Can MPI Be Used for Persistent Parallel Services?

Robert Latham, Robert Ross, and Rajeev Thakur

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{robl,rross,thakur}@mcs.anl.gov

Abstract. MPI is routinely used for writing parallel applications, but it is not commonly used for writing long-running parallel services, such as parallel file systems or job schedulers. Nonetheless, MPI does have many features that are potentially useful for writing such software. Using the PVFS2 parallel file system as a motivating example, we studied the needs of software that provide persistent parallel services and evaluated whether MPI is a good match for those needs. We also ran experiments to determine the gaps between what the MPI Standard enables and what MPI implementations currently support. The results of our study indicate that MPI can enable persistent parallel systems to be developed with less effort and can provide high performance, but MPI implementations will need to provide better support for certain features. We also describe an area where additions to the MPI Standard would be useful.

1 Introduction

Achieving good performance on today's high-end computers involves effectively utilizing a variety of network interconnects, a large number of compute resources, and high-quality algorithms. Application developers make heavy use of libraries and tools to manage this complexity while still delivering high performance. For their work, Parallel application writers commonly choose the message-passing model, embodied by the MPI Standard [10]. MPI defines a rich API that can be used across many disparate hardware platforms and provides many useful features such as datatype packing, collective communication, nonblocking communication, and dynamic process management. High-quality MPI implementations further provide heterogeneous communication and deliver high performance.

Parallel system services, as opposed to applications, are usually not written in MPI. One would imagine, however, that MPI's portability, performance, and features should make it an attractive candidate for implementing parallel system services as well. Why, then, don't services use MPI? Could they? We investigate these issues in detail in this paper. For concreteness, we use the parallel file system PVFS2 [12] as an example for studying the needs of such software. We have been heavily involved in the development of PVFS2 and are familiar with its requirements. PVFS2 and its predecessor, PVFS [2], represent a decade of

parallel file system research and engineering. PVFS2 was written to deliver high performance at scales of hundreds of servers and tens of thousands of clients and has done so on some of the world's fastest and largest classes of supercomputers, such as IBM BG/L, Cray XT-3, and large Linux clusters.

We first give a brief overview of PVFS2 and its architecture. Then, using PVFS2 as an example, we study the needs of software for persistent parallel services and examine how well MPI is equipped to meet those needs. We find in most cases that the MPI Standard supports the features we need. Some helpful features, however, are not available in some commonly deployed MPI implementations. We also describe an area that would benefit from additions to the MPI Standard.

2 PVFS2: A Persistent Parallel Service

A *persistent parallel service* is system software that manages multiple hardware components to provide a single logical resource for use by parallel applications. It is persistent in the sense that it exists beyond the life of a single application, typically running for weeks or months at a time. A parallel file system is an example of a persistent parallel service.

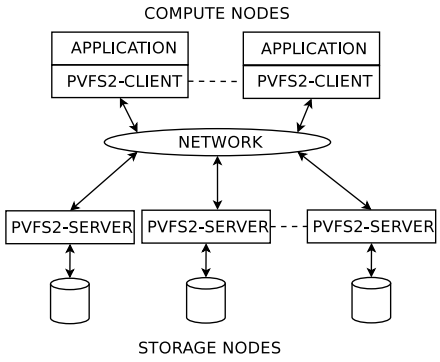


Fig. 1. *PVFS2 architecture:* `pvfs2-client` *forwards kernel-level requests to* `pvfs2-server` *processes running on the servers. In turn,* `pvfs2-server` *deals with managing data on storage devices.*

PVFS2 [12] is a high-performance parallel file system being developed as a joint project by Argonne National Laboratory, Clemson University, and the Ohio Supercomputer Center. PVFS2 comprises multiple persistent servers. File striping across these servers enables multiple clients to access different parts of a file in parallel, resulting in high performance. PVFS2 software on the client side hides all these details from the client and instead presents a single logical view of a file.

PVFS2 provides many features such as native support for popular networking technologies (e.g. Myrinet, InfiniBand, and TCP/IP), multiple APIs (POSIX, MPI-IO), user-controlled striping of files across nodes, a well-defined interface for describing new data distribution schemes, support for heterogeneous clusters, and distributed metadata. It uses commodity network and storage hardware and is easy to install (no kernel patch). The familiar UNIX file tools (such as `ls`, `cp`, and `rm`) can be used on PVFS2 files and directories.

In the following sections, we use PVFS2 as an example to study the common needs of persistent parallel services and then investigate how well MPI supports those features.

3 Service Identification

Any persistent service needs to handle the important issue of locating the servers. For traditional network services, the IP address and port number are often listed in a configuration file. PVFS2 follows a similar approach. The configuration files for PVFS2 servers list all the servers that form the parallel file system. Each server reads this list at startup. A PVFS2 client uses its own configuration file to locate PVFS2 servers (see Figure 2). This file resembles a Unix `/etc/fstab` file and provides the network address of any one of the PVFS2 servers, a mount point on the client system, and a few other parameters. The client inquires with the listed server about the file system, obtains a complete listing of all the servers, and then begins interacting with the file system.

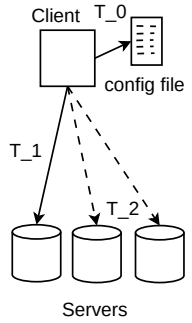


Fig. 2. *Client establishing connections to PVFS2 servers. The client has to consult a configuration file and connect to one of the PVFS2 servers before discovering where the other servers are.*

If PVFS2 used MPI, it could use MPI's features that enable service identification. The MPI name publishing interface (MPI_PUBLISH_NAME, MPI_LOOKUP_NAME) provides a method for clients and servers to exchange information. Clients could use a well-known key to discover an initial contact point. This key would provide service discovery that is independent of the underlying network interconnect or the MPI implementation. Clients would be insulated from server changes, be it a different port, host, or even interconnect, without system administrators needing to update client-side configuration files. MPI might still need some sort of configuration information, but at least we would be able to concentrate that information into a single source, instead of one source for MPI and another for PVFS2.

In practice, however, MPI implementations currently do not support this functionality as well as needed. For this functionality to be usable, MPI implementations must support name publishing and resolution across independently started MPI processes – PVFS2 servers are not restarted with every new client application. We ran tests with several commonly deployed MPI implementations and found that they support this mode of operation, but only under certain conditions (summarized in Table 1). For example, the processes must be part of the same MPD ring in MPICH2 [11], and Open MPI [7] programs require special measures when launching the `orted` daemons. This additional component (MPD or orted) must also be persistent and able to tolerate node failure.

Table 1. *Capabilities of MPI implementations. An ideal implementation would have a Y in all columns.*

Feature	MPICH2 1.0.3	Open MPI 1.0.1	BGL-MPI V1R2M1
Published name appears to other singleton processes	N	N	N
Connect/Accept work under singleton MPI_INIT	N	N	N
MPI_COMM_JOIN works under singleton MPI_INIT	Y	N	N
Does not require a previously established MPI environment (e.g. lamboot, MPD, others)	N	N	N
MPI datatype processing supports heterogeneous architectures	N	N	N
Support for `external32`	N	N	N

4 Establishing Connection

After clients have discovered what services are running, they need to connect to those services. The traditional Unix socket model has the familiar TCP `accept/connect` handshake. Other protocols have analogous mechanisms. PVFS2 uses an abstraction that is layered on top of the connection mechanisms of multiple networks, providing portability.

The use of MPI could simplify this process greatly. MPI's dynamic process functionality supports two different ways for clients to establish communication with servers. One approach has the server process call MPI_COMM_ACCEPT, waiting for a corresponding client-side call to MPI_COMM_CONNECT. MPI_COMM_JOIN provides another approach for two processes that already share a UNIX network socket to establish MPI communication. In both cases, the functions returns an MPI intercommunicator, over which the clients and servers can communicate. Furthermore, the accept/connect functions in MPI are *collective*. A group of clients can connect to a group of servers at the same time, and the resulting intercommunicator can be used for communication between any client and any server.

These MPI functions provide a simpler interface than do the corresponding Unix socket ones, abstracting away details such as allocating a socket and setting protocol-specific values in data structures. In addition, they are portable: the MPI implementation takes care of implementing the connection mechanism over the underlying network protocol, freeing the system software developer from the effort.

Taking an MPI approach to client connections introduces a few challenges, however. The accept/connect method needs the name of an open MPI port. If the name-publishing interface in an MPI implementation works across independently launched MPI programs (as described in Section 3), MPI_PUBLISH_NAME and MPI_LOOKUP_NAME can be used to obtain the MPI port name. Otherwise, unwieldy implementation-specific strings would have to be passed around by hand. MPI_COMM_JOIN does not have a dependency on the name-publishing interface. For situations where the name publishing approach is not feasible, this allows a UNIX socket and familiar IP and port locations to be used for

service identification. The socket is used only for the initial handshake; all other communication goes over the native transport used by the MPI implementation.

5 Fast Data Transfer

A persistent parallel service needs fast data transfer between clients and servers. PVFS2 has a few specific needs in this area.

- It needs fast communication of data between clients and servers over a number of different networking technologies, using the fastest protocol for each network, for example TCP over Ethernet, GM or MX over Myrinet, the native InfiniBand protocol over InfiniBand.
- For control messages between client and server (not for data), it needs support for heterogeneity, because clients and servers could run on different architectures. For example, Argonne's IBM BG/L system has a mix of PPC64, PPC32, and IA32 nodes.
- It needs support for communicating noncontiguous data efficiently.
- It needs support for asynchronous communication.

A substantial amount of code has been written in PVFS2 to support these needs. PVFS2 uses an abstraction called the Buffered Message Interface (BMI) [3] for portable high-performance communication over multiple networks. For control messages, PVFS2 defines an encoding scheme that converts all commands to a fixed-length, little-endian format, which allows PVFS2 clients and servers to have any mix of byte endianess or word size. (Defining this encoding correctly took many iterations.) PVFS2 implements its own way of communicating noncontiguous data, which required several thousand lines of code.

MPI is a perfect fit for all these requirements. MPI provides a portable interface for communication, and MPI implementations do the job of implementing that interface efficiently on the underlying network. The MPI Standard supports heterogeneous communication through the use of MPI datatypes. MPI implementations, however, vary in their support for heterogeneity. For example, MPICH-1 does support heterogeneous mode architectures, whereas MPICH-2 and Open MPI at present do not. The MPI Standard is limited in that there is no universal way to express certain sized types, such as 64-bit integers, and PVFS2 file handles are 64-bit values. Nonetheless, we could use MPI_LONG_LONG, which is often 64 bit; if not, we could use two MPI_INT types. MPI also supports communication of noncontiguous data through derived datatypes. Additionally, the MPI_TYPE_CREATE_STRUCT routine provides a way to create a user-defined MPI datatype out of arbitrary application data types. MPI implementations, however, have historically not performed well on derived datatypes. Nonetheless, various research efforts have demonstrated that derived datatypes can be implemented in a way that delivers good performance [13, 15]. We hope MPI implementations will devote effort to optimizing derived datatypes. MPI also supports nonblocking communication, which allows us to overlap communication with disk I/O.

These features of MPI make it ideally suited for use in data communication, although better support is needed from implementations in the areas of communication between heterogeneous nodes.

6 Fault Tolerance

Any persistent parallel software needs to be resilient against faults as far as possible. The robustness depends on how well the software itself is designed and implemented and on the robustness of the external components that the software uses.

In a cluster environment, each PVFS2 server represents a potential point of failure, and error recovery becomes an important consideration. To that end, the PVFS2 system operate in a stateless manner: there are no locks to revoke or leases to offer, and client tracking is not necessary. This stateless nature makes recovering from server failure much easier. PVFS2 can retry operations in order to hide transient problems. If a server failure occurs, PVFS2 operations will time out and return an error to the caller. If a server has been restarted (by hand or perhaps by a failover script), the newly restarted server will be able to service the client request.

If PVFS2 were implemented by using MPI, it would require the MPI implementation to be resilient against failure. The MPI Standard itself does not say much about fault tolerance; it is left as a quality of the implementation. But MPI does have some features that can help in writing resilient programs. For example, MPI has a well-defined mechanism for error returns from functions, and users can specify their own error handlers. The default error handler is that the entire job aborts on error, but users can change that to "errors return" or define their own error handler. MPI also has the notion of intercommunicators for two groups of processes (for example, clients and servers) to communicate. When two independently started processes connect to each other and communicate over the intercommunicator, the failure of one process need not cause the other process to die.

Most MPI implementations, unfortunately, are not robust against errors. For example, if the connection between two processes is lost, the entire MPI job may abort; or if a single process is killed, the entire MPI job may get killed. This kind of failure will not be good for a parallel file system that uses MPI. Although there are some efforts at building fault-tolerant MPI implementations [1, 6], more work is needed in this area.

Another area where MPI can help is in the parity calculation for a software-RAID like approach providing fault-tolerance for data stored on the parallel file system. Gropp et al. [8] proposed a *lazy redundancy* scheme that makes use of both MPI-IO consistency semantics and the MPI collective functions MPI_REDUCE_SCATTER and MPI_REDUCE. Implementing this scheme becomes much easier when PVFS2 servers are based on MPI, because the servers could simply use these collective calls (more on this in Section 7).

The processes providing the parallel service can only communicate with each other once they have established an MPI communicator. At one extreme we could establish many two-process communicators. Having all these communicators makes the system resilient to failure. If a process dies, communication can be carried out over a different communicator. On the other hand, so many communicators greatly complicates any all-to-all or one-to-many messaging algorithms. At the other extreme we could establish an all-encompassing communicator spanning all processes. In exchange for simplified communication, such a system would be more fragile. If any process died, the surviving process would need to detect that failure and coordinate the creation of a new all-encompassing communicator Further, we would need this reconstruction process to maintain the properties of MPI communicators (context, fixed identifiers) that make them so useful.

7 Collective and Aggregate Operations

In PVFS2, many operations require multiple steps performed across many servers. Creating a new file requires instantiating a single metadata entry and a data file entry on each server. Removing a file requires removal of the corresponding metadata and directory entries, followed by removal of the data file from each server. A `stat` system call needs to collect partial file size information from each server before returning the total size of a file. While the client code makes just one function call for these operations, the underlying library carries out a one-to-many operation. The client library posts these messages as nonblocking sends to the servers and waits for their response.

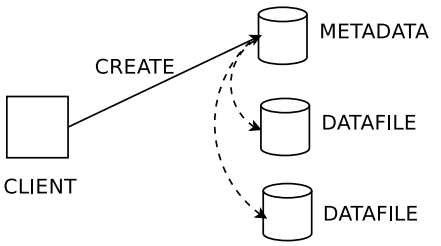
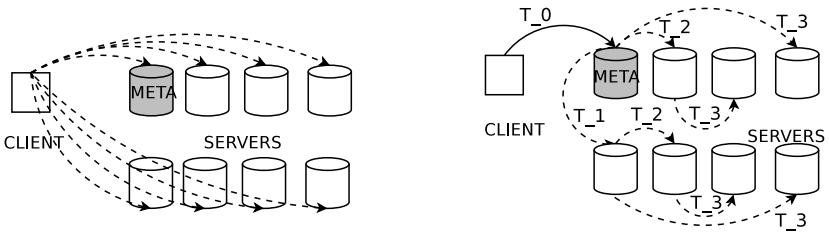


Fig. 3. *An aggregate operation lets a single* `create` *request initiate creation of the metadata entry and datafile entries on each server. The servers could potentially be better connected to each other than clients (as in a WAN), yielding fewer messages, better performance, and lower latency.*

An alternative approach would have clients send a single "create file" message to one of the servers and have servers then orchestrate actions on the client's behalf, as described in [4]. This approach simplifies the synchronization of operations and leads to the natural use of structured communication patterns such as broadcasting an operation request by using a tree-based algorithm as shown in Figure 4(b). We call these higher-level messages "aggregate operations" because they result in a collection of operations across multiple servers.

Aggregate operations also make deployment over the wide-area more efficient. We can easily imagine a topology where the servers are located near to each other while the clients may be quite far away, network-wise. These aggregate messages



(a) Independent file removal. 8 timesteps

(b) Collective file removal. 3 timesteps

Fig. 4. *File removal requires deletion of the data file on each server. The independent approach has little room for optimization, requires careful coordination to keep metadata consistent, and needs $O(N)$ timesteps to complete. The collective approach simplifies metadata updates and requires only $O(log(N))$ timesteps.*

mean fewer network round trips between clients and servers and lower latency. The servers can exchange messages with each other over their local network and send a single response over the long-haul, high-latency link.

MPI is well known for its collective operations, such as broadcast, allreduce, and scatter/gather. Many implementations have optimized collective operations [14]. The collective communication operations in MPI are defined to be collective over a communicator; all processes in the communicator must call them. In an application, this requirement is easy to meet. In PVFS2, however, the servers do not know which client will issue the collective operation, for example, which client will want to delete a file. PVFS2 needs to be able to respond to unpredictable client requests. In an MPI environment, servers would naturally post nonblocking collective calls or a broadcast with a "wildcard" (ANY_SRC) root that would be specified later. These calls, however, do not exist in MPI; MPI collectives are blocking calls. While there was a proposal in the MPI-2 Forum for nonblocking collectives, these did not make it into the final standard. The MPI forum decided those who needed nonblocking collectives could implement them with a thread which in turn called the blocking collective equivalent. In a server environment, however, spawning a thread for each potential client becomes untenable as the number of clients scales to the thousands and beyond. Some implementations have extensions that support these features, for example, in IBM's MPI [9] (although it has been deprecated). We are investigating the issue of how nonblocking (or wildcard) collectives could be supported as an extension to MPI, what their semantics would be, and how they could be implemented efficiently. Further, we will have to address how to provide an efficient collective implementation while also solving the fault tolerance issues brought

up in Section 6. We plan to develop a prototype implementation to explore these issues.

8 Conclusions

Writing parallel system software can be a significant undertaking. A production parallel file system such as GPFS, GFS, Lustre, or PVFS2 takes many years to develop and stabilize. Much of this effort goes into implementing many of the features that MPI already supports, and this duplicate effort could be avoided. While there are some challenges in implementing system software using MPI today, they are due mainly to the limitations of MPI implementations rather than deficiencies in the MPI Standard itself. At the same time, the addition of nonblocking collectives to MPI would make it an even more natural basis for building parallel system software.

The requirements we have discussed apply to more than just PVFS2 or other parallel file systems. For example, resource managers could use MPI dynamic process functions to launch parallel jobs (via MPI_COMM_SPAWN), and system monitoring daemons could use MPI datatypes and support for heterogeneous communication to monitor disparate resources. Desai et al. [5] used MPI to implement a variety of system-level application utilities, such as file staging, file synchronization, and a parallel shell.

We note that using MPI for implementing persistent system services does not restrict user applications to being MPI applications. The PVFS2 client could determine whether MPI has been initialized (by calling MPI_INITIALIZED) and then call MPI_INIT if it hasn't been. Clients and servers can then communicate using MPI even if they were not started as MPI programs. (Again, all implementations need to support this feature of MPI, called "singleton init."). We would expect that MPI-using applications would call MPI_INIT before making any system service calls. It would of course be an error for an application to call MPI_INIT twice.

In summary, we would like to implement PVFS2 using MPI. We hope MPI implementers will take up the challenge and develop high-quality implementations that can be used to develop system software such as a parallel file system.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

1. George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. MPICH-V: Toward a scalable fault

tolerant MPI for volatile nodes. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–18, Los Alamitos, CA, 2002. IEEE Computer Society Press.

2. Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.
3. Phillip Carns. Design and analysis of a network transfer layer for parallel file systems. Master's thesis, Clemson University, Clemson, S.C., July 2001.
4. Phillip H. Carns. *Achieving Scalability in Parallel File Systems*. PhD thesis, Dept. of Electrical and Computer Engineering, Clemson University, Clemson, SC, May 2004.
5. Narayan Desai, Rick Bradshaw, Andrew Lusk, and Ewing Lusk. MPI cluster system software. *Lecture Notes in Computer Science*, (3241):277–286, September 2004. 11th European PVM/MPI Users' Group Meeting.
6. Grahm Fagg and Jack Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *Lecture Notes in Computer Science*, pages 346–353, 2000. 7th European PVM/MPI Users' Group Meeting.
7. Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.
8. William D. Gropp, Robert Ross, and Neill Miller. Providing efficient I/O redundancy in MPI environments. *Lecture Notes in Computer Science*, 3241:77–86, September 2004. 11th European PVM/MPI Users' Group Meeting.
9. International Business Machines Corporation. *IBM Parallel Environment for AIX 5L: MPI Subroutine Reference*, third edition, April 2005.
10. Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.
11. MPICH2. `http://www.mcs.anl.gov/mpi/mpich2`.
12. The PVFS2 parallel file system. `http://www.pvfs.org/pvfs2`.
13. Robert Ross, Neill Miller, and William Gropp. Implementing fast and reusable datatype processing. *Lecture Notes in Computer Science*, 2840, September 2003. 11th European PVM/MPI Users' Group Meeting.
14. Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *International Journal of High-Performance Computing Applications*, 19(1):49–66, Spring 2005.
15. Jesper Larsson Traff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. Flattening on the fly: Efficient handling of MPI derived datatypes. In *PVM/MPI 1999*, pages 109–116, 1999.