BMI: A Network Abstraction Layer for Parallel I/O

Philip Carns
Parallel Architecture Research Laboratory
Clemson University
Clemson, SC
pcarns@parl.clemson.edu

Walter Ligon III
Parallel Architecture Research Laboratory
Clemson University
Clemson, SC
walt@parl.clemson.edu

Robert Ross
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL
rross@mcs.anl.gov

Pete Wyckoff
Ohio Supercomputer Center
Columbus, OH
pw@osc.edu

Abstract

As high-performance computing increases in popularity and performance, the demand for similarly capable input and output systems rises. Parallel I/O takes advantage of many data server machines to provide linearly scaling performance to parallel applications that access storage over the system area network. The demands placed on the network by a parallel storage system are considerably different than those imposed by message-passing algorithms or datacenter operations; and, there are many popular and varied networks in use in modern parallel machines. These considerations lead us to develop a network abstraction layer for parallel I/O which is efficient and thread-safe, provides operations specifically required for I/O processing, and supports multiple networks. The Buffered Message Interface (BMI) has low processor overhead, minimal impact on latency, and can improve throughput for parallel file system workloads by as much as 40% compared to other more generic network abstractions.

1. Introduction

Parallel I/O has become an increasingly important aspect of high performance computing, especially as advances in processing power have steadily outpaced advances in disk throughput. In recent years, parallel computers, including commodity clusters in particular, are being scaled dramatically in order to meet the need for greater computing resources. Parallel I/O throughput must scale accordingly to insure that these systems perform well for real world applications.

These circumstances have led to renewed research interest in the area of scalable file systems for parallel applications. In particular, new file systems such as the Parallel Virtual File System 2 (PVFS2) [12] have been created with the intention of addressing the needs of next generation systems. PVFS2 focuses on insuring robust, scalable operation while keeping the infrastructure flexible enough to adapt to new technologies. Several components are necessary in order to make this happen. One such component is an efficient network abstraction layer that allows a file system to operate transparently on top of a variety of interconnection networks. Many general purpose network abstractions already exist for high performance computing, but none have been tailored explicitly to the unique requirements of the parallel I/O domain.

In this paper, we first present an overview of PVFS2 to serve as background for the problem domain. We then describe why a special purpose network abstraction is necessary for parallel I/O. Section 2 introduces the Buffered Message Interface (BMI) as a solution to this problem, focusing on how it meets the challenges of parallel I/O as stated earlier. We will also offer examples of how it maps onto real world PVFS2 operations. Section 3 briefly describes the challenges involved in implementing support for various networks. Section 4 provides preliminary performance results of BMI over the GM protocol with a particular emphasis on network patterns common to PVFS2. Section 5 describes various related works and outlines why alternatives to BMI are not suitable for the problem domain covered in this text. The paper ends with concluding statements

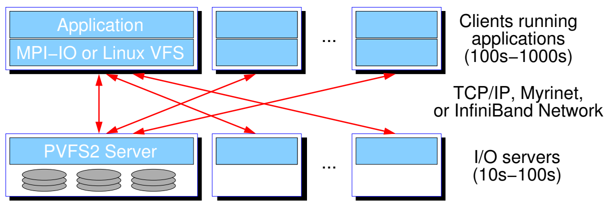
Application
MPI–IO or Linux VFS
...
Clients running applications (100s–1000s)
TCP/IP, Myrinet, or InfiniBand Network
PVFS2 Server
...
I/O servers (10s–100s)

Figure 1. High level PVFS2 system diagram

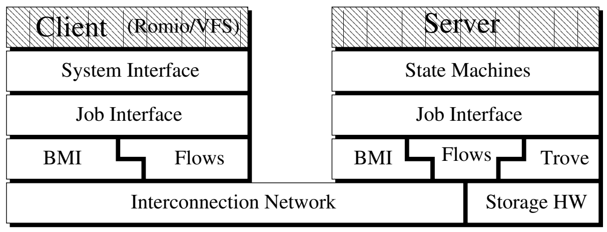

Figure 2. Primary PVFS2 I/O components

and a summary of intended future work.

1.1. PVFS2

PVFS2 is a new file system design from the Parallel Virtual File System project team. It incorporates lessons learned from the original PVFS file system [4] and emphasizes the ability to adapt to new technology by way of a highly modular infrastructure. PVFS2 uses a client/server architecture, with both the server daemon and client side libraries residing fully in user space. There may be any number of servers, and each server may provide either metadata, file data, or both. Metadata refers to attributes such as timestamps and permissions as well as file system specific parameters. File data refers to the actual data stored in the system. This data is distributed according to rules that are selectable by the user. The default scheme is to stripe data evenly in a similar manner to that of a RAID array. Metadata may also be distributed, though at the granularity level of one server per individual file or directory.

A high level diagram of the PVFS2 system architecture is shown in Figure 1. Note that there is no need for a shared storage infrastructure; each server manages its own local resources. The arrows represent communication within the file system. There is no communication between servers or between clients. Clients communicate exclusively with the servers responsible for the resources that they wish to access.

Figure 2 shows a diagram of the primary internal I/O components of PVFS2. The lowest level network abstraction is provided by a component known as the Buffered Message Interface [3]. The counterpart disk abstraction, which provides both stream and key/value style access to local storage resources on each server, is called Trove. These two components are coordinated by Flows, which handle buffering, scheduling, and datatype processing between network and disk for bulk transfers. All of these components (along with other peripheral components beyond the scope of this paper) are coordinated by the Job interface, which manages threading and provides a consistent interface for testing of completion of any pending low-level I/O operation, regardless of which underlying component is ultimately responsible for it. Both the servers and client libraries are implemented through the use of concurrent state machines which operate on top of the Job interface.

The following list summarizes the critical requirements that we have identified for a network abstraction layer in this problem domain:

- efficiency: necessary to achieve the performance goals of the file system
- parallel I/O access pattern support: to accommodate the discontiguous and highly concurrent network accesses seen in parallel I/O workloads
- multiple concurrent networks: allows the file system to operate on top of a variety of interconnects
- thread safety: to support high throughput multi-threaded environments on file servers
- explicit buffer management: optionally take advantage of user level or RDMA networks which manipulate pinned memory regions
- client / server model: necessary for a dynamic application and server interaction
- fault tolerance: ability to isolate network errors and integrate with file system level fault tolerance scheme
- minimal exposed state: to enhance scalability
- simple high level API: to reduce complexity of the file system

2. The Buffered Message Interface

The Buffered Message Interface was designed to meet the requirements for a parallel file system as outlined in the previous section. It is a software component intended for use by system level services. BMI uses a layered interface model; it presents a high level API for BMI users while also providing an internal device API for specific network implementations. The latter interface eases the task of porting to new network infrastructures. Each device resides in an independent module. In this section we will describe the design and implementation aspects of BMI that enable it to accommodate the requirements of the parallel I/O problem domain.

2.1. API

Many of the requirements listed in section 1.1 are met by way of intelligent API decisions. BMI implements a nonblocking interface for all network I/O operations. The basic model is to first *post* an operation and then *test* the operation until it is completed. Completion in this model refers to *local* completion; it offers no guarantee of success on the remote peer. The nonblocking interface allows many network operations to be in service concurrently, each possibly in a different stage of communication. Operations are referenced by unique identifiers while they are in service.

BMI embraces the client/server model used by parallel file systems through the use of *tags* and *unexpected messages*. Tags are integer parameters which can be used to match messages exchanged as part of a single overall file system operation. Normally, incoming messages are paired with receive operations with the proper sender, size, and tag parameters. Unexpected messages are an exception, however, in that they do not require a matching receive to be posted. Instead, the receiver simply polls to check for new unexpected messages. If such a message is found, then a descriptor is filled in that describes the parameters of the message and provides a pointer to the data buffer. This reduces complexity on the server side because the server does not have to anticipate buffer use in advance. Instead it can just react to incoming messages and use them to initiate service state machines.

The BMI API also allows multiple application or server components to use the same interface concurrently. BMI supports this foremost by being fully thread safe. Secondly, it introduces *contexts* to help differentiate between independent higher level callers. Each component that uses BMI will receive its own unique context, which is local to that host. This context can then be used to differentiate between operations posted by each component, both at post time and at test time. This allows components (or threads) to test for completion of any pending operations without the risk of receiving notification of completion for an operation posted by a different component.

A clean separation of the module API and the user API allows BMI to operate on top of a variety of network protocols. New modules can be activated dynamically as needed. The core BMI code multiplexes time between active modules to insure that progress is made on multiple networks if they are used simultaneously.

The BMI API also strives to reduce complexity for components which are built on top of it. There is no explicit queue management. Receive buffers do not need to be posted in advance of communication, though doing so will improve performance for some networks. All BMI modules must provide implicit flow control. Opaque address references are resolved from human readable URL style references. The syntax allows specification of multiple network addresses for each host. These decisions isolate network complexity from the core functionality of the parallel file system.

2.2. Performance

BMI implements several features that are intended to improve efficiency. Some of these are simply optimizations on the basic API model outlined earlier. One important optimization on the post and test model is that any post function may elect to indicate *immediate completion* at post time. Immediate completion means that the operation has successfully finished during the execution of the post call; therefore, no testing step is necessary. In some scenarios, such as very small sends, or receives for which the data has already been buffered, this will avoid the overhead of calling an extra function to retrieve the status information.

Many modern user level network protocols as VIA [14] or GM [11] rely on the use of message buffers that are pinned into physical memory before transmission. BMI accommodates this by providing functions for allocating and releasing buffers that are optimized for use by BMI. The use of these functions is optional, however, and BMI will handle buffering internally if needed. This is important for client library usage in which there is no opportunity to register buffers in advance.

PVFS2 supports the use of arbitrary data types to describe patterns of offsets and sizes within a file. It also allows data to be striped across an arbitrary number of hosts. These two features lead to scenarios in which communication must be carried out from a set of many noncontiguous buffers. BMI allows sets of noncontiguous buffers to be sent or received in a single function call through the use of *list* operations. List operations are similar to their traditional send and receive counterparts, except that they operate on an array of memory offsets and sizes rather than a single buffer. This can cut down drastically on the number of messages necessary to transfer a complex data pattern between two hosts. Some BMI implementations may directly support list operations and use hardware-provided scatter/gather support to move the noncontiguous buffers.

2.3. Scalability

The ability to handle a large number of concurrently posted operations is critical to file system scalability. The *user pointer* field associated with each operation is one feature designed to help in that regard. The user pointer is an opaque value that may be set by the caller at post time. It is returned unchanged to the caller when a test indicates completion. It provides a mechanism for the caller to map completed operations back to some higher level data structure outside of BMI after calling a test function. For example, on the server side it may map the operation back to a state machine that tells the server what to do next. Thus, no matter how many operations are in flight, the originating function or data structure can be located for each completion with O(1) complexity.

It is also important for scalability to insure that the caller does not have to execute a test function separately for each pending operation. This would consume too much CPU time even with just a few posted operations. There are two variations on the test function that overcome this problem. One variation allows a single call to test for completion of any of a set of specified operations. Another variation allows a single call to test for completion of any previously posted operation without specifying the operation identifiers. This last function is significant because it prevents a busy server or library from having to construct a list of operations to test on; instead, it just checks for any possible operation that may have completed in a single function call.

A final key to BMI API scalability is that it is connectionless. There is no state to maintain for a given peer on the network, and no connection to set up or tear down in preparation. This simplifies communication and aids in scalability when communicating with thousands of hosts. Note that if BMI is built on top of a network that uses a connection oriented model, then BMI will manage the connections transparently underneath the API, likely by caching previously used connections in hope of later reuse.

2.4. Fault Tolerance

Clean handling of file system and network faults is necessary for modern large scale file systems. BMI addresses this issue by working in concert with fault handling capability at higher levels of the file system. For example, BMI does not automatically retry transmission of failed network messages. This is impractical in the general case because network messages in parallel file systems are typically just a single part of a larger multistep operation. Automatic retransmission at the network API level could lead to inconsistent requests if a server is restarted or fails over, or it could simply lead to duplicate operations. For this reason, BMI relies on the server or client libraries to determine the appropriate retransmission points. It accommodates this decision by preserving network address information and transparently reconnecting or utilizing secondary network interfaces as needed. BMI also improves fault tolerance by providing the ability to cancel network operations that have been posted but have not yet completed, therefore giving higher level components a clean interface to handle time out conditions.

3. BMI Module Implementation

BMI has been implemented and used extensively as the networking foundation for the PVFS2 file system. BMI currently supports three different protocol modules: TCP/IP, GM, and InfiniBand [1]. These three were chosen for the initial implementation because they represent some of most popular cluster interconnects and because they demonstrate how to implement BMI modules for a variety of dissimilar networks. We also sought to choose interconnects that would help to validate the generality and efficiency of the BMI model.

The TCP/IP module was implemented first, and posed particular challenges. First of all, TCP/IP sockets are not connectionless or message oriented, so these features had to be emulated and hidden from the user. This is done with a socket management layer that tracks existing sockets, opens new ones as needed, and polls over them to determine when to push more data. This socket management layer can optionally utilize the Linux epoll() interface, when available, to improve scalability with large numbers of open connections. All communication is done in nonblocking mode. FIFO queues are used to preserve message ordering and insure that message boundaries are observed. Small incoming messages may be buffered on the receiving side, while larger messages require that a receive is posted before data begins transmitting in order to insure that adequate memory space is available.

The GM and InfiniBand modules are similar to each other in many ways. The primary challenges for these protocols revolve around memory management. Both require receives to be posted in advance, which leads to the use of a combination of rendezvous and eager message modes [8], as well as unique flow control schemes. Both also require the use of pinned memory buffers, which must be handled within the module if the caller does not provide regions that are already pinned for communication. The GM module uses a pool of internally registered memory buffers as intermediate areas for communication in this case. These intermediate buffers also provide a means to handle discontiguous access by packing or unpacking during the memory copy phase. The GM module makes use of the message priority facility in GM to differentiate control messages from data payload messages, and uses FIFO queues to constrain resource (memory or token) utilization.

4. Performance Results

In this section we analyze the performance of BMI for a sampling of workloads that are relevant to the parallel I/O domain. The TCP/IP module is the most mature and optimized of the three modules. However, the experiments in this text will be carried out using the GM module in order to

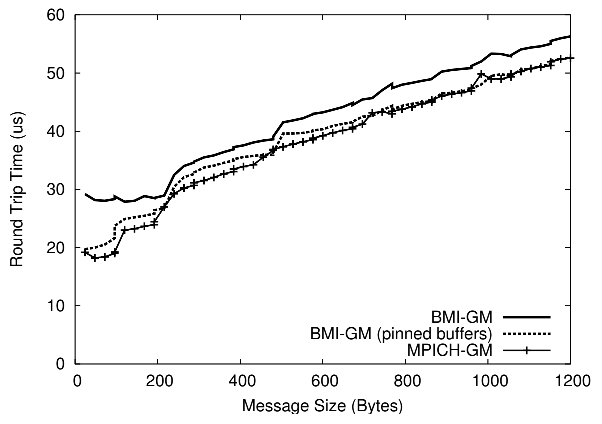

Figure 3. Round trip latency with various message sizes

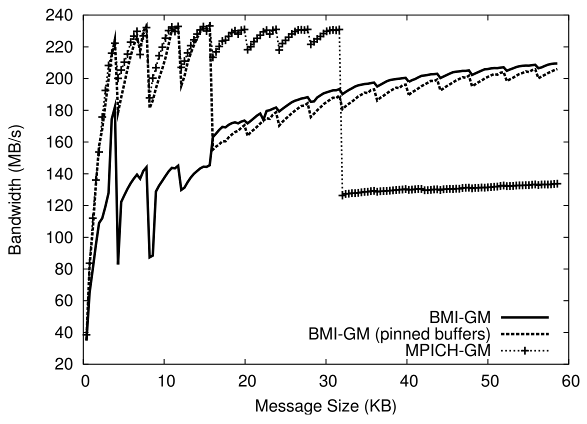

Figure 4. Point-to-point bandwidth for 120 MB transfer with various message sizes

reflect the performance typical of higher speed system area networks. Our intent is to focus on high level BMI characteristics rather than the nuances of individual modules. We will use MPICH-GM as a reference point for comparison in all experiments. MPICH-GM is a mature MPI implementation based on MPICH [8] that utilizes the GM API.

All tests were carried out using the Jazz Linux cluster at Argonne National Laboratory. Each compute node consists of a single Pentium Xeon 2.4 GHz with at least one gigabyte of RAM. The nodes are connected by both a 100 Mb/s Ethernet network and a 2 Gb/s Myrinet 2000 network. The system software includes Linux kernel version 2.4.29, GNU C library version 2.2.4, and MPICH-GM version 1.2.6..13b. All test programs were compiled with gcc using "-O3" as the only compiler switch.

The test programs were implemented from scratch because no existing network benchmarks are capable of exercising the BMI application interface. The performance tests are separated into two categories. The first category consists of point-to-point comparisons between BMI and MPI. The primary purpose of these tests is to establish the baseline overhead introduced by the BMI in comparison to a vendor optimized network abstraction. The second category also compares BMI to MPI, but focuses on network patterns that emulate the behavior of parallel file systems with multiple clients and servers.

4.1. Point-to-point Benchmarks

The first experiment measures the round trip latency of both BMI-GM and MPICH-GM when using nonblocking API functions. The round trip latency may give some indication of the minimum time required to complete a request/acknowledgment pair in PVFS2 on a Myrinet network. Figure 3 shows the results of this benchmark as the message size is varied from 24 to 1,200 bytes. This range of message sizes reflects the typical range of request protocol messages exchanged by PVFS2. For BMI, we show the results of experiments both with and without memory buffers explicitly pinned in advance. MPICH does not provide an explicit mechanism for pinning memory buffers.

We see in these results that BMI-GM without explicit memory management exhibits a round trip time that is 3 to 10 μs slower than MPICH-GM. This is because MPICH-GM takes advantage of an internal memory registration cache that allows it to avoid much of the overhead of memory deregistration or buffer copying during the duration of the experiment. BMI-GM cannot leverage this technique because it would conflict with the cache used by MPICH-GM when the two are linked together. A shared cache used by both components would be necessary in order to overcome this implementation obstacle.

Fortunately, PVFS2 almost exclusively uses buffers that have been registered in advance for request protocol communication. All PVFS2 messages are encoded into an architecture independent network format prior to transmission. This provides a convenient opportunity to place the data into an optimized buffer. The results for BMI-GM with pinned buffers more accurately represent this scenario. In this case BMI-GM closely approximates the round trip latency performance of MPICH-GM. For messages larger than 700 Bytes the latencies are nearly identical.

The second experiment measures the bandwidth between two hosts communicating via BMI or MPI. Nonblocking calls are used in both cases. All sends and receives are posted at the beginning of the test run. Figure 4 shows the results of this test. The bandwidth is computed as the total amount of data transferred divided by the time required to complete the transfer. This experiment may provide an estimate for the upper limit on performance of an I/O operation between a single client and server in a PVFS2 file system. However, it should be noted that PVFS2 defaults to using relatively large (at least 64 KB) messages for large contiguous I/O operations.

Figure 4 reveals several interesting trends. First of all, BMI-GM switches between eager and rendezvous mode messaging at the 16 KB message size point, while MPICH-GM switches at 32 KB. This accounts for the large throughput changes at those points on the graph and may indicate a potential tuning parameter for future BMI-GM experiments. Once MPICH-GM and BMI-GM are both using rendezvous mode, the BMI-GM implementation demonstrates a performance improvement of over 50% even without advance registration of memory buffers. The results are much different in eager mode. BMI-GM without explicit registration of memory regions is very erratic in eager mode. As in the round trip latency measurements, the BMI-GM eager mode bandwidth with registration comes close to the performance of MPICH-GM but does not quite match it.

4.2. Parallel I/O communication patterns

Although the point to point experiments of section 4.1 are useful to some extent in measuring baseline performance, they do not necessarily reflect the usage patterns for which BMI was designed. To capture that environment, we must use a benchmark that emulates the network patterns seen in common parallel file system operations. We therefore chose to construct a benchmark that replicates the network activity resulting from a relatively large contiguous parallel read from a set of file servers. The benchmark can run on an arbitrary number of hosts which are divided into N servers and M clients. Each of the M clients must send a request and receive an acknowledgment from each of the N servers. Each client will then receive a stream of data from each server. The stream of data is broken into individual messages to facilitate overlap with disk access and to accommodate the default PVFS2 data distribution pattern. This complete simulation results in $N \times M \times 2$ small messages for the request and acknowledgment phase, followed by the transmission of many large messages for the actual file data. This benchmark includes no disk activity.

Four key points differentiate this test from the earlier point to point experiments of section 4.1:

- unexpected messages: PVFS2 requests are received by servers as unexpected messages because we do not know in advance which clients will be active, or how big their request packets will be. We emulate this capability in MPI by using the MPI_Iprobe() and MPI_Recv() functions.

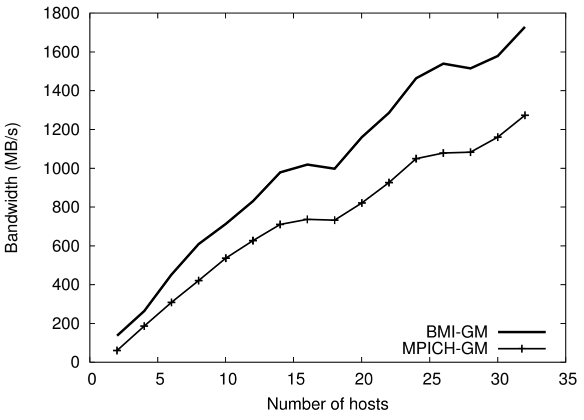

Figure 5. Aggregate read pattern, 10MB per client/server pair

- discontiguous buffers: PVFS2 by default uses bulk data messages with a maximum buffer size of 256 KB. However, these messages must be broken into separate 64 KB buffers on the client side to accommodate the striping of data in 64 KB increments across servers. The alignment and size of these buffers depends on the exact offset and size of the I/O operation, so we handle them using list I/O operations in BMI and hvector datatypes in MPI.
- concurrent operation: Every client and server in PVFS2 must process a large number of concurrent messages from different hosts during the course of a large read operation. There is no predetermined aggregate structure that would allow for the use of collective operations. BMI manages pending operations using a variant of the test function that checks for completion of any previously posted operation, BMI_testcontext(), while MPI uses MPI_Testsome().
- memory registration: To emulate the behavior of PVFS2, all requests and acknowledgment messages are transmitted to and from memory regions that have been pinned in advance when possible. The server processes also use pinned buffers for bulk I/O. The client processes do not use preallocated regions, primarily because the target buffers in this case represent application memory buffers for a file system read operation, which are not controlled by the PVFS2 library.

For this experiment, we chose a request size of 25 bytes and a response size of 400 bytes to approximate typical request and response sizes in PVFS2. We also chose for each client/server pair to exchange 10 MB of data messages. This results in a total aggregate data transfer size of

N × M × 10 MB. Figure 5 shows the results of this experiment. The total number of hosts is varied between 2 and 32, evenly divided into client and server classifications at each data point. The aggregate bandwidth is computed as the total amount of data (not counting requests and acknowledgments) that is transferred divided by the time required for the slowest of the M concurrent clients to finish its portion of the transfer.

At the single server, single client data point, the BMI-GM throughput is more than twice that of MPICH-GM. As more hosts are added, the difference varies between 35% and 42%. The BMI-GM performance eventually reaches an aggregate value of 1728 MB/s for 16 servers and 16 clients. These improvements are largely due to specific features of BMI that address the needs of the parallel I/O environment, including a lightweight mechanism to test for completion of many pending operations, flexible handling of different types of memory buffers, efficient support for discontiguous memory regions, and an API that explicitly manages unexpected messages.

5. Related Work

The fields of message passing and high-performance I/O have fostered many research efforts into special purpose network abstractions. Three projects of particular relevance are summarized below.

The Message Passing Interface (MPI) [9] is a specification for application level message passing. Although many concepts from the MPI API and model were adopted for use in BMI, there are a number of practical concerns that precluded the use of MPI as our messaging system. At a high level, it did not seem appropriate to force all applications interacting with the file system to be MPI applications. Also current MPI implementations do not deal well with faults in the network or nodes, often causing the termination of entire groups of processes: it is key that the impact of faults be minimized in a file system. Finally, the dynamic process components of many MPI implementations are still under development, making it difficult if not impossible to connect clients to servers at runtime.

The same arguments preclude the use of the MPICH or MPICH2 ADI interfaces [8] [10]. The ADI is an abstract device interface upon which these popular MPI implementations are built. This interface relies on many MPI constructs and is not designed for use in environments as dynamic as that of a parallel file system. At the time of this writing they also lack features that are needed for efficient mapping to PVFS2 usage, such as user pointers, unexpected messages, and thread safety.

uDAPL and kDAPL [6] are user level and kernel level (respectively) APIs created by the Direct Access Transport Collaborative industry group to provide a common interface for RDMA based networks. Examples of such networks include InfiniBand, VIA, and iWARP [13]. The work of the DAT Collective is meant to be leveraged in several areas, the most prominent of which is the Direct Access File System [5] protocol. uDAPL is also being used as a starting point for the Interconnect Transport API [7] to be specified as a standard by the Open Group. uDAPL and kDAPL share many features with BMI, such as efficiency, discontiguous buffer support, and thread safety. They were also clearly designed with file system access in mind. However, these APIs are not appropriate for PVFS2 because they require RDMA support and would not map well to commonly deployed legacy protocols such as TCP/IP. uDAPL and kDAPL also present a relatively low level interface and lack features such as unexpected message support. uDAPL was not publicly available when BMI development began.

Portals [2] is a message passing architecture developed by Sandia National Laboratory and the University of New Mexico. It was originally targeted for the Puma lightweight kernel, but has since been ported to Linux. Portals emphasizes scalability and ability to operate on a variety of parallel computer networks. It features the ability to directly access remote memory without operating system intervention, and is designed to be able to make progress on communication without application processing. However, Portals exposes a low level interface which requires management of event queues and remote memory addresses. Portals also lacks flow control which must be provided by the user. Like uDAPL, Portals also was not publicly available when BMI development began.

6. Conclusions and Future Work

We have found that it is critical to select a network abstraction layer that is appropriate for the system software that will use it. In particular, we identified the key requirements of the parallel I/O problem domain, and implemented the Buffered Message Interface to meet those requirements. It performs well for the type of network workloads common to high volume parallel file systems while introducing little overhead to the communication path. BMI already serves as the networking foundation for the PVFS2 file system and has demonstrated how file system operations can be efficiently mapped to a high performance API.

BMI also serves as a tool to explore further networking research in the context of parallel I/O without imposing fundamental changes upon the file system itself. In particular, we intend to implement more network modules, examples of which might include a shared memory module or a reliable UDP module. There is also room for optimization of existing modules through techniques such as memory registration caching.

Acknowledgments: We gratefully acknowledge use of "Jazz," a 350-node computing cluster operated by the Mathematics and Computer Science Division at Argonne National Laboratory as part of its Laboratory Computing Resource Center.

References

[1] InfiniBand Trade Association. InfiniBand architecture specification, release 1.0, October 2000.

[2] Ron Brightwell, Rolf Riesen, Bill Lawry, and A. B. Maccabe. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, April 2002.

[3] Philip H. Carns. Design and analysis of a network transfer layer for parallel file systems. Master's thesis, Clemson University, Clemson, SC, December 2001.

[4] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[5] DAFS Collaborative. Direct access file system protocol, v1.0, August 2001.

[6] DAT Collaborative. uDAPL and kDAPL API specification, v1.0, 2002.

[7] Interconnect Software Consortium. Interconnect Transport API specification. `http://www.opengroup.org/icsc/`.

[8] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message-passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[9] Message Passing Interface Forum. MPI-2: A Message Passing Interface Standard. *High Performance Computing Applications*, 12(1–2):1–299, 1998.

[10] MPICH2. `http://www.mcs.anl.gov/mpi/mpich2/`.

[11] Myricom, Inc. The GM message passing system. `http://www.myri.com`.

[12] Parallel Virtual File System 2. `http://www.pvfs.org/pvfs2`.

[13] James Pinkerton, Ellen Deleganes, and Michael Krause. Internet draft: Sockets Direct Protocol (SDP) for iWARP over TCP. `http://www.ietf.org/internet-drafts/draft-pinkerton-iwarp-sdp-01.txt`, September 2004.

[14] VI Architecture specification revision 1.0. `http://www.viarch.org`, December 1997.