Resources-conscious Asynchronous High-speed Data Transfer in Multicore Systems: Design, Optimizations, and Evaluation

Tan Li, Yufei Ren, Dantong Yu*, Shudong Jin
Stony Brook University, Stony Brook, NY, 11794, USA
{tan.li,yufei.ren,shudong.jin}@stonybrook.edu
*Brookhaven National Laboratory, Upton, NY 11973, USA
dtyu@bnl.gov

Abstract—One constant challenge in multicore systems is to utilize fully the abundant resources, while assuring superior performance for individual tasks, particularly, in Non-uniform Memory Access (NUMA) systems where the locality of access is an important factor. To achieve this goal requires rethinking how to exploit parallel data access and I/O related optimizations. In the context of developing software for high-speed data transfer, we offer a novel design using asynchronous processing, and detail the advantages of resources-conscious task scheduling. In our design, multiple sets of threads are allocated to the different stages of the processing pipeline based on the capacity of resources, including storage I/O, and network communication operations. The threads in these stages are executed in an asynchronous mode, and they communicate efficiently via localized mechanisms in NUMA systems, e.g., task grouping, buffer memory, and locks. With this design, multiple effective optimizations are seamlessly integrated particularly for improving the performance and scalability of end-to-end data transfer. To validate the benefits of the design and optimizations therein, we conducted extensive experiments on the state-of-the-art multicore systems. Our results highlighted the performance advantages of our software across different typical workloads, compared to the widely adopted data transfer tools, GridFTP and BBCP.

Keywords—Multicore systems; Input/Output; High-speed data transfer; Parallelism; Asynchronous processing

I. INTRODUCTION

Data-intensive applications often demand advanced computer architectures with a large number of cores, a deep memory hierarchy, and ultra high-speed input/output (I/O), including access to storage and the communication network stack. These applications frequently rely on the high-speed data transfer tools that can adapt to large-scale computer nodes to move efficiently an increasing large volume of data. Usually, data transfer plays a dominant role in the overall time and cost for executing the application [1]. Hence, highly efficient data transfer systems urgently are needed to exploit the available systems’ parallelism and concurrency, so to maximize the full potential of abundant hardware resources. However, achieving this end is often not trivial and involves several challenging issues:

1) Consistent high performance across workloads: Delivering a large volume of content, e.g., scientific data and high-resolution images and videos, consume the most bandwidth among all applications in the Internet today. To expedite bulk data transfer can ensure highly efficient usage of data path. On the other hand, small network transactions dominate all transactions to the online services with backend data storage systems, e.g., key-value stores. To meet the requirement for latency for individual users demands expediting the processing speed of these small data requests. The real workload is the mix of these two types. A good data transfer system should guarantee consistent performance for different workloads.

2) Effective NUMA-aware support: The Non-uniform Memory Access (NUMA) architecture is ubiquitous in contemporary computer systems. The increasing performance gap between on-chip data sharing and off-chip data transfers necessitates having NUMA-awareness for high-speed data transfer, as we showed in our previous study [2]. Existing data transfer tools rely on the default OS behavior which often adopts a thread-independent task dispatch, i.e. it dynamically chooses the CPU on which the user-space thread will run, and potentially will move that thread later to a different NUMA node so to balance the system load. Thread scheduling and migration that ignores the characteristics of NUMA architecture will affect the performance of I/O-intensive applications. Therefore, instead of using default OS scheduling, high-speed data transfer software must resort to implementing its own thread-dependent scheduler. As shown in Figure 1, an application is more familiar with its own threads and I/O devices in use than the OS, and thereby can schedule all its parallel cooperative threads and data to the NUMA node that is aligned to its target devices. This ability significantly reduces the overhead of inter-node communication, and improves both the application’s performance and the system’s overall efficiency.

3) Parallelism of system I/O resources and the data transfer pipeline: Modern high-end systems often are equipped with multiple host adapters for both network communication and data storages. For full utilization, each type of I/O and its associated adaptor may exhibit unique behavior under various access patterns, e.g. kernel-bypass and parallel access. To obtain the maximum aggregate performance, the
data transfer software needs efficient parallel accesses to those resources with an adaptive access pattern. For this purpose, the software also must incorporate effective new designs, e.g. multithreading, multi-process, and event-driven, to assure that its end-to-end data transfer pipelines can run in parallel, and thus are capable of scaling their performance up to multi-core systems.

Although there have been many attempts to take advantage of the multi-threading and multi-streaming techniques, the existing software designs still lack efficiency. For example, GridFTP [3], a popular data transfer tool in grid computing, uses a single thread to handle both the network and storage I/Os, and does not capitalize on the multi-core parallelism [4]. The BBCP software [5], another widely adopted data movement tool in the DOE complex, employs a single thread for storage I/O, regardless of the characteristics of the storage device. Our previous work on RFT tool [4] relies on specific RDMA hardware to offload data transfer tasks to the I/O devices. More importantly, they do not incorporate the asymmetrical NUMA architecture of the multi-core servers, and so can not attain optimal performance. Thus, a new design is indispensable to ensure the best performance within each component of the data transfer pipeline.

In this paper, we describe our novel research and development of the efficient high-speed data transfer software, targeting the aforementioned challenges. Specifically, we present a scalable framework for executing tasks that (1) uses system profiling, abstracts I/O resources and pre-allocates capacity-aware thread pools to reuse thread and optimize resource utilization globally, (2) employs a multi-staged asynchronous request processing pipeline and capacity-based resource assignment methodology to maximize the system’s parallelism and aggregate performance, (3) provides a storage-centric task mapping and NUMA-aware thread scheduling to ensure affinitized data movement and communications, (4) offers an extensible structure to integrate various I/O tuning techniques. Subsequently, we implemented the framework of the Multicore-aware Data Transfer (MCDT) utilities, and showed how it seamlessly integrates multiple optimization strategies to attain high performance consistently. Finally, we undertook a comprehensive comparison of our proposed MCDT and the state-of-the-art GridFTP and BBCP software. The evaluation confirmed that the MCDT software is the best-performing data transfer system that requires no special hardware support.

The remainder of this paper is organized as follows: Section II presents the research background. The framework design is detailed in Section III. In Section IV, we describe the reference implementation of our design, termed MCDT, and its integrated optimization strategies. Section V provides the evaluation and analytical results, followed by our conclusion and plans for future work.

II. BACKGROUND AND RELATED WORK

The line rates of commodity Ethernet already reached 100 Gbps in the market of switches and routers, and 40 Gbps in server adaptors [6]. To keep up with this rapid advancement in the network hardware, current high-speed data transfer must be redesigned to efficiently support the multicore architecture. The work described in [7] showed that network processing at the receive side of 10 Gbps Ethernet could easily overwhelm the capacity of two cores in an Intel Xeon Quad-Core processor. Other research [8] revealed that the benefit of a multi-core system could be nullified by not considering various affinities along the end-to-end transmission flow.

Affinity control is one effective way to harness multiple cores for high-speed data transfer. Many studies depend on the techniques, such as Receive-side Scaling (RSS), Receive Packet Steering (RPS) and Receive Flow Steering (RFS), to acquire the flow-level affinity. The basic mechanism of those technologies is to distribute the incoming packets among multiple receive queues for scalability, where each queue is handled by a dedicated CPU core. The subsequent packets of the same flow are directed to interrupt the same core in order to achieve data locality. For example, in [9], the authors implemented a source-aware affinity interrupt-scheduling system, SAI, to group onto the same core all interrupts for the same I/O request. The proposed software proved to be more effective than the OS default irqbalance service. Another study [10] described a scalable network socket function design, Affinity Accept. It utilized RSS and RFS to bind all activities related to a given connection onto a single core to minimize the contention for shared caches among multiple flows. However, the researchers [8] found that those solutions were unable to achieve high throughput over a 10 Gbps link, even using multiple parallel flows. The limitation results from inability to control the entire end-to-end path and an unawareness of the behavior of the application that consumes packets.

To scale the performance to 40 Gbps or more requires more informed usage of the CPU resource when processing data transfer tasks. Several recent efforts have demonstrated
the effectiveness of the application-level affinity control to I/O operations. For example, MICA [11], a fast scalable in-memory key-value store, divided the memory storage into several partitions managed by individual cores, and distributed the client’s requests to the relevant core based on user supplied information, such as the key, by using Intel DPDK I/O operations. The research in [12] presented a framework for executing tasks that maps coupled applications to the same processor node, thereby minimizing data exchanges between CPU nodes. However, only a few attempts have been made to scale the performance of an end-to-end path over the modern hardware. Among them, the authors in [13] proposed MultiEdge, an edge-based communication system that assured the affinity of all application processing on a single CPU to enhance data locality. However, their implementation only targeted at a moderate transfer rate over 1/10 Gbps network links.

This paper provides a novel design that takes advantage of the application-level affinity control to bind all data access and processing of the same I/O device into its local NUMA node. This design is also aware of the characteristics of I/O hardware and request partitions. It adopts storage-centric and capacity-aware task scheduling to maximize the overall throughput while ensuring low latency per task.

III. FRAMEWORK AND PROTOCOL DESIGN

A. Framework design overview

An inherent weakness of the traditional data transfer software design for multicore systems is that many of them use single-threaded sequential processing in their design, such as FTP and Linux scp. When the network performance needs to be be scaled up for large datasets, the data transfer applications involve the data-intensive operations that are stressing the CPU cores, memory, and I/O subsystems. To disperse the load stress and match the distributed nature of modern multicore systems, the sequential design must be substantially renovated with an asynchronous and multithreaded mechanism that can decouple and parallelize the entire data transfer processing pipeline.

To address this problem, we present a simplified version of our processing workflow in Figure 2. It divides an end-to-end data path into a series of stages, and spawns dedicated threads for each of them. Explicit task and event queues are allocated to connect these stages, and thread synchronization primitives are used to enable concurrent accesses. Thereby, the most salient feature of this design is that the threads from different stages can execute asynchronously. It naturally provides the desired features that match with the aforementioned design principles for high performance data transfer. To elaborate on this, we list the following supported features.

1) Asynchronous processing: The proposed workflow model is I/O resource-centric. All data transfer tasks and system I/O devices have their dedicated thread pools and data structures for managing all involved entities. This approach naturally separates task management, network I/O and storage I/O processing. Each stage in the model selects a combination of pipelined, concurrent, and event-driven operations to maximize the performance of its resources.

2) Resident service: The workflow is designed to be a daemon process in the system, and supports multi-users and the concurrent request processing. All the threads and memory are allocated and scheduled via a centralized control which facilitates the batch processing and the global optimization of related resources.

3) Storage-centric design: This design focuses on improving the performance of the system storages which usually are the bottleneck along the end-to-end data transfer path. We treat each storage unit adaptively via storage-aware preallocation of threads and storage-centric scheduling of tasks. The data transfer tasks are grouped according to their targeted storages, and served by statically-attached I/O threads. This configuration facilitates coordinated accesses to the storages, and also enables other potential global optimizations, for example, the file sorting that is described later.

4) Capacity-based scheduling: The bandwidth capacity of I/O devices is another key aspect in our design. The resource allocation module first assigns the threads according to the device’s bandwidth capacity. Subsequently, the capacity of a data transfer task is decided by the proposed protocol, as described later in Section III-D. To determine a task’s capacity actually constitutes the process of pinpointing the I/O bottleneck of the entire data transfer pipeline that involves both the data source and data sink. Lastly, the thread and memory resource assigned to the given task is computed, based on the capacities of the task and I/O devices involved. This capacity-based design balances the demands of individual tasks while improving the aggregate performance of the entire system.

Figure 3 depicts our multicore-aware data transfer design as a four-layered framework. Especially, when the system starts, the initialization layer executes once for preallocating resources. The remaining three layers comprise the pipeline for processing data transfer requests. This layered architecture allows us to present the various components
B. Initialization (INI) layer

The primary function of the initialization layer is to create the profile of the system resources, and allocate the corresponding management components (such as data structures and thread pools) for them.

1) User-assisted resource abstraction: The initialization layer profiles all system I/O resources, and creates an abstraction for each of them; it includes the major characteristics of an I/O device. For example, for each network interface, it contains the network’s logical name, address and physical bandwidth capacity; for a storage device, it includes the storage type (HDD, SSD, or memory), partition name, volume and also bandwidth capacity. We designed a user-assisted abstraction where users can guide the abstraction process via a static configuration file. For example, a network interface can be used for network connection, or for external storage, e.g. Lustre [14]. Accordingly, a user needs to inform the application of the exact purpose of an interface when automatic detection can not figure it out.

2) Capacity-aware and NUMA-aware thread preallocation: The second purpose of our initialization layer is to create the management components for the resources. All network I/O threads (senders and receivers) and storage I/O threads (readers and writers) are preallocated, and then grouped into thread pools, each of which is attached to the corresponding abstraction object of a I/O device. The number of I/O threads per pool is determined by the type of the associated device and its bandwidth capacity. The higher that the bandwidth is, the more threads are allocated for it. For storage devices, multiple I/O threads will be allocated to the storage device that has a good random access performance, e.g. SSD drives and memory-based storages. In contrast, only a single thread is assigned to the device that performs better on sequential access, e.g. HDD drives. Meanwhile, all threads belonging to the same I/O device are created or dynamically arranged to assure an affinity to the same NUMA node to which the I/O device is attached. Therefore, this capacity-aware and NUMA-aware preallocation module guarantees pertinent resource preparation while enforcing localized thread binding. Furthermore, by preallocating and recycling the I/O threads upon task completion, we avoid the overhead of dynamically and repetitively creating and terminating threads.

C. Request management (RM) layer

1) User request interface: The user service interface module is responsible for retrieving user requests. It also handles any completion or error event from the lower layers, and maintains all data transfer sessions. On the other hand, it is in charge of communicating with users to report the completion of data transfers, performance data, and errors.

2) Storage-centric task composition and scheduling: Upon receiving user requests from the request interface, this module decomposes them into one or multiple data transfer tasks. Figure 4 gives an example. Each storage device is associated with a task queue for all tasks of reading/writing data from/into it. A user request is passed from the upper module and decomposed by the task composition and scheduling module, and then is regrouped into tasks according to data location. Finally, the resultant tasks are dispatched to the corresponding task queues. Task regrouping here parallelizes task handling across different storages while localizing the storage I/O accesses.
D. Protocol and event processing (PEP) layer

To coordinate the resource allocation and data transfer between two entities, namely the data source and data sink, we designed a communication protocol for setting up connections and negotiating resources. An important objective here is to offer sufficient I/O resources to each task while avoiding any deadlock and conserving resources. To serve a data transfer task, we devised four types of I/O threads, readers and senders on the data source, and writers and receivers on the data sink. However, either the source or the sink is unaware of the availability of threads at the other end. Therefore, a resource negotiation protocol is required to coordinate resource allocations at both sides. Figure 5 illustrates the capacity-based resource negotiation protocol. Herein, $Cap$ is the bandwidth capacity of tasks or I/O devices. The source first determines the task capacity locally, and applies a certain number of I/O threads according to the ratio of the task’s capacity and the device’s physical capacity, having acquired both the readers and senders, the source initiates a control channel to the data sink, and notifies the latter that the source is ready. The results of task capacity and the number of senders are sent to the sink for negotiating its resource. The sink then computes the final task capacity, and gets I/O threads accordingly. Finally, the sink sends back the confirmed data connections and task capacity, and then the source relinquishes any excessive senders and readers with the references of these confirmed parameters. The size of data block, viz. the processing unit for disk access and network communication, is also negotiated at the same time. This mechanism guarantees each task has sufficient thread resources, while maximizing the number of concurrent tasks. After negotiating resources and assigning I/O threads, the data source initiates all data connections.

E. Data access and transmission (DAT) layer

The buffer management module uses a NUMA-aware bulk memory allocation. It first allocates a large trunk of memory, and then partitions it into small buffers. These buffers are then attached to a task that is ready for transfer. The number of buffers is determined by the number of assigned network threads or per user’s requests. Meanwhile, all buffers are pinned to the NUMA node with the targeted network interface by default since TCP/IP has a more intensive protocol processing load than that of a storage I/O. The buffers are also reused over the course of data transfers.

The two modules at the bottom layer of Figure 3 are encapsulated by network I/O threads (senders/receivers) and storage I/O threads (readers/writers). On the data source side, readers produce loaded buffers, and senders consume them, and return them to the list of free buffers. When the whole data transfer is finished, the I/O threads are all returned to their respective pools for reuse, and a completion event is posted to the upper layer. The I/O threads are designed to be self-suspended before they are initialized with assigned tasks or after their assigned task is completed.

IV. IMPLEMENTATION

We implemented the proposed design model, named the multicore-aware data transfer (MCDT) software. It comprises of two components, a frontend agent and a background daemon (Figure 6). The agent is responsible for receiving the command line requests from users, relaying them to the daemon, and then reporting the performance data or error message during and after data transfers. The entire software is implemented using a peer-to-peer design, hence it naturally can support the third-party data transfer.

The general process for handling a user request is as follows: Firstly, MCDT accepts a user’s command line request via the Control (CTL) entity which then initiates a ssh connection to both the data source and data sink hosts. Subsequently, the Data source (SRC) and Data sink (SNK) agents (entities) are launched, and they then post the requests to their local daemon processes. The latter retrieves the requests, and starts to communicate with the remote peer.

A. Daemon implementation

Figure 7 shows the multithreaded architecture and data structure of the background daemon process. It acts as the rendezvous point to collect the data transfer requests system wide, manages the I/O-related resources, and bootstraps the data transfer tasks intelligently. There are three categories of data structures and threads in the daemon process. We detail them in the following sections.
Transferring Complete/Cancelled/Error control message that comes through the link.

...and dispatches the link to the right task according to the first list...

...listens on a well-known port, accepts incoming connections, and the daemon process maintains an abstraction list of all network and storage devices. The system-wide...specifically, the daemon process maintains an abstraction list of all network and storage devices. The users’ session list contains all the user requests that currently are in processing. Among all running threads, the request poller retrieves request from a local request pipe, and inserts a session element to the user session list. Thereafter, it analyzes and partitions the request into several tasks, and then distribute them to different task queues that correspond to different storage devices. Meanwhile, the link dispatcher listens on a well-known port, accepts incoming connections, and dispatches the link to the right task according to the first control message that comes through the link.

1) Universal data structure and threads: This category of data structures and threads is unique and shared in the system. They correspond to the shared resources and entities system wide. Specifically, the daemon process maintains an abstraction list of all network and storage devices. The users’ session list contains all the user requests that currently are in processing. Among all running threads, the request poller retrieves request from a local request pipe, and inserts a session element to the user session list. Thereafter, it analyzes and partitions the request into several tasks, and then distribute them to different task queues that correspond to different storage devices. Meanwhile, the link dispatcher listens on a well-known port, accepts incoming connections, and dispatches the link to the right task according to the first control message that comes through the link.

2) Per-storage data structure and threads: This category of data structure and threads are defined for I/O devices. Given the storage-centric design, the task queues and their poller threads are created and statically attached to their storage respectively. The universal request poller thread adds the requests into the storage queue. Afterwards, the task poller thread of the storage activates a request. Thereafter, its task controller thread removes it from the storage queue. Furthermore, in our implementation, each of storage and network devices has both inbound and outbound data access, and thus, for a clean thread design, they have two separate thread pools for inbound and outbound access.

3) Per-task data structure and threads: This type of data structures and threads are assigned to newly arrived tasks. The task controller thread maintains the task’s status, manages its data structure and its associated threads during the life cycle of a task. We use a state transition diagram (Figure 8) to depict the life cycle of a data transfer task in a daemon process. At the beginning, the task is kept in the task queue and in the “waiting” state. Once it is retrieved by a task poller thread, it enters the “active” state, and the daemon creates a task controller thread for it. The controller thread then makes a reservation for thread resources from the corresponding I/O thread pool.

If the target pool is empty, the target task enters the “suspend” state, and the task controller becomes inactive. Once the resource pool is not empty upon other tasks’ completion, the task controller then wakes up to query the pool again, and the corresponding task returns to the “active” state. After this resource negotiation and assignment, the task obtains all the required resources, viz. the assigned I/O threads and the task then enters the “transferring” state. When the data transfer is finished, each I/O thread returns and posts a completion event to the task controller thread. The task then moves to the final “exit” state. After collecting all the completion events, the task controller thread returns all the I/O threads to the thread resource pools, wakes up any suspended task and updates the user session list.

**B. Optimizations**

Next in this paper, we detail a fast, lightweight, stream-lined MCDT implementation that is not only NUMA-aware, but also integrates the following techniques to expedite the task processing pipeline. We then evaluate their effectiveness of the optimizations in the next section.

1) Kernel-bypass storage I/O for SSD access: This direct I/O operation offers a kernel-bypass option for accessing storage. The storage I/O module in MCDT integrates this technique with large data buffers in the user memory space to access SSD and even other types of media.

2) File sorting for HDD reading: The HDD storages are known to perform much better by assuring sequential accesses. To achieve this, a single I/O thread is assigned to the HDD storage while pre-allocating resources. Then,
we optimize the task queue with file-level sorting according to the starting physical address of a file. This approach is particularly useful for transferring small file since all data within a small file are likely located at adjacent blocks.

3) Serializing data blocks for HDD writing: Sequential writing to HDD can also optimize I/O performance. After the network receiver thread fills a data buffer, it inserts the buffer to the list of the loaded buffers according to the offset of buffered data, instead of simply appending the buffer to the list in an FIFO order. Hence, all the buffers in the loaded list are sorted to maximize the write performance.

V. EXPERIMENTAL EVALUATION

In this section, we first describe the configuration of the testbed and the workload. Next, we we quantitatively evaluate and analyze several optimizations in MCDT one by one. Lastly, we comprehensively compare the performance of MCDT with the other widely used data transfer tools.

A. Testbed Specifications and Workload

The evaluation testbed includes two state-of-the-art multicore systems (Figure 9): Table I shows the specification of hardware. We configured the storage of the SuperMicro host with four RAID0 disks, and each of which consists of eleven HDDs. All disks in the server are partitioned into four groups, each of which is attached to the LSI RAID controller card on the motherboard, and appears as one single storage device in Linux system. Meanwhile, the IBM system in the testbed includes two LSI Nytro WarpDrive cards, each connected to a separate NUMA node via a PCI slot. Two hosts have a 40Gbps Ethernet adapter for their network connection. All these I/O devices are installed to the PCI Gen3 slots in the IBM system. Both servers run a CentOS system with the 2.6.32-431 Linux kernel.

Two types of workloads were tested: 1) To test the transfer of bulk data with our MCDT, we filled the SSD disks on the IBM host with a few big files (50-200 GBytes), and sent them to the HDDs on our SuperMicro host via the 40G link; and 2) for the transfer of small files, we created 4000 files in each RAID0 disk on this SuperMicro host, and sent them to the /dev/null of the IBM host. The size of the files in the small workload follows a log-normal distribution, as illustrated by the histogram in Figure 10. During transfer, we used Ganglia [15] to record the overall bandwidth and the CPU utilization. In addition, we monitored the execution time of each task via the task controller thread in the MCDT.

B. Evaluation of Optimizations

In this section, we quantitatively evaluated the different optimizations that we introduced in Section IV-B. We also compared the performance of our software in two scenarios: with and without the optimizations. This helped us to understand the effectiveness of each design on different types of storage device and user scenarios.

1) Effectiveness of sequential writing: The optimization of sequential writing is usually useful to accessing HDDs. Figure 11 shows the bandwidth performance while transferring a single 200 GBytes with different sizes of data blocks from the SSD on the IBM system, to a RAID0 disk on the SuperMicro host. Herein, the bottleneck of the whole data transfer pipeline resides in writing to storage. MCDT attained on average a 2.88% improvement in bandwidth when we adopted sequential writing as an optimization approach. Figure 11 shows that the bigger improvement when the block size is small. The reason is that for a fixed file size, a smaller block size leads to a larger number of blocks. Since MCDT thus has more cached blocks to work on, the advantage of sorting becomes more obvious under this circumstance. Generally here, sequential writing does not greatly affect performance since the test is done in a Local Area Network environment where there are few out-of-order transferred data blocks. We envision that such
an optimization can bring more benefit to the real Wide Area Network data transfers which share network with some competing transfers, and thus have a much more complicated network environment.

2) Effectiveness of file sorting: As discussed in Section IV-B, MCDT sorts all files in each queue according to the files’ starting address. Figure 12 compares the sorted and unsorted transfers of 4000 files from a single RAID0 disk. The effectiveness of file sorting is confirmed by an increment of 37% to 47% in bandwidth since file sorting significantly reduces randomness while accessing storage.

3) Effectiveness of NUMA-awareness: NUMA-awareness is another critical optimization in the MCDT system. Figure 13 presents the comparison results from two types of workloads: transfers of bulk data and of small files. On average, the NUMA optimization delivers a 14.9% increase in bandwidth and a 12.7% decrease in latency for bulk data transfer, and for small request processing, a 17.9% improvement in bandwidth and a 12.5% reduction in latency. Instead of depending on default OS scheduling, MCDT utilizes its own pre-allocation and buffer management modules to pin the I/O threads and their data to the NUMA node that is connected directly with the I/O device involved. These results confirm the effectiveness of the mechanism of thread-dependent scheduling in MCDT across different workloads.

C. Comparative Evaluation

This section compares the performance of MCDT with two popular data transfer software used in the high performance computing community, GridFTP 5.2.5 and BBCP 12.08.17.00.0. To ensure the best practices for GridFTP, we enabled its threaded option, disabled all authentication operations and utilized its extended block mode (MODE E) for all data transfers [16]. For BBCP, we also used its available options to avoid the overheads of checking DNS and the space at the data sink. In all tests, both GridFTP and BBCP uses 16 parallel TCP streams, viz., the best case we observed on our testbed. The execution time of a single task reflects a similar trend with the bandwidth performance, and thus is not covered here.

Bulk data transfer: we processed the bulk data workload by different numbers of parallel file transfers. Each file was sent to a separate RAID0 disk on SuperMicro server, resulting in a maximum of four parallel files. The transfer of a single file only involves one SSD. During parallel transfers of multiple files, multiple instances of GridFTP and BBCP are launched, one for each file, so to assure concurrency. In all test cases, we needed to create a single MCDT daemon process to transfer all the data. Through concrete experiments, we found that all of the three software systems performed well at 2M block size, e.g., as shown in Figure 13, and thereby we set a block size of 2M for all comparison studies. As shown in Figure 14, MCDT performs the best in all test instances, and scales well from the single-
file case to the two-file case. When there are more than two parallel files, the MCDT’s bandwidth exceeds 18 Gbps, which is very close to the bandwidth limit of writing into the LSI RAID controller in the SuperMicro host. Both MCDT and BBCP outperformed GridFTP with fewer CPU cycles, because they utilize direct I/O operations, and also employ separate threads for storage and network I/O operations. Besides, MCDT also achieves 28% more bandwidth than BBCP while it still has a similar CPU load, because it takes advantage of the multi-threaded storage I/O for accessing SSD, and of sequential writing optimization.

Small requests transfer: To evaluate the capability to handle a large number of small data requests, we transferred all 4000 files that reside in each of the four RAID0 disks on the SuperMicro host, viz, 16,000 files in total, to “/dev/null” of the IBM host. Again, for GridFTP and BBCP, we created four concurrent instances, one for each RAID0 disk. Figure 15 compares their overall bandwidth performance and CPU usage with different block sizes. Due to the single thread implementation of GridFTP, its aggregate CPU usage did not exceed 400%, and its bandwidth performance was lower than that of BBCP and MCDT. The CPU usage on the data source host was higher than that on the data sink host, because this test case involved no disk write at the data sink, only read data from the disks at the source host. MCDT retained consistently a higher bandwidth performance than did the other two, i.e. 107.7% higher bandwidth than GridFTP, and 63.2% more than BBCP on average, while in the meantime with a higher CPU usage. This proves that the MCDT software system, among the three transfer tools, has the best CPU scalability, and can better utilize the multicore and I/O resources. In particular, when the block is small (32 KB), MCDT demonstrated a significantly better capability of handling a large number of tasks and data blocks than did the other two.

Another interesting observation is that, 512 KB is the “sweet spot” and favored in all small request cases, as shown in Figure 12, Figure 13(b), and Figure 15(a). The reason is that when the block size is small, there are more data blocks to process, and each block incurs a constant amount of overhead, and thereby MCDT incurs more aggregated overhead. On the other hand, when the block size is too big (for example, greater than 2 MB), most small files cannot even fill up a single block. A large portion of the data block is wasted, which introduces unnecessary overheads as well. Furthermore, as depicted in Figure 10, the size of a large portion of small files is around 512 KB. For these reasons, a block size of 512 KB was our best practice.

To evaluate the individual optimization strategies employed in MCDT and quantify its gain in performance, we analyzed all the previous results, and undertook more experimental comparisons, e.g., MCDT with direct I/Os versus GridFTP. Table II presents the final results. We list our observations and analytic results as follows: 1) For small file transfers, asynchronous processing was shown to have significant performance advantages over GridFTP.
It was less effective in bulk data transfer tests since we chose to transfer a single file per storage device, so to avoid the performance bottleneck of writing to HDD. Small file tests involved processing more requests and incurred more events wherein asynchronous processing was more effective than synchronous processing. There was no benefit in sorting files as only one file per task is involved in bulk data transfer. 2) In contrast to GridFTP and BBCP, MCDT utilized multiple parallel reader threads for accessing data in SSD, and contributed another 17.21% more bandwidth to bulk data transfer. 3) Direct I/O delivered the major performance gain in the bulk data transfers. It is especially effective for loading data from SSD by a large block size. 4) The small request tests did not involve disk writing, direct I/O and multithreaded storage access. NUMA-awareness, asynchronous processing, and more importantly file sorting were the major contributors.

VI. Conclusion and Future Work

This paper describes the design of our high-speed data transfer software using a novel asynchronous processing mechanism. This design integrates various components of the multi-staged end-to-end data transfers, such as storage I/O, network communication, and request handling. Asynchronous processing maximizes their concurrency, and thereby provides improved scalability and utilization of multiple types of resources in a modern multi-core system, including processors and I/O devices. Furthermore, the proposed framework opens the opportunity to integrate and implement effective optimizations. The evaluation results provided in this paper quantitatively justified the effectiveness of each component in MCDT, and demonstrated that the combination of the asynchronous processing design and the integrated optimizations ensures superior performance in different scenarios of data transfers.

Our future work will extend the current framework to enable the flow-level and core-level affinity configurations. We also will consider integrating asynchronous disk I/O (Linux libaio) in accessing storage systems, and will introduce a load balancing mechanism between task queues.

ACKNOWLEDGMENT

The authors are grateful to the facility donation from Mellanox Technologies, Inc. and LSI Corporation.