Moving Large Data Sets Over High-Performance Long Distance Networks

Stephen W. Hodson, Stephen W. Poole, Thomas M. Ruwart, Bradley W. Settlemyer
Oak Ridge National Laboratory
One Bethel Valley Road
P.O. Box 2008 Oak Ridge, 37831-6164
{hodsonsw,spoole}@ornl.gov,tmruwart@ioperformance.com,settlemyerbw@ornl.gov

***Abstract*—In this project we look at the performance characteristics of three tools used to move large data sets over dedicated long distance networking infrastructure. Although performance studies of wide area networks have been a frequent topic of interest, performance analyses have tended to focus on network latency characteristics and peak throughput using network traffic generators. In this study we instead perform an end-to-end long distance networking analysis that includes reading large data sets from a source file system and committing large data sets to a destination file system. An evaluation of end-to-end data movement is also an evaluation of the system configurations employed and the tools used to move the data. For this paper, we have built several storage platforms and connected them with a high performance long distance network configuration. We use these systems to analyze the capabilities of three data movement tools: BBcp, GridFTP, and XDD. Our studies demonstrate that existing data movement tools do not provide efficient performance levels or exercise the storage devices in their highest performance modes. We describe the device information required to achieve high levels of I/O performance and discuss how this data is applicable in use cases beyond data movement performance.**

I. Introduction

The time spent manipulating large data sets is often one of the limiting factors in modern scientific research. In fields as diverse climate research, genomics, and petroleum exploration, massive data sets are the rule rather than the exception. With multi-Terabyte and Petabyte data sets becoming common, previously simple tasks, such as transferring data between institutions becomes a hugely challenging problem. The push to Exascale computing promises to increase the difficulty level in at least two ways. First, with a projected main memory size of 32-128 Petabytes for Exascale computers, typical future data sets will be significantly larger than the total amount of storage in existing file systems. Further, because of the staggering operational costs of Exascale machines, we expect that only a small number of such machines will exist, increasing the number of users interested in remote data analysis.

Quite simply, existing networking and storage infrastructure is not up to the task of transferring multi-Petabyte data sets on a frequent basis. The notion that researchers will use a web browser and wide area network (WAN) to move the bulk of a 100 Petabyte data set seems unlikely. Additionally, glib arguments involving a vehicle filled with magnetic tapes does not provide the type of performance we desire. Researcher's will need a responsive data transfer tool with high degrees of both resilience and performance to analyze the massive data sets generated by Exascale computer systems.

In this project we examine the performance of three data movement tools: BBcp, GridFTP, and XDD. Both BBcp and GridFTP are popular data movement tools in the HPC community. However, both tools are intended for a wide audience and are not optimized to provide maximum performance. In constructing our test bed, we have focused on technologies and performance constraints that are applicable to Exascale performance levels. In particular, we have focused on analyzing the impacts of distance (i.e. network latency) in transferring large data sets and the importance of achieving device-level performance from each component participating in the data transfer.

In the remainder of this section we describe related work in measuring and improving the performance of long distance data transfers. In section 2 we describe the conceptual model used to improve the performance of long distance data transfers, what we call the impedance matching problem. In section 3 we describe the configuration of our test bed and how it relates to existing Exascale computing technology projections. In section 4 we compare the performance of the three data movement tools, and examine the individual impacts of our various storage optimizations. In section 5 we discuss our results and describe future modifications we plan for our data movement tool.

A. Related Work

The problem of efficiently moving data sets over long distance networks has been studied by several research teams. Traditional file transfer protocol (FTP) clients and secure shell (SSH) transfer tools were never designed to achieve high levels of performance for large scientific data sets. The most popular tool for moving large data sets is likely GridFTP [1], a component of the Globus toolkit [2]. The Globus GridFTP client, *globus-url-copy*, provides support for multiple high performance networking options including UDT and Infiniband. It also supports parallel file system aware copying via the use of transfer striping parameters. Our tool, XDD, provides less security than GridFTP, and focuses on disk-aware I/O techniques to provide high levels of transfer performance.

Another popular high performance transfer tool is BBcp [4]. BBcp is designed to securely copy data between remote sites

and provides options for restarting failed transfers, using direct I/O to bypass kernel buffering, and an ordered mode for ensuring data is both read and written in strict serial order. Our tool, XDD, attempts to copy many of the features available in BBcp, while also providing a disk-aware implementation.

Efficient use of both dedicated and shared 10 Gigabit Ethernet network links have been studied by several research teams. Marian, et al., examined the congestion algorithm performance of TCP flows over high latency dedicated 10 Gigabit Ethernet network connections and found that modern congestion control algorithms such as HTCP and CUBIC provide high levels of performance even with multiple competing flows [8]. Wu, et al., and Kumazoe, et al., studies the impacts of congestion control on shared 10 Gigabit Ethernet links, and found that congestion control algorithms strongly impacted performance at long distances [6], [13]. XDD provides a configurable number of threads and supports modified TCP window sizes to provide performance in shared network scenarios; however, XDD is primarily designed as an high-end computing (i.e. Exascale) data movement tool where circuit switched dedicated network links are more likely to be available.

An alternative approach to transferring large data sets between two file systems on either end of a long haul network is to rely on wide area network (WAN) file systems. OceanStore is a prototype WAN file system designed to provide continuous access to data from all over the globe [5]. The Lustre parallel file system has also been extended to support wide area networks [12], and by using an optimized Lustre-aware file copy tool, such as spdcpy [10], it may be possible to relocate data within a geographically dispersed Lustre file system with high levels of performance. The original YottaYotta product line provided a service similar to long distance file transfers by replicating file system disk blocks over a wide area network connection [3].

II. High Performance Data Transfers

A file transfer can be decomposed into five primary components: the source storage devices, the source host, the network, the destination storage host, and the destination storage devices. Fundamentally, a high performance file transfer is a matter of tuning each component to provide high levels of individual performance and matching the performance of each connected component. This problem formulation can be though of as special type of an impedance matching problem where each component must provide data throughput compatible with the connected components. Many of the difficulties of a high performance file transfer are ensuring that each component is both executing in its fastest performance mode and not interfering the performance of the adjacent components.

The difficulty in achieving end-to-end high bandwidth for large file transfers is ensuring that each component is moving data at the required rate and not interfering with the performance of the adjacent components. Consider two possible configurations for a long distance file transfer. In the first scenario, we see a simple system composed of two hosts, each with a single hard drive and a network interface card plugged into a shared switch. Existing transfer tools such as SCP and FTP were designed with this type of scenario in mind. The second scenario shows a more realistic configuration for use in a HPC systems center. The hosts are composed of multiple processors with large amounts of main memory, the hosts have several direct connections to both high speed storage arrays and high performance networking equipment. In such scenarios, large scale data transmissions are the rule rather than the exception.

A. The Storage and Network System Hierarchy

The Storage and network subsystem hierarchy shown in figure 1, describes the levels of hardware and software that an I/O request must traverse in order to initiate and manage the movement of data between the application memory space and a storage device or network. The I/O request is initiated by the application when data movement is required. For a storage operation such as a read or write, the request is processed by several layers of system software such as the file system manager, logical device drivers, and the low-level hardware device drivers. During this processing the application I/O request may be split into several "physical" I/O requests that are subsequently sent out to the appropriate storage devices to satisfy these requests. These physical I/O requests must pass through the Fibre Channel Switch that makes the physical connection between the Host Bus Adapter on the computer system and the storage device, in this case a disk array controller. The disk array controller will take the I/O requests and generate one or more I/O requests for the individual disk drives in the array. Each disk drive processes its request and data is eventually transferred between the disk drive and the application memory space with multiple opportunities for buffering the data along the way.

Network send and receive operations are processed by an entirely different set of software and hardware layers. These include the socket layer, TCP layer, IP layer, device drivers, and switches. During this processing the application I/O request is split into "segments" that get buffered and eventually presented to the Network Interface Card (NIC) device driver. The NIC then transfers these segments on a FIFO basis to their respective destination addresses. There are multiple levels at which the acknowledgement of the receipt of data (ACKs) occurs and it is beyond the scope of this discussion to describe ACKs other than to mention that they are handled transparently to the data transmission/receipt as far as the application program is concerned.

The following sections present a more detailed description of each level in the hierarchy with respect to its function and performance implications.

1) Computer System: The Computer System is a critical piece of the Storage Subsystem Hierarchy in that it encapsulates all the software components and the necessary interface hardware to communicate with the physical connections (i.e. the Host Bus Adapters or NICs. The components within the Computer System include the processors, memory, and

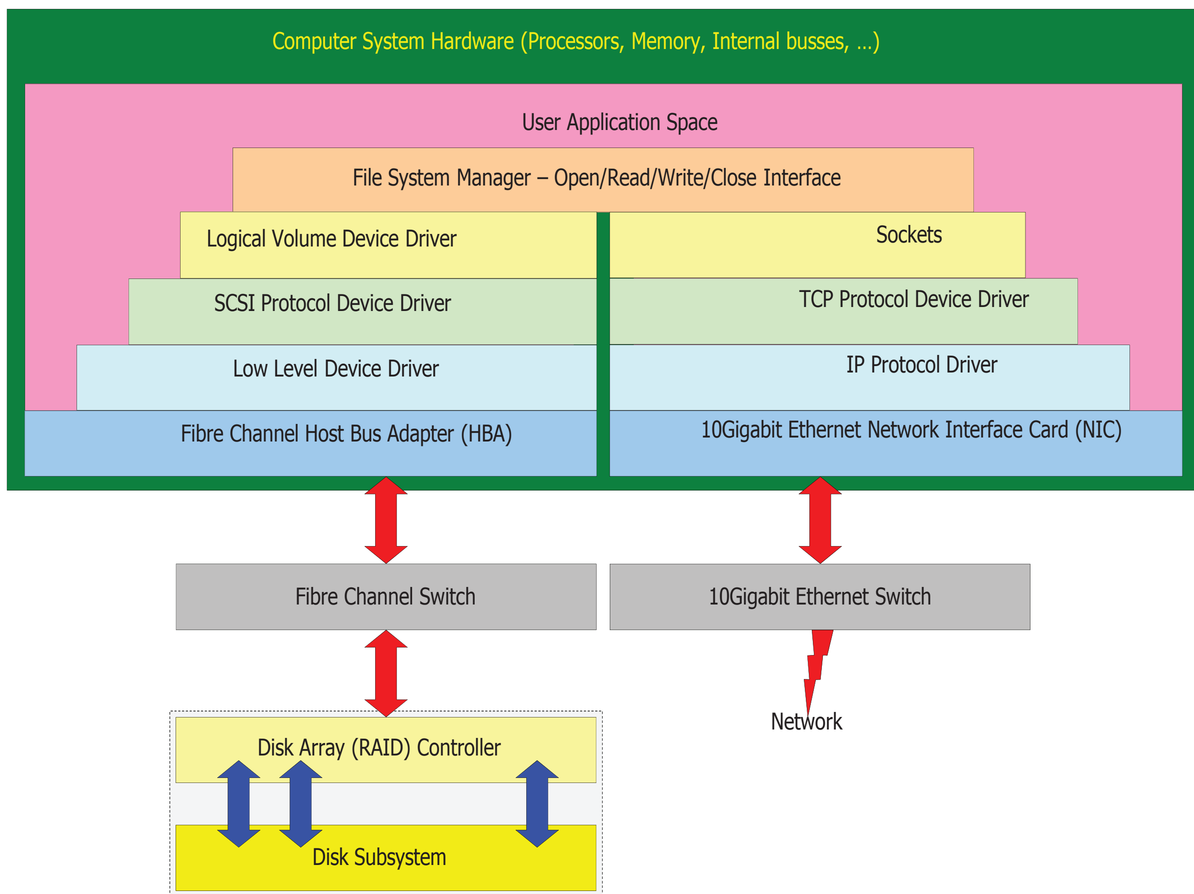


Fig. 1. The Storage and Network Subsystem Hierarchy. Every storage or network I/O operation initiated by an application must traverse the hierarchy. Performance is determined by how well the I/O behaves from one layer to the next in the respective hierarchy.

internal busses that connect the memory to the processors and to the Host Bus Adapters or NICs. The performance characteristics of each of these major components (i.e. the clock-speed, number of processors, processor architecture, memory bus bandwidth, etc.) plays a significant role in the overall performance of the Storage Subsystem as will be demonstrated in a later section. However, given the fastest hardware available, the Storage Subsystem will only perform as well as the underlying software, starting with the Application Program.

2) Application Program: The term Application Program as it is used here is any program running on the Host Computer System that requires data movement between the memory in the host computer and a Storage Unit. Application programs can be either typical User programs or can be parts of the Operating System on the host computer such as the paging subsystem. In any case, these programs have the ability to make I/O requests to any of the lower layers in the hierarchy if the Operating System provides an appropriate programming interface to do so. For example, the benchmark program used to gather statistical data for this project can access a storage unit through the file system manager, the logical volume device driver(s), or through the storage device drivers. In general, Applications that can manage and access the lower levels of the hierarchy achieve better performance than Applications that must traverse through the higher level layers such as the File System Manager.

3) File System Manager: The File System Manager provides a level of abstraction to the Application Program in order to simplify the process of accessing data on the storage subsystem as a whole. Because of the amount of indirect I/O processing that can accompany a single Application I/O request (such as space allocation, inode lookups, etc.), the I/O performance through the File System Manager can become unnecessarily slow when moving very large amounts of data. For example, a typical application opens a file and begins reading data from that file into the application's memory buffer. When the application calls the read() function for X number of bytes, the File System Manager will check to see if that data is already in a system buffer. If the data is already in memory, the File System Manager will copy X bytes from the buffer into the application's memory buffer. However, if the data is not in memory, the File System Manager will issue a read request to the underlying storage unit (which could be a Logical Volume or a physical storage device), for N*Y bytes where Y is the size of a system buffer and N is the

number of buffers required such that N*Y is greater than or equal to X. Once the required bytes have been transfered to memory from the underlying storage unit, the File System Manager will copy the requested data into the application memory buffer thereby completing the I/O request. When reading relatively small amounts of data (on the order of megabytes), the overhead involved with copying data from the system buffers to the application buffer is minimal. But when the amount of data gets very large (gigabytes on up) then the memory-copy overhead becomes a significant limiting factor in the overall bandwidth at which data can be read into the application memory buffer. Other examples of File System Manager overhead include the explicit and implicit overhead involved with allocating space on the storage unit and the size of individual transfers, both of which become critical for sustained, high-bandwidth movement of large amounts of data. Explicit space allocation is simply the amount of time taken to allocate space for an I/O operation. If an application is writing a 1 Gigabyte file 4096-bytes per write request then the file system manager might have to perform a block allocation for each of the 250,000 write requests from that applicaion. On the other hand, if the application requested that the File System Manager pre-allocate 1 Gigabyte of space for subsequent write requests to that file the the file system manager only need perform the space allocation once, at the beginning of the overall operation. Implicit space allocation overhead results from having the File System Manager performing space allocations on a per-request basis and having those allocations made at non-contiguous locations on the storage unit. The effect of this may not be discovered until the file is read back and the read bandwidth is low because the underlying storage devices are not able to stream due to the non-contiguous location of the blocks being read in. Some of the allocation problems can be addressed by having the application use very large transfer sizes, on the order of tens to hundreds of Megabytes per transfer. For all the reasons mentioned above, it is necessary to minimize the involvement of the File System Manager whenever possible. This can be accomplished by using Direct I/O, file pre-allocation, and large transfer sizes.

4) Logical Volume Device Drivers: The Logical Volume Device Drivers provide a mechanism to easily group storage devices into a single logical device in order to increase storage capacity, performance, and/or to simplify the manageability of large numbers of storage devices. The Logical Device Driver presents a single device object to the File System Manager or Application. The Logical Device Driver is then responsible for taking a single I/O request from the Application or the File System Manager and mapping this request onto the lower level storage devices, which may be either actual storage devices or other logical volumes. There are many ways to configure a logical volume that consists of multiple underlying storage devices. One common configuration is to stripe across (also known as a wide-stripe) all the storage devices in an effort to increase available bandwidth or throughput (operations per second). In a wide-striped logical volume, data is laid out on the underlying storage device in "stripe units". A stripe unit is the amount of sequential data that is transferred to/from a single storage device within the logical volume before moving to the next storage device in the volume. In theory the stripe unit can be any number of bytes from a single 512-byte sector to several megabytes. In practice however, the stripe unit has a minimum size of a memory page (4096 bytes normally) and is usually an integer power of 2 bytes in size (i.e. 4096, 8192, 16384, etc.). Finally, it is important to note that the stripe unit size a constant within a logical volume and cannot be changed without restructuing all the data on the logical volume.

5) The I/O Protocol Driver: The I/O Protocol Driver is responsible for translating the I/O request from the upper level device/protocol drivers into a form that fits the I/O protocol used to communicate the request to the underlying storage or network devices. In general, an internal I/O request consisting of a command (read/write or send/recv), a data buffer address, and a data transfer length. This request is converted into one or more SCSI commands for storage or a network send/receive operations for network I/O which are then passed to the proper Host Bus Adapter or Network Interface Card as appropriate.

6) Low-Level Device Driver: This device driver takes the high-level information (i.e. the SCSI command) from the I/O Protocol Driver and interfaces directly with the host-bus adapter that will perform the actual data transfer between the storage or network device and memory. For example, given a PCIe-to-Fibre Channel Host Bus Adapter, this device driver will set up the host bus adapter with the address of the SCSI command buffer, the application data buffer, and the target device and then tell the host bus adapter to begin the operation. The host bus adapter will transfer the SCSI command buffer to the intended target device. At some later time the target device will request a data transfer operation that will be managed in part by the host bus adapter. At the end of the entire operation, an interrupt is generated to notify the Low-Level Device Driver of the completion status the I/O operation. Under normal circumstances, the Low-Level Device Driver then propagates the completion status to the upper-level drivers, eventually reaching the User Application Program.

7) Physical Connection Layer: This layer defines the hardware that physically attaches the host-bus adapter to the storage device or one computer to another via a network. These connections can be as simple as a single 3-foot cable or as elaborate as a multi-stage communication fabric spanning many miles. The most common storage interconnect is Fibre Channel or Ethernet in one flavor or another. Either one of these can be used to connect a storage device directly to a host computer system or to form a network that allows multiple host computer systems to access multiple storage subsystems. These multi-host, multi-device configurations are commonly referred to as Storage Area Networks or SANs. A Storage Area Network is the most flexible in terms of multiple access paths to a single storage device, multi-host shared access, fault tolerance, and performance. However, this flexibility also means increased complexity in managing all the nodes connected to the SAN, whether they are host computers or disk devices.

8) Storage Device and Storage Units: The distinction between a Storage Device and Storage Unit is that a Storage Device is made up of one or more Storage Units but can appear to be a single device. The example is that of a Disk Array which is a Storage Device that contains several individual Storage Units (disk drives) but can appear to the system as a single, very large, disk drive. In the case of a disk array, the I/O request is received from the host bus adapter and is divided up into one or more I/O requests to the underlying disk drives. Storage Units are individually addressable storage devices that cannot be further subdivided into smaller physical units. The principle example of this is a Disk Drive.

9) Network I/O: Network I/O is fundamentally different than Storage I/O because inbound Network data is unsolicited. This means that it can arrive at the network device without any prior request from an application program. Therefore the Network subsystem as a whole has to be able to accept incoming network data and store it until such time as an application requests that data from the Network subsystem. Furthermore, the Network subsystem is a shared resource among all the applications that perform Network I/O. As a result the Network subsystem currently relies on placing incoming data into system buffers that are later copied into an application buffer upon request. For large amounts of data being received by an application the memory copy from the network buffer to the application buffer can become significant thereby limiting the achievable bandwidth to an artificially low rate. Similarly, when sending data, the data in the application buffer is copied into one or more network buffers which are eventually placed onto the network wire by the Network Interface Card. Again, the data copy operations can be a bandwidth-limiting factor for applications moving large data sets.

B. Impedance Matching Problem

Each layer of software and/or hardware between the Application and the Storage Device adds overhead and other anomalies that can result in highly irregular performance as viewed by the Application. Overhead is essentially the amount of time it takes for the I/O request to traverse the specific layer. The source of overhead in each layer is specific to a layer and is not necessarily constant within a layer. An example of this is the overhead induced by the Physical Connection layer. A physical connection consisting of a short cable introduces virtually no overhead since the propagation of a signal at the speed of light over a 3-foot distance is not significant. On the other hand, propagation of a packet of data traversing a 10,000-mile network through multiple switching units will introduce noticeable overhead.

An interesting artifact resulting from the interaction of the components in the Storage and Network Subsystem Hierarchy is analogous to the Impedance Matching problem in electrical signal propogation on a wire. The term Impedance Matching is used as an analogy to what happens when there is a mismatch of operational characteristics between two interacting objects. In an electrical circuit, an impedance mismatch has an effect on the performance of the circuit in terms of its gain or amplitude at particular frequencies. In the Storage Subsystem Hierarchy, an impedance mismatch has more to do with things like I/O request size and alignment mismatches that have an effect on the performance (bandwidth or transaction rate) of the storage subsystem as viewed by the application. The effects of these mismatches can be viewed from several different perspectives including the Application perspective, the Device perspective (network or storage), and the System perspective. The effects of this phenomenon are presented in the sections that follow.

1) The Impedance Matching Problem as it Relates to Large-Scale Data Transfers: Simply increasing the number of hosts participating in a file transfer is not a tractable path towards exa-scale dataset movement. For example, to move one Petabyte of data per day requires a sustained, average, end-to-end bandwidth of 12 GBytes per second for 86,400 seconds (one day). In order to achieve this requires a storage subsystem at the source and destination sites, capable of sustaining 12 GBytes/sec reading (source-side) or writing (destination-side) data for at least one day. This implies, of course, that each storage system should be able to accommodate one Petabyte of data. This also implies that the system as a whole can compensate for occasional sluggishness in the network or storage devices by moving data faster for brief periods of time in order to maintain the *average* rate of 12 GBytes/second. Fundamentally, to sustain the required bandwidth requires storage and network devices that can run at a bandwidth 20% to 50% (or more) higher than the sustained bandwidth. The 20%-50% comfort margin is necessary to accommodate inevitable variations in the bandwidth of the individual components.

In order to properly match the impedance of a large-scale data transfer from one end of the transfer to the other requires attention to the number of application threads used to perform storage or network I/O, the size of the transfers, and eliminating as many memory-to-memory copy operations as possible.

The number of threads used to perform storage I/O depends largely on the characteristics of the storage device. For a single disk drive, two or three threads is more than sufficient to keep the disk drive busy. For a logical volume composed of several disk arrays each of which has many disk drives, it is entirely possible that many more threads are required to effectively keep all the underlying disk drives busy and streaming. However, if there are too many threads, then coordinating the order in which I/O operations are issued to the storage device becomes important. Otherwise, it is possible to overwhelm and confuse the underlying storage controllers and devices to the extent that that drop out of streaming mode resulting in a significant drop in delivered bandwidth (see figures 2 and 3).

It will be shown that for storage devices, larger transfer sizes result in higher sustained bandwidth and less of a dependency on the number of threads. The size of a transfer request should be at least the size of an entire stripe of the logical volume or some integer multiple of that size. Keeping the I/O requests

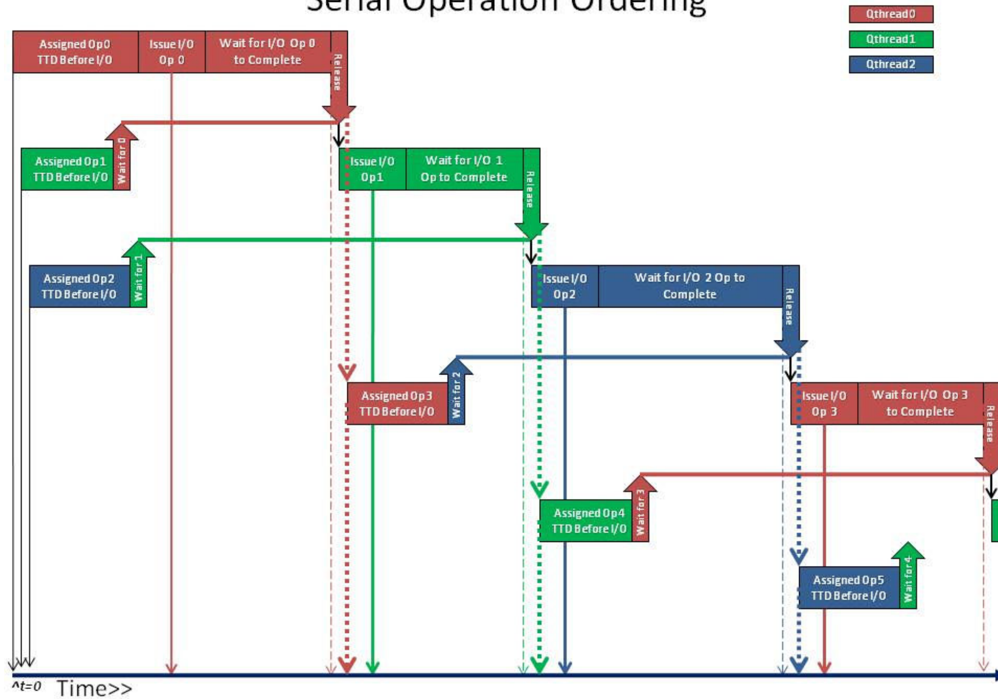


Fig. 2. Serial Operation Ordering forces each I/O operation to wait for the previous operation to complete before it can start.

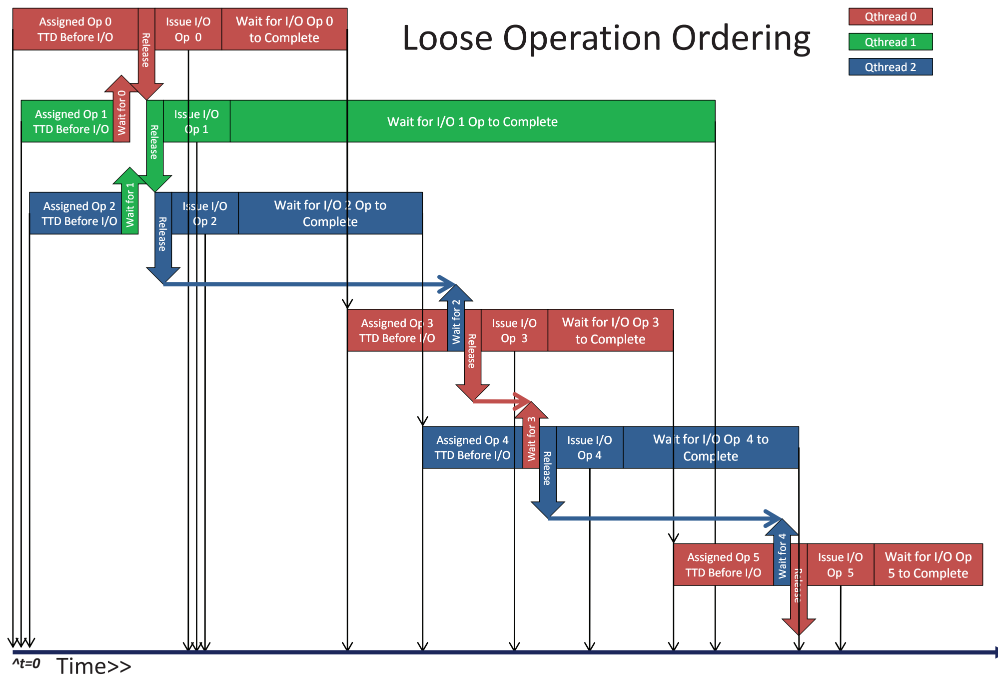


Fig. 3. Loose Operation Ordering causes each I/O operation to wait for the previous operation to release it from a semaphore before it can schedule its own operation. Hence, I/O operation N will release operation N+1 just before operation N gets scheduled. This will allow operaiton N+1 to get scheduled before N has completed thereby hiding the inter-operation latency.

aligned on stripe-boundaries is also important but is more a function of the underlying file system and its allocation policies.

Eliminating memory-to-memory copies is relatively straight forward with file system I/O. It requires the use of Direct I/O which is supported by most all current-generation file systems. The use of Direct I/O imposes some restrictions on the I/O requests but they are easily met and it is not necessary to cover all those details in this paper.

That said, it is important to note that Direct I/O is not readily available for Network I/O. The concept of Direct I/O for networks is also referred to as Zero-Copy. The problem is that there has been little effort into putting Zero-Copy into the standard releases of the Linux Operating System which is the OS-of-Choice in the HPC arena. As a result of this it was discovered that in order to maximize the bandwidth of transferring data from an application buffer on one computer system to an application buffer on another computer system over a network it is necessary to use more threads rather than larger transfer sizes. As our results show, the size of the transfer, beyond a minimum size of about 8 MBytes, has no affect on the end-to-end bandwidth of transferring a large amount of data. On the other hand, the number of threads used to transfer the data from one end to the other has a linear-scaling affect on the bandwidth up to about 96% of the wire speed of the network.

In the end, it was discovered that maximizing the storage-to-memory bandwidth on the storage subsystem required coordinated, well-formed, Direct I/O from a relatively small number of threads. Maximizing the memory-to-memory bandwidth between two applications over a network required reasonably-sized transfer sizes but a relatively large number of threads operating in simultaneous pairs. Putting this all together requires precise management of threads performing storage I/O and Network I/O on both ends of an end-to-end data copy of a large amount of data over a relatively long period of time. The precise management of these threads and their I/O behavior is, in fact, the impedance matching function as it were.

C. The Data Movement Problem

Data analysis in the high performance computing world is currently limited in scale by an I/O bottleneck that forces processors to idle while waiting for data from disk. In this storage challenge entry we perform a detailed examination of one aspect of the I/O bottleneck: long distance data movement. The time spent manipulating large data sets is often one of the limiting factors in modern scientific research. In fields as diverse high-energy physics, petroleum exploration, and genomics, massive data sets are the rule rather than the exception. Modern middleware and parallel file systems go to great lengths to accelerate complicated data access patterns; however, when datasets become truly large, even simple tasks, such as moving data between two remote sites can become a challenging endeavor.

XDDCP uses the End-to-End options to accomplish the task of copying data from one computer to another over a network. The End-to-End operation is accomplished by a *matched pair* of XDD instances with one instance running on the source computer and the other instance running on the destination computer. The source computer is defined to be the system that reads in the original file and transmits its contents over a network to the destination computer which is defined to be the system that writes the copy of the file to stable storage. Data movement is always from the source to the destination. It is important to note that the source and destination instances of XDD must be "matched" in the sense that they each have the same queue depth and that they agree on which network addresses and ports to use. For E2E operations the queue depth can either be specified explicitly using the "-queuedepth" option or it can be specified implicitly using the option: -e2e destination hostname:base_port,number_of_ports When the queue depth is specified implicitly, it will take precedence over the "-queuedepth" option if it is also specified. Furthermore, the queue depth implied on the "-e2e destination" option is the sum of all "number_of_ports" specified for a given XDD run. XDD can use either buffered I/O (figure 6) or Direct I/O (figure 7) for reading the source file or writing the destination file. Buffered I/O will increase the CPU overhead and possibly result in lower overall bandwidth performance of an E2E operation. Therefore, it is recommended that Direct I/O be used whenever possible. The following diagrams illustrate the basic difference between buffered I/O and Direct I/O as they relate to an E2E operation.

III. Methodology

As part of our work, it is necessary to transfer multi-Terabyte data sets over long distances. In order to measure the effectiveness of our transfers we built a testbed capable of shipping data thousands of miles along a networking loop. Although the data began and ended at our laboratory, we used independent components for each portion of the loop so that the source and destination were completely separate.

A. Storage Infrastructure

Out storage testbed consisted of six Infortrend EonStor S16F-R1430 disk arrays. The arrays were equipped with two controllers and 16 Hitachi DeskStar E7K500 disks. Each controller contained a single PC3200 DIMM, four 4Gbps Fibre Channel ports, and supported up to 1024 queued commands. The disks were 7,200 RPM serial ATA (SATA) disk drives with 500GB of capacity and an 8MiB cache. Although the controllers were capable of redundant operation, we disabled the redundancy features and used each controller independently. The controllers were configured to provide RAID level 5 in a 7+1 configuration with 64KiB stripe size. Each controller was connected into a storage area network (SAN) with a single 4Gbps Fibre Channel connection.

The storage network fabric was a Brocade Silkworm 4100 switch. The switch supported up to 32 4Gbps Fibre Channel connections with 256Gbps of end-to-end aggregate bandwidth. For our configuration it was necessary to split the 6 shelves (12 controllers) into 4 file systems. We constructed 12 zones such

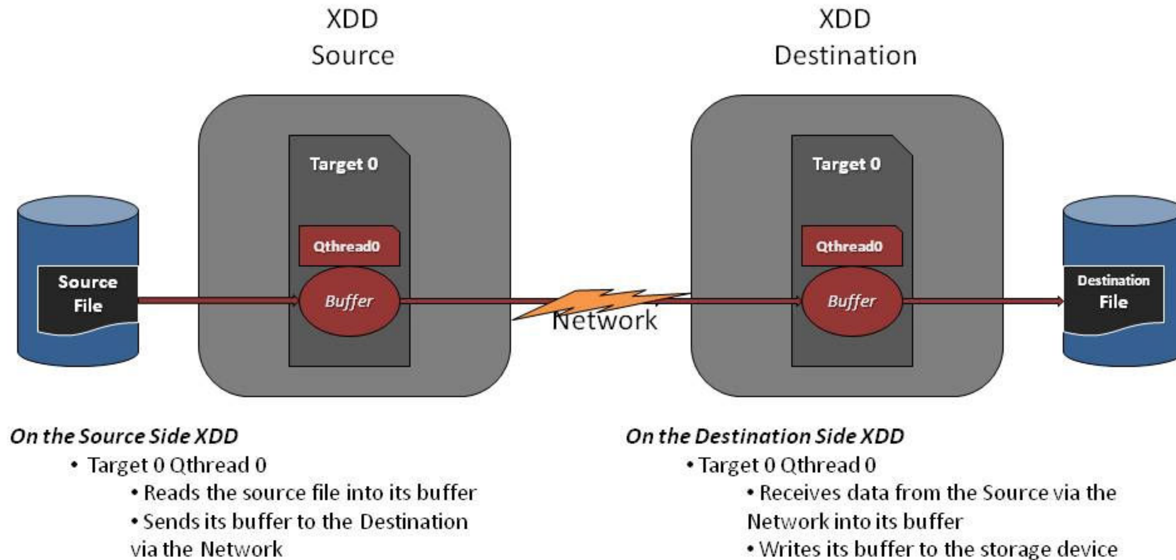


Fig. 4. A basic XDD End-to-End data copy operation from the Source file to the Destination file using a single thread on each system.

that the "A" controllers from the first three shelves connected to the host block adapters (HBAs) for the first source-side host, the "B" controllers for the first 3 shelves connected too the first destination-side host, the "A" controllers for the last 3 shelves connected to the second source-side host, and the "B" controllers connected to the second destination-side host.

B. Host Infrastructure

Our host configurations used standard commodity parts and the Linux operating system. All four systems were identical with the exception that the machines named pod7, pod9, and pod10 had dual Quad-Core AMD Opteron 8382 processors, whereas our fourth system, pod11, had dual Quad-Core AMD Opteron 2358 SE processors. The former processors ran at 2.6GHz while the latter processor ran at 2.4GHz. All four systems had 32GB of main memory, a built-in Gigabit Ethernet network interface, a Myricom Myri-10G Dual-Protocol network interface card (NIC), and two dual port QLogic ISP2432-based 4Gb Fibre Channel host bus adapters.

The Gigabit Ethernet port was connected to the National Center for Computational Science (NCCS) management network. The Myricomm NIC was connected to a Fujitsu XG2000 10 Gigabit Ethernet switch. The XG2000 is a 20-port enterprise-class 10Gig Ethernet switch. The XG2000 was configured into Virtual LAN (VLAN) environments such that pod7 and pod10 were in a single VLAN and pod9 and pod11 were in a separate VLAN. The isolated VLANs were connected to the long haul network described later. Finally, each host had three Fibre Channel connections into the Fibre Channel switch as described in Section III-A.

The host systems ran Fedora 13 version of the Linux operating system using kernel version 2.6.33.3-85. To aggregate the three storage units into a single file system per host we constructed a single volume group striped across each physical volume with a 64KiB stripe size. We then constructed a local XFS file system on each host using the default file system parameters with the exception that we used 4KiB block sizes. The XFS file system introspects into Linux logical volume manager (LVM) so that file system requests are aligned with the LVM stripe size to as great a degree as possible.

C. Network Infrastructure

The two VLANs described in Section III-B were interconnected by the Department of Energy's UltraScience Net (USN), an ultra-scale network testbed for large-scale science [9]. The USN facility included a dedicated OC192 connection that connected Oak Ridge National Laboratory (ORNL) to Fermilab in Chicago, Pacific Northwest National

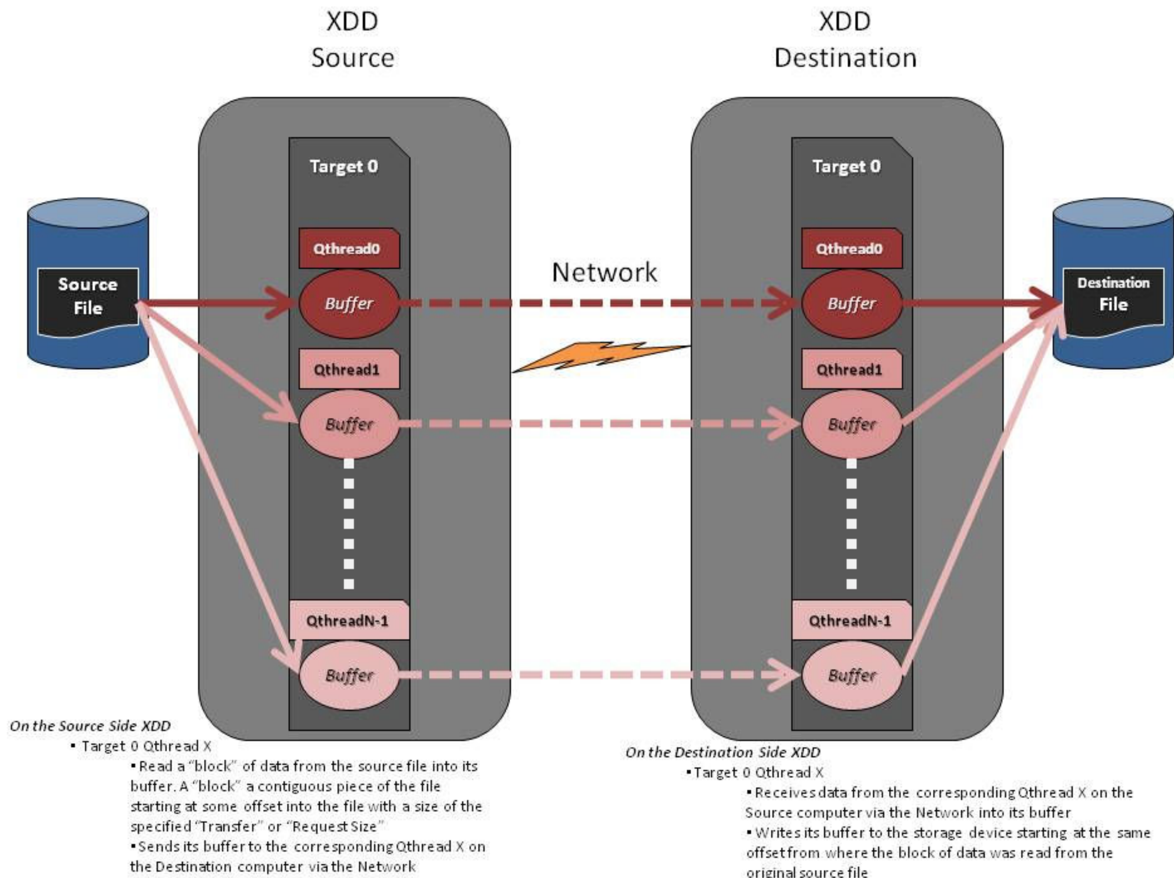


Fig. 5. An XDD End-to-End data copy operation using multiple threads to move data in parallel from the Source file to the Destination file.

Network Loop Details	Distance (mi)	RTT (ms)
ORNL-ORNL	0.2	0.28
ORNL-Chicago-ORNL	1400	26.8
ORNL-Chi.-Sea.-Chi.-ORNL	6600	128
ORNL-Chi.-Sea.-Sunnyvale	8600	163

TABLE I
ULTRASCIENCE NET WIDE AREA NETWORK LATENCIES MEASURED WITH THE PING UTILITY.

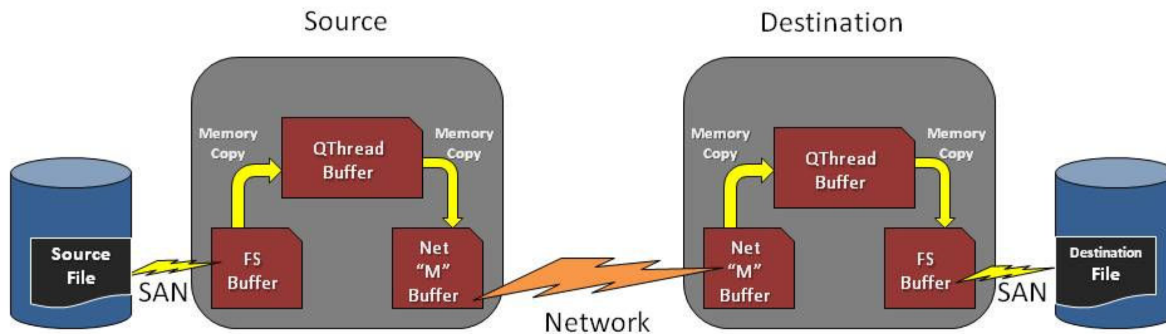
Laboratory in Seattle, and the Stanford Linear Accelerator Center in Sunnyvale, California. The network was constructed with two lambdas; however, since we configured the network to both begin and end at Oak Ridge, we were only able to achieve single OC192 connection speeds (9.6Gbps). Effectively, we were able to construct network loops of different lengths by constructing a cross-connection within the core switch of each participating network site. Table I shows the round-trip-time (RTT) measured with the ping utility for each of the wide area network configurations.

D. Data Movement Software

Several software packages have been built for high performance data movement. In this section we focus on describing two of the most popular tools, BBcp and GridFTP, and our own data movement software package, XDD.

1) BBCP: Originally built to transfer large data sets as part of the BaBar Collaboration [11], BBCP was developed by the the SLAC National Accelerator Laboratory (formerly the Stanford Linear Accelerator Center). The major advantage of BBCP versus traditional file transfer tools such as FTP and SCP is increased performance for large data transfers, particularly over large distances. Additionally, BBCP does not require a long running server process, instead, upon invocation, BBCP will spawn a process on both the source and destination endpoints. The processes will then perform the file transfer requested by the user.

BBCP supports a host of data transfer options included multiple I/O threads, network compression, and an append mode that allows a previously cancelled transfer to be resumed. Additionally, BBcp provides support for an *ordered* transfer mode that ensure file data is read and written in serial order, and an *un-buffered* mode that support direct I/O file access as long as the file size is a multiple of 8KiB. Although we tested



Using Buffered I/O on each file system

- Disk to File System Buffer
- File System Buffer to Qthread Buffer (copy)
- Qthread Buffer to Network Mbufs (copy)
- Mbuf to NIC on Source
- NIC to NIC – Source to Destination
- NIC to Mbuf on Destination
- Mbuf to Qthread Buffer (copy)
- Qthread Buffer to File System Buffer (copy)
- File System Buffer to Disk

Fig. 6. An XDD End-to-End data copy operation using Buffered I/O to read the Source file and write the Destination file. Also note that the Network data is buffered in the MBufs.

the un-buffered and ordered transfer modes, we achieved the highest levels of file transfer performance by only setting the request buffer sizes.

2) GridFTP: GridFTP is a core service within the Globus Grid Toolkit, and provided by Globus' GridFTP servers and the *globus-url-copy* GridFTP client. GridFTP provides features for cluster-to-cluster end-to-end file transfers including file striping support, load balancing across independent numbers of source and destination hosts, multiple network protocols, and a data source plugin architecture that may be effective for improving file I/O performance. GridFTP requires the use of the Grid Security Infrastructure (GSI) to provide certificate based authentication for all file transfers.

For our testing we are using a single GridFTP server on the destination and globus-url-copy from the source machine. We are using the TCP network transport and set the TCP buffer size to 32MiB. We used the default routines for accessing file data which use the standard open, read, write and close system calls. GridFTP does not provide a direct I/O option for circumventing Linux kernel copies.

E. XDD

XDD was originally designed as a tool for characterizing the performance of disk subsystems from a single system [**?**]. Originally conceived as a command-line based UNIX program designed to consistently reproduce performance measurements for various disk I/O access patterns. XDD is capable of utilizing disk subsystems extremely efficiently, and producing much higher file throughput than many other disk accessing tools (e.g. the standard UNIX tool dd).

Although not designed to transfer file data over the network, the efficient disk access routines in XDD made it an excellent starting point for building a disk-aware file transfer utility. We added a very simple networking implementation that transforms XDD into a point-to-point data mover. In general we have attempted to copy the user interface of BBcp where possible. Transfers are initiated with a single command from the source machine, with the remote server being started on an as needed basis. There are no long running servers required to run XDD. Additionally, we have implemented a restart/resume capability that allows remote transfers that have failed or been cancelled during transfer to continue where the transfer left off. XDD also supports a configurable number of I/O threads; however, the number of network and disk threads must be the

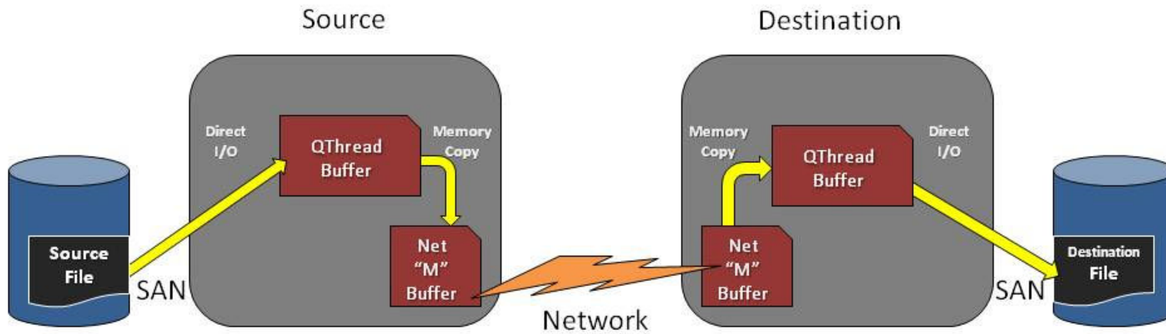


Fig. 7. An XDD End-to-End data copy operation using Direct I/O to read the Source file and write the Destination file. Note the absence of the File System Buffers - data is transferred directly from the Source file into the application buffer and from the application buffer to directly to the Destination file.

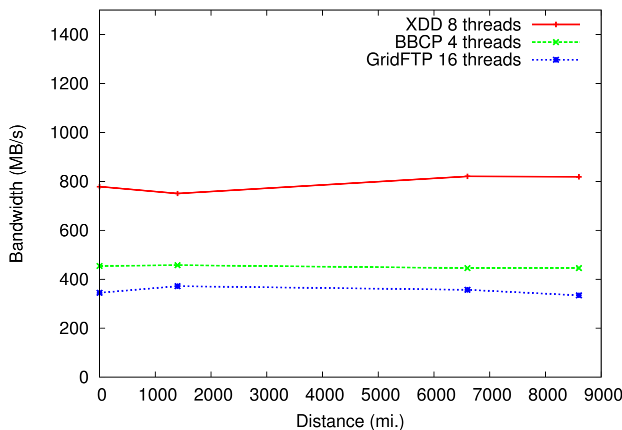


Fig. 8. Disk-to-disk file transfer bandwidth for each tool's highest performance configuration.

same.

IV. Long Distance Data Transfer Performance

As we described in Section II, sustained performance for a long distance data transfer requires efficient performance from each hardware component of the transfer and a mechanism for coupling the highest performance modes for each device into a high performance implementation. Figure 8 demonstrates the effectiveness of our approach for XDD in comparison to BBcp and GridFTP. For each tool, we present the best performing configuration over all of the network distances measured. While the file transfer performance for XDD is clearly superior to the transfer performance of BBcp and GridFTP, by providing a few tuning options dependent upon the transfer distance, it is possible to improve the performance of XDD a further 10%. The remainder of this section provides a detailed performance analysis of the hardware components and the three file transfer software packages.

A. Component Capabilities

We begin our performance analysis by examining the isolated performance of the disk arrays, network, and hosts. Each component participating in our transfer is capable of saturating a 10Gb interface with data; however, the parameters required to achieve high performance are not identical for each

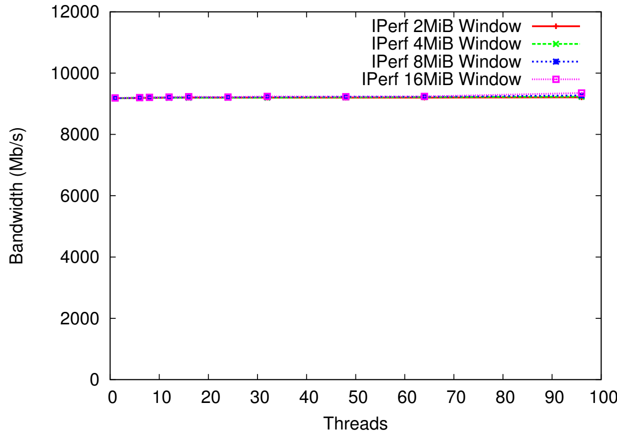


Fig. 9. USN network loop bandwidth measurements with IPerf cross-connected through Oak Ridge site.

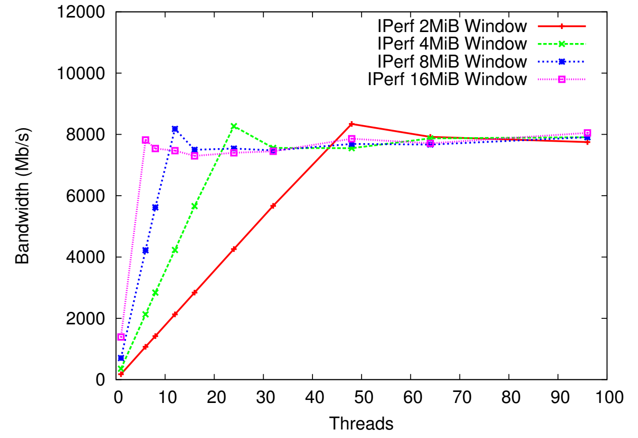


Fig. 12. USN network loop bandwidth measurements with IPerf cross-connected through Seattle site.

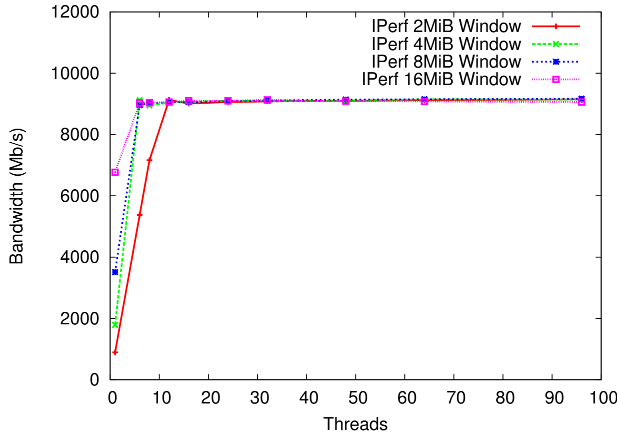


Fig. 10. USN network loop bandwidth measurements with IPerf cross-connected through Chicago site.

Fig. 11. Network bandwidth measured with IPerf over the Chicago loop.

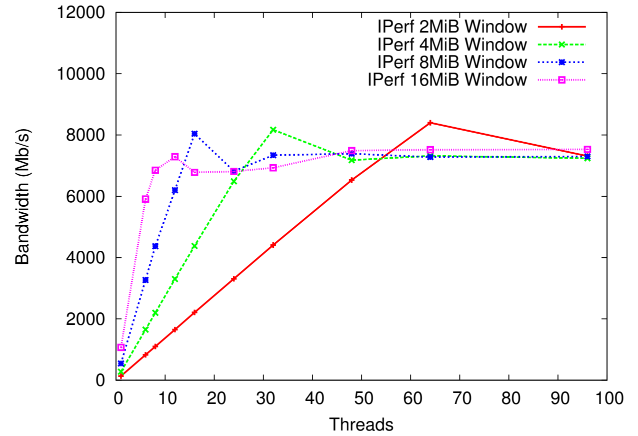


Fig. 13. USN network loop bandwidth measurements with IPerf cross-connected through Sunnyvale site.

component. In order to construct a 10Gb data transfer it is necessary to measure the overheads associated with matching each device's high performance modes.

B. Network Performance

Table I shows the distance and roundtrip latency measurements for each of the USN network loop configurations. Figures 9, 11, 12 and 13 show the network bandwidth in Megabits per second (Mbps). To perform the measurements we ran Iperf for 60 seconds while varying both the number of TCP flows and the TCP window size. The maximum possible TCP window size was limited to 16MiB by the sysctl configuration in Linux. In figure 9 we can see that with low latencies, network bandwidth can be maxed out by a single flow with a moderate window size. In figure 11,the latency is increased and it becomes necessary to increase the number of flows but not the TCP window size to achieve maximum throughput. At long distances, as in figure 12 and figure 13, high degrees of bandwidth are achievable, however we must employ multiple flows and a larger TCP window size. For example, over the Sunnyvale loop, it is necessary to use at least 8 flows to achieve 7290Mbps (911MB/s) with a 16MiB TCP window size.

C. Disk Array Write Performance

XDD was originally designed as a tool for measuring I/O system throughput. We have used the performance measurement features within XDD to profile writing 200GiB of random data to pod10 to measure the impacts of request size and I/O thread count on disk array write bandwidth. Our experiments demonstrated that 200GiB of data moved enough data to provide easily reproducible results. For all of these tests we have configured XDD to use direct I/O; for truly large transfers, the performance overheads associated with kernel buffer management make it impossible to operate our storage arrays within their highest performance mode. Figure 14 shows the performance of writing data without imposing any thread ordering. Because the array controllers

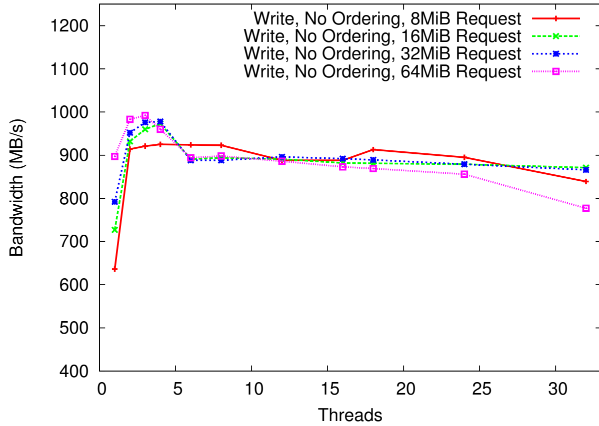


Fig. 14. Disk array write performance without request ordering.

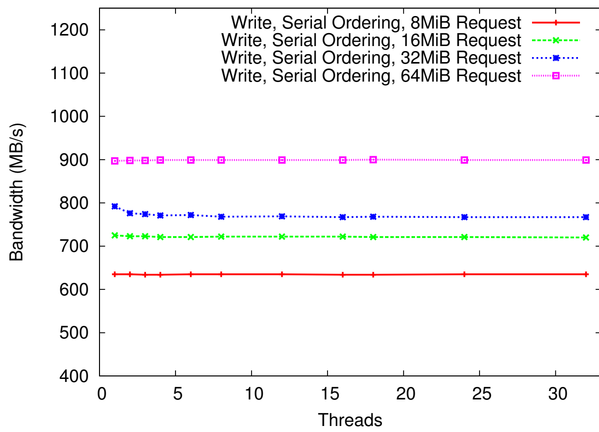


Fig. 15. Disk array write performance with serial request ordering.

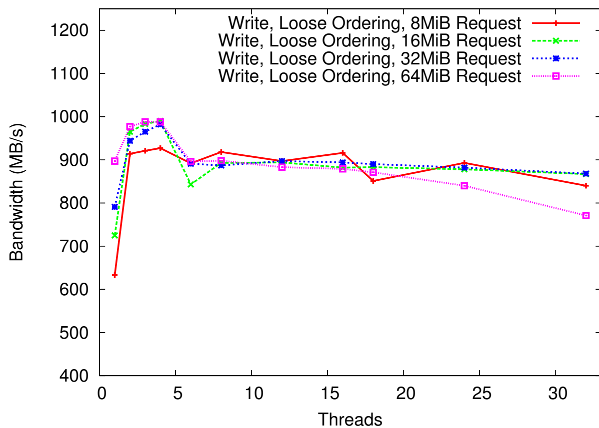


Fig. 16. Disk array write performance with loose request ordering.

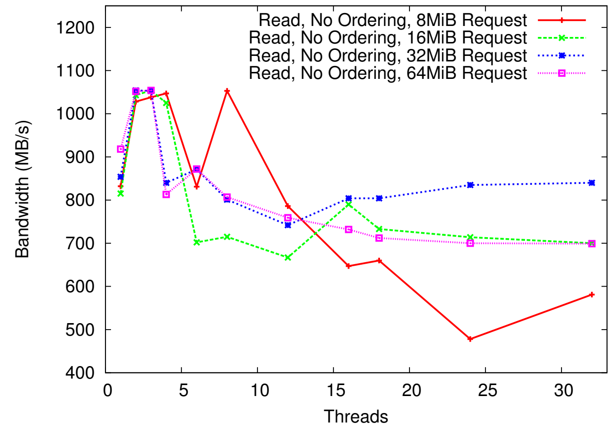


Fig. 17. Disk array read performance without request ordering.

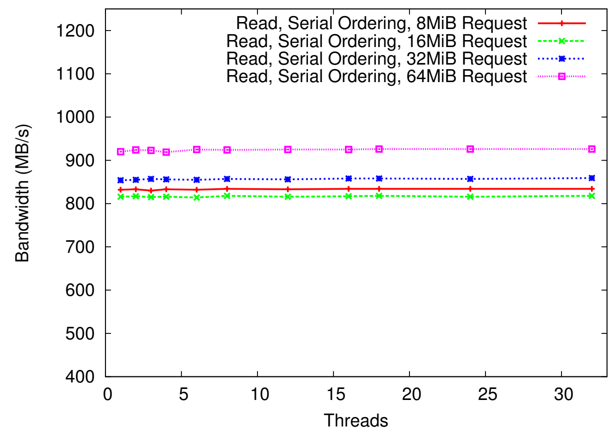


Fig. 18. Disk array read performance with serial request ordering.

support up 1024 queued SCSI commands, for most reasonable request sizes and thread counts the file write performance is excellent. In fact, for smaller numbers of threads, the no-order I/O thread scheduling strategy produces the highest measured file write bandwidth in all of our testing. Figure 16 shows the file write bandwidth with loosely ordered I/O requests. The observed file write bandwidths are virtually identical to the no-ordering strategy with the exception that performance is slightly lower due to increased synchronization overhead. The results of writing the file with serial ordering, shown in figure 15, are far different than the first two I/O request scheduling strategies. With serial ordering, the thread count is not a relevant factor in file write bandwidth; instead write performance is dictated almost solely by the request size. With a request size of 64MiB we were not able to achieve the performance levels offered by the more relaxed request scheduling algorithms; however, the stable disk performance levels are encouraging and an area of future development.

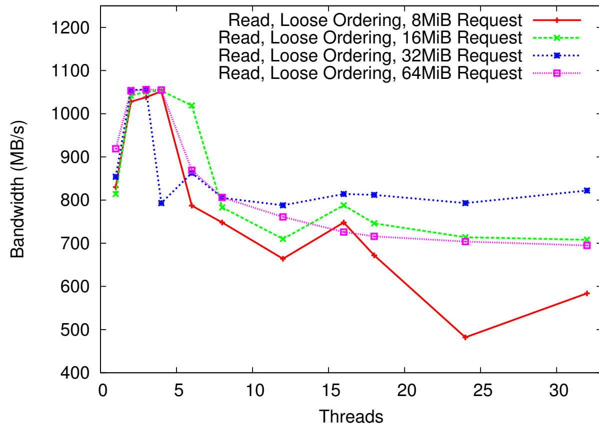


Fig. 19. Disk array read performance with loose request ordering.

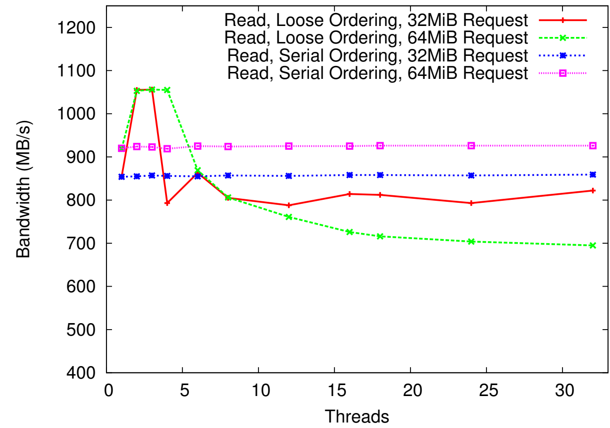


Fig. 20. Candidate disk read strategies for file transfers.

D. Disk Array Read Performance

We used the same configuration to measure the file read performance from disk, reading the 200GiB data file with varying request sizes and thread counts. Again we measured only direct I/O, as buffered I/O file system bandwidths were much lower. High levels of disk array read performance are more difficult to achieve in many cases because read requests for large file transfers typically do not complete in the cache. The incoming read requests need to be sequentially ordered to a much greater degree than write requests to achieve high levels of file system bandwidth. Still, our disk arrays supported some degree of data pre-fetching, caching and request re-ordering to achieve high levels of performance even when presented with non-sequential workloads. Balancing the overhead of mostly serializing the file request offsets while not contending for software locks is challenge for read intensive workloads. Figure 17 shows the performance of writing data without imposing any thread ordering. Although small numbers of reading threads provides high levels of file read bandwidth, without thread ordering large numbers of I/O threads cause the disk read performance to decrease dramatically. We then expect that loose ordering might provide enough request serialization to achieve consistently excellent file read bandwidth. Figure 19 demonstrates that for 32MiB request sizes it is the case that read performance is generally excellent, and the peak performance is the highest observed performance mode for our disk arrays. Figure 18 again shows that the serial ordering option decouples file read performance from the I/O thread count, and I/O request size again becomes the major performance factor.

E. Disk Array Performance Pairing

In order to understand file transfer performance from the disk perspective it is important to examine the how the file write strategy and file read strategy are paired together. Figure 20 and figure 21 show the I/O bandwidths for the request scheduling strategies and request sizes most practical for reading and writing a file, respectively. In XDD it is necessary to use the same request size on both the source and destination endpoints.

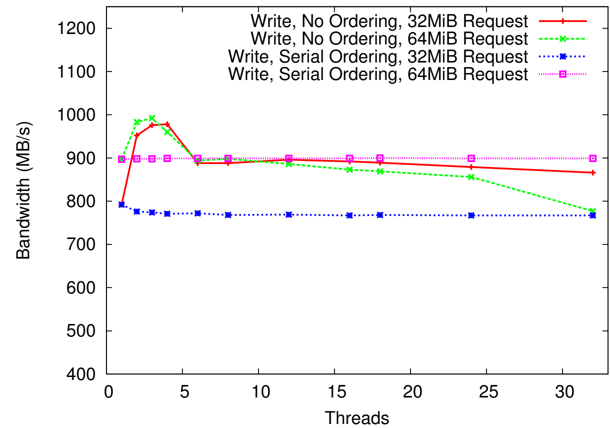


Fig. 21. Candidate disk write strategies for file transfers.

Figure 22 shows the I/O bandwidths associated with reading a with 32MiB requests and loose ordering and writing a file with 32MiB requests and no ordering. The highest performance modes are relatively well matched, and performance does not degrade excessively for either the source or destination as the thread count is increased (which may be necessary to saturate high latency networks). Figure 23 shows the same file read and write pairing with the exception that the request size is increased to 64MiB. Although the highest performance modes for the file read and write are well matched, we note that as the number of threads is increased, performance declines much faster for file reads

Due to the insensitivity to thread count, we have also examined the benefits of pairing serial ordered file reads and file writes. Figure 24 shows the file read and write bandwidth associated with serial ordering and a 32MiB request size. Although the highest performance modes available in the loose/no ordering pairings, the performance stability as the thread count is increased make serial ordering attractive for

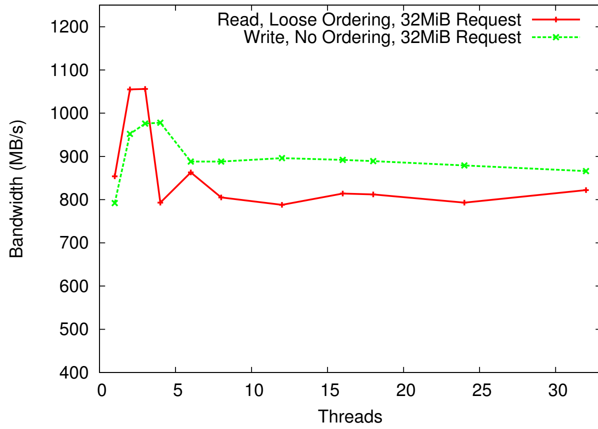


Fig. 22. Disk array read performance with serial request ordering.

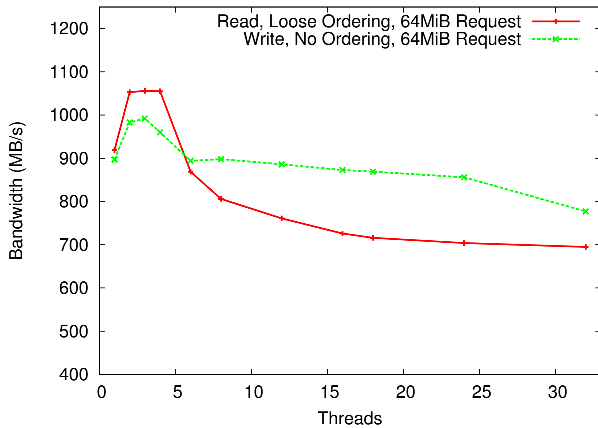


Fig. 23. Disk array read and write performance with serial ordering and 64 Mebibyte requests.

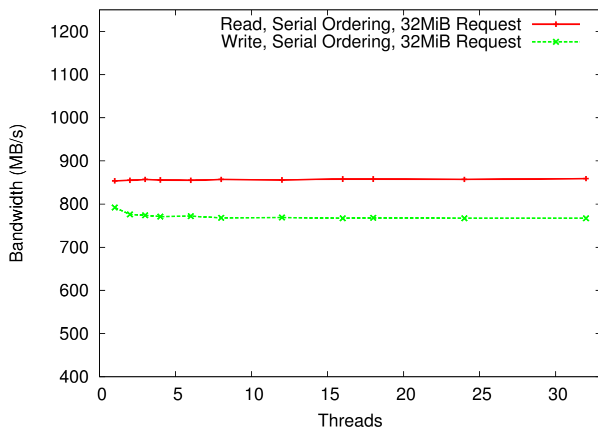


Fig. 24. Disk array read and write performance with serial ordering and 32 Mebibyte requests.

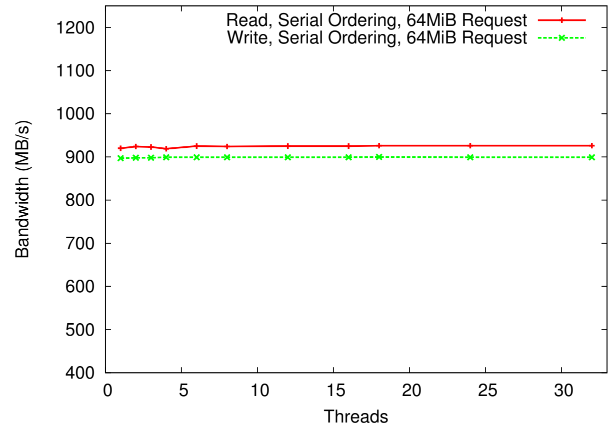


Fig. 25. Disk array read and write performance with serial ordering and 64 Mebibyte requests.

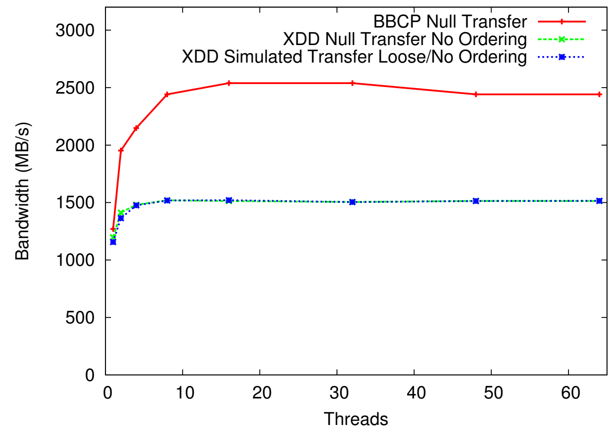


Fig. 26. Software overhead for BBCP and XDD measured as bandwidth.

extremely high latency networks. Similarly, the 64MiB request size pairing is also interesting, though software issues limited our testing with this combination for actual file transfers.

F. Host Performance

Figure 26 examines the software and host overheads of BBcp and XDD. To evaluate the host performance, our initial idea was to perform a transfer of 200GB of data from /dev/zero to /dev/null over the localhost connection. To ensure better measurement stability we used the taskset utility to assign the source process to the first CPU socket (processors 0-3) and the destination process to the second CPU socket (processors 4-7). Upon examining the BBcp source code, we learned that BBCP does not actually perform I/O to /dev/zero or /dev/null, instead it detects the special file endpoints and avoids the file I/O calls entirely. We implemented similar behavior in XDD to compare the implementations, and have labeled the lines as “Null Transfers” to annotate that the file I/O stack is not engaged. In the case of XDD we use the no ordering request scheduling option to most closely match the behavior

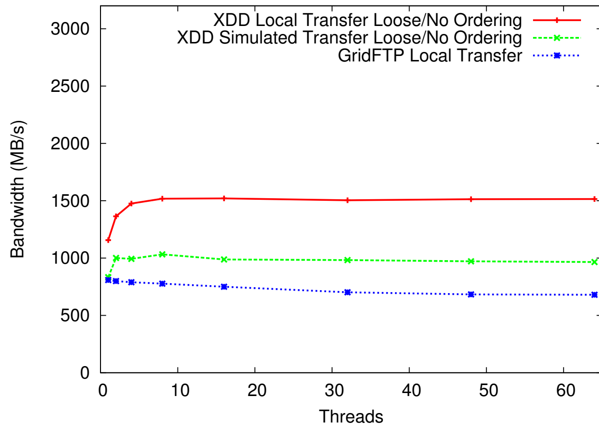


Fig. 27. Software overheads for GridFTP and XDD measured as bandwidth.

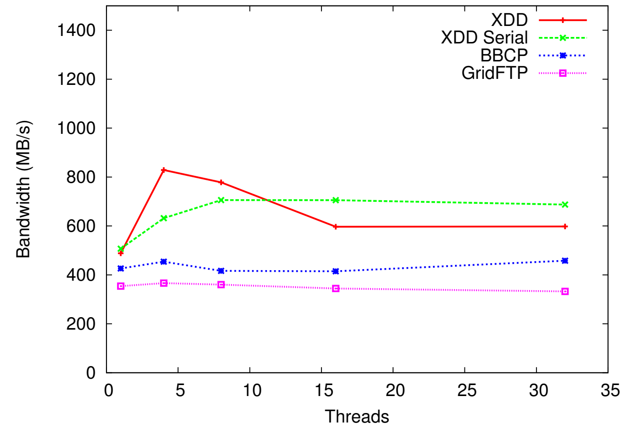


Fig. 28. Comparison of transfer performance over the ORNL loop.

of BBcp. We also provided data for a "Simulated Transfer" with XDD that uses a Null device and loose ordering on the source side (which more closely matches the direct I/O access implemented by XDD), and /dev/null with no ordering for the destination endpoint. The BBcp implementation clearly outpaces the performance of XDD. We attribute this primarily to the simplicity of BBcp which leads to a very fast code path. XDD expends CPU cycles managing the request ordering which leads to much larger software overheads. Also note that since the transfer is occurring entirely within a single host, the observed bandwidth numbers can likely be doubled to determine the capability of the host as a single endpoint in a file transfer.

Figure 27 provides similar data for comparing the host and implementation efficiencies for XDD and globus-url-copy, the Globus GridFTP client. For reference, we again provide data for the simulated transfer using XDD; however, the GridFTP client is not capable of direct I/O so we also provided measurements for a buffered I/O transfer from /dev/zero to /dev/null for both software tools. The lines titled "Local Transfer" indicate that the source reads data from /dev/zero, transmits the data over a socket connected to localhost, and writes the data to /dev/null. The XDD data clearly demonstrates that the cost of reading data from /dev/zero is significant, and that techniques such as direct I/O are important for avoiding unnecessary memory copies. One problem with GridFTP is that it provides no mechanism for leveraging direct I/O (and our measurements also indicate that direct I/O is not a performance boost for BBcp on our hardware either). Further, GridFTP seems to exhibit poor scalability on our hosts, as adding threads reduces performance even for small numbers of threads.

G. End-to-End File Transfer Performance

Having examined the performance of each of the individual components in the end-to-end file transfer, we note that we expect to be disk limited when using a single source host and single destination host. Our 9.6Gbps USN network can theoretically provide 1200MB/s of data throughput; however, our disk arrays are only barely capable of 1000MB/s. Further, we know that long distance network loops will require multiple flows, and the disk arrays perform best with multiple flows as well. However, the implementation of data movement from disk to network is critical to achieving performance.

1) Low Latency Networks: Figure 28 shows the end-to-end file transfer performance of a 200GiB file copied from pod7 to pod9 over the USN looped through the ORNL cross connection for XDD, BBcp, and GridFTP. For each tool we used a 32MiB request size. We have also included the transfer performance for XDD with serial ordering at both the source and destination. The best performing XDD configuration is almost twice as fast as the best performing BBcp configuration. Although we tried to further improve BBcp performance by using ordered and un-buffered configuration options, we achieved the highest observed performance by setting only the buffer size to 32MiB. The Globus GridFTP client provides far fewer optimization options, and so we were only able to set the request sizes. More interestingly, even though XDD is the only transfer tool capable of effectively leveraging the file system's direct I/O capabilities, the performance difference between proper impedance matched settings (4 threads achieving 829MB/s) and improperly matched transfer settings (e.g. 16 threads achieving 596MB/s) are nearly 30% different. Clearly proper impedance matching is important.

2) Increasing Latency: Figures 29, 30, and 31 show the end-to-end transfer performance over the Chicago, Seattle, and Sunnyvale network loops respectively. Again we see that the request scheduling approach and explicit buffer management used in XDD provides better transfer performance than BBcp and GridFTP. We also again note that choosing the best parameters is critical to maximizing the file transfer performance. Finally, we note that over the USN Chicago loop XDD using 16 threads with serial request ordering results in extremely poor file transfer performance. During our testing we occasionally observed that serial scheduling transfers exhibited poor performance. Although we weren't able to consistently reproduce the problem, we believe that

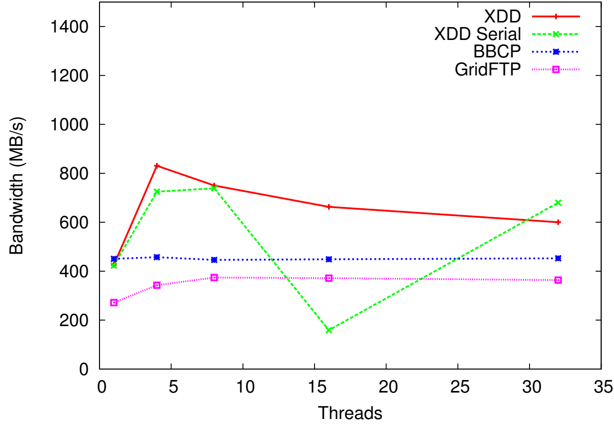


Fig. 29. Comparison of transfer performance over the Chicago loop.

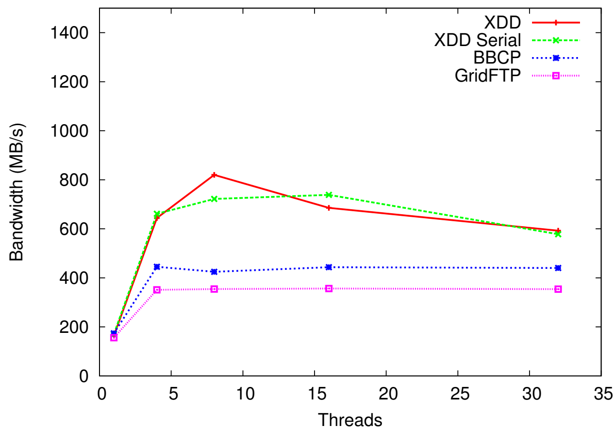


Fig. 30. Comparison of transfer performance over the Seattle loop.

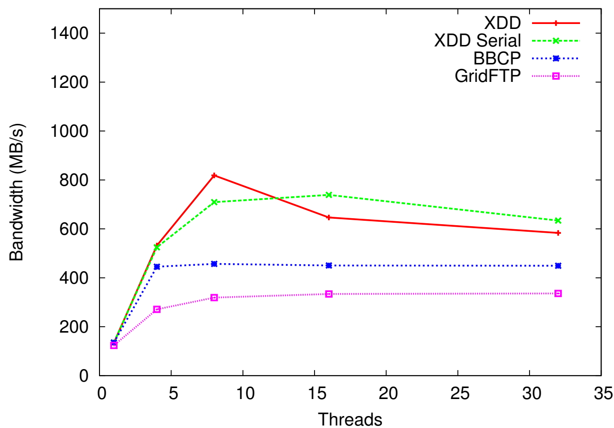


Fig. 31. Comparison of transfer performance over the Sunnyvale loop.

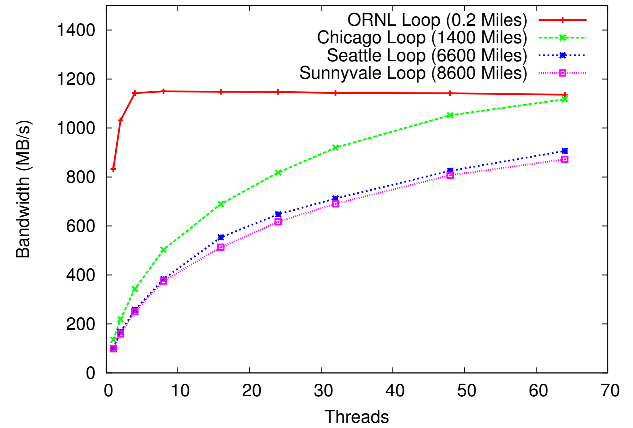


Fig. 32. XDD 2 Hosts transfer performance over dedicated WAN link.

an implementation defect is likely the cause of the instability, and we intend to further examine the issue as part of our further exploration of serial request ordering for end-to-end file transfers.

3) Multiple Hosts: Finally, while an efficient implementation will be critical in moving data at extreme performance levels, scalability beyond single endpoint transfers will be necessary as networking technology moves to 40Gb Ethernet and 100Gb Ethernet. We have implemented both multi-host and multi-NIC capabilities within XDD. Figure 32 shows the performance using two source endpoints and two destination endpoints to transfer a 200GB file over each of the USN network loops. XDD is able to saturate the network by adding flows at each endpoint. For this experiment we did not use a parallel file system, instead we replicated the on the local source file systems, and transferred half of the file to each destination endpoint. Although we could use custom data distributions with a file system such as PVFS to reconstruct the file on the destination side [7], we are interested in alternative methods of accelerating parallel file system based transfers. Although BBcp does not offer multi-host capabilities, GridFTP offers extremely flexible multi-host support. However, due to the network bottleneck limiting the performance of this test, we did not perform testing with GridFTP.

V. Discussion

Our experience developing file transfer capabilities for XDD has demonstrated that an impedance matching model is an effective scheme for high performance end-to-end file transfers. Leveraging the highest performance modes of each of the constituent devices is critical in achieving high performance file transfers. Earlier tools, such as BBcp and GridFTP, provide adequate performance for many types of data transfers; however, with XDD we have focused on pushing file transfer performance as far toward device speeds as possible. We believe that the types of techniques described in this paper are critical to achieve the I/O performance goals set forth for Exascale computing initiatives.

The key benefits offered in XDD's end-to-end file transfer capabilities are the I/O device scheduling strategies and the explicit buffer management. By carefully managing how the file I/O requests are issued to the underlying storage devices we can extract the maximum performance from storage devices ranging from single disks to complicated storage arrays. Additionally, while it may seem most beneficial to perform direct memory access (DMA) between the storage devices and network interface(s), our results indicate that an impedance matching approach that manages the buffers within the file transfer tool provides access to device level performance. The end-to-end performance levels attained by XDD are only possible by taking the time to understand how the application requests are serviced by the file system, how the file system requests are serviced by the operating system, and how the operating system requests are serviced by the storage hardware.

VI. Future Work

In the immediate future we are planning to push XDD to perform full performance end-to-end file transfers over a dedicated 40Gigabit Ethernet network. In order to achieve 40Gigabit performance levels, with the highest efficiency possible, our intent is to use both multiple hosts and multiple NICs per host. At present, we use TCP/IP as our only network protocol; however, the extensive kernel level interactions and buffer copies requred by TCP offer several opportunities for improvement with alternative network protocols. We intend to explore zero-copy networking approaches based on the Infiniband software stack and user-space UDP-based protocols such as UDT in the near future.

Additionally, to achieve 5-15GB/s of file system bandwidth it will be necessary to leverage parallel file system technology to produce a consistent file at the destination site. Our goal is not to abandon our disk-aware approach to end-to-end transfers, but instead to augment our request scheduling techniques to work with parallel file systems. This may involve modifying the underlying parallel file system to expose detailed disk layout information, or running XDD from within the file system I/O nodes. As we test file transfer techniques and approaches at 40Gigabit (5GB/s), we will also keep an eye toward 100Gigabit (12.5GB/s) networks and beyond. A fully saturated 100Gigabit connection will be capable of transferring nearly 1 Petabyte of data per day.

Acknowledgments

This work was supported by the Department of Defense (DoD) and used resources at the Extreme Scale Systems Center, located at Oak Ridge National Laboratory (ORNL) and supported by DoD. This research also used resources at the National Center for Computational Sciences at ORNL, which is supported by the U.S. Department of Energy Office of Science under Contract No. DE-AC05-00OR22725.

References

[1] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster, *The Globus striped GridFTP framework and server*, Proceedings of the 2005 ACM/IEEE conference on Supercomputing (Washington, DC, USA), SC '05, IEEE Computer Society, 2005, pp. 54–.

[2] The Globus Alliance, *The Globus Toolkit*, http://www.globus.org/toolkit/.

[3] R. Barrett, S. Hicks, S. Hodson, J. Lothian, S. Poole, and N.S.V. Rao, *Yottayotta gsx-3000 integration and performance report*, Tech. report, ORNL, 2005.

[4] Andrew Hanushevsky, *BBCP*, http://www.slac.stanford.edu/~abh/bbcp/.

[5] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao, *Oceanstore: an architecture for global-scale persistent storage*, SIGARCH Comput. Archit. News **28** (2000), 190–201.

[6] Kazumi Kumazoe, Masato Tsuru, and Yuji Oie, *Performance of high-speed transport protocols coexisting on a long distance 10-gbps testbed network*, Proceedings of the first international conference on Networks for grid applications (ICST, Brussels, Belgium, Belgium), GridNets '07, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007, pp. 2:1–2:8.

[7] R. Latham, N. Miller, R. B. Ross, and P. H. Carns, *A next-generation parallel file system for Linux clusters*, LinuxWorld Magazine (2004).

[8] T. Marian, D.A. Freedman, K. Birman, and H. Weatherspoon, *Empirical characterization of uncongested optical lambda networks and 10gbe commodity endpoints*, Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on, 28 2010-july 1 2010, pp. 575 –584.

[9] N.S.V. Rao, W.R. Wing, S.M. Carter, and Q. Wu, *UltraScience Net: network testbed for large-scale science applications*, Communications Magazine, IEEE **43** (2005), no. 11, S12 – S17.

[10] Kenneth Matney Sr., Sarp Oral, and Shane Canon, *A first look at scalable I/O in Linux commands*, The 9th LCI International Conference on High-Performance Clustered Computing, 2008.

[11] Andrew Hanushevsky Stanford, Andrew Hanushevsky, and Marcin Nowak, *Pursuit of a scalable high performance multi-petabyte database*, 16th IEEE Symposium on Mass Storage Systems, IEEE Computer Society, 1999, pp. 169–175.

[12] Joshua Walgenbach, Stephen C. Simms, Kit Westneat, and Justin P. Miller, *Enabling lustre wan for production use on the teragrid: a lightweight uid mapping scheme*, Proceedings of the 2010 TeraGrid Conference (New York, NY, USA), TG '10, ACM, 2010, pp. 19:1–19:6.

[13] Yixin Wu, Suman Kumar, and Seung-Jong Park, *On transport protocol performance measurement over 10gbps high speed optical networks*, Proceedings of the 2009 Proceedings of 18th International Conference on Computer Communications and Networks (Washington, DC, USA), ICCCN '09, IEEE Computer Society, 2009, pp. 1–6.