

Improving the Performance of Remote I/O Using Asynchronous Primitives *

Nawab Ali and Mario Lauria
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{alin, lauria}@cse.ohio-state.edu

Abstract

*An increasing number of scientific applications need efficient access to large datasets held at remote storage facilities. However, despite the availability of high-speed Internet backbones, the performance penalty of remote I/O is still relatively high compared to local I/O. In this paper we describe different ways in which asynchronous primitives can be used to improve the performance of remote I/O in the Grid environment. We have implemented and evaluated three optimization techniques using asynchronous primitives. These primitives have been integrated into SEMPLAR, a high-performance, remote I/O library based on the SDSC Storage Resource Broker. Based on measurements of representative high-performance applications running on three different clusters, we show that different optimization techniques work best for each specific combination of application and platform characteristics. We achieved over 90% overlap between the computation and I/O phase of two applications by using an asynchronous version of remote I/O primitives. We were able to increase the average read and write bandwidth of the ROMIO *perf* benchmark by 96% and 43% respectively by moving data concurrently over multiple remote connections. Finally, we experienced an improvement of up to 84% in the average write bandwidth when using asynchronous, on-the-fly data compression.*

1 Introduction

A growing number of scientific applications in experimental physics, computational biology, astrophysics and other engineering fields need access to terabytes and petabytes of data. The fact that this data is frequently the

result of large collaborations combined with its sheer magnitude makes it difficult for researchers to store copies of these datasets locally.

This problem has given rise to the concept of large data centers acting as data repositories. As a consequence of these trends, high-performance applications increasingly need access to data stored at remote locations. This trend has been further encouraged by the availability of high-speed wide area networks and by the development of Grid technologies for distributed computing [9, 18].

While the wide availability of remote I/O tools solves the problem of storing and accessing large volumes of data, there is still a definite performance gap between local and remote data access. Even on high-bandwidth wide area networks like the TeraGrid [8], local data access is at least an order of magnitude faster than remote I/O [9]. This performance gap is evident with respect to both the available bandwidth and the latency of the transfer over wide area networks.

The performance of remote I/O in grid computing is documented in our prior work [9] and also in the following work by Casanova [15]. Previous work has shown that a number of techniques such as parallel TCP streams [9, 15], tuning of TCP buffers [19, 29] and overlapping communication and disk I/O [13, 23] can be used to improve the throughput of WAN connections. Our contribution here is to show how asynchronous primitives can be used to improve the performance of remote file I/O.

In our previous work we focused on increasing the remote I/O bandwidth of parallel applications [9]. We designed a scalable, high-performance, remote I/O library called SEMPLAR that performs I/O over the Internet. SEMPLAR is based on the SDSC Storage Resource Broker (SRB) [7, 9, 12]. SRB is a middleware which provides applications with storage virtualization and a logical, remote filesystem. SEMPLAR uses multiple, parallel TCP streams [10, 25], each with a separate endpoint to connect to the SRB server. We have integrated SEMPLAR with MPI-IO [17] to preserve the parallelism of the transfer all the way

*This work is supported in part by the National Partnership for Advanced Computational Infrastructure, the Ohio Supercomputer Center through grants PAS0036 and PAS0121 and by NSF grant CNS-0403342. Mario Lauria is partially supported by NSF DBI-0317335. Support from Hewlett-Packard is also gratefully acknowledged.

to the application level, thereby enabling high-bandwidth, remote data access from within parallel applications.

In this paper we present three different ways in which asynchronous primitives can be used to improve the performance of remote I/O in the Grid environment. We believe that asynchronous I/O is a flexible programming model that provides end users and library developers with the ability to hide the I/O latency using different techniques. The most obvious and time-honored approach is to remove I/O from the critical path of the application by overlapping the application's computation and I/O phase. Most scientific applications alternate between computation and I/O phases [26]. This is largely because applications such as simulation codes often write data to secondary storage as *snapshots* every few timesteps for future analysis. Long-running scientific applications also *checkpoint* data regularly, saving the simulation state for recovery in case of a system crash. Visualization tools tend to read large amounts of data periodically for subsequent computation [20, 21]. Scientific applications therefore provide an ideal scenario for overlapping the computation phase with the I/O phase using asynchronous I/O.

Asynchronous primitives can be used to further enhance I/O performance in other less obvious ways. In this paper we show a method of increasing the remote I/O bandwidth of applications by using multiple concurrent connections from a single node. The crucial aspect in this experiment is the ability to transfer data on two different TCP streams by using SEMPLAR's asynchronous interface. The use of asynchronous primitives enables the transfer on both connections to advance *simultaneously*, ideally doubling the observed throughput. This *split-TCP* approach is not feasible with synchronous I/O because the blocking nature of the programming model prevents any concurrent data transfer.

We also show how on-the-fly data compression can be used to reduce the network traffic during remote I/O. For the compression to actually produce any performance advantage, the following condition must be satisfied: $(T_{Comp} + T_{Comp_xmit} + T_{Decomp}) < T_{Uncomp_xmit}$, where T_{Comp} and T_{Decomp} refer to the time taken to compress and decompress the data respectively and T_{Comp_xmit} and T_{Uncomp_xmit} refer to the time taken to transmit the compressed and uncompressed data respectively. The benefits afforded by compression are often negated by the overhead of the compression algorithm. Asynchronous I/O enables applications to perform compression outside the application's critical path by overlapping compression with I/O. Further, running the application on multi-core processors or on dual CPU nodes will ensure that the application's performance is not adversely affected by the overhead associated with compression.

We have enhanced SEMPLAR to include support for

asynchronous, remote I/O. Since SRB does not provide native support for asynchronous I/O, we have implemented a multi-threaded solution with dedicated I/O threads. We present the performance results from two representative applications and a microbenchmark. We also demonstrate the scalability of our approach vis-à-vis synchronous, remote I/O techniques.

The main contribution of this paper is to demonstrate that the real benefit of asynchronous primitives in the context of remote I/O is the flexibility in implementing different optimization approaches. While asynchronous I/O has been studied extensively by other researchers [16, 22, 24], most of the work has focused on evaluating its performance with respect to local file I/O. We believe that this paper is the first such study that attempts to address the relevance of non-blocking I/O in the grid environment. One of the lessons learned in the course of our experiments is that no single optimization technique works equally well in every situation. Therefore it is important to provide the application developer with a choice of different optimization mechanisms. Prior research into asynchronous I/O has focused mainly on overlapping either communication or computation with I/O. In this paper we present other less obvious techniques such as using multiple, concurrent TCP streams and on-the-fly data compression as examples of how an asynchronous I/O API can be used to improve the performance of remote I/O. We hope our results will motivate the designers of current remote access tools to enhance their programming interface with asynchronous primitives.

The rest of the paper is organized as follows. Section 2 summarizes the prior research that has been done in remote, asynchronous I/O. Section 3 presents an overview of SEMPLAR. Section 4 characterizes some of the issues related to the design and implementation of asynchronous I/O in SEMPLAR. Section 5 and Section 6 discuss our experimental setup and benchmarks respectively. Section 7 presents the results. Section 8 discusses a few potential limitations of our approach. We end by presenting our conclusions and future work in Section 9.

2 Related Work

A significant amount of prior research has focused on hiding the I/O latency of parallel applications. More *et al.* implemented a multi-threaded MPI-based I/O library called MTIO [22]. MTIO uses a single I/O thread to perform collective I/O in the background. The authors reported an overlap of up to 80% between computation and I/O for the benchmark program on the IBM SP2. Dickens *et al.* also studied the use of threads to improve the collective I/O performance of applications [16]. However, instead of performing the entire collective I/O in the background, the authors suggest overlapping only the actual write phase with the foreground computation and communication. The

above two papers focus exclusively on local I/O. They are not designed to deal with the issues of bandwidth, high network latency and other performance tradeoffs that are associated with wide area networks. Our work, in contrast focuses on a multi-threaded approach to achieve good remote I/O performance in a Grid environment.

RFS [20] is a high-performance remote I/O facility for ROMIO which improves the I/O performance of applications by adopting active buffering with threads (ABT) [21]. ABT combines background I/O using threads with aggressive buffering to minimize the I/O latency. The authors reported a considerable overlap between computation and collective I/O in simulation runs using ABT. Our work also focuses on overlapping the computation and I/O phases of parallel applications. However, instead of staging the data locally and performing the transfer in the background, we use asynchronous I/O to achieve this overlap. We also demonstrate additional advantages of providing an asynchronous I/O API to applications such as performing remote I/O concurrently over multiple TCP streams and on-the-fly data compression. While ABT may have obvious advantages such as being completely transparent to users, we feel that asynchronous I/O is a powerful programming model that offers significant performance improvements over other models. Section 4.1 discusses the advantages of asynchronous, remote I/O in more detail.

Global Access to Secondary Storage (GASS) [14] provides high-performance remote file access by using aggressive caching schemes for some common grid file access patterns. Our work instead addresses more general workloads and is not tuned for any specific file access patterns. Pai *et al.* [24] proposed an asymmetric, multi-process, event-driven (AMPED) web server architecture that attempts to overlap CPU processing with disk access and network communication. The AMPED web server architecture uses *helper* processes to handle the *local* disk I/O calls asynchronously. In contrast, our work is geared towards providing an asynchronous API for *remote* file I/O.

3 SEMPLAR

SRB-Enabled MPI-IO Library for Access to Remote Storage (SEMPLAR) [9] is a remote, parallel I/O library that combines the standard programming interface of MPI-IO with the remote storage functionality of the SDSC Storage Resource Broker (SRB). SEMPLAR uses parallel TCP streams to maximize the remote data throughput of applications.

3.1 Storage Resource Broker

The Storage Resource Broker (SRB) is a data management system that provides applications with a logical namespace to manage the data. It was developed by the

San Diego Supercomputer Center (SDSC) to provide support for data-intensive computing. SRB can be viewed as a distributed logical filesystem (SRBFS) which exports a uniform interface to data distributed across multiple, heterogeneous storage systems [7, 9, 12].

The SRB system consists of clients, servers and the Metadata Catalog Service (MCAT). The MCAT subsystem manages the attributes associated with the SRB system objects. The I/O interface exported by SRB is semantically equivalent to the POSIX file I/O API [9, 12].

3.2 SEMPLAR Design and Implementation

SEMPLAR is a high-performance, parallel I/O library that enables data-intensive applications to perform I/O over wide-area networks. It aims to provide scientific applications with storage virtualization and high I/O bandwidth. SEMPLAR stripes the application data across multiple, concurrent TCP streams. Each I/O stream has a separate endpoint between the application cluster and the storage repository [9].

We have integrated the SRB remote filesystem with ROMIO [6], the MPI-IO implementation from Argonne National Laboratory. ROMIO uses the *Abstract-Device Interface for I/O (ADIO)* [27] framework to implement parallel I/O APIs. A parallel I/O API can be implemented portably on multiple filesystems by implementing it on top of ADIO. The ADIO implementation is then optimized for different filesystems [28]. This framework enables the library developer to exploit the specific high-performance features of individual filesystems while providing a portable implementation of the parallel I/O API [9]. Figure 1 [9] shows how MPI-IO can be implemented on multiple filesystems by using the ADIO framework.

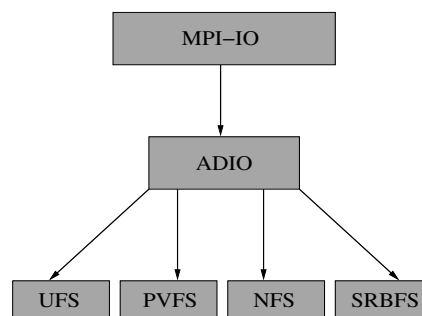


Figure 1. The ROMIO Architecture: Implementing MPI-IO portably on different filesystems by using ADIO.

SEMPLAR provides a high-performance implementation of ADIO for the SRB filesystem. A parallel application running on multiple cluster nodes opens individual TCP connections from each node to the SRB server. The

network connection is established during the call to the `MPI_File_open` function. The application then stripes its I/O data across the TCP streams to perform parallel I/O. The parallelism associated with striping data across multiple, concurrent TCP streams results in higher aggregate data throughput. The fact that each node establishes a separate TCP connection to perform I/O avoids the problem of the end nodes becoming the bottleneck. It also results in better network utilization. The `MPI_File_close` function terminates the connection to the SRB server [9].

4 Multi-threaded Asynchronous Remote I/O

The performance gap between local I/O and remote I/O is dependent on two main factors: *I/O Bandwidth* and *I/O Latency*. Our previous work [9] focused on providing parallel applications with high I/O bandwidth over wide area networks. This paper presents multiple techniques for increasing the performance of remote I/O by providing applications with a non-blocking API. The asynchronous API can be used for traditional optimization techniques such as overlapping the computation and I/O phases of applications. It can also be used to provide higher network throughput by using multiple concurrent TCP streams and by performing on-the-fly data compression.

4.1 The Case for Remote Asynchronous I/O

Asynchronous I/O is a powerful programming model that offers applications a significant performance advantage. It enables applications to parallelize I/O and computation, resulting in increased CPU utilization and enhanced I/O performance. There are however, some drawbacks associated with asynchronous I/O. It prevents applications from reusing the I/O buffers until the non-blocking I/O operation is complete. It is also not transparent to users, exposing them to a new set of I/O APIs. Further, asynchronous I/O is just an enabling technology. Applications may have to be rewritten to experience significant performance improvements when using non-blocking I/O calls.

Even with the above caveats, asynchronous I/O is an important I/O model. It is part of the POSIX and MPI-2 standards and as such is available on several filesystems. It also provides a high-performance I/O platform, particularly for remote I/O. Asynchronous primitives can enable an application to concurrently transfer data across multiple TCP streams. This can result in a significant increase in I/O performance. Data striping can also be used for redundancy purposes with multiple streams carrying the same data to the destination. The first data stream to reach the destination end-point is accepted while the others are ignored. This technique could result in lower I/O latency, particularly if the data streams follow different paths to the destination. Moreover, applications which use asynchronous primitives

for performing I/O can take advantage of the parallelism offered by the new multi-core architectures. These are some of the reasons why we feel that asynchronous I/O is an excellent model for performing remote I/O.

4.2 Asynchronous I/O Interface Design

The asynchronous I/O programming interface is implemented using two threads: the *Compute Thread* and the *I/O Thread*. The threads share an I/O queue that contains the application I/O requests. When a MPI application makes a call to an asynchronous I/O function such as `MPI_File_irewrite`, the compute thread places the I/O request in the I/O queue and returns immediately. The I/O thread dequeues the I/O queue in FIFO order and calls the corresponding synchronous function to service the request. Figure 2 shows the architecture behind SEMPLAR's asynchronous I/O interface. This architecture allows the remote filesystem (SRBFS) to complete the I/O in the background while the compute thread is performing the computation, thus allowing for overlapping between computation and I/O. Also, by implementing the asynchronous I/O calls over the corresponding synchronous functions, we ensure that the asynchronous capability of the remote filesystem is orthogonal to other I/O optimizations [22]. The compute thread can periodically check the status of the I/O requests by making calls to MPI-IO functions such as `MPIO_Wait` and `MPIO_Test`.

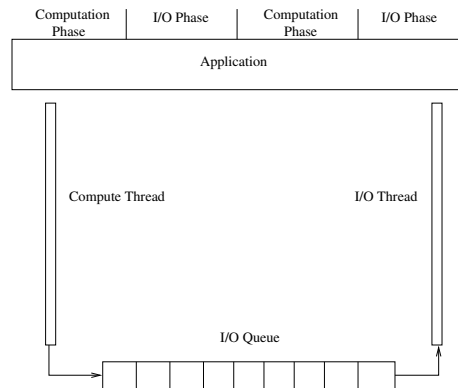


Figure 2. A multi-threaded framework for implementing asynchronous primitives in SEMPLAR.

4.3 Asynchronous I/O Implementation

The SRB filesystem does not provide a native asynchronous I/O implementation. We used a multi-threaded design to enable asynchronous I/O on SRB. Despite the additional complexity, there are advantages in having multiple threads for implementing asynchronous I/O. Since threads share their address space, we can avoid the overhead of

copying I/O data between the compute and I/O threads. Threads also enable true asynchronous I/O on filesystems such as SRB that do not support asynchronous I/O natively [22].

We have used the POSIX *pthread* library in our implementation. The POSIX *pthread* library is a kernel-level thread library which is scheduled by the operating system kernel. This avoids the problem often seen in user-level thread libraries where a blocking thread can block the entire process [16]. SEMPLAR's asynchronous I/O subsystem can be configured to use either a single or multiple I/O threads. The decision on how many threads to use depends on a number of factors. In general, a single I/O thread reduces the cost of thread scheduling and switching. The asynchronous I/O calls are also faster because they do not have to spawn a new thread. However, the cost of scheduling multiple threads on recent operating systems is much lower than before. For example, the Linux 2.6 kernel thread scheduling algorithm has a constant overhead ($O(1)$). Moreover, the overhead associated with spawning a new thread can be avoided by pre-spawning a pool of I/O threads. In SEMPLAR, the advantage of using multiple I/O threads depends mainly on the number of TCP streams associated with each thread. If all the I/O threads share a single TCP connection, the asynchronous I/O requests are mapped to a single network stream. This reduces the parallelism offered by multiple I/O threads. The ideal scenario would involve associating a single TCP stream with each I/O thread. We conducted the first set of experiments (Section 7.1) using a single I/O thread. The first call to an asynchronous MPI file I/O function spawns the I/O thread. Subsequent calls do not create new I/O threads. We used pre-spawned, multiple I/O threads for the second set of experiments (Section 7.2).

The I/O threads suspend themselves whenever the I/O queue is empty by waiting on a condition variable. The main computation thread signals the I/O threads whenever it places a new I/O request in the queue. This avoids the problem of *busy wait* where the I/O threads continuously poll the I/O queue for new requests.

Section 3.2 provides a description of implementing the MPI-IO interface on the SRB filesystem. We extended this implementation by adding support for the following asynchronous I/O calls: `MPI_File_iread`, `MPI_File_iwrite`, `MPIO_Wait` and `MPIO_Test`.

5 Experimental Setup

This section describes the setup we used to evaluate the asynchronous I/O performance of SEMPLAR [9]. The DAS-2 cluster represents a high-latency (~ 182 ms), low-bandwidth environment while the OSC cluster represents a low-latency (~ 30 ms), high-bandwidth environment. The TG-NCSA cluster is part of the TeraGrid and as such is connected to a low-latency (~ 30 ms), high-bandwidth (40Gbps)

backbone. The TG-NCSA cluster represents the TeraGrid-enabled grid environment. Figure 3 [9] provides a representation of the experimental setup.

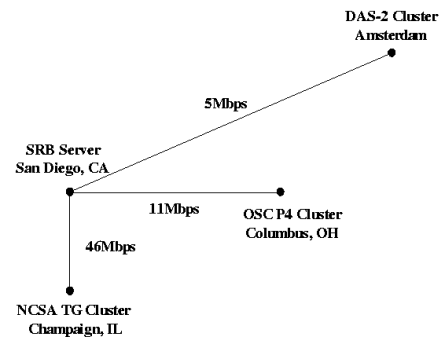


Figure 3. The Experimental Setup

SDSC SRB Server - The SDSC SRB team manages a production SRB server on `orion.sdsc.edu`. `orion` is a high-end SUN Fire 15000 machine. It contains 36, 900MHz SPARC III+ processors, 144GB of memory, 6 Gigabit Ethernet interfaces for data, 1 Gigabit Ethernet interface for control information and 16 T9940B tape drives. The machine runs the Solaris 9 operating system.

Distributed ASCI Supercomputer 2 - DAS-2 [2] is a wide-area distributed cluster designed by the Advanced School for Computing and Imaging (ASCI) in the Netherlands. It consists of 200 Dual Pentium-III nodes. Each node contains two 1GHz Pentium-III processors, 1GB of RAM, a 20 GB local IDE disk, a Myrinet interface card and an on-board Fast Ethernet interface. Each compute node is connected to the outside world by a 100Mbps link. The nodes run the Red Hat Enterprise Linux operating system. We used the nodes located at Vrije Universiteit, Amsterdam for our experiments.

OSC Pentium 4 Xeon Cluster - The Ohio Supercomputer Center [5] Pentium 4 cluster is a distributed/shared memory hybrid system. The cluster consists of 512, 2.4GHz Intel Xeon processors. Each node has two 2.4GHz processors, 4 GB of memory, a 100Base-T Ethernet interface and one Gigabit Ethernet interface. The nodes run the Red Hat Enterprise Linux operating system.

NCSA TeraGrid cluster - The TeraGrid cluster at the National Center for Supercomputing Applications (NCSA), consists of 887 IBM cluster nodes. Each of the 256 *Phase 1* nodes contains dual 1.3 GHz Intel Itanium 2 processors while the remaining 631 *Phase 2* nodes contain dual 1.5 GHz Intel Itanium 2 processors each. Each node is also equipped with a Gigabit Ethernet interface. The cluster runs the SuSE Linux operating system.

6 Benchmarks

We used two applications and a microbenchmark to evaluate SEMPLAR. This section gives a brief description of the benchmarks. We used these applications as benchmarks because they are representative of scientific applications. They have loops in their bodies and are I/O intensive.

ROMIO perf - `perf` is a sample MPI-IO benchmark included in the ROMIO source code. It measures the I/O performance of a filesystem. Each process writes a data array to a shared file at a fixed location using `MPI_File_write`. The data is then read back using `MPI_File_read`. The location from which a process reads and writes data is determined by its rank. The benchmark uses individual file pointers and non-collective calls to perform I/O [9, 11]. We modified the `perf` source code to use asynchronous I/O.

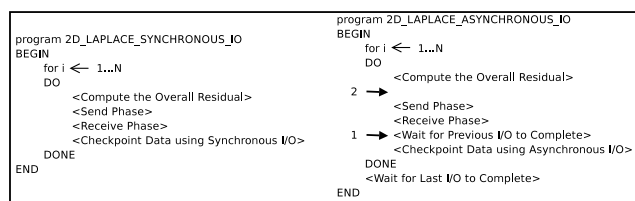


Figure 4. 2D Laplace Solver Pseudocode

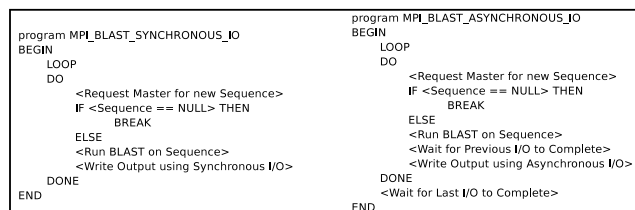


Figure 5. MPI-BLAST Pseudocode

2D Laplace Solver - This benchmark was developed by the Science and Technology Group at the Ohio Supercomputer Center [5]. The *2D Laplace Solver* solves Laplace’s equation on a two-dimensional grid of fixed size. The benchmark was extended to add MPI-IO calls to write a checkpoint file periodically [11]. It uses individual file pointers and non-collective calls to write the data to a shared file. Figure 4 shows the pseudocode of the 2D Laplace solver. It also contains the changes to the code that are required to use asynchronous I/O.

MPI-BLAST - Sequence alignment is a method used to discover the inherent relationships among biological sequences such as DNA and proteins [3]. The *Basic Local Alignment Search Tool (BLAST)* provides a method for searching protein and nucleotide databases to detect local alignments for any particular sequence [1].

MPI-BLAST¹ is a MPI wrapper to enable BLAST to run on Message Passing machines. MPI-BLAST designates one compute processor to be the *master*. The remaining processors are called *workers*. The *master* processor manages the sequence query file. On receiving a request from a *worker*, the *master* sends a sequence to the *worker*. The *worker* then searches the database for the sequence using BLAST. Each *worker* uses individual file pointers and non-collective calls to write the BLAST output to independent, remote files. Figure 5 shows the pseudocode of the MPI-BLAST program. The asynchronous version of the code will run faster because it allows the computation phase of one iteration to overlap with the I/O phase of the previous iteration.

7 Results

This section presents the I/O performance results for the 2D Laplace solver, ROMIO `perf` and the MPI-BLAST benchmarks. We ran the benchmarks on the DAS-2, NCSA TeraGrid and the OSC P4 Xeon clusters. The SDSC SRB server (version 3.2.1) running on `orion.sdsc.edu` was used to interface with the data repository at SDSC. We have integrated SEMPLAR with `mpich-1.2.6`.

7.1 Overlapping Computation and I/O

In this experiment we use asynchronous primitives to enable the overlap of computation with remote I/O. We quantify the performance benefits that can be obtained with SEMPLAR and we also describe a counter-intuitive result that can be produced by neglecting the possibility of resource contention.

MPI-BLAST - Figure 6 shows the performance of the MPI-BLAST benchmark on the DAS-2, OSC P4 and the TG-NCSA clusters. The BLAST database contains a subset of the sequences of all human ESTs in GenBank at UCSC (687,158 sequences for a total size of 256MB). The query file managed by the MPI-BLAST *master* processor is a subset of the BLAST database and contains 2425 sequences (1MB). BLAST generates about 50KB of data for each sequence. As we increase the number of processors in the system, we notice a decrease in the execution time of the benchmark for both synchronous and asynchronous I/O. This is because the fixed set of query sequences is divided among an increasing number of processors. This results in each processor searching for fewer sequences in the database, thereby reducing the duration of the application’s I/O and computation phase. The MPI-BLAST result also shows that asynchronous I/O provides consistent performance improvements over synchronous I/O on all the

¹MPI-BLAST was developed independently at the Ohio State University. It is not related to the mpiBLAST project at the Los Alamos National Laboratory.

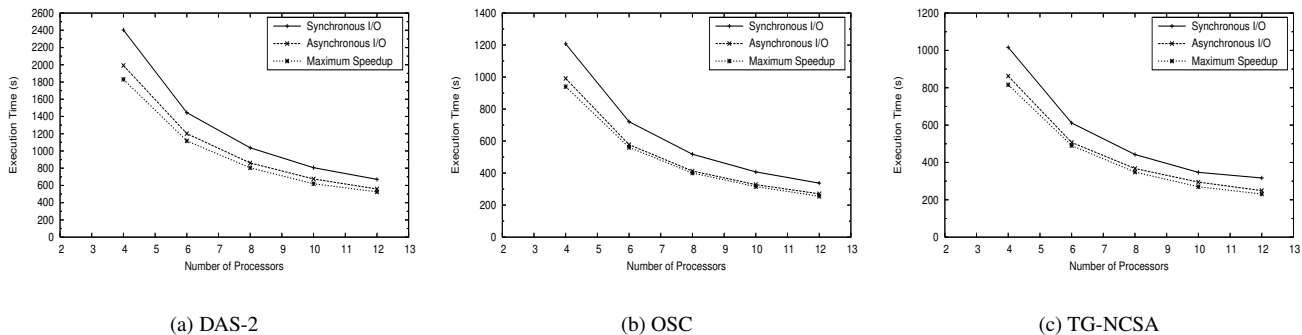


Figure 6. MPI-BLAST Execution Time

three clusters. On the DAS-2 cluster, the average execution time of the benchmark increased by 20% for the synchronous I/O run. We observed similar results on the other two clusters. On the OSC P4 and the TG-NCSA cluster, the average execution time of the benchmark increased by 26% and 22% respectively for the synchronous I/O run.

We also computed the expected execution time assuming the complete overlap of the computation and I/O phases and the corresponding maximum speedup. We measured the duration of the computation and I/O phases of the applications; the larger of these two times represents the expected execution time of the application when the computation and I/O phases are completely overlapped. We then derived the speedup by comparing the measured execution time with the expected time.

On the DAS-2 cluster, we were able to achieve 92% of the maximum expected speedup. On the OSC P4 and the TG-NCSA cluster, we managed to achieve 97% and 96% of the maximum expected speedup respectively.

2D Laplace Solver - Figure 7 shows the performance of the MPI Laplace solver benchmark for a 3001x3001 grid. The 2D Laplace solver writes about 250MB of data to the remote filesystem. We see a consistent decrease in the execution time of the benchmark with an increase in the number of processors. This can be explained by the fact that the amount of data being written by an individual processor reduces because of the fixed grid size. This reduces the duration of the application's I/O phase. We can also see that asynchronous I/O shows a consistent performance improvement over synchronous I/O. On the DAS-2 cluster, the average execution time of the benchmark increased by 7% for the synchronous I/O run. On the OSC P4 and the TG-NCSA cluster, the average execution time of the benchmark increased by 9% and 6% respectively for the synchronous I/O run. These results prove that asynchronous I/O can im-

prove the performance of scientific applications by reducing the latency penalty of remote I/O.

On the DAS-2 cluster, we were able to achieve 96% of the maximum expected speedup for the 2D Laplace solver. On the OSC P4 and the TG-NCSA cluster, we managed to achieve 97% of the maximum expected speedup.

As can be seen from Figure 7, the execution time of the benchmark on the TG-NCSA and the OSC clusters is significantly smaller than the execution time on the DAS-2 cluster. The variations in execution time can be explained by the fact that the TG-NCSA and the OSC clusters are connected to the SDSC SRB server over high-speed networks. The DAS-2 cluster which is located in Amsterdam is connected to the SRB server over a transoceanic, low-bandwidth link. This increases the duration of the I/O phase of the benchmark on this cluster and explains the high average execution time of the benchmark.

While asynchronous I/O does offer a potential for significant performance improvement, the actual improvement in execution time is a function of the individual application. In a perfectly balanced application where the duration of the computation phase is equal to the I/O phase, asynchronous I/O can improve the application's execution time by up to 50% by overlapping the two phases. In the 2D Laplace benchmark the ratio between the I/O phase and the computation phase is 9:1, resulting in a performance improvement of only 6%-9%. The MPI-BLAST benchmark performs much better, showing an improvement in execution time of 20%-26%. This can be attributed to the fact that the computation to I/O ratio is only 4:1 for MPI-BLAST.

2D Laplace Solver with Two TCP connections - One important difference between the 2D Laplace and MPI-BLAST benchmarks is that the former performs a substantial amount of MPI communication between the nodes.

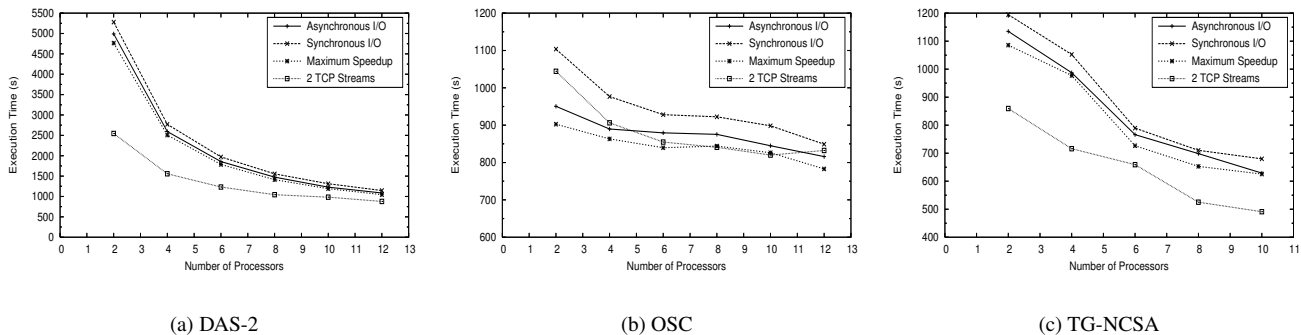


Figure 7. 2D Laplace Solver Execution Time

In such circumstances, the restructuring of the application code must be performed carefully because of possible resource conflicts.

In a separate experiment we took the original 2D Laplace Solver code and applied the modifications described below in Section 7.2 so that each node used two connections to the SRB server instead of one. The doubling of effective bandwidth decreases the average execution time by 38% on the DAS-2 cluster and by 23% on the TG-NCSA cluster (Figure 7), achieving an even higher performance improvement than the one brought about by overlapping of I/O and computation. On the OSC P4 cluster the individual nodes do not have a public IP address. They are instead connected to the outside world through a Network Address Translation (NAT) host. The bottleneck represented by the NAT host reduces the advantage of doubling the number of connections seen on the other clusters.

When we ran a version of the code modified to include both optimizations (overlapping + double connection), the resulting execution time was approximately the same as the highest of the two (overlapping alone), i.e. the advantage of the double connection was lost. We determined that the reason for this unexpected result is the I/O bus contention between the interconnect and Ethernet network cards. We were able to reduce the execution time to the level of the double connection experiment by removing the overlap between remote I/O and MPI communication (results not shown). The restructuring of the code needed to achieve this is shown in Figure 4, where the `wait` call was moved from position 1 to 2. Since most of the “computation” phase is actually spent in executing the MPI send/receive calls, the advantage of the second optimization is lost. Note that all the clusters used in the experiments have separate networks for MPI and TCP communication, which excludes the possibility of the contention being on the network. We

ran our experiments with one task per dual processor node to exclude the possibility of threads being starved of CPU cycles.

7.2 Multiple TCP Connections per Node

In this experiment we use the asynchronous I/O functionality to further stripe the application data over concurrent TCP connections. In the original SEMPLAR, a file is striped across the nodes of the client cluster and each node transfers its section of the file over its own connection to the SRB server. In this experiment each node takes advantage of the asynchronous extension of SEMPLAR to open two connections to the server. It then stripes its section of file across them. The use of asynchronous primitives enables the transfer on both connections to advance *simultaneously*, ideally doubling the observed throughput.

The doubling of the number of connections per node is achieved by calling `MPI_File_open` twice on the same file. Each of the two file descriptors thus obtained is associated with a different connection and therefore it can be read/written independent of the other. The net effect of performing asynchronous read/write calls on the two descriptors is the creation of two I/O threads, each one serving a different connection. Obviously, using this scheme an arbitrary number of connections can be created per node. However, in this paper our goal was simply to demonstrate the flexibility of asynchronous I/O calls and consequently we did not try to optimize this number. Ideally, this feature should be implemented at the library level so that the details associated with creating multiple connections per node is abstracted from the user. We intend to implement this in the future versions of SEMPLAR.

Figure 8 shows the I/O performance of the `perf` benchmark on the DAS-2 and the TG-NCSA cluster. Each node reads and writes an array of size 32MB to the remote SRB

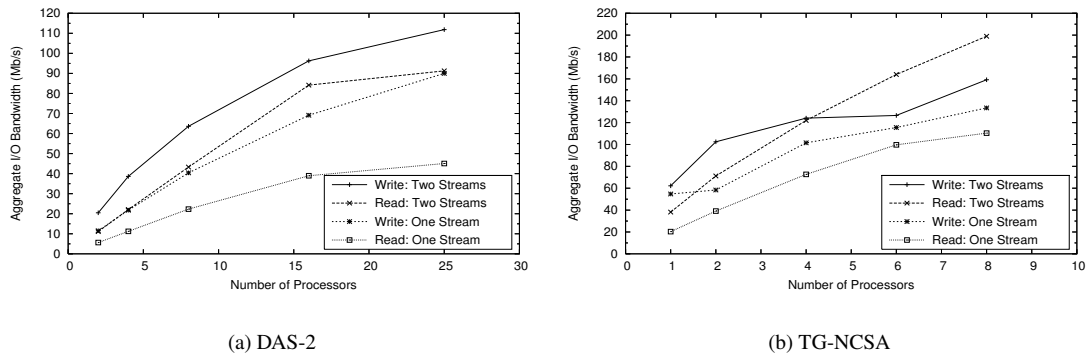


Figure 8. perf I/O Performance using Two Concurrent Transfers per Nodes

server. On the DAS-2 cluster the average write bandwidth using two TCP streams was 43% more than the bandwidth achieved using a single TCP stream. The average read bandwidth using two TCP streams also showed a significant performance improvement (96%) over the bandwidth available on a single TCP stream. We observed similar results on the TG-NCSA cluster. The average write bandwidth using two TCP streams was 24% more than the bandwidth achieved on a single TCP stream. The average read bandwidth using two TCP streams showed an improvement of 75% over the bandwidth available on a single TCP stream. These results show that even with the additional overhead of handling two connections on the same node, asynchronous I/O can significantly improve the remote I/O bandwidth of parallel applications.

7.3 On-the-fly Data Compression

On-the-fly data compression can be used to reduce the network traffic during remote I/O. In the experiment described here we show the results of using SEMPLAR for on-the-fly data compression. Each node reads a 100MB text file consisting of nucleotide sequences for the human EST and compresses the data before writing it to the remote SRB filesystem. We wrote an MPI application which uses SEMPLAR's asynchronous I/O interface to compress the data before performing remote I/O. The application uses individual file pointers and non-collective calls to write the compressed data to independent files. We ran each task on a dedicated dual processor node. The loop structure of the program and the placement of the asynchronous calls ensured that the transfer and compression of two consecutive 1MB blocks were pipelined.

The compression routine we used was taken from the LZO [4] library which implements a relatively fast compression algorithm. From direct measurements (not shown),

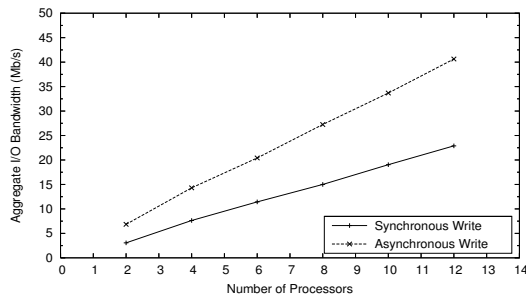
the time taken to compress the data was approximately two orders of magnitude smaller than the time required to transmit the compressed data. Clearly, this points to the opportunity afforded by our asynchronous interface to perform more advanced forms of on-the-fly preprocessing on the data (e.g. more sophisticated compression algorithms).

Figure 9 shows the aggregate I/O bandwidth of the compression benchmark on the DAS-2 and the TG-NCSA cluster. On the DAS-2 cluster the average, aggregate write bandwidth increased by 83% when using asynchronous I/O. On the TG-NCSA cluster, the average, aggregate write bandwidth increased by 84%. These results are indicative of the fact that high-performance applications can combine on-the-fly data compression with asynchronous, remote I/O to increase their I/O bandwidth.

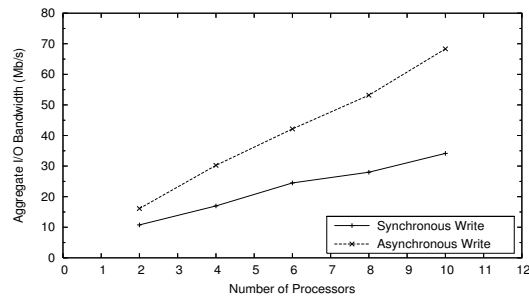
8 Discussion

While each of the nodes at the client end use one or more TCP streams for remote I/O, the streams all connect to a single SRB server. In such a scenario, the SRB server could become a potential bottleneck. However, most data-centers hosting large amounts of data are likely to use high-end machines with multiple processors and network cards to run the SRB server. In our experiments, the machine hosting the SRB server, `orion.sdsc.edu` is a high-end SUN Fire 15000 machine. It contains 36, 900MHz SPARC III+ processors, 144GB of memory and 7 Gigabit Ethernet interfaces. The SRB server can also be configured to run in a federated mode where one server can act as a client to other servers. As such, most SRB systems in production environments should be able to handle multiple, concurrent connections from a large number of clients.

This paper presents three techniques for improving the remote I/O performance of applications. Having a num-



(a) DAS-2



(b) TG-NCSA

Figure 9. On-the-fly Data Compression

ber of different, orthogonal optimizations to choose from is particularly useful because a single approach may not work well in a specific scenario. For example, the contention for the I/O bus between the interconnect and Ethernet network cards can offset the performance improvement achieved by overlapping the communication and I/O phase of an application. Further, for clusters where the nodes do not have public IP addresses, the NAT host can become a potential bottleneck. In both these situations, we can use asynchronous, on-the-fly data compression to increase the application's I/O bandwidth. The effectiveness of the compression approach itself is dependent on the compression algorithm as well as the data that is being compressed. In our experiments, the aggregate write bandwidth experienced an improvement of up to 84% when using compression. However, the performance improvement due to compression may vary among different applications.

9 Conclusions and Future Work

High-speed remote I/O is still a challenge in high-performance computing. Despite the availability of fast wide area networks, high network latency continues to be a bottleneck for remote I/O. In this paper we describe the use of asynchronous primitives to improve the performance of remote I/O. We mask the remote I/O latency by overlapping the application's computation phase with its I/O phase using asynchronous primitives. We also show a method of increasing the I/O bandwidth of applications by striping data across multiple, asynchronous TCP streams and by performing on-the-fly data compression. Having a number of different orthogonal optimizations to choose from is particularly useful because a single approach may not work well in all scenarios.

We present results for a representative, high-

performance computing workload on three different clusters. The 2D Laplace solver experienced up to 97% overlap between the computation and I/O phases by using asynchronous I/O. The MPI-BLAST benchmark experienced an average performance improvement of 20%–26%. The average read and write bandwidth measured using the ROMIO `perf` benchmark improved by 96% and 43% respectively by using multiple, asynchronous I/O streams.

In the future we plan to use data redundancy across multiple, concurrent I/O streams to reduce the network latency of remote I/O. We would also like to study the effect of asynchronous primitives on remote, collective I/O.

10 Acknowledgments

We wish to thank Reagan Moore, Marcio Faerman and Arcot Rajasekar of the Data Intensive Group (DICE) at the San Diego Supercomputer Center for giving us access to the SRB source code. We are especially indebted to Henri Bal of Vrije Universiteit, Amsterdam for giving us access to the DAS-2 cluster. We also wish to thank Rob Pennington and Ruth Aytz at the National Center for Supercomputing Applications (NCSA) for allowing us to use the NCSA TeraGrid cluster for our experiments.

References

- [1] BLAST. <http://www.ncbi.nlm.nih.gov/BLAST>.
- [2] DAS-2. <http://www.cs.vu.nl/das2>.
- [3] LSDCG. <http://lsdcg.cse.ohio-state.edu>.
- [4] LZ0. <http://www.oberhumer.com/opensource/lzo>.
- [5] Ohio Supercomputer Center. <http://www.osc.edu>.
- [6] ROMIO: A High-Performance, Portable MPI-IO Implementation. <http://www.mcs.anl.gov/romio>.
- [7] SDSC Storage Resource Broker. <http://www.sdsc.edu/srb>.
- [8] TeraGrid. <http://www.teragrid.org>.

- [9] N. Ali and M. Lauria. SEMPLAR: High-Performance Remote Parallel I/O over SRB. In *5th IEEE/ACM International Symposium on Cluster Computing and the Grid*, Cardiff, UK, 2005.
- [10] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Data Management and Transfer in High-Performance Computational Grid Environments. *Parallel Computing*, 28(5):749–771, 2002.
- [11] T. Baer. Parallel I/O Experiences on an SGI 750 Cluster. http://www.osc.edu/~troy/cug_2002.
- [12] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Canada, 1998.
- [13] K. Bell, A. Chien, and M. Lauria. A High-Performance Cluster Storage Server. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 311–320, 2002.
- [14] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, pages 78–88, May 1999.
- [15] H. Casanova. Network Modeling Issues for Grid Application Scheduling. *International Journal of Foundations of Computer Science (IJFCS)*, 6(2):145–162, 2005.
- [16] P. M. Dickens and R. Thakur. Improving Collective I/O Performance Using Threads. In *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 38–45, April 1999.
- [17] M. P. I. Forum. MPI-2: Extensions to the Message-Passing Interface. <http://www.mpi-forum.org/docs/docs.html>, 1997.
- [18] I. Foster, D. Kohr, Jr., R. Krishnaiyer, and J. Mogill. Remote I/O: Fast Access to Distant Storage. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 14–25, San Jose, CA, 1997.
- [19] J. Lee, D. Gunter, B. Tierney, B. Allcock, J. Bester, J. Bresnahan, and S. Tuecke. Applied Techniques for High Bandwidth Data Transfers Across Wide Area Networks. In *Proceedings of International Conference on Computing in High Energy and Nuclear Physics*, Beijing, China, September 2001.
- [20] J. Lee, X. Ma, R. Ross, R. Thakur, and M. Winslett. RFS: Efficient and Flexible Remote File Access for MPI-IO. In *Proceedings of the IEEE International Conference on Cluster Computing*, September 2004.
- [21] X. Ma, M. Winslett, J. Lee, and S. Yu. Improving MPI-IO Output Performance with Active Buffering Plus Threads. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2003.
- [22] S. More, A. Choudhary, I. Foster, and M. Xu. MTIO - A Multi-Threaded Parallel I/O System. In *Proceedings of the 11th International Symposium on Parallel Processing*, pages 368–373, 1997.
- [23] E. Nallipogu, F. Ozguner, and M. Lauria. Improving the Throughput of Remote Storage Access through Pipelining. In *Proceedings of the Third International Workshop on Grid Computing*, pages 305–316, 2002.
- [24] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.
- [25] L. Qiu, Y. Zhang, and S. Keshav. On Individual and Aggregate TCP Performance. In *Proceedings of the Seventh Annual International Conference on Network Protocols*, pages 203–212, 1999.
- [26] E. Smirmi and D. Reed. Lessons from Characterizing the Input/Output Behavior of Parallel Scientific Applications. *Performance Evaluation: An International Journal*, 33(1):27–44, June 1998.
- [27] R. Thakur, W. Gropp, and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, 1996.
- [28] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, 1999.
- [29] B. L. Tierney, J. Lee, B. Crowley, M. Holding, J. Hylton, and F. L. Drake, Jr. A Network-Aware Distributed Storage Cache for Data-Intensive Environments. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 185–193, 1999.