Special Issue Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2014, Vol. 28(4) 435–449
© The Author(s) 2014
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342014548864
hpc.sagepub.com
SAGE

Extreme-scale computing services over MPI: Experiences, observations and features proposal for next-generation message passing interface

Judicael A Zounmevo[1], Dries Kimpe[2], Robert Ross[2] and Ahmad Afsahi[1]

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 aspects which require various workarounds. Based on use cases, we derive a wish list for both MPI implementations and the MPI forum to facilitate the adoption of MPI by large-scale persistent services. The proposals in the wish list go beyond the sole needs of distributed services; we contend that they will benefit mainstream HPC applications at extreme scales as well.

Keywords
Message passing interface (MPI), distributed services, storage, fault-tolerance, cancellation

1. Introduction

High-performance computing (HPC) distributed services, such as storage systems, are hosted in servers that span several nodes. They interact with clients that connect and disconnect on demand, and require network transports that offer high bandwidth and low latency. These services are typically written in user space and require user-space networking Application Programming Interfaces (APIs). However, 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 Berkeley Software Distribution (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 a 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. For example, the LAPI (Banikazemi et al., 1999), DCMF (Krishnan et al., 2008) and PAMI (Kumar et al., 2012) network APIs were all released by a single company for its Scalable POWERParallel (Banikazemi et al., 1999) and Blue Gene (Krishnan et al., 2008; Kumar et al., 2012) 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 the message passing interface (MPI) as part of their software stack. 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 (Zounmevo and Afsahi, 2014). MPI has, in effect, become the BSD socket of HPC programming. Plus,

[1]ECE Department, Queen's University, Kingston, ON, Canada
[2]Argonne National Laboratory, Argonne, IL, USA

Corresponding author:
Judicael A Zounmevo, ECE Department, Queen's University, Kingston, ON, K7L 3N6, Canada.
Email: judicael.zounmevo@queensu.ca

since MPI is one of the primary ways of programming supercomputers, the bundled MPI implementation is typically well tuned and routinely delivers optimal network performance (Lu et al., 2011).

Currently though, MPI is not typically used in components of the software stack that extend beyond a single application such as distributed services. Plus, unlike application-type HPC programs which are relatively short-lived, distributed services are often persistent because they bear no concept of job completion. Persistence as required in distributed services also implies stricter fault-handling approaches, as the consequences of service abortion usually extend beyond a single job as is usually the case in computing applications.

In this work, we evaluate the use of MPI as a network portability layer for cross-application services. We first describe the challenges, workarounds and aspects of the service design that are made easy, robust or difficult by the semantics of MPI. In particular, we analyze how the service design, which is resiliency-conscious and geared towards scalability with respect to the number of allowed concurrent clients, maps to the features offered by MPI. An emphasis is put on aspects of the MPI specification which serve the needs of the service particularly well. Similarly, we provide various analyses which show aspects of MPI that can be improved to better serve some use cases backed by the service design. These analyses actually make the case for feature additions or improvements in MPI. We argue that these feature proposals, presented as a wish list, are timely and are even relevant to mainstream MPI-based HPC applications on future machines. 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.

This paper is based on an improved version of the service design used in Zounmevo et al. (2013). Additionally, the paper presents a more thorough discussion on the approach used to realize service connection–disconnection over MPI. An emphasis is put on how the absence of non-blocking versions of certain non-communicating MPI routines limits the service to the point where we have to trade scalability for safety. We also add an analysis of communication cancellation scenarios and their resulting consequences.

The rest of the paper is organized as follows. Section 2 presents the distributed storage service which is used as the basis for the MPI-related discussion. Section 3 discusses the features that a network transport should offer to ease the design of the distributed service. Section 4 evaluates our design on top of MPI. Section 5 offers a number of ideas and suggestions intended to ease the adoption of MPI by persistent distributed services. Section 6 discusses the related work. Section 7 summarizes our conclusions and discusses future work.

2. Service overview: The distributed storage

As a concrete example of distributed services, this section presents the highly available distributed storage system used as discussion support in this work. The storage system is made of input/output (I/O) servers that export a unified view of the underlying storage to a set of clients (mostly compute nodes running application software). Clients typically connect to the storage service at the start of a job, periodically issue 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. In the current version of the implementation, an I/O is either exclusively ‘read’ or exclusively ‘write’; the same I/O being ‘read/write’ is not supported.

All the I/O requests start with a remote procedure call (RPC) initiated from a client to a single server that 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 server for any I/O request. That server manages striping and replication on behalf of the client and provides a single response for client-side I/O requests. The contacted server 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. The single server contacted by a client is not fixed; it is decided by a placement policy which falls outside the scope of this work. The placement policy can lead the same client to contact two distinct I/O servers for two distinct I/O requests even if both requests target the same data. For any given I/O request, we designate the contacted I/O server as primary server or server_0; all the other servers involved are referred to as secondary servers.

When the payload of an I/O is small enough, it is simply embedded in the RPC request or response respectively for write and read. For large messages, however, the protocol in Figure 1, meant for bulk data transfers, is used to allow multiple servers to one-sidedly copy chunks of the I/O payload into or from the client’s memory. The lifetime of a large-payload I/O starts with the client registering an I/O buffer in step *a*. A subset of the entire registered buffer is then made available (published) for remote access in step *b*. The distinction between buffer registration and publishing is a performance-conscious choice for network transports where a single registration cost can be amortized

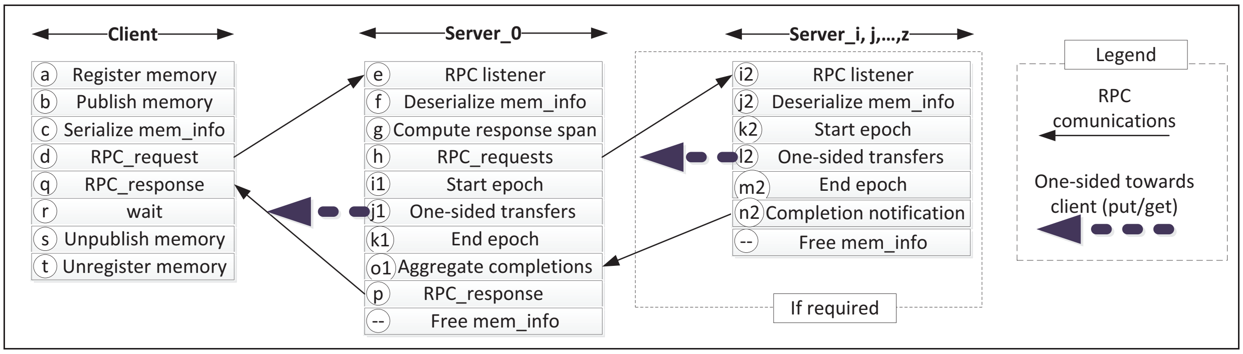

Figure 1. I/O transfer protocol.

over multiple I/O. As a consequence, if the buffer or any of its spanning address superset was already registered, the lifetime of a large-payload I/O could start at step *b* and end at step *s*. Step *t* undoes registration. Ideally, registration does not grant remote access. Remote-access granting starts with publishing in step *b*. Then, in step *c* the remote access information pointed to by the publishing handle is serialized into the RPC packet meant to initiate the I/O. The access information is not tailored for a specific remote server. As a consequence, server_0 can resend it to any other servers involved in the associated I/O, so that they can seamlessly one-sidedly access the client.

The I/O request sent by the client in step *d* is caught by the RPC listener of server_0 in step *e*. The server reproduces the memory access information in step *f*, and in step *g*, determines all the other (secondary) servers that must be seamlessly involved in fulfilling the I/O. Server_0 involves the secondary servers by dispatching the new intermediate RPC requests (step *h*). Then in steps *i1*, *j1* and *k1*, server_0 transfers its chunk of I/O payload to or from the client. In parallel, the secondary servers fulfill steps *i2* to *m2* to transfer their respective partitions of the I/O payload. Server_0 gets blocked in step *o1* until it receives a completion acknowledgement from each of the secondary servers involved. Server_0 sends a final RPC response to the client in step *p*.

For certain network transports, the completion of step *q* at the client side could guarantee that the whole I/O payload has been transferred. However, since the one-sided payload transfer of the secondary servers and the RPC response which contains their completion acknowledgment have different destinations, they might complete out of order; leading to the final reply from server_0 to the client potentially completing before some payload transfers. For network transports where such an out-of-order completion might occur, a custom safe completion subprotocol can be implemented by sending the right control data in the RPC response in steps *p* and *q* and by checking it in step *r*. Step *r* is blocking. After the memory region is unpublished in step *s*, it is expected to become inaccessible to remote servers.

The I/O protocol presented in Figure 1 reduces the client-side protocol complexity since the client is not logically involved in the transfers. As the client is oblivious to all the secondary servers, the protocol simplifies the handling of client and server failure. It also defers flow control to the I/O server and eases the burden of high ratio of compute nodes to I/O nodes in current and future HPC systems. The work on Mercury (Soumagne et al., 2013), which is centered around RPC over the same protocol, provides in-depth comparisons with other existing RPC frameworks and protocols.

3. Transport requirements

The set of network features required by the service are as follows.

Connection–disconnection: The network transport must be able to efficiently handle the random addition and removal of clients.

Autonomous data transfer: All network operations need to support asynchronous operations. Host threads cannot be dedicated to message progression because they are a scarce resource in the server nodes.

Two-sided communications: RPC invocations are built on top of a two-sided communication semantic. A tag mechanism is required to differentiate among messages between the same two peers. The tag space must be big enough to ensure that collisions are unlikely.

One-sided communications: For the client to be effectively oblivious of stripping, replication and the secondary servers in particular (Figure 1), it has to be a passive target in a true one-sided communication scheme.

Scalability: The scalability of a distributed service is determined, among other things, by the scalability of its underlying network. In the particular case of this storage system, the network is expected to allow a large number of simultaneous clients, each potentially having multiple concurrent I/O requests.

Failure mitigation: The failure of an isolated server node should not halt the service. Replication actually makes isolated servers hot-replaceable. Additionally, the service should not be brought down by a failed client, as many other clients are potentially connected to the servers at any given time. A failed client should also not impact other unrelated clients connected to the same servers. For the service to proactively deal with these resiliency concerns, some support is required from the network to detach failed nodes without global impact on the healthy server nodes. Additionally, when a client or server process stops responding, their targeting peers should be able to cancel any pending network operation. For clients, the cancelled operation can be retried to a different healthy or more available server. For servers, resources can be freed and redirected towards other healthy clients.

4. MPI as a network API

When running over MPI, the storage server is an MPI job created in `MPI_THREAD_MULTIPLE` mode. The MPI-based server can only interact with other MPI jobs as clients. As a consequence, even a client made of a single standalone process must be a single-process MPI job. The client-side application binary links with a library that directs its I/O calls to the storage server. The library ensures that client applications that are not built as MPI applications do transparently execute `MPI_Init_thread` during the initialization of the client-side I/O library. Each server node hosts exactly one process of the overall distributed storage job. In order to handle large numbers of simultaneous requests in a scalable manner, each I/O process has:

- A dedicated connection thread that listens on a connection channel and executes the procedure required for first contacts from soon-to-be clients.
- A dedicated listener thread that receives all the requests from already existing clients and transforms them into a work item that is enqueued for a threadpool to handle. This thread fulfils the RPC listener.
- A threadpool made of non-dedicated worker threads that handle the potentially heavy tasks from the work queue. Unless otherwise specified in a configuration file, the size of the threadpool on any node is exactly $n - 2$ if the node can run n hardware threads simultaneously. Oversubscribing is thus avoided.

The two dedicated threads do very little except for receiving connection and RPC requests, the goal being to be as reactive as possible to concurrent requests. We present observations of certain limitations of MPI or MPI implementations in this section. The executions are done with Open MPI-1.6.5, MPICH-3.0.4 and MVAPICH2-1.9. MPICH is the only implementation that can run the fully fledged server, as none of the other two MPI distributions simultaneously provides a working implementation of MPI-3.0 one-sided, a support for `MPI_THREAD_MULTIPLE` and the client–server connection features of MPI. However, we did observe the behavior of Open MPI and MVAPICH for isolated requirements of the design. Our first test system (C1) is a four-node cluster with each node having eight CPU cores and 8 GB of RAM. The second system (C2) is a 310-node cluster with eight CPU cores and 36 GB of RAM per node. Both systems have Mellanox ConnectX QDR InfiniBand and gigabit Ethernet. The following subsections describe how specific aspects of the client–server orchestration are fulfilled in MPI.

4.1 Connection/disconnection

At initialization time, the storage server opens a port with `MPI_Open_port`, and then the connection thread of each individual server process waits in `MPI_Comm_accept`. The service is expected to already be running by the time any client attempts connection; and of course, the service does not spawn its clients. In a multi-process MPI program, the job as a whole could create a single I/O connection; but any subset of its processes could create distinct I/O connections as well. When a set of processes connect as a single client group, they must also disconnect as a whole.

The client-side connection routines operate over a communicator encompassing all the processes in a client group. The routine executes `MPI_Comm_dup`, `MPI_Comm_connect`, `MPI_Intercomm_merge` and `MPI_Win_create_dynamic` in sequence. `MPI_Intercomm_merge` creates an intercommunicator `io_comm` from the intracommunicator returned by `MPI_Comm_connect`. `MPI_Win_create_dynamic` is called over `io_comm`, which is also the communicator that is subsequently used for all RPCs.

At the server side, the connection thread gets blocked inside `MPI_Comm_accept` in a loop which breaks out only when the server enters termination mode. Every time `MPI_Comm_accept` returns, a work item is queued for the worker threadpool. The handler associated with the connection work item does `MPI_Intercomm_merge`, `MPI_Win_create_dynamic` and `MPI_Win_lock_all` in sequence. Then a server-side connection object is created and registered with the request listener.

The disconnection procedures essentially undo what the connection procedures did. However, since they contain steps that we analyze in depth later on, we present their pseudo-codes in support of the upcoming discussion. The client-side disconnection executes the

```
1  client_disconnect (client_connection_t* conn_obj) {
2     clean_zombie_receives (conn_obj);
3     MPI_Barrier (conn_obj–>app_comm_dup);
4     notify_server_for_disconnection (); //This is an RPC
5     MPI_Win_free (&conn_obj–>win);
6     MPI_Comm_free (&conn_obj–>io_comm);
7     MPI_Comm_disconnect (&conn_obj–>connect_comm);
8     MPI_Comm_free(&conn_obj–>app_comm_dup);
9     free(conn_obj);
10 }
```

Listing 1. Client-side disconnection pseudo-code.

```
server_disconnect_work_handler (work_t * work) {
   conn_obj = get_conn_from_work_item (work);
   clean_zombie_receives (conn_obj);
   MPI_Win_unlock_all (conn_obj –> win)
   MPI_Win_free(&conn_obj –> win);
   MPI_Comm_free(&conn_obj –> io_comm);
   MPI_Comm_disconnect(&conn_obj –> connect_comm);
   free(conn_obj);
   free(work);
}
```

Listing 2. Server-side disconnection pseudo-code.

procedure shown in Listing 1; `clean_zombie_receives` (line 2) is described in Section 4.4. A barrier is required (line 3) to ensure that all the processes in the client group have entered their disconnection routine before the server is contacted. Since the server does not actively wait for clients to disconnect, it has to be notified over the RPC listening channel. A process in the client group sends the notification at line 4 of Listing 1. After `client_disconnect` exits, the client job becomes totally detached from the server.

At the server side, after it receives the disconnection request, the RPC listener removes the connection object concerned from the list of endpoints to listen on , and enqueues a threadpool work item with the handler shown in Listing 2. With the exception of `clean_zombie_receives`, `server_disconnect_work_handler` mostly undoes what each server process did to create the connection object in the first place.

4.2 The RPC listener

In each server node, the RPC listener is a dedicated thread that listens to RPC requests coming from either the clients or from the other servers. Since each connection object has its own I/O communicator (`io_comm`), the RPC listener has one `MPI_Irecv` pending per connection object. All the `MPI_Irecv` of the listener are posted with `MPI_ANY_SOURCE` and `MPI_ANY_TAG` so as to allow the server to accept unexpected requests from unknown sources. In a loop, the RPC listener gets blocked in `MPI_Wait_some` until at least a request is received. Then it creates and enqueues a work item.

4.3 I/O life cycles

4.3.1 RPC and MPI two-sided communications. RPCs are the main service-level primitive and are implemented on top of MPI two-sided communications. Each RPC is a round-trip communication made of a request and a response. A request is internally directly mapped to an `MPI_Isend` call while a response is mapped to an `MPI_Irecv` call. For performance reasons, we ensure that the size of an RPC never exceeds the threshold of MPI Eager protocol message sizes. The RPC header bears the address of the sending endpoint; in this case, the rank of the process. A tag mechanism is used to distinguish between multiple simultaneously pending requests from the same client. The tag is a service-level concept but it trivially maps to MPI tags as used in two-sided communications. The service has to guarantee that requests never collide; as a consequence, the tag space for each connection object is expected to be big enough to be considered infinite during the lifetime of a typical client. 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 indicates values of $2^{31} - 1$, $2^{30} - 1$ and $2^{31} - 1$ for Open MPI, MPICH and MVAPICH respectively, which make sufficiently big tag spaces. Communications between servers must use the RPC route as well in order for all control message communications to benefit from the resiliency policies implemented in the service. The resiliency policy is uniformly implemented at RPC level, on top of any transport, such as MPI, that the service is running on.

4.3.2 Bulk data transfer and MPI one-sided communications. The MPI implementation of the memory registration/deregistration of the bulk data transfer protocol (steps *a* and *t* in Figure 1) are noop. Only memory publishing/unpublishing (steps *b* and *s*) is implemented, via `MPI_Win_attach` and `MPI_Win_detach`. At the server side, the one-sided payload transfers resort to passive target epochs and request-based one-sided communications (`MPI_Rput` and `MPI_Rget`). The *Start epoch* step of the protocol does not map to any MPI equivalent as an epoch was already opened over the clients at connection time. As for the *End epoch* step, it maps to testing at MPI level the completion of the requests returned by `MPI_Rput` and `MPI_Rget`. In the protocol, *End epoch* (steps *k1* and *m2* in Figure 1) is supposed to be blocking. In the actual design, the worker threads cannot afford to be blocked, even in portions of the protocol, such as steps *k1*, *o1* and *m2*, where the protocol dictates a wait. In general, we achieve any blocking step of the protocol by reposting its associated work item with the same handler. The handler determines the step, thus, there is no progression in the protocol as long as the handler of the next step is not attached to the work item. This approach allows a few worker threads to handle any number of I/O. As every request-based blocking completion routine of MPI (e.g. `MPI_Wait`) has a non-blocking equivalent (e.g. `MPI_Test`), designing a non-blocking handler for blocking steps was straightforward.

The design of the I/O protocol predates the availability of the MPI-3.0 specification. As a result, we had previous implementations based on MPI-2.2 RMA and on two-sided emulation of the one-sided communications (Soumagne et al., 2013). The performance results of this RPC-based I/O protocol over two-sided emulation of MPI one-sided routines are available for Cray interconnect and InfiniBand in Soumagne et al. (2013).

4.4. Cancellation

No RPC, and consequently no I/O, is allowed to be blocked forever. As a result, all RPCs bear a timeout after which they must expire. As a reminder, a client-side expired RPC is retried, possibly to another server decided by the placement policy. We emphasize that the protocol of Figure 1 is for a single I/O. As a result, when a cancellation occurs, the client must retry the whole protocol. No aspect of the previously cancelled RPC can influence the new retried RPC. A server does not retry expired RPCs; it relies on clients to re-initiate the whole operation which leads to the server issuing the RPC in the first place.

4.4.1 MPI_Cancel and cancellation outcomes. The RPC cancellation implementation is over MPI's own cancellation mechanism (`MPI_Cancel`). According to the MPI specification, the communication is either entirely cancelled or entirely completed. From the sender side, the transmission finishes if it has started before the cancellation becomes effective. At the receiver side, the destination buffer either receives the whole data or is untouched. This behavior is welcome in the service design because it considerably eases the management and analyses of post-cancellation consistency in the service. The only important information a server holds about clients is the connection objects. This choice favors resiliency by making any server easily replaceable. It is not harmful to have temporary I/O objects in inconsistent state inside server processes when an I/O is cancelled; servers discard or recycle those objects without danger upon cancellation at their own level. In summary, the objects whose consistency states matter in situations of cancellation are the payload buffers. In particular, we need to care about the client-side buffer and the storage buffer in each server. Cancellation for any RPC which, like disconnection requests, does not involve any payload is always without consequence for buffer consistency.

When an RPC is cancelled by its initiator, both the request which maps to `MPI_Isend` and the response which maps to `MPI_Irecv` are cancelled. From the other end, cancellation, if initiated by the replier, maps to cancelling `MPI_Isend` only. The RPC-level cancellations can lead to four immediate MPI-level outcomes:

1. **Effective at request transmission:** The cancellation occurs before the request is even sent by MPI. In this case, both the send and the receive are effectively cancelled at MPI level. If the request was meant to initiate an I/O, the I/O is never seen by any server. Any subsequent retry occurs as if the previously cancelled I/O never existed. This cancellation outcome can occur if the request takes longer than required to be sent.
2. **Effective at reply reception:** The cancellation occurs after the request is sent but before the reply reaches the RPC initiator. In this case, only the receive is effectively cancelled. Since the request has been already sent, the reply arrives anyways. With no more receive to consume it, it sits at the receiver side inside the MPI middleware message queue. We name those messages *zombie receives*.
3. **Effective at reply transmission:** This scenario always occurs inside a server. If the server RPC was created as an intermediate one to fulfil an initial client request, then the client RPC ends up timing out as well and gets cancelled.
4. **Ineffective:** The cancellation is issued too late to be effective on any send or receive. The immediately initiating RPC completes successfully. If that RPC is an intermediate one, then the initial client RPC can end up completing as well.

Cancellation can then lead to two kinds of undesirable consequence, namely *zombie receives* and *partially modified buffers*. Persistent partially modified buffers are very infrequent. As long as a successful client-side retry ends up happening for an RPC which was cancelled, the final buffer meant to be written into, or at least an equivalent of that buffer, ends up getting exactly the expected data. For a large-payload read, the client-side buffer could be left partially modified by cancelled RPC between the servers. Transfers between two peers is still subject to the all-or-nothing pattern but all servers might not transfer their partitions of the payload if some of them perform any kind of effective cancellation. In this case however, the client buffer ends up being overwritten by the full payload upon the first successful retry of the initial requester. Then, the buffer remains unchanged even if it is overwritten by portions of the same data by any pending cancelled I/O. For a write, a retry can involve replicas, in which case the same server-side buffer is not modified but its equivalent in a replica server is, leading to the last stored data being the right one.

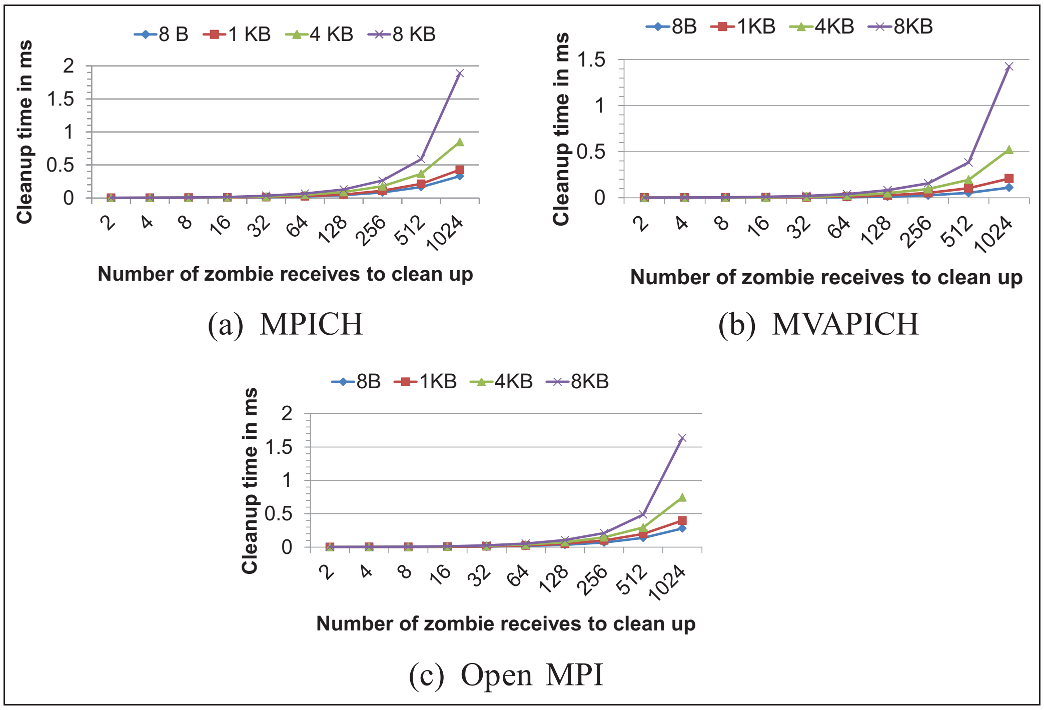

Figure 2. Zombie receive cleanup latency.

4.4.2 Dealing with zombie receives. A zombie receive is essentially an MPI-level unexpected message queue (UMQ) item (Zounmevo and Afsahi, 2014) which is abandoned forever. It could also be sitting inside the network device, waiting to be extracted by MPI. The tag management guarantees for a given client that any response to a cancelled RPC will no longer be matched inside MPI for approximately `MPI_TAG_UB` subsequent RPC requests, failed ones included. On top of the resource consumption issue, this situation can create the following confusion 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 group C_i, its `context_id` can be recycled into a new communicator created for a subsequent client group C_j. Then, a server can mistakenly consume zombie C_i requests as if they were sent by the new C_j client group. In order to avoid that situation, all the zombie receives must be matched and removed from inside the MPI middleware buffers before the destruction of their associated communicator in the disconnection routines.

It is challenging to safely spot zombie receives from legitimate ones as long as the client is still issuing RPCs. However, right after the disconnection request is received by the servers and the reply reaches the client, each end has the guarantee that no more MPI-level two-sided communication will arrive over the connection object about to be disconnected.

In a loop, each zombie receive is discovered via `MPI_Iprobe` with `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, and matched with a dummy receive. The cleanup stops when no more items are found by `MPI_Iprobe`. The cleanup has a de facto local semantic. Since all the cleaned-up receives were messages already sitting locally inside the MPI middleware, no network communication is involved. Figure 2 shows the latency of zombie receive cleanup for RPCs made of a small control message (8 B) to some pretty large buffers containing embedded I/O payload (up to 8 KB). Each test is performed over 100 iterations on the system C2. As shown in Figure 2(a), Figure 2(b) and Figure 2(c), less than 2 ms is required to remove up to 1024 zombie receives even when there is a payload of of 8 KB. In comparison, the average rotational latency of the fastest mechanical hard drive in existence to date (15,000 rpm) (Xiao and Yu-An, 2009) is 2 ms for a single access. Assuming that 1) no peer is abruptly killed by any unforeseen fault and 2) no MPI one-sided communication is involved in servicing the RPCs of interest, these results show that there is an insignificant penalty to managing the only negative consequence of resorting to `MPI_Cancel` for RPC resiliency.

4.4.3 Concluding remarks on MPI adequacy for service-level cancellation. Cancellation as provided by MPI for two-sided communications is adequate to fulfil the needs of the service. It is worth mentioning that `MPI_Cancel` helps fulfil a key scalability trait of the service because it always exits immediately, potentially before the actual cancellation occurs. As a result, clients can, in a

Table 1. Behavior 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`			
Two-sided; Eager, survivor is sender	N/A	SSS + GCE	SSS + TF
Two-sided; Rendezvous, survivor is sender	N/A	Undefined	TC
Two-sided; survivor is receiver	N/A	Undefined	TC
One-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.

timely fashion, issue retries to other potentially more available servers and take advantage of replication whenever possible.

Unfortunately, the situation is different for one-sided communications. Once an I/O request times out and its associated RPC gets cancelled, the protocol (Figure 1) requires the access to the client buffer to be revoked for the cancelled I/O. Service-level buffer access removal maps to `MPI_Win_detach`, which cannot be safely invoked until any one-sided data transfer targeting the attached buffer is completed. Furthermore, one-sided communications cannot be cancelled at MPI level, leading to the impossibility of de-allocating resources in a timely fashion. For any given peer, such a situation can become unmanageable when any of the involved remote peers becomes indefinitely unavailable, due to a fault for instance. In conclusion, MPI provides none of the one-sided communication-related mechanisms required for implementing the resiliency policy of the service.

4.5 Client-server and failure behavior

The storage server always runs with `MPI_ERRORS_RETURN` set for communicators and windows. We consider two kinds of failure: abort and crash. In the abort failure, the faulty process is still alive but behaves inappropriately. 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 segmentation fault, is a brutal death caused by a node shutdown, for instance. The tests in this subsection are run on the cluster C2, with a single process per node. The experiments described in this section test isolated features or their equivalents. For instance, because MPI-3.0 one-sided is not supported by all three MPI implementations, `MPI_Win_lock` is used where the fully fledged server uses `MPI_Win_lock_all`.

4.5.1 Server failures. Ideally, a server or an I/O node that fails should not bring down the service. We would like such a server to rejoin the storage system when fixed. The observations are as follows:

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 behavior as *Undefined* because it varies. In most cases, the communication returns immediately with an error message; this is the ideal case. This behavior is observed for two-sided communications, collectives, and even one-sided communications, except for `MPI_Win_unlock` which gets blocked forever. In a few cases, receives get blocked, and so do large sends.

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

4.5.2 Client failures. 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 realize that both crash and abort failures in client jobs produce the same behaviors in the storage job. The results are presented in Table 2.

In both Open MPI and MPICH, the storage job survives client job failure (crashes or abortions). No communication attempt with a deceased client brings down the storage job. The communication behaviors 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 explicitly or implicitly communicating routine (including `MPI_Finalize`). Except for Rendezvous sends and

Table 2. Storage job behavior 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`		
Two-sided; Eager, server is sender	SSS + TCD	SSS + GCE
Two-sided; Rendezvous, server is sender	TC	TC
Two-sided; server is receiver	TC	ER + GCE
One-sided (only over io_comm)	TC	TC
Collectives	TC	ER + GCE

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

Table 3. Object limits.

	Open MPI	MPICH	MVAPICH
Communicators	65,532 + UB	2045 + ER + GCE	2018 + ER + GCE
One-sided windows	65,532 + UB on C1; NL on C2	2045 + ER + GCE	2042 + ER + GCE
DDTs	NL	NL	RE + ER + GCE
Pending non-blocking two-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.

`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 the failed client without custom mechanisms.

4.5.3 Summary of failure behavior for typical RPCs. The observations in Table 1 and Table 2 lead to the following conclusions. With Open MPI, a failed server terminates the whole storage service. A failed client does not terminate the storage system but could halt its activity because worker threads get trapped in failed communications. As enough worker threads get trapped, the storage system could become completely irresponsive. With MPICH, an aborted server or a failed client do not prevent the storage system from operating as long as all the in-progress and subsequent RPCs are only related to small payload I/O, that is, only Eager two-sided communications are performed. RPCs leading to large payload I/O or disconnection requests could make the storage system irresponsive. No conclusion can be made for MVAPICH because of the impossibility of observing client-server behavior.

4.6 Object limits

Each client group leads to the creation of two explicit communicators in each server process. Depending on the implementation, a third implicit communicator might be created for the one-sided communication 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). Noncontiguous 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 noncontiguity layout at the source is different from that of the target. In summary, we estimate that a server needs three communicators (including the implicit one-sided window one if required), one one-sided window, three DDTs and two pending non-blocking point-to-point communications (NBPtP) to service a single instance of any kind of I/O to a client. We verify through emulation tests whether a server can service a million-process client job by creating 3,000,000 communicators, 1,000,000 one-sided windows, 2,000,000 non-blocking posted NBPtP and 3,000,000 hindexed DDTs. Each category of objects is created in a separate test. The results are presented in Table 3. To detect resource exhaustion limits, we run each test on both systems C1 and C2, each time with a single rank per node, even for the client job.

In addition to the observed limits, we report 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 implementation 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 (Table 3). The limit seems to be a design choice because it does not depend on the amount of memory available on the node. For one-sided windows, Open MPI succeeds in creating all the required objects on C2 but hits a limit on C1. One can notice that Open MPI and MVAPICH can create more one-sided windows than communicators. DDTs and NBPtP seem to be limited only by available memory. Open MPI tends to get blocked 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 behavior in situations of resource exhaustion, we substantially increased the number of objects. We observe that it behaves similarly to MVAPICH when resources are exhausted. Resource-exhaustion-induced limits are not an issue per se; it is how the MPI implementation reacts to them that can make a difference, especially for the required persistence of the service.

The object limit observations lead to the following conclusions. When the storage server runs out of communicators or RMA windows, Open MPI will halt the whole system, without killing it. MPICH and MVAPICH keep the storage system running but subsequent client connections fail. Then when resource exhaustion occurs for DDTs, Open MPI would kill the storage system while MPICH and MVAPICH would keep it running after generating error messages. Finally, when resource exhaustion occurs for non-blocking two-sided communications, the storage system would simply terminate for all three MPI implementations.

5. Observations and 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 recommendations 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.

5.1 Fault-tolerance

5.1.1 Enforcing MPI_ERRORS_RETURN. For most contemporary HPC applications, the reasonable action in the 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 (Graham and Dongarra, 2004); recent similar proposals (Hursey et al., 2011; Bland et al., 2012) 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 current specification, by honoring `MPI_ERRORS_RETURN`, MPI implementations can already enable various workarounds to keep jobs alive after an isolated crash. When those crashes occur, and mechanisms are put in place to detect deceased processes, new healthy subsets of the surviving processes can be built incrementally with `MPI_COMM_SELF` by using approaches similar to the one described in Dinan et al. (2011). 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 (Bland et al., 2012).

5.1.2 A plea for a more cancellation-friendly MPI. From crash-resilient jobs (Graham and Dongarra, 2004; Hursey et al., 2011; Bland et al., 2012) to MPI rank replication on spare hardware (Ferreira et al., 2011), fault-tolerance in MPI has gained more momentum recently. Checkpoint recovery (Hursey et al., 2007), which is probably the most widespread fault-tolerance approach in MPI, is even offered in many MPI implementations. However, as stated in Gropp and Lusk (2004), the optimal fault-tolerance strategy at extreme scales is still not clear. In fact, even if MPI were to become fault-tolerant in the most idealistic way, programs and libraries built with MPI might still have their own additional requirements for resiliency. On top of any integrated fault-tolerance mechanism, the approach of MPI features as resiliency primitives should be encouraged to allow MPI applications, libraries and frameworks to easily realize their own fault-tolerance needs. The second of the three fault-management concepts advocated in Bland et al. (2012) aligns with this suggestion.

Cancellation could be a very useful resiliency primitive. Most large-scale filesystems such as PVFS2 (Latham et al., 2006) or Lustre (Knepper et al., 2012) support request timeout. The same goes for most network technologies that could be used as transport for those systems. For instance, BSD sockets and InfiniBand support communications with timeout. With respect to MPI, the third of the three fault-management concepts advocated in Bland et al. (2012) recommends the absence of indefinite time deadlock or wait in case of failure. While timeout-enabled MPI communications will certainly find substantial adoption, cancellation could be even more useful.

Cancellation is an even more fundamental primitive than timeout, and as such, it gives a bit more flexibility. First, timeout can easily be built on top of cancellation. Second, cancellation does allow a timed communication to be killed before any associated timeout. An example of use case is the situation where two redundant resources are solicited for the same operation, and the first completed operation renders the second one useless. Cancellation is a crucial mechanism for reclaiming resources when faults or other exceptional circumstances arise.

While MPI-3.0 extended the use of request objects (for example for non-blocking collectives), it is unfortunately still erroneous to issue `MPI_Cancel` on any native non-blocking MPI function not associated with two-sided communications. Cancellation for network operations is widely known to be challenging. However, we showed in Section 4.4 that cancellation does not have to be perfect to be useful. MPI does offer means of cancelling custom functions built with generalized requests, but this feature is definitely less powerful compared with natively cancellable functions. In fact, generalized requests cannot transform non-cancellable existing MPI functionalities into cancellable ones. Effort should be invested in making most, if not all, MPI routines cancellable.

5.2 Generalization of non-blocking operations

The current connection–disconnection handling at the server side is non-scalable (Listing 1, Listing 2). The issue is the blocking nature of the non-communicating MPI collective routines involved. Currently, connection and disconnection must not just be strictly serialized; they must occur in the exact same order on each and every server node to guarantee safety in all scenarios. That order, in the case of connections, is imposed by the root process specified in `MPI_Comm_accept`. In the case of disconnections, we impose it by having the root of any client group contact the exact same server every time (line 4 of Listing 1). Imposing the exact same server as the root to contact for those procedures hurts our resiliency, load balancing and scalability efforts which allows a client to retry any RPC with an alternative server node when one is not being responsive. Currently, the root server is a single point of failure in the service because, unlike the other servers, it does not have any strictly equivalent replica.

To understand why we have to resort to serialization and ordering of the connections and disconnections, let us assume the absence of those constraints and analyze how the service as a whole behaves as a consequence. We define:

- n as the number of distinct clients simultaneously asking for disconnection;
- w_i as the maximum number of disconnection work items that could be simultaneously handled by the server i;
- t_i as the number of CPU cores on the server i; we avoid oversubscribing the servers, so the upcoming analysis assumes that each server can run more than two hardware threads simultaneously;
- S as the set of servers;
- D_i as the set of connection objects being simultaneously disconnected by the server node i.

With the blocking routines offered by the current MPI specification for the disconnection procedures, since two threads are dedicated to connection and RPC listening, we obviously have

$$\forall i \in S, \quad w_i = t_i - 2 \tag{1}$$

The situation modeled by equation (2) results in the complete deadlock of the service as a whole, along with all the clients which had already started their disconnection procedure. $|D_i \cap D_j| = 0$ simply means that servers i and j are waiting for each other in collective calls over sets of mutually exclusive service-level disconnections. In concrete terms, they all get blocked in the respective first collective call encountered in the disconnection procedure (line 5 of Listing 2). Then $(|D_i| = w_i) \wedge (|D_j| = w_j)$ means that all the threads of each of the servers are already occupied by the blocking server-level disconnection work items. There is no worker left for any of the two servers to, by chance, pick a work item that could transform $|D_i \cap D_j| = 0$ into $|D_i \cap D_j| > 0$:

$$\begin{aligned} \exists\{i,j\} \subset S \text{ such that } (|D_i| = w_i) \wedge (|D_j| = w_j) \\ \wedge\, |D_i \cap D_j| = 0 \end{aligned} \tag{2}$$

The aforementioned deadlock is guaranteed to be avoided only if the number of simultaneous disconnection requests is strictly less than the sum of the number of work items that any two servers can handle in parallel (equation (3)):

$$\begin{aligned} \forall\{i,j\} \subset S \text{ such that } (|D_i| = w_i) \wedge (|D_j| = w_j), \\ \boldsymbol{n} < \boldsymbol{w_i} + \boldsymbol{w_j} \Rightarrow |D_i \cap D_j| \geq 1 \end{aligned} \tag{3}$$

$$\forall\{\boldsymbol{i},\boldsymbol{j}\} \subset \boldsymbol{S}, \boldsymbol{n} < \boldsymbol{w_i} + \boldsymbol{w_j} \tag{4}$$

Equation (3) states that a standstill will always *eventually* be broken as long as we guarantee $\boldsymbol{n} < \boldsymbol{w_i} + \boldsymbol{w_j}$. Intuitively, equation (3) means that as long as we guarantee $\boldsymbol{n} < \boldsymbol{w_i} + \boldsymbol{w_j}$, if $|D_i| = w_i$, any existing standstill, due to D_i and D_j being disjoint, will be broken at the latest by the time $|D_j| = w_j$, and vice versa. In summary, equation (4) is the absolute safety condition. As per equation (4), no more than 11 simultaneous clients are allowed on our eight-CPU-core-per-node systems if deadlock is to be deterministically avoided. This

Listing 3. Server-side non-blocking disconnection pseudo-code.

```
 1  server_idisconnect_work_handler (work_t* work) {
 2    MPI_Request reqs [3];
 3    conn_obj = get_conn_from_work_item (work);
 4    clean_zombie_receives (conn_obj);//This has a local semantic
 5    MPI_Win_unlock_all (conn_obj->win); //Will return immediately
 6    MPI_Win_ifree (&conn_obj->win, &req [0]);
 7    MPI_Comm_ifree (&conn_obj->io_comm, &req [1]);
 8    MPI_Comm_idisconnect (&conn_obj->connect_comm, &req [2]);
 9    work->handler = server_idisconnect_completion_work_handler;
10    update_work_args_with_reqs (work, reqs);
11    enqueue_work (work);
12  }
13  server_idisconnect_completion_work_handler (work_t* work) {
14    MPI_Request reqs [3]; int flag; MPI_Status statuses [3];
15    fill_reqs_from_work_item(work, reqs);
16    MPI_Testall (3, reqs, &flag, statuses);
17    if (flag) {
18      conn_obj = get_conn_from_work_item (work);
19      free (conn_obj);
20      free (work);
21    }
22    else
23      enqueue_work (work);
24  }
```

number of clients is obviously far too small. Unfortunately, CPU core count per node in nowadays' systems is not very high, and the future might not promise substantially higher core counts because of hardware-level contention. Consequently, enforcing equation (4) is not a viable option for an extreme-scale service. In fact, it is perfectly possible for a single multi-million process MPI job to open thousands of I/O connections which will all be requested for termination right before `MPI_Finalize`, leading to a massive amount of simultaneous disconnection requests sent to the service.

One can notice that no amount of CPU cores per server node can solve the problem created by equation (2). In fact, as per equation (1), equation (4) leads to equation (5). The opposite of equation (5) is expressed by equation (6) which is impossible to overcome. In fact, assuming that equation (6) is already true, every time t_i or t_j is increased by 1, one just needs to increase n by 1 to maintain the deadlock risk:

$$\forall\{i,j\} \subset S, n < t_i + t_j - 4 \quad (5)$$

$$\exists\{i,j\} \subset S, \text{such that } n \geq t_i + t_j - 4 \quad (6)$$

The single-point-of-entry approach automatically guarantees, via the unique contact server, that the simultaneous number of disconnections does not go beyond the safe limit. If the deadlock situation must be deterministically avoided while not resorting to a single point of entry, the servers must coordinate among themselves to guarantee that the sum of all the disconnections that are currently activated does not go beyond the safe limit. This coordination requires a storage-wide integer token that each server must acquire and modify atomically before propagating a disconnection request to the other servers. The global token must also be atomically decremented when a disconnection request is done being fulfilled. Servers do not necessarily know if the token has already reached the maximum that guarantees safety. As a result, they must still perform the acquisition sometime just to realize that the update is not possible. Finally, the comparison to test if the safe limit of simultaneous disconnections is reached is an inequality test; as a result, a regular `compare_and_swap` operation cannot be used to realize the acquire–modify operation in a single MPI call. The server trying to modify the token is left with the slow sequence `MPI_Win_lock`, `MPI_Get`, `MPI_Win_flush`, `MPI_Put`, `MPI_Win_unlock`, or some equivalent. This is to show that the safe alternative to the single point of entry does not fix the serialization issue inherent to the single point of entry, but more importantly, that alternative is more complex and more expensive.

Assuming the existence of non-blocking versions of some of the functions involved in the disconnection procedures, consider how the hypothetical approach of Listing 3 voids all the aforementioned problems. The disconnection work could be executed by resorting to the non-blocking collective cleanup functions on lines 6, 7, 8 of Listing 3. Then, on lines 9 to 11, a completion work is enqueued with the non-blocking handler at lines 13 to 24 of Listing 3. We emphasize that by the time the disconnection is initiated, the clients associated with the connection object must have already completed all their I/O, meaning that no one-sided communication is still in flight. As a result, MPI_Win_unlock_all will return immediately at line 5. Further, clean_zombie_receives at line 4 will also return immediately because it has a local semantic. With the possibility of handling the disconnection in a non-blocking way, even a single thread can progress any number of disconnections to completion. In fact, equation (1) is now replaced by equation (7), leading to equation (4) always being true for any finite number of clients:

$$\forall i \in S, w_i = \infty \tag{7}$$

In general, 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. Having a thousand blocking routines pending requires no less than a thousand threads. In comparison, a threadpool of just a very few threads can do the same job if non-blocking routines are available. In fact, it is useful to re-emphasize that, as shown by the previous disconnection deadlock hazard analysis, a higher degree of parallelism is not an alternative to the provision of non-blocking routines and vice versa. Both concepts do offer overlapping benefits but are not interchangeable.

Furthermore, any blocking operation can be made non-blocking as long as consistency constraints are respected. Plus, by keeping blocking versions of potentially problematic non-blocking operations, users always have a safe alternative API handy to accomplish the same task in situations where consistency can be difficult to reason about. Similarly, a user can elect to use the blocking version of a functionality if a fast reaction is required.

Non-blocking operations seem to be a necessary condition for efficient cancellation as well. If cancellation was possible at all with blocking operations, two threads would be required for it to be effective on any single routine: one for the blocking call, and a second one to issue the cancellation. The MPI_Test family of routines coupled with non-blocking versions of most routines would open up multiple possibilities for algorithms, programming paradigms (e.g. event-driven coding), performance and scalability decisions in code running on top of MPI. We recommend continuing the effort to extend the set of non-blocking functions in MPI.

5.3 Object limits and resource exhaustion

HPC compute jobs rarely need thousands of communicators or RMA windows. The situation could be very different for persistent distributed services. Similarly, DDTs and non-blocking operations might exist in large quantities in large compute jobs but their numbers are usually reasonable in each process of the job. As shown by our design, any single process in persistent services can manage very large quantities of these objects at large scales. As a consequence, by-design limits should be seriously lifted up to allow a broader set of use cases and to get MPI implementations ready for extreme-scale uses. Even at extreme scales, most programs will be naturally bound in many respects by architecture limits, such as word or pointer sizes in C. Thus, more natural by-design limits could be INT_MAX, UINT_MAX or LONG_MAX, ULONG_MAX for instance.

Furthermore, resource exhaustion is one of the most trivial expectations in very large programs. As a result, it is planned for and is therefore rarely an unmanageable issue. In fact, our storage service has a robust flow control policy that makes clients wait and retry in situations of resource exhaustion. Such a policy which delays selected clients without disrupting the service for everybody is much better than a sudden crash which renders the service totally unavailable. Ideally, MPI should provide upper layers with non-fatal methods of discovering resource exhaustion, so that these upper layers can decide and trigger workarounds if possible.

6. Related work

The widespread adoption of MPI has attracted upper-level programming models such as PGAS (Bonachea and Duell, 2004) and MapReduce (Lu et al., 2011). The study in Latham et al. (2006) stated that MPI could be used for monitoring daemons, a class of persistent services. MPI has also been used for file-staging and parallel shell design (Desai et al., 2004). The I/O delegate proposal (Nisar et al., 2008) allows an MPI job to transit its I/O requests through another MPI job linked to the target filesystem. The work resorts to dynamic process management, but it is not a pure client-server design, as presented in our experiment.

From a purely storage point of view, the most widespread use of MPI is via its MPI-IO parallel file manipulation features (Ching et al., 2003). MPI-IO has been used over GPFS (Blas et al., 2010), Lustre (Logan and Dickens, 2008), NFS (Calderón et al., 2002), PVFS2 (Huaiming et al., 2011) and OrangeFS (Tanimura et al., 2013). MPI-IO is used at filesystem level; that is, on top of an existing storage system. In comparison, our experimentation uses MPI below the storage system.

7. Conclusion and future work

Portability and high performance are two attractive traits of MPI. As MPI runs on top of the network fabric of its hosts, the aforementioned traits allow programmers to uniformly and efficiently target a very disparate set of supercomputers, each with its own architecture and network API. With the same concerns of portability and performance, we implemented in MPI the network layer of a resilient, scalability-conscious distributed HPC storage system.

Many MPI-3.0 features such as dynamic windows and request-based RMA communications greatly facilitate the design. However, in certain areas, work-arounds and design concessions were needed. The service exhibits use cases where non-blocking versions of non-communicating MPI routines would make a sound difference for performance, scalability and even safety. Another scalability issue created by the adoption of MPI resides in the runtime limit imposed on certain objects such as communicators. Furthermore, unlike regular MPI jobs, services are persistent and less tolerant to unplanned termination. As a consequence, the fault-handling of MPI was also a major issue about which we provided analyses and recommendations. Finally, we provided an analysis of the suitability of `MPI_Cancel` for the service design and discussed how it can be generalized to allow custom application or service-level resiliency design.

Our list of MPI-directed suggestions is primarily meant to widen MPI adoption to include more service-oriented HPC software. Nevertheless, we believe that even within the HPC 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 collaborate with both the MPI forum and implementers to ensure that MPI remains a driving force for future software and hardware architectures. We are also working on non-blocking designs of non-communicating MPI routines such as the connection/disconnection functions.

Acknowledgements

We thank the anonymous reviewers for their insightful comments.

Funding

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

References

Banikazemi M, Govindaraju R, Blackmore R, et al. (1999) Implementing efficient MPI on LAPI for IBM RS/6000 SP systems: Experiences and performance evaluation. In: *Proceedings of the 13th international and 10th symposium on parallel and distributed processing (IPPS/SPDP)*, pp. 183–190.

Bland W, Bouteiller A, Herault T, et al. (2012) An evaluation of user-level failure mitigation support in MPI. In: Träff JL, Benkner S and Dongarra JJ (eds) *Recent Advances in the Message Passing Interface (Lecture Notes in Computer Science*, vol. 7490). Berlin/Heidelberg: Springer.

Blas JG, Isaila F, Carretero J, et al. (2010) Implementation and evaluation of file write-back and prefetching for MPI-IO over GPFS. *International Journal of High Performance Computing Applications (IJHPCA)* 24(1): 78–92.

Bonachea D and Duell J (2004) Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking (IJHPCN)*1(1–3): 91–99.

Calderón A, García F, Carretero J, et al. (2002) An implementation of MPI-IO on expand: A parallel file system based on NFS servers. In: Kranzlmüller D, Volkert J, Kacsuk P, et al. (eds) *Recent Advances in Parallel Virtual Machine and Message Passing Interface (Lecture Notes in Computer Science*, vol. 2474). Berlin/Heidelberg: Springer, pp. 306–313.

Ching A, Choudhary A, Coloma K, et al. (2003) Noncontiguous I/O accesses through MPI-IO. In: *Proceedings of the 2003 IEEE/ACM international symposium on cluster computing and the grid (CCGrid)*, pp. 104–111.

Desai N, Bradshaw R, Lusk A, et al. (2004) MPI cluster system software. In: Kranzlmüller D, Kacsuk P and Dongarra J (eds) *Recent Advances in Parallel Virtual Machine and Message Passing Interface (Lecture Notes in Computer Science*, vol. 3241). Berlin/Heidelberg: Springer, pp. 277–286.

Dinan J, Krishnamoorthy S, Balaji P, et al. (2011) Noncollective communicator creation in MPI. In: Cotronis Y, Danalis A, Nikolopoulos D, et al. (eds) *Recent Advances in the Message Passing Interface (Lecture Notes in Computer Science*, vol. 6960). Berlin/Heidelberg: Springer, pp. 282–291.

Ferreira K, Stearley J, Laros JH, et al. (2011) Evaluating the viability of process replication reliability for exascale systems. In: *Proceedings of the 2011 international conference for high performance computing, networking, storage and analysis (SC)*, pp. 1–12.

Graham FE and Dongarra JJ (2004) Building and using a fault-tolerant MPI implementation. *International Journal of High Performance Computing Applications (IJHPCA)*18(3): 353–361.

Gropp W and Lusk E (2004) Fault tolerance in message passing interface programs. *International Journal of High Performance Computing Applications (IJHPCA)*18(3): 363–372.

Huaiming S, Yanlong Y, Xian-He S, et al. (2011) Server-side I/O coordination for parallel file systems. In: *Proceedings of the 2011 international conference for high performance computing, networking, storage and analysis (SC)*, pp. 1–11.

Hursey J, Graham RL, Bronevetsky G, et al. (2011) Run-through stabilization: An MPI proposal for process fault tolerance. In: Cotronis Y, Danalis A, Nikolopoulos D, et al. (eds) *Recent Advances in the Message Passing Interface (Lecture Notes in Computer Science*, vol. 6960). Berlin/-Heidelberg: Springer, pp. 329–332.

Hursey J, Squyres J, Mattox T, et al. (2007) The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In: *Proceedings of the 2007 IEEE international parallel and distributed processing symposium (IPDPS)*, pp. 1–8.

Knepper R, Michael S, Johnson W, et al. (2012) The Lustre file system and 100 gigabit wide area networking: An example case from SC11. In: *Proceedings of the 2012 IEEE international conference on networking, architecture and storage (NAS)*, pp. 260–267.

Krishnan M, Nieplocha J, Blocksome M, et al. (2008) Evaluation of remote memory access communication on the IBM Blue Gene/P supercomputer. In: Mohr B, Träff JL, Worringen J, et al. (eds) *Proceedings of the 2008 international conference on parallel processing (ICPP)*, pp. 109–115.

Kumar S, Mamidala A, Faraj D, et al. (2012) PAMI: A parallel active message interface for the Blue Gene/Q supercomputer. In: *Proceedings of the 2012 IEEE international parallel and distributed processing symposium (IPDPS)*, pp. 763–773.

Latham R, Ross R and Thakur R (2006) Can MPI be used for persistent parallel services? In: Mohr B, Träff JL, Worringen J, et al. (eds) *Recent Advances in Parallel Virtual Machine and Message Passing Interface (Lecture Notes in Computer Science*, vol. 4192). Berlin/Heidelberg: Springer, pp. 275–284.

Logan J and Dickens P (2008) Towards an understanding of the performance of MPI-IO in Lustre file systems. In: *Proceedings of the 2008 IEEE international conference on cluster computing (Cluster)*, pp. 330–335.

Lu X, Wang B, Zha L, et al. (2011) Can MPI benefit Hadoop and MapReduce applications? In: *Proceedings of the 2011 international conference on parallel processing workshops (ICPPW)*, pp. 371–379.

Nisar A, Liao W and Choudhary A (2008) Scaling parallel I/O performance through I/O delegate and caching system. In: *Proceedings of the 2008 international conference for high performance computing, networking, storage and analysis (SC)*.

Soumagne J, Kimpe D, Zounmevo JA, et al. (2013) Mercury: Enabling remote procedure call for high-performance computing. In: *Proceedings of the 2013 IEEE international conference on cluster computing (Cluster)*.

Tanimura Y, Filgueira R, Kojima I, et al. (2013) MPI collective I/O based on advanced reservations to obtain performance guarantees from shared storage systems. In: *Proceedings of the 2013 IEEE international conference on cluster computing (Cluster)*, pp. 1–5.

Xiao L and Yu-An T (2009) TPL: A data layout method for reducing rotational latency of modern hard disk drive. In: *Proceedings of the 2009 WRI world congress on computer science and information engineering (WCECS)*, pp. 336–340.

Zounmevo JA and Afsahi A (2014) A fast and resource-conscious MPI message queue mechanism for large-scale jobs. *Journal of Future Generation Computer Systems*30: 265–290.

Zounmevo JA, Kimpe D, Ross R, et al. (2013) Using MPI in high-performance computing services. In: *Proceedings of the 2013 European MPI users' group meeting (EuroMPI)*, pp. 43–48.

Author biographies

Judicael A Zounmevo is currently a postdoctoral appointee in the Mathematics and Computer Science Department of Argonne National Laboratory, IL. He received his PhD at Queen's University in Kingston, ON, Canada, in 2014 under the supervision of Ahmad Afsahi. He is interested in HPC communication runtimes such as MPI and in HPC-targeted operating systems.

Dries Kimpe is an assistant computer scientist at Argonne National Laboratory, IL. He obtained his Master's from the University of Ghent and his PhD from the Catholic University of Leuven. His main research topics are parallel I/O, distributed file systems and parallel programming models.

Robert Ross is a pioneer in the design of parallel file systems and high-performance interfaces for managing large datasets. He led the development of the parallel virtual file system (PVFS) used in many academic, industry and laboratory settings, including the nation's leadership-class computing facilities. He leads storage research in the DOE SciDAC Enabling Technology Center for Scientific Data Management and is Associate Director of the SciDAC Institute for Ultra-Scale Visualization, where he is developing tools to help researchers address challenges in storage, retrieval and the extraction of meaning from very large scientific datasets. Ross is a Fellow of the University of Chicago/Argonne Computation Institute.

Ahmad Afsahi is currently an Associate Professor in the Department of Electrical and Computer Engineering at Queen's University in Kingston, ON, Canada. He received his PhD degree in Electrical Engineering from the University of Victoria, BC, in 2000. His research activities are in the areas of parallel and distributed processing, network-based high-performance computing, high-speed networks and power-aware high-performance computing. He has mainly been focused on improving the performance and scalability of communication subsystems, messaging layers and runtime systems for high-performance computing and data centers.