# RFS: Efficient and Flexible Remote File Access for MPI-IO

Jonghyun Lee[*‡] Xiaosong Ma[†§] Robert Ross[*] Rajeev Thakur[*] Marianne Winslett[‡]

[*] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.

[†] Department of Computer Science, North Carolina State University, Raleigh, NC 27695, U.S.A.

[‡] Department of Computer Science, University of Illinois, Urbana, IL 61801, U.S.A.

[§] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37381, U.S.A.

{jlee, rross, thakur}@mcs.anl.gov, ma@csc.ncsu.edu, winslett@cs.uiuc.edu

## Abstract

*Scientific applications often need to access remote file systems. Because of slow networks and large data size, however, remote I/O can become an even more serious performance bottleneck than local I/O performance. In this work, we present RFS, a high-performance remote I/O facility for ROMIO, which is a well-known MPI-IO implementation. Our simple, portable, and flexible design eliminates the shortcomings of previous remote I/O efforts. In particular, RFS improves the remote I/O performance by adopting active buffering with threads (ABT), which hides I/O cost by aggressively buffering the output data using available memory and performing background I/O using threads while computation is taking place. Our experimental results show that RFS with ABT can significantly reduce the remote I/O visible cost, achieving up to 92% of the theoretical peak throughput. The computation slowdown caused by concurrent I/O activities was 0.2–6.2%, which is dwarfed by the overall performance improvement in application turnaround time.*

## 1 Introduction

The emergence of fast processors and high-bandwidth, low-latency interconnects has made high-performance commodity-based clusters widely available. These clusters are gaining popularity as an affordable, yet powerful, parallel platform compared to commercial supercomputers, because of the clusters' excellent cost-performance ratio. For many computational scientists, clusters are an attractive option for running parallel scientific codes that require a large number of computing resources.

Scientific applications are typically I/O intensive. For example, most simulation codes periodically write out the intermediate simulation data to local secondary storage as *snapshots* for future time-dependent visualization or analysis. *Checkpoint* files also need to be written in case of system crash or application failure. Many visualization tools read large amounts of data from disks into memory for visualization. Despite recent improvements in disk performance, local I/O performance is still a serious performance bottleneck for these data-intensive applications. Many research efforts have addressed this slow I/O problem through utilizing I/O parallelism [5, 11, 12, 17, 20].

In addition to their local I/O needs, scientific simulations and visualization codes often need to store or retrieve data at a remote file system that might be geographically separated from where the codes are running. For example, a scientist at North Carolina State University chooses to run her simulation code on a parallel platform at Argonne National Laboratory, yet she needs to visualize and analyze the simulation data on her local machine, which is equipped with advanced graphics facilities. She also wants to visualize other people's data stored on a remote file archive. For applications that run in such distributed setups, remote I/O performance becomes an even bigger concern than local I/O performance, because the network bandwidth between two platforms is usually much lower than the aggregate disk bandwidth of the local file systems. This situation, along with the emergence of Grid computing [8], motivated researchers to develop efficient and convenient access to remote data for scientific applications [3, 4, 9, 12, 18].

The traditional approach to address the remote I/O problem is to manually stage either data or application from one platform to the other, so that data can be accessed locally. However, staging imposes several potential problems. Data staging requires enough storage space on the local machine to hold all the remote files, and such space may not always be available. Data staging can also result in excessive data transfer and consistency problems. Application staging seems useful when the amount of data to be accessed is large, but the application must be ported to the other machine, which may have a totally different architecture and software base. Further, the results of visualization or analysis may have to be transferred back to the original machine.

Moreover, staging is typically performed either before or after the application run, preventing possible overlap between them and lengthening turnaround time. Lastly, manual staging is cumbersome and inefficient.

To overcome such shortcomings, we seek to provide efficient remote data access functionality in a parallel I/O library for scientific applications. Our approach enables automatic data migration between two machines, minimizing user intervention and unnecessary data transfer. Hints can be provided by the user to tune behavior if necessary. We focus on optimizing remote write performance because a large fraction of scientific codes, especially simulations, are write intensive and do very few reads. For the testbed implementation of this work, we chose ROMIO [20], a popular implementation of the MPI-IO specification in the MPI-2 standard [15]. The MPI-IO interface is the de facto parallel I/O interface standard, used both directly by applications and by high-level libraries such as Parallel NetCDF [13] and HDF5 [1]. Supporting remote I/O through MPI-IO thus enables many applications to perform remote I/O without code changes.

The main contributions of this work are as follows. First, we propose a simple, portable, and flexible design of a remote I/O facility called RFS (Remote File System), which eliminates the shortcomings of previous remote I/O efforts, and describe its implementation with ROMIO. Second, as an optimization for remote writes, we integrate active buffering with threads (ABT) [14] with RFS, to hide the cost of remote I/O by overlapping it with the application's computation. A previous study [14] showed that ABT is especially useful for slower file systems and hence is an excellent choice for use in I/O to remote file systems. We also optimize RFS performance with ABT through two schemes for temporary local staging during the application's run. Lastly, we provide efficient support for noncontiguous I/O in RFS through portable decoding of recursive datatypes using MPI-2 features.

The rest of this paper is organized as follows. Section 2 reviews previous remote I/O efforts and ABT. Section 3 describes the design and implementation of RFS in detail. Section 4 presents the experimental results obtained with the RFS implementation. Section 5 discusses optimization of reads and failure handling. Section 6 concludes the paper.

## 2 Related Work

### 2.1 Remote I/O Support for Scientific Workloads

Traditional wide-area distributed file systems such as Network File System (NFS) [2] and the Andrew File System (AFS) [16] provide convenient and often transparent interfaces for access to remotely located file systems. However, these file systems were originally designed and implemented as general-purpose distributed file systems whose main purpose is efficient sharing of files in distributed systems, and thus their performance is not optimized for large-scale scientific workloads.

For high-performance transfer of large-scale data, several specialized tools have been designed [3, 4, 9, 12, 18, 21]. Among them, RIO (Remote I/O) [9] was an early effort that provided a preliminary design and proof-of-concept implementation of remote file access in ROMIO. RIO's client-server architecture ensured implementation portability by exploiting the intermediate ADIO (Abstract Device I/O) [19] layer in ROMIO, which hides the details of different file system implementations. RFS is also implemented at the ADIO layer and eliminates several limitations of RIO. For example, RIO used dedicated "forwarder nodes" for message aggregation and asynchronous I/O. Extra processors, however, are not always available or convenient to use [14]. Thus, RFS removes such a requirement. Also, RIO seriously restricts the relationship between the numbers of client and server processes, while RFS can work with any number of processes on both sides. Finally, RFS removes RIO's dependency on the Globus Toolkit [8] and allows users to choose their own communication protocol. These points are discussed further in Section 3.

As another effort to support remote I/O in an I/O library, active buffering was integrated with compressed data migration in the Panda parallel I/O library [12]. A variant of nondedicated I/O processors was used for data reorganization and client-side compression, and dedicated migration processors were used for active buffering and migration. With a typical parallel execution setup, this approach reduces both visible I/O time and migration time significantly. But like RIO, dedicated processors are required.

GASS (Global Access to Secondary Storage) [4] provides remote file access services that are optimized according to several I/O patterns common in high-performance Grid computation. Examples of these I/O patterns include multiple readers reading a common file concurrently and multiple writers writing to a file in an append only manner (i.e. a log file). Our work, instead, addresses more general workloads, with the focus on optimizing write-intensive workloads through latency hiding. Although RFS is expected to work well with most of I/O patterns addressed by GASS, some of them can be further optimized through prefetching and caching. Section 5 discusses these issues.

GridFTP [3] is a high-performance, secure, robust data transfer service in the Data Grid [6] infrastructure. GridFTP extends conventional FTP to meet high-performance data transfer requirements in Grid computing [8]. The enhanced features of GridFTP include parallel and striped transfer, partial file transfer, secure channels, and TCP buffer size control. Although not an I/O library, GridFTP offers the APIs required to build client and server codes and can be

used as a means of data transfer for RFS.

Kangaroo [18] is a wide-area data movement system, designed to provide data transfer services with high availability and reliability. Both RFS and Kangaroo optimize remote output operations by staging the data locally and transferring them in the background. However, our work differs from Kangaroo in that we view the remote I/O problem from the MPI-IO perspective and thus address collective I/O, noncontiguous I/O, and MPI-IO consistency issues in the remote I/O domain, while Kangaroo focuses on the more basic and lower level remote I/O solutions. Also, Kangaroo relies on disk staging only, while RFS performs hybrid staging that uses both disks and available memory through ABT. Kangaroo adopts a chainable architecture, consisting of servers that receive data and disk-stage them locally and movers that read the staged data and send them to another server. This is especially useful if links between the two end points are slow or down. Like GridFTP, Kangaroo can be used to transfer data for RFS.

### 2.2 Active Buffering with Threads

Active buffering [14] reduces apparent I/O cost by aggressively caching output data using a hierarchy of buffers allocated from the available memory of the processors participating in a run and writing the cached data in the background after computation resumes. Traditional buffering aggregates small or noncontiguous writes into long, sequential writes, to speed them, but active buffering tries instead to completely hide the cost of writing. Active buffering has no hard buffer space requirement; it buffers the data whenever possible with whatever memory available. This scheme is particularly attractive for applications with periodic writes because in-core simulations do not normally reread their output in the same run. Thus, once output data are buffered in memory, computation can resume before the data actually reach the file system. Also, computation phases are often long enough to hide the cost of writing all the buffered output to disk. Unlike asynchronous writes provided by the file system, active buffering is transparent to users and allows user code to safely rewrite the output buffers right after a write call. Active buffering can also help when asynchronous I/O is not available.

Active buffering originally used dedicated processors for buffering and background I/O [12]. Later, active buffering with threads [14] was proposed for I/O architectures that do not use dedicated I/O processors, such as ROMIO. In ABT, data are still buffered using available memory, but the background I/O is performed by a thread spawned on each processor. Local I/O performance obtained from the ABT-enabled ROMIO shows that even without dedicated I/O processors, active buffering efficiently hides the cost of periodic output, with only a small slowdown from concurrent computation and I/O [14].

## 3 Design

As mentioned earlier, RFS exploits the intermediate ADIO layer in ROMIO for portability. ADIO defines a set of basic I/O interfaces that are used to implement more complex, higher-level I/O interfaces such as MPI-IO. For each supported file system, ADIO requires a separate implementation (called a "module") of its I/O interfaces. A generic implementation is also provided for a subset of ADIO functions. When both implementations exist, either the generic function is called first, and it may in turn call the file system-specific function, or the file system-specific function is directly called.

RFS has two components, a client-side RFS ADIO module and a server-side request handler. On the client where the application is running, remote I/O routines are placed in a new ADIO module also called RFS. When called, RFS functions communicate with the request handler located at the remote server to carry out the requested I/O operation. On the server where the remote file system resides, the request handler is implemented on top of ADIO. When it receives I/O requests from the client, the server calls the appropriate ADIO call for the local file system at the server. Figure 1 illustrates this architecture.

More detail on the design and implementation of RFS is presented below.

### 3.1 RFS ADIO Module

The goal of the RFS project is to provide a simple and flexible implementation of remote file access that minimizes data transfer, hides data transfer costs through asynchronous operation, and supports the MPI-IO consistency semantics. To this end, we implemented the following basic RFS ADIO functions:

- `RFS_Open`[1], `RFS_Close` - open and close a remote file.
- `RFS_WriteContig`, `RFS_ReadContig` - write and read a contiguous portion of an open file.
- `RFS_WriteNoncontig`, `RFS_ReadNoncontig` - write and read a noncontiguous portion of an open file.

These RFS functions take the same arguments as do the corresponding ADIO functions for other file system modules. One requirement for `RFS_Open` is that the file name contain the host name where the remote file system resides

[1] The prefix `RFS` denotes an RFS-specific function. Generic ADIO functions start with the prefix `ADIO`.

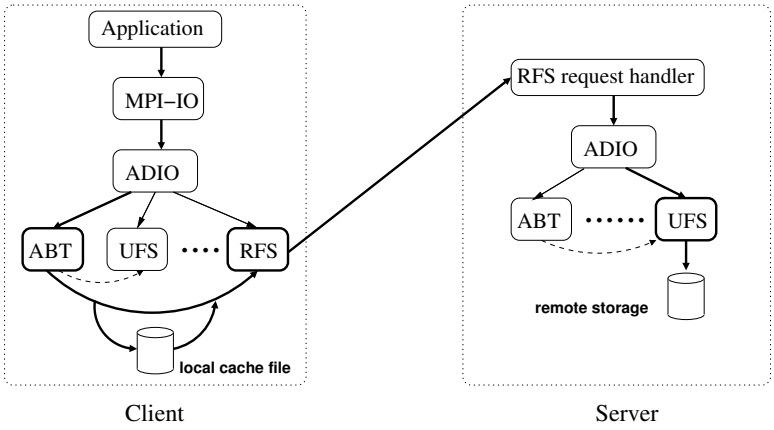



**Figure 1. RFS architecture. The bold arrows show the data flow for an ABT-enabled RFS operation that writes the data to the Unix file system on the remote server.**

and the host port number where the server request handler listens. For example, if we need to access the file system on elephant.cs.uiuc.edu through port 12345, we use the prefix "`rfs:elephant.cs.uiuc.edu:12345:`" before the file name.[2]

The RFS implementation of the remaining functions required for an ADIO module can be divided into three categories. First, some ADIO functions have a generic implementation that calls `ADIO_WriteContig`, `ADIO_WriteNoncontig`, `ADIO_ReadContig`, or `ADIO_ReadNoncontig`. With the RFS implementation of those functions, the ADIO functions that have a generic implementation can still be used without any changes. For example, `ADIO_WriteColl`, an ADIO function for collective writes, can use the RFS implementation of `ADIO_WriteContig` or `ADIO_WriteNoncontig` for all data transfer. Second, like the `seek` operation in ordinary file systems, some ADIO function calls have no discernible effect until a subsequent call is made. In order to reduce network traffic, these ADIO function calls can be deferred and piggybacked onto later messages. For such functions, RFS provides a simple client-side implementation that checks for errors and returns control immediately to the application. For example, when `ADIO_Set_view` is called at the client by `MPI_File_set_view` to determine how data will be stored in a file, the client implementation first checks for errors and returns. Then RFS waits until the next read or write operation on that file and sends the view information to the server together with the I/O operation. The appropriate implementation of `ADIO_Set_view` is chosen by the server based on its local file system. The user can also choose to defer `RFS_Open` until the first read or write operation on the file by passing a hint, and can defer `RFS_Close` until all the buffered write operations are completed. Third, the ADIO functions that cannot be implemented in the previous two ways have their own implementation in RFS (e.g., `ADIO_Delete` to delete files with a given file name).

Providing specialized noncontiguous I/O support is key in local I/O but is even more important in the remote I/O domain because latencies are higher. For noncontiguous file access, ROMIO uses *data sieving* [20] to avoid noncontiguous small I/O requests when support for noncontiguous I/O is not available from the ADIO implementation. For noncontiguous reads, ROMIO first reads the entire extent of the requested data and then selects the appropriate pieces of data. For writes, ROMIO reads the whole extent into a buffer, updates the buffer with pieces of output data, and writes the whole buffer again. This approach makes sense in the local I/O environment where the cost of moving additional data is relatively low. However, in the network-constrained environment of remote I/O, reducing the amount of data to be moved is just as important as reducing the number of operations.

RFS's specialized implementation can significantly reduce the amount of data transferred in read and write cases. This is especially useful in the write case because for a noncontiguous write, we would be required to read this large region from across the network, modify it, and write it back. The RFS server can use data sieving locally to the server to optimize local data access.

For noncontiguous writes, RFS packs the data to be written using `MPI_Pack` and sends the packed data as the `MPI_PACKED` datatype to the server, to reduce the amount of data transferred. Similarly, for noncontiguous reads, data are first read into contiguous buffer space on the server, sent back to the client, and unpacked by the client using the user-specified datatype. In both cases, the datatype that describes how the data should be stored in memory (called the "buffer

[2] ROMIO's file naming convention is to use the prefix "`<file system name>:`" to specify the file system to be used.

```
RFS_handle RFS_Make_connection(char *host, int port);
int RFS_Writen(RFS_handle handle, char *buf, int count);
int RFS_Readn(RFS_handle handle, char *buf, int count);
int RFS_Close_connection(RFS_handle handle);
```

**Figure 2. C-style communication interface prototypes used in RFS.**

datatype") need not be transferred between the client and server. For example, it is not important for noncontiguous write operations whether or not the data are in packed form, as long as they have the correct number of bytes to write. However, file view information must be sent to the remote server, to describe how data should be stored on disks on the server. The file view information contains a datatype (called the "filetype"), which can be a recursively defined derived datatype. To portably pass a complex recursive datatype to the remote server, we use `MPI_Type_get_envelope` and `MPI_Type_get_contents` in MPI-2 for datatype decoding. Using these two functions, we perform a preorder traversal of the given recursive datatype and pack the results into a buffer string, which is sent to the remote server. The server reads the buffer string and recursively recreates the original derived datatype. The file view information is sent once whenever there is an update (e.g., `MPI_File_set_view` is called). Ching et al. [7] took a similar approach to pass datatypes between client and server in the Parallel Virtual File System (PVFS) [5] on a single site, but we believe that this is its first use in remote I/O.

As briefly mentioned in Section 2, RFS requires no specific communication protocol for data transfer. Instead, RFS defines four primitive communication interface functions (Figure 2) that implement simple connection-oriented streams, and allows users to choose a communication protocol for which an implementation of the four interface functions is available. For example, users can pick GridFTP for its secure data transfer or can use a hybrid protocol of TCP and UDP, such as Reliable Blast UDP [10], for better transfer rates. The current implementation uses TCP/IP.

### 3.2 Integration with the ABT Module

ABT is implemented as an ADIO module that can be enabled in conjunction with any file system-specific ADIO module (Figure 1). For example, when ABT is enabled with the module for a particular file system, read and write requests to ADIO are intercepted by the ABT module, which buffers the data along with the description of the ADIO call and then returns control to the application.[3] In parallel, the background thread performs I/O on the buffered data using the appropriate ADIO file system module functions. At the user's request, ABT can intercept and defer file close operations until the buffered writes for that file are completed. Thanks to the stackable ABT module, the integration of ABT and RFS required few code changes.

To optimize RFS performance with ABT, we augmented ABT with two temporary local disk staging schemes. First, when there is not enough memory to buffer the data for an I/O operation, ABT does not wait until a buffer is released, because that may be very slow with RFS. Instead, ABT immediately writes the data into a local "cache" file created in the fastest file system available on the client ("foreground staging"). The description of the I/O operation is still buffered in memory, along with the size and the offset of the data in the cache file. For each data buffer that is ready to write out, the background I/O thread first checks the location of the data. If the data are on disk, the thread allocates enough memory for the staged data and reads the data from the cache file. Once the data are in memory, the requested I/O operation is performed.

Second, to reduce the visible I/O cost even more, during each computation phase, we write some of the memory-buffered data to the cache file in the background, to procure enough memory space for the next set of output requests ("background staging"). For that purpose, it is helpful to know how much time we have before the next output request is issued and how much data will be written during the next set of output requests. RFS can obtain such information by observing the application, or users can provide it as hints. This scheme is especially well suited for typical simulation codes that write the same amount of data at regular intervals. If the amount of data or the interval between two consecutive output phases changes over time, we can still use the average values as estimates. However, we want to avoid unnecessary background staging, because staged data will have to be read back into memory before being sent to the server, thus possibly increasing the overall execution time.

We use a simple performance model to determine how much data to stage in the background before the next output phase begins. Suppose that the maximum available buffer size is $ABS_{max}$ bytes, the currently available buffer size is $ABS_{now}$ bytes, the remaining time till the next output is $I$ seconds, and the expected size of the next output is $N$ bytes. If $ABS_{now}$ is smaller than $\min(ABS_{max}, N)$, which is the amount of buffer space needed, we perform

[3] Instead of allocating a monolithic buffer space at the beginning of an application run, ABT uses a set of small buffers, dynamically allocated as needed. Each buffer has the same size, preset according to file system performance characteristics. If the data to be buffered are larger than this buffer size, they are further divided and stored in multiple buffers.

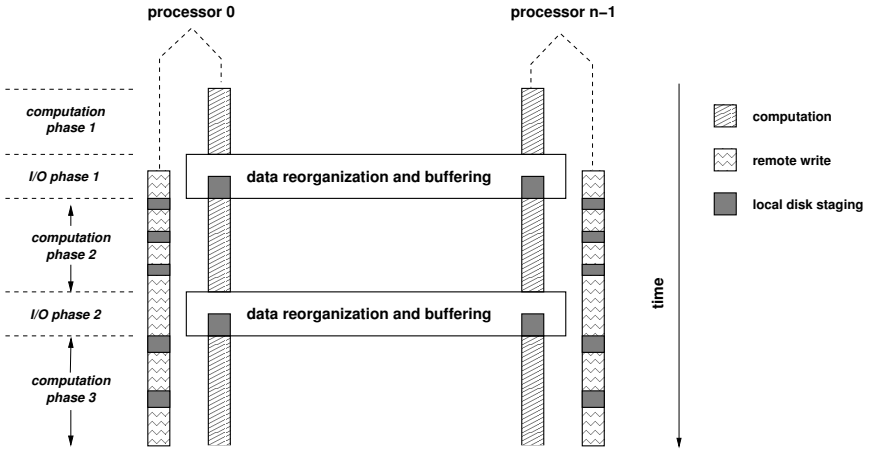



**Figure 3. Sample execution timeline of an application run at the client, when using RFS with ABT. Both foreground and background staging are being performed. The foreground staging is visible to the application, while the background staging is hidden by the concurrent computation.**

background staging. If we write $x$ bytes to the cache file before the next output phase begins, it will take an estimated $\frac{x}{B_w}$ seconds, where $B_w$ is the write bandwidth of the chosen file system. Also, for $I - \frac{x}{B_w}$ seconds, remote writes will transfer $B_x(I - \frac{x}{B_w})$ bytes of data, where $B_x$ is the remote write bandwidth that also considers the write time at the server and the response time. Therefore, with this scheme, the available buffer size $I$ seconds from now will be $ABS_{now} + x + B_x(I - \frac{x}{B_w})$, and this value should be equal to or greater than $\min(ABS_{max}, N)$. Solving this equation for $x$, we get

$$x \geq \frac{\min(ABS_{max}, n) - ABS_{now} - I \cdot B_x}{1 - \frac{B_x}{B_w}} \qquad (1)$$

This is a rough estimate; for example, it assumes that all the data to be transferred are buffered in memory and does not consider the read cost for the data buffered in the local cache file. The current implementation spreads the staging of $x$ bytes over the current computation phase (i.e., staging a fraction of $x$ bytes before writing each buffer remotely), adjusting $x$ if $B_x$ changes over time. Figure 3 depicts a sample execution timeline of an application using RFS with ABT at the client, with these optimizations in place.

### 3.3 RFS Request Handler on the Server

An RFS request handler on a server is an MPI code that can run on multiple processors and whose functionality is relatively simple compared with that of the client RFS module. The server receives the I/O requests from client processes, performs them locally, and transfers back to the client an error code and the data requested (for reads). Currently clients are mapped to server processes in a round-robin manner. If a single server process needs to handle multiple clients, it spawns a thread for each client that it has to serve, and handles the I/O requests concurrently. Although concurrent I/O operations on a common file could be requested on a server process, it need not be concerned about the file consistency semantics, as MPI by default requires the user to be responsible for consistency when having multiple writers on a common file.

## 4 Experimental Results

Our experiments used Chiba City, a Linux cluster at Argonne National Laboratory, for the client-side platform. Chiba has 256 compute nodes, each with two 500 MHz Pentium III processors, 512 MB of RAM, and 9 GB of local disk space. All the compute nodes are connected via switched fast ethernet. For the servers, we used two Linux PCs at different locations. Elephant is at the University of Illinois, with a 1.4 GHz Pentium 4 processor and 512 MB of RAM. Tallis is at Argonne, with a 1 GHz Pentium III processor and 256 MB of RAM. The MPICH2 implementation of MPI is used on all the platforms.

Table 1 shows the network and disk bandwidth measured between Chiba and each server with concurrent senders/writers. Both network and disk throughput are fairly stable and do not vary much as the number of senders/writers increases. As in a typical remote I/O setup, the network between Chiba and Elephant is slower than Elephant's local disk. However, Tallis has a very slow disk, even slower than the network connection to Chiba, thus simulating an environment with a high-performance backbone network.

We first measured the apparent I/O throughput with different file system configurations. The goal was to show how

**Table 1. Aggregate network and disk bandwidth with each server. The numbers in parentheses show the 95% confidence interval.**

| | No. Procs | 4 | 8 | 12 | 16 |
|---|---|---|---|---|---|
| Elephant | network | 11.7325 ($\pm$ 0.035) MB/s | 11.75 ($\pm$ 0.018) MB/s | 11.74 ($\pm$ 0.025) MB/s | 11.69 ($\pm$ 0.10) MB/s |
| | disk | 15.49 ($\pm$ 0.16) MB/s | 15.46 ($\pm$ 0.29) MB/s | 15.31 ($\pm$ 0.13) MB/s | 15.07 ($\pm$ 0.18) MB/s |
| Tallis | network | 11.75 ($\pm$ 0.010) MB/s | 11.75 ($\pm$ 0.020) MB/s | 11.77 ($\pm$ 0.020) MB/s | 11.77 ($\pm$ 0.0049) MB/s |
| | disk | 1.95 ($\pm$ 0.0092) MB/s | 1.95 ($\pm$ 0.0046) MB/s | 1.95 ($\pm$ 0.0060) MB/s | 1.95 ($\pm$ 0.0068) MB/s |

close the RFS throughput can be to the performance of the bottleneck in the remote I/O path and also how efficiently RFS with ABT can hide the visible cost of remote writes by overlapping them with the subsequent computation. For an application that writes periodically, we used a 3-D parallel Jacobi relaxation code. Jacobi is iterative; in each iteration, it updates the values of cells in a grid with the average value of their neighbors from the previous iteration. The code writes its intermediate "snapshot" to disk after a user-controllable number of iterations have passed since the last output.

The 3-D global array for the Jacobi code is distributed across the processors using an HPF-style `(BLOCK, BLOCK, BLOCK)` distribution, and for each snapshot, this array is collectively written to a common file in row-major order by ROMIO.[4] Because of this "nonconforming" distribution, each collective write involves data reorganization between processors. We used 4, 8, 12, and 16 processors, and the number of partitions in each dimension of the global array was determined by `MPI_Dims_create`. We fixed the amount of data on each processor at 32 MB (a $128 \times 128 \times 256$ double array). For example, with 16 processors, the number of partitions in each dimension is [4, 2, 2], and thus the global array is a $512 \times 256 \times 512$ double array. All the participating processors write to the common file (except for one configuration), each responsible for writing 32 MB of data per snapshot.[5]

We used six file system configurations:

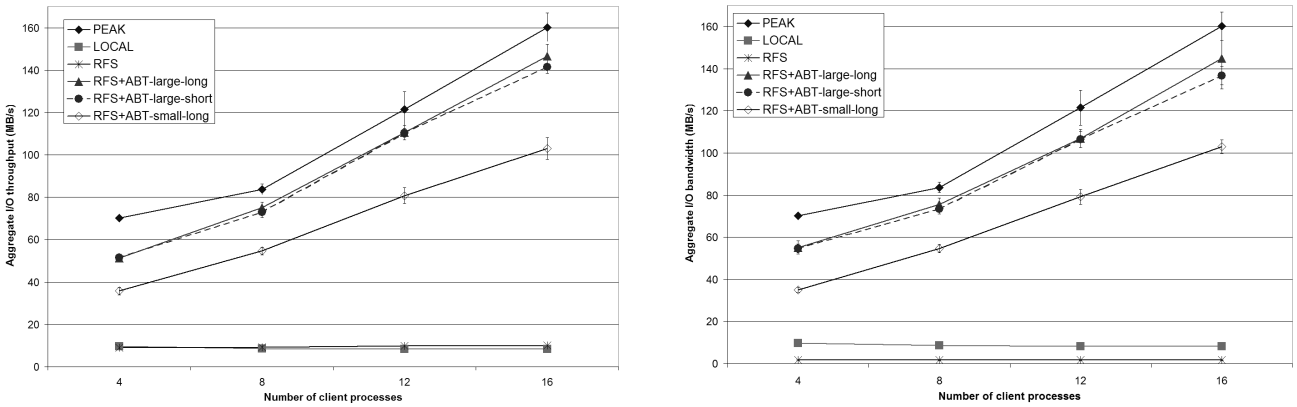
- PEAK: local write without actually writing the data to disks, simulating infinitely fast disks (using ROMIO's TESTFS ADIO module). The visible write cost includes only the communication for data reorganization, and we used this as the theoretical peak performance.
- LOCAL: local file system write.[6]
- RFS: RFS write without ABT. The entire data transfer cost is visible here.
- RFS+ABT-large-long: RFS write with ABT. The total buffer size on each processor is equal to or larger than the amount of a single output operation that the processor is responsible for ("large"). The length of a computation phase is long enough to hide an entire remote write operation ("long"). Thus, no local staging (either foreground or background) happens. Here we set the buffer size to 32 MB.
- RFS+ABT-large-short: The same configuration as the previous one, but the length of a computation phase is shorter than the time required to transfer one snapshot. This situation may trigger background staging to obtain enough buffer space for the next snapshot. We controlled the length of each computation phase so that 45–75 % of each snapshot could be transferred.
- RFS+ABT-small-long: The same as RFS+ABT-large-long, except that the total buffer size is not big enough to hold a single snapshot. This will trigger foreground staging, whose cost is visible. We set the buffer to 16 MB.

Figure 4 shows the aggregate application-visible I/O throughput (data size divided by the response time for the I/O calls) measured with the two remote I/O setups and the file system configurations described above. The visible I/O cost for "LOCAL" and "RFS" includes the time for the output data to reach the destination file system (without an explicit "sync") and the time to receive an error code for the write operation from the server. For the configurations with ABT, the visible I/O cost is the cost of local memory buffering and foreground staging, if performed. For each run, we

[4] In collective I/O, all the processors cooperatively participate to carry out an efficient I/O. Global data layouts in memory and on disk are used to optimize the I/O requests, forming fewer, larger sequential accesses instead of having many small I/O requests.

[5] Although we used the same number of writers as the number of processors for test purposes, the number of RFS writers should be carefully selected, considering the aggregate network and disk performance with concurrent senders/writers, because too many writers can hurt performance. ROMIO allows users to control the number of writers (called "aggregators").

[6] Since a shared file system on Chiba was not available at the time of these experiments, we simulated a shared file system by having one aggregator gather and reorganize all the data and write to its local disk. Many clusters use NFS-mounted shared file systems, whose performance is often much lower than that of our simulated shared file system.




(a) Between Chiba and Elephant

(b) Between Chiba and Tallis

**Figure 4. Aggregate application-visible write bandwidth with different file system configurations.**

executed the Jacobi code with five iterations, writing up to 2.5 GB of data remotely; the numbers in the graph were averaged over five or more runs. The error bars show the 95% confidence interval.

The "PEAK" I/O throughput increases as the number of processors and the amount of data grow, reaching 160.2 MB/s with 16 processors, although it does not scale up well. Since this configuration does not involve disk operations, the performance is limited by the message passing performance on Chiba done via the fast ethernet. The "LOCAL" I/O throughput is up to 9.8 MB/s and does not scale up because we used only one writer.

Between Chiba and Elephant, the network is the main performance-limiting factor for remote I/O. Thus, as shown in Figure 4(a), the RFS throughput reaches 10.1 MB/s with 16 processors, about 86% of the network bandwidth between the two platforms. The gap between the RFS and network throughput occurs because RFS writes also include the response time for disk writes at the server and the time to transfer the error code back to the client. Our tests with RFS reads yielded similar results. As the number of writers increases, the aggregate RFS throughput also increases slightly, because the data reorganization throughput increases, too. In this setup, RFS performance is comparable to or even higher than the local I/O performance.

With ABT in place, however, the visible write throughput increases significantly because ABT efficiently hides the remote I/O cost. In "RFS+ABT-large-long," where we have enough buffer space and long computation phases, the visible I/O throughput reaches 146.7 MB/s with 16 processors, about 92% of the theoretical peak, a factor of 14.5 improvement over the RFS performance and a factor of 17.7 improvement over the local I/O performance. The gap between the peak and the RFS performance is due mainly to the cost of copying data to the active buffers and the slowdown caused by concurrent foreground buffering and background remote I/O.

When the computation phase is not long enough to hide an entire remote output operation ("RFS+ABT-large-short"), the visible I/O throughput is still comparable to the throughput obtained with long computation phases. In our experiments, the difference in throughput does not exceed 4% of the long computation phase throughput, proving that background staging with our performance model can procure enough buffer space for the next snapshot.

When the total buffer space is smaller than the size of a snapshot ("RFS+ABT-small-long"), RFS has to perform foreground staging, whose cost is completely visible. For our tests, we used `fsync` to immediately flush the staged data to disk, because we wished to see the effect of local staging of larger data: if the amount of data to be staged is small, as in our experiments, the staged data can fit in the file cache, producing a very small local staging cost. Even with `fsync`, RFS with ABT can still improve the remote write performance significantly, reaching 103.1 MB/s with 16 processors, an improvement of a factor of 10.2 over RFS alone and a factor of 12.4 over local write performance. Without `fsync`, we obtained performance very close to that with 32 MB of buffer.

Figure 4(b) shows the I/O bandwidth obtained between Chiba and Tallis. Here, the very slow disk on Tallis is the performance bottleneck, limiting the RFS performance to less than 2 MB/s. We obtained up to 1.7 MB/s of RFS write throughput, roughly 87.1% of the disk bandwidth on Tallis. Read tests produced similar results. Nevertheless, the performance of RFS with ABT between Chiba and Tallis is close to the performance between Chiba and Elephant, making the performance improvement even more dramatic. For example, the aggregate visible I/O throughput with RFS+ABT-large-long reaches 145.0 MB/s with

**Table 2. The amount of data staged at the client in RFS+ABT-large-short.**

| | No. Procs | 4 | 8 | 12 | 16 |
|---|---|---|---|---|---|
| Chiba to Elephant | foreground | 0.0 MB (0.0%) | 0.0 MB (0.0%) | 0.8 MB (0.05%) | 1.6 MB (0.08%) |
| | background | 168.0 MB (32.8%) | 322.4 MB (31.5%) | 571.2 MB (37.2%) | 798.4 MB (39.0%) |
| Chiba to Tallis | foreground | 12.0 MB (2.3%) | 26.0 MB (2.5%) | 46.4 MB (3.0%) | 71.0 MB (3.5%) |
| | background | 317.0 MB (61.9%) | 614.0 MB (60.0%) | 961.6 MB (62.6%) | 1320.0 MB (64.5%) |

16 processors, about 86.8 times higher than the RFS write throughput and about 8.3 times higher than the local write throughput. With a 16 MB buffer, 103.0 MB/s throughput was achieved with 16 processors and `fsync`, a factor of 61.6 improvement over RFS writes and a factor of 12.4 improvement over local writes. The reason we could still obtain excellent visible I/O performance with this slow remote file system is that the client buffers data, and thus, with the help from background staging, the buffering cost does not vary much with different servers.

We cannot easily compare the performance of RFS directly with that of RIO. RIO was a one-time development effort, so today RIO depends on a legacy communication library, making it impractical to run RIO in our current environment. Also, the experiments presented in the original RIO paper [9] were conducted in a *simulated* wide-area environment, where the RIO authors partitioned a parallel platform into two parts and performed TCP/IP communication between them, instead of using a real wide-area environment as we have. Moreover, the RIO authors measured the sustained remote I/O bandwidth with a *parallel* file system at the server for blocking and nonblocking I/O (equivalent to our I/O operations without and with ABT), while we measured the visible I/O bandwidth with a *sequential* Unix file system at the server.

However, we can still speculate on the difference in remote I/O performance with RFS and with RIO. According to the RIO authors, RIO can achieve blocking remote I/O performance close to the peak TCP/IP performance with large messages. Our experiments show that remote I/O without ABT can achieve almost 90% of the peak TCP/IP bandwidth even with a sequential file system at the other end. With smaller messages, however, RIO's blocking I/O performance dropped significantly, because of the communication overhead with RIO's dedicated forwarders. Since all remote I/O traffic with RIO goes through the forwarders, a single I/O operation between a client and a server process involves four more messages than with RFS, two outbound and incoming messages between the client process and the client-side forwarder and two between the server process and the server-side forwarder. These can cause significant overhead for an application with many small writes that uses RIO. RFS, on the other hand, does not use intermediate forwarders and let clients directly communicate with servers, effectively reducing the message traffic compared to RIO. For this reason, we expect RFS to be more efficient than RIO in many situations.

To test the performance model with the background staging, we measured the amount of data staged both in the foreground and the background at the client in RFS+ABT-large-short (Table 2). The numbers were averaged over five or more runs; the numbers in parentheses are the percentage of staged data out of the total data in four snapshots.[7] If our performance model accurately predicts the amount of data that should be staged, then there should be no foreground staging, because the total buffer size is same as the size of a snapshot. The numbers obtained confirm this claim. In both setups, less than 4% of the output data are staged in the foreground.

Also, an accurate model should minimize the amount of data staged in the background; otherwise, unnecessary staging will make the overall transfer longer. However, it is difficult to measure the exact amount of unnecessarily staged data because the amount of data transferred during each computation phase can vary as a result of network fluctuation and slowdown from multithreading. In "RFS+ABT-large-short," we roughly estimated the length of each computation phase to be long enough to transfer over the network about 70–75% of a snapshot for the Chiba-Elephant setup and 45–50% of a snapshot for the Chiba-Tallis setup. Thus, in theory, 25–30% of a snapshot for the Chiba-Elephant setup and 50–55% of a snapshot for the Chiba-Tallis setup should be staged in the background, to minimize the visible write cost for the next snapshot. When background staging is in place, however, smaller amounts of data than estimated above may be transferred during a computation phase, because background staging takes time. Also, since the unit of staging is an entire buffer, often we cannot stage the exact amount of data calculated by the model. Thus, the amount of data staged in the background for each snapshot should be larger than the portion of a snapshot that cannot be transferred during a computation phase with RFS alone. Our performance numbers show

[7] Among the five snapshots in each run, the first cannot be staged in the foreground, and the last cannot be staged in the background in this configuration.

**Table 3. The computation slowdown caused by concurrent remote I/O activities.**

| | No. Procs | 4 | 8 | 12 | 16 |
|---|---|---|---|---|---|
| Chiba to Elephant | RFS+ABT-large-long | 2.33% | 0.68% | 0.53% | 1.41% |
| | RFS+ABT-large-short | 6.24% | 3.62% | 1.97% | 2.46% |
| Chiba to Tallis | RFS+ABT-large-long | 5.67% | 5.25% | 5.35% | 2.11% |
| | RFS+ABT-large-short | 0.90% | 1.17% | 0.45% | 0.24% |

that 31–39% of the output for the Chiba-Elephant setup and 60–65% of the output for the Chiba-Tallis setup were staged in the background, slightly more than the estimated numbers above. Based on these arguments and our performance numbers, we conclude that the amount of unnecessarily staged data by RFS is minimal.

Finally, we measured how much these background remote I/O activities slow the concurrent execution of the Jacobi code through their computation and inter-processor communication. Table 3 shows the average slowdown of the computation phases with various configurations when the 32 MB buffer was used. All the measured slowdown was less than 7%, which is dwarfed by the performance gain from hiding remote I/O cost.

# 5 Discussion

## 5.1 Optimizing Remote Read Performance

This work focuses on optimizing remote write performance through ABT for write-intensive scientific applications. Reads are typically not a big concern for such applications, because they often read a small amount of initial data and do not reread their output snapshots during their execution. A restart operation after a system or application failure may read large amounts of checkpointed output, but restarts rarely occur. The current implementation of ABT requires one to flush the buffered data to the destination file system before reading a file for which ABT has buffered write operations.

However, applications such as remote visualization tools may read large remote data. The traditional approaches to hide read latency are to prefetch the data to be read and to cache the data for repetitive reads, and we are adding such extensions to ABT, using local disks as a cache. More specifically, we are providing a flexible prefetching interface through the use of hints, so that background threads can start prefetching a remote file right after an open call on that file. When a read operation on a file for which prefetching is requested is issued, RFS checks the prefetching status and reads the portion already prefetched locally, and the portion not yet prefetched remotely. Cached files can be read similarly, performing remote reads for the portions that are not cached locally. GASS [4] has facilities for prefetching and caching of remote files for reads. GASS transfers only entire files, however, an approach that can cause excessive data transfer for partial file access (e.g., visualizing only a portion of a snapshot). We instead aim to provide finer-grained prefetching and caching that use byte ranges to specify the data regions to be prefetched and cached. Our extensions will comply with the MPI-IO consistency semantics.

## 5.2 Handling Failures

Through ABT and hints, the current RFS implementation allows the user to defer file open, write, and close calls to reduce the network traffic and response time. The original call to such functions returns immediately with a success value, and if an error occurs during a deferred operation, the user will be notified after the error. Thus, the user needs to be aware of the possibility of delayed error message as the cost of this performance improvement. If timely error notification is important, the user should avoid these options.

A failed remote open notification will be received when the following I/O operation on the specified remote file fails. A failed write notification can be delayed until a sync or close operation is called on the file. The MPI-IO standard says that `MPI_File_sync` causes all previous writes to the file by the calling process to be transferred to the storage device (`MPI_File_close` has the same effect), so delaying write error notification until a sync or close operation does not violate the standard. If the user chooses to defer a file close, too, and a write error occurs after the original close operation returns, then the error can be delayed until `MPI_Finalize`.

Some RFS operations that are deferred by default, such as setting the file view and file seeks, can be checked for errors at the client and the user can be notified about such errors, if any, before the operations are executed at the server. Thus, they do not need separate failure handling.

# 6 Conclusions

We have presented an effective solution for direct remote I/O for applications using MPI-IO. The RFS remote file

access component has a simple and flexible I/O architecture that supports efficient contiguous and noncontiguous remote accesses. Coupling this with ABT provides aggressive buffering for output data and low-overhead overlapping of computation and I/O. Our local data staging augmentation to ABT further enhances ABT's ability to hide true I/O latencies. Our experimental results show that the write performance of RFS without ABT is close to the throughput of the slowest component in the path to the remote file system. However, RFS with ABT can significantly reduce the remote I/O visible cost, with throughput up to 92% of the theoretical peak (determined by local interconnect throughput) with sufficient buffer space. With short computation phases, RFS still reduces visible I/O cost by performing a small amount of background staging to free up sufficient buffer space for the next I/O operation. The computation slowdown caused by concurrent remote I/O activities is under 7% in our experiments and is dwarfed by the improvements in turnaround time.

As discussed in the previous section, we are currently enhancing ABT for RFS reads, by introducing prefetching and caching. Future work includes experiments with alternative communication protocols and parallel server platforms.

## Acknowledgments

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38. This research was also supported through faculty start-up funds from North Carolina State University and a joint faculty appointment from Oak Ridge National Laboratory.

## References

[1] NCSA HDF home page. http://hdf.ncsa.uiuc.edu.

[2] NFS: Network File System protocol specification. RFC 1094.

[3] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, and S. Tuecke. Data management and transfer in high performance computational grid environments. *Parallel Computing Journal*, 28(5):749–771, 2002.

[4] 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 Workshop on Input/Output in Parallel and Distributed Systems*, 1999.

[5] P. Carns, W. L. III, R. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the Annual Linux Showcase and Conference*, 2000.

[6] A. Chervenak, I. Foster, C. Kesselman, S. Salisbury, and S. Tuecke. The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.

[7] A. Ching, A. Choudhary, W.-K. Liao, R. Ross, and W. Gropp. Efficient structured access in parallel file systems. In *Proceedings of the International Conference on Cluster Computing*, 2003.

[8] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[9] I. Foster, D. Kohr, Jr., R. Krishnaiyer, and J. Mogill. Remote I/O: Fast access to distant storage. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems*, 1997.

[10] E. He, J. Leigh, O. Yu, and T. DeFanti. Reliable Blast UDP: Predictable high performance bulk data transfer. In *Proceedings of the International Conference on Cluster Computing*, 2002.

[11] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 1994.

[12] J. Lee, X. Ma, M. Winslett, and S. Yu. Active buffering plus compressed migration: An integrated solution to parallel simulations' data transport needs. In *Proceedings of the International Conference on Supercomputing*, 2002.

[13] J. Li, W.-K. Liao, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of SC2003*, 2003.

[14] 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*, 2003.

[15] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Standard*. 1997.

[16] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. Andrew: A distributed personal computing environment. *Communications of ACM*, 29(3):184–201, 1986.

[17] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies*, 2002.

[18] D. Thain, J. Basney, S.-C. Son, and M. Livny. The Kangaroo approach to data movement on the Grid. In *Proceedings of the Symposium on High Performance Distributed Computing*, 2001.

[19] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation*, 1996.

[20] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[21] J. Weissman. Smart file objects: A remote file access paradigm. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems*, 1999.