Harnessing Multicore Processors for High-Speed Secure Transfer

John Bresnahan,[1,2,3] Rajkumar Kettimuthu,[1,2] Mike Link,[1,2] Ian Foster[1,2,3]

[1]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439
[2]Computation Institute, University of Chicago, Chicago, IL 60637
[3]Department of Computer Science, University of Chicago, Chicago, IL 60637

{bresnaha, kettimut, mlink, foster}@mcs.anl.gov

Abstract—A growing need for ultra-high-speed data transfers has motivated continued improvements in the transmission speeds of the physical network layer. As researchers develop protocols and software to operate over such networks, they often fail to account for security. The processing power required to encrypt or sign packets of data can significantly decrease transfer rates, and thus security is often sacrificed for throughput. Emerging multicore processors provide a higher ratio of CPUs to network interfaces and can, in principle, be used to accelerate encrypted transfers by applying multiple processing and network resources to a single transfer. We discuss the attributes that network protocols and software must have to exploit such systems. In particular, we study how these attributes may be applied in the GridFTP code distributed with the Globus Toolkit. GridFTP is a well-accepted and robust protocol for high-speed data transfer. It has been shown to scale to near-network speeds. While GridFTP can provide encrypted and protected data transfers, it historically suffers transfer performance penalties when these features are enabled. We present configurations to the Globus GridFTP server that can achieve fully encrypted high-speed data transfers.

***Index Terms*—Secure data transfer, GridFTP, Encryption, Parallel streams**

I. Introduction

Multicore CPUs are an emerging technology that promises to provide a high level of parallel processing within a single computer system. Along with the growth in CPU cores, the transmission speeds of the physical network hardware have significantly improved. Harnessing these two hardware improvements to accommodate the growing need for high-speed data transfers presents a significant challenge. Data transfers are typically treated as serial streams, while multicore processing can be leveraged only in parallel. In order to take advantage of these enhancements, applications must be identified and threaded programming models created that can properly scale to parallel CPUs while still interacting with the network properly.

Protocols such as GridFTP [1] have proven to be able to reliably transmit data at network speeds. However, achieving these speeds requires the user to ignore important aspects of security, such as data channel encryption and integrity protection. The processing power required for these security features adds a significant amount of latency to each packet and thus slow transfer rates significantly. In this paper we present techniques to harness the power of a multicore CPU to do high-speed, fully encrypted data transfers.

The remainder of the paper is organized as follows. In Section II we present related work. Section III explains the problem of parallel processing using modern security protocols. Section IV presents the programming model used in the Globus Toolkit and explains why this is a successful approach for utilizing multicore CPUs in an I/O system. Section V presents a study showing the results of scaling up the number of processors in a fully encrypted transfer to achieve network speeds.

II. Related Work

Hardware accelerators have been used to address the Secure Socket Layer (SSL) performance problems. An accelerator is a card that plugs into PCI slot or SCSI port and contains a co-processor that performs part of SSL processing. Network interface cards with offloaded SSL and IPSec [2] have also been produced. We want to achieve high-speed secure transfers with general-purpose hardware so that it can be used more commonly. We expect that multicore processors will become more common than SSL/IPSec offload engines. Further, we would like to utilize for high-speed secure data transfers the parallel and higher processing power that the multicore technology promises to achieve. Offload techniques do not help in achieving processing parallelism for secure processing on a single node.

III. Parallel Streams

To achieve processing parallelism on safe or private transfers, we must ensure that secure processing is performed on different portions of the data stream at the same time. Further, this simultaneous processing must be performed on different CPUs. While simple in principle, this concept presents interesting problems for network protocols and software implementations. Encryption protocols that use cipher block chaining, such as TLS/SSL [3], require that data be decrypted in the same order that it was encrypted. Further, the way that bytes in a stream are processed varies with their position in the stream. Thus, not only does the value of the

byte being processed matter, but also when it was processed.

These issues introduce difficulties when breaking up the streams for parallelization. For the reasons described above, we cannot take portions of a single data stream and process them in parallel against the same security context. To properly follow secure protocols, we cannot process any one byte until the previous byte has been processed. Thus there can be no parallelism against a single security context.

We can solve this problem by creating many distinct security contexts for a single data transfer. A simple way to realize this approach with existing network protocols is by using parallel streams. Parallel streams are common in data transfers as a means of achieving network optimization [4,5]. In order to minimize penalties associated with TCP slow start and dropped packets, many TCP streams are used for the same logical transfer, thus reducing the penalties associated with any one packet loss. This technique can also be leveraged for use in parallel encryption. Each stream has its own security context and is independent with regard to security processing.

IV. Asynchronous Event Model

Solving this problem in software requires some type of threaded I/O model. In order to get many parallel data streams processing at once, multiple threads of execution must be occurring on different CPUs. This type of parallelism can happen via threads in a single user process or by making use of multiple processes. The Globus Toolkit achieves such parallelism via an asynchronous event model and thread pools [6]. We present here the advantages of the asynchronous thread pool model.

In an asynchronous event model, the software developer posts I/O requests to the system. When the request is fulfilled (or an error occurs), the user is notified via a *callback* function defined in the developer's own process space. Between the time of posting the request and the time the application receives the callback, the developer's code is free to do whatever it likes, including posting more I/O requests. The I/O subsystem services the developer's request by creating background threads. These threads handle the framing of I/O and interact with the operating system to send and receive messages without interfering with the developer's code.

The Globus Toolkit's I/O library, Globus XIO [7], is a protocol abstraction layer that allows developers to ignore the specific protocol in use and just post requests for read and write. Part of servicing secure protocol requests involves signing or encrypting the posted data. In Globus XIO, this task is performed as part of the event processing and is therefore handled by one of the system's background threads. If many events are posted at once, as is the case for parallel TCP protocols, many threads can be processing the posted events in parallel.

As part of this work we set the default number of background threads used by Globus XIO (which can be set by the system administrator to any value) to the CPU count plus one. Because of this asynchronous thread pool model, the developer at no point needs to be aware of the number of background threads. It is strictly a site optimization parameter.

The Globus GridFTP server is built on the asynchronous event model above described. The GridFTP protocol uses parallel streams on the data channel. Because of these two facts we can achieve parallel secure processing. When the number of parallel streams is greater than or equal to the number of system CPUs, optimal processing parallelism occurs. As previously stated, there is no mapping of data stream to CPU. The Globus Toolkit does not dedicate a thread to a parallel stream. Instead, events are posted to the I/O subsystem, and many threads handle the events. In this way the optimal number of threads is not tightly coupled to the optimal number of streams.

V. Experiments

To measure the effectiveness of our approach, we ran several experiments across many different multicore and multi-CPU machines of various architectures. Our goal is to show that given enough CPUs in a single system, the Globus GridFTP server can perform fully encrypted (private) transfers at network speeds. Unfortunately, in some cases the number of CPUs on a single system is inadequate to match the processing requirements of secure transfers. In these cases we can still achieve high-throughput secure data transfers by tying many computers together in a striped transfer. Striped transfers allow portions of the data to come from different network endpoints. Thus, we can use CPUs in many machines for a single data transfer and therefore scale up the processing power in way similar to that of multicore CPUs.

With each stripe we add not only additional CPU resources but also additional memory and network resources. Because these additional resources are not added as more cores are added to a chip, this situation is not ideal. However, it does provide a test bed for finding the approximate optimal number of processors. Because transfer speeds are a function of the bandwidth delay product [8], additional memory should not influence the speed of the transfer. Upcoming multicore technology promises to dramatically increase the number of CPUs in a single system. By using many stripes in this way, we can predict the effectiveness of these future systems with some degree of accuracy.

A. Optimal Parameters on Different Architectures

Tables 1–3 show the results of experiments in which we use the asynchronous event model to take advantage of multiple CPUs. We ran two GridFTP servers on identical machines connected with Gigabit Ethernet on a LAN. The tables show how the use of one or two streams (P1 and P2, respectively) affects transfer rates in a single threaded system versus an asynchronous thread pool system with two threads. With one thread and one stream, only one CPU is working at a time. With one thread and two streams and with two threads and one stream, we again see just one CPU working at a time. We do not achieve concurrency until we have at least two threads and at least two streams.

Four levels of security are tested in the experiments. The row labeled *clear* has no data channel security at all; thus it establishes an upper bound for the other experiments. *Authenticated* data channel security has minimal security

processing; since all processing is done at the beginning of the transfer, little latency is added, and the transfer rates are about equal to that of *clear*. *Integrity* requires security processing on each packet, as does *privacy*; however privacy requires substantially more processing.

Table 1: Pentium 1.1 GHz Dual Core

Security Level	Single P1	Single P2	Pool P1	Pool P2
Clear	814	818	816	818
Authenticated	812	814	813	816
Safe	169	178	164	285
Private	76	78	75	138

Table 2: Optron 64-bit 2.4 GHz Dual Core

Security Level	Single P1	Single P2	Pool P1	Pool P2
Clear	897	897	897	897
Authenticated	897	897	897	897
Safe	254	268	254	471
Private	100	101	100	196

Table 3: Itanium 1.2 Dual Processor

Security Level	Single P1	Single P2	Pool P1	Pool P2
Clear	903	905	903	906
Authenticated	899	899	899	899
Safe	488	517	488	770
Private	177	183	177	340

While two processors are not enough to perform safe or private transfers at network speed, the results do show that transfer speeds are significantly increased by processor concurrency. The transfer rate almost doubles with the addition of the second CPU. This result implies that, given enough cores, GridFTP encrypted transfers can scale to network speeds.

B. Decoupling Threads and Parallel Streams

While the number of streams in a transfer does affect the CPU processing parallelism, it is also a network protocol optimization parameter. It would not be ideal to have a tight coupling of parallel data stream to the number of cores. In particular, we do not want a situation where the number of streams is optimal for the network but suboptimal for CPU parallelism nor the other way around.

To gain insight into this situation, we ran the full cross-product of transfer with thread counts ranging from one to sixty-four and parallel streams ranging from one to sixty-four. We ran the experiments on dual-core machines for both safe and private transfers.

The results (Fig. 1) show that one thread per CPU and one parallel stream per CPU provide the best performance. As the number of threads increases, the performance for all levels of parallel streams slowly decreases. This result shows that without the benefit of a corresponding CPU, an additional thread can have detrimental effects. Our results also show that the relative differences for most levels of parallelism are not affected by the thread count. This situation suggests that once the minimal threshold for each parameter is exceeded; there is little correlation between the two optimization parameters.

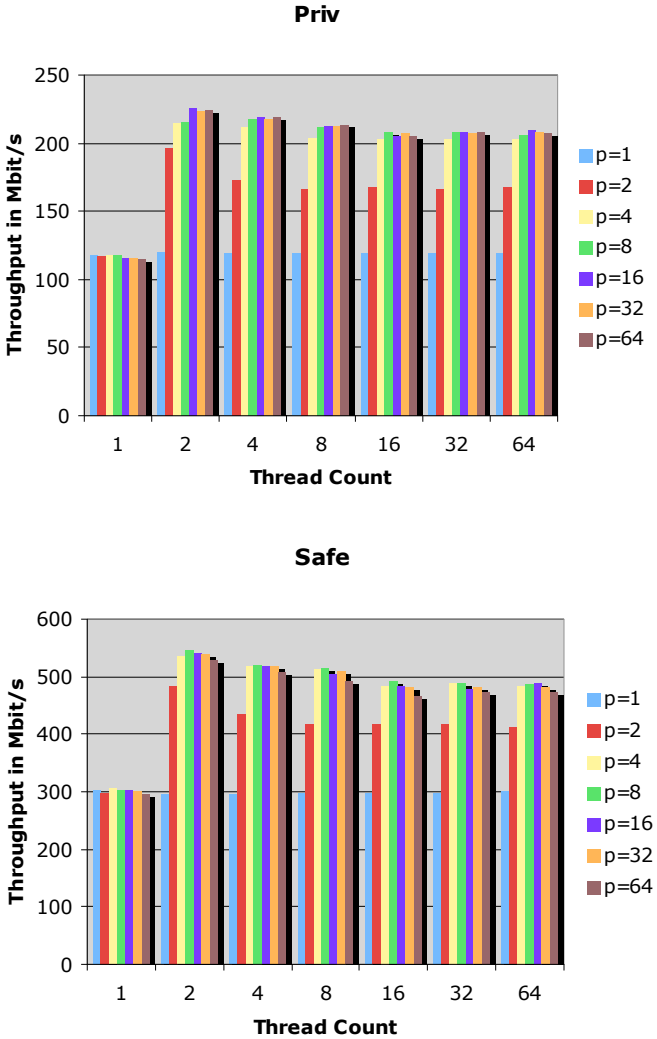

Fig. 1: Effect of parallel streams: private transfers (*top*) and safe transfers (*bottom*)

C. Striping

The initial results suggest that a processor with more than four cores is needed to sustain one gigabit per second of a private data transfer on general-purpose hardware. Because we did not have access to such a machine, the environment was simulated by using the striping feature of the Globus GridFTP server. We performed private and safe transfers using the University of Chicago TeraGrid [9]. Each server was run on a dual 64-bit Itanium 1.2 GHz processor with 4 MB of RAM. Four stripes were used on both the sender and the receiver. Since each node had two CPUs, there were eight CPUs in all. Two types of configurations were used in each iteration of the experiment, one where each stripe had one thread and one where each had two threads. Because we were simulating a case where a secure transfer has enough processing power to move as fast as an insecure transfer, we ran clear and

authenticated data single CPU transfers to establish the target throughput. The results are shown in Fig. 2.

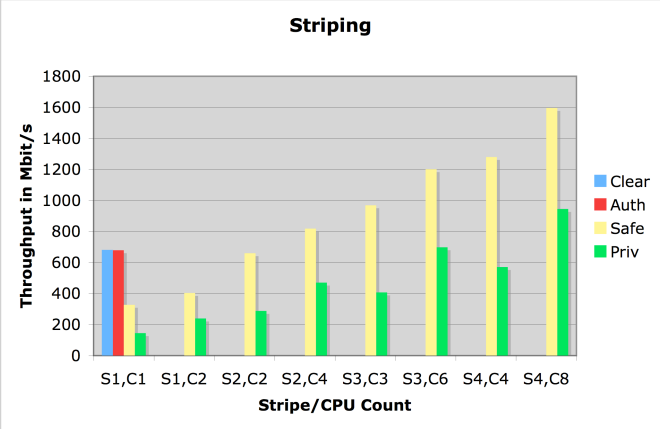

Fig. 2. Effect of striping on transfer rates

As expected, the results show that as we add stripes, the transfer rates increase. If, however, one looks only at the CPU count, the performance does not strictly increase. The results show how stripes play a role. Because private data transfers are more CPU intensive than safe data transfer, the effect of the superfluous resources that come with an additional stripe is less pronounced on the private data session. In a safe data transfer the additional network resources are more available since the processing latency is lower. Further, there is a benefit to having an extra system CPU even when the I/O subsystem has a single thread. Kernel-level services and the TCP software stack use it on the GridFTP application's behalf.

VI. Summary

In this work we examined the trend in CPU hardware from a data transfer point of view. We presented a programming model and network protocol architecture that allows many cores to be harnessed together to participate in a single, coordinated data transfer. The additional processing power has allowed us to solve a long-standing problem of having to choose between insecure data channels or slow transfers. Our results were gathered using the Globus GridFTP server.

From our results we see that private data transfer require six CPUs to achieve network speeds and that safe transfers require between two and four CPUs depending on the effects of striping. Commodity general-purpose computer systems with eight-core CPUs will be commonplace in the near future, and the software solution that we have presented here is immediately available.

Acknowledgments

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357 and in part by SciDAC-2 CEDPS.

References

[1] W. Allcock, GridFTP: Protocol Extensions to FTP for the Grid, Global Grid ForumGFD-R-P.020, 2003.

[2] S. Kent and R. Atkinson, Security Architecture for the Internet Protocol, IETF RFC 2401, 1998.

[3] T. Dierks and E. Rescorla, The Transport Layer Security (TLS) Protocol Version 1.1, IETF RFC 4346, 2006.

[4] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, The Globus Striped GridFTP Framework and Server, SC'05, ACM Press, 2005.

[5] T. J. Hacker, B. D. Noble, and B. D. Athey, Improving Throughput and Maintaining Fairness Using Parallel TCP. IEEE InfoCom, 2004.

[6] http://www.globus.org/toolkit/docs/3.2/developer/globus-async.html.

[7] W. Allcock, J. Bresnahan, R. Kettimuthu,. and J. Link, J., The Globus eXtensible Input/Output System (XIO): A Protocol-Independent I/O System for the Grid. Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models, 2005.

[8] T. Lakshman and U. Madhow. The Performance of TCP/IP for Networks with High Bandwidth-Delay Products and Random Loss. IEEE/ACM Transactions on Networking, pages 336–350, June 1997.

[9] http://www.teragrid.org/.