University of California, Santa Cruz

Large Genomic Data Transfer Methods

GNET: Technical Report

HTTP://GNET.SOE.UCSC.EDU

Authors:

Manikandan Punniyakotti
MANI@UCSC.EDU

Sam Wood
SBWOOD@UCSC.EDU

June 22, 2011

1. Introduction

As genome sequencing becomes cheaper and more frequent there is a rising demand amongst bioinformatics researchers for a quick method of transferring large datasets over long distances for collaboration. For example, the *1000 Genomes Project* provides a database of freely available sequenced genomes with the aim of promoting research in understanding disease through genotype variation. The complete uncompressed human genome in the KOREF_20090131 dataset is 3.0GB in size (FASTA format without metadata or annotations)[7], and 892MB using Gzip compression. Whereas annotated genomes that include metadata such as sequence alignment (SAM or BAM format) can greatly increase the file size and reduce the effectiveness of compression. For example, the NA12878 uncompressed annotated human genome (BAM format) from the Broad Institute is 337GB.

Data transfers of dozens of genomes between data centers on opposite sides of the United States are daily occurrences. This report examines networking and protocol performance issues that arise when transferring genomic data between hosts on a high speed network such as Internet2, with long round trip times (RTTs). This type of connection is known as a long fat network (LFN) due to the large bandwidth delay product, and requires a different treatment than typical data transfer solutions for LANs or MANs. We examine paths over high speed networks on the continental United States that have RTTs in the range of 40ms to 100ms, bandwidth from 1Gbps to 40Gbps, and 5 to 15 hops. These paths differ from other LFNs such as geosynchronous orbit satellite connections in that satellites typically have higher RTTs, less bandwidth, and fewer hops. It is known that the default TCP buffer sizes and congestion avoidance algorithms are not tuned for such networks, nor are the typical application file transfer protocols that use TCP, including FTP, HTTP, and SCP.

The goal of this paper is to provide a set of guidelines to end hosts of a large genomic data transfer that will reduce the total transmission duration (compared to existing methods), although most of these suggestions should be applicable to any large data transfer over LFNs. To be sure, this paper's scope is limited to changes on the end hosts and assumes an immutable intermediate network. Secondary goals include providing secure encryption and network fairness. We justify these guidelines through extensive emulation, with realistic parameters from network traces. Specifically, we use the Dummynet network emulator to model a LFN with characteristics similar to that of several Internet2 paths used in current large genomic data transfers. Also, we examine three large data transfer applications: GridFTP, FDT, and paraFetch, along with their respective optimal TCP settings. A key component to minimizing transfer delay is understanding the interplay between TCP and the data transfer application.

Lastly, we comment on novel genome-specific data compression techniques for genomic data. Indeed, we argue that the best way to reduce data transfer duration is to reduce the amount of data sent over the network, and genomic data in particular can be redundant. We conclude with future work that exploits the redundant nature inherent to genomic data, as well as possible network changes that could further reduce transfer duration.

1.1 Contents

Chap. 2 gives a brief background and commentary to the tools and practices currently used by data centers to transfer large datasets across high capacity, high RTT links.

Chap. 3 details a test plan and a discussion about the relevance of the test plan's results in genomic data distribution. Additionally, we provide preliminary results for several network scenarios.

Chap. 4 lists our suggestions and details what remains as further research. In particular, we comment on the role of genome-specific compression for genomic data in data transfer and validation of our emulated network.

2. Background

TCP is very popular for unicast communications and has been packaged with commodity operating systems and networking APIs. Hence TCP is widely used by networking applications over a variety of networking media. Additionally, TCP solutions are preferred from a logistical standpoint since many network administrators rate limit UDP traffic or block it all together, whereas TCP is universally available.

However, for long-haul high bandwidth networks (commonly called Long Fat Networks) commodity TCP has been found to be less suitable. This disadvantage is because even if the link is slightly error-prone, TCPs conservative congestion control mechanisms reduce the throughput heavily by underutilizing the large bandwidth delay product. Also, TCP provides reliability through ACKS and retransmissions and the latency of a packet recovery is at least an RTT which is unsuitable for such long delay links. Also, TCP requires large buffers at the end hosts to fully fill the bandwidth delay product. In order to perform efficient bulk data transfers over such networks without changing the underlying network architecture, and with TCP as the transport layer, several approaches such as tuning the TCP parameters at the end-hosts, using better congestion control algorithms, and using sophisticated data transfer tools are employed.

2.1 TCP Parameters

TCP tuning generally refers to adjusting the TCP buffers that correspond to the TCP windowing mechanism. Most applications do not try to understand the network in detail, nor learn the distance to the other end of the communication. A solution to this oversight is TCP auto-tuning with pre-configured limits. Sender-side auto-tuning was introduced in Linux 2.4 while receiver-side support was added in Linux 2.6. However, some of the default values that are used for auto-tuning still are not optimized for LFNs.

Fasterdata[1] suggests several changes that need to be made to the Linux TCP kernel settings, typically stored in "/etc/sysctl.conf", to improve TCP auto-tuning. For TCP max buffer size, 16MB is recommended for most 10Gbps paths and 32 MB is suggested for very long RTT, 10Gps or 40Gps paths. (see Fig. 2.1).

The auto-tuning "maximum TCP buffer" limits should be changed to 16MB as well, while leaving the minimum and default TCP buffer sizes as their defaults (see Fig. 2.2).

There are settings available to regulate the size of the queues between the kernel network subsystems and the driver for the network interface card. There are two queues to consider: txqueuelen is the transmit queue size and the netdev_backlog determines the reciever queue size. The receiver's queue fill up when an interface receives packets faster than the kernel can process them. If this queue is too small then packets will drop at the receiver, rather than due to the network. Fasterdata recommends a value of 30000 for the receiver's incoming packet backlog queue (the net.core.netdev_max_backlog parameter).

[1] Fasterdata [5] is a knowledge base dedicated to informing network administrators on how to transfer large (hundreds of gigabytes to terabytes) datasets over LFNs. It is part of the Energy Sciences Network (ESnet) a high-speed network serving the United States Department of Energy.

```
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
```

Figure 2.1: Recommended TCP max buffer sizes for LFNs

```
net.ipv4.tcp_rmem = 4096 87380 16777216
net.ipv4.tcp_wmem = 4096 65536 16777216
```

Figure 2.2: Recommended TCP auto-tuning buffer sizes for LFNs

Linux supports pluggable congestion control algorithms. Fasterdata recommends CUBIC [9] or HTCP since they do not rely on RTT values for adjusting the congestion window sizes (the net.ipv4.tcp_available_congestion_control parameter). Fasterdata recommends NIC tuning by modifying /etc/rc.local to load the settings at boot time. Specifically, they recommend a transmission queue length of 10000 for 10Gbps NIC cards.

SpeedGuide [15] is a website dedicated to informing network administrators on improving Broadband Internet performance. They recommend enabling selective acknowledgments, enabling TCP window scaling (to allow window sizes to exceed 65535), and disabling timestamps to save 12 bytes of header overhead (with the caveat that some congestion control protocols require accurate timestamps). See Fig. 2.3, and the description below detail several net.ipv4.* parameters.

Note that tcp_rmem and tcp_wmem take three parameters: a minimum, initial and maximum buffer size. They are used to set the bounds on autotuning and to balance memory usage while under memory stress. To be sure, these control the actual memory usage (not just TCP window size) and include memory used by the socket data structures as well as memory wasted by short packets in large buffers. It is suggested that the maximum values should to be larger than the BDP of the path by some suitable overhead.

tcp_sack: enables Selective Acknowledgments (SACK) to handle lossy connections. This option selectively acknowledges each segment in a TCP window. This makes it possible to only retransmit specific parts of the TCP window which lost data and not the whole TCP window. This means that if a certain segment of a TCP window is not received, the receiver will not return a SACK for that segment. The sender will then know which packets where not received by the receiver, and will hence retransmit that packet.

tcp_timestamps: used to calculate the Round Trip Time by some congestion control protocols. Adds an additional 12 bytes to the packet header.

tcp_wmem: has 3 parameters which apply to each TCP socket:

- The first value designates the minimum TCP send buffer space available for a single TCP socket. This space is always allocated for a specific TCP socket as soon as it is opened. This value is normally set to 4096 bytes (4 kilobytes).
- The second value designates the default buffer space allowed for a single TCP socket. If the buffer exceeds this limit, it may get hampered if the system is currently under heavy load and does not have enough memory space available. Packets could be dropped if the system is so heavily loaded that it can not give more memory than this limit. This value overrides the

```
net.ipv4.tcp_sack = 1
net.ipv4.tcp_window_scaling = 1
net.ipv4.tcp_timestamps = 0
```

Figure 2.3: Recommended SpeedGuide.net TCP settings

/proc/sys/net/core/wmem_default value that is used by other protocols, and is usually set to a lower value than the core value.

- The third value designates the maximum TCP send buffer space. This defines the maximum amount of memory a single TCP socket may use. However, if you ever do need to change it, you should keep in mind that the /proc/sys/net/core/wmem_max value overrides this value, and hence this value should always be smaller than that value.

tcp_rmem: has 3 parameters which apply to each TCP socket:

- The first value designates the minimum receive buffer for each TCP connection, and this buffer is always allocated to a TCP socket, even under high pressure on the system. This value is set to 4096 bytes (4 kilobytes).
- The second value designates the default receive buffer allocated for each TCP socket. This value overrides the /proc/sys/net/core/rmem_default value used by other protocols.
- The third value designates the maximum receive buffer that can be allocated for a TCP socket. This value is overridden by the /proc/sys/net/core/rmem_max if the ipv4 value is larger than the core value.

In addition to the net.ipv4.* parameters, one can also set the maximum buffer size that applications can request: the maximum acceptable values for SO_SNDBUF and SO_RCVBUF arguments to the setsockopt (system call) can be limited with net.core.* variables: the maximum receive window (rmem_max) and the maximum send window (wmem_max).

2.1.1 perfSONAR

PerfSONAR is an infrastructure for network performance monitoring, making it easier to solve end-to-end performace problems on paths crossing several networks. It composes of several network monitoring tools, including: BWCTL (Bandwidth Test Controller) that can use Iperf, Thrulay or Nuttcp; OWAMP (One Way Ping); NDT (Network Diagnostic Tool); ping and traceroute.

As a part of this project, we setup a perfSONAR node at UCSC to monitor the paths to genomic data destinations around the Internet. We studied the characteristics of these paths and used them to configure our Dummynet testbed so that our experiments yield realistic and accurate results (see Sec. 3). We used perfSONAR to obtain the following parameters: RTT, packet loss rate, end-end throughput, and the number of hops.

2.2 Data Transfer Tools

The tools that are used for data transfer play a vital role in minimizing the total end-to-end transmission duration. Tools that support TCP transfers through parallel streams are needed since frequently multiple streams perform better with packet loss.

Also, WAN transfers have much higher latency than LAN transfers but many tools such as SCP or SSH assume a LAN and use internal congestion control mechanisms that are inappropriate for our scenarios. We examined three open source and publicly available tools that are currently used by researches for large data transfers: FDT, GridFTP and paraFetch. In addition, we examine GridFTP's UDT extension and comment on Aspera, a proprietary UDP based tool.

2.2.1 Fast Data Transfer (FDT) Application

FDT is an application to do efficient data transfers over wide area networks with standard TCP. It is claimed that FDT is capable of reading and writing at disk speed over such networks. This tool is java based, and can run on all major platforms. FDT uses the capabilities of the Java NIO libraries and is based on an asynchronous, flexible multithreaded system. It supports several features including parallel data transfer, resuming a file transfer session without loss[2], and continuous streaming of a list of files using a managed pool of buffers. A large set of files can be sent and received at full speed without having to restart the network traffic between files.

This tool is simple to install and use: FDT has a server and client that are included in a single JAR file and includes two scripts to run the server and the client separately. Once the JAR file is placed on both hosts, and the server is started using the provided script, the tool is ready for use: just start the client script with appropriate options. Also, installation and operation does not require administrative privileges. FDT was the easiest to install and configure.

2.3 GridFTP Application

GridFTP ([1],[4]) is an extension of the standard FTP protocol (RFC959) and it is defined as part of the Globus toolkit. The Globus Alliance develops Grid technology to make resource management, security, and data management standardized and straightforward. GridFTP was developed to provide a more reliable and high performance file transfer for Grid computing applications that need to transmit very large files quickly and reliably. From a practical perspective, GridFTP was the most difficult to configure and install due to the multitude of features that it supports. For example, we were unable to compile GridFTP with OpenSSL support without root privileges.

GridFTP includes features such as: security with Grid Security Infrastructure (GSI), third party transfers (a local client can initiate remote transfers between servers), parallel and striped transfer, multiple source to single destination transfer, partial file transfer (transfers can be resumed from a specific point or transmission of just a subset of a file), fault tolerance and restart (can handle unavailability or sever problems and automatically restart after a problem), automatic TCP optimization (negotiation of TCP buffer sizes and window sizes to provide transfer speeds and reliability), data port range (to allow working around firewalls), intermediate proxies, and UDT [8] support.

GridFTP provides an extended FTP protocol on top of the Globus eXtensible Input/Output System (XIO) [2]. XIO is a middle layer framework that provides read, write, open, and close file stream semantics to higher level protocols. In particular,

[2]If a partially downloaded file already exists then FDT will not re-download this portion of the file.

GridFTP can flexibly switch between TCP, UDT and other transport drivers without modifying the GridFTP client or server. Indeed, Bio-Mirror [3], a website developed at the Genome Informatics Lab of the Indiana University Biology Department, provides mirroring of biology data sets using GridFTP with the UDT driver.

2.4 paraFetch Application

paraFetch is a bulk data transfer tool developed in-house at UCSC for Genome data transfers between data centers. It is entirely written in C and compiling the source code and installing is relatively difficult. paraFetch fetches files behind a webserver and hence we needed to setup paraFetch only on the client side provided we have a webserver already setup on the server side that hosts the files to be transferred. paraFetch supports both HTTP and HTTPS protocols. For our experiments we used Nginx [11], a lightweight webserver that supports both HTTP and HTTPS protocols and does not require root to install. paraFetch uses at most 50 parallel TCP streams using HTTP's support for requesting specific bytes of a file. We extended the 50 parallel stream limit by recompiling the source, however with this modification we were still unable to conduct transfers of more than 200 parallel streams. It remains unclear whether this limitation is due to paraFetch or Nginx.

Since paraFetch does not use any special algorithms for utilizing multiple streams, writing to disk, or congestion control, it provides an ideal multi-stream TCP baseline to compare the other tools.

3. Evaluation

We examined the protocols and TCP settings of Chap. 2 in a range of scenarios drawn from a full factorial experimental. The factors and levels are listed in Fig. 3.1. Although our focus is on LFNs, we have included a range of scenarios in order to better understand the interplay between TCP settings, data transfer application, and path characteristics. The levels in Fig. 3.1 were picked partly from closely examining two specific LFN paths that are currently used in genomic transfer (Sec. 3.1), and from conversations with UCSC network administrators regarding observed network characteristics. In addition to these factors, we also examined disk-to-disk and memory-to-memory data transfers, as each tool uses a different methodology for disk IO.

In order to run such a large range of configurations we decided to use the the Dummynet [14] network emulator which allows us to easily emulate a network with a specific latency, packet loss rate, bandwidth, and jitter. Fig. 3.2 details the Dummeynet configuration: three identical hosts[1] are connected by Ethernet in a linear topology, where the intermediate host runs Dummynet and forwards packets between the other two hosts, according to the emulated network characteristics.

The main metric of interest is total end-to-end transmission duration, with the goal of minimizing this metric. A secondary goal is maintaining network fairness: an optimal solution should be fast but not introduce packet loss in existing sessions. We used packet loss in non-genomic TCP data transfer sessions during genomic data transfer as a measure of fairness. Lastly, the solution must be compatible with some form of encryption, whether it is built-in or uses an existing tool such as SSH.

3.1 Granular, Multi-hop Case Study

Prior to running the full factorial experiment, we examined two Internet2 network configurations that are currently used in genomic data transfer: UCSC to the the Baylor College of Medicine in Houston, Texas (case A); and UCSC to the Broad Institute (case B) in Massachusetts. These two cases were picked since all three participate in genomic data transfer and their geographical distances result in a medium range RTT of roughly 42ms, and a long range RTT of 92ms, respectively. One would expect many genomic data transmission within the United States over Internet2 to fall within these ranges. Unlike the full factorial experiment, in these scenarios we chose to model each individual hop in order to model these paths more closely.

The RTT, capacity, number of hops, and packet loss parameters for Dummynet were set for the corresponding scenario using data collected from perfSONAR (see 2.1.1) between UCSC and the University of Texas (for case A), and between UCSC and the M.I.T. Lab for Nuclear Sciences (for case B). We were unable to find perfSONAR hosts directly at the Baylor College of Medicine nor the Broad Institute. Another caveat is that we used 1Gbps Ethernet throughout all of our scenarios, when in reality the Internet2 backbone supports up to 40Gbps. To be sure, most end hosts are connected to a switch with a lower rate.

[1] 2 Processor AMD Opteron(tm) Processor 246 HE, 8GB RAM, 1Gbps Ethernet

Latency	10ms	40ms	60ms	80ms	100ms
Packet Loss	0pkt/sec	1pkt/sec	2pkt/sec	3pkt/sec	4pkt/sec
Bandwidth	100Mbps	250 Mbps	500Mbps	750Mbps	1000Mbps
Jitter	0	$\frac{1}{3}$ Latency	$\frac{1}{2}$ Latency	$\frac{2}{3}$ Latency	Latency

Figure 3.1: Full factorial experiment factors

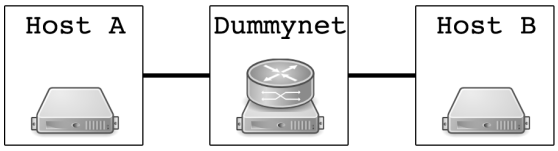

Figure 3.2: Testbench configuration

hop#	Est. RTT Delay(ms)	Dummynet Two-way Prop. Delay(ms)
1	0.173	0
2	1.062	0
3	7.983	4
4	0.151	0
5	30.843	14
6	0.254	0
7	0.100	0
8	2.116	2
9	0.100	0
total RTT (ms)	42.782	41 (20+queuing delay)

Table 3.1: Case A: Dummynet pipe delays compared to actual RTTs

`traceroute` was used to calculate the number of hops and intermediate node RTTs from UCSC to the corresponding destination. Dummynet allows multi-hop configurations with varying delays, but only at a millisecond granularity. Note that using `traceroute` to estimate RTTs between intermediate hops is not precise, as the RTTs include fluctuating queuing delays and are a bidirectional metric. Fig. 3.1 and Fig. 3.2 lists the Dummynet two-way propagation delays side-by-side with the delays calculated using `traceroute`, note that the total Dummynet RTTs are within two milliseconds of the actual RTTs. Additionally, the Dummynet delay is the two-way propagation delay and does not include queueing delays which can vary from system to system. To be sure, we expect that emulating the network using multiple pipes (to model a multi-hop path) is more imprecise than emulating the network with a single pipe, since each packet will require more CPU time and potentially more memory to copy between the pipes. Also, we expect that these scenario results are harder to reproduce since the Dummynet queuing delay is a function of the CPU speed and not a configurable Dummynet primitive. The two-way propagation delays were set by trying to preserve proportions for each hop and maintaining the Dummynet total RTT to be as close as possible to the actual RTT. For these scenarios, we used the recommended TCP settings for the end hosts throughout all of these experiments (Sec. 2.1), except that we left TCP timestamps enabled. We used the `htcp` TCP congestion control algorithm.

hop#	Est. RTT Delay(ms)	Dummynet Two-way Prop. Delay(ms)
1	0.099	0
2	0.957	0
3	8.066	2
4	0.117	0
5	75.762	7
6	0.013	0
7	0.061	0
8	0.129	0
9	0.143	0
10	0.048	0
11	0.294	0
12	0.024	0
13	0.135	0
14	7.412	1
15	0.035	0
total RTT (ms)	93.295	91 (10+queuing delay)

Table 3.2: Case B: Dummynet pipe delays compared to actual RTTs

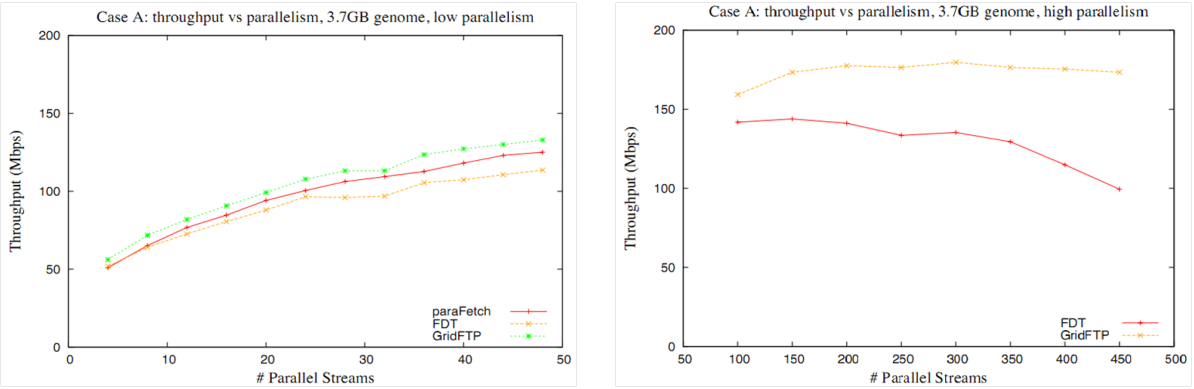

Figure 3.3: Case A: memory-to-memory, varying parallelism

Fig. 3.3 (Fig. 3.5) demonstrates the throughput in the memory-to-memory Case A (Case B) scenario, with a varying amount of parallelism, whereas Fig. 3.4 (Fig. 3.6) has the disk-to-disk results. The drop in the memory-to-memory scenario of Fig. 3.6 is due to two failed tests. It remains unclear why Case B has such poor performance: we believe it may be due to problems emulating a multitude of pipes; further investigation of the Dummynet emulator is needed. We conducted all the future experiments using a single pipe in order to avoid the Dummynet CPU effects which are exacerbated by multiple pipes.

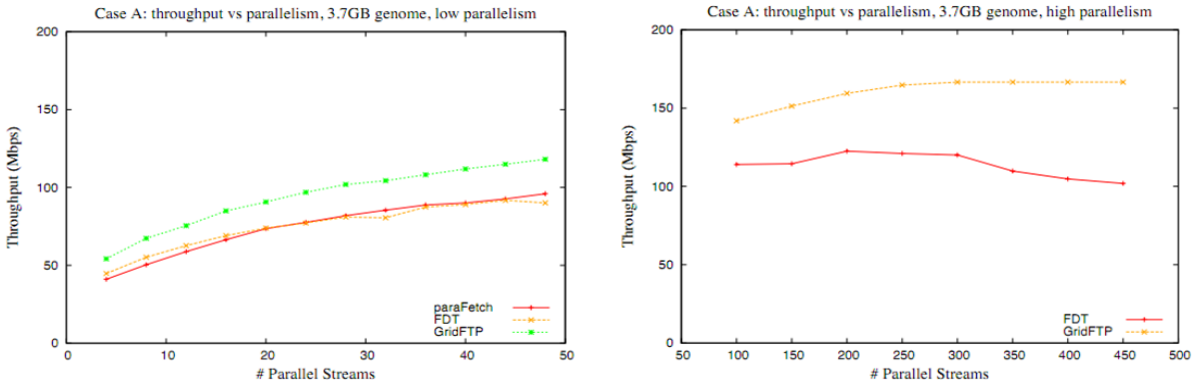

Figure 3.4: Case A: disk-to-disk, varying parallelism

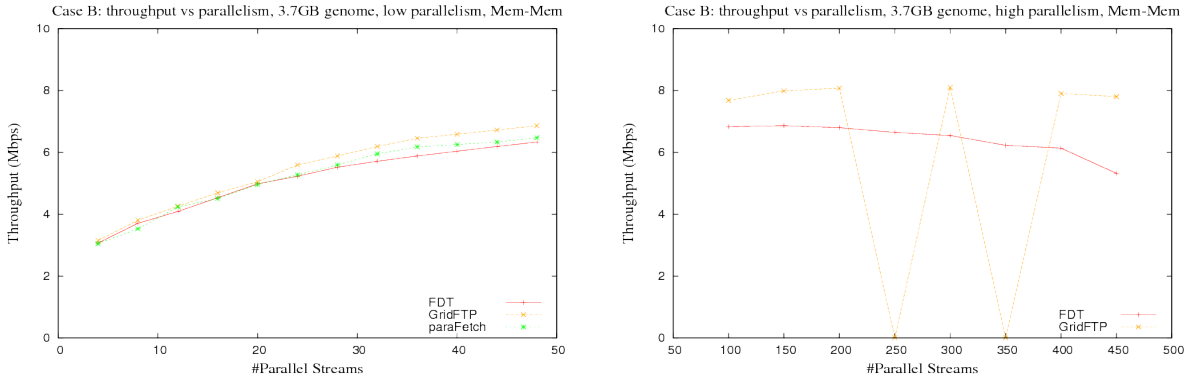

Figure 3.5: Case B: memory-to-memory, varying parallelism

3.2 Full Factorial Experiment

Fig.3.7-3.12 give per tool throughput of memory-to-memory and disk-to-disk tests for paraFetch, FDT, and GridFTP respectively. These tests were supposed to provide an upper-bound to the maximum achievable throughput for a given RTT, since there was maximum bandwidth with 0 packetloss.

Fig. 3.17 demonstrates some preliminary UDT results using GridFTP's UDT driver. We conducted several tests with a varying number of streams and did not observe any significant difference in throughput. Note that as with the other applications, UDT has slightly better performance in all memory-to-memory transfers. Unique to UDT, however, is that the throughput stayed within 200 to 350 Mbps for all RTTs tested. This stability suggests that UDT's congestion control is less dependent on RTT than the other applications. UDT's congestion control and reliability is similar to TCP in that it uses negative acknowledgments to explicitly indicate packet loss, while TCP's selective acknowledgments indicate all packets that are received. However, UDT's NACKs are timer based: a cumulative NACK is sent at a fixed frequency, whereas TCP's SACKs are event driven. More investigation is needed in order to understand UDT's effectiveness

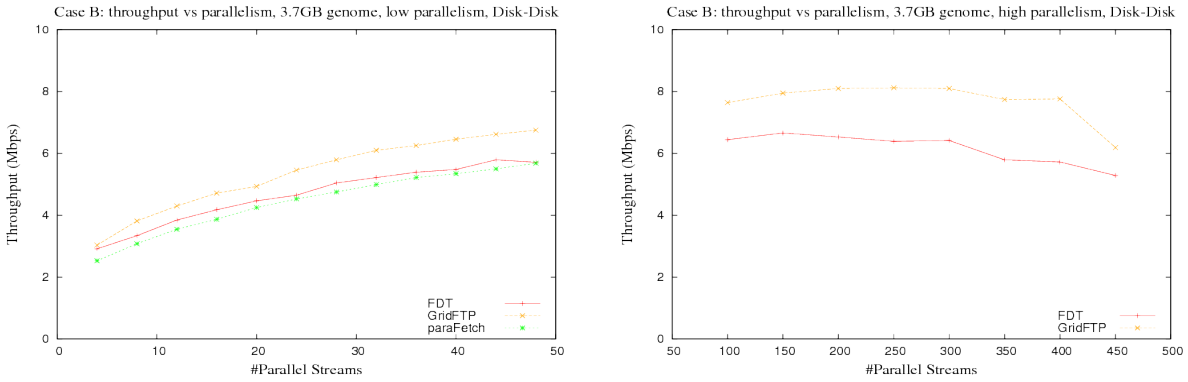

Figure 3.6: Case B: disk-to-disk, varying parallelism

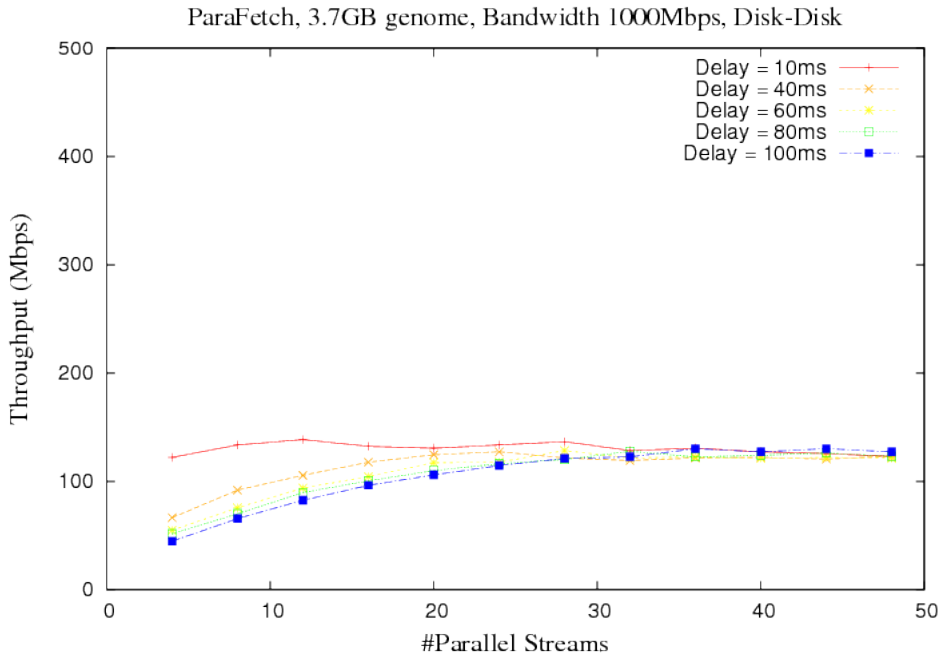

Figure 3.7: paraFetch memory-to-memory throughput at different RTTs with varying parallelism, 1Gbps bandwidth, 0 packet loss, recommended TCP settings and htcp TCP congestion control.

and determine what properties of the protocol make the throughput less dependent on RTT.

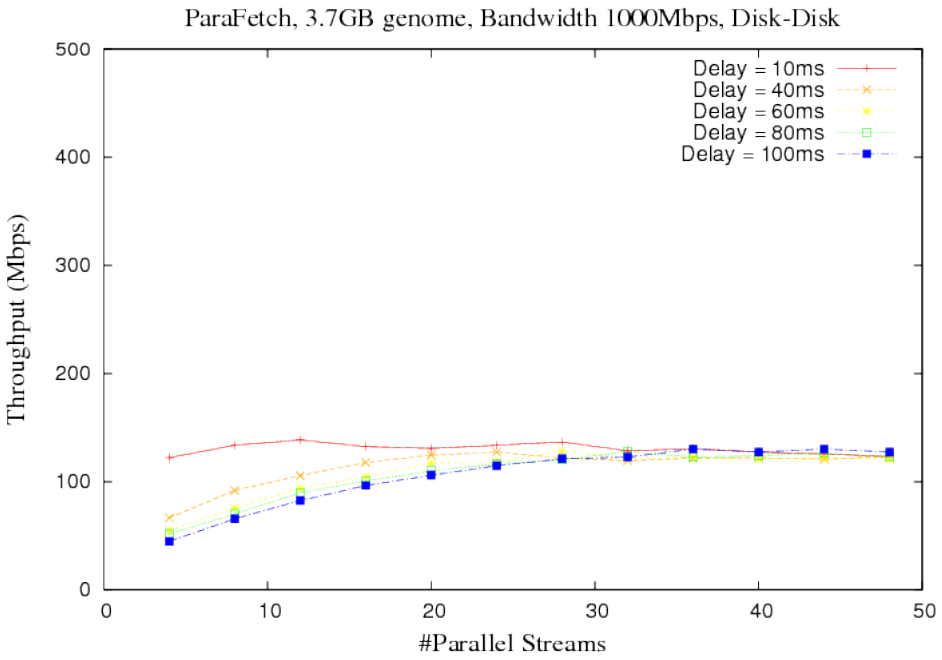

Figure 3.8: paraFetch disk-to-disk throughput at different RTTs with varying parallelism, 1Gbps bandwidth, 0 packet loss, recommended TCP settings and htcp TCP congestion control.

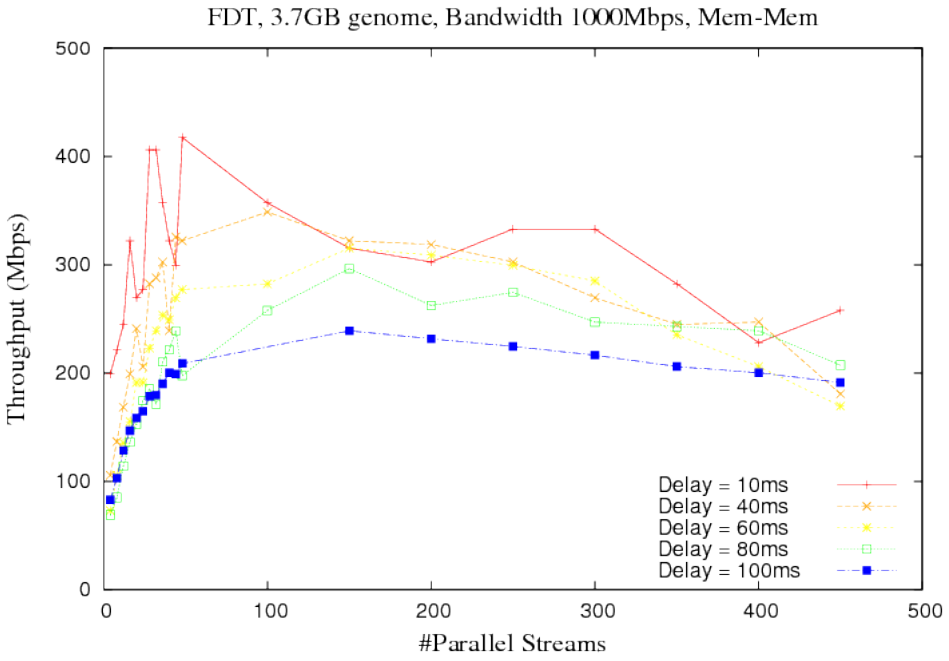

Figure 3.9: FDT memory-to-memory throughput at different RTTs with varying parallelism, 1Gbps bandwidth, 0 packet loss, recommended TCP settings and htcp TCP congestion control.

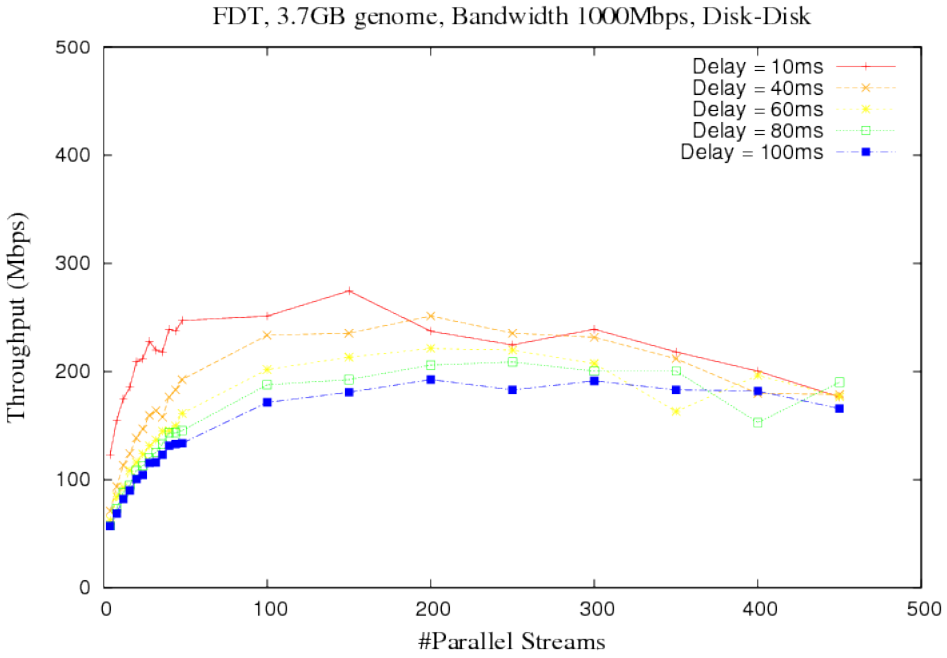

Figure 3.10: FDT disk-to-disk throughput at different RTTs with varying parallelism, 1Gbps bandwidth, 0 packet loss, recommended TCP settings and htcp TCP congestion control.

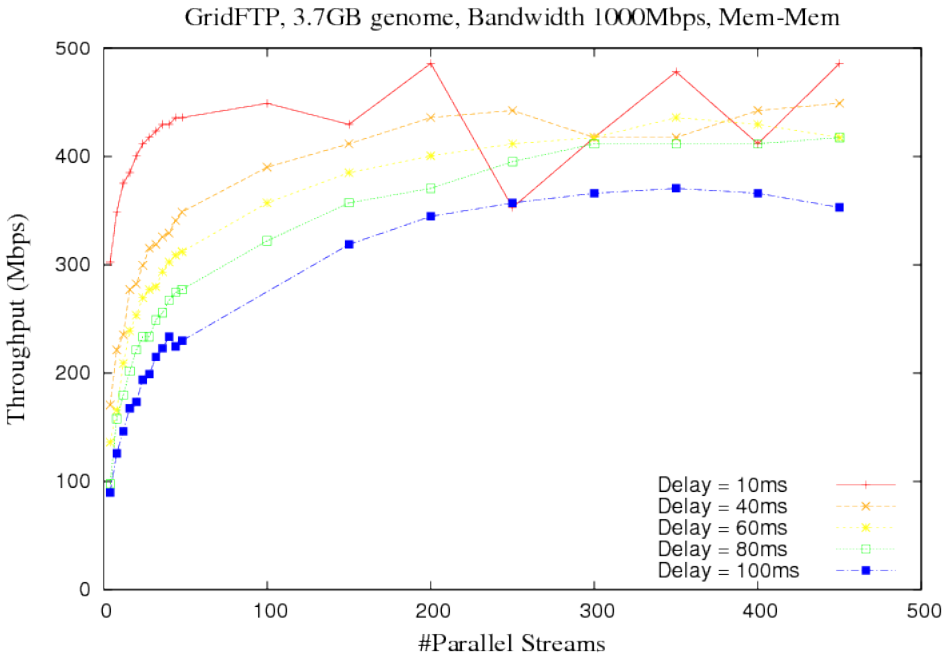

Figure 3.11: GridFTP memory-to-memory throughput at different RTTs with varying parallelism, 1Gbps bandwidth, 0 packet loss, recommended TCP settings and htcp TCP congestion control.

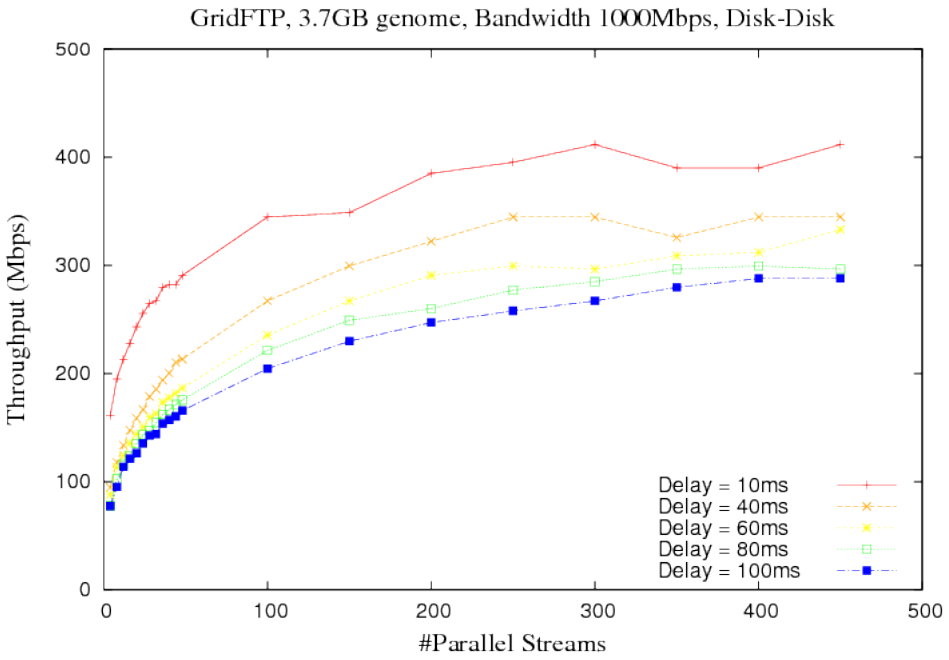

Figure 3.12: GridFTP disk-to-disk throughput at different RTTs with varying parallelism, 1Gbps bandwidth, 0 packet loss, recommended TCP settings and htcp TCP congestion control.

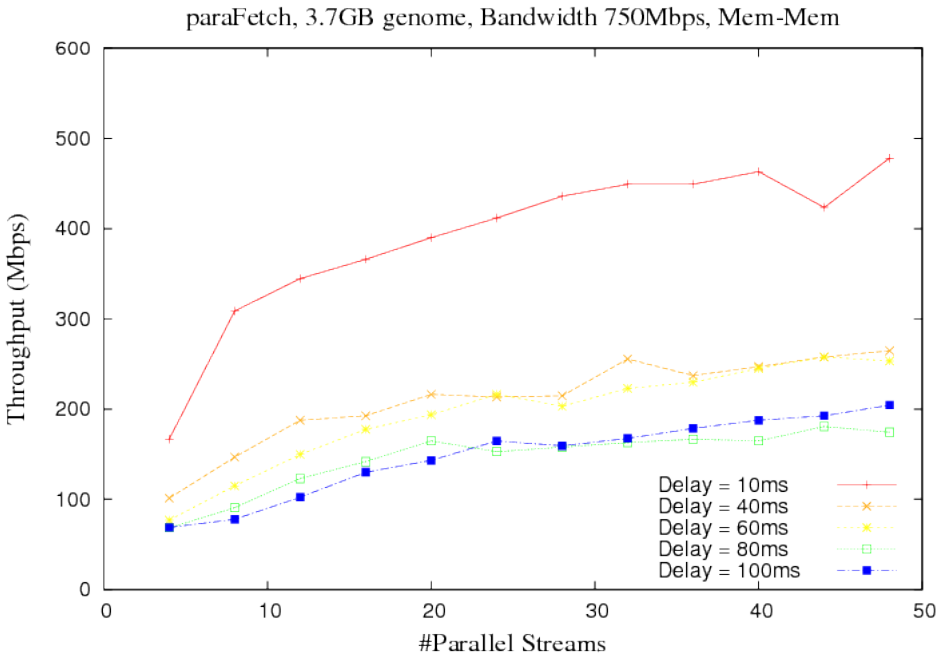

Figure 3.13: paraFetch memory-to-memory throughput at different RTTs with varying parallelism, 750Mbps bandwidth, 0 packet loss, recommended TCP settings and htcp TCP congestion control.

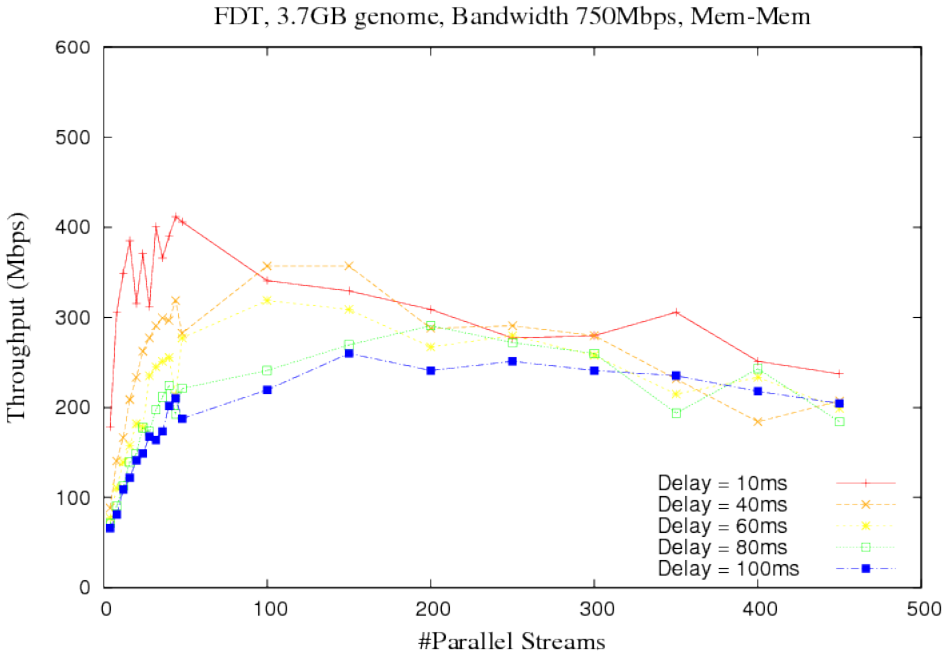

Figure 3.14: FDT memory-to-memory throughput at different RTTs with varying parallelism, 750Mbps bandwidth, 0 packet loss, recommended TCP settings and htcp TCP congestion control.

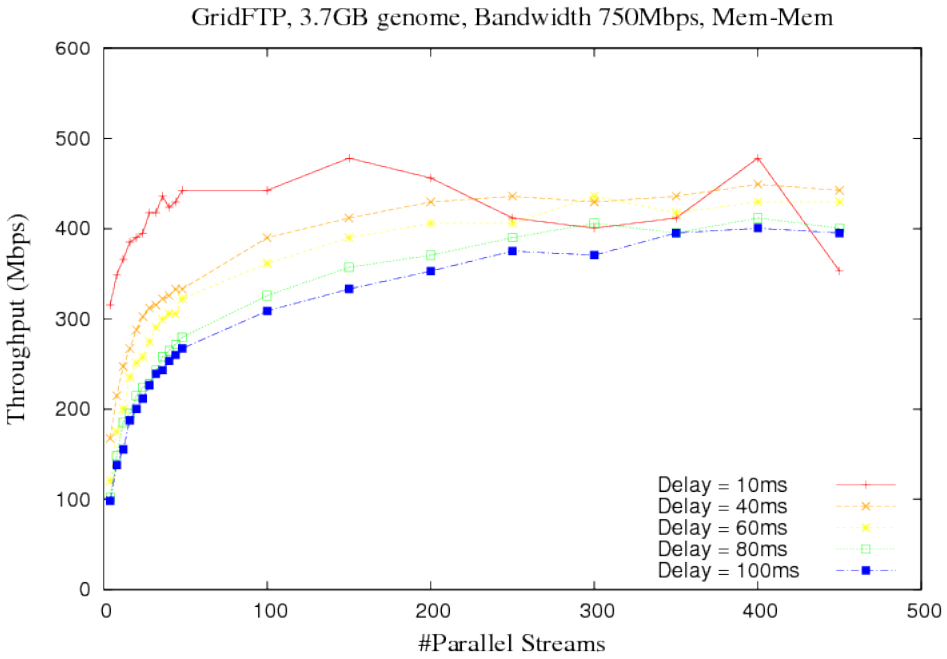

Figure 3.15: GridFTP memory-to-memory throughput at different RTTs with varying parallelism, 750Mbps bandwidth, 0 packet loss, recommended TCP settings and htcp TCP congestion control.

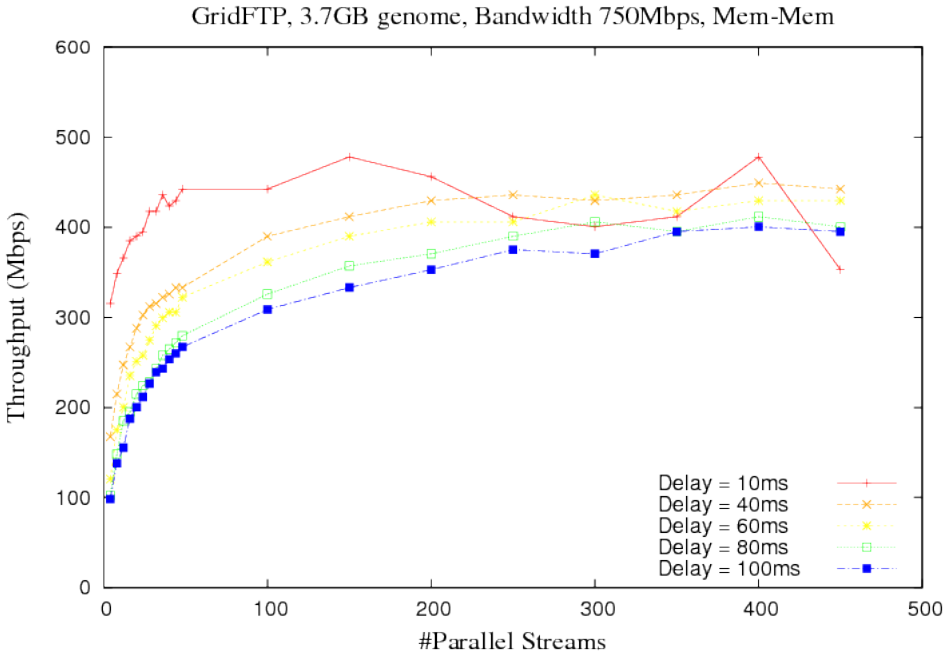

Figure 3.16: GridFTP memory-to-memory throughput at different RTTs with varying parallelism, 750Mbps bandwidth, 0 packet loss, recommended TCP settings and htcp TCP congestion control.

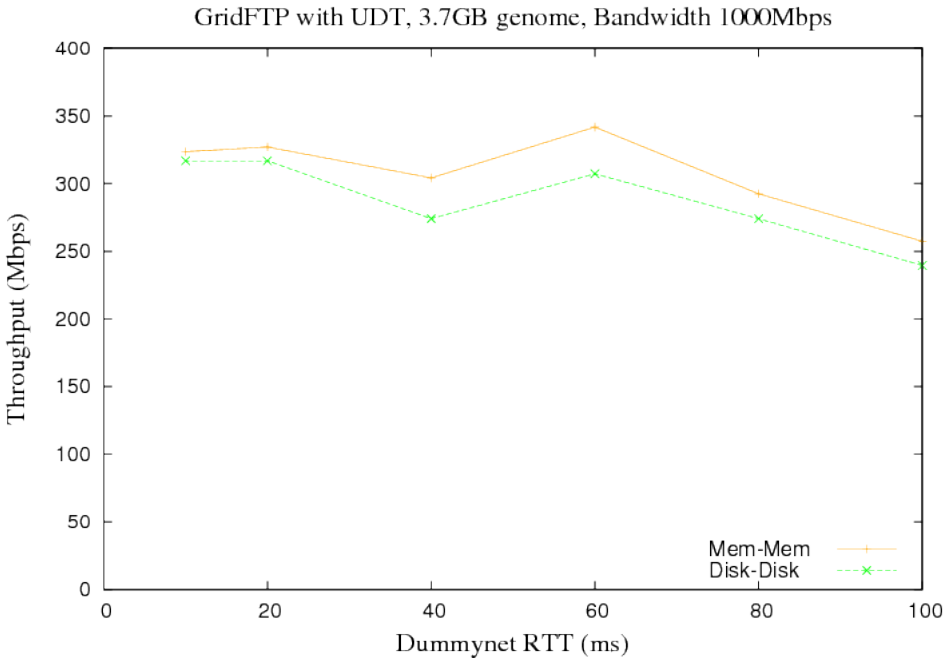

Figure 3.17: UDT memory-to-memory throughput at different RTTs with a single stream, 1Gbps bandwidth, 0 packet loss.

4. Conclusion

Given the results from the subset of the full-factorial experiment, we recommend GridFTP with TCP tuning as an open source data transfer application (of the three tools we examined). GridFTP consistently acheived higher throughput than the other applications tested and has a larger feature set, including the ability to use alternative transport layer protocols and proxying. In particular, we believe that further testing of GridFTP's UDT extension is necessary, and an analysis of UDP based protocols is necessary for comparison. Additional "real-world" experiments must be conducted with these three tools to verify the accuracy of our emulation (Sec. 4.1). Independently, we belive additional genome-specific compression techniques can further increase throughput (Sec. 4.2).

4.1 Validation

Further validation is needed to verify the accuracy of the Dummynet emulator results. As was done in [1], we will use the PlanetLab [13] research network to repeat several experiments conducted in Dummynet. Specifically, we plan on conducting a large data transfer (using the three data transfer tools) from a UCSC PlanetLab instance to an instance on the East Coast of the United States. PlanetLab will also enable us to test the proxy features of GridFTP. It is known that when using TCP one can achieve better performance by first sending the data to an intermediate proxy that is along the path from the sender to the receiver. In this case, each TCP connection will have a smaller RTT and packet loss will not affect the entire TCP session.

Genomic data distribution across the United States could use a hub-and-spoke topology with a dedicated proxy in between coast to coast senders and receivers. This approach is especially useful in cases where data must be sent from one data center to many, say $n-1$, data centers: the number of transfers is reduced from $O(n^2)$ to $O(n)$, albeit with the drawback of introducing a single-point of failure and a bottleneck.

4.2 Content-Specific Compression

An alternative approach to reducing end-to-end transmission duration is to reduce the amount of data transmitted using preprocessing such as genome-specific compression. Indeed, [16] details a novel reference method of compression for genomic data that can reduce a 2986.8MB FASTA file to under 18.8MB by first applying Huffman encoding and then storing the differences (using a modified version of the Unix diff tool) between a reference genome and the genome to be compressed. [6] provides a similar technique that also allows tunable lossy compression. The reference-based method could be applied for genomic data transfer by having clients first receive a large reference genome and then receive the differences for the subsequent genomes. To be sure, such a method is more CPU intensive then the current method of hosting gzipped files, however the CPU cost can be amortized across delivering the file to multiple clients (the compression is only computed once for each new genome).

For unannotated FASTA data the possible bandwidth savings using this type of compression are immense. Depending on data access characteristics, a single reference

data-set per organism could be efficiently distributed using BitTorrent (since the single reference is used for all other genomes belonging to the same organism), while the individual differences (deltas) would be distributed directly over HTTP. This type of distribution would be effective with repeated long tail access patterns, where the size of the deltas are quite small (megabytes) and an individual delta would be accessed infrequently.

As a quick prototype, we implemented a simpler version of reference-based encoding as a proof-of-concept. Specifically, instead of using diff as in [16], we used bsdiff [12], which uses Larsson-Sadakane suffix sorting [10]. For this test, we used the KOREF_20090131 FASTA dataset as a reference, and the KOREF_20090224 was compressed [7]. We naively broke each chromosome into three equal sized parts and used bsdiff to calculate the deltas for each of the corresponding parts. We split the chromosomes in order to reduce the amount of time and memory bsdiff uses when calculating the deltas. Note that [16] uses a more sophisticated method of minimization to split the chromosome into parts. Table 4.2 lists the results of our prototype implementation in comparison to the results of [16]. We managed to compress the 2.9GB file to 11M, in comparison to 18.8M. It took 3 hours 49 minutes to bsdiff the entire genome, and 7 minutes 23 seconds to decompress on a modest 2.2Ghz AMD 64 computer with 2GB of RAM. The compression was done serially, however since we split each chromosome into 3 parts theoretically we could run bsdiff across 72 hosts in parallel, which would drastically reduce the compression time.

Unfortunately, from our simple experiments of diff and bsdiff, we believe annotations (BAM files) destroy delta redundancy. As a work around, we propose separating the annotations (metadata) from the non-annotations and compressing each individually (using two different compression algorithms). It is not immediately apparent whether reference based compression is effective on metadata. A package containing both of these files would be sent and the receiver would uncompress the two files and rebuild the original file, along with an already received reference genome as input.

More research is needed to explore the benefits of referenced based and genome-specific compression with respect to network transfer. Specifically, we need to investigate genomic data access patterns and research ways of applying genome-specific compression to both the raw genomic FASTA data and the respective BAM metadata separately. We believe attempting to compress the combined metadata and raw genomic data together overlooks their unique types of redundancy, especially when using difference or delta-based compression.

4.3 Acknowledgements

The authors would like to thank Katia Obraczka, Brad Smith and J.J. Garcia-Luna-Aceves for their guidance and direction on this project. Additionally, we would like to thank Erich Weiler and Chris Wilks for bringing us up to speed on the project details and configuring the experiments. We thank Subhas Desa and Jim Warner for their conversations on network experimental design.

File	Raw size	bsdiff	Modified diff
chromosome_1.fa	240M	752K	1.3M
chromosome_2.fa	236M	756K	1.3M
chromosome_3.fa	194M	540K	987K
chromosome_4.fa	186M	616K	1.1M
chromosome_5.fa	176M	552K	965K
chromosome_6.fa	166M	488K	885K
chromosome_7.fa	154M	632K	1.0M
chromosome_8.fa	142M	420K	746K
chromosome_9.fa	137M	508K	844K
chromosome_10.fa	132M	428K	750K
chromosome_11.fa	131M	424K	738K
chromosome_12.fa	129M	384K	686K
chromosome_13.fa	111M	284K	508K
chromosome_14.fa	104M	264K	473K
chromosome_15.fa	98M	284K	485K
chromosome_16.fa	87M	332K	555K
chromosome_17.fa	77M	296K	494K
chromosome_18.fa	74M	232K	399K
chromosome_19.fa	62M	224K	390K
chromosome_20.fa	61M	156K	276K
chromosome_21.fa	46M	136K	221K
chromosome_22.fa	49M	156K	256K
chromosome_M.fa	20K	12K	183B
chromosome_X.fa	151M	1.8M	578.9K
2.9G	total	11M	18.8M

Table 4.1: bsdiff based compression versus modified diff

Bibliography

[1] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, and S. Tuecke. Gridftp: Protocol extensions to ftp for the grid. *GWD-R*, page 3, 2001.

[2] W. Allcock, J. Bresnahan, R. Kettimuthu, and J. Link. The globus extensible input/output system (xio): A protocol independent io system for the grid. In *Protocol-Independent I/O System for the Grid. Joint Workshop on HighPerformance Grid Computing and High-Level Parallel Programming Models in conjunction with International Parallel and Distributed Processing Symposium*, 2005.

[3] Bio-Mirror, 2011.

[4] J. Bresnahan, M., G. Khanna, Z. Imani, R. Kettimuthu, and I. Foster. Globus gridftp: what's new in 2007. In *Proceedings of the first international conference on Networks for grid applications*, pages 19:1–19:5, 2007.

[5] ESnet, 2011.

[6] M. Hsi-Yang Fritz, R. Leinonen, G. Cochrane, and E. Birney. Efficient storage of high throughput dna sequencing data using reference-based compression. *Genome Research*, 2011.

[7] Korean Reference Genome, 2011.

[8] Y. Gu and R. L. Grossman. Udt: Udp-based data transfer for high-speed wide area networks. *Computer Networks*, pages 1777 – 1799, 2007.

[9] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 2008.

[10] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258 – 272, 2007. The Burrows-Wheeler Transform.

[11] Nginx, 2011.

[12] Colin Percival, 2003.

[13] PlanetLab, 2011.

[14] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, pages 31–41, 1997.

[15] SpeedGuide.net, 2011.

[16] C. Wang and D. Zhang. A novel compression tool for efficient storage of genome resequencing data. *Nucleic Acids Research*, 2011.