MDTM: Optimizing Data Transfer using Multicore-Aware I/O Scheduling

Liang Zhang, Phil Demar, Wenji Wu
Fermi National Accelerator Laboratory
Batavia, IL, USA
liangz@fnal.gov, demar@fnal.gov, wenji@fnal.gov

Bockjoo Kim
University of Florida
Gainesville, Florida, USA
bockjoo@phys.ufl.edu

Abstract—Bulk data transfer is facing significant challenges in the coming era of big data. There are multiple performance bottlenecks along the end-to-end path from the source to destination storage system. The limitations of current generation data transfer tools themselves can have a significant impact on end-to-end data transfer rates. In this paper, we identify the issues that lead to underperformance of these tools, and present a new data transfer tool with an innovative I/O scheduler called MDTM. The MDTM scheduler exploits underlying multicore layouts to optimize throughput by reducing delay and contention for I/O reading and writing operations. With our evaluations, we show how MDTM successfully avoids NUMA-based congestion and significantly improves end-to-end data transfer rates across high-speed wide area networks.

Index Terms—Big data, data transfer, network, storage, I/O, NUMA.

I. Introduction

Although Big Data has recently become a hot topic in the digital advertising and social media industries, it has been a driving force in scientific discovery for many years [1]. For example, the Large Hadron Collider (LHC) instrument is expected to produce raw data flows at levels in excess of 25GB/s [2]. The U.S. Department of Energy (DOE) supercomputing facilities (NERSC, ALCF and OLCF) annually generate hundreds of petabytes of simulation data [3]. Those extreme data volumes are typically stored in geographically dispersed facilities with large storage systems. Scientists often need to transfer large data sets from a storage system at one facility to computation resources at another facility for analysis or visualization purposes. Therefore, end-to-end data transfer rates, i.e. from source to destination storage systems, are considered to be the key performance metric that has critical impact on the efficiency and success of big data workflows. To optimize end-to-end data transfer rates, a great deal of effort has been invested in upgrading the performance characteristics of storage systems and network infrastructure. For example, DOE sites have widely adopted parallel file systems such as Luster to support high performance I/O and large data sets. Network carriers, such as DOE's Energy Sciences Network (ESnet), have upgraded their networks to 100Gb/s technologies and are planning to deploy 400Gb/s and 1Tb/s capabilities in the future [4-7]. But those storage system and network infrastructure enhancements only contribute to improvement in data transfer rates either on network links or in local storage systems. They do not necessarily lead to a matching improvement in overall end-to-end performance. To optimize end-to-end performance, many science research sites deploy data transfer nodes (DTNs), which are purpose-built Linux servers dedicated to the function of wide area data transfer [8]. DTNs connect the wide area network services to the local storage systems, moving data from the local storage system to remote site over the network and vice versa. As the pivot points in the end-to-end data movement path, DTNs, and the data transfer tools they are using to execute data movement jobs, play a key role in optimizing end-to-end data transfer rates.

To date, data transfer tools such as GridFTP [9-10] and BBCP [11] are commonly deployed on DTNs to support bulk data movement. These tools implement many useful data transfer features, including transfer resumption, partial transfer, third-party transfer, and security infrastructure services. There have been numerous state-of-art enhancements in the current generation of data transfer tools to speed up performance. Parallelisms at all levels (e.g., multi-stream parallelism [9] and multi-path parallelism [13-16]) are now widely implemented for bulk data movement, providing significant improvement in aggregate data transfer throughput. However given the ever increasing data volumes and data movement needs, these data transfer tools may not sufficiently answer the emerging challenges for data movement in big data environments. Even today, data transfer rates are often low enough that only a modest fraction of the network bandwidth available for the transfer gets utilized. In particular, the time to transfer *the lots of small files (LOSF)* [17] is much longer than that to transfer a single file with the same aggregate size, despite optimization efforts such as multi-streaming and pipelining. It is critical to improve data transfer tools' performance on DTNs to optimize end-to-end data movement in big data environments.

This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

The emergence of multicore platform architectures has provided another avenue for improvement in data transfer rates. Today, it is normal for DTN servers to support 16, 24, or even more cores on the motherboard. The memory on the board has expanded to tens of Gigabytes, and can be efficiently accessed with NUMA technology. Disk I/O and network I/O also has many new features. For example, a network interface card (NIC) can use Intel's flow director technology to enable multiple cores to process individual network flows in parallel, thus attaining higher aggregate throughput. With more performance hardware, there is pressure on the software side to be more intelligent and efficient, to take advantage of these hardware enhancements. For example, multiple cores increase opportunities for parallelism, but contention on the interconnections between NUMA nodes could drag throughput down if related threads are misplaced. Similarly, flow director on a NIC could steer individual traffic flows to the cores where their application resides, but the possible OS load balancing can still cause packet-reordering, and result in degraded TCP performance. The current generation of OSes has limited capabilities for such resource management, and may even induce adverse impacts on performance. Therefore, it is up to the data transfer applications themselves to manage the underlying resources as efficiently as possible, particularly the storage and network I/O operations, to achieve high performance. These observations strongly motivate us to develop a new data transfer framework that optimizes data transfer performance on DTN platforms.

We investigate the issues related to designing a data transfer tool that can fully exploit zero-copy, multiple threads, pipeline, and underlying hardware features. In particular, we focus on the scheduling of the threads performing disk I/O and network I/O, to achieve high throughput performance. Our efforts seek to avoid NUMA interconnect contention and delay while protecting from the potential negative impacts of OS load balancing. With these design considerations, we develop Multicore Aware Data Transfer (MDTM) [12].

In this paper, we present MDTM, a bulk data movement tool for use on DTNs to provide high end-to-end data transfer rates across wide area networks and local storage systems. Our primary contribution is a multicore aware scheduler that accommodates I/O reading and writing tasks and assigns I/O threads to specific cores and NUMA modules for optimal throughput. By contrast, existing data transfer tools depend on the OS to place the I/O threads to certain NUMA nodes and those threads might migrate to cores, typically based on load considerations. As a consequence, I/O threads are not specifically placed close to the I/O device controller. Longer delay for reading and writing I/O devices, as well as traffic contention on the NUMA interconnects potentially degrade the end-to-end performance. Our proposed scheduler, on that other hand, takes locality into account when making the allocation decision, and favors I/O operations executed on the same NUMA node as the source/destination devices. Similarly, the memory allocation follows the CPU-affinity principle, with the threads sharing memory are assigned to the same node to avoid inter-node traffic.

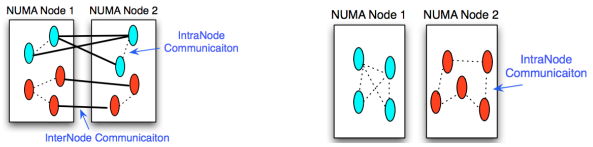
In the final section, we conduct comprehensive evaluations of our MDTM software package, using DTNs at Fermilab (Chicago) and the University of Florida, which are connected via ESnet wide area network. We evaluated the data transfer performances in typical scenarios as well as with various transfer loads similar to realistic transfer tasks. We compare our results with GridFTP. Our evaluations show MDTM software can achieve significantly higher performance than current generation data transfer tools.

II. Background

We first introduce the multicore platform for bulk data movement applications. Next we define the I/O optimization challenges in the scheduling data reading and writing tasks.

A. Inefficiency of OS scheduling for Multicore and NUMA Systems

Mainstream OS schedulers apply thread-independent scheduling policies, which schedule threads independently, regardless of application types and dependencies. Periodically, an OS scheduler will balance load across cores to facilitate load balance. A busy data transfer server, using a multi-stream data transfer tool such as GridFTP or BBCP, will often handle hundreds or thousands of parallel data flows. When a data transfer request arrives, multiple threads are created and distributed across the cores. On



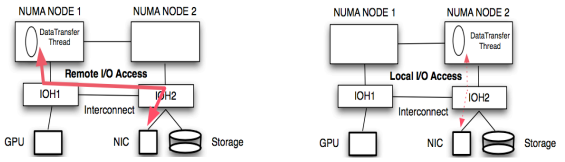
A. Independent scheduling B. Dependent scheduling

Figure 1. Independent vs. Dependent Scheduling

NUMA-based systems, the thread distribution may not account for the NUMA architecture. Even after threads have been created, the kernel may run dynamic load balancing and migrate threads to different cores. Threads that need to access large amounts of data from remote NUMA nodes, may suffer performance degradation due to remote latency overhead and NUMA interconnect contention [18]. Figure 1 shows such a scenario, with two separate data transfers having their threads distributed across NUMA nodes. Default OS scheduling (independent scheduling) may result in many high-latency inter-node communications (solid lines). If threads sharing the same data can be clustered and scheduled on the same NUMA node, communication overheads (dashed lines) can be significantly reduced.

Bulk data transfers involve resource-consuming network I/O operations on the end systems. Investigations indicate that on multicore systems, efficient network I/Os require CPU core affinity on network processing [19]. Figure 2

illustrates the differences in scheduling with and without I/O locality support. Core affinity can significantly increase system performance in terms of reducing contention for shared resources, minimizing software synchronization overheads, and enhancing cache efficiency. However, with existing NIC technologies, multicore systems have difficulty supporting core affinity in network I/O processing. For example, Receive side scaling (RSS) [12] lacks a critical data steering mechanism to automatically steer incoming network data to the same core on which its application thread resides. This absence causes inefficient cache usage whenever an application thread is not running on the same core that RSS has scheduled the received traffic to be processed. The result is degraded performance. Another example is Intel's Flow Director technology that can cause packet reordering if process (or thread) migration occurs [21]. Again, degraded performance will normally be the result. Research results over the past several years clearly show that traditional OS scheduling constrain performance on multicore systems, and in particular on NUMA systems. A more deterministic and application-sensitive approach to scheduling is needed.



A. Scheduling without I/O locality support B. Scheduling with I/O locality support

Figure 2. Scheduling without and with Locality

B. Related Work

D. Tam [22] proposed a sharing-aware scheduling strategy on multicore systems. The researchers designed and implemented a scheme to schedule threads based on sharing patterns detected online before locating sharing threads onto the same processor. Experiments demonstrated the effectiveness of the sharing-aware scheduling strategy. Co-scheduling is another type of thread-dependent scheduling scheme. Co-scheduling is a scheme that schedules related threads or processes to run on different processors simultaneously. Thus far, thread-dependent scheduling mostly lies in the research realm.

Studies on NUMA I/O shows that I/O throughputs can be significantly improved if an application's threads can be placed on cores near the I/O devices they will utilize [19, 20]. However, these studies only consider one type of I/O devices. In the real world, applications typically need to access multiple and different types of I/O devices, including NIC cards and disks. If these I/O devices are located on different NUMA nodes, scheduling for I/O locality becomes much more complicated.

The authors in [23] discuss scheduling policies that address shared resource contention on NUMA systems. Their scheduling policies are built on a classification scheme for threads, and addresses contention in the memory subsystem.

With the proliferation of NUMA systems, user-level control of thread and memory placement has proven to be effective. Many large applications with high numbers of concurrent threads have chosen to handle system resources (thread scheduling, memory placement) within the application itself. OSes such as Linux, Windows, and FreeBSD provide scheduler affinity APIs allow user space applications to specify that threads be pinned to particular cores, and memory be placed on specific NUMA nodes. Using Linux scheduler affinity APIs, S. Blagodurov et al. developed an online user-level scheduler to test the efficiency of scheduling algorithms on real multicore systems [24]. In addition, to facilitating multi-threaded parallel programming, researchers have developed parallel programming runtime libraries. These runtime libraries insulate users from the complexity of user-level control of thread and memory placement.

III. Design of MDTM

MDTM is motivated to address performance bottlenecks on DTNs for large data transfers. In this section, we describe our design rationale behind the MDTM implementation. We also describe the key technologies used in MDTM's multicore-aware I/O scheduling.

Our design goals are: 1) reduce the I/O and memory access delays; 2) prevent traffic contention over the NUMA interconnects; and 3) prevent packet loss on the NIC.

A. Pipelined I/O Centric Design

Data transfer applications are I/O intensive. They read data from source disks or network NICs, and then write to the destination disks or network NICs. It is normal practice for the current generation of data transfer tools to use many threads for execution of parallel I/O operations on different cores to achieve higher throughput. However, we cannot ignore the fact that every I/O controller has the maximum capacity. Too many threads requesting concurrent services from the same I/O controller will saturate the I/O controller, or cause performance degradation from excessive contention. With that in mind, we argue that only a small number of threads should be used to access a specific I/O device simultaneously. The number of threads should be determined by the specific I/O device's capacity. Furthermore, we divide a single transfer task into two parts: reading from the source, and writing to the destination. We make those two sub-tasks work in pipeline fashion by assigning them to different threads. MDTM implements the above considerations by adopting a pipelined I/O centric design.

As shown in Figure 3, MDTM pre-allocates certain number of dedicated I/O threads depending on that specific device's I/O capacity. Typically, four types of I/O threads will be created by the MDTM:

- Disk/storage reader threads to read data from disks or storage systems.
- Disk/storage writer threads to write data to disks or storage systems.
- Network sender threads to send data to networks via NIC.
- Network receiver threads to receive data from network via NIC.

In addition to those I/O threads, MDTM also creates a management thread to handle user requests and management-related functions.

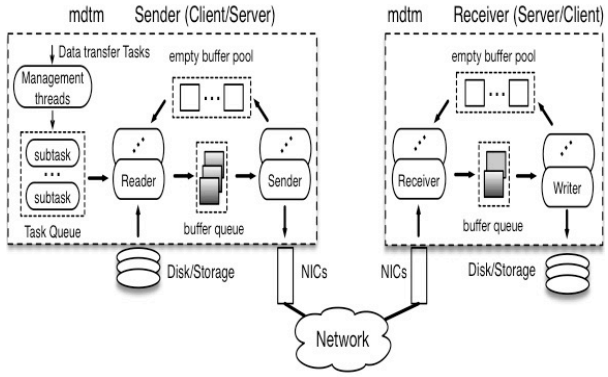


Figure 3. Pipeline I/O Centric Design

After determining the number of threads used for the specific I/O device, MDTM then calls its I/O scheduling service to assign those I/O threads to specific cores. We will discuss the design of the MDTM scheduler in the next section. After the scheduler selects the cores and places those threads on those cores, MDTM pins each thread to the chosen core. This design has two major goals: (1) enforcing I/O locality on NUMA systems; and (2) preventing process migration of I/O threads, thus preserving core affinity. The MDTM design reduces I/O operations' delay by preserving I/O and core affinity, as well as protecting against the adverse effects of kernel load balancing on I/O operations.

Typically, an MDTM I/O thread is dedicated to a specific core. No other threads will be scheduled on that core while the I/O operation is under execution. The MDTM scheduler partitions system cores into two zones – MDTM zone and non-MDTM zone. MDTM I/O threads runs in the MDTM-zone, while other applications threads are confined to the non-MDTM zone. With this design, MDTM eliminates other applications' threads contending with MDTM threads, thus resulting in optimum data transfer performance for an MDTM application. It should be noted that the MDTM management thread is assigned to the non-MDTM zone, since management thread performance considerations are minimal.

Heavy data transfer tasks involve a significant amount of memory buffer operations. To avoid costly memory allocation/de-allocation along the critical path for data transfer, MDTM pre-allocates multiple data buffers, and manages them in a data buffer pool. More importantly, data buffers are pinned and locked to avoid being paged to the swap area or involved in memory migration. Data buffers are recycled and reused.

MDTM executes data transfers in a pipelined manner. In the sender, the management thread first preprocesses the data transfer request. A data transfer task is typically split into multiple subtasks. A subtask can comprise file segments, a group of files, or file folders. Subtasks are then put in a task queue. Disk/storage reader threads keep fetching subtasks from the task queue. For each subtask, data will be first fetched from storage/disk(s) into empty buffers. Filled data buffers are temporarily put in a buffer queue. Concurrently, network sender threads continue fetching filled data buffers from the buffer queue, and sends data to the network in parallel on multiple TCP streams. In the receiver, data will be received into empty buffers via network receiver threads; afterwards, the filled buffers will be passed over to storage/disk writer threads to move the data into storage/disk(s).

B. Application-Level I/O Scheduling

The placement of I/O threads and memory is the critical decision in achieving high performance. Current data transfer applications depend on the operating system (OS) to schedule the user threads and allocate memory. Most existing OS schedulers use a distributed run-queue model, in which the scheduler maintains one run queue per core. The scheduler applies a thread-independent scheduling policy, which schedules threads independently, regardless of application types and dependencies. Periodically, the scheduler balances the load across cores to facilitate system load balance. In the case of NUMA systems, the load balancing is across all NUMA nodes. Such dynamic load balancing may result in frequent thread migration, leading to higher-latency inter-node communications, and an overall degradation in data transfer performance. Furthermore, I/O devices (e.g., NIC and storage) on NUMA systems are connected to processor sockets in a NUMA manner. This results in NUMA effects for transfer between I/O devices and memory banks, as well as CPU I/O access to I/O devices. Investigations show that I/O throughputs can be significantly improved if applications can be placed on cores near the I/O device they use (i.e., I/O locality) [19, 20]. However, existing OSes have very limited support for such IO locality. Processes/threads may end up being scheduled on cores that are distant from the I/O devices they use, leading to high-latency inter-node I/O operations and incurring extra communication overheads. Bulk data transfers involve significant network and disk I/O operations. Thus, using default OS scheduling can lead to significant inter-node I/O operations and severely degrade the overall data transfer performance.

In contrast, MDTM does not totally depend on the OS to schedule I/O threads. It has its own I/O user-space scheduler. The MDTM I/O scheduler uses a novel algorithm to calculate the optimal cores for each I/O thread, and then binds those threads and their memory to the

assigned cores by using 3rd party tools like *libnuma or numactl*. Since the MDTM application has sufficient knowledge about the device(s) where the I/O threads are going to access, the scheduler can leverage those application hints to better place I/O threads than a generic OS scheduler.

In the MDTM code, the function of *mdtm_io_schedule()* is called to perform the scheduling process. The arguments of that function include the I/O thread to be placed, and the device the I/O thread is associated to. If the function returns successful, the I/O thread becomes bound to the core projected to achieve optimal performance.

There are several factors that the MDTM scheduler must take into consideration, including: the locality of cores to the I/O device; the traffic load over the NUMA interconnects; and the load of cores; memory locations, etc. To quantify those key factors, we introduce a cost function for each resource on the path from the I/O device to each of the cores. We assert this I/O scheduling issue can be abstracted as the shortest path problem in graph theory.

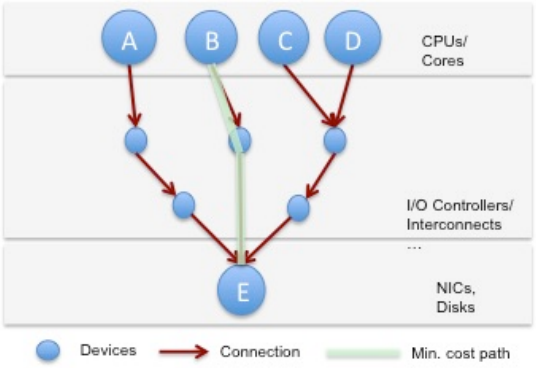


Figure 4. Minimum Cost Path Scheduling

As shown in Figure 4, vertices represent the shared resources in the DTN system, including cores, I/O controllers, network NICs, and disks; while the edge represents the connection between two adjacent resources. A path is a sequence of vertices from the I/O device to the target core,

$P = (v_1, v_2, ..., v_n)$, where v_i is vertices.

Let $e_{i,j}$ be the edge connect vertices v_i and v_j. Given the cost function $f : E \rightarrow R$, the shortest path from core vertices to I/O device vertices is the path that minimize the sum,

$$\min\left\{\sum_{i=1}^{n-1} f\left(e_{i,j}\right)\right\} \quad (1)$$

Among several solutions to the shortest path problem, we choose the Dijkstra's algorithm. Although its complexity is $O(V^2)$, we argue that V will usually be small. The coding of Dijkstra's algorithm is comparatively simple. The function of *mdtm_io_schedule()* implements a route that calculates the path with the minimum cost and the core at other end of the path is the target core to run the given I/O thread. In Figure 4, core B is chosen as the target core to run the I/O thread, which interacts with the device E.

The MDTM scheduler would call *mdtm_io_schedule()* as soon as the application starts, to setup all the I/O threads. The MDTM scheduler monitors system conditions by periodically scanning and updating system's statistics, including core loads, NUMA memory missing rates, etc. Whenever the scheduler detects system performance degradation, it can call *mdtm_io_schedule()*to reschedule the I/O threads, in the hope of improving performance.

IV. Evaluation

For evaluation of the MDTM, we use the experimental systems setup in two remote sites: Fermilab at Chicago and University of Florida at Florida, which are connected via the DOE ESnet network. First, we show the results of transferring one large single file from Fermilab to University of Florida. We explore the effectiveness of application-level I/O scheduling based on MDTM versus pure-OS scheduling as used by GridFTP. We investigate both disk-to-disk transfer and disk-to-memory transfers. Then, we show the results from group of files transfer by using the Linux source tree folder. We study the impact of virtual large file mechanism in MDTM and compare with request-respond file transfer in GridFTP. All the evaluation results prove MDTM provides significant improvement in end-to-end data transfer rates.

A. Experimental Environment

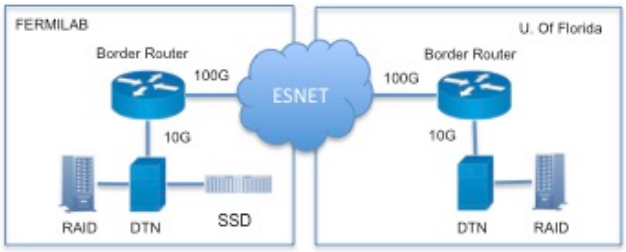
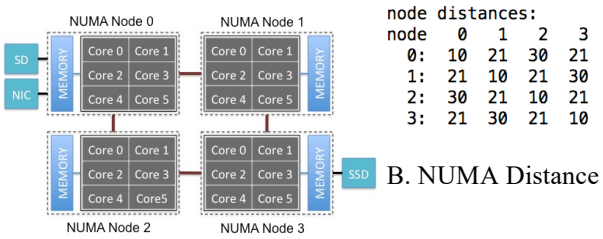


Figure 5. Experimental Systems

In the experimental environment shown in Figure 5, the two DTNs are located at Fermilab (Chicago) and the University of Florida respectively, and connected via ESnet. The ESnet backbone is 100Gbps, and the round trip time between Fermilab and Florida is about 30 milliseconds. The DTNs on both sites utilize 10GE NICs. In our data transfers, the DTN at Fermilab serves as the data source, while the Florida DTN works as the data destination.

The Fermilab DTN, illustrated in Figure 6, hosts four Intel Xeon E5-4607 CPUs each of which has 6 cores. Its 64GB DRAM memory is evenly dispersed in four NUMA nodes. With the command line tool, *numactl*, the NUMA distance looks quite variable for each pair of nodes: the distance between the farthest pair is almost triple that of the node pair in neighbor. The node distance reveals that

choosing local node versus remote nodes will have severe impact on the memory access performance. Besides memory, the I/O devices have different affinities to different NUMA nodes: the RAID disk system is connected NUMA node #0, the SSD disk is connected to node #3 and the 10GE Ethernet NIC is affiliated to NUMA node #0. The DTN at the University of Florida doesn't have a multi-NUMA architecture, with only one Xeon CPU and 4 cores, a 10GE Ethernet NIC and one RAID storage system. Both DTNs are running Linux operating system.



B. NUMA Distance

A. NUMA Architecture

Figure 6. FNAL DTN System

For a realistic performance evaluation of data transfer job, two heuristic types of file transfer jobs are used to measure the end-to-end data transfer rates. The first type of job is to transfer one single 100GB long file and the second is to transfer a Linux source tree folder with comparatively smaller files. The Linux folder contains about 60000 files of varying lengths and types, including regular, symbolic link and directories. The single file tests are designed to evaluate the performance of the data transfer for very large file. The folder transfer tests could be ideal to estimate the performance on LOSF transfer jobs.

B. Single 100GB File Transfer From Disk To Disk

In this section, we will show the effectiveness of the MDTM using the application-level I/O scheduler, versus data movement software like GridFTP that depends on the OS to perform the scheduling.

We evaluate the end-to-end throughput of MDTM for transferring one 100GB long file from Fermilab to Florida and compared it with GridFTP. In both sets, we use the same the block sizes, multiple streams and the default configurations for the network sockets. The 100 GB file is stored on the SSD disk of Fermilab DTN, which internally support parallelism.

Figure 7 shows the results for different working threads or processes. We repeated the test 10 times, and the variations are small. In MDTM, the multiple reading and sending I/O threads work in a pipeline manner and transfer different parts of the 100GB in parallel. We can increase the number of those threads to maximize the utilization of cores. In GridFTP, multiple processes are created to work on different parts of the same large file, which can be controlled via the command line option "concurrency". Given the multiple cores, SSD disk and high performance NIC, both MDTM and GridFTP could benefit from the multiple I/O threads or processes implementation.

The results in Figure 7 show that the MDTM outperforms GridFTP, gaining between 13% and 33% higher throughput across test cases. Because MDTM and GridFTP have similar configuration and implementation in most parts, we attribute the performance gain to the I/O

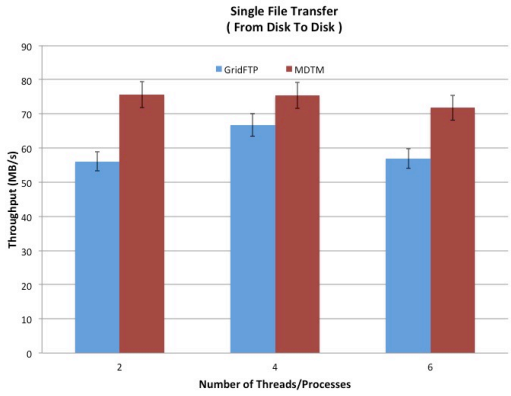


Figure 7. Disk to Disk Throughput

centric design; the MDTM I/O scheduler optimizes the data transfer rates over the traditional OS scheduling.

However, we also observe in Figure 7 that the overall throughputs are low compared with the underlying 10GE NIC and SSD disk. Furthermore, neither MDTM nor GridFTP demonstrates much throughput improvement with respect to the increased number of threads or processes. It seems to reflect a bottleneck on the path from Fermilab DTN to Florida DTN.

We then identify the two factors that could drag down the overall throughput in our experimental systems. First, the disk system on Florida DTN has writing speed around 80MB/s, which constrains the throughput for both MDTM and GridFTP. Secondly, the DTN at University of Florida only has one NUMA node to which all the memory and I/O device are hooked up. Since the MDTM I/O scheduler is designed to take advantage of the underlying NUMA infrastructure to optimize the data transfer rate, the Florida DTN might not receive much benefit from the MDTM I/O scheduler.

C. Single 100GB File Transfer From Disk To Memory

In this section, we show the effectiveness of MDTM with I/O scheduler transferring one 100GB file from Fermilab's SSD disk to the memory of the Florida DTN. We explore the MDTM I/O scheduling effectiveness via bypassing the slow disk on the Florida DTN. The results are shown in Figure 8.

First, we observe that both MDTM and GridFTP see large jumps of throughput values: 5 times for GridFTP and > 10 times for MDTM. That verifies the impact of the low disk of Florida DTN on the end-to-end throughput tests in the previous section. Second, the I/O centric strategy and scheduling algorithm in MDTM outperforms GridFTP in throughput by the multiples: 2.7, 3.29 and 3.57, in the cases of running on two, four and six threads respectively. Third, the throughput of MDTM increases linearly with increased

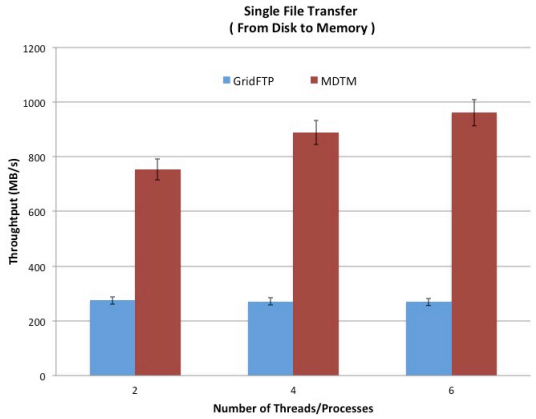


Figure 8. Disk to Memory Throughput

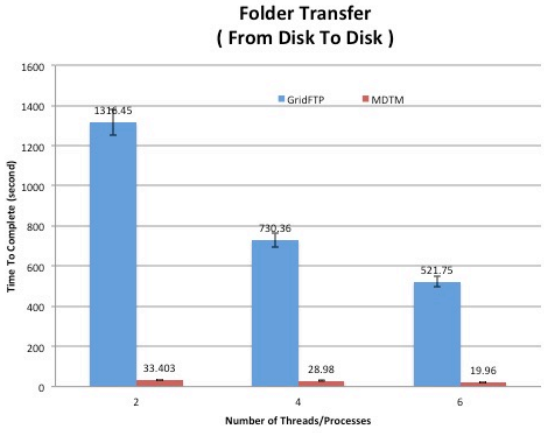


Figure 9. Complete Time for Folder Tranfer

number of threads, while GridFTP does not, which shows MDTM has better CPU utilization.

Last, but not least, MDTM shows much higher network utilization versus GridFTP. Given the underlying 10 Gbps networking bandwidth, the GridFTP uses less than 20% of the available bandwidth, while MDTM's network utilization is almost 80%.

TABLE I. Network Bandwidth Utilization

Threads	2	4	6
GridFTP	22%	21.6%	21.6%
MDTM	60.2%	71.1%	76.8%

D. Group of Files Transfer

In the previous section, we show the effectiveness of I/O scheduling based MDTM in transferring one single large file compared with the OS-dependent GridFTP. In this section, we show the effectiveness of MDTM scheduling in transferring a group of files.

The group of files we use in our test is the source tree of Linux code that contains 56,208 files, and covers a wide spectrum of file types, including regular files, links, directories and etc. The total size of the Linux source tree is about 763MB. Figure 9 shows the time comparison results for transferring the Linux folder, using both the GridFTP and MDTM applications. Again, we repeat the transfers ten times for each case to validate consistency, using 2, 4 and 6 processes. The times to complete the folder transfer job are recorded as the performance metric. As shown in Figure 9, it takes a very long time for GridFTP to finish the folder transfer, given the folder's relatively small total size. On the other hand, MDTM's time to complete the transfer is very short, relatively speaking for each case. From Figure 9, MDTM appears to be 20 ~ 40 faster than GridFTP in transferring the Linux folder.

Figure 9 demonstrates the significant performance improvement of MDTM versus GridFTP in transferring groups of files. However, we do note that the MDTM software uses a virtual large file mechanism [25], which contributes to the performance gain as well. We developed that virtual large file mechanism to facilitate higer performance of group file transfer capabilities for MDTM. Each file in the group is treated as one small part of a single large virtual file. The information about those files are recorded in a separate metadata file. Then, the transfer of the group of files is condensed to transfer of two files: metadata and data files. The large virtual file mechanism optimizes performance by: (1) eliminating protocol processing between the sender and receiver on a per-file basis; and (2) allowing for batch processing of small files in the sender and receiver. MDTM I/O scheduler plus the large virtual file mechanism makes MDTM a much faster data movement application for transfers of group of files than current data movement applications.

V. Conclusion

Moving large-scale bulk data between geographically distributed locations is a challenging problem, which may constrain scientists analyzing and visualizing shared experimental data. In this paper, we identify the performance bottlenecks in the data transfer between DTNs and propose a novel I/O scheduling algorithm, as well as the large virtual file transfer mechanism, to improve the end-to-end data transfer performance issues. The results from our set of tests demonstrate our novel application-level I/O scheduler can optimize the data transfer on multicore and NUMA platforms.

Acknowledgment

DOE's Advanced Scientific Computing Research (ASCR) office has funded Fermilab to work on Multicore-Aware Data Transfer Middleware (MDTM) [12].

References

[1] "Synergistic Challenges in Data-Intensive Science and Exascale Computing", DOE ASCR Data Subcommittee Report 2013.

[2] https://home.cern/about/computing/processing-what-record

[3] TORRELLAS, J. Architectures for Extreme-Scale Computing. Computer 42, 11 (Nov. 2009), 28–35.

[4] Eli Dart, Mary Hester, Jason Zurawski, "Fusion Energy Sciences Network Requirements Review - Final Report 2014", ESnet Network Requirements Review, August 2014, LBNL 6975E

[5] Eli Dart, Mary Hester, Jason Zurawski, Editors, "High Energy Physics and Nuclear Physics Network Requirements - Final Report", ESnet Network Requirements Workshop, August 2013, LBNL 6642E

[6] Eli Dart, Brian Tierney, Editors, "Biological and Environmental Research Network Requirements Workshop, November 2012 - Final Report"", November 29, 2012, LBNL LBNL-6395E

[7] David Asner, Eli Dart, and Takanori Hara, "Belle-II Experiment Network Requirements", October 2012, LBNL LBNL-6268E

[8] https://fasterdata.es.net/science-dmz/DTN/

[9] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, and S. Tuecke, "GridFTP: Protocol Extension to FTP for the Grid," Grid Forum Internet-Draft, Mar. 2001.

[10] B. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu and I. Foster, "The Globus Striped GridFTP Framework and Server," SC'2005, 2005.

[11] BBCP, http://www.slac.stanford.edu/~abh/bbcp/

[12] http://mdtm.fnal.gov

[13] Han, Huaizhong, et al. "Multi-path tcp: a joint congestion control and routing scheme to exploit path diversity in the internet." IEEE/ACM Transactions on Networking (TON) 14.6 (2006): 1260-1271.

[14] Wang, Bing, et al. "Application-layer multipath data transfer via TCP: schemes and performance tradeoffs." Performance Evaluation 64.9 (2007): 965-977.

[15] Iyengar, Janardhan R., Paul D. Amer, and Randall Stewart. "Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths." Networking, IEEE/ACM Transactions on 14.5 (2006): 951-964.

[16] Gunter, Dan, et al. "Exploiting network parallelism for improving data transfer performance." High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:. IEEE, 2012.

[17] Bresnahan, John, et al. "Gridftp pipelining." Proceedings of the 2007 TeraGrid Conference. 2007.

[18] S. Akram, M. Marazkis, and A. Bilas, "NUMA Implications for Storage I/O Throughput in Modern Servers," In 3rd Workshop on Computer Architecture and Operating System co-design (CAOS'12), Paris, France, January 2012.

[19] S. Moreaud, B. Goglin, "Impact of NUMA Effects on High-Speed Networking with Multi-Opteron Machines," In PDCS 2007, Cambridge, Massachussetts (2007)

[20] Intel White Paper, "Improving Network Performance in Multi-Core Systems", 2007

[21] Wu, Wenji et al, "Why Does Flow Director Cause Packet Reordering?", In IEEE Communication Letter (2011)

[22] D. Tam et al. "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors", EuroSys, 2007

[23] S. Zhuravlev, "Addressing shared resource contention in multicore processors via scheduling", ASPLOS, 2010

[24] S. Blagodurov, "User-level scheduling on NUMA multicore systems under Linux", Proc. Of Linux Symposium, 2011

[25] Liang Zhang, Wenji Wu, Phil Demar et al. "The Evaluation of MDTMFTP on ESnet SDN Testbed", Future Generation Computer System, Coming soon