Using MPI in High-Performance Computing Services

Judicael A. Zounmevo[†] Dries Kimpe[§] Robert Ross[§]
Ahmad Afsahi[†]

[†]ECE Dept., Queen's University 19 Union Street, Kingston ON, K7L 3N6, Canada
{judicael.zounmevo, ahmad.afsahi}@queensu.ca

[§]Argonne National Laboratory, 9700 South Cass Avenue Argonne, IL 60439, USA
{dkimpe,rross}@mcs.anl.gov

ABSTRACT

The Message Passing Interface (MPI) is one of the most portable high-performance computing (HPC) programming models, with platform-optimized implementations typically delivered with new HPC systems. Therefore, for distributed services requiring portable, high-performance, user-level network access, MPI promises to be an attractive alternative to custom network portability layers, platform-specific methods, or portable but less performant interfaces such as BSD sockets. In this paper, we present our experiences in using MPI as a network transport for a large-scale, distributed storage system. We discuss the features of MPI that facilitate adoption as well as challenges and recommendations.

Keywords

MPI, MPMD, Distributed Services, Extreme Scale, Fault-Tolerance

1. INTRODUCTION

HPC distributed services, such as storage systems, are hosted in servers that span several nodes. They interact with clients that connect and disconnect on need. They require network transports that offer high bandwidth and low latency; but unlike application-type HPC programs, distributed services are said to be persistent because they bear no concept of completion. These services are typically written in user space and require user-space networking APIs. Unfortunately, for performance reasons, contemporary HPC systems typically employ custom network hardware and software. In order to reduce porting efforts, distributed services benefit from using a portable network API. The most likely low-level networking API for general-purpose programming is the ubiquitous 30-year-old BSD socket API. While BSD sockets are often supported on HPC networks, they are not typically used because of lower bandwidth and higher latencies when compared with native networking libraries. Instead, the HPC community has seen myriad network technologies, many of which have been short-lived. With proprietary HPC system manufacturers, in particular, there is no guarantee that an existing network API will be adopted by the next generation supercomputer. As example, the LAPI [1], DCMF [7] and PAMI [8] network APIs were all released by a single company for its Scalable POWERParallel [1] and Blue Gene [8] series of supercomputers. Unfortunately, none of those network APIs is portable across all or even most systems.

However, all recent HPC systems do include an implementation of MPI as part of their software stack. Since MPI is one of the primary ways of programming these machines, the bundled MPI implementation is typically well optimized and routinely delivers maximum network performance [10]. Thus, MPI shields the HPC community from the aforementioned volatility in network technologies and from low level details such as flow control and message queue management [12]. MPI has, in effect, become the BSD socket of HPC programming.

MPI offers substantially more than portable network access. Currently though, MPI is not typically used in components of the software stack that extend beyond a single application such as distributed storage services. In this work, we evaluate the use of MPI as a high-performance network portability layer for cross-application services. We describe our successes with MPI as well as the workarounds required to close the gap between our needs and the semantics offered by the MPI standard. With respect to portability, we investigate how well a number of widespread MPI implementations follow the MPI standard; and in cases where unspecified behavior is allowed for an MPI feature required by our design, we enumerate the observed outcomes. We finally propose a number of enhancements to the MPI standard and existing implementations thereof meant to facilitate the expanded use of MPI by distributed services.

The rest of the paper is organized as follows. Section 2 discusses the features that a network transport should offer to ease the design of the distributed storage. Section 3 evaluates our design on top of MPI. Section 4 offers a number of ideas and suggestions intended to ease the adoption of MPI by persistent distributed services. Section 5 discusses the related work. Section 6 summarises our conclusions and discusses future work.

2. TRANSPORT REQUIREMENTS

The network transport is meant for a highly available dis-

EuroMPI '13, September 15 - 18 2013, Madrid, Spain

tributed storage service. The service consists of a number of I/O servers, exporting a unified storage view of the underlying storage to a set of clients (typically compute nodes running application software). Clients typically connect to the storage service at the start of a job, periodically issuing I/O requests, and disconnect when the job ends. In our system, the storage service relies on replication to ensure that data remains available even when an I/O server fails. In addition, data is striped across multiple servers, increasing performance by providing parallel access to the underlying storage devices. The client is oblivious to striping and replication; the I/O library on the-client side only exposes traditional read/write semantics. From a networking perspective, two kinds of data transfers are required: control messages and I/O payload (bulk data). Control messages are conveyed through a remote procedure call (RPC) invocation, in which the client sends a request to an I/O server, which executes the request and then sends a response back to the client. Because of striping, a single, sufficiently large I/O access can span several I/O servers. However, to limit client-side complexity and simplify failure handling, a client always contacts a single I/O server (the primary server) that manages striping and replication on behalf of the client and provides a single response for each I/O request. It does so by relaying the request of the client to any secondary server involved (for purposes of striping or replicas) and aggregating the response of each server before responding back to the client. To avoid unnecessary I/O payload copies and to preserve server-side resources, each I/O server involved in the operation directly accesses the buffer (containing the I/O data) on the client, as opposed to relaying all accesses through the primary server. This I/O servicing process is depicted in Figure 1. In the I/O protocol (Figure 1), when more than a single server is involved, steps *i1* to *o1* in the primary server (server_0) can occur in parallel with steps *i2* to *n2* in the other concerned servers. All the arrows linking two steps are RPC; and their destination steps are blocking until the source steps are executed.

RPC invocations are built on top of a two-sided communication semantic. Servers receive (unexpected) requests from clients, while the client expects a response from the same server it contacted in the first place. Thus, we require that clients support *explicit target (expected) send and receive* and that servers additionally support *unknown source (unexpected) receive*. A tag mechanism is required to differentiate among messages between the same two peers (for example, two concurrent read requests between the same server and client). The tag space must be be big enough to ensure that collisions are unlikely (or the transport needs to provide a mechanism to create non overlapping tag spaces).

Unlike high-level application libraries, a distributed storage service has a lifetime that exceeds that of its clients. In addition, the set of clients is dynamic. Therefore the network transport must be able to *efficiently handle the addition and removal of clients*. Furthermore, clients should be isolated as much as possible. Failure of one set of client nodes should not affect other clients.

Because of the projected client-to-server ratio and the expected number of concurrent operations, it is often not practical to create a thread for each client or operation. Therefore, *all network operations need to support asynchronous operation*, together with the ability to efficiently test for completion of pending operations.

Bulk data access uses *a one-sided communication scheme*, in which the client is always the target. This has a number of advantages. For example, it reduces client-side protocol complexity (since the client is not logically involved in the transfer), enabling easy asynchronous progress on the client and simplifying client or server failure. It also defers flow control to the I/O server, which, given the typical ratio of compute nodes to I/O nodes in current and future HPC systems, ensures I/O server responsiveness even when faced with thousands of clients. Due to their nature, one-sided operations are easier to cancel, as only one party needs to take action to cancel the operation. In addition, by restricting the client to only be the target for one-sided operations, the client does not need to know of any secondary servers, thereby making the protocol more robust and flexible.

Failures, both hard (persistent hardware error) and soft (a temporary failure, such as timeout due to load or dropped message), are part of everyday reality for large-scale HPC systems. Specifically, multi-application services (such as distributed storage services) cannot simply restart when faced with failures. Therefore, these services need to proactively deal with failures. From a networking perspective, this involves being prepared to handle peers that do not follow the expected I/O network protocol. For example, a server could fail to send a response within the expected time limit, because of hardware failure of the server or the communication link between client and server. Even if the network itself ensures reliable communication (for example, in-order delivery and no dropped messages), the higher-level software layers typically cannot tell the difference between a failed peer and an overloaded peer (unable to respond in time). For a distributed storage service, for both client and server, the easiest way to deal with these failures is to put an upper time limit on the completion of the I/O operation. If the operation does not complete in time, either party can consider the operation terminated. For clients, the operation can be retried (possibly to another server). Servers rely on the client to retry the operation if needed. In either case, both parties need to be able to release any resources related to the communication, including the cancellation of outstanding receive and send operations and destruction of one-sided resources such as memory registrations. In addition, to protect against data corruption, both parties need to ensure that any slow or in-flight transfers from the now cancelled I/O request can no longer complete. For instance, a client should not mistake the late response to an already cancelled I/O request for the response to the replacing (reissued) I/O request. Likewise, when retrying an operation, one-sided accesses from earlier I/O requests should no longer have access to the client buffer, even if the buffer address did not change. Therefore, *proper cancellation support* is vital for a modern network transport.

3. MPI AS A NETWORK API

With n storage nodes, the storage server is an n-process MPI job running in `MPI_THREAD_MULTIPLE` mode. The unique storage server job is expected to serve any number of clients simultaneously. The server opens ports with `MPI_Open_port`, then waits for connections in `MPI_Comm_accept`. MPI dynamic process management does not intervene between the server and the clients. Clients, which are separate MPI jobs running on non-storage nodes, are always initially unrelated to the server. Since the storage server can only

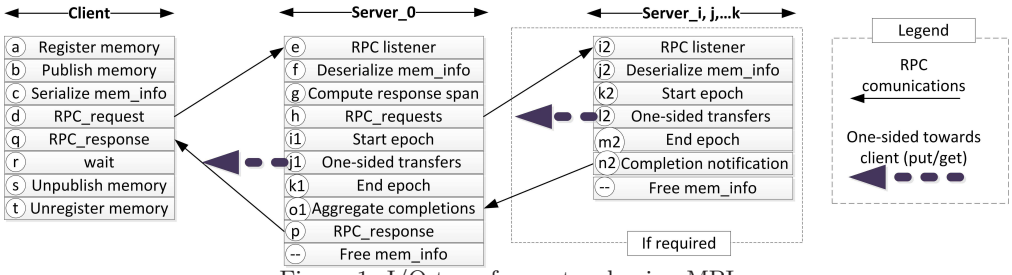

Figure 1: I/O transfer protocol using MPI

interact with MPI processes, even a client made of a single standalone process must be a single-process MPI job. Non-MPI clients can resort to MPI job proxies. In a multi-process client, the job as a whole could create a single client connection; but any subset of its processes could create distinct I/O connections as well. Clients connect on demand when they are ready to issue their first I/O. The connections are then maintained as long as I/O activities are required. The client-side connection protocol executes in sequence `MPI_Comm_connect`, `MPI_Intercomm_merge` and `MPI_Win_create_dynamic` if MPI-3 RMA is available or `MPI_Win_create` otherwise. When all I/O are done, disconnection occurs via `MPI_Comm_disconnect`.

The experiments described in this section are done with OpenMPI-1.6.4, MPICH-3.0.2, and MVAPICH2-1.8.1. Our first test system (C1) is a cluster with 8 GB of RAM per node. The second system (C2) is a cluster with 36 GB of RAM per node. Both systems have QDR InfiniBand and GB Ethernet. MPICH-3.0.2 implements MPI-3.0 while OpenMPI-1.6.4 and MVAPICH2-1.8.1 implement MPI-2.2. The same test codes are used for all three MPI implementations.

3.1 RPC and Two-Sided Communications

RPC, tags, and tag spaces: We implemented RPC operations by directly using the two-sided communications functions offered by MPI. Likewise, support for tags, expected and unexpected (mapped to `MPI_ANY_SOURCE` in combination with `MPI_ANY_TAG`) messages is directly provided by MPI. The MPI standard defines the valid per-communicator tag space to be 0..`MPI_TAG_UB`, where `MPI_TAG_UB` is required to be at least 32,767. Our test indicated values of 2,147,483,647, 1,073,741,823, and 2,147,483,647 for OpenMPI, MPICH and MVAPICH respectively, which make reasonnably big tag spaces.

Cancellation: The storage server fulfills cancellation via `MPI_Cancel`. For an RPC request, cancellation as provided by MPI fulfills our needs. It is always immediate; it thus allows an immediate retry to another server. At the client-side, the tag management guarantees that any response to a canceled RPC will no longer be matched inside MPI before approximately `MPI_TAG_UB` subsequent RPC requests, failed ones included. Unfortunately, cancellation can result in unexpected message queue (UMQ) [12] items that are abandonned forever. On top of the resource consumption issue, this situation creates a false message matching problem in the servers. Message matching in MPI identifies a communicator by its *context_id*. When a communicator is destroyed in a server after the disconnection of a client job CJ_i, its context_id can be recycled into a new communicator created for a subsequent client job CJ_{i+1}. Then, the server mistakenly consumes dead CJ_i requests as if they were sent by the new CJ_{i+1} job. In order to avoid that situation, all the UMQ items associated with dead requests must be cleaned with dummy receives at the destruction of their associated communicator.

3.2 One-Sided Communications

We started this work in MPI-2.2 before the availability of MPI-3.0. We then extended it later to MPI-3.0. The discussion thus concerns both MPI-2.2 and MPI-3.0. To implement one-sided communications, we used passive target (shared lock) MPI-RMA. Unfortunately, a number of compromises had to be made.

Collective window creation: Ideally, we would create the RMA windows on demand for each I/O that is initiated. Unfortunately, the collective nature of RMA window creations prevented that approach. For performance and decoupling reasons, we could not afford to have collective communications in the middle of each I/O. Therefore, all the RMA windows must be created right away when a client job first connects to the storage system.

Constraint of a single epoch per window: RMA-2.2 allows only a single epoch per window. This constraint limits how many concurrent I/Os a single client can have pending. We partially worked around that limitation by having a runtime parameter-controlled maximum number of RMA windows per client. The windows are then managed with a flow-control policy that serializes any new I/O servicing if the maximum number is already pending. With MPI-3.0 RMA, the I/O servers can now manage without serialization an unlimited number of concurrent I/O through a single RMA window per client. When a new I/O starts, it creates the single possible epoch of the window only if there is no other pending I/O towards the same client. An existing epoch is used by any of the subsequent I/Os. When all the RMA communications of an I/O are issued, `MPI_Win_unlock` is called only if that I/O is the last pending one. Any I/O that is not the last pending one simply calls `MPI_Win_flush` to complete its communications. It poses no problem if `MPI_Win_flush` completes one-sided communications not belonging to the specifically ending I/O.

Memory management: With MPI-2.2 RMA, we had to preallocate a reasonnably large memory on client-side at window creation time. All the RMA windows use the same large preallocated memory. The size of that memory is runtime parameter-controlled. By means of custom allocators, the preallocated memory is used as a pool from which buffers are sliced for storage-level memory registration and publishing (steps *a*, *b*, *s*, and *t* of Figure 1). By resorting to MPI-3.0 RMA dynamic windows, we got rid of the fixed preallocated buffer. Memory publishing is now mapped to `MPI_Win_attach`.

Preventing access: In MPI-2.2 RMA, preventing ac-

cesses cannot be made effective. The client-side memory pool is accessible from any server-side epoch. At the cost of extra control message communication, one can enforce access control before a server issues any RMA operation. Already-issued operations, however, cannot be prevented from succesfully reaching the client memory. With MPI-3.0 RMA, the situation is essentially unchanged. The MPI-3.0 specification forbids RMA operations to already-detached target memory. In the absence of more information, we must assume that transgressing the rule could result in fatal outcome in some MPI implementations. What happens while `MPI_Win_detach` is invoked in the middle of an ongoing RMA communication is unknown as well.

Cancellation: In the protocol of Figure 1, storage-level cancellations are expected to happen in steps *k1*, and *m2* which are blocking for pure one-sided reasons. In MPI-2.2 RMA, these steps map to `MPI_Win_unlock`; and in MPI-3.0 RMA, they map to either `MPI_Win_unlock` or `MPI_Win_flush`. Cancellation is therefore not necessarily effective at the communication level. We cannot test or even hypothesize about what could happen if the client were dead. If the client is just unavailable, at the MPI level, the one-sided communications will end up completing. Fortunately, we have no use case that requires a retry from a server to a client; server-side failover to service a response to a client is meaningless. As a result, the only consequence of this cancellation behaviour is server-side resource highjacking while the client is still alive but unresponsive.

3.3 Client-Server and Failure Behaviour

The storage server always runs with `MPI_ERRORS_RETURN` set for both communicators and RMA windows. We considered two kinds of failures: abort and crash. In the abort failure, the faulty process is still alive but behaves inapropriately. It must leave and rejoin the storage system after its problem is fixed. Thus, after the possible cleanup, the process terminates with `MPI_Abort(MPI_COMM_SELF)`. The crash failure, simulated with a provoked segfault, is a brutal death caused by a node shutdown, for instance. This case is just an imperfect approximation of a hardware failure because while a killed process (bus fault, for instance) returns an exit code to the MPI job launcher, some hardware failures might not. The tests in this subsection are run on the cluster C1; with a single process per node.

3.3.1 Server Failures

Ideally, a server or an I/O node that fails should not bring down the whole storage system. We would like such a server to rejoin the storage system when fixed.

Abort failure: Our experiences with aborting are presented in Table 1. Open MPI aborts the job if any subset of `MPI_COMM_WORLD` is aborted. The job survives in MPICH and MVAPICH, and communications between the survivors (`healthy_comm`) proceed without issue. Eager sends to deceased peers complete in both MPICH and MVAPICH. All other communications trap the survivor in the progress engine in MVAPICH. In MPICH we qualify the behaviour as *Undefined* because it varies. In most cases, the communication returns immediately with an error message; this is the ideal case. This behaviour is observed for two-sided, collectives, and even one-sided, except for `MPI_Win_Unlock` which gets blocked forever. In a few cases, large sends and receives of any size get blocked as well.

Table 1: Behaviours in case of isolated server abortion

	Open MPI	MPICH	MVAPICH
No communication	WJA	GCE	TF
Communication over healthy_comm			
Any communication	N/A	Success	Success
Communication over MPI_COMM_WORLD			
2-sided; Eager, survivor is sender	N/A	SSS+GCE	SSS+TF
2-sided; Rendezvous, survivor is sender	N/A	Undefined	TC
2-sided; survivor is receiver	N/A	Undefined	TC
1-sided	N/A	Undefined	TC
Collectives	N/A	Undefined	TC

***WJA**: Whole job abortion; **GCE**: Graceful continuation and exit; **TF**: Trapped in `MPI_Finalize`; **TC**: Trapped in communication; **SSS**: Sender-side success*

Table 2: Storage job behaviours after client job failure

	Open MPI	MPICH
No communication	TCD	GCE
Internal storage job communications		
Any communication	Success+TCD	Success+GCE
Communication over io_comm and connect_comm		
2-sided; Eager, server is sender	SSS+TCD	SSS+GCE
2-sided; Rendezvous, server is sender	TC	TC
2-sided; server is receiver	TC	ER+GCE
1-sided (only over io_comm)	TC	TWU
Collectives	TC	ER+GCE

***GCE**: Graceful continuation and exit; **TCD**: Trapped in `MPI_Comm_disconnect`; **TC**: Trapped in communication; **TWU**: Trapped in `MPI_Win_unlock`; **SSS**: Sender-side success; **ER**: Error return*

Crash failure: In the case of crash failure simulations in a process, all three MPI implementations simply kill the job.

3.3.2 Client Failures

A compute node or process connects (disconnects) to (from) the storage system on the fly by using `MPI_Comm_connect` over `MPI_COMM_SELF` and `MPI_Comm_disconnect`. The servers are started as a single MPI job. One of them opens a port (`MPI_Open_port`) and writes it in a file. For simplicity, no name publishing has been implemented yet. Client processes read the port information in a file.

The experiments in this subsection have been done only with Open MPI and MPICH because we have not been able to successfully use port and connection-disconnection functions in MVAPICH. We run three servers on three nodes and a single client on a fourth node. As a reminder, each client shares an intercommunicator `connect_comm` and an intracommunicator `io_comm` with the storage system. We realized that both crash and abort failures in client jobs produce the same behaviours in the storage job. The result are presented in Table 2. In both Open MPI and MPICH, the storage job survives client job failure (crashes or abortions). No communication with a deceased client brings down the storage job. The communication behaviours are similar for both the intercommunicator and the intracommunicator. In general, in Open MPI, except for Eager sends, the storage processes get trapped in any communication (including `MPI_Finalize`). Except for Rendezvous send and `MPI_Win_unlock`, where it gets trapped, MPICH returns an error message, and the execution completes gracefully. The immediate return and error messages allow the storage job to free resources in a timely fashion and to detect failed clients without custom mechanisms.

3.4 Object Limits

To allow each server process to access the client's buffer directly (Figure 1), each client must share a connection with all the available servers at the time of its first I/O. As a result, for each client connection, a server maintains two explicit MPI communicators. The first one, an intercommunicator, is created from the connection `MPI_Comm_accept` and is later used for unexpected communications. The second one, an intracommunicator, is used for expected communications. Depending on the implementation, a third implicit communicator might be created for the RMA window. Furthermore, in large systems, servers will have to maintain a very large number of handles to support non blocking two-sided operations. The same is true for derived datatypes (DDTs). Non contiguous I/O operations require unique hindexed types for each I/O. In fact, an I/O transfer from a server has to resort to two separate DDTs because the non contiguity layout at the source is different from that of the target. In summary, we estimate that a server needs 3 communicators (including the implicit RMA window one if required), 1 RMA window, 3 DDTs and 2 pending non blocking point-to-point (NBPtP) to service a single instance of any kind of I/O to a client. We verified through emulation tests whether a server can service a million process client job by creating 3,000,000 communicators, 1,000,000 RMA windows, 2,000,000 non blocking posted NBPtP and 3,000,000 hindexed DDTs. Each category of object is created in a separate test. The results are presented in Table 3. To detect resource exhaustion limits, we ran each test on both systems C1 and C2, each time with a single rank per node. In addition to the observed limits, we reported how each MPI implementation behaves after the limit is reached. For the NBPtP tests, the limit is determined by whichever of `MPI_Isend` or `MPI_Irecv` fails first. In Table 3, we omit the cluster name in the result when the implemention behaves similarly on both. In the observations, "Resource exhaustion" might include situations related to pinning or mapped memory, for instance.

All three implementations have a hard limit for the number of communicators. The limit seems to be a design choice because it does not depend on the amount of memory available on the node. For RMA windows, Open MPI succeeds in creating all the required RMA windows on C2 but limits it on C1. It is not clear whether this is a resource exhasution issue, as the failure results in a hanging application. One can notice that Open MPI and MVAPICH can create more RMA windows than communicators. DDTs and NBPtP seem to be limited only by available memory. Open MPI tends to block forever in both by-design and resource-imposed limits. MVAPICH either continues gracefully or crashes after returning an error code. MPICH has successfully created all the DDTs and non blocking communications required to service a million clients. However, in order to observe its behaviour in situations of resource exhaustion, we increased substantially the number of objects. We observed that it behaves similarly to MVAPICH when resources are exhausted.

Table 3: Object limits

	Open MPI	MPICH	MVAPICH
Communicators	65532+UB	2045+ER +GCE	2018+ER +GCE
RMA windows	65532+UB on C1; NL on C2	2045+ER +GCE	2042+ER +GCE
Derived datatypes	NL	NL	RE+ER +GCE
Pending non blocking 2-sided	RE+crash on C1; NL on C2	NL	RE+crash

***NL**: No limit; **UB**: Unlimited blocking; **ER**: Error return; **RE**: Resource exhaustion; **GCE**: Graceful continuation and exit*

4. WISH LIST

In this section, we highlight a number of problem areas and formulate some recommendations, for MPI implementors and the MPI forum. While some of these recommendations follow from our desire to use MPI in a non traditional setting, this does not preclude the usefulness of our requests for more mainstream MPI applications. Furthermore, we believe that as these applications evolve to support the fundamentally different environment presented by future exascale systems, some of the features described in this section will be required for all HPC software domains.

For most contemporary HPC applications, the most reasonable action in case of failures is to restart the job. However, when failed components can be reconstructed (for example from a replica), restarting is not always the best solution because other applications might depend on the same service. The issue of MPI jobs surviving the crash of a subset of `MPI_COMM_WORLD` has been previously studied [5]; recent similar proposals [6, 2] were also put forth during the standardization efforts of MPI-3.0. Unfortunately, none of these proposals made it into the standard. However, even without any change to the curent specification, by honoring `MPI_ERRORS_RETURN`, MPI implementations can already enable various workarounds to keep jobs alive after an isolated crash. While the optimal fault-tolerance strategy at extreme scale is still being debated, we urge the community to *provide some mechanism to continue MPI functionality in the presence of failures*, given the already-large demand for this capability [2].

Cancellation is a vital mechanism for reclaiming resources when faults or other exceptional circumstances arise. Cancellation for two-sided communications has been in MPI since MPI-1.0. It was built around MPI request objects and required a non blocking communication semantic. While MPI-3.0 extended the use of request objects (for example, for non blocking collectives), it is unfortunately still erroneous to isssue `MPI_Cancel` on any request not associated with two-sided communications. Cancellation for network operations is widely known to be challenging. However, we showed in Section 3.1 that cancellation does not have to be perfect to be useful. Unless other mechanisms are put in place for reclaiming resources, for example, by adding timeout functionality to MPI calls to enforce a response in reasonable time in case of peer failure, *efforts should be invested in making most, if not all, MPI routines cancellable.*

Blocking routines can be a hindrance to both performance and scalability. In some cases, concurrent requests are required in order to extract maximum hardware efficiency. At the same time, not all large HPC systems support the creation of an unlimited number of threads; and on systems that do, thread resource consumption typically prohibits creating a large number of threads. MPI, since its inception, has acknowledged this fact by supporting non blocking communication primitives (such as `MPI_Isend`). While MPI-3.0 has added a number of non blocking equivalents (for example non blocking collectives), not all MPI functions have a non blocking equivalent. Especially for failure handling, non blocking routines are critical. For example, in the situations

of indefinitely long blocking (described in Section 3.3) an I/O server could be connected to a very large number of compute clients, every one requiring `MPI_Comm_disconnect`. This MPI call is blocking and does not have a non blocking equivalent. Either the server needs to serialize client disconnections or it handles them with potentially impracticable numbers of threads. The lack of non blocking functionality also hinders the adoption of other programming models, such as event-driven programming. In our prototype, the lack of a non blocking version of `MPI_Comm_accept` forced every I/O server to create a thread dedicated to calling the accept function. *We recommend continuing the effort to extend the set of non blocking functions.*

If MPI is to scale from small systems to exascale systems, MPI implementors should be careful not to introduce extra restrictions, for example, on the number of communicators. Even on existing large-scale systems, using thousands of communicators does not seem unreasonable. *We recommend the removal of artificial scalability limits* where possible, and provisioning applications with non fatal methods to discover these limits when they cannot be removed (for example, because of hardware limitations or performance reasons).

5. RELATED WORK

A study of the possibility of MPI adoption for the storage architecture of PVFS2 and parallel persistent services in general was presented in [9]. The study stated that MPI could be used for a broader range of parallel utilities such as system monitoring deamons. In particular, MPI has been used for file staging and parallel shell design [4]. The I/O delegate proposal [11] allows an MPI job to transit its I/O requests through another MPI job linked to the target filesystem. It resorts to dynamic process management, but it is not a pure client-server design. The compelling aspects of MPI have attracted other data-oriented distibuted services and runtimes as well. MapReduce [10], for instance, has been studied and layered on top of MPI. Moreover, the Partitioned Global Address Space (PGAS) languages have considered MPI for its wide adoption, performance, and richness of programming models [3]. Except for the PVFS2 study [9], none of the cited works were examples of unrelated MPI jobs linked by a client-server relation; and to the best of our knowledge, no implementation of PVFS2 over MPI exists yet.

6. CONCLUSION

We implemented in MPI the network layer of a distributed HPC storage system. While this removed the need to port our network layer to different machine architectures, we found that in certain areas, workarounds or design concessions were needed. However, the challenges encountered were not sufficient to give up on the portability, performance, and the relatively high-level communication functionality offered by MPI. Instead, we derived from our experience a list of suggestions that might widen MPI adoption to include more service-oriented HPC software. In addition, we believe that even within the application community, there is a trend toward a more modular, service-oriented architecture. One example of this is in situ analysis or covisualization. Therefore, many of the recommendations made in this paper are likely to have a broader impact. As future work, we intend to work with both the MPI forum and implementers to ensure that MPI remains a driving force for future software and hardware architectures.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada Grant #RGPIN/238964-2011, Canada Foundation for Innovation and Ontario Innovation Trust Grant #7154.
This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

7. REFERENCES

[1] M. Banikazemi, R. Govindaraju, R. Blackmore, and D. Panda. Implementing Efficient MPI on LAPI for IBM RS/6000 SP Systems: Experiences and Performance Evaluation. In *IPPS'99/SPDP'99*, pages 183–190, 1999.

[2] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. An Evaluation of User-Level Failure Mitigation Support in MPI. EuroMPI'12, pages 193–203, 2012.

[3] D. Bonachea and J. Duell. Problems with Using MPI 1.1 and 2.0 as Compilation Targets for Parallel Language Implementations. *Int. J. High Perform. Comput. Netw.*, 1(1-3):91–99, 2004.

[4] N. Desai, R. Bradshaw, A. Lusk, and E. L. Lusk. MPI Cluster System Software. In *PVM/MPI*, pages 277–286, 2004.

[5] G. E. Fagg and J. J. Dongarra. Building and Using a Fault-Tolerant MPI Implementation. *Int. J. High Perform. Comput. Appl.*, 18(3):353–361, 2004.

[6] J. Hursey, R. L. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, and D. G. Solt. Run-through Stabilization: an MPI Proposal for Process Fault Tolerance. EuroMPI'11, pages 329–332, 2011.

[7] M. Krishnan, J. Nieplocha, M. Blocksome, and B. Smith. Evaluation of Remote Memory Access Communication on the IBM Blue Gene/P Supercomputer. In *ICPP-W '08*, pages 109–115, 2008.

[8] S. Kumar, A. Mamidala, D. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. In *IPDPS'12*, pages 763–773, 2012.

[9] R. Latham, R. Ross, and R. Thakur. Can MPI be Used for Persistent Parallel Services? EuroPVM/MPI'06, pages 275–284, 2006.

[10] X. Lu, B. Wang, L. Zha, and Z. Xu. Can MPI Benefit Hadoop and MapReduce Applications? In *ICPPW'11*, pages 371–379, 2011.

[11] A. Nisar, W.-k. Liao, and A. Choudhary. Scaling Parallel I/O Performance through I/O Delegate and Caching System. SC '08, pages 9:1–9:12, Piscataway, NJ, USA, 2008. IEEE Press.

[12] J. A. Zounmevo and A. Afsahi. An Efficient MPI Message Queue Mechanism for Large-scale Jobs. In *ICPADS*, pages 464–471, 2012.