An MPI-IO Interface to HPSS[1]

Terry Jones, Richard Mark, Jeanne Martin
John May, Elsie Pierce, Linda Stanberry
Lawrence Livermore National Laboratory
7000 East Avenue, L-561
Livermore, CA 94550
johnmay@llnl.gov
Tel: 510-423-8102
Fax: 510-422-4293

Abstract

This paper describes an implementation of the proposed MPI-IO [5] standard for parallel I/O. Our system uses third-party transfer to move data over an external network between the processors where it is used and the I/O devices where it resides. Data travels directly from source to destination, without the need for shuffling it among processors or funneling it through a central node. Our distributed server model lets multiple compute nodes share the burden of coordinating data transfers.

The system is built on the High Performance Storage System (HPSS) [2, 12, 14], and a prototype version runs on a Meiko CS-2 parallel computer.

1 Introduction

The Scalable I/O Facility (SIOF) project[2] is an effort to develop an I/O system for parallel computers that offers both high aggregate bandwidth and the ability to manage very large files [8]. To meet these needs, the SIOF project is developing a hardware infrastructure that will connect the processors in a parallel computer to multiple storage devices through a Fibre Channel network [3]. The project is also developing an application programming interface (API) that will give large scientific codes flexible, efficient access to the I/O system without forcing programmers to manage low-level details. This paper describes the implementation of the SIOF API software.

1.1 Background

Parallel programs use a number of strategies to manage large data sets. Most parallel computers offer a global file system that all the processors can access. Data typically travels over the internal communication network between the compute nodes and one or more I/O nodes, which manage a set of storage devices. This arrangement gives all the nodes access to all the files, but I/O traffic must compete with regular message traffic for access to the communication network. Depending on the configuration of the system, the I/O nodes may become a bottleneck to data transfer. Some parallel machines have local disks attached to each compute node, and programs may write separate files from each node to its local disk. This approach offers high aggregate bandwidth, since nodes

[1] This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-Eng-48. Specific funding came from the Gas and Oil Information Infrastructure initiative and the DOE Digital superLab project.
[2] SIOF should not be confused with a different project that has a similar name, the Scalable I/O Initiative [9].

transfer data through separate, dedicated channels. However, merging the data in these separate files or reading the files on a different set of nodes can be inconvenient.

The SIOF architecture uses a separate I/O network to connect each compute node to the storage devices. This allows I/O to proceed in parallel at a high aggregate bandwidth. It also lets a program treat data distributed over multiple devices as a single logical file. The system supports third-party transfers, so one compute node can orchestrate data transfers between a file and several processors. SIOF uses the High Performance Storage System (HPSS) [2, 12, 14] to manage distributed files and third-party transfers. HPSS is a joint development project of IBM and several U.S. national laboratories and supercomputer centers. Since the HPSS API is designed mainly for shared-memory systems, and since it requires programmers to specify many details of a parallel transfer, the SIOF project is developing a separate API for message-passing systems. This API is based on the proposed MPI-IO standard [5], which in turn is based on the popular MPI (Message-Passing Interface) standard [10]. Our initial implementation of the SIOF hardware and software runs on a Meiko CS-2 parallel computer.

1.2 The SIOF application programming interface

Several MPI-IO development efforts are now underway; however because the architecture of our underlying I/O system is unique, the SIOF implementation has some noteworthy features:

- Control of access to a given open file is centralized, but data transfer is distributed and direct.
- A *distributed server model* spreads the burden of controlling different open files among multiple compute nodes.
- When multiple processes participate in *collective* read or write operations, the system can determine dynamically how to group these requests for high throughput based on the transfer size, the distribution of data among the compute nodes and I/O devices, and other parameters.

The next two sections give the background to SIOF, HPSS, and MPI-IO. Section 4 examines the architecture of our MPI-IO implementation, and Section 5 describes how we manage collective I/O requests. Section 6 reports the current status of the project and our future plans for it. We conclude in Section 7 with a summary of the SIOF API.

2 The Scalable I/O Facility and HPSS

The SIOF project goal is to provide a "network-centered, scalable storage system that supports parallel I/O" [8]. To this end, SIOF is collaborating with HPSS developers to extend the HPSS environment in two areas. First, it is addressing some issues (such as protocols and security) pertaining to network-attached peripherals [15]. Second, it is providing an MPI-IO interface to the HPSS client API, as described in the sections that follow.

The SIOF implementation uses a crosspoint-switched FC fabric that will connect the processors of a Meiko CS-2 directly to disk arrays, parallel tapes, and frame buffers. Each compute node is capable of independent I/O across the FC fabric so that all nodes may perform I/O in parallel. The SIOF API orchestrates coordinated accesses across the processors in a distributed computation, with each processor working on a part of a file. The envisioned architecture is shown in Figure 1.

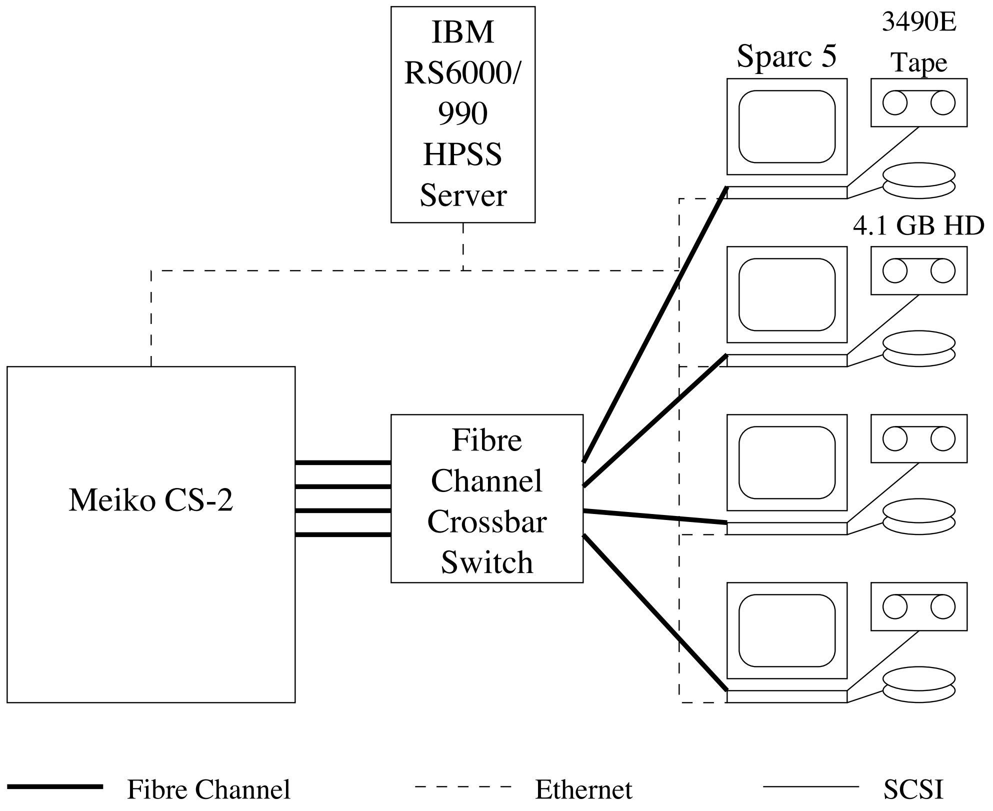

Figure 1: The Scalable I/O Facility architecture consists of a Meiko CS-2, an IBM RS6000 HPSS server, and a collection of tape drives and disk arrays. A Fibre Channel network connects the storage devices to the Meiko compute nodes.

The SIOF extensions rely on HPSS to achieve this implementation. HPSS is a standards-based, modular, hierarchical storage system that supports scalability in a variety of parallel environments. HPSS consists of multiple layers of interfaces that provide secure parallel access to the files of a storage system. The interfaces are implemented using DCE (Distributed Computing Environment) [6] and Transarc's Encina [13] transaction-based remote procedure calls.

The higher-level interfaces implement the administration (e.g., naming) and security (e.g., access control) of the storage system, while the lower-level interfaces implement the mechanics of the parallel data transfers required by file access commands. The interfaces of particular interest to the SIOF are the *data movers.*

For any given data transfer there is a mover on the application (client) side, and a corresponding mover on the HPSS side. These two movers determine the details of a given data transfer from a data structure called an IOD (I/O descriptor). This descriptor treats a data transfer as a mapping from the source of the transfer into a data transfer stream and a corresponding mapping from the transfer stream into the destination or sink of the transfer (see Figure 2).

In the simplest case, both the source and sink of the transfer are one contiguous block of bytes. But with distributed files and distributed applications, blocks may be discontiguous at either the source or the destination. The descriptive flexibility of the HPSS IOD allows a single transfer to consist of bytes striped across multiple nodes, multiple storage devices, or both. In each IOD, the transfer stream mappings from the source and to the sink are represented by a list of one or more descriptors, where each entry on the list describes a contiguous block of bytes.

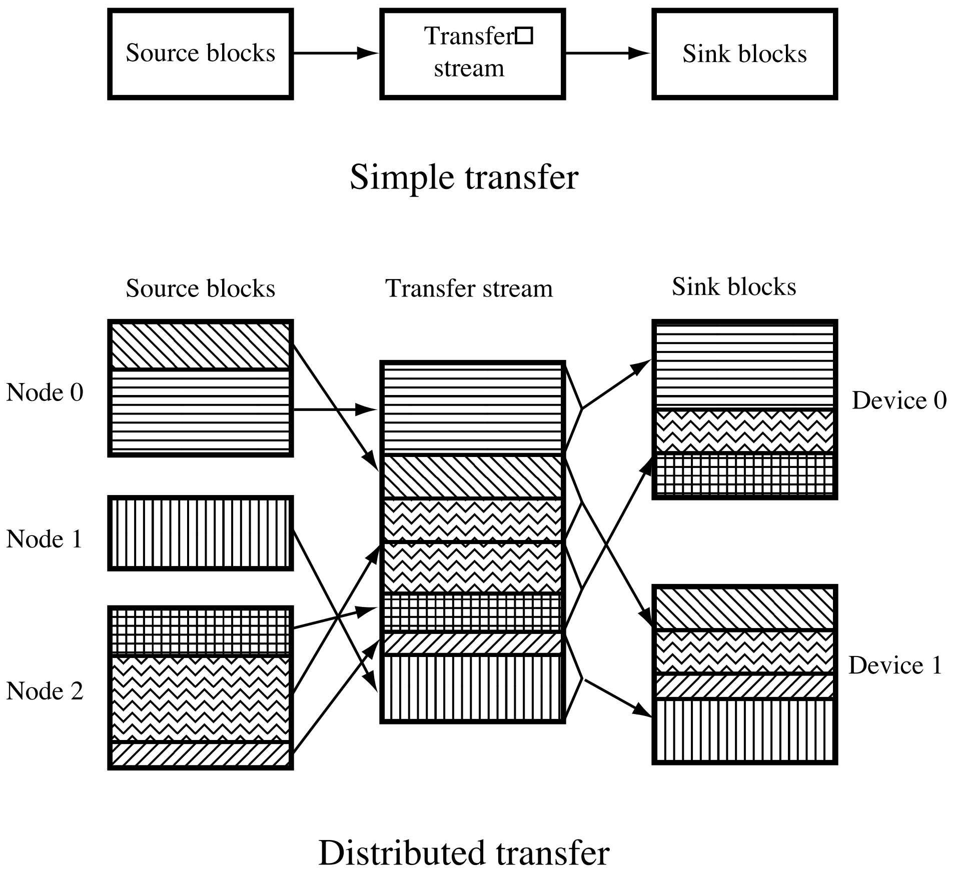

Figure 2: In a simple transfer, is mapped from a source into a transfer stream and then to a sink with no reordering. In a distributed transfer, HPSS can move blocks of any size from source to sink in any order or interleaving.

To initiate a simple contiguous transfer, a client application can call HPSS through Unix-like **read** and **write** commands. These interfaces construct an IOD for the transfer automatically. More complex transfers require the application to construct its own IOD with the necessary mappings and to deliver this IOD to HPSS through **readlist** and **writelist** commands.

Application programs using the SIOF API (rather than the HPSS API) do not create IODs directly. Instead, the SIOF API code creates IODs using the MPI datatypes (see Section 3) specified in MPI-IO **open**, **read**, and **write** operations. The motivation for hiding the details of the IOD construction is twofold: the client application can use the simpler MPI-like interface to describe the transfer, and the new interface layer introduces the possibility of optimizing transfers.

The HPSS architecture is also designed to allow for potential optimizations as a data transfer request is processed. The higher-level servers rewrite the client IOD and dispense one or more IODs to HPSS movers to complete a given transfer. In the process, HPSS uses its knowledge of how the file being accessed is distributed across storage devices, and which HPSS movers control those devices, to break the client IOD into component IODs to be distributed to the HPSS movers. Although HPSS does not currently reorder or combine client IODs, its design allows this in the future.

Note that although there is one logical mover for the client and another for HPSS for each transfer, there may in fact be multiple movers active on both sides: one per participating node on the client side, and one per device on the HPSS side. These movers are threads that are spawned when a transfer begins, and they terminate when the transfer is complete.

As shown above, the architecture of HPSS allows full generality of how source blocks are mapped into a data transfer and thence into sink blocks, but it is expected that the best performance will always be achieved when source and sink blocks match in size and number exactly. This will reduce contention (e.g., where more than one node is attempting to access the same device) and allow maximum parallelization of the transfer.

3 MPI-IO

In choosing an interface for SIOF that would work well in message passing programs, we examined several parallel file systems. We selected MPI-IO because it offers a good range of parallel I/O features and because it appears to have a good chance of becoming a widely-implemented standard.

MPI-IO was first developed by a group of researchers independently of the MPI Forum. Although the work used many ideas from MPI, it was not a formal part of the standard. However in April 1996, the MPI Forum voted to begin work on an I/O chapter, and the MPI-IO standard was presented as the initial proposal. As a result, while MPI-IO is now more likely to become a standard, its interface may change significantly as more people contribute to it. In this paper, we refer to the version of MPI-IO that existed before the MPI Forum made any changes.

Like most parallel I/O libraries, MPI-IO supports *collective requests,* where multiple processes participate in an operation such as reading or writing a file. In many cases, an I/O system can gather collective requests from multiple nodes into a single I/O request. It can often complete this joint request more efficiently than a group of independent requests. In some implementations of parallel I/O systems, collective requests may perform an implicit barrier synchronization on the participating nodes. The synchronization allows a server to collect data from all the nodes participating in the operation before completing the operation. However, the MPI-IO standard does not require synchronization and warns users not to depend on it.

In a parallel environment, multiple processes can access a file simultaneously. Parallel processes often make interleaved accesses, and they may also access separate portions of the file. Some parallel file systems have an interface that is based on the POSIX [7] standard for file I/O, but this interface is designed for an environment where files are not shared by multiple processes at once (with the exception of pipes and their restricted access possibilities) [11]. Furthermore, POSIX file operations do not allow access to multiple discontiguous parts of the file in a single operation.

MPI uses user-defined and built-in datatypes to describe how data is laid out in a memory buffer. In MPI-IO, datatypes used in this way are called *buffer types*. MPI-IO also uses MPI datatypes to describe the partitioning of file data among processes. A *file type* describes a data pattern that can be repeated throughout the file or part of the file. A process can only access the file data that matches items in its file type. Data in areas not described by a process' file type (holes) can be accessed by other processes that use complementary file types.

MPI associates a datatype with each message. The length of the message is an integral number of occurrences of the datatype. This method of defining a message is more portable than specifying the message length in bytes. Similarly, MPI-IO defines a third datatype called an *elementary type* or *etype*. Both the buffer type and the file type contain an integral number of e-types. This allows offsets into a file to be expressed in e-types rather than bytes. Using MPI datatypes has the advantage of added flexibility and expressiveness, as well as portability.

4 SIOF API architecture

Having chosen MPI-IO as our application programming interface, we designed our implementation with several goals in mind:

- Make efficient use of I/O resources, including the storage devices, the external network, the processing nodes, and HPSS.

- Avoid creating bottlenecks that would limit the scalability of the I/O system.

- Minimize barrier synchronizations among the processes of the application, since these can slow down operation and a risk of deadlock.

This section describes how we designed the SIOF API to achieve these goals.

We consider an application to have one process per node, and in this description we assume that all of the application code executes in one thread per process. The SIOF API spawns several additional threads in each process that share its address space.

The thread executing the application is called the client thread, and each process spawns a server thread when the API is initialized. The client thread includes code that implements the interfaces of the MPI-IO functions. The server thread executes the code that issues requests to HPSS.

One server thread manages a given open file on behalf of all the nodes, and servers on different processors can manage different open files. This prevents any single node from having to manage all the open files. We call this aspect of the architecture the *distributed server model*. Conceptually, the server and client threads could be separate processes, since they share no data structures. However MPI cannot at present direct messages to different processes on the same node, so using MPI for communication requires the server and client to reside in the same process.

Any time a client thread needs to operate on a file, it sends a request via MPI to the server thread on the appropriate node. Each server maintains a table of the open files it manages. When a request arrives, the server looks up the HPSS file descriptor and other information about the file and then spawns a driver thread to issue the HPSS request.

When this request is complete, the driver thread sends a response message to the client and then terminates. The client thread receives the message, and the original MPI-IO call returns a result to the application program.

4.1 Opening and closing a file

Opening a file in MPI-IO is always a collective operation, which means that all the nodes in the program (or a specific subset of them) participate. The nodes select a server by hashing the file name and other parameters to the **open** call to produce a node number. Since all the nodes must specify the same parameters to the call, they will all select the same node without needing to communicate with each other. The server's node number is stored in a local file table on each node for use in future requests.

Each node sends a request to the server as soon as it is ready; there is no barrier synchronization upon opening a file. When the server receives the first **open** request for a given file, it creates an entry in the file table and spawns a driver thread to call HPSS. Subsequent requests from other nodes to open the same file will find a matching request in the file table. If the HPSS **open** call has already completed, the server will send a reply containing the data from the completed (or possibly failed) call. If the HPSS call is still pending, the new request will be placed in a queue. As soon as the driver thread completes the HPSS call, it will send responses to the nodes with queued requests. This arrangement guarantees that each **open** request generates exactly one call to HPSS, and requests from other nodes to open the same file share the results of this call.

Closing a file is also a collective operation, and the nodes again send individual requests to the server. This time, however, the server delays closing the HPSS file until all the requests have arrived. Therefore, closing a file is a synchronizing operation. This is necessary because the file cannot be closed until all the nodes are finished with it, and any errors that occur when HPSS closes the file must be reported to all participating nodes. Moreover, if the close operation does not synchronize, a node might treat a file as if it were closed and its buffers flushed when the file is in fact still open and handling requests from other nodes.

4.2 Reading and writing

Programs can read and write files collectively or independently, and they can intermix these operations freely on the same file (provided that all nodes that open a file participate in the collective operations).

For an independent read or write operation, the client first spawns a mover thread that will copy data between the memory buffer and the network channel to the storage device. When this thread has started, the client sends a read or write request to the server. The request includes the information that the server will need to construct an HPSS IOD (see Section 2). The server spawns a driver thread to issue the HPSS **readlist** or **writelist** call. HPSS transfers the data directly between the node and the storage device and then returns from the **readlist** or **writelist** call. Part of the return data is a structure called an IOR (I/O reply), which the driver thread sends back to the mover before terminating. The driver thread the IOR to its own record of the transfer, then returns status information to the client thread and terminates. The SIOF API code in the client thread transforms the status information into MPI-IO return data before finally returning from the MPI-IO call.

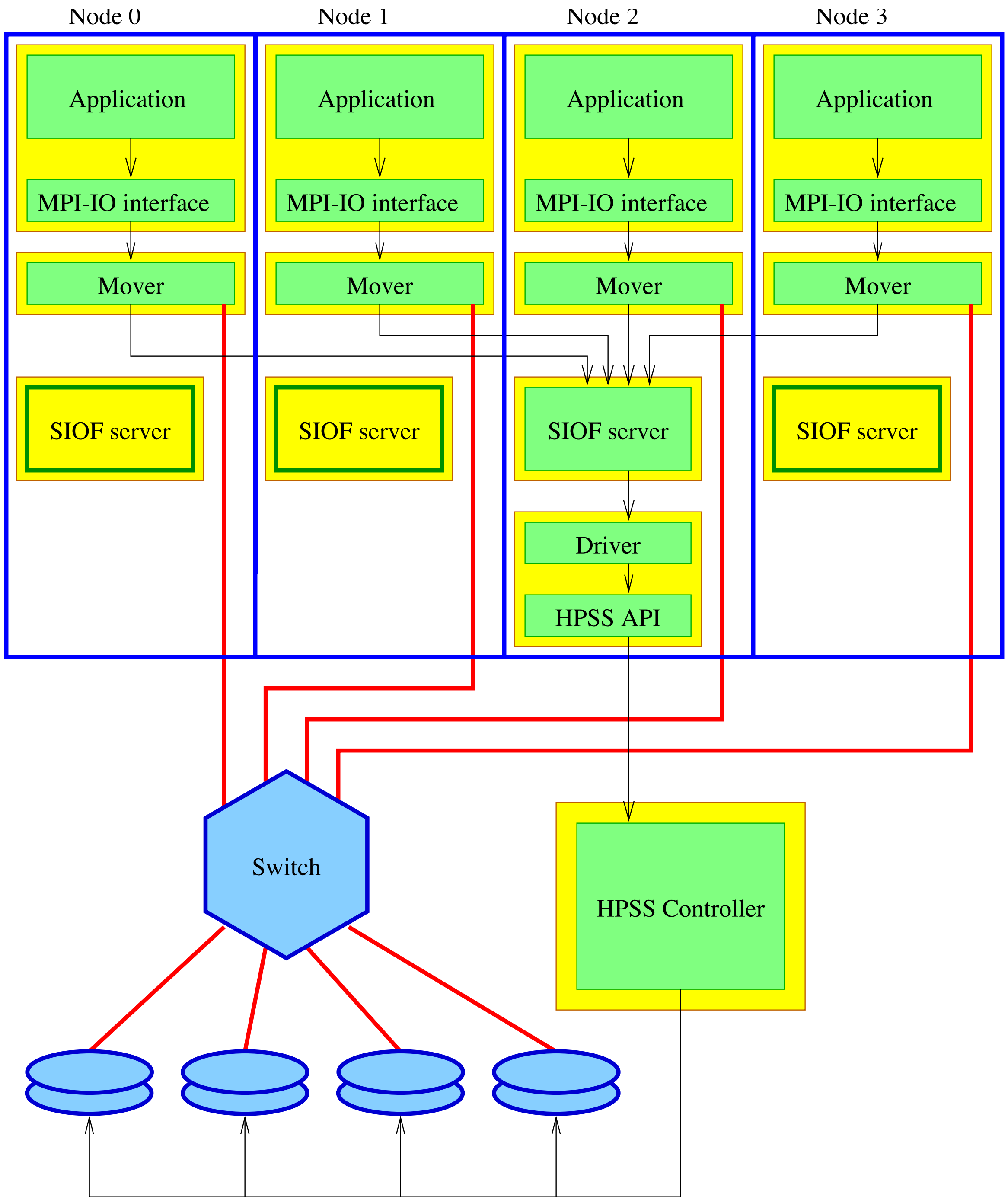

Figure 3: The SIOF API is implemented in several threads on each node (shown here as four large, vertical rectangles). Outer shaded boxes represent threads; inner boxes are functional modules within a thread. Boxes with heavy outlines show servers not participating in an operation. For read and write operations, control is centralized at a server thread, but data travels through separate, high-bandwidth channels between devices and compute nodes.

Collective operations require a few extra steps. The details appear in Section 5, but the main difference from independent operations is that the server may gather up several requests from different nodes and issue them together in a single HPSS call, as depicted in Figure 3.

4.3 File types and buffer types

Section 3 noted that MPI-IO programs can use file types and buffer types to access discontiguous regions of data. MPI-IO translates these datatypes into an internal format called a *chunk map*. A chunk map is a list of contiguous data blocks, and it contains only the information that the SIOF API needs from an MPI datatype to construct an IOD.

Because MPI specifies no functions for accessing the layout information in a datatype, the SIOF API code must explicitly read the internal data structures of the MPI implementation on which it is based (MPICH [1]). One reason for using chunk maps is to isolate the system-dependent code as much as possible, so most of the SIOF API code works with chunk maps rather than MPI datatype structures.

The SIOF API stores the chunk map of the file type for each node and each open file in the server thread's file table. When a file is read or written, the server constructs an HPSS IOD for the data to be transferred, with source and sink mappings for each contiguous chunk of data to be accessed. It passes this IOD to a single HPSS call.

Meanwhile, the mover thread parses the chunk map corresponding to its node's buffer type to determine which data to access in memory. The SIOF API does not compare buffer types with file types or decompose them with respect to each other; HPSS and the client mover thread can each behave as if the other is accessing a single, contiguous stream of data.

5 Managing collective operations

The SIOF API currently supports four types of data access: the independent **read** and **write** operations, and collective versions called **read-all** and **write-all**. Structuring the server to permit collective operations on reads and writes requires that several issues be addressed:

- How are collective operations implemented?
- How is the decision made to dispatch them?
- What optimizations are available for collective operations?

This section discusses these three issues.

5.1 Collective implementation

A group of client nodes initiates a collective operation by sending requests to the server. The server queues these requests as they arrive in a *data access list*, which it traverses either after the receipt of a client's message for a file operation or after a predetermined period of time has elapsed, whichever comes first. As the server traverses the list, it updates a *dispatch priority* for each pending collective operation. The dispatch priority determines when the server will initiate the data access; if the priority is over a predetermined threshold, the server spawns a thread to issue the HPSS **readlist** or **writelist** call. If a collective write request includes overlapping file accesses by different nodes, the server constructs an IOD that resolves the conflict in a well-defined way.

The data access list also records the number of outstanding clients, which is needed to handle cases where the server dispatches a request before all clients have checked in. The number of clients is initially the number of nodes that have jointly opened the same file, but if two or more dispatches are used for the same operation, it will be the number of clients remaining for the operation (i.e., the number not already checked in and previously dispatched).

5.2 Determining dispatch priority

How the dispatch priority is determined will have a strong effect on performance and utilization of the I/O system. For example, one can imagine a scenario in which 15 clients of a 16-client application check in at nearly the same time, but the 16th client checks in much later. In such a scenario, it may be more efficient to dispatch the 15 pending requests as a group and wait for the 16th request separately. On the other hand, issuing a request too soon will reduce the ability of the SIOF API library to amortize latency costs involved in setting up a data access. A number of factors may play a part in determining the dispatch priority. At the present, our implementation for MPI-IO **read-all** and **write-all** operations blocks until all client nodes have checked in. However, we plan to investigate several algorithms to determine their effect on utilization and performance. These algorithms will consider, to varying degrees, the time since a request was first issued, information on which clients have checked in, the transfer size, the granularity of the file types, and whether the access is to tape or disk.

5.3 Optimizations

The architecture of the SIOF API makes several optimizations feasible. These include:

- Asynchronous operation.
- Grouping accesses on the same storage device.
- Grouping accesses on the same processor.
- Coalescing small accesses.

The first optimization reduces a server's sensitivity to the latency of HPSS calls. By spawning a thread for each such call, the server can handle multiple requests concurrently. (However independent requests for access to the same file are serialized to preserve atomicity.)

Grouping accesses to the same storage device can help improve cache performance. For example, certain decompositions of matrices among processors can produce requests for small, interleaved chunks of data [4]. By constructing IODs so that requests for sequential data appear in order, the server can increase the probability of cache hits on a disk. On the other hand, sending small blocks of data between one disk and multiple nodes in round-robin order may produce excessive switching latency in the external network. In that case, it may be better to group requests so that data residing on one node is accessed sequentially. Performance tuning will help us determine how best to arrange the parts of a collective request.

Even if there is no locality to be exploited in a collective operation, grouping requests from multiple nodes into a single **readlist** or **writelist** call can amortize one-time expenses incurred in I/O operations, such as the cost of an RPC transaction between the parallel computer and the HPSS controller.

6 Current status and future work

We demonstrated a prototype of the SIOF API at the Supercomputing '95 conference. This version included independent and collective reading and writing with independent file pointer and explicit file offsets. The prototype supported file types and buffer types using any MPI datatype. We are currently working on a production quality version of the SIOF API that will implement the full MPI-IO specification. We plan to include this interface in a future release of HPSS.

Meanwhile, the MPI-IO specification itself is changing. The MPI Forum continues to work on a standard for parallel I/O, and we plan to incorporate whatever changes they make to MPI-IO.

In addition to the interface changes, we are also considering structural changes to the SIOF API that will improve its efficiency and robustness. The robustness changes will mainly involve implementing the MPI exception handling mechanism.

To improve efficiency, we would like to reduce the amount of processing that the server threads do. For example, we are considering whether independent read and write requests could be sent directly from the requesting node to the HPSS server instead of being routed through the server node for the open file. Implementing this change would require finding a way for multiple nodes to access the same HPSS file correctly and efficiently. Another change we are considering is to use the **MPI_Reduce** command to combine requests from multiple nodes for a single collective operation. At present, when a group of nodes issues a collective write request, for example, the server receives messages from each requesting node and builds up a description of the operation incrementally. As an alternative, **MPI_Reduce** could use a customized combiner function to distribute the work of creating a collective request over many nodes. The server node would then receive a single, merged request instead of many separate parts. The reduction in memory management and message handing activity on the server node would be significant, but this approach would cause collective operations to synchronize the nodes. For loosely-synchronous programs, the synchronization delays could outweigh other gains. Therefore, we will have to consider carefully the merits of implementing this strategy.

7 Conclusion

The SIOF API is a new implementation of the proposed MPI-IO standard. It is designed as a high-level user interface for the HPSS file system, and its initial implementation is on a Meiko CS-2 parallel computer. Because HPSS supports third-party transfers over an external network, our implementation can transfer data in parallel between processors and storage devices while presenting a global view of the file system that all nodes can access. Our distributed server model spreads the burden of coordinating data transfers over multiple nodes. Control of a given open file is centralized, but data transfer can proceed in parallel. We believe this combination of features will offer the high aggregate I/O bandwidth for large data transfers that many parallel scientific codes need.

References

[1] P. Bridges, N. Doss, W. Gropp, E. Karrells, E. Lusk and A. Skjellum, Users' Guide to MPICH, A Portable Implementation of MPI. http://www.mcs.anl.gov/-mpi/mpi/mpiuserguide/paper.html.

[2] R. Coyne, H. Hulen, and R. Watson, The High Performance Storage System, in *Proceedings of Supercomputing '93,* November 1993.

3. Fibre Channel Association, *Fibre Channel: Connection to the Future,* Austin, Texas, 1994.

[4] D. Kotz and N. Nieuwejaar, Dynamic file-access characteristics of a production parallel scientific workload, ⊡ *Proceedings of Supercomputing '94,* November 1994.

[5] MPI-IO Committee, MPI-IO: A Parallel File I/O Interface for MPI, Version 0.5. http://lovelace.nas.nasa.gov/MPI-IO.

[6] Open Software Foundation, *OSF DCE Application Development Reference,* Prentice-Hall, Englewood Cliffs, N.J., 1993.

[7] *Portable Operating System Interface (POSIX)—Part 1: System application programming interface (API).* IEEE Standard 1003.1-1990.

[8] Scalable I/O Facility. http://www.llnl.gov/liv_comp/siof/siof.html.

[9] Scalable I/O Initiative. http://www.cacr.caltech.edu/SIO/.

[10] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI: The Complete Reference,* MIT Press, Cambridge, Mass., 1996.

[11] W. R. Stevens, *Unix Network Programming,* Prentice-Hall, Englewood Cliffs, N.J., 1990.

[12] D. Teaff, R. W. Watson, and R. A. Coyne, The architecture of the High Performance Storage System (HPSS), in *Proceedings of the Goddard Conference on Mass Storage and Technologies,* March 1995.

[13] Transarc Corporation, Encina product information. http://www.encina.com/-Public/ProdServ/Product/Encina.

[14] R. Watson and R. Coyne, The parallel I/O architecture of the High Performance Storage System (HPSS), in *IEEE Symposium on Mass Storage Systems,* September 1995.

[15] D. Wiltzius, Network-attached peripherals (NAP) for HPSS/SIOF. http://www.llnl.gov/liv_comp/siof/siof_nap.html.