Hindawi
Scientific Programming
Volume 2018, Article ID 4120561, 8 pages
https://doi.org/10.1155/2018/4120561

Research Article

NUMA-Aware Thread Scheduling for Big Data Transfers over Terabits Network Infrastructure

Taeuk Kim[1], **Awais Khan**[1], **Youngjae Kim**[1], **Preethika Kasu**[2] and **Scott Atchley**[3]

[1]*Sogang University, Seoul, Republic of Korea*
[2]*Ajou University, Suwon, Republic of Korea*
[3]*Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA*

Correspondence should be addressed to Youngjae Kim; youkim@sogang.ac.kr

Received 29 December 2017; Accepted 27 March 2018; Published 7 May 2018

Academic Editor: Basilio B. Fraguela

Copyright © 2018 Taeuk Kim et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The evergrowing trend of big data has led scientists to share and transfer the simulation and analytical data across the geodistributed research and computing facilities. However, the existing data transfer frameworks used for data sharing lack the capability to adopt the attributes of the underlying parallel file systems (PFS). LADS (Layout-Aware Data Scheduling) is an end-to-end data transfer tool optimized for terabit network using a layout-aware data scheduling via PFS. However, it does not consider the NUMA (Nonuniform Memory Access) architecture. In this paper, we propose a NUMA-aware thread and resource scheduling for optimized data transfer in terabit network. First, we propose distributed RMA buffers to reduce memory controller contention in CPU sockets and then schedule the threads based on CPU socket and NUMA nodes inside CPU socket to reduce memory access latency. We design and implement the proposed resource and thread scheduling in the existing LADS framework. Experimental results showed from 21.7% to 44% improvement with memory-level optimizations in the LADS framework as compared to the baseline without any optimization.

1. Introduction

The continuous inflation in data generation is raising the sharing and collaboration needs for effective simulations and real-time analysis. Such sharing and collaboration need a massive-scale data transfer across geodispersed data centers [1]. The Brookhaven National Lab (BNL) cooperates with European Large Hadron Collider (LHC) in the ATLAS experiment where more than 3,000 scientists participate and produce petabytes of simulation and analytical data, motivating collaboration ventures [2]. Such large-scale collaboration environments highly motivate us to revisit the architecture of existing end-to-end data transfer tools such as bbcp [3], LADS [1], and GridFTP [4].

In an end-to-end data transfer between geodispersed data centers, there are three significant factors concerning the data transfer performance and throughput, that is, network, storage, and memory. However, in realistic large-scale HPC environments, the network infrastructure delivers a high bandwidth and it is further improving [5], for example, ESnet of DOE [6]. So, we do not consider network as a major bottleneck in limiting the data transfer activities in large-scale HPC environments equipped with high-speed network connectivity [5, 6].

The parallel file system (PFS) [7, 8] based storage-backends deployed in the data centers act as a bottleneck when thread count exceeds the service rate of object storage server (OSS) or multiple threads access the same object storage target (OST). LADS [1], high-speed end-to-end data transfer tool between data centers, minimizes this I/O contention by being aware of data chunk's layout and scheduling threads based on it. The memory bottleneck can be incurred in NUMA environment when threads access remote NUMA node's memory. If the buffer which LADS uses to transfer data is allocated in a different NUMA node from I/O threads, I/O threads need to access remote NUMA node during data transfer and it makes memory access consume longer latency. Currently, LADS tool does not offer any solution to overcome

this problem. In this paper, we propose MTS (Memory-aware Thread Scheduling) method to solve the memory bottleneck issues with LADS data transfer tool.

The data transfer frameworks such as GridFTP [4], bbcp [3], and LADS [1] have been designed to ensure high-speed data transmission. However, GridFTP and bbcp are designed on file-based data transfer, whereas LADS is optimized for object-based data transfer where multiple threads can work on multiple object chunks simultaneously to improve end-to-end data transfer speed. With the improvement in network infrastructure, the parallel and distributed file systems such as Lustre [8], Gluster [7], and Ceph [9] are improving their storage and computing frameworks in order to derive the maximum bandwidth. GridFTP [4] and bbcp [3] cannot gain high benefits from these file systems since they are designed without considering the underlying file system, whereas LADS [1], due to its layout-aware nature of the data transfer mechanism, can fully utilize the benefits of these underlying parallel file systems. Besides, LADS uses Common Communication Interface (CCI) [5] to exploit the high-speed terabit networks.

In this paper, we emphasize the possible bottlenecks and opportunities such as high-speed network and NUMA architecture in end-to-end data transfer path. An end-to-end data transfer can meet multiple bottlenecks such as (i) storage, (ii) CPU, and (iii) memory. The storage becomes a bottleneck when data transfer software is unaware of the underlying file system architecture; for example, parallel file systems use chunking and striping techniques to store data in a more efficient fashion. So, the data transfer tool can efficiently utilize storage bandwidth by having the knowledge about storage layout, without which storage bandwidth can be underutilized. The CPU bottleneck occurs, if data transfer tool does not take into account the utilization of multicores while transferring data. The underutilization can happen when the threads are less than the CPU cores and some of the cores remain idle. The overutilization can occur, where thread count is very high with respect to available CPU cores.

The memory contention can occur in two conditions. Firstly, when multiple threads or processes are accessing the same shared memory region. Secondly, when threads hosted on CPU sockets access the remote memory of other CPU sockets. Both these contentions can lead to increased memory access latency. LADS framework addresses the storage and CPU bottlenecks by implementing layout-aware multithreaded architecture [1]. However, LADS does not consider the memory bottleneck issues.

In this paper, we address the memory bottleneck issues by proposing memory buffer partitioning at each CPU socket and scheduling threads with being aware of NUMA architecture. Partitioning the memory buffer reduces the shared memory regions and thread scheduling reduces the remote memory access across CPU sockets.

This paper makes the following contributions:

(i) Increase in the number of I/O threads in LADS leads to contention in memory controller. To address this issue, we propose *Multiple Memory Buffers* (MMB), which distributes the RMA buffer across all the CPU sockets to reduce memory controller congestion.

(ii) The distributed RMA buffer at each CPU socket alone is not sufficient to improve the memory latency problem. In some cases, threads may try to access the remote RMA buffer hosted at different CPU sockets. To avoid such remote memory access, we design and implement *Memory-aware Thread Scheduling* (MTS) to schedule threads to access only the RMA buffer hosted on the same CPU socket. MTS reduces overall memory latency by eliminating all accesses to remote memory.

(iii) We conduct a comprehensive evaluation for our proposed ideas using a file size distribution based on a snapshot of the real peta-scale file system at ORNL [10]. We compare the performance of our proposed MMB and MTS with default settings where it uses single RMA buffer and where it applies NUMA binding to threads. From our experimental results, we have observed that our proposed idea yields up to 44% higher data transfer rate than the default settings.

The rest of this paper is organized as follows. Section 2 describes the LADS architecture and implementation details. Section 3 outlines the design and implementation of the proposed memory-level optimizations. Experimental setup and evaluation results are presented in Section 4. Section 5 describes the related works and we conclude in Section 6.

2. Layout-Aware Data Scheduling

Data sharing and scientific collaborations are advancing in recent years. Tools like GridFTP [4], bbcp [3], and LADS [1] are developed for efficient data transfer across geodistributed data storage facilities. LADS [1], an end-to-end data transfer tool, exploits the underlying storage architecture for optimizing the bulk data movement between data centers connected via high-speed terabit network. LADS uses Common Communication Interface (CCI) to fully utilize the terabit network [5] capabilities. The work proposed in this paper is an extension of LADS data transfer framework. This section describes LADS architecture.

Threads and Work Queues. LADS consists of three different types of threads, that is, Master (MT), Communication (CT), and I/O thread. The Master thread splits the workload in chunks and makes each chunk into a task. These tasks are inserted into the OST queue. In particular, there are as many OST queues as the number of OSTs in the Lustre file system. Master thread schedules I/O threads to OST queues and the I/O threads dequeue the tasks from OST queues to perform I/O operations. On the other hand, Communication thread manages the end-point communication between source and sink. The I/O thread loads the data chunks from storage to RMA buffer at source and stores them from RMA buffer to storage at sink. Both the Master and the Communication thread own work queues which hold the requests to transfer data objects to each other.

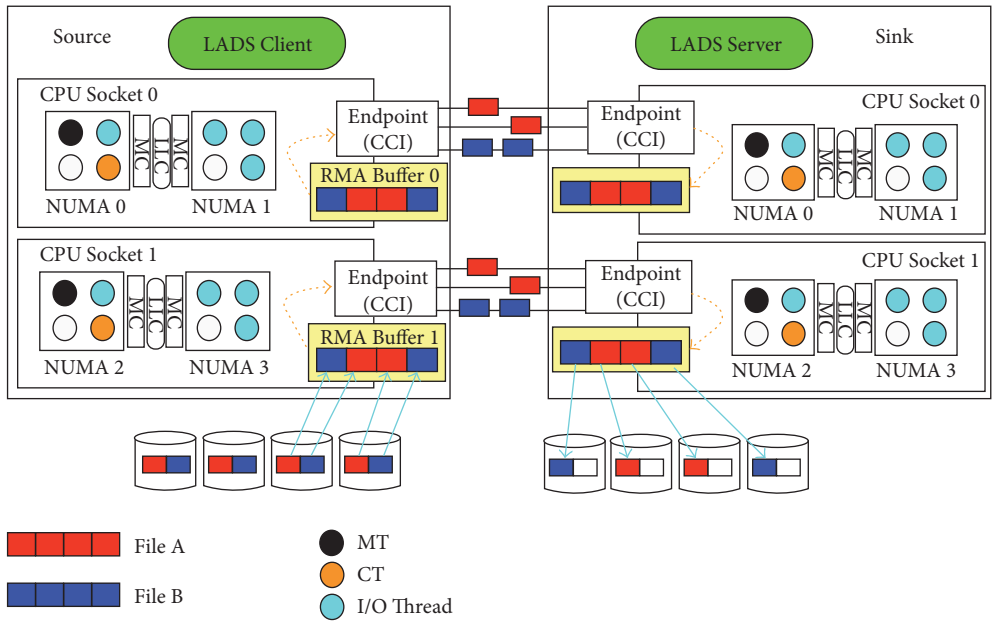


Figure 1: An overview of proposed memory-level optimizations in LADS [1] framework.

Communication Protocol. From now, the notation of source-end in the data transmission will be SRC, and the notation of sink-end in the data transmission will be SNK.

Step 1. The SRC MT catches the layout of data chunks for the requested file and adds the request to the SRC Communication Thread Work Queue (CWQ). The SRC CT sends a new file request to the sink-end via CCI end-point connection. Similarly, at the sink-end, SNK CT upon receiving the request forwards it to SNK Master Thread Work Queue (MWQ).

Step 2. The SNK MT upon receiving the request creates a new file with the same name as in the request. The file id corresponding to the newly created file is added to SNK CWQ in the request form and is sent to source-end. After receiving the request at source-end, SRC CT directs the request to SRC MWQ. The SRC MT loads the chunks information into SRC OST queues.

Step 3. Once the SRC MT inserts the data chunk into the SRC OST queue, SRC MT wakes up the SRC I/O threads according to the number of OSTs in the underlying SRC Lustre file system. Every SRC I/O thread at source-end then traverses the OST queues, gets the chunks information, reads the chunks from the physical OST storage, and loads them into the SRC RMA buffer. Then, SRC I/O thread inserts a request into the SRC CWQ to send the data chunk's information. SNK CT receives the request, and it gets the data chunk from SRC RMA buffer via RDMA access.

Step 4. At SNK, after reading the data chunk from SRC RMA buffer, the SNK CT loads the data chunk into the SNK RMA buffer. Then the SNK I/O threads are scheduled to the SNK OST queues by the SNK CT to write the data chunk from the SNK RMA buffer to the SNK physical OST storage. The SNK CT sends the transfer completion message and repeats Steps 3 and 4, till all the data chunks of all the files are transferred.

3. Design and Implementation

This section describes the proposed optimization methods and its design and implementation details.

3.1. Overview. Figure 1 shows an overview of the proposed memory-level optimizations in LADS software framework [1]. The proposed approach consists of three major elements, (i) distributed RMA buffers at each CPU socket, (ii) Socket-based Memory-aware Thread Scheduling (SMTS), and (iii) NUMA node based Memory-aware Thread Scheduling (NMTS). The proposed optimizations are highly flexible and can be applied to n CPU sockets. Each CPU socket consists of cores, shared Last Level Cache (LLC), and one or more memory controllers, whereas the number of cores, LLC, and memory controller depend on the CPU type used. If there are multiple sets of cores and memory controllers in a CPU socket, then there will be more than one NUMA node per CPU socket. As shown in Figure 1, each CPU socket hosts three types of threads, that is, Master (MT), Communication (CT), and I/O thread; each thread performs a specific functionality similar to LADS architecture [1]. Proposed optimization methods are an extension of existing LADS implementation.

The Master thread captures the layout of files to be transferred from Lustre [8] object storage targets (OSTs) and schedules I/O threads to execute I/O operations specific to these OSTs. The I/O threads read the data chunks from the object storage into the RMA buffer and notify the Communication thread, which is responsible for creating an

end-point connection via CCI API [5] and sending the data chunks in RMA buffer from source side to sink side.

3.2. MMB: Multiple Memory Buffers. The LADS data transfer framework relies on single RMA buffer, where all the I/O threads load and store the data chunks between RMA buffer [1] and underlying storage system. However, the use of single RMA buffer can cause two problems: (i) memory latency caused by remote CPU socket's memory access and (ii) contention on the memory controller. When threads are created, LADS schedules the threads to random CPU cores. Generally, I/O threads are scattered on all CPU sockets, that is, several I/O threads from different CPU sockets access a specific CPU socket's RMA buffer. This incurs a number of remote memory accesses which have a longer latency than local memory access. Moreover, in LADS the number of I/O threads is configurable and in ideal scenario, it considers the number of OSTs belonging to the target Lustre file system for the number of I/O threads. But, in realistic scenarios, data centers using Lustre may exceed hundreds or thousands of OSTs, so the number of I/O threads will be configured according to the number of CPU cores. If a high number of I/O threads corresponding to the number of cores in a multicore environment access the single RMA buffer, the contention will occur in the memory controller of the CPU socket or NUMA node hosting the RMA buffer. We propose *Multiple Memory Buffer* (MMB) scheme, which distributes the RMA buffer to each CPU socket's memory in the existing LADS framework. Figure 1 shows the RMA buffer distribution to each CPU socket. This partitioning of RMA buffer per CPU socket gives significant benefits. First, it reduces the memory controller contention significantly which is caused by single RMA buffer on increasing the number of I/O threads. Second, it reduces the memory latency due to the less number of remote memory accesses of I/O threads.

3.3. MTS: Memory-Aware Thread Scheduling. This section describes the details about the two levels of *Memory-aware Thread Scheduling* (MTS). First, we discuss the Socket-based Thread Scheduling and second, we present NUMA node based Thread Scheduling when CPU socket has multiple NUMA nodes.

3.3.1. Socket-Based Thread Scheduling. The partitioning of RMA buffer across each CPU socket gives the privilege to reduce the memory controller contention. But the remote memory access is still possible and it can increase memory access latency as compared to local memory access. To solve the remote memory access problem, we propose *Socket-based Memory-aware Thread Scheduling* (SMTS) scheme. SMTS schedules I/O threads between CPU sockets in such a way that all the I/O threads should access RMA buffer residing on the same CPU socket. Also, the RMA buffer is registered by the Communication thread and accessed via RDMA read/write operations through the end-point. So, a single and dedicated Communication thread is required per CPU socket to manage each RMA buffer. In our approach, each CPU socket has only one Master and Communication thread and configurable number of I/O threads. All threads and connections created per CPU socket are independent of other CPU sockets. The Master thread in each connection controls the chunk-level scheduling and transmits only the assigned files and I/O threads also work against the same objects designated by the Master thread. In this way, all I/O threads are pinned to their local CPU sockets' cores.

3.3.2. NUMA Node Based Thread Scheduling. Here, we discuss the NUMA-aware optimizations required in LADS framework for CPU sockets equipped with multiple NUMA nodes. Considering the fact that it is possible to associate multiple NUMA nodes per CPU socket [11], it is highly necessary to schedule threads onto NUMA nodes at each CPU socket. To address this kind of cases, we propose *NUMA node based Memory-aware Thread Scheduling* (NMTS). NMTS is shown in Figure 1 as threads pinned at each NUMA node in every CPU socket. The motivation behind addressing thread scheduling at NUMA node is to avoid remote memory access inside a CPU socket when the socket is equipped with multiple NUMA nodes. To schedule threads inside a CPU socket, two elements are taken into account, (i) *interaction between threads* and (ii) *fairness of core usage*. In LADS [1] framework, to complete data transfer more efficiently, threads interact with each other at a high extent.

At first, Master and Communication threads interact, when data transfer begins. LADS framework has its own data structures for file metadata and keeps information about the file size, fd, and data chunk layout of the files required to transfer. Both the Master and Communication thread maintain work queues. In the rest of the paper, we will use MWQ to denote Master thread's work queue and CWQ to represent Communication thread's work queue. The Master thread schedules the requested file to CWQ at the start of data transfer. The Communication thread also sends a file request to the MWQ, when a file request is received from the sink side. In case of high transfer traffic, the interaction between Master and Communication threads increases. So, in such scenarios the placement of Master and Communication threads plays a major role. If Master and Communication threads are placed on different NUMA nodes, the performance can degrade due to remote memory access inside CPU socket.

Moreover, Master thread manages the information about OSTs containing data chunks in queue. The number of queues is equivalent to the number of OSTs in target Lustre file system. The I/O threads access these OST queues when loading data chunks from storage to RMA buffer to get information about the chunk. So, placing I/O threads near Master thread is also an important factor in performance optimization. However, due to the high number of I/O threads, it is not possible to keep all the I/O threads in same NUMA node with Master thread. Scheduling all the I/O threads on single NUMA node incurs core contention among I/O threads. Therefore, firstly we suggest scheduling Master and Communication threads on the same NUMA node and then placing I/O threads to NUMA nodes as close as possible to Master thread. We distribute I/O threads across the NUMA nodes to enhance the fairness of per core usage as shown

in Figure 1. Figure 1 shows that Master and Communication threads are hosted by same NUMA node and I/O threads are hosted near to the Master thread whereby load is evenly distributed among the NUMA nodes' cores.

4. Evaluation

4.1. Experimental Setup. In this experimental setup, we use a private testbed with eight nodes connected by InfiniBand (IB) EDR (100 Gb/s). The nodes use the IB network to communicate with each other. We use E5-2650 v4 with two CPU sockets (two NUMA nodes for each CPU socket and six cores for each NUMA node), 128 GB DRAM, running with CentOS 7.3, Linux kernel 3.10.0-514.21.1.el7.x86_64. Two of the eight nodes are used as data transfer nodes (DTN) for source and sink hosts. The other six nodes are used as two different storage systems. We built two storage systems for more realistic experiments.

(i) *Testbed-I:* when building a peta-scale storage system with Lustre file system, it uses hundreds of object storage servers with tens of object storage targets in each server. For example, Spider II is a center-wide, Lustre-based system for one of the fastest supercomputer in US, Titan with two namespaces [1]. Each namespace has 144 OSSs, which manage seven OSTs and each OST is configured with 10 HDDs in RAID-10 [1]. That is, the I/O of the PFS used in realistic environment is too fast to use all the available bandwidth of the PFS. In order to emulate such environments, for each source and sink host, we use memory file system mounted on a server node via NFS v4.0 as a high performance file system.

(ii) *Testbed-II:* we experimented with a small-scale testbed equipped with Lustre file system for each source and sink host, which is configured using one OSS, one MDS, and eight OSTs, each mounted over 600 GB 10K RPM 6 Gbps SAS, 2.5″ HotPlug HDD. For each file system, we created 8 logical volume drives on top of the HDDs to make each disk become an OST. We set stripe count to be 1 and stripe size to be 1 MB.

We developed an in-house memory benchmark program in C++ to measure the memory bandwidth between different NUMA nodes of source and sink hosts.

From Table 1, we can notice that in-socket remote NUMA node is 4% slower when compared to in-socket local NUMA node, whereas remote CPU socket NUMA node is 32% slower than in-socket local NUMA node.

We use two representative file distributions to have two file groups appropriate for our small-scale testbed setup: one for small number of big files with 8 × 3 GB files, referred to as big-file workload, and the other for large number of small files with 6,000 × 1 MB files, referred to as small-file workload. HPC file size distribution follows a binomial distribution in terms of file system space occupancy and number of files: larger files occupy most of the file system space, but with fewer numbers of files. On the other hand, small files have a more number of files, but the file system space occupancy is very small [1].

Table 1: Memory BW of local and remote NUMA node.

CPU socket & NUMA node locality	BW (MB/s)
In-socket local NUMA node	3160.43
In-socket remote NUMA node	3038.85
Remote socket NUMA node	2131.24

Table 2: The evaluation of thread placement across CPU sockets. Total 8 I/O threads are used in this experiment.

Thread placement	Throughput (MB/s)
All threads placed on same CPU socket	2065.21
CT placed on different CPU socket	1914.02
MT placed on different CPU socket	1719.80

For the convenience of analysis, we define the following schemes:

(i) Baseline: it uses a single RMA buffer. The RMA buffer has a different physical memory location depending on which NUMA node it is placed. Experiment is performed according to the position of RMA buffer and NUMA binding. In the experiment, N*i* means that the single RMA buffer is located at NUMA node *i*, and NB*i* means that the single RMA buffer is located at NUMA node *i* with I/O threads bound on same node. If I/O thread count exceeds the number of cores, I/O threads are distributed to NUMA node *i* and to its nearest node evenly.

(ii) MMB: the RMA buffer is partitioned by CPU socket. In our experiment, a single RMA buffer of baseline is partitioned into two.

(iii) MTS: it applies both Memory-aware Thread Scheduling Algorithms (SMTS and NMTS) to MMB.

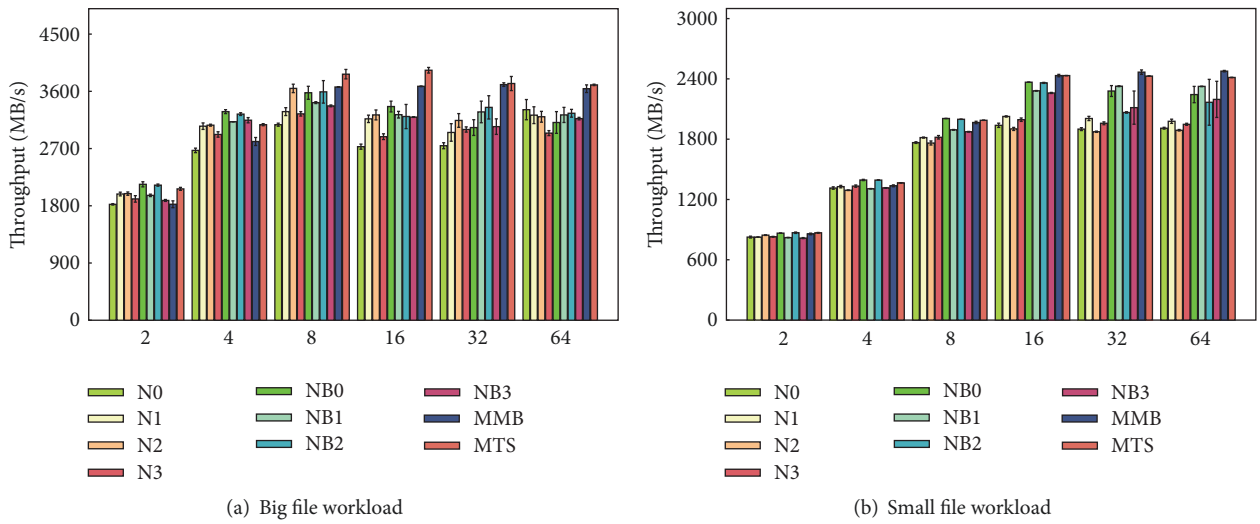
For each iteration of memory file system and Lustre experiment, we cleared page caches of the source, sink, and storage servers for a fair evaluation.

4.2. Results

Evaluation of Memory Access Speed by Intimacy of Thread Type. In LADS, threads share data structures; for example, MT and CT share the MWQ and CWQ whereas I/O threads share OST queues with MT. RMA buffer is shared between CT and I/O threads. Thus, increasing the locality between threads and shared data structures (memory variables and queues) benefits to improve the performance.

In this experiment, we perform the evaluation according to the arrangement of MT, CT, and I/O threads on same and different CPU sockets.

Table 2 shows the throughput comparison for various placements of CT and MT on CPU sockets. The experiment was made using small files with Testbed-I. We can clearly see that placing CT and MT on the same NUMA node which has the RMA buffer helps in improving the data transfer rate. The



(a) Big file workload

(b) Small file workload

Figure 2: Performance comparison for Memory-aware Thread Scheduling using Multiple Memory Buffers per CPU socket. The x-axis shows the I/O thread count. Ni and NBi are for baseline experiments. The error bar depicts min. and max. deviation from the average of 3 iterations.

MTS is aware of thread types and prioritizes to schedule them on the same NUMA nodes. In the following evaluations, we run experiments by fixing the placement of the CT and MT on the same NUMA node with the RMA buffer in cases of NUMA binding.

Evaluating MTS with High-Speed PFS. To verify the effectiveness of the proposed MMB and MTS idea, we compare the data transmission throughput with baseline using a single RMA buffer. Particularly, we experiment by increasing the number of I/O threads to evaluate the performance according to CPU core utilization. Figures 2(a) and 2(b) show the comparison results for big and small file workloads, respectively, with Testbed-I. In order to confirm the performance limitation of the single RMA buffer and the maximum performance achievable by the NUMA binding, we evaluate the baseline by changing the NUMA node position of the single RMA buffer with and without the NUMA binding of threads. The RMA buffer locates on NUMA node i in experiments Ni and NBi, and I/O threads are bounded with considering the location of RMA buffer in NBi.

In Figure 2(a), we first analyze the results with the single RMA buffer (refer to results with labels in Ni and NBi). We can observe about 23% performance improvement on average with NUMA binding over without NUMA binding. As the number of I/O threads increases, overall performance improvement is observed till 8 I/O threads, whereas increasing I/O threads to 16, there is no significant improvement in the performance. This is due to the thread saturation of underlying file system. Moreover, we observe that performance is slightly lower in 16 I/O threads. We suspect the reason of performance degradation to be memory contention and remote memory access by the single RMA buffer.

Second, we compare the baseline results with MMB and MTS. From the results, we can confirm that overall performance improvement by MMB over the baseline varies from 13.0% to 34.7% and with MTS, its improvement is from 21.7% to 44%. That is, we observe that MTS further increases the performance of MMB by about 10%. Overall, data transfer rate increases as the number of I/O threads increases up to 16, but after that, improvement is not significant. We also see that, up to 2–4 I/O threads, MMB and MTS have little effect. When CPU cores are not fully utilized, single RMA buffer implementation outperforms partitioning the RMA buffer, because where performance storage is not fully exploited due to small number of I/O threads, dividing RMA buffer and binding threads to NUMA node incurs additional overhead. On the other hand, from 8 I/O threads, MMB and MTS show improved performance over the baseline. Particularly, in 16 I/O threads, we observe that the average performance improvements of MMB and MTS are 9.4–15.1% and 16.9–23%, respectively, over the baseline with NUMA binding. This shows that MMB and MTS contribute on the memory-level optimization in data transfer where storage is sufficiently fast.

In Figure 2(b), we have observations similar to the big file workload experiment. Up to 4 I/O threads, there is almost no performance difference. On the other hand, after 8 I/O threads, we can observe that MMB and MTS have on an average higher performance than baseline when we are over 16 threads. We observe little performance impact by MTS over MMB for small file workload, whereas MTS shows higher performance than MMB for big file workload.

Evaluating MTS with Slow PFS. If the storage performance is low, the performance improvement of MMB and MTS can be reduced. Thus, in the next experiment, we test with the source and sink hosts on which the small-scale Lustre file system is mounted (Testbed-II). Figure 3 shows the results of the experiment with big and small file workloads in Testbed-II. First, we observe that the maximum data rate is smaller than the experiment in Testbed-I. We see up to 1363 MB/s transfer rate with 8 I/O threads in big file workload and 435 MB/s

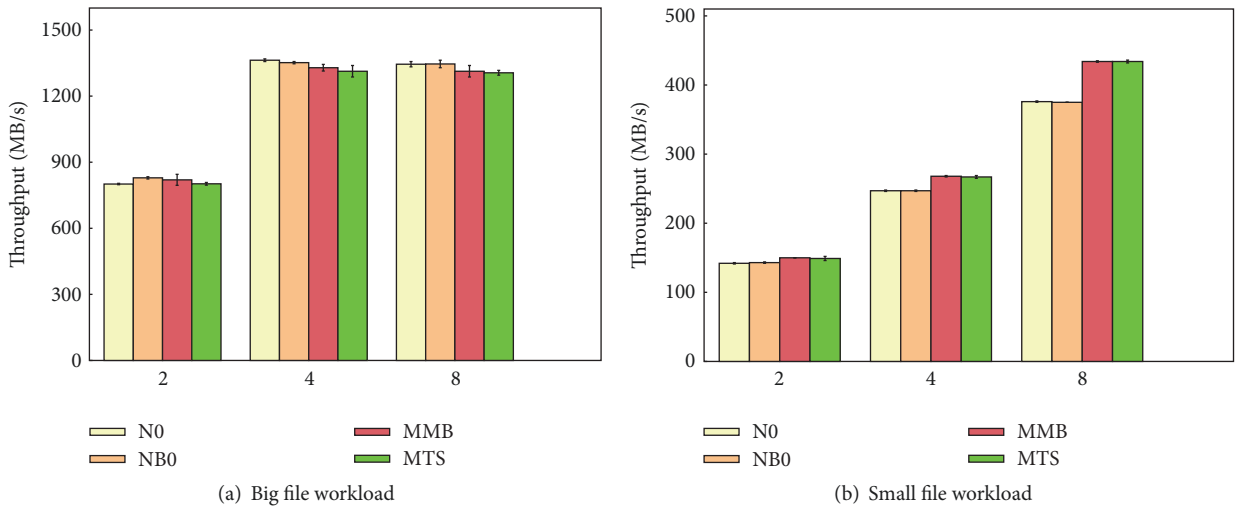


FIGURE 3: Performance comparisons for MMB and MTS with Lustre file systems. The x-axis shows the number of I/O threads. The error bar depicts minimum and maximum deviation from the average of 3 iterations.

rate in small file workload. Unfortunately, in Figure 3(a), we see that MMB and MTS have slight performance impact. In Figure 2(a), MMB and MTS improve the performance on 2000–3000 MB/s range of data transfer rate, but in small-scale Lustre file system which has just 1 OSS and 8 OSTs, MMB and MTS do not show any performance improvement. But, Figure 3(b) shows that in small file workload MMB and MTS depict performance improvement by 9.7% on average and up to 15.7% with 8 I/O threads. Because in baseline, only one pair of MT and CT sends 6,000 files metadata to other work queues sequentially, whereas in MMB and MTS, two pairs of MT and CT work on 3,000 files metadata simultaneously. It is confirmed that if the PFS I/O performance is low, it is difficult to take advantage of the NUMA effect. On the other hand, Testbed-I (memory-based network file system) represents the high-speed PFS environment and confirmed the effects of memory-level optimizations (MMB and MTS). Though the storage is not so fast that makes a bottleneck in end-to-end data transfer, dividing RMA buffer to each CPU socket increases the performance in large number of small files.

5. Related Work

The recent literatures addressing the high-speed data transfer tools include GridFTP [4], bbcp [3], and LADS [1]. bbcp transfers large amount of data efficiently using multiple TCP streams. However, bbcp uses single I/O thread and is unaware of the underlying file system layout. GridFTP [4] supports striping which allows data transmission to multiple peers when data is stored across a set of storage systems. But GridFTP does not consider the existing file system level data chunk layout of the files that are striped on the object-based parallel and distributed file systems. GridFTP and bbcp core design relies on file-based data transfer, whereas LADS [1] is a layout-aware object-based data scheduler which considers the storage layout and uses Common Communication Interface (CCI) [5] to exploit the high-speed network. Moreover, LADS implements a multithreaded architecture to benefit from parallel file system.

Along the line of advancements in storage and network, the CPU and memory performance is also improving by including the NUMA architecture. Multithreaded applications need to reflect such NUMA-awareness in their design to achieve the highest bandwidth from CPU and memory. Thread and memory locality in NUMA architecture plays a crucial role in performance. With benchmarking with STREAM triad, a research has shown that when benchmark allocates threads on a local NUMA node, the memory access is much higher than placing threads on a remote NUMA node through 1-hop and 2-hop [12]. Another study addressing NUMA-aware Thread Scheduling includes RAMSYS [2]. RAMSYS is a high-speed data transfer tool which uses dedicated threads in asynchronous fashion to incorporate pipelining in each step. But RAMSYS schedules threads without taking into account the underlying storage layout. Also, RAMSYS allocates single task queue on PFS and pushes all I/O requests to single task queue in a similar fashion to single storage device, whereas our approach uses multiple task queues as the number of OSTs in underlying Lustre file system, which enhances the performance on storage level. Our approach is an extension of existing LADS [1] architecture.

6. Conclusion

The advancement in storage, network, and CPU architecture has directed the existing data transfer software to align their design in order to achieve high throughput. Currently, the majority of data transfer tools such as GridFTP [4] and bbcp [3] do not support such design optimizations. LADS [1], a data transfer software designed for high-speed networks, considers the underlying storage architecture. However, LADS ignores the NUMA architecture. In this paper, we

propose a NUMA-aware resource and thread scheduling for optimized data transfer in high-speed network. Our approach involves three major sections, (i) distributed RMA buffer to each CPU socket, (ii) high-level Socket-based Thread Scheduling, and (iii) low-level NUMA node based Thread Scheduling. Our approach not only reduces the memory controller contention but also improves the memory access latency. The evaluation has shown improvement up to 44% for high performance file system when compared to baseline.

Disclosure

The portion of this work was presented as work in progress (1-page abstract) in the 2nd Joint International Workshop on Parallel Data Storage and Data-Intensive Scalable Computing Systems (PDSW-DISCS), held in conjunction with SC'17 [13].

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korean Government (MSIT) (no. 2015-0-00590, High Performance Big Data Analytic Platform Performance Acceleration Technologies Development). This work also used the resources of the Korea Institute of Science and Technology Information (KISTI), in Daedeok Science Town in Daejeon, South Korea. The authors thank Dr. Sungyong Park for his constructive comments that have significantly improved the paper.

References

[1] Y. Kim, S. Atchley, G. R. Vallée, and G. M. Shipman, "LADS: Optimizing Data Transfers Using Layout-Aware Data Scheduling," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies FAST'15*, 2015.

[2] T. Li, Y. Ren, D. Yu, and S. Jin, "RAMSYS: Resource-aware asynchronous data transfer with multicore systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1430–1444, 2017.

[3] A. B. Hanushevsky, "Peer-to-Peer Computing for Secure High Performance Data Copying," Tech. Rep. SLAC-PUB-9173, 2002.

[4] I. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," in *Network and Parallel Computing*, vol. 3779 of *Lecture Notes in Computer Science*, pp. 2–13, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[5] S. Atchley, D. Dillow, G. Shipman et al., "The Common Communication Interface (CCI)," in *Proceedings of the 2011 IEEE 19th Annual Symposium on High-Performance Interconnects (HOTI)*, pp. 51–60, Washington, DC, USA, August 2011.

[6] ESnet, "Energy Sciences Network (ESnet)," http://www.es.net.

[7] A. Davies and A. Orsaria, *Scale Out with GlusterFS*, Houston, TX, USA, 2013, http://dl.acm.org/citation.cfm?id=2555789.2555790.

[8] F. Wang, H. S. Oral, G. M. Shipman, O. Drokin, D. Wang, and H. Huang, "Understanding Lustre Filesystem Internals," Tech. Rep. ORNL/TM-2009/117, 2009.

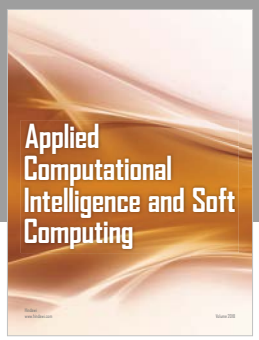
[9] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of the 7th symposium on Operating systems design and implementation. 1em plus 0.5em minus 0*, pp. 307–320, USENIX Association, 2006.

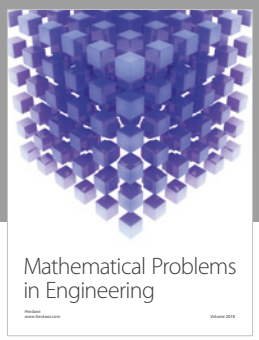
[10] ORNL, "Oak Ridge Leadership Computing Facility," https://www.olcf.ornl.gov/.

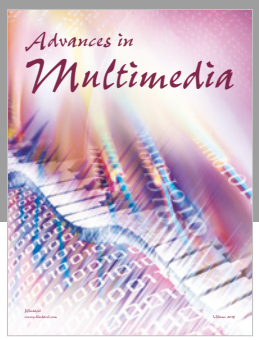
[11] Dell, *NUMA Best Practices for Dell PowerEdge 12th Generation Servers*, 2013, http://en.community.dell.com/techcenter/b/techcenter/archive/2013/01/09/.

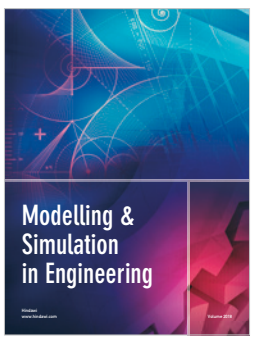
[12] T. Li, Y. Ren, D. Yu, and S. Jin, "Analysis of NUMA Effects in Modern Multicore Systems for the Design of High-performance Data Transfer Applications," *Future Generation Computer Systems*, vol. 74, pp. 41–50, 2017, http://www.sciencedirect.com/science/article/pii/S0167739X16305799.

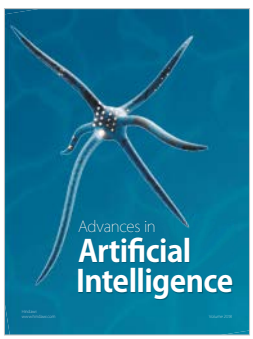
[13] T. Kim, A. Khan, Y. Kim, S. Park, and S. Atchley, "NUMA-Aware Thread and Resource Scheduling for Terabit Data Movement," in *Proceedings of the (Work In Progress) PDSW-DISCS'17 (held in conjunction with SC'17)*, p. 1, Denver, CO, USA, 2017.

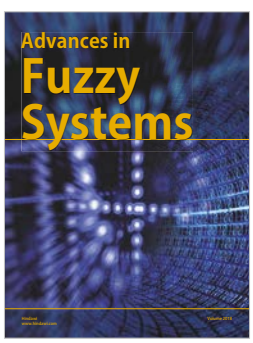


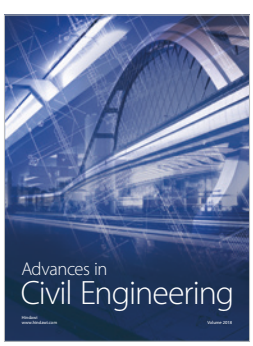


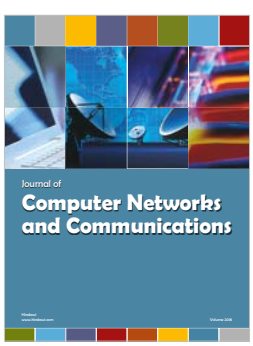


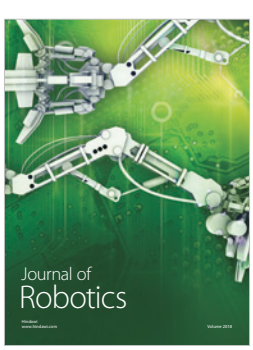


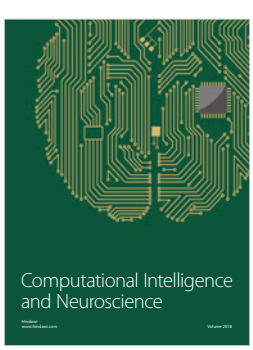












Hindawi

Submit your manuscripts at
www.hindawi.com

Copyright of Scientific Programming is the property of Hindawi Limited and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.