Remote I/O: Fast Access to Distant Storage

Ian Foster David Kohr, Jr.* Rakesh Krishnaiyer

Jace Mogill†

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

Abstract

As high-speed networks make it easier to use distributed resources, it becomes increasingly common that applications and their data are not colocated. Users have traditionally addressed this problem by manually staging data to and from remote computers. We argue instead for a *remote I/O* paradigm in which programs use familiar parallel I/O interfaces to access remote filesystems. In addition to simplifying remote execution, remote I/O can improve performance relative to staging by overlapping computation and data transfer or by reducing communication requirements. However, remote I/O also introduces new technical challenges in the areas of portability, performance, and integration with distributed computing systems. We propose techniques designed to address these challenges and describe a remote I/O library called RIO that we are developing to evaluate the effectiveness of these techniques. RIO addresses issues of portability by adopting the quasi-standard MPI-IO interface and by defining a RIO device and RIO server within the ADIO abstract I/O device architecture. It addresses performance issues by providing traditional I/O optimizations such as asynchronous operations and through implementation techniques such as buffering and message forwarding to offload communication overheads. Microbenchmarks and application experiments demonstrate that our techniques can improve turnaround time relative to staging.

1 Introduction

Improvements in networking and software infrastructure are making it easier for programmers to execute programs at remote sites and to write programs that use resources at multiple locations. One consequence of remote execution is that a program may be geographically separated from the files that it accesses. This separation can significantly increase conceptual and temporal overheads in program development and execution. Ideally, we would like to enable programs to access data in a manner independent of data and program location. In our experience, the key challenges that must be addressed before we can provide this capability are portability (across different networks and filesystems), performance (in potentially high-latency, low-bandwidth, heterogeneous networks), and integration into distributed computing environments.

Historically, the high-performance computing community has achieved remote data access by manually *staging* input data from its home filesystem to the computer where a program is to execute; this process is then reversed for output data. However, this approach is clumsy, prevents overlapping of communication and computation, and can result in excessive data transfer in situations where a program accesses only part of a file. Distributed filesystems [18] also support remote data access, but performance and administrative problems often render them inappropriate for high-performance computing.

We propose an alternative approach to remote data access in which programs use *remote I/O libraries* to access files located on remote filesystems in a manner that is independent of physical location. In contrast to distributed filesystems, remote I/O libraries use parallel I/O interfaces and focus on high-performance transfer. We believe that this narrow focus can allow remote I/O libraries to meet requirements for performance, flexibility, and convenience without introducing undue complexity in their implementation.

As part of an investigation of the remote I/O concept, we have designed and implemented a prototype remote I/O library called RIO. RIO achieves portability by adopting the I/O interface defined by MPI-IO [6, 17] and by exploiting features of the ADIO abstract I/O device [24], providing a RIO device that translates ADIO calls into communications to remote RIO servers. Performance issues are addressed by the use of dedicated forwarder nodes, buffering, and support for asynchronous and collective operations. RIO uses the Nexus communication library [12] for client/server communication, hence providing access to configuration and security mechanisms provided by the Globus wide area computing toolkit [11].

We have performed experiments in a controlled multicomputer environment to evaluate the effectiveness of our techniques. Microbenchmarks demonstrate that RIO can

*Author's current affiliation: Silicon Graphics Inc., 2011 Shoreline Blvd., Mountain View, CA 94043.

†Author's current affiliation: NASA/Goddard Space Flight Center, Greenbelt, MD 20771.

drive networks at close to their peak performance; these experiments also allow us to quantify the benefits of optimizations such as asynchronous operations. Application experiments illustrate the feasibility of remote I/O in a representative application. In particular, we demonstrate enhanced performance relative to staging.

The ideas and results presented in this paper are preliminary, and represent just a first step towards high performance remote I/O. Nevertheless, we believe that the paper makes useful contributions in four related areas. Specifically, we

- introduce the concept of remote I/O, explain why it is important, and motivate its requirements;
- discuss networking issues that make remote I/O challenging, and propose library facilities that address these challenges;
- present experimental results that demonstrate the efficacy of our design techniques, and indicate where more work is needed; and
- show how to integrate a remote I/O library with mechanisms that support operation in a distributed environment.

2 The Remote I/O Problem

We first expand upon why remote I/O is important, discuss networking issues that remote I/O libraries must address, and review other approaches to the remote data access problem.

2.1 Motivation

One may wish to use remote computational or data resources because they provide a unique capability, such as a supercomputer or database, or simply because they are available: e.g., in a network of workstations, a load-sharing system such as Condor [16] can map tasks to idle resources. In either case, filesystems may be geographically separated from computers. This need to access "remote" filesystems arises frequently even within a single site: for example, it is often undesirable for all filesystems to be crossmounted on all machines, or even impossible to crossmount specialized filesystems such as tape archives. In addition, even when disks are crossmounted, performance may be poor when using conventional I/O techniques.

Programmers have traditionally resorted to staging techniques when a program and its data are not colocated. However, there can be significant advantages to having a uniform interface to local and nonlocal filesystems. As we discuss below in Section 2.3, this uniform interface can be provided in a number of ways. In the article, we focus on the use of user-level I/O libraries designed to provide high-performance access from parallel programs to remote parallel file systems. We call these libraries remote I/O libraries, and argue that they offer performance, flexibility, and convenience advantages relative to existing techniques such as manual staging and deployment of distributed filesystems.

Performance. Remote I/O can reduce total wall clock time by allowing overlapping of data transfer and computation. In a best-case situation, overlapping can reduce execution time by up to a factor of two. For example, a climate model that executes for 8 seconds per simulated day requires 8 hours to perform a 10-year simulation. If this model produces 8 MB output per day, then a 10-year simulation produces 29 GB of output data and would require an additional 6.4 hours to transfer that data over a 10 Mb/sec network. If, on the other hand, computation and communication could be completely overlapped, total turnaround time would be reduced from 14.4 to 8 hours.

In some situations, remote I/O can reduce total execution time even without overlap. For example, an OC12 ATM network provides 80 MB/sec peak throughput, significantly better than present-day commodity disk drives. Suppose a system with commodity disks and conventional serial filesystems is connected to a high-speed network, which in turn provides a datapath to a parallel filesystem. An application running on such a system could reduce the time spent on I/O by performing remote I/O on the parallel filesystem. Other researchers have shown that as network performance has improved, it has become feasible to assemble collections of networked workstations whose performance, including I/O, scales in the same manner as if they were tightly-coupled multicomputers [8, 25].

Flexibility. Remote I/O can provide a higher-level specification of I/O operations than does staging and hence permit greater flexibility in terms of how I/O is performed. For example, a program may need to access just selected components of remote data sets. If the identity of those data elements is computed during program execution, a staging approach often transfers more data than is necessary, wasting both disk and network resources. In contrast, a remote I/O library can choose whether to prefetch the entire dataset or transfer only those elements specifically requested by the application, depending on available resources. Hibbard et al. [13] prototyped the latter strategy in the I-WAY networking experiment, fetching data from a remote IBM SP data server only when a user zoomed in on a particular area within a virtual reality browser.

Convenience. Remote I/O allows programs to execute at remote sites without programmer management of data transfer. In contrast, staging can require that the user learn details of remote filesystems, transfer data among potentially complex directory hierarchies, translate data formats, and manage multiple copies of their datasets. Norman et al. [20] report that such issues were a major source of complexity in their distributed simulations of galactic collisions. A remote I/O library can automate all of these issues, including data format conversion if an I/O API is used that supports file accesses in terms of data types rather than characters. MPI-IO has this property. In a different regard, conversations with users reveal that some are uncomfortable leaving sensitive data in remote file systems, but are happy to transfer such data over networks to an application, perhaps over a secure network. Remote I/O makes this transfer possible without requiring that data be encrypted prior to writing it to files at a remote site.

Remote I/O also simplifies matters when computation may be moved between computers, for example in a checkpoint/restart system such as Condor. If staging is used, then input and partial output files must be restaged in order for the program to be restarted at another location. In contrast, a program that uses remote I/O can simply continue execution and access the same files as before.

Remote I/O also has the advantage of making intermediate results available before execution is complete, hence enabling real-time data analysis and computational steering. Finally, remote I/O can reduce requirements for often scarce disk space resources, by avoiding a need to stage data at remote locations. We often find that users are allocated significant compute resources at a remote site, but limited disk resources.

2.2 Wide-Area Computing Issues

Remote I/O libraries, like parallel I/O libraries, must orchestrate efficiently the transfer of data between a multiprocessor user application and a filesystem. Remote I/O is complicated, however, by the following issues not encountered in typical parallel computing environments. We note that our investigations thus far with our prototype library have focused chiefly on performance; we intend to address the other issues discussed below in future work.

Performance Characteristics. A remote I/O library running over a continental network can see a combined roundtrip communication and I/O latency of 100 msec. This is three to four orders of magnitude more than the roundtrip communication time found in a typical parallel computer (tens or hundreds of microseconds) and one order of magnitude more than the typical time for an I/O node to perform a disk access on behalf of a compute node ($\sim$10 msec). The bandwidth offered by the network over which a remote I/O library operates may be significantly lower than the internal communication network of a parallel computer; however, rapid increases in the speed provided by local and wide area networks will soon lead to situations in which the networks used within and outside parallel computers have comparable performance. For example, an OC48 ATM network provides 310 MB/sec, better than many contemporary multiprocessors, and also faster than many file systems.

Heterogeneity and Configuration. The computer, network, and storage systems used by a remote I/O system often include a heterogeneous mixture of hardware, software, and protocols. In such environments, selecting optimal I/O strategies is more difficult than in the relatively homogeneous environments in which parallel I/O libraries typically operate. A related issue is that the quality of service (QoS: e.g., average bit rate, or reliability) offered by the remote I/O network may be extremely variable, in which case we may require specialized techniques to shield an application from this variability, for example, buffering, or creating local copies to avoid loss of data over unreliable links. Alternatively, QoS may be controllable by a remote I/O library or user application, in which case accurate estimates of QoS requirements can improve both overall application performance and resource utilization.

Naming and Security. Remote I/O systems often connect computers and filesystems located in different administrative domains. This causes difficulties for both naming and security. We require a global name space for files, but different sites will use different filesystem structures. While distributed filesystems such as AFS create a global structure, we may not have that luxury. Uniform Resource Locators (URLs) represent an alternative approach. In addition, authentication, authorization, and privacy all become problematic issues in a distributed environment.

Fault tolerance. Distributed systems are inherently less reliable than nondistributed systems. For convenience and to ensure the integrity of data files, it is desirable for remote data access systems to tolerate brief, transient failures of networks and hosts. In case of more severe errors, it should be feasible to restart failed operations, such as from a checkpoint of an application's internal state. At the same time, the performance characteristics of distributed systems can motivate the use of strategies such as caching that can complicate recovery from failures.

2.3 Approaches to Remote Data Access

Other approaches to the remote data access problem fall into three general categories: distributed filesystems, parallel filesystems, and remote execution systems.

Traditional distributed filesystems (NFS [21] to some extent, and AFS [18] and DFS to a greater extent) provide a convenient interface for remote I/O: a uniform file name space is provided, and files are accessed with conventional read and write statements. However, these systems typically do not achieve good performance for high-performance computing workloads: they were designed primarily for a different class of users, e.g., software developers. For example, NFS bandwidth over an Ethernet LAN may be 1-3 Mb/sec, but an optimized communication library can achieve close to 10 Mb/sec. The lack of explicit interfaces for collective I/O also hinders performance optimization. In addition, distributed filesystems introduce significant implementation complexity and administrative overhead, which tend to hinder their widespread deployment. Web-based distributed file systems [1, 27] reduce implementation and administration costs but do not improve performance. Data servers such as DPSS [25] and MARS [4] use networked disk servers to provide high-speed streaming access to distributed data, but do not provide specialized support for access from parallel programs.

In contrast, parallel filesystems (e.g., [19, 7]) and I/O libraries (e.g., [3, 6, 22]) address performance issues directly by defining I/O interfaces that allow identification and optimization of collective I/O operations, by incorporating specialized buffering techniques, by supporting asynchronous operations, and by incorporating techniques (e.g., disk-directed [15], server-directed [22], and two-phase [23] I/O) for transferring data efficiently from compute nodes to disks. However, these systems are not designed to address the complex configurations, unique performance tradeoffs, and security problems that arise in wide area environments.

Finally, remote execution systems such as Condor [16] and WebOS [27] redirect Unix filesystem calls to a home filesystem, hence enabling location-independent execution of tasks scheduled to remote computers. However, these systems do not support parallel I/O interfaces or access to parallel filesystems.

In summary, existing techniques address issues relating either to distributed execution or to parallel performance, but not both. What is lacking is an approach that provides the high-performance characteristics of parallel I/O libraries while addressing the unique requirements of networked environments. This is the goal of our remote I/O work.

3 The RIO Remote I/O Library

To support our investigations of remote I/O, we have developed a remote I/O library called RIO. In this section, we describe how RIO addresses issues of portability, performance, and integration with wide area computing environments.

3.1 Portability of Interface

An I/O library is most useful if it supports both a wide range of application I/O patterns and multiple filesystems. It is not our goal to innovate in the area of I/O interfaces, and so we adopt the quasi-standard MPI-IO [6, 17]. This interface incorporates support for collective operations, asynchronous operations, and other I/O abstractions that have been found useful for high-performance parallel I/O. Whether the requirements of remote I/O motivate modifications or extensions to the MPI-IO interface remains to be seen, but our initial approach is to use MPI-IO unchanged.

We use the small code fragment of Figure 1 to illustrate the MPI-IO interface and its use in parallel programs. This Fortran code fragment is taken from an application called BTIO, which we discuss in more detail in Section 4.3. This program is designed to be executed by multiple processes. In brief, the program first opens a file for writing by calling `MPI_Open`, specifies the place at which writing should occur by calling `MPI_File_set_view`, repeatedly writes the file by calling `MPI_File_write_at`, and finally closes the file by calling `MPI_Close`.

We now describe the various MPI-IO calls in more detail. The `MPI_Open` call is used to open the file. Its arguments specify an MPI communicator representing the set of processes on which the file is to be opened (`comm_solve`), the file name (`filenm`), the mode in which the file is to be opened (for writing, in our case), an array of hints (here empty), and two output arguments: the file pointer and an error code. This function must be called collectively by all processes in the process group represented by `comm_solve`, with all processes passing the same values for the input arguments. The call opens the file and creates a distinct file pointer for each process, with all file pointers initially pointing to the beginning of the file.

The `MPI_File_set_view` routine changes the processes' view of the data in the file. This call can be used to provide different processes with different views; here, we indicate that all processes are to view the file as an array of double precision values. This call is also collective. Finally, the `MPI_File_write_at` routine writes the solution array to the file. This call is not collective, and so can be called and is executed independently by each process.

3.2 Portability of Implementation

Portability is a challenging problem in a remote I/O library because there may be no commonality in architecture between the computer on which an application runs and the potentially many remote filesystems that the program accesses. We address the portability problem by exploiting features of the ADIO implementation of MPI-IO [24]. ADIO adopts a modular design in an attempt to maximize code reuse across filesystems. High-level I/O libraries (in our case, MPI-IO) invoke services provided by a set of ADIO "devices," each providing low-level support for a particular I/O system (e.g., Unix, Intel PFS, IBM PIOFS).

As illustrated in Figure 2, RIO exploits the ADIO framework in two ways. On the client side, we provide a RIO device that implements ADIO calls as interactions with remote RIO servers. The servers themselves also use ADIO calls, in this case to access the remote filesystem in a system-independent fashion. This approach of simultaneously layering *below* ADIO (on the client side) and *above* ADIO (on the server side) greatly reduces implementation costs. On the client side, we need not implement all of MPI-IO nor be concerned with remote filesystem details. Instead, we can focus our attention on a small number of portable low-level functions. On the server side, we can operate on any system supported by ADIO.

3.3 Performance

A remote I/O library can use various strategies to transfer data between client and server. Research in parallel I/O has identified collective operations, nonblocking operations, and buffering as important techniques for maximizing performance on parallel filesystems. All of these techniques can improve the performance of applications that access remote data. We provide collective and nonblocking operations in RIO. However, as we shall see, the characteristics of networked systems favor designs that differ substantially from those for parallel filesystems.

Our RIO prototype uses the Nexus communication library [12] for client-server communications; Nexus, MPI, or potentially other communication mechanisms may be used within an application. When opening a file, a designated client process first attempts to connect to a server gateway process. The client and server then exchange information about file type and file access patterns, and the server issues an ADIO open call to open the relevant file(s). The client and server then establish the communication structure to be used for subsequent read and write operations. Finally, both client and server establish local data structures representing the open file; on the client side, a "file descriptor" is returned, encoding a reference to the client-side data structure.

Let P_C denote the number of processes executing at the client and P_S the number at the server. Following an open call, each client process can read and write at a distinct location in the file, using either collective or non-collective I/O operations. In a simple implementation of remote I/O, each client process keeps track of its own location within the file, and implements a read or write operation as a separate remote procedure call (i.e., a round-trip communication) to a server process, regardless of whether the operation was collective. A round trip is required even for write calls, in order to provide a return code.

An analysis of the various inefficiencies inherent in this simple approach allows us to introduce some of the optimizations used in RIO.

Forwarder Nodes. Client and server processes communicate directly. A disadvantage of this strategy is that a single process may have to use two communication methods: e.g., on the IBM SP, a vendor-supplied MPI library and TCP/IP. This simultaneous use can introduce significant overheads due to the need to manage two communication interfaces [10] or may be disallowed entirely if network interfaces can be accessed only from dedicated service nodes. Hence, we introduce *forwarder nodes* (analogous to

```
      call MPI_Open(comm_solve, filenm, MPI_WRONLY+MPI_CREATE,
     $              MPI_INFO_NULL, fp, ierr)

      call MPI_File_set_view(fp, 0, MPI_DOUBLE_PRECISION,
     $                       MPI_DOUBLE_PRECISION, MPI_INFO_NULL, ierr)

      call initialise

      ...

      do step = 1, niter
        call adi
        if (mod(step, wr_interval) .eq. 0) then
          call MPI_File_write_at(fp, iseek, u(1,0,jio,kio,cio), count,
     $                           MPI_DOUBLE_PRECISION, mstatus, ierr)
        endif
      enddo
      call MPI_Close(fp, ierr)
```

Figure 1: Code fragment from an application (BTIO) showing use of MPI-IO calls

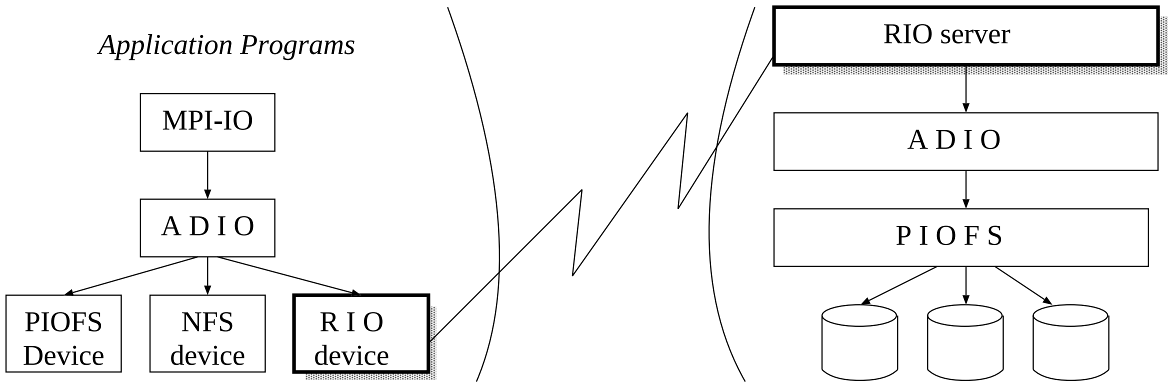

Figure 2: RIO architecture, showing how RIO layers below ADIO in the client and above ADIO in the server

the dedicated I/O nodes used in some I/O systems), to which each client process forwards communications destined for the server, and which handles communications from the server to client processes. These forwarder nodes must use both MPI and TCP, but are dedicated and hence can be optimized for this purpose. The forwarder nodes can also be used to throttle traffic to avoid network saturation [26]. In our current work, the client and server each use a single forwarder. However, multiple forwarders can be advantageous if there are multiple network interfaces or if compression, message digest, or encryption techniques are to be applied to data.

Exploitation of Collective Operations. In the simple implementation scheme outlined above, each client process communicates independently with the server, even when engaged in a collective operation. Hence, a single client-side collective call requires P_C messages and results in P_C independent I/O operations at the server. Both the multiple communications and multiple I/O operations can be inefficient in some situations. Multiple communications can be avoided by collecting the communications performed by the P_C clients (e.g., at the forwarder) and transferring them to the server in a single message. Multiple server I/O operations can be avoided by tagging client messages to indicate when they refer to collective calls, and then invoking a collective I/O operation at the server. The latter strategy is straightforward if $P_S = P_C$, since the server can issue open, read, and write operations identical to those performed by the client. The situation remains straightforward if P_C is an integer multiple of P_S, or vice versa, as the calls issued by the client are easily mapped to server processes. In other situations, it can be hard to translate a client-side collective operation into an efficient collective operation at the server.

Reduction of Round-Trip Costs. The round trip performed for each read and write operation can take 100 msec or more in a wide area environment, significantly more than an I/O operation. RIO seeks to reduce these costs by incorporating support for asynchronous I/O operations. Asynchronous operations allow several I/O operations to be outstanding at once, hence enabling pipelining of I/O operations in the network and I/O system, and overlapping of computation and I/O in the application. An asynchronous operation is initiated in RIO by sending the usual request message from the client node to its forwarder in a nonblocking fashion, so that control returns immediately to the application. The operation completes when the reply message arrives at the client node. The application can test or wait for completion using standard MPI-IO calls.

Another approach that we have yet to evaluate is to reduce the number of communication operations by client-side buffering, either independently by each client process, collectively by multiple client processes, or by the forwarder. Another interesting possibility that needs to be explored further is to use as a cache the potentially large aggregate memory or local disk space of the parallel computer on which the application runs. These approaches appear particularly important in situations where a program reads the same file multiple times. Here, a staging approach might perform much better than a simple remote I/O library that performs no caching. However, a sophisticated remote I/O library could in principle stage the file to local memory or disk and keep it there, if it were able to determine that repeated reads were to be performed.

Figure 3 outlines the structure that results when we introduce these three optimizations. The figure shows the client-side buffers (here associated with client processes), the forwarder processes, and the translation of a client-side collective I/O call (`MPI_READ_ALL`) into a collective I/O call at the server.

3.4 Integration

Because RIO is designed to execute in a wide area environment, its implementation must address issues of naming, configuration, and security. In the following, we explain how these issues can be addressed by using mechanisms provided by the Globus distributed computing toolkit [11].

Naming. RIO uses a URL-like notation to provide a uniform name space for files. A file is opened with a call of the form

MPI_Open(..., "x-rio://*host*:*port-num*/*path*", ...)

where the *host* and *port-num* identify a RIO server and *path* identifies a file managed by that server. In the future, we may substitute Uniform Resource Names (URNs) for URLs, to permit location-independent naming of cachable or replicated resources such as databases.

Configuration. RIO permits the use of Globus configuration mechanisms. For example, when establishing a Nexus connection between client and server, RIO can use the Metacomputing Directory Service (MDS) [9] to determine network availability, current load, and access mechanisms. RIO also can interact with Globus schedulers to reserve capacity on networks that support quality of service negotiation.

Security. A remote I/O system may be required to verify a user's identity (authentication), to determine whether and how a user is able to access a file (authorization), and to ensure the integrity and privacy of data transferred over public networks. We design RIO to incorporate the solutions to these problems provided by Globus.

The current Globus system supports a global "Globus id" but requires that a user have an account at a site before it can use that site's resources. Globus provides a cryptographically secure mapping from Globus id to local ids, hence allowing a user to authenticate once (to Globus) and subsequently access resources at any Globus site where the user has an account. These mechanisms can easily be adapted for use by RIO. Authentication is performed by Nexus when a RIO client connects to a RIO server. If authentication succeeds, the local user id of the Globus user is also established, and hence the file access rights of the Globus user at that site are determined. Once authentication is in place, Globus/Nexus mechanisms can be used to apply digital signatures for message integrity and/or encryption for privacy. If desired, these mechanisms can be applied only when communicating over networks defined to be insecure.

In the longer term, we expect Globus—and hence RIO—to eliminate the requirement that a user have a local account at every site. Access control lists are one approach to authorization in this regime. Cryptographically signed "use condition certificates" [14] represent another promising approach.

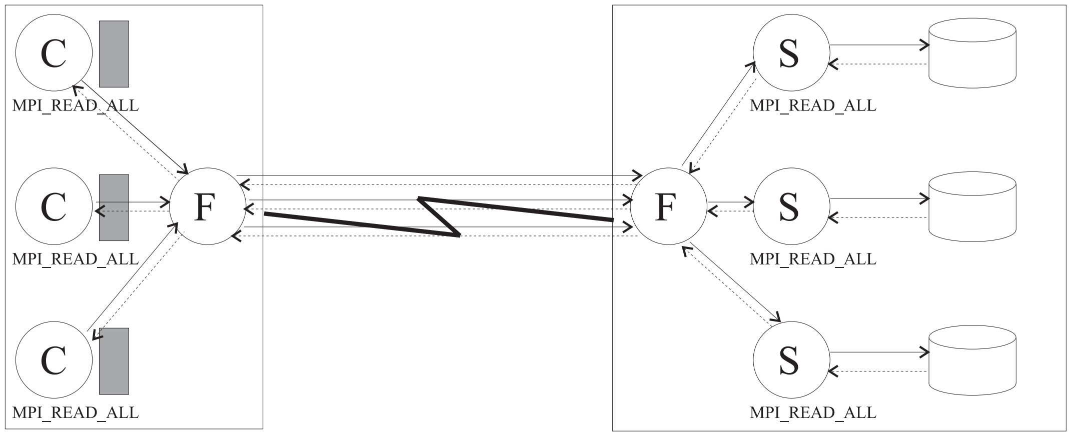

Figure 3: RIO's optimized I/O strategy, showing the client (C), forwarder (F), and server (S) processes, and the communications performed following a collective read operation.

4 Experimental Studies

We report on experiments designed to determine the basic performance characteristics of RIO and to provide a preliminary evaluation of RIO's utility for applications. These experiments comprise a series of microbenchmarks similar to those used traditionally for evaluation of I/O library performance, plus a single application.

4.1 Experimental Platform

In selecting an experimental platform, we must trade off our interest in exploring true remote I/O against the need for a controlled environment in which the impacts of different performance issues can be easily measured. These considerations motivate us to define a testbed comprising two partitions of the same IBM SP multicomputer. Within each partition, communication can occur via vendor-supplied MPI, while TCP/IP (over the high-performance switch) is used between partitions. The client runs in one partition and the server in the other. Because of our use of forwarder nodes, this simple configuration has performance characteristics similar to (though somewhat better than) two IBM SPs connected by a high-bandwidth local or metropolitan area network. While intrapartition communication peaks at over 55 MB/sec with latencies of around 50 μsec, interpartition communication peaks at 22 MB/sec with latencies of around 320 μsec.

All experiments were performed on the IBM SP2 at Argonne National laboratory and used IBM PIOFS version 1.2 as the "remote" filesystem. All nodes used in our experiments were SP thin nodes (roughly equivalent to RS/6000 Model 390, with at least 256 MB memory) running AIX 4.2. PIOFS distributes files across multiple PIOFS servers [2]. At ANL, there are 4 such servers and each server has four 9 gigabyte SSA disks attached to it. Each file consists of a set of cells, and each cell is stored on a particular server node. The default number of cells is the number of PIOFS servers; if the number is greater, cells are striped across servers in a round-robin fashion. A file is divided into basic striping units (BSUs), which are assigned to cells in a round-robin fashion. The default BSU size is 32 KB. In some situations, tuning of these various parameters can significantly affect performance. We used default values in all experiments.

PIOFS performance is sensitive to the size of the data being read and written. Small ($<$ 8 KB) accesses that do not hit in the disk block cache require roughly 2 msec. Much higher performance can be achieved for larger read and write sizes.

4.2 Microbenchmark Results

Our microbenchmarks are designed to reveal how RIO read and write bandwidths vary as functions of P_C and read or write size. Each microbenchmark uses repeated read or write operations to transfer contiguous chunks of data from or to a single shared file. The bandwidth is measured by dividing the amount of I/O performed by the elapsed time as seen at the client side. We show results from using both RIO blocking and nonblocking operations. In the case of nonblocking operations, the elapsed time includes the time required to complete all outstanding I/O requests. Each measurement represents the average of a large number (typically a thousand or more) transfers; the number of transfers is chosen such that each experiment runs for at least a minute.

There is significant variation in the times obtained for large transfer sizes, probably due to contention at the PIOFS server nodes. All experiments were performed without any dedicated access to the SP, so there is contention with other jobs accessing PIOFS and the high performance switch.

Figure 4 shows the transfer rates (totals summed over P_C processes) that we measured for $P_C = P_S =$1, 2, and 4, using RIO blocking calls for different access sizes. It also shows the I/O bandwidth obtained from using PIOFS directly using P_C processes. We also include in the graph horizontal lines representing the bandwidth measured with two simple ping-pong programs. The line labeled "Client/server forwarding" was obtained with a program that bounces large messages between a client process and a server process, via the intervening forwarders. Hence, it approximates the best data transfer rate that can be obtained for synchronous op-

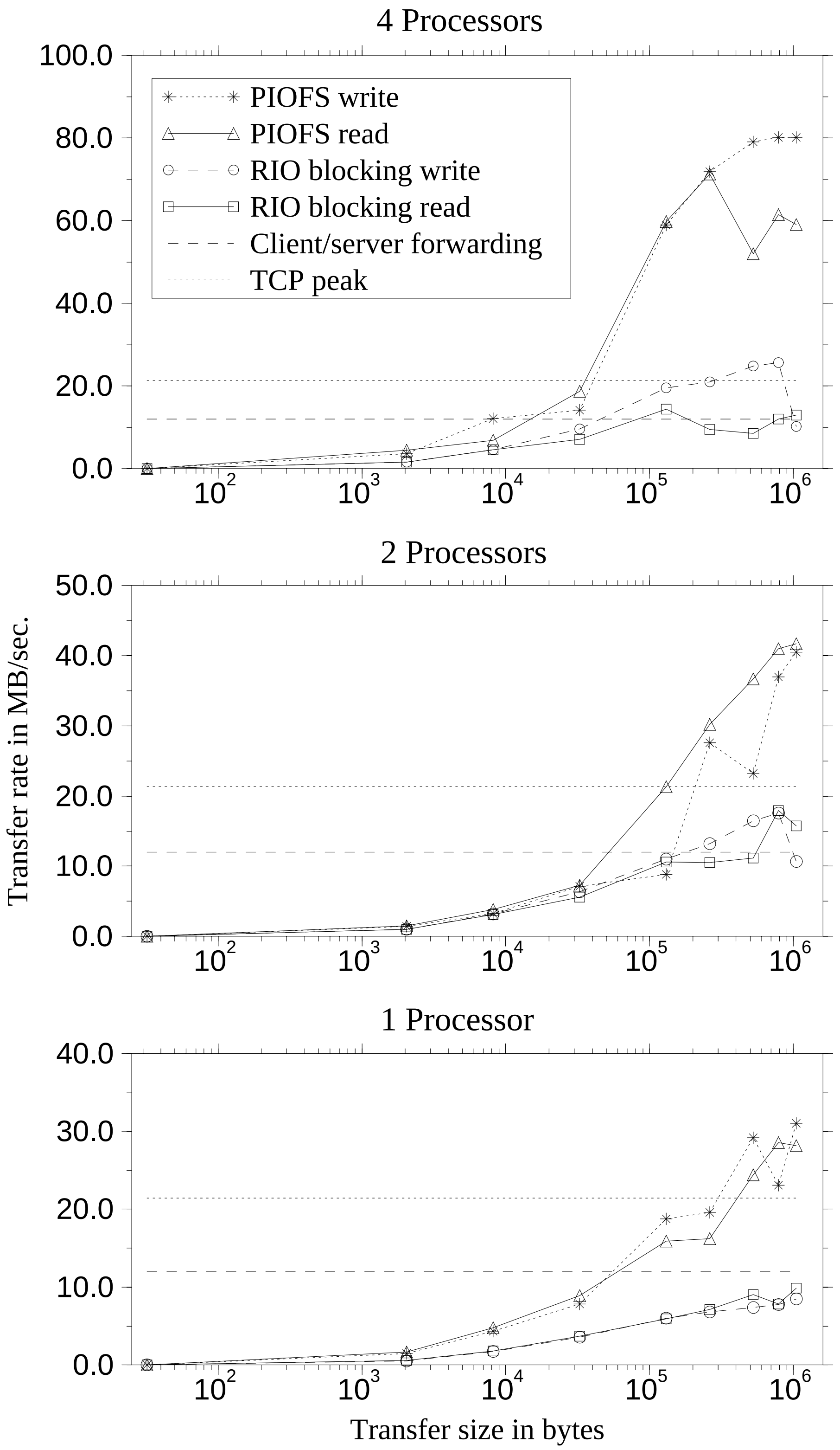

Figure 4: I/O performance as measured with microbenchmark programs that use (a) local PIOFS, on $P = 1$, 2, and 4 processors; and (b) blocking RIO calls, with $P_C = P_S = 1$, 2, and 4. Also shown is the performance achieved by the client-server forwarding and TCP peak benchmarks described in the text.

erations between a single client and a single server in our architecture. The line labeled "TCP peak" was obtained with a simpler program that uses TCP to bounce large messages back and forth between two processes. The first number (12.0 MB/sec) is less than the second (21.4 MB/sec) because in the latter case, a round-trip client/server communication involves just two messages, while in the former, there are four additional messages: outbound and inbound messages between forwarder and nonforwarder nodes at both the client and server.

Examining Figure 4, we see that PIOFS performance increases with the number of processors, reaching around 80 MB/sec with four application nodes. These results match those of other researchers. RIO sustained bandwidth for $P_C = P_S = 1$ increases with data size, but does not exceed the client/server forwarding bandwidth of 12.0 MB/sec. This result is to be expected as the forwarding bandwidth is also measured with a similar configuration of one client and one server process. For larger numbers of clients, we are able to exceed this bandwidth, as the increased number of messages enhances pipelining of communications. Nevertheless, the maximum bandwidth is still limited by the link between the forwarders. We obtain a peak RIO bandwidth of around 18 MB/sec when $P_C = P_S = 2$ and 26 MB/sec when $P_C = P_S = 4$.

In the next set of experiments, we fixed the number of server nodes at two and measured RIO bandwidth for different numbers of client processes, using both blocking and nonblocking operations. Figure 5 shows the transfer rates (totals summed over P_C processes) measured for different access sizes. The top row shows results obtained using nonblocking RIO calls and the bottom row displays the performance using blocking RIO calls.

In general, nonblocking RIO outperforms the blocking version, with peak values of 26 MB/sec and 22 MB/sec obtained for read and write operations respectively. We also observe that when using nonblocking calls, performance is limited by the forwarder link for large transfer sizes even with only two server nodes. The optimum number of server nodes to be used in a given system depends on the performance characteristics of the RIO network and the application I/O request sizes, but it seems likely that it will often be small.

These results show that RIO is able to drive the single TCP connection between the clients and servers at close to its peak bandwidth, at least for large messages. We see also that in our experimental configuration, the principal obstacle to improved performance is the capacity of this network. Faster networks and improved forwarder structures are two possible approaches to improving performance. We can derive maximum advantage from RIO by using asynchronous operations; much of the performance improvement in this case comes from the pipelining of requests along the path from clients to servers. As we shall see in the next section, this advantage is even greater if an application can be structured so as to permit overlap of computation with I/O.

4.3 Application Results

We use the BTIO benchmark from the NAS I/O benchmark suite [5], specifically, the program `BTIO-simple-mpiio`. This benchmark simulates the I/O required by a pseudo-time-stepping flow solver. It implements an approximate factorization algorithm with the requirement that after every k iterations, a three-dimensional solution vector—of size N^3—is written to a disk file. A total of I iterations of the algorithm are performed. The application code is in Fortran and uses the MPI-IO interface to write output data to a single file. The application does not perform any read operations.

In our experiments, we consider problem sizes $N = 32$, 64, and 80; total data written in these three cases is 52 MB, 420 MB, and 1 GB, respectively. We define the elapsed time as the wall clock execution time for the application (including both computation and I/O), and the application sustained I/O transfer rate as the total amount of I/O performed divided by the total elapsed time. The elapsed time includes both the time for computation and I/O inside the application, and so the resulting I/O transfer rate is different from that measured in the microbenchmarks, where no significant computation is performed.

`BTIO-simple-mpiio` performs many small writes in an irregular pattern and hence performs poorly on PIOFS, due to the high PIOFS overhead associated with small writes. Hence, we produced a modified version of the benchmark that redistributes the output data before the solution vector is written to disk. In the optimized code, each node essentially collects data from other nodes into a temporary contiguous write buffer; each processor then performs a single write operation at each dump. This optimization might well be performed automatically by an MPI-IO implementation of collective write. However, the MPI-IO implementation used in our studies did not incorporate this optimization at the time we made these measurements.

We use four processors to execute the application in all experiments. We also use four server nodes for those experiments performed with the original version of the code; for the optimized version, just one server node is used, because the optimized version performs I/O in larger chunks and so the network becomes the bottleneck in those cases. We have observed similar behavior to that reported below when using different numbers of application and server processors, and hence for brevity, we do not report those results.

We measure elapsed time for four configurations: when using PIOFS directly (i.e., without using RIO); when using RIO (blocking calls) to transfer data from the application to the RIO server, which then makes the PIOFS calls; when using RIO (nonblocking calls); and when data is first written to PIOFS directly, without RIO, and then transferred to a user filesystem with ftp. The latter configuration corresponds to the use of manual staging. The column labeled *Compute* gives the execution time for the application when no I/O is performed, and the last column indicates the percentage improvement over PIOFS+ftp obtained when using nonblocking RIO. In the nonblocking RIO version of the original code, up to 64 I/O operations may be outstanding at once. In the nonblocking RIO version of the optimized code, we issue an asynchronous write for each dump and then proceed with computation, waiting for completion only at the start of the next dump. This operation can be asynchronous because writes are performed from the temporary write buffer.

Tables 1 and 2 show the elapsed times and application sustained transfer rates measured for both the original and optimized versions of the program. The optimized version of BTIO performs better than the original, due to the reduced number of write operations. We see that nonblocking calls significantly affect performance in all cases. This is because in the absence of nonblocking calls, the round-trip message exchange between application and server is a significant source of overhead. When nonblocking operations

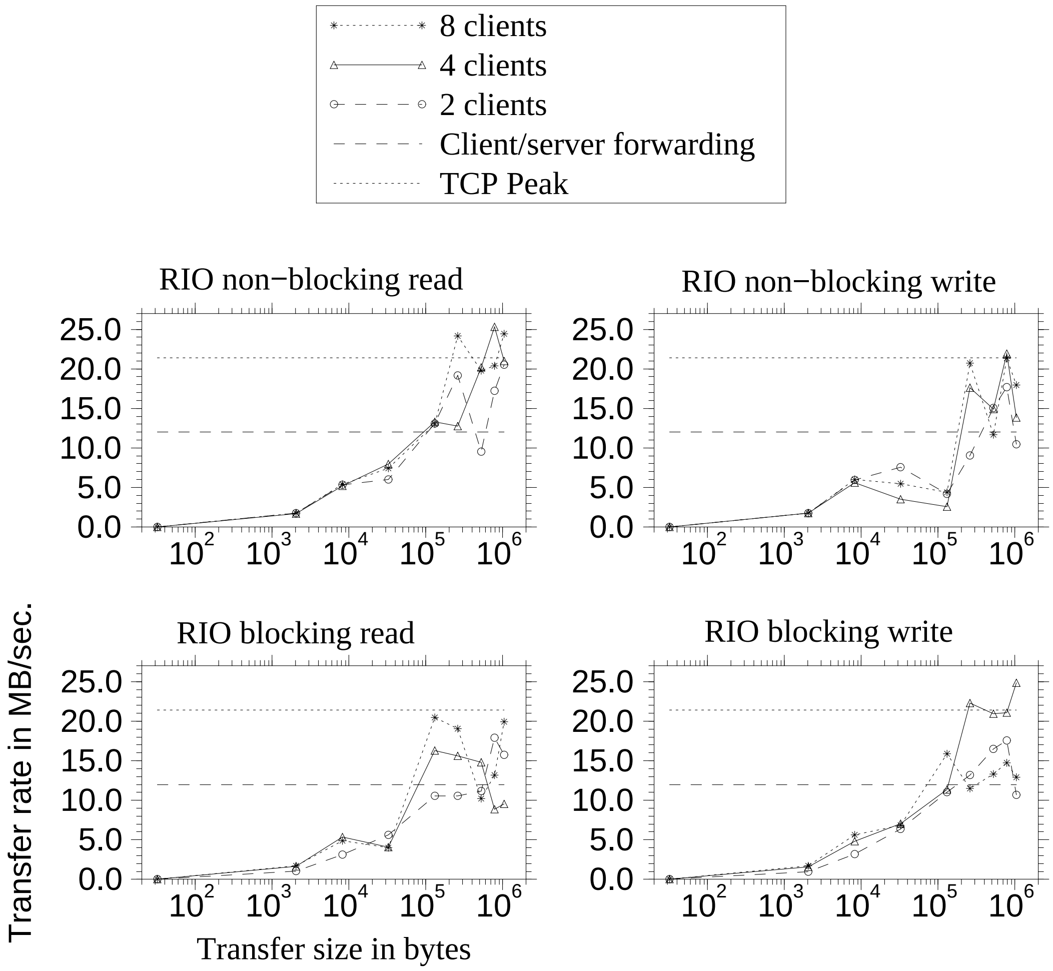

Figure 5: I/O performance as measured with blocking and nonblocking versions of a RIO microbenchmark, with a fixed $P_S = 2$ and for $P_C = 2$, 4, and 8.

Table 1: Execution times in seconds for the original and optimized BTIO application.

Version	N	I	k	Compute	Local PIOFS	PIOFS+ftp	RIO (blocking)	RIO (nonblock)	Improvement
Original	32	200	5	86.84	206.12	211.97	199.98	196.06	8.1%
Original	64	200	5	624.52	837.60	883.26	987.18	941.98	–6.2%
Original	80	50	1	304.04	857.75	968.86	1080.66	959.21	1.0%
Optimized	32	200	5	86.84	100.06	105.91	107.52	89.42	18.4%
Optimized	64	200	5	624.52	668.84	722.30	676.78	648.40	11.4%
Optimized	80	50	1	304.04	424.71	535.82	487.98	442.65	21.0%

Table 2: Application sustained I/O transfer rates for the original and optimized BTIO applications, in MB/sec).

Version	N	I	k	Local PIOFS	PIOFS+ftp	RIO (blocking)	RIO (nonblock)
Original	32	200	5	0.2543	0.2473	0.2622	0.2674
Original	64	200	5	0.5008	0.4749	0.4249	0.4453
Original	80	50	1	1.1938	1.0569	0.9476	1.0676
Optimized	32	200	5	0.5240	0.4950	0.4876	0.5863
Optimized	64	200	5	0.6271	0.5807	0.6197	0.6469
Original	80	50	1	2.4111	1.9110	2.0984	2.3133

are used, performance improves because multiple I/O operations are pipelined (as in the original code) and because I/O and round-trip overheads are overlapped with computation (as in the optimized code). As a result, throughput is close to what we get when accessing PIOFS directly. In fact, because PIOFS does not support nonblocking operations, RIO performs better than local PIOFS in some cases. The maximum application sustained transfer rate is 2.41 MB/sec for local PIOFS and 2.31 MB/sec for RIO.

Finally, we see that the total execution time when using RIO is, in most cases, less than the total turnaround time when staging is used (PIOFS+ftp). For the optimized code with $N = 80$, RIO is 21 percent faster. This result is due to the overlapping of computation and data transfer achieved by RIO and illustrates how remote I/O can improve application performance compared with traditional techniques for remote data access, while at the same time providing a more convenient interface.

5 Conclusions

We have argued for the importance of remote I/O as a tool for high-performance, low-overhead distributed computing. Remote I/O libraries allow programs to use familiar parallel I/O interfaces to access data contained in remote filesystems. In principle, they can improve performance, enhance flexibility, and reduce complexity in applications that must access nonlocal data. We have identified some of the challenges that must be overcome before these benefits can be realized; these include high latencies, low bandwidths, complex configurations, and security. We have also described a prototype remote I/O library called RIO that incorporates solutions to some of these problems. Performance experiments in a controlled multicomputer environment show that RIO introduces little overhead and can achieve improved turnaround time compared to remote execution combined with staging.

The work presented here is just a first step toward a truly usable remote I/O facility for high-performance computing applications. Our next step will be to deploy the RIO prototype in a wide area computing testbed. Our first target is the sites connected by the ESnet and CAIRN networks, in particular Argonne, Berkeley, and USC/ISI. This environment will enable us to evaluate our techniques more realistically. We are particularly interested in understanding how well applications perform when using remote I/O techniques in high-latency environments. The seamless access to remote file systems provided by RIO is less useful if the programmer must use different program structures and algorithms to achieve acceptable performance in low-latency and high-latency environments. Hopefully, appropriate prefetching and caching strategies in the remote I/O library will mitigate the impact of high latency, but this remains to be seen.

Another set of experiments that we plan to conduct in wide area testbeds is a detailed comparison with distributed filesystems for a range of scientific applications. We are interested in understanding the regimes in which distributed file systems and remote I/O perform more effectively. This work may motivate proposals for alternative data transfer mechanisms in distributed file systems.

A final issue that we want to investigate relates to network quality of service. Because wide area networks are typically shared media, communication performance can be variable. In our experiments to date, we have not considered the impact of this variability on performance. We propose to study the impact of network performance variability on I/O performance and to examine the feasibility of using network quality of service reservation techniques to improve network predictability. One issue to be considered here is whether current quality of service classes are appropriate for remote I/O. We are also interested in determining whether it is feasible to use adaptive techniques in the remote I/O library or in the application to provide more robust performance in the face of variability.

Acknowledgments

We thank David Lifka, Rajeev Thakur, and Steven Tuecke for their invaluable assistance with this work, and the Cornell Theory Center and Argonne National Laboratory's Center for Computational Science and Technology for access to their IBM SP systems. The paper was much improved by numerous insightful comments from Orran Krieger.

This research was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, in support of the multiagency Scalable I/O project.

References

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the operating system at the user level: The UFO global file system. In *1997 Annual Technical Conference on UNIX and Advanced Computing Systems (USENIX'97)*, January 1997.

[2] F. Bassow. *IBM AIX Parallel I/O File System: Installation, Administration, and Use.* IBM, Kingston, N.Y., May 1995. Document Number SH34-6065-00.

[3] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz. Jovian: A framework for optimizing paral-

lel I/O. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, October 1994.

[4] M. Buddhikot, G. Parulkar, and J. Cox. Design of a large scale multimedia storage server. In *Proc. INET '94*, 1994.

[5] R. Carter, B. Ciotti, S. Fineberg, and B. Nitzberg. NHT-1 I/O benchmarks. Report RND-92-016, NAS, NASA Ames Research Center, Moffett Field, CA, Nov 1992.

[6] P. Corbett, D. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: A parallel file I/O interface for MPI. Technical Report NAS-95-002, NAS, NASA Ames Research Center, Moffett Field, CA, January 1995. Version 0.3.

[7] P. Corbett, D. Feitelson, J.-P. Prost, and S. Baylor. Overview of the Vesta parallel file system. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 1–16, 1993.

[8] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson. A quantitative analysis of cache policies for scalable network file systems. In *Proc. 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 150–160, 1994.

[9] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 56h IEEE Symp. on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, 1997.

[10] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40:35–48, 1997.

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

[12] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[13] W. Hibbard, J. Anderson, I. Foster, B. Paul, C. Schafer, and M. Tyree. Exploring coupled atmosphere-ocean models using Vis5D. *International Journal of Supercomputer Applications*, 10(2):211–222, 1996.

[14] W. Johnston and C. Larsen. A use-condition centered approach to authenticated global capabilities: Security architectures for large-scale distributed collaboratory environments. Technical Report 3885, LBNL, 1996.

[15] D. Kotz. Disk-directed I/O for an out-of-core computation. In *Proc. 4th IEEE Symp. on High Performance Distributed Computing*, pages 159–166, August 1995.

[16] M. Litzkow, M. Livney, and M. Mutka. Condor - a hunter of idle workstations. In *Proc. 8th Intl Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[17] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, 1997. http://www.mpi-forum.org.

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

[19] Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, June 1997.

[20] M. Norman, P. Beckman, G. Bryan, J. Dubinski, D. Gannon, L. Hernquist, K. Keahey, J. Ostriker, J. Shalf, J. Welling, and S. Yang. Galaxies collide on the I-WAY: An example of heterogeneous wide-area collaborative supercomputing. *International Journal of Supercomputer Applications*, 10(2):131–140, 1996.

[21] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proc. Summer USENIX*, pages 119–130, June 1985.

[22] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, California, December 1995.

[23] R. Thakur and A. Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[24] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of The 6th Symposium on the Frontiers of Massively Parallel Computation*, October 1996.

[25] B. Tierney, W. Johnston, L. Chen, H. Herzog, G. Hoo, G. Jin, and J. Lee. Distributed parallel data storage systems: A scalable approach to high speed image servers. In *Proc. ACM Multimedia 94*. ACM Press, 1994.

[26] B. Tierny, W. Johnston, J. Lee, and G. Hoo. Performance analysis in high-speed wide area IP over ATM networks: Top-to-bottom end-to-end monitoring. Technical report, LBNL, 1996.

[27] A. Vahdat, P. Eastham, and T. Anderson. WebFS: A global cache coherent filesystem. Technical report, Department of Computer Science, UC Berkeley, 1996.