Impact of High Performance Sockets on Data Intensive Applications [*]

Pavan Balaji[†], Jiesheng Wu[†], Tahsin Kurc[‡],
Umit Catalyurek[‡], Dhabaleswar K. Panda[†], Joel Saltz[‡]

[†] Dept. of Computer and Information Science
The Ohio State University, Columbus, OH, 43210
{balaji,wuj,panda}@cis.ohio-state.edu

[‡] Dept. of Biomedical Informatics
The Ohio State University, Columbus, OH, 43210
{kurc.1,catalyurek.1,saltz.3}@osu.edu

Abstract

The challenging issues in supporting data intensive applications on clusters include efficient movement of large volumes of data between processor memories and efficient coordination of data movement and processing by a runtime support to achieve high performance. Such applications have several requirements such as guarantees in performance, scalability with these guarantees and adaptability to heterogeneous environments. With the advent of user-level protocols like the Virtual Interface Architecture (VIA) and the modern InfiniBand Architecture, the latency and bandwidth experienced by applications has approached to that of the physical network on clusters. In order to enable applications written on top of TCP/IP to take advantage of the high performance of these user-level protocols, researchers have come up with a number of techniques including User-Level Sockets Layers over high performance protocols. In this paper, we study the performance and limitations of such a substrate, referred to here as SocketVIA*, using a component framework designed to provide runtime support for data intensive applications. The experimental results show that by reorganizing certain components of an application (in our case, the partitioning of a dataset into smaller data chunks), we can make significant improvements in application performance. This leads to a higher scalability of applications with performance guarantees. It also allows fine grained load balancing, hence making applications more adaptable to heterogeneity in resource availability. The experimental results also show that the different performance characteristics of SocketVIA allow a more efficient partitioning of data at the source nodes, thus improving the performance of the application up to an order of magnitude in some cases.*

Keywords: VIA, High-Performance Networking, Sockets, Data Intensive Computing, PC Clusters

[*]This research was supported by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #ACI-0203846, #ACI-0130437, #ACI-9982087, #EIA-9986052 and #CCR-0204429, Lawrence Livermore National Laboratory under Grant #B500288 and #B517095 (UC Subcontract #10184497), Ohio Board of Regents BRTTC #BRTT02-0003, and Department of Energy's Grant #DE-FC02-01ER25506.

1 Introduction

Quite a number of research projects in high-end computing focus on development of methods for solving challenging compute intensive applications in science, engineering and medicine. These applications are generally run in batch mode and can generate very large datasets. Advanced sensor technologies also enable acquisition of high resolution multi-dimensional datasets. As a result, there is an increasing interest in developing applications that interactively explore, synthesize and analyze large scientific datasets [15]. In this paper, we refer to these applications as data intensive applications.

Being built from commodity hardware, PC clusters are becoming cost-effective and viable alternatives to mainstream supercomputers for a broad range of applications, including data intensive applications. A challenging issue in supporting data intensive applications on these platforms is that large volumes of data should be efficiently moved between processor memories. Data movement and processing operations should also be efficiently coordinated by a runtime support to achieve high performance. Together with a requirement in terms of good performance, such applications also require guarantees in performance, scalability with these guarantees, and adaptability to heterogeneous environments and varying resource availability.

Component-based frameworks [6, 10, 17, 20] have been able to provide a flexible and efficient environment for data intensive applications on distributed platforms. In these frameworks, an application is developed from a set of interacting software components. Placement of components onto computational resources represents an important degree of flexibility in optimizing application performance. Data-parallelism can be achieved by executing multiple copies of a component across a cluster of storage and processing nodes [6]. Pipelining is another possible mechanism for performance improvement. In many data intensive applications, a dataset can be partitioned into *user-defined data chunks*. Processing of the chunks can be pipelined. While computation and communication can be overlapped in this manner, the performance gain also depends on the granularity of computation and the size of data messages (data chunks). Small chunks would likely result in better load balance and pipelining, but a lot of messages are generated with small chunk sizes. Although large chunks would reduce the number of messages and achieve higher communication bandwidth, they would likely suffer from load imbalance and less pipelining.

With the advent of modern high speed interconnects such as GigaNet [11], Myrinet [7], Gigabit Ethernet [14], InfiniBand Architecture [2] and the Quadrics Network [19], the bottleneck in communication has shifted to the messaging software. This bottleneck has been attacked by researchers, leading to the development of low-latency and high-bandwidth user-level protocols [12, 13, 18]. Along with these research efforts, several industries have taken up the initiative to standardize high-performance user-level protocols such as the Virtual Interface Architecture (VIA) [8, 12].

A number of applications have been developed on kernel-based protocols such as TCP/UDP using the sockets interface. To support such applications on high performance user-level protocols without any changes to the application itself, researchers have come up with a number of techniques. These techniques include user-level sockets layers over high performance protocols [3, 16, 21]. Applications written using kernel-based sockets layers are often developed keeping the communication performance of TCP/IP in mind. High performance substrates, on the other hand, have different performance characteristics compared to kernel-based sockets layers. This becomes a fundamental bottleneck in the performance such high performance substrates are able to deliver. However, changing some components of an application, such as the size of the data chunks that make up the dataset, allows the applications to take advantage of the performance characteristics of high performance substrates making them more scalable and adaptable.

In this paper, we study the efficiency and limitations of such a substrate, referred to here as SocketVIA, in terms of performance and the flexibility it allows, in the context of a component framework designed to provide runtime support for data intensive applications, called DataCutter [6]. In particular, we investigate answers to the following questions:

- *Can a high performance substrate allow the implementation of a scalable interactive data-intensive application with performance guarantees to the end user?*
- *Can a high performance substrate improve the adaptability of data-intensive applications to heterogeneous environments?*

Our experimental results show that by reorganizing certain components of the applications, significant improvements in performance can be obtained. This leads to higher scalability of applications with performance guarantees. It also enables fine grained load balancing, thus making applications more adaptable to heterogeneous environments and varying resource availability.

The rest of the paper is organized as follows. In Section 2, we give an overview of data intensive applications. In Section 3, we talk about the performance issues in designing runtime support for data intensive applications. The software infrastructure used in the experiments is described in Section 4. We present some experimental results in Section 5, and conclude the paper in Section 6.

2 Overiew of Data Intensive Applications

As processing power and capacity of disks continue to increase, the potential for applications to create and store multi-gigabyte and multi-terabyte datasets is becoming more feasible. Increased understanding is achieved through running analysis and visualization codes on the stored data. For example, interactive visualization relies on our ability to gain insight from looking at a complex system. Thus, both data analysis and visual exploration of large datasets play an increasingly important role in many domains of scientific research. We refer here to applications that interactively query and analyze large scientific datasets as data-intensive applications.

An example of data-intensive applications is digitized microscopy. We use the salient characteristics of this application as a motivating scenario and a case study in this paper. The software support required to store, retrieve, and process digitized slides to provide interactive response times for the standard behavior of a physical microscope is a challenging issue [1, 9]. The main difficulty stems from handling of large volumes of image data, which can range from a few hundreds of Megabytes to several Gigabytes per image. At a basic level, the software system should emulate the use of a physical microscope, including continuously moving the stage and changing magnification. The processing of client queries requires projecting high resolution data onto a grid of suitable resolution and appropriately composing pixels mapping onto a single grid point.

Consider a visualization server for digitized microscopy. The client to this server can generate a number of different types of requests. The most common ones are *complete update queries*, by which a completely new image is requested, and *partial update query*, by which the image being viewed is moved slightly or zoomed into. The server should be designed to handle both types of queries.

Processing of data in applications that query and manipulate scientific datasets can often be represented as an acyclic, coarse grain data flow, from one or more data sources (e.g., one or more datasets distributed across storage systems) to processing nodes to the client. For a given query, first the data of interest is retrieved from the corresponding datasets. The data is then processed via a sequence of operations on the processing nodes. For example, in the digitized microscopy application, the data of interest is processed through *Clipping*, *Subsampling*, *Viewing* operations [5, 6]. Finally, the processed data is sent to the client.

Data forming parts of the image are stored in the form of blocks or data chunks for indexing reasons, requiring the entire block to be fetched even when only a part of the block is required. Figure 1 shows a complete image being made up of a number of blocks. As seen in the figure, a partial update query (the rectangle with dotted lines in the figure) may require only part of a block. Therefore, the size and extent of a block affect the amount of unnecessary data retrieved and communicated for queries.

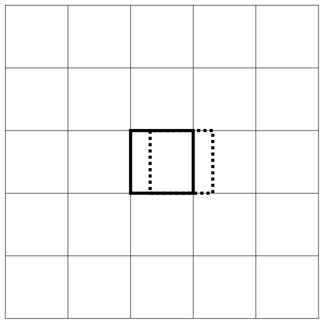


Figure 1. Partitioning of a complete image into blocks. A partial query (rectangle with dotted lines) requires only a part of a block.

3 Performance Issues in Runtime Support for Data Intensive Applications

3.1 Basic Performance Considerations

For data-intensive applications, performance can be improved in several ways. First, datasets can be declustered across the system to achieve parallelism in I/O when retrieving the data of interest for a query. With good declustering, a query will hit as many disks as possible. Second, the computational power of the system can be efficiently used if the application can be designed to exploit data parallelism for processing the data. Another factor that can improve the performance, especially in interactive exploration of datasets, is *pipelined* execution. By dividing the data into chunks and pipelining the processing of these chunks, the overall execution time of the application can be decreased. In many applications, pipelining also provides a mechanism to gradually create the output data product. In other words, the user does not have to wait for the processing of the query to be completed; partial results can be gradually generated. Although this may not actually reduce the overall response time, such a feature is very effective in an interactive setting, especially if the region of interest moves continuously.

3.2 Message Granularity vs. Performance Guarantee

The granularity of the work and the size of data chunks affects the performance of pipelined execution. The chunk size should be carefully selected by taking into account the network bandwidth and latency (the time taken for the transfer of a message including the protocol processing overheads at the sender and the receiver ends). As the chunk size increases, the number of messages required to transfer the data of interest decreases. In this case, bandwidth becomes more important than latency. However, with a bigger chunk size, processing time per chunk also increases. As a result, the system becomes less responsive, i.e., the frequency of partial/gradual updates decreases. On the other hand, if the chunk size is small, the number of messages increases. As a result, latency may become a dominant factor in the overall efficiency of the application. Also, smaller chunks can result in better load balance among the copies of application components, but communication overheads may offset the performance gain.

Having large blocks allows a better response time for a complete update query due to improved bandwidth. However, during a partial update query, this would result in more data being fetched and eventually being wasted. On the other hand, with small block size, a partial update query would not retrieve a lot of unnecessary data, but a complete update query would suffer due to reduced bandwidth.

In addition to providing a higher bandwidth and lower latency, high performance substrates have other interesting features as demonstrated in Figures 2(a) and 2(b). Figure 2(a) shows that high performance substrates achieve a required bandwidth at a much lower message size compared to kernel-based sockets layers such as TCP/IP. For instance, for attaining bandwidth ‘B’, kernel-based sockets need a message size of U1 bytes, whereas high performance substrates require a lower message size of U2 bytes. Using this information in Figure 2(b), we observe that high performance substrates result in lower message latency (from L1 to L2) at a message size of U1 bytes. We also observe that high performance substrates can use a message size of U2 bytes (from Figure 2(a)), hence further reducing the latency (from L2 to L3) and resulting in better performance.

3.3 Heterogeneity and Load Balancing

Heterogeneity arises in several situations. First, the hardware environment may consist of machines with different processing power and memory capacity. Second, the resources can be shared by other applications. As a result, the availability of resources such as CPU and memory can vary dynamically. In such cases, the application should be structured to accommodate the heterogeneous nature of the environment. The application should be optimized in its use of shared resources and be adaptive to the changes in the availability of the resources. This requires the application to employ adaptive mechanisms to balance the workload among processing nodes depending on the computation capabilities of each of them. A possible approach is to adaptively schedule data and application computations among processing nodes. The data can be broken up into chunks so as to allow pipelining of computation and communication. In addition, assignment of data chunks to processing units can be done using a demand-driven scheme (see Section 4.1) so that faster nodes can get more data to process. If a fast node becomes slower (e.g., due to pro-

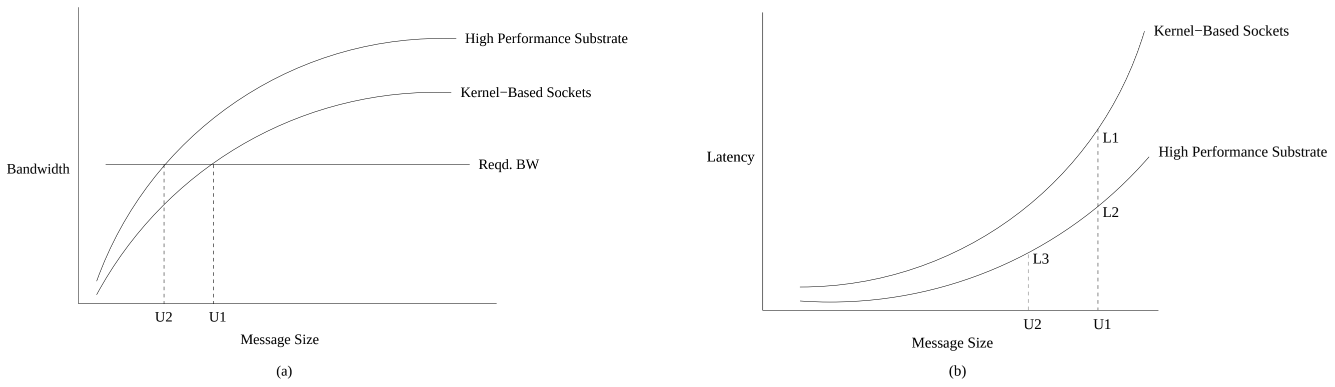


Figure 2. (a) High Performance Substrates achieve a given bandwidth for a lower message size compared to Kernel-Based Sockets, (b) High Performance Substrates can achieve a direct and indirect improvement in the performance based on the application characteristics

cesses of other applications), the underlying load balancing mechanism should be able to detect the change in resource availability quickly.

4 Software Infrastructure used for Evaluation

In terms of application development and runtime support, component-based frameworks [6, 10, 17, 20] can provide an effective environment to address performance issues in data intensive applications. Components can be placed onto different computational resources, and task and data-parallelism can be achieved by pipelined execution of multiple copies of these components. Therefore, we use a component-based infrastructure, called DataCutter [6], which is designed to support data intensive applications in distributed environments. We also employ a high performance sockets interface, referred to here as SocketVIA, designed for applications written using TCP/IP to take advantage of the performance capabilities of VIA.

4.1 DataCutter

In this section we briefly describe the DataCutter framework [6]. DataCutter implements a filter-stream programming model for developing data-intensive applications. In this model, the application processing structure is implemented as a set of components, referred to as *filters*, that exchange data through a *stream* abstraction. The interface for a *filter*, consists of three functions: (1) an initialization function (*init*), in which any required resources such as memory for data structures are allocated and initialized, (2) a processing function (*process*), in which user-defined operations are applied on data elements, and (3) a finalization function (*finalize*), in which the resources allocated in *init* are released.

Filters are connected via *logical streams*. A *stream* denotes a uni-directional data flow from one filter (i.e., the producer) to another (i.e., the consumer). A filter is required to read data from its input streams and write data to its output streams only. We define a *data buffer* as an array of data elements transferred from one filter to another. The original implementation of the logical stream delivers data in fixed size buffers, and uses TCP for point-to-point stream communication.

The overall processing structure of an application is realized by a *filter group*, which is a set of filters connected through logical streams. When a filter group is instantiated to process an application query, the runtime system establishes socket connections between filters placed on different hosts before starting the execution of the application query. Filters placed on the same host execute as separate threads. An application query is handled as a *unit of work* (UOW) by the filter group. An example is a visualization of a dataset from a viewing angle. The processing of a UOW can be done in a pipelined fashion; different filters can work on different data elements simultaneously. Processing of a UOW starts when the filtering service calls the filter **init** function, which is where any required resources such as memory can be pre-allocated. Next the **process** function is called to read from any input streams, work on the data buffers received, and write to any output streams. A special marker is sent by the runtime system after the last buffer to mark the end for the current UOW (see Figure 3(a)). The **finalize** function is called after all processing is completed for the current UOW, to allow release of allocated resources such as scratch space. The interface functions *may* be called again to process another UOW.

The programming model provides several abstractions to facilitate performance optimizations. A *transparent filter copy* is a copy of a filter in a filter group (see Figure 3(b)). The filter copy is transparent in the sense that it shares the same *logical* input and output streams of the original filter. A transparent copy of a filter can be made if the semantics of the filter group are not affected. That is, the output of a unit of work should be the same, regardless of the number of transparent copies. The transparent copies enable data-parallelism for execution of a single query, while multiple filter groups allow concurrency among multiple queries.

The filter runtime system maintains the illusion of a single logical point-to-point stream for communication be-

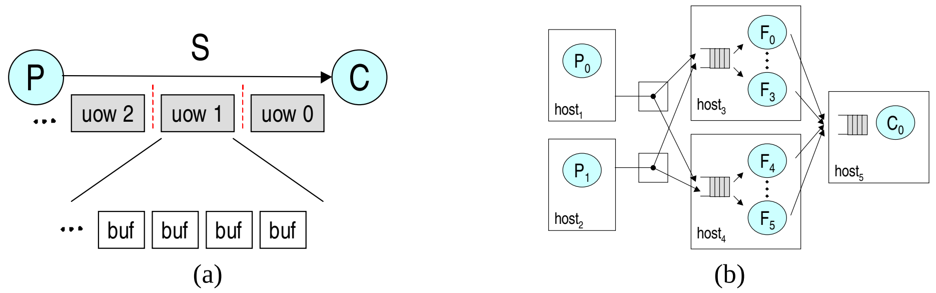


Figure 3. DataCutter stream abstraction and support for copies. (a) Data buffers and end-of-work markers on a stream. (b) P,F,C filter group instantiated using transparent copies.

tween a logical producer filter and a logical consumer filter. It is responsible for scheduling elements (or buffers) in a data stream among the transparent copies of a filter. For example, in Figure 3(b), if copy P_1 issues a buffer write operation to the logical stream that connects filter P to filter F, the buffer can be sent to the copies on $host_3$ or $host_4$. For distribution between transparent copies, the runtime system supports a Round-Robin (RR) mechanism and a Demand Driven (DD) mechanism based on the buffer consumption rate. DD aims at sending buffers to the filter that would process them fastest. When a consumer filter starts processing of a buffer received from a producer filter, it sends an acknowledgment message to the producer filter to indicate that the buffer is being processed. A producer filter chooses the consumer filter with the minimum number of unacknowledged buffers to send a data buffer to, thus achieving a better balancing of the load.

4.2 SocketVIA

Inspite of the development of low-latency and high-bandwidth user-level protocols, a large number of applications have been developed previously on kernel-based protocols such as TCP and UDP. Some of these applications took years to develop. Trying to rewrite these applications on user-level protocols is highly time-consuming and impractical. On the other hand, the sockets interface is widely used by a variety of applications written on protocols such as TCP and UDP.

The cLAN network is a hardware implementation of the Virtual Interface Architecture (VIA). There are two typical socket implementations on the cLAN network. One is to keep the legacy socket, TCP/UDP and IP layers unchanged, while one additional layer is introduced to bridge the IP layer and the kernel level VI layer. The LANE (LAN Emulator) implementation of the socket layer is such an implementation using an IP-to-VI layer [12]. Due to the system call overhead (including the kernel-context switch, flushing of the cache, flushing of the TLB, bottom-half handlers, etc) and multiple copies involved in this implementation, applications using LANE have not been able to take complete advantage of the high performance provided by the underlying network. Another type of socket implementation on the cLAN network is to provide socket interface using a user-level library based on the user-level VIA primitives. Our implementation falls into this category. We refer to our sockets layer as *SocketVIA* in the rest of this paper. Since the implementation of SocketVIA is not the main focus of the paper, we just present the micro-benchmark results for our sockets layer in the next section. For other details related to the design and implementation of SocketVIA, we refer the reader to [4].

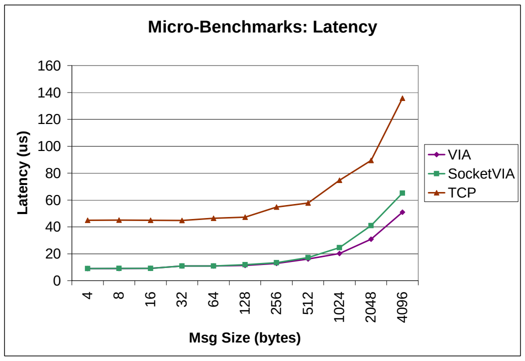
5 Performance Evaluation

In this paper, we present two groups of results. First, we look at the peak performance delivered by SocketVIA in the form of latency and bandwidth micro-benchmarks. Second, we examine the direct and indirect impacts on the performance delivered by the substrate on applications implemented using DataCutter in order to evaluate both latency and bandwidth aspects in a controlled way. The experiments were carried out on a PC cluster which consists of 16 Dell Precision 420 nodes connected by GigaNet cLAN and Fast Ethernet. We use cLAN 1000 Host Adapters and cLAN5300 Cluster switches. Each node has two 1GHz Pentium III processors, built around the Intel 840 chipset, which has four 32-bit 33-MHz PCI slots. These nodes are equipped with 512MB of SDRAM and 256K L2-level cache. The Linux kernel version is 2.2.17.

5.1 Micro-Benchmarks

Figure 4(a) shows the latency achieved by our substrate compared to that achieved by the traditional implementation of sockets on top of TCP and a direct VIA implementation (base VIA). Our sockets layer gives a latency of as low as 9.5μs, which is very close to that given by VIA. Also, it is nearly a factor of five improvement over the latency given by the traditional sockets layer over TCP/IP.

Figure 4(b) shows the bandwidth achieved by our substrate compared to that of the traditional sockets implementation and base cLAN VIA implementation. SocketVIA achieves a peak bandwidth of 763Mbps compared to 795Mbps given by VIA and 510Mbps given by the traditional TCP implementation; an improvement of nearly 50%.



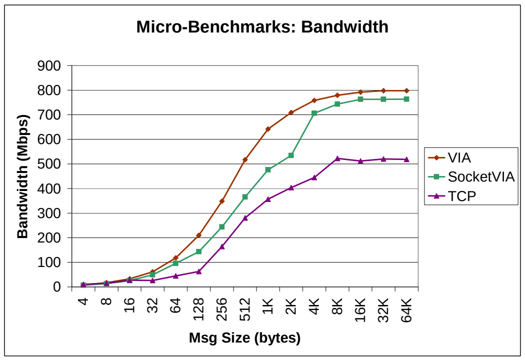


Figure 4. Micro-Benchmarks (a) Latency, (b) Bandwidth

5.2 Application Performance and Behavior

5.2.1 Experimental Setup

In these experiments, we used two kinds of applications. The first application emulates a visualization server. This application uses a 4-stage pipeline with a visualization filter at the last stage. Also, we executed three copies of each filter in the pipeline to improve the end bandwidth (Figure 5). The user visualizes an image at the visualization node, on which the visualization filter is placed. The required data is fetched from a data repository and passed onto other filters, each of which is placed on a different node in the system, in the pipeline.

Each image viewed by the user requires 16MB of data to be retrieved and processed. This data is stored in the form of chunks with pre-defined size, referred to here as the distribution block size. For a typical distribution block size, a complete image is made up of several blocks (Figure 1). When the user asks for an update to an image (partial or complete), the corresponding chunks have to be fetched. Each chunk is retrieved as a whole, potentially resulting in some additional unnecessary data to be transferred over the network.

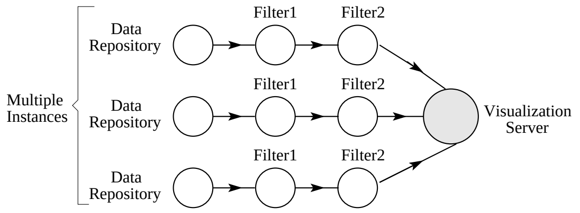


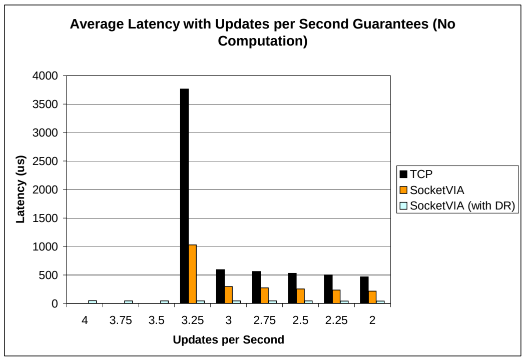
Figure 5. Guarantee Based Performance Evaluation: Experimental Setup

Two kinds of queries were emulated. The first query is a complete update or a request for a new image. This requires all the blocks corresponding to the query to be fetched. This kind of update is bandwidth sensitive and having a large block size would be helpful. Therefore, as discussed in the earlier sections, for allowing a certain update rate for the complete update queries, a certain block size (or larger) has to be used.

The second query is a partial update. This type of query is executed when the user moves the visualization window by a small amount, or tries to zoom into the currently viewed image. A partial update query requires only the excess blocks to be fetched, which is typically a small number compared to the number of blocks forming the complete image. This kind of update is latency sensitive. Also, the chunks are retrieved as a whole. Thus, having small blocks would be helpful.

In summary, if the block size is too large, the partial update will likely take long time, since the entire block is fetched even if a small portion of one block is required. However, if the block size is too small, the complete update will likely take long time, since many small blocks will need to be retrieved. Thus, for an application which allows both kinds of queries, there would be a performance tradeoff between the two types of queries. In the following experiments, we show the improved scalability of the application with socketVIA compared to that of TCP with performance guarantees for each kind of update. The results are presented in section 5.2.2.

The second application we look at is a software load-balancing mechanism such as the one used by DataCutter. When data is processed by a number of nodes, perfect pipelining is achieved when the time taken by the load-balancer to send one block of the message to the computing node is equal to the time taken by the computing node to process it. In this application, typically the block size is chosen so that perfect pipelining is achieved in computation and communication. However, the assumption is that the computation power of the nodes does not change during the course of the application run. In a heterogeneous, dynamic environment, this assumption does not hold. In our experiments, in a homogeneous setting, perfect pipelining is achieved at 16KB and 2KB for TCP/IP and VIA, respectively. This means that the block size required in TCP/IP is significantly larger than that in VIA. However, on heterogeneous networks, when a block size is too large, a mistake by a load balancer (sending the data block to a slow node) may become too costly (Figure 6). Performance impact with such heterogeneity is presented in section 5.2.3.



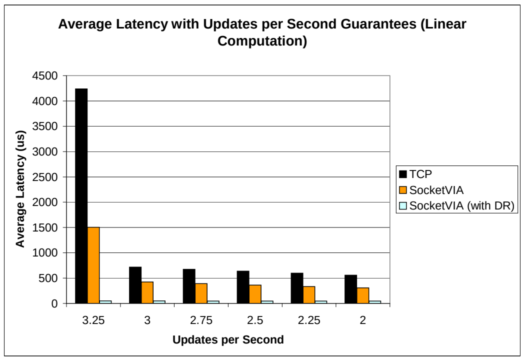


Figure 7. Effect of High Performance Sockets on Average Latency with guarantees on Updates per Second for (a) No Computation Cost and (b) Linear Computation Cost

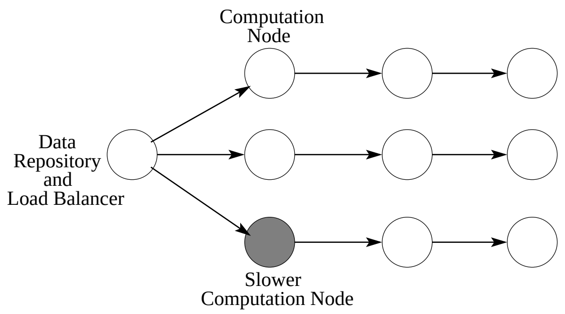


Figure 6. Effect of Heterogeneous Clusters: Experimental Setup

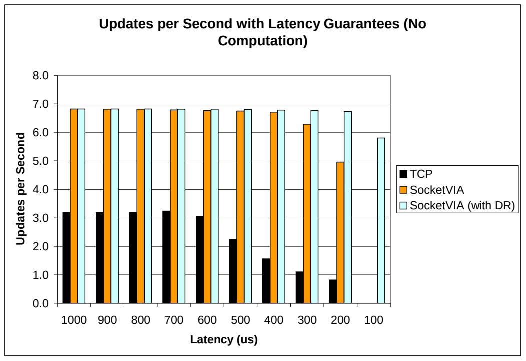
5.2.2 Guarantee based Performance Evaluation

Effect on Average Latency with guarantees on Updates per Second: In the first set of experiments, the user wants to achieve a certain frame rate (i.e., the number of new images generated or full updates done per second). With this constraint, we look at the average latency observed when a partial update query is submitted. Figures 7(a) and 7(b) show the performance achieved by the application. For a given frame rate for new images, TCP requires a certain message size to attain the required bandwidth. With data chunking done to suit this requirement, the latency for a partial update using TCP would be the latency for this message chunk (depicted as legend 'TCP'). With the same chunk size, SocketVIA inherently achieves a higher performance (legend 'SocketVIA'). However, SocketVIA requires a much smaller message size to attain the bandwidth for full updates. Thus, by repartitioning the data by taking SocketVIA's latency and bandwidth into consideration, the latency can be further reduced (legend 'SocketVIA (with DR)'). Figure 7(a) shows the performance with no computation. This experiment emphasizes the actual benefit obtained by using SocketVIA, without being affected by the presence of computation costs at each node. We observe, here, that TCP cannot meet an update constraint greater than 3.25 full updates per second. However, SocketVIA (with DR) can still achieve this frame rate without much degradation in the performance. The results obtained in this experiment show an improvement of more than 3.5 times without any repartitioning and more than 10 times with repartitioning of data. In addition to socketVIA's inherently improving the performance of the application, reorganizing some components of the application (the block size in this case) allows the application to gain significant benefits not only in performance, but also in scalability with performance guarantees.

Figure 7(b) depicts the performance with a computation cost that is linear with message size in the experiments. We timed the computation required in the visualization part of a digitized microscopy application, called Virtual Microscope [9], on DataCutter and found it to be 18ns per byte of the message. Applications such as these involving browsing of digitized microscopy slides have such low computation costs per pixel. These are the applications that will benefit most from low latency and high bandwidth substrates. So we have focused on such applications in this paper.

In this experiment, even SocketVIA (with DR) is not able to achieve an update rate greater than 3.25, unlike the previous experiment. The reason for this is that the bandwidth given by SocketVIA is bounded by the computation costs at each node. For this experiment, we observe an improvement of more than 4 and 12 times without and with repartitioning of data, respectively.

Effect on Updates per Second with Latency Guarantees: In the second set of experiments, we try to maximize the number of full updates per second when a particular latency is targeted for a partial update query. Figures 8(a) and 8(b) depict the performance achieved by the application. For a given latency constraint, TCP cannot have a block size greater than a certain value. With data chunking done to suit this requirement, the bandwidth it can achieve is quite limited as seen in the figure under legend 'TCP'. With the same block size, SocketVIA achieves a much better performance, shown by legend 'SocketVIA'. However, a re-chunking of data that takes the latency and bandwidth of SocketVIA into consideration results in a much higher performance, as shown by the performance numbers for 'SocketVIA (with DR)'. Figure 8(a) gives the performance with no computation, while computation cost, which varies linearly with the size of the chunk, is introduced in the experiments for Figure 8(b). With no computation cost, as the



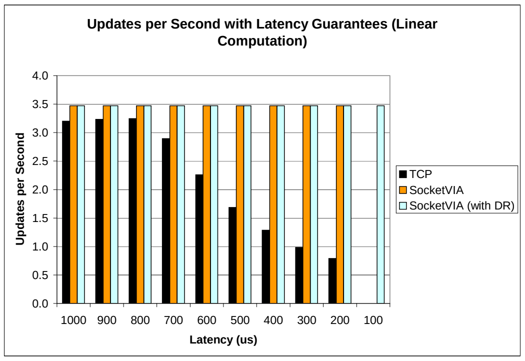


Figure 8. Effect of High Performance Sockets on Updates per Second with Latency Guarantees for (a) No Computation Cost and (b) Linear Computation Cost

latency constraint becomes as low as 100μs, TCP drops out. However, SocketVIA continues to give a performance close to the peak value. The results of this experiment show an improvement of more than 6 times without any repartitioning of data, and more than 8 times with repartitioning of data. With a computation cost, we see that for a large latency guarantee, TCP and SocketVIA perform very closely. The reason for this is the computation cost in the message path. With a computation cost of 18ns per byte, processing of data becomes a bottleneck with VIA. However, with TCP, the communication is still the bottleneck. Because of the same reason, unlike TCP, the frame rate achieved by SocketVIA does not change very much as the requested latency is decreased. The results for this experiment show a performance improvement of up to 4 times.

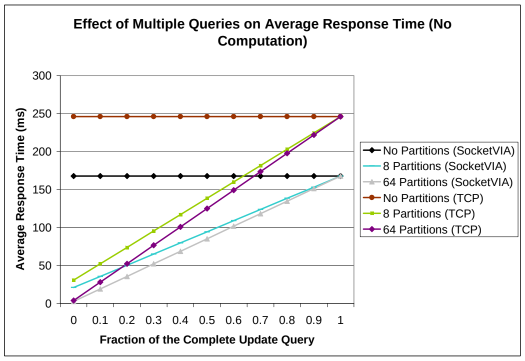
Effect of Multiple queries on Average Response Time: In the third set of experiments, we consider a model where there is a mixture of two kinds of queries. The first query type is a zoom or a magnification query, while the second one is a complete update query. The first query covers a small region of the image, requiring only 4 data chunks to be retrieved. However, the second query covers the entire image, hence all the data chunks should be retrieved and processed. Figures 9(a) and 9(b) display the average response time to queries. The x-axis shows the fraction of queries that correspond to the second type. The remaining fraction of queries correspond to the first type. The volume of data chunks accessed for each query depends on the partitioning of the dataset into data chunks. Since the fraction of queries of each kind may not be known a priori, we analyze the performance given by TCP and SocketVIA with different partition sizes. If the dataset is not partitioned into chunks, a query has to access the entire data, so the timings do not vary with varying fractions of the queries. The benefit we see for SocketVIA compared to TCP is just the inherent benefit of SocketVIA and has nothing to do with the partition sizes. However, with a partitioning of the dataset into smaller chunks, the rate of increase in the response time is very high for TCP compared to SocketVIA. Therefore, for any given average response time, SocketVIA can tolerate a higher variation in the fraction of different query types than TCP. For example, for an average response time of 150ms and 64 partitions per block, TCP can support a variation from 0% to 60% (percentage of the complete update queries), but fails after that. However, for the same constraint, SocketVIA can support a variation from 0% to 90% before failing. This shows that in cases where the block size cannot be pre-defined, or just an estimate of the block size is available, SocketVIA can do much better.

5.2.3 Effect of SocketVIA on Heterogeneous Clusters

In the next few experiments, we analyze the effect of SocketVIA on a cluster with a collection of heterogeneous compute nodes. We emulate slower nodes in the network by making some of the nodes do the processing on the data more than once. For host-based protocols like TCP, a decrease in the processing speed would result in a degradation in the communication time, together with a degradation in the computation time. However, in these experiments, we assume that communication time remains constant and only the computation time varies.

Effect of the Round-Robin scheduling scheme on Heterogeneous Clusters: For this experiment, we examine the impact on performance of the round-robin (RR) buffer scheduling in DataCutter when TCP and SocketVIA are employed. In order to achieve perfect pipelining, the time taken to transfer the data to a node should be equal to the processing time of the data on each of the nodes. For this experiment, we have considered load balancing between the filters of the Visualization Application (the first nodes in the pipeline, Figure 6). The processing time of the data in each filter is linear with message size (18ns per byte of message). With TCP, a perfect pipeline was observed to be achieved by 16KB message. But, with SocketVIA, this was achieved by 2KB messages. Thus, load balancing can be done at a much finer granularity.

Figure 10 shows the amount of time the load balancer takes to react to the heterogeneity of the nodes, with increasing factor of heterogeneity in the network. The factor of heterogeneity is the ratio of the processing speeds of the fastest and the slowest processors. With TCP, the block size



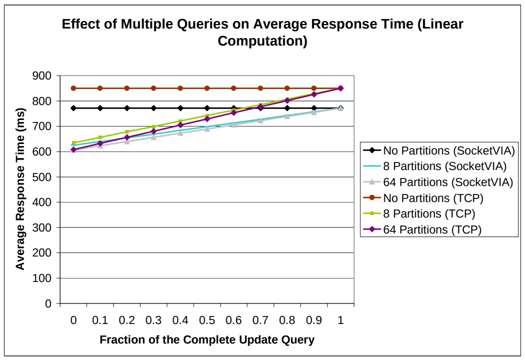


Figure 9. Effect of High Performance Sockets on the Average Response Time of Queries for (a) No Computation Cost and (b) Linear Computation Cost

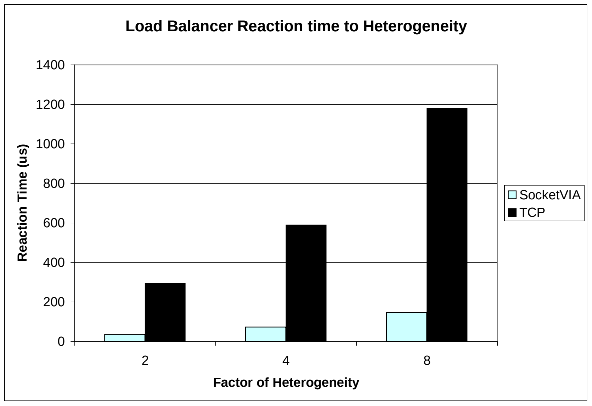


Figure 10. Effect of Heterogeneity in Processing Speed on Load Balancing using the Round-Robin Scheduling Scheme

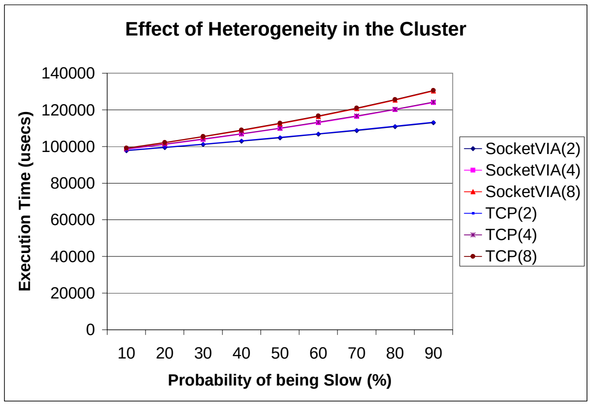


Figure 11. Effect of Heterogeneity in Processing Speed on Load Balancing using the Demand-Driven Scheduling Scheme

is large (16KB). So, when the load balancer makes a mistake (sends a block to a slower node), it results in the slow node spending a huge amount of time on processing this block. This increases the time the load balancer takes to realize its mistake. On the other hand, with SocketVIA, the block size is small. So, when the load balancer makes a mistake, the amount of time taken by the slow node to process this block is lesser compared to that of TCP. Thus the reaction time of the load balancer is lesser. The results for this experiment show that with SocketVIA, the reaction time of the load balancer decreases by a factor of 8 compared to TCP.

Effect of the Demand-Driven scheduling scheme on Heterogeneous Clusters: For this experiment, we examine the impact on performance of the demand-driven (DD) buffer scheduling in DataCutter when TCP and SocketVIA are employed. Due to the same reason as the Round-Robin scheduling (mentioned in the last subsection), a block size of 2KB was chosen for socketVIA and a block size of 16KB for TCP.

Figure 11 shows the execution time of the application. The node is assumed to get slow dynamically at times. The probability of the node becoming slower is varied on the x-axis. So, a probability of 30% means that, 30% of the computation is carried out at a slower pace, and the remaining 70% is carried out at the original pace of the node. In Figure 11, the legend socketVIA(n) stands for the application running using socketVIA and a factor of heterogeneity of ‘n’. The other legends are interpreted in a similar manner.

We observe that application performance using TCP is close to that of socketVIA. This is mainly because of the fact that demand-driven assignment of data chunks to consumers allows more work to be routed to less loaded processors. In addition, pipelining of data results in good overlap between communication and computation. Thus, our results show that if high-performance substrates are not available on a hardware configuration, applications should be structured to take advantage of pipelining of computations and dynamic scheduling of data. However, as our earlier results show, high-performance substrates are desirable for performance and latency guarantees.

6 Conclusions and Future Work

Together with a pure performance requirements, data intensive applications have other requirements such as guarantees in performance, scalability with these guarantees and adaptability to heterogeneous networks. Typically such applications are written using the kernel-based sockets interface over TCP/IP. To allow such applications take advantage of the high performance protocols, researchers have come up with a number of techniques including High Performance Sockets layers over User-level protocols such as Virtual Interface Architecture and the emerging InfiniBand Architecture. However, these sockets layers are fundamentally limited by the fact that the applications using them had been written keeping the communication performance of TCP/IP in mind.

In this paper, we study the capabilities and limitations of such a substrate, termed *S*ocketVIA, in performance, with respect to a component framework designed to provide runtime support for data intensive applications, termed as *DataCutter*. The experimental results show that by reorganizing certain components of the applications, we can make significant improvements in the performance, leading to a higher scalability of the applications with performance guarantees and fine grained load balancing making them more adaptable to heterogeneous networks. The experimental results also show that the different performance characteristics of SocketVIA allow a more efficient partitioning of data at the source nodes, thus improving the performance up to an order of magnitude in some cases. This shows that together with high performance, low-overhead substrates provide the ability to applications to simultaneously meet quality requirements along multiple dimensions. These results have strong implications on designing, developing, and implementing next generation data intensive applications on modern clusters.

As a future direction, we plan to investigate DataCutter with the push/pull data transfer model using RDMA operations in the emerging networks.

References

[1] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, November 1998.

[2] Infiniband Trade Association. http://www.infinibandta.org.

[3] P. Balaji, P. Shivam, P. Wyckoff, and D. K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *the Proceedings of IEEE International Conference on Cluster Computing*, pages 179–186, Chicago, Illinois, 23-26 September 2002.

[4] P. Balaji, J. Wu, T. Kurc, U. Catalyurek, D. K. Panda, and J. Saltz. Impact of High Performance Sockets on Data Intensive Applications. Technical Report OSU-CISRC-1/03-TR05, The Ohio State University, Columbus, OH, January 2003.

[5] M. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a framework for data-intensive wide-area applications. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW2000)*, pages 116–130. IEEE Computer Society Press, May 2000.

[6] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, October 2001.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network. http://www.myricom.com.

[8] P. Buonadonna, A. Geweke, and D. E. Culler. BVIA: An Implementation and Analysis of Virtual Interface Architecture. In *Proceedings of Supercomputing*, 1998.

[9] U. Catalyurek, M. D. Beynon, C. Chang, T. Kurc, A. Sussman, and J. Saltz. The virtual microscope. *IEEE Transactions on Information Technology in Biomedicine*. To appear.

[10] Common Component Architecture Forum. *http://www.cca-forum.org*.

[11] GigaNet Corporations. http://www.giganet.com.

[12] GigaNet Corporations. cLAN for Linux: Software Users' Guide.

[13] Myricom Corporations. The GM Message Passing System.

[14] H. Frazier and H. Johnson. Gigabit Ethernet: From 100 to 1000Mbps.

[15] Roch Guerin and Henning Schulzrinne. *The Grid: Blueprint for a New Computing Infrastructure*, chapter Network Quality of Service. Morgan Kaufmann, 1999.

[16] J. S. Kim, K. Kim, and S. I. Jung. SOVIA: A User-level Sockets Layer over Virtual Interface Architecture. In *the Proceedings of IEEE International Conference on Cluster Computing*, 2001.

[17] R. Oldfield and D. Kotz. Armada: A parallel file system for computational. In *Proceedings of CCGrid2001*, May 2001.

[18] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of Supercomputing*, 1995.

[19] F. Petrini, W. C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *Hot Interconnects*, 2001.

[20] B. Plale and K. Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *HPDC*, August 2000.

[21] H. V. Shah, C. Pu, and R. S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *the Proceedings of CANPC workshop (held in conjunction with HPCA Conference), pages 91-107*, 1999.