google

Cloud. This approach only improves network transfer performance. Users need to duplicate large files with additional storage I/O operations, and maintain them using extra storage space. Furthermore, most existing file packing utilities, such as *tar*, do not support concurrent data access. Therefore, this method does not compensate for the performance loss of accessing small files from the file system.

2.2. Motivation

Most existing data movement tools only provide a low transfer rate per DTN [19] for moving small files. In order to saturate high-speed interconnections for moving LOSF, existing solutions, such as GridFTP [64], scale up the number of DTNs to increase the concurrency of storage access and network streams to improve the aggregate data transfer throughput. This approach significantly increases the complexity of data transfer and exclusively dominates an important fraction of system resources.

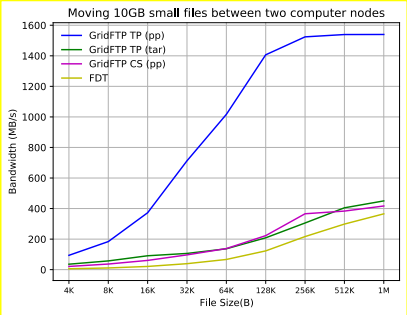


Fig. 1. Moving files using GridFTP TP and CS modes (with pipelining enabled), on-the-fly *tar*, and FDT.

We tested existing data movement solutions with their latest releases, including GCT GridFTP 6.2, mdtmFTP 1.1.1 and FDT 0.24.0, in the FlashLite cluster [14][21]. Each FlashLite compute node has 4.8TB of Intel NVMe SSD DC P3600 [31]. A set of files was transferred from Ext4 to XFS local file systems across two compute nodes connected using 56Gbps InfiniBand. The size of files varies from 4KB to 1MB, and for each size, the total amount of data is fixed to 10GB. Each tool was tested with up to 64 concurrent data transfer streams. In addition, GridFTP and mdtmFTP were examined using both third-party (TP) and client-server (CS) modes, while GridFTP pipelining (pp) was enabled. GridFTP was also tested with its on-the-fly *tar* mode. The optimum results are presented for each file configuration, as shown in **Fig. 1**. Overall, GridFTP TP mode is one of the fastest small file movement solutions. As file size is reduced from 1MB to 4KB, the performance of GridFTP TP mode decreases from around 1.5GB/s to 100MB/s, close to 15 times slower. In contrast, FDT performance degrades roughly 70 times from 350MB/s for 1MB files to around 5MB/s for 4KB size. mdtmFTP failed to complete the 10GB data set due to memory faults, but it handles a smaller scale of data set, such as 1GB, with a performance similar to GridFTP.

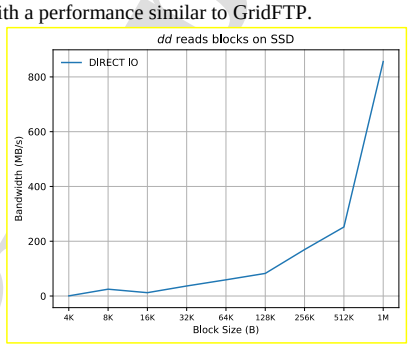


Fig. 2. The storage I/O performance of accessing SSDs.

3. The bottlenecks of moving small files

End-to-end data transfer in a wide area environment involves many components along the path from the source to the destination, which at least include storage systems (at both ends), and networks in between. An optimized data movement mechanism requires an efficient interaction between storage systems and network connections, which requires the adjustment of many system parameters, such as the socket buffer size, the number of parallel network streams and the concurrency of storage I/O accesses.

Specifically, the following system-level activities across the data transfer path impact moving LOSF: 1) reading meta-data of a large directory at the source end; 2) reading file data for a large number of small files; 3) transferring small files using TCP; 4) the inefficiency issue of using TCP to transport data over long-distance networks; 5) creating small files at the

destination side. The low efficiency of using TCP to move data has been addressed in most existing tools. Importantly, all tools examined by us have already been shown to be efficient on high latency networks [16] and we do not further investigate this aspect. Accordingly, in this paper we focus on the impact of network and storage I/O bottlenecks exclusively.

To address small network I/O issues, we propose to pack small files into large memory buffer before moving them through networks, in contrast to other utilities that concatenate the files into larger ones. In this section, we mainly identify the impact of small storage I/O. Accessing small files is typically a meta-data intensive activity, and a bottleneck is created by both the storage device characteristics and the file system management techniques. Presently, solid-state drives (SSD) and hard disk drives (HDD) are popular choices of storage devices for constructing data centers or storage servers. Both devices favor larger size I/O requests instead of small ones. NAND flash memory-based SSDs provide a much lower latency than HDDs, and typically perform orders of magnitude faster for random I/O. State-of-the-art SSDs package multiple components that increase the internal parallelism and speed up large I/O operations. To takea advantage of this architectural improvement, concurrent data access is required to handle a large number of requests that are smaller than the size of a page or a block [42], as discussed later with **Fig. 3**.

Fig. 2. shows the performance gap between large and small I/O operations using the Unix *dd* tool to access an SSD device. In particular, *dd* reads a data block from devices into memory on a compute node of FlashLite while varying the data block size. The block sizes range from 4KB to 1MB. Even with *direct_io* enabled, blocks between 4KB and 128KB are around 10~100 times slower than those around 1MB. Specifically, the following command was used to read a block of the SSD device file, in which ${SSD_DEV}=/dev/nvme0n1p1 on FlashLite and ${BlockSize} increased from 4KB to 1MB.

dd if=${SSD_DEV} of=/dev/null bs=${BlockSize}

Normally, file systems are tuned for accelerating common access patterns, such as large I/O operations, at the expense of other special workloads. Many file systems provide improvements to compensate for small file performance degradation by mitigating the meta-data overhead. Ext4 [20], XFS [56] and Btrfs [10], use tree-based data structures to manage directory entries, and very small files can be contained in *i*-nodes. However, small files are still processed less efficiently than large ones, as shown in **Fig. 3**. The smallest file size examined in this paper is 4KB, which represents a pathological case to study. Mounting a file system on Linux requires the data block size not be larger than a 4KB memory page.

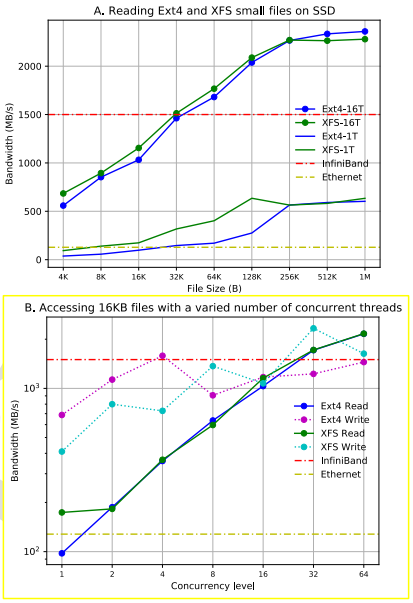


Fig. 3. Accessing small files on SSDs using Ext4 and XFS.

Typically, each data transfer is acknowledged after the file descriptor is closed at the destination end, and received data is flushed to system buffers, which is subsequently synchronized to storage by Linux in background. To examine SSD storage bottleneck, we collected the performance of accessing small files using concurrent threads, from Ext4 and XFS file systems respectively. For each size, the total amount of data is fixed to 10 GB and uniformly distributed to 64 sub-directories. Workloads are evenly partitioned across threads, and reading files follows the order returned by *getdents*() system call. The optimal performance is achieved using multiple I/O threads to take advantage of the NVMe command queue, as shown in **Fig. 3**. The reading results of 16 threads (16T) and a single thread (1T) are presented in **Fig.**

3.A. **Fig. 3**.B illustrates reading and writing 16KB files while adjusting the number of threads. When the concurrency level is smaller than 16, writing is faster than reading because data is committed to Linux system buffer instead of SSD storage. When the concurrency level increases more than 16, file system locking constrains writing performance, and this makes writing slower than reading. **Fig. 3** also compares storage performance with network bandwidths for FlashLite local InfiniBand (IB) and Ethernet connections. The comparison provides high-level guidance on detecting the bottleneck of LOSF movement. It drives us to create a performance model to analyze the bottleneck of moving small files.

In case a single storage node cannot provide sufficient I/O throughput to saturate the network, multiple nodes are required, for example by using a parallel file system. However, network-based file systems, such as parallel file systems and distributed file systems, exacerbate the small I/O bottleneck, because each client requests target files sequentially and experiences the extra path of the storage network. Typically, several rounds of communications between front-end clients and back-end storage and meta-data servers are required to fetch a single file and the overhead is not amortized for small files. Recently, some distributed file systems such as HDFS [49] and TýrFS [47], improved small file access. Typically, these optimizations attach small files with their meta-data and maintain them on meta servers instead of data servers. Most of these optimizations are conceptually similar to those approaches adopted by local file systems, such as Ext4 and XFS, that attach tiny files to *i*-nodes. These approaches are mainly designed for active data sets and not appropriate for data migration which normally consists of a large amount of cold data. A recent related work by our co-authors [64] has examined using GridFTP to move data across parallel file systems. This paper focuses on using local file systems to verify the storage I/O bottleneck of small file movement with our performance model.

4. A performance model for moving small files

We built a data transfer pipeline model to investigate the performance of small file transmission. To simplify analysis without loss of generality, this model assumes: 1) small files are packed into large chunks before network shipment, and 2) network transfers are overlapped with storage access.

Data packing can reduce per file network transfer overhead and maximize network usage, and a similar approach is adopted in mdtmFTP [36]. The pipeline idea has been used widely in many domains to improve overall system efficiency. For example, in computer architecture, pipeline processing splits instructions into several stages and multiple instructions are overlapped during execution. Increasing pipeline depth improves CPU performance. Many parallel algorithms are designed to overlap CPU time with storage and network I/O access to achieve an optimal performance. We apply the same concept to investigate the optimal options of small file transfer.

Table 1. Parameters used in the model.

Parameters	Description
L	The total size of all the files
S	File size
$R_{Max}(S)$	The maximum rate of storage read for size S
$W_{Max}(S)$	The maximum rate of storage write for size S
N_{Max}	The maximum rate of network transfer
BDP	Network bandwidth delay product
p	The number of parallel transfer streams
l	The length of a memory buffer
n	The number of memory buffers
$T_R(l)$	Storage read time of a memory buffer
$T_W(l)$	Storage write time of a memory buffer
$T_N(l)$	Network transfer time of a memory buffer
$T_M(l)$	The maximum time of 3 pipeline stages
$T_{Total}(L)$	The total time of data movement

4.1. The performance model

Given a set of small files with the total size L, small files are packed into n memory buffers and the length of each buffer is l ($BDP \leq l \leq L$ and $L = n * l$) . Transferring each buffer as a normal file naturally leverages the existing optimized mechanism of moving large files. Transporting each memory chunk consists of 3 stages: storage read, network transfer, and storage write, which are overlapped using the pipeline. The time of each stage is denoted as $T_R(l)$, $T_W(l)$ and $T_N(l)$ respectively for storage read and write and

network transfer. $T_{Max}(l)$ represents the maximum one of these three stages. The total data transfer time is shown in equation (1). **Fig. 4**.A illustrates an ideal case in which $T_R(l) = T_W(l) = T_N(l)$.

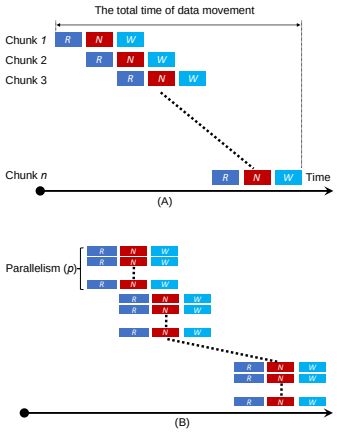


Fig. 4. I/O overlapping of data transfers.

$$T_{Total}(L) = T_R(l) + T_W(l) + n * T_{max}(l) \quad (1)$$

Combining parallel data streams with the pipeline forms a parallel data transfer pipeline, as shown in **Fig. 4**.B. Given p parallel streams, the data transfer is parallelized, and the total transfer time is shown in equation (2). The speed up of the data transfer pipeline is illustrated in equation (3). When the number of chunks is large enough and 3 stages consume a similar time, the pipeline achieves up to $3p$ performance improvement. The parallelism level and the data packing size can be adjusted to optimize the aggregate data transfer rate. Reducing the total transfer time needs to minimize the maximum one of the three stages. Using this model, we analyze optimization options that mitigate the bottleneck for moving data across computers.

$$T_{Total}(L) = T_R(l) + T_W(l) + (n/p) * T_{max}(l) \quad (2)$$

$$Speedup = \frac{(T_R(l)+T_W(l)+T_N(l))*n}{T_R(l)+T_W(l)+(n/p)*T_{max}(l)} \leq (3 * p) \quad (3)$$

4.2. Moving data between computers

Given a network link that connects two computers has a fixed bandwidth, the maximum achievable network bandwidth is denoted as N_{Max}. According to our previous experiments, the storage performance is proportional to file size S. For a given file size S, the maximum storage read rate at the source end and write rate at destination are represented by $R_{Max}(S)$ and $W_{Max}(S)$ respectively. Note that the source and destination ends may install different file systems. The bottleneck of data transfer is illustrated in equation (4).

$$\begin{cases} Network\ bottlenecks,\ if\ N_{Max} < min\{R_{Max}(S),\ W_{Max}(S)\} \\ Storage\ bottlenecks,\ if\ N_{Max} > min\{R_{Max}(S),\ W_{Max}(S)\} \end{cases} \quad (4)$$

The threshold file size, as defined in equation (5), identifies which system bottleneck constrains data movement. To move files smaller than the threshold size, only optimizing network performance cannot compensate for the degraded storage performance. Typically, escalating network bandwidth also increases the threshold size.

$$S_{Threshold} = S,\ if\ N_{Max} = min\{R_{Max}(S),\ W_{Max}(S)\} \quad (5)$$

Normally, committing files to storage devices is much slower than reading them. At the destination end, the workload of moving small files is typically a burst small data writing pattern. Many approaches have been proposed to optimize this pattern, such as a burst buffer [40], log structured file systems [43], and a virtual file system [32]. Some of these techniques [41][58] provide an in-memory store to accelerate small write operations. In our experiments, memory capacity on each compute node is large enough to accommodate received files. At the source end, reading performance is normally accelerated using prefetching and caching, which are not efficient enough to eliminate the bottleneck of LOSF movement.

Up to now, our model mainly focuses on optimizing performance in a stable environment. However, transferring data using shared resources, such as a parallel file system in a HPC center and remote network connections across a metro area, may experience considerable performance variability. Existing adaptative approaches [17] tune data transfer performance by adjusting data transfer parameters, such as the number of parallel streams and concurrent files, according to real-time background traffic and historical I/O workloads analysis. Implementing our model is orthogonal these adaptive approaches.

4.3. Performance model implementation

To analyze small file movement performance in a cluster environment, we extended GridFTP to support the data transfer pipeline. Our extension is based on the Globus implementation of GridFTP [26] with version GCT 6.2 and focuses on improving small file transmission between two storage servers.

4.3.1. GridFTP overview

To achieve optimum performance, GridFTP runs on Data Transfer Nodes (DTNs) [19], which are compute systems dedicated for wide area data movements. GridFTP supports two modes of data transfer: the client-server mode (CS) and third-party mode (TP). With the CS mode, the client transfers data to and from the remote server. Using the TP mode, the client controls moving data between two servers. Typically, GridFTP optimizes data movement by appropriately adjusting 3 parameters: *pipelining* (*-pp*), *parallelism* and *concurrency* (*-cc*). The architecture of GridFTP is illustrated in **Fig. 5**.

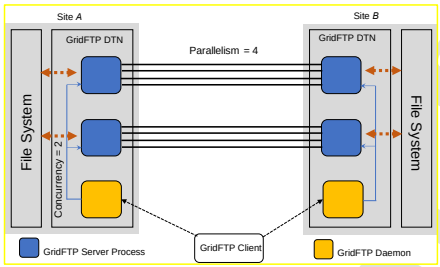


Fig. 5. GridFTP architecture with the third-party mode.

GridFTP concurrency enables transferring multiple files simultaneously. Each level of concurrency launches a separate process of the GridFTP server, and one process only handles a single file at a time. GridFTP parallelism supports splitting a single file across multiple TCP streams to optimize the network bandwidth utilization. Parallelism helps improve the performance for transferring large files. GridFTP pipelining increases the efficiency of transferring a large number of small files using a long round-trip time (RTT) network by reducing data channel idle times and avoiding shrinking the TCP window size. When the size of a single file is less than network BDP, waiting for an acknowledgment for each transfer before starting the next one leads to idleness and lowers the utilization of network bandwidth. Pipelining enables issuing multiple outstanding transfer commands to the same data channel to improve network utilization for moving small files. Typically, combining concurrency and pipelining optimizes the performance of moving lots of small files. Apart from these three parameters, GridFTP also supports on-the-fly tar [52] to tolerate FTP's low efficiency of handling small files.

Note that GridFTP pipelining is orthogonal to the data transfer pipeline which we built to minimize small I/O impacts on data movement.

4.3.2. GridFTP extension

The Grid Community Toolkit provides GridFTP implementation with 1) a server implementation called *globus-gridftp-server*; 2) a scriptable command line client called *globus-url-copy*; and 3) a set of development libraries. GridFTP third-party (TP) mode moves data between two end hosts or sites, mediated by a third host, using separated control and data channels. GridFTP TP mode supports moving data using multiple data transfer node (DTNs) per site. Its implementation achieves a high performance with sophisticated engineering efforts, as shown in **Fig. 1**. GridFTP TP transfer parallelizes moving a set of files using multiple processes. In contrast, its client server (CS) mode copies files between a remote host and the local storage. GridFTP client adopts an asynchronous event driven model [22] to parallelize FTP and storage I/O operations.

We aim to demonstrate the effectiveness of data transfer pipeline for the scenario of moving data between two hosts. The data transfer pipeline is realized by extending the GridFTP client-server mode to save engineering overhead. There are different approaches to implement the pipeline. We used a threading model to mask the latency of storage access with network transfers. Specifically, to overlap transferring data via FTP and accessing files for each concurrency, a dedicated thread is added for each concurrency to read small files and aggregate them in memory buffers. Each memory buffer contains both data and meta-data for each file. Each aggregated buffer is delivered to the FTP thread, *i.e.*, the main thread of the client, which is responsible for sending

data over sockets. When each buffer is received at the destination end, the thread unpacks received data to regenerate the original files.

A preparation stage parses metadata for the target folder and to detect file size for all the data by invoking *stat*() on each file. This stage is parallelized using concurrent I/O threads and fetched meta-data is cached automatically by the Linux Virtual Filesystem (VFS) directory entry cache (*dcache*) and *i*-node cache (*icache*) for subsequent file data access. Therefore, this stage incurs almost no extra overhead. Different from the original GridFTP that uses *readdir*() POSIX function to traverse a directory to compose the list of file names to be moved, we called *getdents*() system call to save meta-data I/O operations using a large memory buffer. Workloads are distributed across I/O threads uniformly and the files of the same directory are assigned to a single thread with the best effort. A command line argument, *-pbs*, is provided to allow users to specify the size of packing buffer. Please refer to the Appendix for more details of our implementation with an example. Although our extension only enhanced GridFTP CS mode, the same optimization approaches are also applicable to TP mode.

5. Performance analysis

We examined the LOSF bottleneck by testing our data transfer pipeline extension and GridFTP GCT6.2 version on FlashLite [14]. Each FlashLite compute node contains large amounts of main memory and high-speed secondary storage, SSDs and is equipped with high performance networking, such as Dual rail 56Gbps Mellanox InfiniBand (IB) fabric. The CentOS 7 operating system, with kernel version 3.10.0-693, is installed on each node. Our experiments reserved two compute nodes that manage local SSDs using Ext4 and XFS respectively. Specifically, each node has the following configuration:

- 2 x Xeon 2.5GHz 12 core Haswell processors;
- 512GB DDR memory (256GB per socket);
- 3 x 1.6 TB Intel P3600 NVMe drives of storage;
- 2 x Mellanox 56Gb/s FDR InfiniBand adapter.

As shown in **Fig. 1**, our previous experiments detect that GridFTP performance starts decreasing for files smaller than 256KB in the same environment. This indicates that BDP is roughly 128KB~256KB, and this is consistent with the perceived latency and bandwidth of FlashLite IB connection, which are around 0.125ms and 1.5GB/s. Therefore, we focused on files ranging from 4KB to 128KB, with the same data set used in Section 2 and 3. For all the experiments, vanilla GridFTP third-party mode and our extension are compared using the same level of parallelism, which was always set to 1, and pipelining was always enabled unless otherwise stated. Concurrency was varied between 1 and 64. The size of data packing buffer was adjusted between 1~20MB. Each experiment was repeated at least 3 times, depending on variations observed, and Linux page cache, *dcache* and *icache* were cleared for each run and the files moved to the destination end were deleted before the next run. The averaged results with standard deviation are reported.

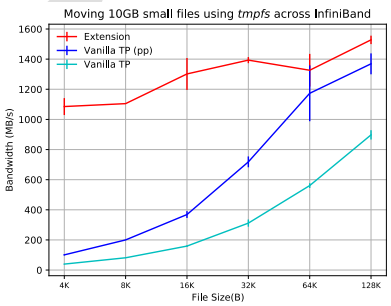


Fig. 6. The performance of transferring small files using *tmpfs*.

5.1. Network efficiency

We first verified the network efficiency by moving files between two compute nodes using *tmpfs*, which is a temporary file system that stores data in volatile memory instead of a persistent storage device. Using *tmpfs* on FlashLite, 16 concurrent I/O threads achieve 3.5GB/s bandwidth for 4KB reads and 32 GB/s for 1MB files, the speed of which is fast enough to match the perceived 1.5GB network bandwidth. GridFTP vanilla was also examined without enabling *-pp* (i.e., GridFTP *pipelining*) to provide a baseline for examining network efficiency.

Fig. 6 shows the maximum performance for each size with 3 modes. When pipelining is enabled, the

network efficiency of GridFTP vanilla is improved significantly for 128KB files, almost reaching the top level of network performance. However, as the file size decreases to 4KB, the transfer rate of vanilla pipelining reduces more than 10 times to around 100MB/s. In comparison, the extension version outperforms both vanilla versions and achieves around 1.1GB/s even for 4KB files. This comparison illustrates that packing small files into a large buffer can efficiently optimize the per file overhead of network transfer protocol.

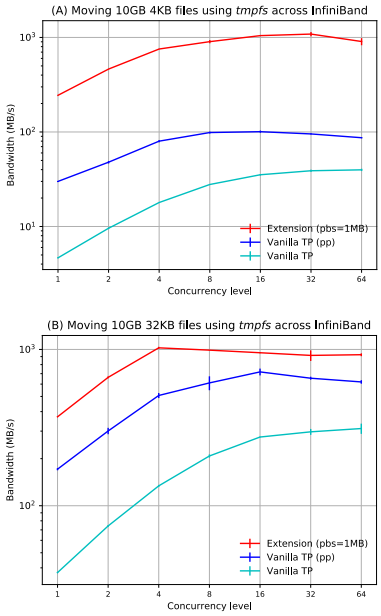


Fig. 7. The performance of moving 4KB and 32KB files using *tmpfs*.

Fig. 7 compares the data transfer performance while adjusting the concurrency level for 4KB and 32KB files. The performance of vanilla versions tops with around 8~16 concurrency levels. The superiority of the extension decreases gradually with the increased file size. For 32KB files shown in **Fig. 7**.B, the extension only scales up to 4~8 concurrency levels. Although our extension has significantly improved GridFTP CS mode performance, its scalability is constrained by CS mode internal structure. As we discussed earlier, GridFTP client parallelizes data transfer using asynchronous events. However, this approach sequentially processes many small events, such as creating a file name and reversing storage space remotely. These small events accumulate for a large number of files to impact scalability negatively.

The size of the packing buffer is also critical to achieve the optimal performance. Typically, increasing the buffer size improves network efficiency by reducing per file transfer protocol overhead. However, a buffer size that is too big decreases data movement efficacy because a large pipeline tail and head decrease storage and network overlap. Using a single concurrency level, the size of the packing buffer was examined for different file lengths, as shown in **Fig. 8**. When the buffer size increases from 1MB to 20MB, the performance peaks with a 16MB buffer for the 10GB data set.

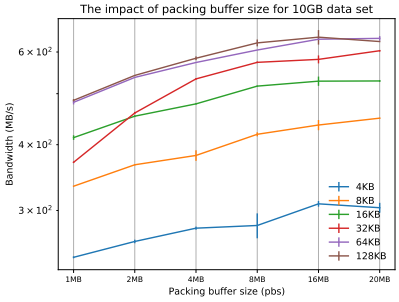


Fig. 8. Packing buffer size evaluation.

5.2. Storage constraints

As shown in Section 3, Ext4 and XFS provide similar reading performance. However, writing small files using XFS is faster than Ext4 with a high level of concurrency. We add storage constraints to the data transfer pipeline by moving files from Ext4 to XFS across two compute nodes. This configuration can highlight the storage efficiency of our extension. The results of maximum performance for each mode are displayed in **Fig. 9**. The vanilla version received less impact from storage constraints, while the extension version performs better overall. The performance degradation of the extension is caused mainly by two factors: the decreased storage performance and the CS

mode scalability issue. The latter cause makes 128KB files perform slower than the vanilla version.

The data transfer rate and file system performance were compared in **Fig. 10**, while adjusting the concurrency level from 1 to 64 for 4KB and 8KB files respectively. The extension is tightly bounded by the Ext4 reading curve when *cc* <= 16 and the XFS writing cure when *cc* > 16. In comparison, the vanilla version doesn't efficiently utilize the increased storage I/O bandwidth. This demonstrates the storage I/O efficiency of the data transfer pipeline. However, the vanilla version cannot scale to more than 16 streams, as shown in Sec 5.1. The extension curve goes flat in **Fig. 10** due to the storage constrains.

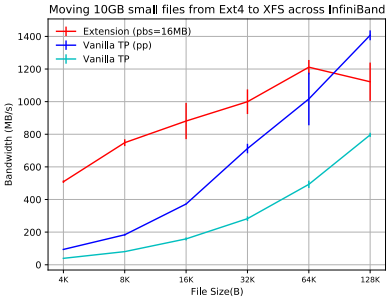


Fig. 9. The performance of the GridFTP extension.

5.3. A real-world case study

To study a real-world case, we tested moving a folder of Linux source code containing ~68,000 files and 4,532 sub-directories, with 898MB total amount of data. The median file size of the folder is 3,808 bytes. Existing tools were examined by moving the Linux folder from Ext4 to XFS across two compute nodes. The optimal performance achieved by each tool and associated configurations are shown in **Table 2**.

With the on-the-fly tar feature, GridFTP packs small files into a single archival file at the source end before transferring them, while the destination end extracts small files after receiving the archive. The parameter studies for this feature was performed in the previous literature [52]. Overall, this feature is enabled using the Globus XI/O Pipe Open (Popen) Driver [52] that leverages Unix pipes to connect *tar* and GridFTP commands in a single command line. All versions of GridFTP were examined with up to 64 concurrency levels and a 4MB TCP buffer (*-tcp-bs=4096000*) was used. FDT was tested using maximally 64 parallel streams and its server specified a 4MB I/O buffer (*-bs=4M*). mdtmFTP separates network threads from storage threads. We tested mdtmFTP with up to 8 parallel streams, 8 storage threads and 8 network threads.

As shown in **Table 2**, our GridFTP extension is the fastest option and it finished the movement using around 1.5 seconds with 16 concurrent threads and a 1MB package buffer. This performance result corresponds to around 600MB/s bandwidth and is more than 4 times faster than GridFTP vanilla third-party mode with pipelining enabled.

The data transfer pipeline extension also outperforms GridFTP on-the-fly tar mode, which uses a large archival file to move LOSF. We guess that the performance superiority is created by two reasons. First, our extension overlaps network transfer with storage accesses. Second, *tar* does not support parallel data streaming to access small files.

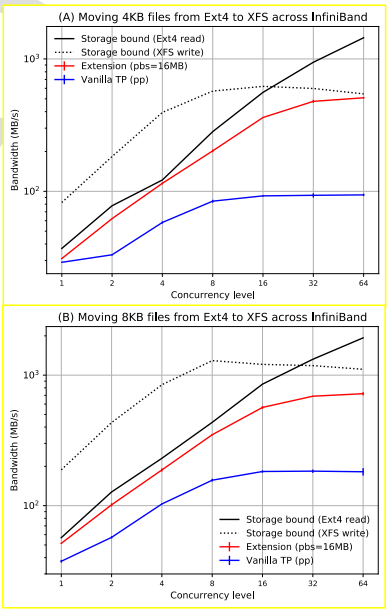


Fig. 10. The performance of moving 4KB and 8KB files.

Table 2. The performance of moving a Linux source folder.

Transfer tools	Transfer mode	Time(s)	Configurations	DTN count needed to saturate a 100Gbps link
GridFTP extension	Client-server	1.5	16 concurrent threads, and 1MB packing buffer.	21
GridFTP vanilla	Third-party (pp)	6.5	64 concurrent threads.	91
GridFTP vanilla	On-the-fly-tar	8.66	16 concurrent threads.	121
mdtmFTP	Third-party	11.03	4 disk I/O threads, 4 network I/O threads, and 8 parallel streams.	154
GridFTP vanilla	Client-server	15.13	64 concurrent threads.	211
FDT	Client-server	45	8 parallel streams.	627

5.4. Estimated Data Transfer Resource Optimization

Based on the data transfer improvement, we estimate the optimal resource usage for moving files between storage clusters. Presently, the data transfer rate across large-scale computing facilities is often more than 100 Gb/s [64]. Typically, each computing center uses a parallel file system, such as IBM Spectrum Scale [30] or Lustre [37], to provide a scalable storage capacity and an aggregated I/O bandwidth. Normally, one or several DTNs are used to relay data between the backend storage cluster and the remote network connection. A typical scenario of moving files across storage clusters is illustrated in **Fig. 11**. Normally, large files are striped across multiple storage servers to accelerate parallel data access. However, small files are not partitioned if they are smaller than the size of a partition block. Therefore, the set of small files to be moved should spread across an enough number of storage servers to provide an aggregated I/O performance that matches the target network bandwidth.

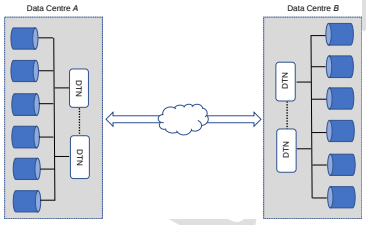


Fig. 11. Data movement across two data centers.

We assume the backend parallel file system provides a performance scalable in the number of clients. The maximum storage read and write rates achieved by each client, *i.e*. a DTN, are represented as $R_{DTN}^{max}(S)$ and $W_{DTN}^{max}(S)$ respectively, which are also proportional to file size S according to our experiments. An optimal data transfer performance can be achieved only when the aggregated DTN relay rate matches the network bandwidth. The number of DTNs required to optimize the network bandwidth is calculated using equation (6):

$$n_{DTN} = \max\{N^{max}/R_{DTN}^{max}(S), N^{max}/W_{DTN}^{max}(S)\} \quad (6)$$

Overall, moving smaller files need more DTNs to optimize network usage. To decrease system resource consumption, a straightforward approach is to increase the performance of each I/O thread. In comparison to GridFTP TP mode, our extension achieves the performance improvement while using 4 times fewer concurrent threads. Mapping this advantage to a scenario of using multiple DTNs, our optimization is able to improve data transfer rate per DTN significantly. If we assume each compute node acts as a DTN, we estimated the total number of DTNs required for each solution to saturate a 100Gpbs link, as shown in the right most column of **Table 2**. Theoretically, our approach uses around 4~5 times fewer system resources than GridFTP TP mode.

6. Conclusions and future work

Previous research on moving lots of small files has mainly focused on optimizing network performance.

We identify that moving files smaller than 1MB across existing high-speed remote network links is mainly constrained by file system throughput. Perhaps with hindsight this is obvious, but it is not widely promoted as a performance constraint. We built a data transfer pipeline model by extending GridFTP to examine the small I/O impact on data movement. We demonstrated several engineering solutions, including packing small files into a large memory buffer and overlapping network transfer with storage read and write, that can alleviate the storage bottleneck.

The pipeline extension improves the small file transfer rate for more than 5 times in comparison to existing approaches. We believe our extension can optimize transferring small files in two aspects. First, it increases the data transfer performance per DTN. Given a fixed number of DTNs, the aggregate transfer rate is improved accordingly. Second, to saturate the bandwidth of a given network connection, it requires around 4~5 times fewer number of DTNs for moving small files.

Our experiments have demonstrated the advantage of our approach by interacting with local file systems using a typical POSIX file system interface, mainly for Grid computing scenarios. Theoretically, our design should work with other storage systems that support the same interface, such as cloud file systems and pareallel file systems, probably with a minimum engineering extension. Although our approach doesn't directly handle cloud storage objects due to the different storage format and access interface, we believe the principle of overlapping I/O operations and packing small data using large memory buffers should also benefit moving data in cloud storage systems. Most parallel file systems, distributed file systems, and cloud storage systems are built on top of a cluster of storage nodes. Each storage node is typically managed using a local file system, such as Ext4, XFS, Btrfs and ZFS. Cloud storage objects and parallel files are normally managed using these local file systems. For example, the default configuration of BeeGFS [7] manages file data using XFS and organizes meta-data using Ext4. Specifically, data chunks of large files, small files and meta-data of each file and directory are stored as individual files in local file systems. Other systems, such as Lustre [37] and OpenStack Swift [45], adopt a similar policy to map data to native file systems. To support applications that rely on the POSIX file system interface in clouds, cloud file systems, such as s3fs-fuse [49] and Goofys [24], map Amazon S3 objects to files. For these storage clusters, the local file systems characterize their fundamental storage I/O behavior.

However, our approach still cannot fundamentally eliminate the bottleneck caused by small storage I/O. In order to address this issue, our future work will explore the feasibility of optimizing LOSF movement from the file system layer by using large I/O requests to improve small file access. Many scientific applications organize a large number of small files using directories. Moving small files is normally applied to the granularity of folders and the data access pattern of which is not random. Local file systems, such as Ext4, already places many small files into contiguous storage regions. Acquiring those large contiguous regions can be achieved by using specific tools or by extending the file system with new APIs to allow access to contiguous blocks. This approach would significantly improve I/O efficiency for accessing lots of small files. Therefore, to eliminate the small file bottleneck fundamentally for data movement, we need to extend the large space allocation policy to a whole folder. This would increase data contiguity by concatenating all of its small files, including both file data and meta-data, into large contiguous regions. Additionally, extending the file system to provide new APIs would allow users to access these contiguous blocks as a whole, instead of reading a large number of small files one by one. This feature would fundamentally eliminate the storage bottleneck of data movement. We believe that accelerating the access pattern of traversing a whole directory will not only improve data movement, but also benefit more general applications, such as image processing, bioinformatics, and machine learning training, to name a few.

Acknowledgments

The FlashLite computer system is provided by the Research Computing Centre at the University of Queensland (UQ) and was originally funded as part of an Australian Research Council Linkage Infrastructure Equipment grant. We thanks Justin Luong for his assistance in this work. We also thank our UQ

colleagues Irek Porebski, Sarah Walters, Ashley Wright and Michael Mallon for their assistance in this work. We also acknowledge our colleagues at IBM, Andrew Beattie and Grant Smith, and their collaboration in the IBM Centre of Excellence and Innovation at the University of Queensland, particularly for the provision of the Carlo system.

References

[1] Abhishek Gupta, Rick Spillane, Wenguang Wang, Maxime Austruy, Vahid Fereydouny, and Christos Karamanolis. Hybrid Cloud Storage: Bridging the Gap between Compute Clusters and Cloud Storage. ACM SIGOPS Operating Systems Review. Volume 51Issue 1August 2017

[2] Alvise Dorigo, Peter Elmer, Fabrizio Furano, and Andrew Hanushevsky. XRootD - A Highly Scalable Architecture for Data Access. WSEAS Transactions on Computers 4(4):348-353, April 2005.

[3] Amazon Cloud Migration, https://aws.amazon.com/cloud-migration/

[4] Amazon S3, https://aws.amazon.com/s3/

[5] Amazon Web Services, Accelerate the Migration of 10,000 Small Files From NFS to Amazon S3 Using AWS DataSync. Retrieved April 8, 2022 from https://www.youtube.com/watch?v=IHMZT2S_XuI

[6] BBCP. Retrieved Apr 18, 2022 from http://www.slac.stanford.edu/~abh/bbcp/.

[7] BeeGFS. Retrieved Aug 18, 2021 from https://www.beegfs.io/

[8] Brent Welch and Geoffrey Noer. Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions. In Proceedings of IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST '2013), 2013.

[9] Bradley W. Settlemyer, Jonathan D. Dobson, Stephen W. Hodson, Jeffery A. Kuehn, Stephen W. Poole, and Thomas M. Ruwart. A technique for moving large data sets over high-performance long distance networks. In Proceedings of 27th Symposium on Mass Storage Systems and Technologies, pages 1-6, May 2011.

[10] Btrfs. Retrieved Apr 18, 2022 from https://btrfs.wiki.kernel.org/index.php/Main_Page

[11] Cern. FDT. Retrieved Apr 18, 2022 from http://monalisa.cern.ch/FDT/

[12] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. Wide-Area Analytics with Multiple Resources. In Proceedings of the Thirteenth EuroSys Conference (EuroSys '18), April 2018.

[13] Chongjin Xie. Datacenter optical interconnects: Requirements and challenges. In Proceedings of 2017 IEEE Optical Interconnects Conference (OI), 2017.

[14] David Abramson. FlashLite: a high-performance machine for data intensive science. In Proceedings of IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), Melbourne, VIC, Australia, 14-17 December 2015. Piscataway, NJ, United States: IEEE - Computer Society. doi: 10.1109/ICPADS.2015.1

[15] David DeLuca. Synchronizing your data to Amazon S3 using AWS DataSync. AWS Storage Blog. 24 AUG 2021. Retrieved April 8, 2022 from https://aws.amazon.com/blogs/storage/synchronizing-your-data-to-amazon-s3-using-aws-datasync/

[16] Deepak Nadig, Eun-Sung Jung, Rajkumar Kettimuthu, Ian Foster, Nageswara S.V. Rao, Byrav Ramamurthy. Comparative Performance Evaluation of High-performance Data Transfer Tools. In Proceedings of the 2018 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS).

[17] Engin Arslan, Kemal Guner, and Tevfik Kosar. HARP: Predictive Transfer Optimization Based on Historical Analysis and Real-time Probing. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16), 2016.

[18] Esma Yildirim, JangYoung Kim, and Tevk Kosar. How GridFTP pipelining, parallelism and concurrency work: A guide for optimizing large dataset transfers. In Proceedings of SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC'2012), pp. 506-515, 2012, IEEE.

[19] ESnet, Science DMZ: Data Transfer Nodes, Retrieved Apr 18, 2022 from https://fasterdata.es.net/science-dmz/DTN/

[20] Ext4 Data Structures and Algorithms. Retrieved Apr 18, 2022 from https://www.kernel.org/doc/html/latest/filesystems/ext4/index.html

[21] FlashLite. Retrieved Apr 18, 2022 from https://rcc.uq.edu.au/flashlite.

[22] Globus Toolkit 6.0 Developer's Guide. Retrieved Apr 18, 2022 from https://software.xsede.org/gt6/docs/appendices/developer/index.html

[23] Globus Toolkit 6.0 Release Manuals. Retrieved Apr 18, 2022 from https://software.xsede.org/gt6/docs/

[24] Goofys. Retrieved Apr 18, 2022 from https://github.com/kahing/goofys

[25] Google Storage Transfer Service. Retrieved Apr 18, 2022 from https://cloud.google.com/blog/products/storage-data-transfer/best-practices-for-large-scale-migrations-to-google-cloud

[26] Grid Community Toolkit. Retrieved Apr 18, 2022 from https://github.com/gridcf/gct

[27] Ian Foster, "Globus Online: Accelerating and Democratizing Science through Cloud-Based Services," IEEE Internet Computing, vol. 15, no. 3, pp. 70,73, May-June 2011

[28] IBM Aspera, IBM Aspera Direct-to-Cloud Storage. IBM Cloud Technical Whitepaper. Dec 2018.

[29] IBM Research Editorial Staff. Efficient Deep Learning Training on the Cloud with Small Files. Retrieved Apr 18, 2022 from https://www.ibm.com/blogs/research/2018/12/training-cloud-small-files/

[30] IBM Spectrum Scale. Retrieved Aug 18, 2022 from https://www.ibm.com/docs/en/spectrum-scale

[31] Intel® SSD DC P3600 Series Spefication. Retrieved Apr 18, 2022 from https://ark.intel.com/content/www/us/en/ark/products/80995/intel-ssd-dc-p3600-series-2-0tb-2-5in-pcie-3-0-20nm-mlc.html

[32] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09), 2009.

[33] John Bresnahan, Michael Link, Rajkumar Kettimuthu, Dan Fraser, and Ian Foster. Gridftp pipelining. In Proceedings of TeraGrid '2007, 2007.

[34] Ju-Lien Lim and Vinod Pulkayath. Best practices for accelerating data migrations using AWS Snowball Edge. AWS Storage Blog. 27 NOV 2020. Retrieved April 8, 2022 from https://aws.amazon.com/blogs/storage/best-practices-for-accelerating-data-migrations-using-aws-snowball-edge/

[35] Lawrence Berkeley National Laboratory. ESnet's Petascale DTN Project Speeds up Data Transfers between Leading HPC Centers, 2017. Retrieved April 8, 2022 from https://cs.lbl.gov/news-media/news/2017/esnets-petascale-dtn-project-speeds-up-data-transfers-between-leading-hpc-centers/.

[36] Liang Zhang, Wenji Wu, Phil DeMar, and Eric Pouyoul. mdtmFTP and its evaluation on ESNET SDN testbed. Future Generation Compter Systems, 79: 199-204, 2018.

[37] Lustre. Retrieved Aug 18, 2021 from https://www.lustre.org/

[38] National Research Council, Frontiers in Massive Data Analysis, ISBN: 978-0-309-28778-4, The National Academies Press, 2013.

[39] Marcin Krotkiewski. Dealing with small Files in HPC Environments: automatic Loop-Back Mounting of Disk Image. Partnership for Advanced Computing in Europe Report. 2017.

[40] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the Role of Burst Buffers in Leadership-Class Storage Systems. In Proceedings of 28th IEEE Symposium on Mass Storage Systems and Technologies (MSST '12). IEEE, San Diego (2012).

[41] Memcached. Retrieved Apr 18, 2022 from https://memcached.org/

[42] Myoungsoo Jung and Mahmut Kandemir. Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications. In Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '13). 2013. Pages 203-216, https://doi.org/10.1145/2465529.2465548

[43] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP), 1991.

[44] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In Proceedings of 2008 USENIX Annual Technical Conference (USENIX '08), 2008.

[45] OpenStack Swift. Retrieved April 8, 2022 from https://docs.openstack.org/swift/latest/

[46] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-File Access in Parallel File Systems. In Proceedings of 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09), Rome, 2009.

[47] Pierre Matri, María Pérez, Alexandru Costan, and Gabriel Antoniu. TýrFS: Increasing Small Files Access Performance with Dynamic Metadata Replication. In Proceedings of 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2018), Washington, 2018.

[48] Raajay Viswanathan, Ganesh Ananthanarayanany, and Aditya Akella. CLARINET: WAN-Aware Optimization for Analytics Queries. In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16), November 2016.

[49] S3fs-fuse. Retrieved April 8, 2022 from https://github.com/s3fs-fuse/s3fs-fuse

[50] Salman Niazi, Mikael Ronström, Seif Haridi, and Jim Dowling. Size Matters: Improving the Performance of Small Files in Hadoop. In Proceedings of the 19th International Middleware Conference (Middleware '18), November 2018.

[51] Radu Tudoran, Alexandru Costan, and Gabriel Antoniu. OverFlow: Multi-Site Aware Big Data Management for Scientific Workflows on Clouds. IEEE Transactions on Cloud Computing. Vol. 4-1, 2016.

[52] Rajkumar Kettimuthu, Steven Link, John Bresnahan, Michael Link, and Ian Foster. Globus XI/O Pipe Open Driver: Enabling GridFTP to Leverage Standard Unix Tools. In Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery (TG '11), vol. 20, pp. 1-7, 2011.

[53] Rajkumar Kettimuthu, Zhengchun Liu, David Wheelerd, Ian Foster, Katrin Heitmann, and Franck Cappello. Transferring a Petabyte in a Day. Future Generation Computer Systems, vol. 88, pp. 191–198, 2018.

[54] Ruchisree Das, Subir Bandyopadhyay, Saja Al Mamoori, and Arunita Jaekel. Dynamic Provisioning of Fault Tolerant Optical Networks for Data Centers. In Proceedings of IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE), 2017.

[55] R. Zamudio, D. Catarino, M. Taufer, B. Stearn and K. Bhatia. Topaz: Extending Firefox to Accommodate the GridFTP Protocol. 2007 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2007, pp. 1-8, doi: 10.1109/IPDPS.2007.370549.

[56] Silicon Graphics Inc. XFS Algorithms & Data Structures. 3rd edition, 2006.

[57] Takashi Kurimoto, Shigeo Urushidani, Hiroshi Yamada, Kenjiro Yamanaka, Motonori Nakamura, Shunji Abe, Kensuke Fukuda, Michihiro Koibuchi, Yusheng Ji, Hiroki Takakura, and Shigeki Yamada. A Fully Meshed Backbone Network for Data-Intensive Sciences and SDN Services. In Proceedings of 8th International Conference on Ubiquitous and Future Networks (ICUFN), 2016.

[58] Teng Wang, Sarp Oral, Yandong Wang, Brad Settlemyer, Scott Atchley, and Weikuan Yu. BurstMem: A High-Performance Burst Buffer System for Scientific Applications. In Proceedings of 2014 IEEE International Conference on Big Data (Big Data), 2014.

[59] Uptime Institute Global Data Center Survey 2021 - Growth stretches an evolving sector. 2021.

[60] Vaibhav Singh. File storage migration to AWS in one month using AWS DataSync and Amazon FSx. AWS Storage Blog. 26 APR 2021. Retrieved April 8, 2022 from https://aws.amazon.com/blogs/storage/file-storage-migration-to-aws-in-one-month-using-aws-datasync-and-amazon-fsx/

[61] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. The Globus Striped GridFTP Framework and Server. In Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC'05), 2005.

[62] XRootD, https://xrootd.slac.stanford.edu/

[63] Yongki Kim. Cost efficiently migrate billions of small files with AWS Snowball. AWS Storage Blog. 26 OCT 2021. Retrieved April 8, 2022 from

https://aws.amazon.com/blogs/storage/cost-efficiently-migrate-billions-of-small-files-with-aws-snowball/

[64] Yuanlai Liu, Zhengchun Liu, Rajkumar Kettimuthu, Nageswara S.V. Rao, Zizhong Chen, and Ian Foster. Data transfer between scientific facilities - bottleneck analysis, eaninsights, and optimizations. In Proceedings of 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID' 2019), Cyprus, 2019.

[65] Zhengchun Liu, Prasanna Balaprakash, Rajkumar Kettimuthu, and Ian Foster. Explaining Wide Area Data Transfer Performance. In Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17), 2017.

[66] Zhengchun Liu, Rajkumar Kettimuthu, Ian Foster, and Nageswara V. Rao. Cross-geography scientific data transferring trends and behavior. In Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18), 2018.

Appendix: Core Pseudo Code for GridFTP Extension

GridFTP extension optimizes small file movement for GCT GridFTP 6.2 [26]. It packs small files into large memory buffers and transfers them using GridFTP. Its implementation is built by extending *globus-url-copy* and *globus-gridftp-server*. This appendix presents the core algorithms of *globus-url-copy* extension that pack memory buffers and forward them to GridFTP client. It mainly consists of two entities: a storage I/O (*sio*) thread and a dispatcher, which are shown in **Fig. 12** using an ftp *put* example.

One dispatcher and its associated *sio* thread are invoked for each concurrency. The dispatcher is realized using Globus Asynchronous Event Handling model [22]. These two entities collaborate using *pthreads* synchronization.

The details of each entity are explained using its pseudo code with the following data transfer example, which invokes GridFTP client command line tool to move files from a local directory, *${src_path}*, to a target path *${target_path}* on a remote server, denoted by *${SERVER_IP}:${SERVER_PORT}*, with the FTP protocol. It enables the data transfer pipeline to pack small files into 16MB buffers (*-pbs 16000000*) with 32 concurrent threads (*-cc 32*):

```
globus-url-copy -cc 32 -pbs 16000000 ${src_path} ftp://${SERVER_IP}:${SERVER_PORT}/${target_path}
```

1. Storage I/O Thread

The pseudo code of *sio* is depicted in Algorithm 1. Its main loop consists of 3 major actions: 1) waiting to receive file access requests from dispatcher (line 6); 2) checking file size for received requests (line 7 to 10); 3) in case an enough number of small files are received (i.e., the total amount of which is not smaller than *pbs*), it reads files (line 16~19), packs them into *packing_buffer* (line 20~46), and then forwards each buffer to its dispatcher (line 27~29 and 43~45). *no_more_requests* is set when all files in the source directory have been dispatched.

The dispatcher mainly extends *globus_l_guc_transfer_kickout*() in gct/gass module. It is invoked after *pending_dirs_queue* and *pending_files_queue* are initialized, which only contain the source path to be moved ${*src_path*}, as shown in Algorithm 2. One dispatcher is invoked per concurrency and works with its *sio* thread. It first checks whether its *sio* thread is waiting for requests. If so, it sends files in *pending_files_queue* to the I/O thread and notifies it to start transferring data using ftp (line 4~8). If *pending_files_queue* is empty, the dispatcher acquires a directory from *pending_dirs_queue* and detects files contained in the directory by *calling parse_dir*(), which puts detected files into *pending_files_queue*, and finally registers *dispatcher_worker*() to GridFTP event engine as an immediate event to (line 9~11) to trigger.

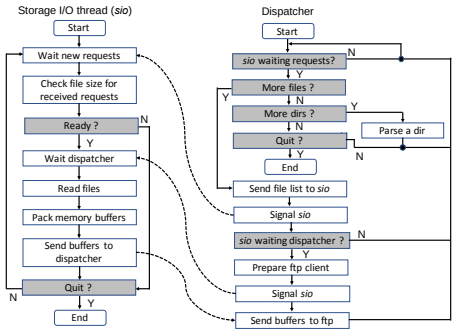


Fig. 12. The extension of GridFTP client.

Module *ftp_transfer*() is responsible for forwarding packed memory buffers to GridFTP client, as shown

in Algorithm 3. As depicted in Algorithm 4, after each transfer is finished, GridFTP client invokes *globus_l_gass_copy_ftp_put_done_callback*(), which calls *dispatcher_worker*() again.

Algorithm 1. Packing small files into memory buffers.

```
1  procedure storage_io_worker(cc_id, pbs, packing_threshold, chunk_size)
2    received_size = 0; metadata_size = MAX_PATH_LEN + EXTRA
3    init(new_requests_queue[cc_id]); init(packing_queue)
4    while(TRUE)
5      if(received_size < pbs and !no_more_requests)
6        waiting_on_requests(sio_worker[cc_id], new_requests_queue[cc_id]);
7        for(file in new_requests_queue[cc_id])
8          if(size(file) <= packing_threshold)
9            enqueue(packing_queue, file)
10           received_size += size(file)
11     if(received_size >= pbs or no_more_requests)
12       waiting_on_dispatcher_ready()
13       offset = 0; packed_size = 0
14       packing_buffer = allocate_memory(chunk_size);
15       while(!empty(packing_queue) and packed_size <= pbs)
16         file = dequeue(packing_queue)
17         fd = open(file->local_path)
18         num_bytes = read(fd, tmp_buffer)
19         close(fd)
20         if(remaining_space < nbytes + metadata_size)
21           globus_i_gass_copy_buffer_t *buffer_entry = allocate_memory(...)
22           buffer_entry->bytes = packing_buffer
23           buffer_entry->nbytes = offset
24           buffer_entry->offset = global_offset
25           buffer_entry->bytes = packing_buffer
26           buffer_entry->last_data = FALSE
27           lock(dest_mutex[cc_id])
28           enqueue(buff_entry, dest_queue[cc_id])
29           unlock(dest_mutex[cc_id]))
30           packing_buffer = allocate_memory(chunk_size)
31           packed_size += offset
32           offset = 0
33         add_metadata(tmp_buffer, num_bytes + metadata_size, file->remote_path)
34         memcpy(packing_buffer + offset, tmp_buffer, num_bytes + metadata_size)
35         offset += num_bytes + metadata_size
36
37       if (offset) /*send the last piece of packing buffer*/
38         globus_i_gass_copy_buffer_t *buffer_entry = allocate_memory(...)
39         buffer_entry->nbytes = offset
40         buffer_entry->offset = global_offset
```

```
41              buffer_entry->bytes = packing_buffer
42              buffer_entry->last_data = TRUE
43              lock(dest_mutex[cc_id])
44              enqueue(buff_entry, dest_queue[cc_id])
45              unlock(dest_mutex[cc_id]))
46              packed_size += offset
47          received_size -= packed_size
48      if(no_more_requests and empty(packing_queue))
49          break
```

Algorithm 2. Scheduling file access requests to *sio* threads and coordinating file transfers.

```
1   init(pending_dirs_queue, pending_files_queue, src_path)
2   dispatch_worker(cc_id)
3     if(is_waiting_on_requests(sio_worker[cc_id]))
4       if(!empty(pending_files_queue[cc_id]))
5         while(!empty(pending_files_queue[cc_id]))
6           enqueue(new_request_queue[cc_id], dequeue(pending_files_queue[cc_id])
7         singal(sio_worker[cc_id], READY)
8         ftp_transfer(cc_id)
9       else if(!empty(pending_dirs_queue))
10        parse_dir(dequeue(pending_dirs_queue))
11        globus_callback_register_oneshot(dispatcher_worker, cc_id)
12      else
15        singal(sio_worker[cc_id], NO_MORE_REQUESTS)
```

Algorithm 3. Transfer memory buffers through GridFTP client.

```
1   ftp_transfer(cc_id)
2     globus_gass_copy_size_ftp()
3     globus_ftp_client_put(globus_gass_copy_ftp_put_done_callback, cc_id)
4     signal(sio_worker[cc_id], READY)
5     while(!empty(dest_queue[cc_id]))
6       buffer_entry = dequeue(dest_queue[cc_id])
7       globus_ftp_client_register_write(buffer_entry,
              globus_gass_copy_ftp_write_callback, cc_id)
```

Algorithm 4. Buffer transfer complete callback.

```
1   globus_l_gass_copy_ftp_put_done_callback(cc_id)
2     globus_callback_register_oneshot(dispatcher_worker, cc_id)
```

Author Biography

David Abramson is a Professor of Computer Science, and currently heads the University of Queensland Research Computing Centre.

He has been involved in computer architecture and high performance computing research since 1979.

He has held appointments at Griffith University, CSIRO, RMIT and Monash University.

Prior to joining UQ, he was the Director of the Monash e-Education Centre, Science Director of the Monash e-Research Centre, and a Professor of Computer Science in the Faculty of Information Technology at Monash.

From 2007 to 2011 he was an Australian Research Council Professorial Fellow.

David has expertise in High Performance Computing, distributed and parallel computing, computer architecture and software engineering.

He has produced in excess of 230 research publications, and some of his work has also been integrated in commercial products. One of these, Nimrod, has been used widely in research and academia globally, and is also available as a commercial product, called EnFuzion, from Axceleon.

His world-leading work in parallel debugging is sold and marketed by Cray Inc, one of the world's leading supercomputing vendors, as a product called ccdb.

David is a Fellow of the Association for Computing Machinery (ACM), the Institute of Electrical and Electronic Engineers (IEEE), the Australian Academy of Technology and Engineering (ATSE), and the Australian Computer Society (ACS).

His hobbies include recreational cycling, photography and making stained glass windows. He is also an amateur playwright, and author of Purely Academic.

Author Biography

Zhengchun Liu is a Computer Scientist at the Data Science and Learning division of Argonne National Laboratory, and a Scientist At-Large of the Consortium for Advanced Science and Engineering at The University of Chicago. Before this, he was a research scientist at the University of Chicago from 2018.03 to 2019.08, and a Postdoc at the Mathematics and Computer Science division of Argonne National Laboratory from 2016.09 to 2018.03. He focuses his research at the intersection of machine learning and computer system.

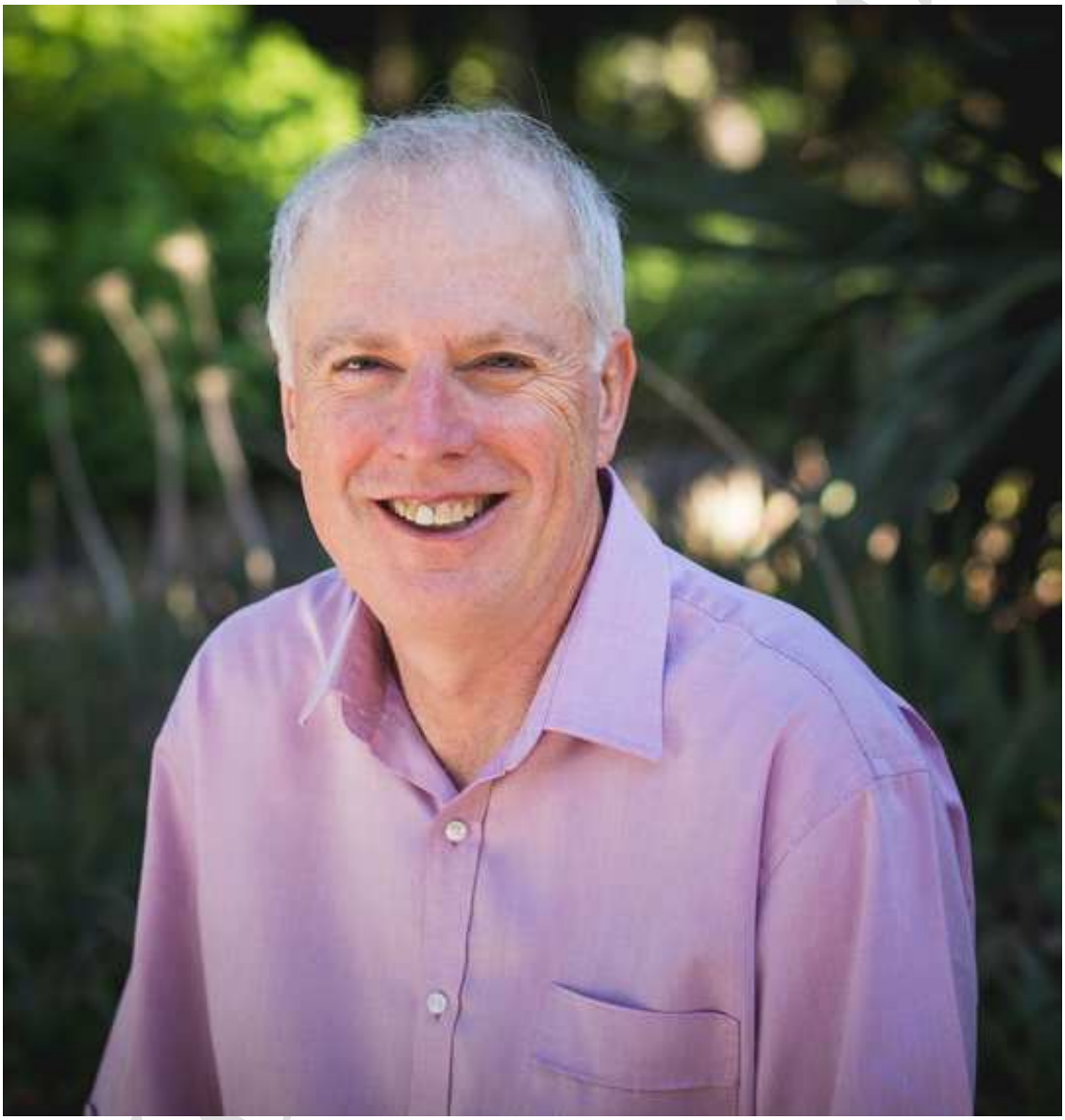
Author Biography

Dr. Rajkumar Kettimuthu is a Computer Scientist and Group Leader at Argonne National Laboratory, a Senior Scientist at University of Chicago and a Senior Fellow at Northwestern-Argonne Institute of Science and Engineering. His research interests include intelligent compute and data infrastructure for science, advanced wired and wireless communications for science, and Quantum networks. Data transfer protocol and tools developed by him and his colleagues at Argonne are deployed in 60K+ locations in six continents moving 5 Petabytes+ of data every day. He has co-authored 150+ peer-reviewed articles. He is a recipient of R&D 100 award. He is a distinguished member of ACM and a senior member of IEEE.

Author Biography

Dr. Chao Jin is a researcher working at the Research Computing Centre, the University of Queensland. His general research interests include distributed systems, parallel computing, and storage systems. He currently focuses on scalable and high throughput distribute systems, energy efficient computing, and performance optimization of shared memory systems.

Author Photo

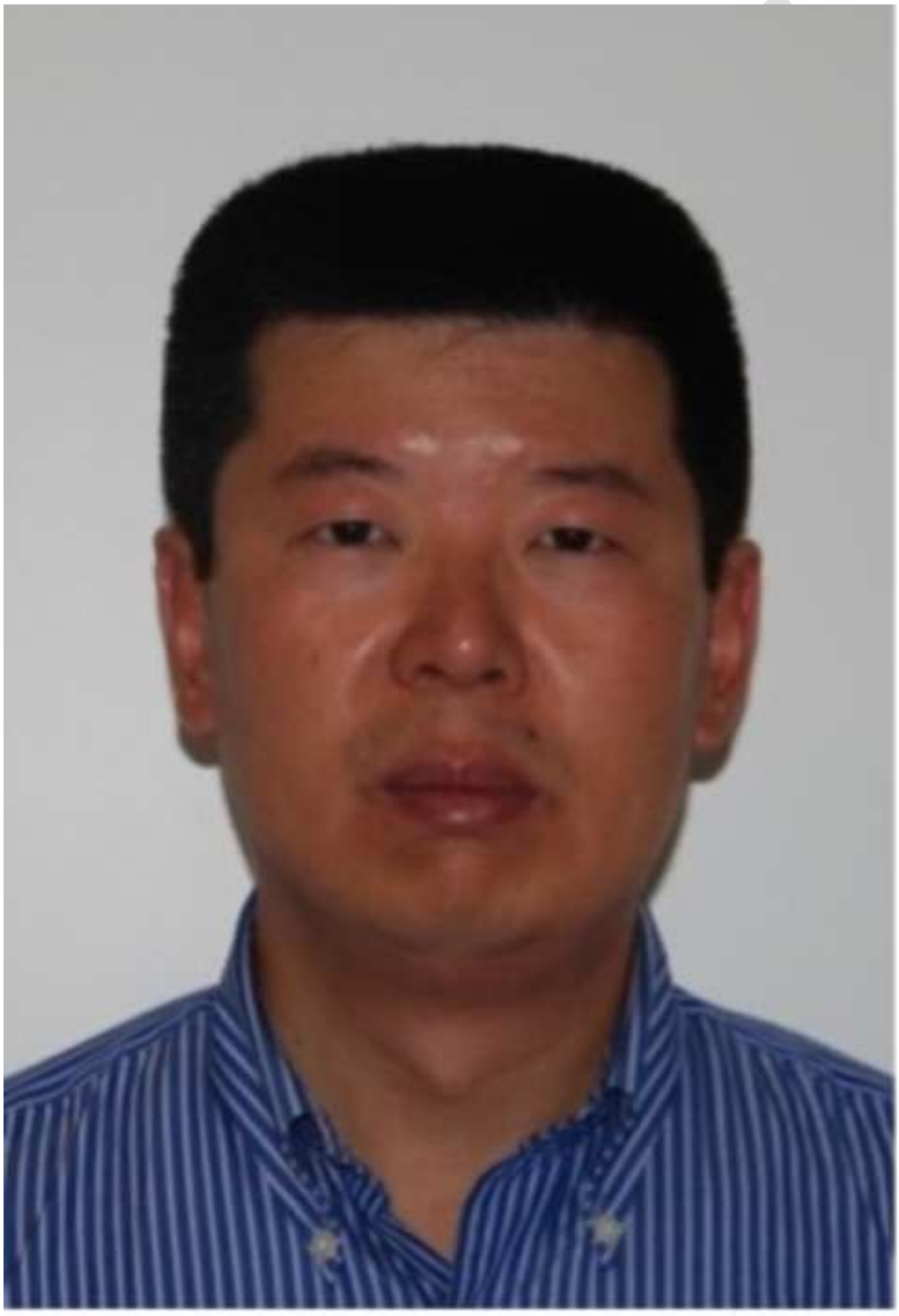


Author Photo

Click here to access/download;Author Photo;Zhengchun_Liu headshots-201704.jpg

Author Photo

Author Photo



Author Names and their Contribution

Authors and contributions

Chao Jin performed much of the background research and experimental work described in the paper. He is also the primary author of the paper itself.

David Abramson is the project lead and was responsible for all stages of the paper including the initial research formulation, production of the paper and editing the work.

Jake Carroll contributed to much of the technical evaluation of the work.

Zhengchun Liub, and Rajkumar Kettimuthub

Both of these authors collaborated on the research formulation and experimental work and contributed to the editing of the paper. They are responsible for the original GridFTP research and development that formed the base of this project.

Declaration of interests

☐ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☒ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

David Abramson reports financial support, administrative support, and travel were provided by University of Queensland.