The Interoperable Message Passing Interface (IMPI) Extensions to LAM/MPI

Jeffrey M. Squyres† Andrew Lumsdaine†
William L. George‡ John G. Hagedorn‡ Judith E. Devaney‡

Abstract—**Interoperable MPI (IMPI) is a protocol specification to allow multiple MPI implementations to cooperate on a single MPI job. Unlike portable MPI implementations, an IMPI-connected parallel job allows the use of vendor-tuned message passing libraries on given target architectures, thus potentially allowing higher levels of performance than previously possible. Additionally, the IMPI protocol uses a low number of connections, which may be suitable for parallel computations across WAN distances. The IMPI specification defines a low-level wireline protocol that MPI implementations use to communicate with each other; each point-to-point and collective function in MPI-1 automatically uses this low-level protocol when communicating with remote a MPI implementation. When running IMPI jobs, the only change visible to the user is the sequence of steps necessary to run the job; any correct MPI program will run correctly under IMPI. In this paper, we provide an overview of IMPI, describe its incorporation into the LAM implementation MPI, and show an example of its use.**

I. Introduction

Since the publication of the MPI-1 standard [1], a large number of high-quality MPI implementations have been made available. The ease of obtaining an MPI implementation has led to a new level of portability; parallel codes can now run on a variety of different operating systems and architectures simply by recompiling the same source code with a different implementation of MPI. Indeed, all parallel vendors now have their own implementations of MPI that are optimized for their architectures. There are also a number of freely available implementations of MPI, notably LAM/MPI [2] from the University of Notre Dame, and MPICH [3] from Argonne National Laboratory.

Since only the functionality of the MPI API is specified, each implementation is unique in its underlying abstractions and assumptions. Vendor implementations, for example, are tuned for specific architectures to optimize performance. So in one sense, one of the goals of the MPI standard has almost worked against it – all current MPI implementations are unable to interoperate with each other. It is not possible, for example, to run a single parallel job in a that spans multiple machines (from different vendors) and still use the respective vendors' highly-tuned MPI implementations. While the freely available implementations support heterogeneous environments, message passing performance suffers since the freely available implementations do not provide vendor-quality architecture specific optimizations.

The Interoperable MPI (IMPI) Steering Committee was formed to solve these kinds of issues. The committee consisted of vendors who already have high performance MPI implementations, with the National Institute of Standards and Technology (NIST) facilitating the meetings.

† Dept. of Comp. Sci. and Eng., University of Notre Dame, Notre Dame, IN 46556 (squyres@cse.nd.edu, Lumsdaine.1@nd.edu).

‡ National Institute of Standards and Technology, Gaithersburg, MD 20899 (william.george@nist.gov, john.hagedorn@nist.gov, judy.devaney@nist.gov)

The IMPI Steering Committee has published a proposed standard for interoperability between MPI implementations that will address these issues [4]. The main idea of the proposed standard is to mandate a small set of protocols for starting a multi-implementation MPI job, passing user messages between the implementations, and shutting the job down. Note, however, that the proposed IMPI standard does *not* mandate any behavior within an MPI implementation – it only mandates behavior *between* MPI implementations.

A. LAM/MPI's Role in IMPI

The LAM/MPI team was asked to join the IMPI Steering Committee as a non-voting member part way through the process. This allowed an implementation of IMPI to become publicly available both as a "proof of concept" work as well as a verification and validation mechanism for the proposed standard.

Implementing IMPI in LAM/MPI continues a long-standing history of freely available implementations of MPI providing not only the first implementation of MPI functionality, but also providing both impetus and a code base for vendor MPI implementation efforts. In addition to incorporating IMPI extensions into LAM/MPI, the LAM/MPI team wrote an implementation-independent IMPI server [5] (described in Section II-A).

B. Related Work

The PVMPI [6], [7] project from the University of Tennessee was a first attempt to join multiple MPI implementations in a single job. It utilized PVM as a communications bridge between incompatible MPI implementations. PVMPI used non-MPI functions to join separately started MPI jobs and communicate between them. While this approach was successful in creating a larger MPI universe, it did not provide a seamless communication realm, and forced users to understand both PVM and MPI in order to write heterogenous programs.

The PVMPI project evolved into the MPI Connect effort [8]. While using many of the same ideas from PVMPI, MPI Connect utilizes the profiling layer in MPI to intercept messages for remote ranks and re-send them using PVM. While using many fewer non-MPI functions than PVMPI, MPI Connect still relies on PVM and some non-portable function calls. Additionally, even with the mostly-native interface to MPI, only an intercommunicator is provided between MPI implementations, such that collective communications are not possible both with MPI-1 implementations,[1] and between different MPI-2 implementations.

[1] While it is usually possible to merge an intercommunicator into an intracommunicator and use it to perform collective communications, it is unlikely that `MPI_INTERCOMM_MERGE` will work between multiple MPI implementations.

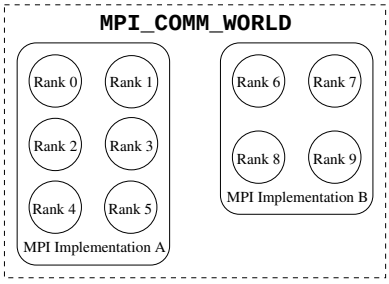

Fig. 1. The MPI_COMM_WORLD communicator is defined to include the ranks from all MPI implementations.

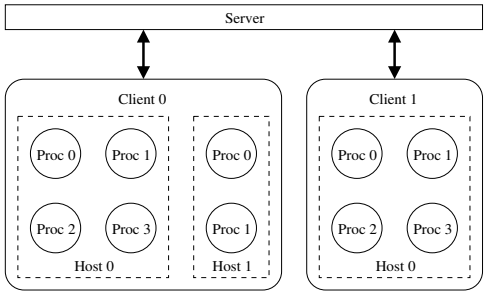

Fig. 2. The IMPI server is a rendezvous point for the MPI implementations. In this example, two clients are shown. The first has two hosts, one with four procs, the other with two procs. The second client only has one host which contains four procs. That this is a possible host decomposition for Fig. 1.

Unify [9], a project from the Engineering Research Center at the University of Mississippi, was designed as an upgrade tool for PVM programmers to port their software to MPI. Unify consisted of a subset of the MPI-1 API built on top of PVM. This allowed users to mix PVM and MPI calls in the same program. While this project did not allow a single job to span multiple MPI implementations, it does show the use of multi-protocol message passing, which is critical in IMPI (as well as other message passing systems).

C. Paper Overview

Section II gives an overview of the IMPI standard, and introduces terminology used in this paper. Section III presents an overview of the internals of LAM/MPI. Section IV describes how IMPI was implemented into LAM/MPI. Section V presents some timing results using IMPI across a WAN. Finally, in Section VI, we discuss our conclusions and list future work.

II. IMPI Overview

One of the main goals of IMPI is to provide a programming interface identical to MPI while utilizing multiple implementations of MPI in a single parallel job. That is, provide implementation-spanning communicators that can be used in the same way that MPI intracommunicators are used. Programs that run correctly with MPI should require no source code changes to run correctly under IMPI. One major issue that previous efforts were not able to solve because they could not affect the underlying MPI implementation is the completeness of MPI_COMM_WORLD.

To make the programming model truly transparent, all ranks (regardless of which MPI implementation they are in) should share a common MPI_COMM_WORLD. Fig. 1 shows the IMPI-defined MPI_COMM_WORLD for a typical multi-implementation job. Each process receives a unique rank number (the order is strictly defined in the IMPI standard). All of the MPI-1 functions may be used inside of MPI_COMM_WORLD; there are no restrictions on the type of communication performed between the multiple MPI implementations.

MPI-2 [10], [11] functions, however, are not presently supported in IMPI. More specifically, the behavior of MPI-2 functions is not defined on communicators that contain ranks from multiple MPI implementations.

The IMPI standard is divided into four parts: startup/shutdown protocols, a data transfer protocol, collective algorithms, and a centralized IMPI conformance testing methodology.

A. Terminology

This text (and the IMPI standard) commonly uses the term "MPI implementations" to refer to the separate MPI universes that are joined by IMPI into a single parallel job. This term is not only used to distinguish between multiple implementations of MPI, but also between multiple instances of the same implementation of MPI. For example, IMPI can join two LAM/MPI instances that were started independently (perhaps because of large network distance).

There are four types of IMPI entities: a server, clients, hosts, and procs. They are described as follows (and shown in Fig. 2):

- The *server* is a rendezvous point for the MPI implementations to meet upon startup.
- There is one *client* per MPI implementation; it acts as a representative for that implementation at the server. There can be a maximum of thirty-two clients in a single IMPI job.
- Each client must have one or more *hosts*.
- Each host must have one or more *procs* (processes). Although the specific use of hosts is implementation dependent, their abstract purpose is to group procs in a single MPI implementation. For example, hosts can be used to group procs physically located on the same SMP.

The IMPI server is considered a separate part of the IMPI environment – it is not tied to any particular MPI implementation. The University of Notre Dame has published an implementation of the IMPI server that is publicly available for download.

B. Startup / Shutdown Protocols

A two-step process is needed to launch MPI-spanning parallel jobs. A "server" process is first launched that will act as a rendezvous point for each implementation; its purpose is to collect and disseminate job-specific information. The server is specifically designed to be as "dumb" as possible – it only knows how to connect and authenticate a client, then rebroadcast data to all the clients (the content of which it does not understand).

After each of the clients is authenticated to the server, it sends information about its local MPI job, such as how many hosts it represents, how many procs each hosts has, etc. The server collects messages from all the clients and broadcasts a collated copy back to each client. In this way, each client learns information about all the other clients and can independently construct an identically ordered MPI_COMM_WORLD.

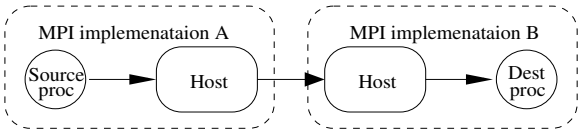

Fig. 3. Sample routing of a message from one proc to another.

Additionally, several variables are exchanged between clients during startup. For example, the maximum message packet size that is supported by each client is included in the collated data. Each client compares its maximum size to the maximum size supported by each other client, and chooses the smaller of the two. This is referred to as "negotiation" between clients, even though the decisions are performed in an independent and distributed fashion. Note that all negotiation is done on a client-pairwise basis; the values decided upon for clients A and B may be different than the values used between clients B and C.

After startup, the hosts create a fully connected mesh of TCP/IP sockets between themselves (this implies that the clients disseminate the relevant collated information to their hosts). TCP/IP was chosen as the interconnection network because it is a least common denominator that can be assumed between parallel resources. The TCP/IP mesh will be used to pass user MPI messages between MPI implementations. Once this mesh is created, the server and clients sit idle until the end of the job.

This startup protocol is likely to happen during an implementation's MPI_INIT (but may occur sooner). Once MPI_INIT has completed, MPI_COMM_WORLD can be used to communicate with any rank in the job. Section II-C describes the protocols used to pass user MPI messages between procs.

In MPI_FINALIZE, each proc will send a message to its local host indicating that it is shutting down. When the host receives the shutdown message from all of its procs, it transmits a shutdown message to its client. Similarly, when the client gets shutdown messages from all of its hosts, it transmits a shutdown message to the server. Finally, the server shuts down when it receives shutdown messages from all of its clients.

C. Data Transfer Protocol

Messages within a single MPI implementation can utilize the vendor-tuned code for high bandwidth/low latency message passing. But when a point-to-point message is sent to a rank in another MPI implementation, it must be relayed through the local host to the remote host. The remote host will then ensure that the message is routed to the destination proc. Note that IMPI only mandates the protocol between the hosts – no restrictions are placed upon how messages are routed within the procs and hosts in a single MPI implementation. This allows MPI implementations to use the most efficient methods of routing within their local communication space. Fig. 3 shows a possible message routing between two MPI implementations.

The IMPI Steering Committee decided on this design desspite potentially creating a bandwidth bottleneck for two reasons:
1. Communications latency using IMPI protocols is likely to be orders of magnitude greater than vendor-tuned latency, regardless of whether a direct connection is made between IMPI entities or when additional hops are used. Indeed, communications between implementations currently must use TCP/IP, which is likely to be much slower than an implementation's native message passing mechanism.
2. The increase in complexity and potential loss of efficiency incurred by forcing every rank to be concerned with not only making communication progress within its own implementation but also with non-local ranks using the IMPI protocols was judged to be too great.

Loosely translated: communications using IMPI are expected to be slow. Adding additional hops not only adds little little additional overhead (since the TCP/IP communication will be slow anyway), it greatly simplifies the implementation. This raises an issue that, while affecting the design of this implementation, is outside the scope of this paper: while correct MPI applications can run without modification over IMPI, they should be modified to minimize communication between IMPI hosts.

C.1 Flow Control

Communication between hosts is packetized. A packet throttling mechanism prevents resources from being consumed without bound in hosts. Two values are negotiated at startup – `ackmark` and `hiwater` (where $1 \leq$ `ackmark` $\leq$ `hiwater`). These two numbers create a sliding window for acknowledgements – a protocol ACK is required for every `ackmark` packets received, but a sender may send up to `hiwater` packets before waiting for the protocol ACK from the first `ackmark` packets.

C.2 Data Protocols

There are two basic protocols for sending user MPI messages between hosts. Both protocols include unique message ID numbers to allow for matching of requests, acknowledgements, etc.

The "short" protocol is for non-synchronous mode messages that are less than `maxdatalen` bytes (where `maxdatalen` is negotiated during startup). Such messages are sent immediately, and are only subject to the flow control mechanism. "Long" messages are classified either as message that are longer than `maxdatalen` bytes or require some kind of explicit synchronization (such as synchronous mode messages).

Long messages are fragmented in to packets of size `maxdatalen` bytes. The first packet is sent eagerly (just like a short message), and is marked as the first of a long message. The destination host will return an acknowledge when it has allocated enough resources to receive the full message. Upon receiving the acknowledgement, the sender will queue the remaining packets to be sent to the destination.

Synchronous message always use the long message protocol; the acknowledgement packet from the destination host serves as an indicator to the MPI implementation that the synchronous mode send may complete.

C.3 Cancellation

IMPI also supports message cancellation. If no part of a message has been sent to the destination host, cancellation is local. If at least one packet has been sent, a cancel request must be sent containing the message ID to be canceled.

On the remote host, if the message has not been received by the user program, it can be canceled. Either way, the host must send back an ACK indicating whether or not the message was successfully canceled.

C.4 Finalization

When a host will no longer require its IMPI channels for communication (e.g., when all of its procs invoke MPI_FINALIZE), it will send finalization packets to each of the other hosts. The host may not close a socket until it receives a finalization acknowledgement from the host on the other end of the socket.

This allows some flexibility to MPI implementations. For example, if a proc tries to send a message to a proc on a host that has shutdown, a high quality implementation will likely fail the message immediately. However, the implementation is free to send the message anyway using its normal queue mechanism, where the message will eventually be dropped (potentially causing deadlock in the user's application).

D. Collective Algorithms

For collectives involving multiple implementations to work, the exact algorithm and communication pattern must be mandated so that each implementation knows its role in the overall collective action. The proposed IMPI standard includes pseudocode algorithms for each of the MPI-1 collective routines. Note that not only are the behaviors of collective functions such as MPI_BARRIER, MPI_BCAST, and MPI_SCATTER mandated, but communicator constructor and destructor functions such as MPI_COMM_CREATE and MPI_COMM_FREE must also be mandated to ensure that communicator contexts are unique across all MPI implementations.

Since the TCP/IP communications between hosts is likely to be slow compared to message passing between procs in a single implementation, the IMPI-mandated collective algorithms attempt to minimize communication across inter-implementation channels. Most collectives have "local" phase (within a single implementation) and "global" phase (coordination between all implementation). The behavior of local phases is not mandated, allowing implementations to use optimized mechanisms to achieve the mandated results. Actions during the global phases are mandated with pseudocode in the IMPI standard.

For example, a barrier across multiple implementations uses two local phases and a global phase. Each implementation has a *local master* rank among the ranks performing the barrier. The first phase has each rank synchronize with their local master (i.e., within their respective implementations). The second phase is a synchronization between the local masters. The final phase is a second local synchronization. Fig. 4 shows a barrier distributed between four MPI implementations.

E. NIST Conformance Tester

NIST has implemented an IMPI conformance testing tool. A Java applet is available on the main IMPI web page (`http://impi.nist.gov/IMPI/`) that can provides access to a back-end IMPI simulator. The simulator can emulate an IMPI server and any number of IMPI hosts and procs.

C source code is also available on the web page that can be compiled and linked against an IMPI implementation. It can then be run in conjunction with the NIST Java applet to test the local IMPI implementation against the NIST simulator. A number of test scripts (which are very similar to C MPI programs) can be sent to the C program from the back-end simulator to be executed. These scripts test various aspects of the local IMPI implementation. Once an IMPI implementation passes the tests with all of the NIST tests, it can theoretically interoperate with any other IMPI implementation that passed the NIST tests.

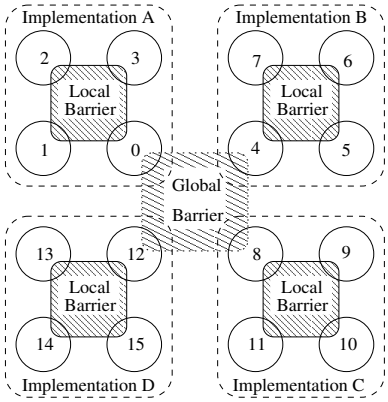

Fig. 4. MPI_BARRIER across four MPI implementations, each with four local ranks. Phase one is a local synchronization. Phase two is a global synchronization between representative ranks of each implementation. Phrase three is a final local synchronization.

III. LAM/MPI Overview

The Local Area Multicomputer (LAM) implementation of MPI grew out of the Trollius project from the Ohio Supercomputer Center [12]. The Trollius project was originally targeted at Transputers, but eventually grew in scope to include general parallel computing. With a rich set of infrastructure and communication tools, an MPI layer was added to the top level of the Trollius software. Over time, the MPI software has become the main use of LAM.

After the original LAM/MPI developers individually left the OSC, LAM/MPI became an orphaned project. Through ties with the original developers, the Laboratory for Scientific Computing at the Notre Dame agreed to become the owners of LAM. The web site, mailing list, and source code repository moved to `nd.edu` in early June, 1998.

The design of IMPI reflects, in part, its heritage in LAM/MPI. Two of the original LAM/MPI developers – who were each representing different vendors – were chapter authors on the IMPI Steering Committee. As such, the overall design of IMPI is similar to that of LAM itself.

A. Features

LAM/MPI is more than just a communication library for MPI – it contains a rich set of features that are attractive to both developers and end users.

- A full implementation of the MPI-1.2 standard.[2]
- Implementation of much of the MPI-2 standard, including dynamic processes, one-sided communication, C++ bindings for MPI-1 functions, and parallel I/O.[3]

[2] With the exception of the ability to MPI_CANCEL sent messages. Canceling sent messages in a parallel environment is an extremely difficult problem; since very few LAM/MPI users have asked for this functionality, the LAM team has decided not to implement this functionality.

[3] LAM includes the ROMIO package from Argonne National Labs [13], [14] as the implementation of parallel I/O.

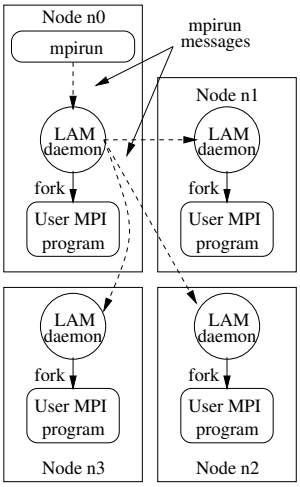

Fig. 5. Example showing how `mpirun` works in the LAM/MPI environment. `mpirun` sends execution messages to the local LAM daemon, who, in turn, distributed them to the remote LAM daemons. Each daemon then starts up the user MPI program.

- Support for two kinds of shared memory (on node) / TCP/IP (off-node) multi-protocol message passing.
- Persistent MPI run-time environment that provides (among other things) fast parallel job startup, robust process control, and run-time monitoring of parallel jobs.

B. Run Time Environment

LAM/MPI provides a persistent run-time environment for MPI programs. Users initially launch LAM daemons on each machine that they wish to use in MPI with the `lamboot` command. The LAM daemons are mainly used for process control, an out-of-band communication channel for meta data, and a monitoring/debugging tool for user programs. Once the LAM daemons have been launched, MPI programs can be launched across the resulting "parallel machine". Fig. 5 shows an example of how the LAM daemons are used to `mpirun` user programs.

LAM/MPI provides a convenient command, `lamclean`, than can be used to kill all running user programs in a booted LAM, and clean up any unreceived messages. `lamclean` is frequently used to kill runaway or deadlocked processes, especially while developing and debugging user MPI applications.

When the user is finished with MPI, they can kill any running programs and take down the run-time environment with the `wipe` command.

C. Code Structure

The communication library of LAM/MPI is divided into three parts: the MPI layer, the request progression interface (RPI) [15], and the Trollius core. The MPI layer is actually a somewhat-thin layer on top of Trollius and RPI functionality. A typical MPI function is fairly simplistic – it checks parameters, performs some "bookkeeping" functionality, and uses the underlying RPI or Trollius for many of the more complex functions. The RPI is discussed in Section III-E. Finally, Trollius contains many "kitchen sink" kinds of functions, and provides a backbone for most services in LAM (including the LAM daemons) that is invoked throughout LAM. Fig. 6 shows a diagram of the LAM communication library.

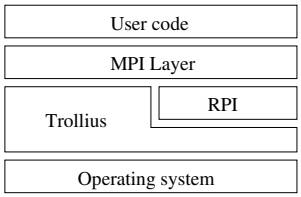

Fig. 6. Main components of the LAM/MPI communication library.

D. MPI Layer

One of the main functions of the MPI layer is to create and maintain communication queues. All send and receive communications within LAM/MPI are collectively known as *requests*. Unifying all types of communication under a single nomenclature allows the use of a uniform management system.

For example, MPI_SEND generates a request that contains information such as the buffer, count, datatype, tag, destination rank, and communicator of the message to be sent. If the queue is empty, the request is passed directly to the RPI to be processed immediately (this is known as the *short-circuit optimization*). If the queue is not empty, the new request is marked as *blocking* (described below), and placed on the queue. The RPI is then invoked to progress the queue. Since the request was marked as blocking, the RPI will not return until the message has been fully sent.

E. Request Progression Interface

Requests are processed through LAM's request progression interface (RPI). The RPI is responsible for all aspects of communication with other MPI ranks – it progresses the communication requests that were formed and queued in the MPI layer. That is, the RPI is responsible for actually moving data from one rank to another. Once the RPI finishes a request, it marks the request as completed (the MPI layer will dequeue it).

The RPI was designed to be a separate layer; the API for the RPI consists of ten primitives, and is documented in [15]. Maintaining a separation between abstract message passing and device-specific drivers is not only good software engineering, it also allows the addition of native support for new communication devices without changing any other parts of LAM/MPI.

There are two classifications of RPIs: daemon-based (lamd) and client-to-client (c2c). The lamd RPI uses the LAM daemons for all user communications. Fig. 7 shows the hops that a message must travel from rank A to rank B using the lamd RPI (note the similarity to the IMPI design shown in Fig. 3). Although incurring extra hops, the lamd RPI allows for extra monitoring and debugging capabilities. Additionally, the lamd RPI allows for some degree of true asynchronous communication. Since the LAM daemon is running in a separate process, it can make progress on message passing regardless of what the user application is doing.

c2c RPIs do not use the LAM daemons for user communications. Instead, some other interconnection network is used, which greatly decreases message latency. Note that while it is assumed that clients will be directly connected to each other (in

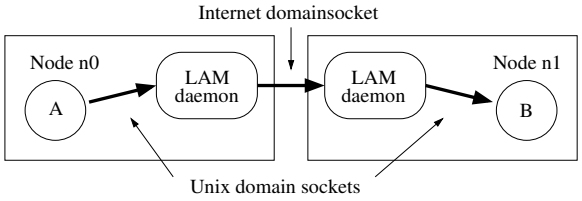

Fig. 7. The lamd RPI design. Each message must make three hops to get to its destination.

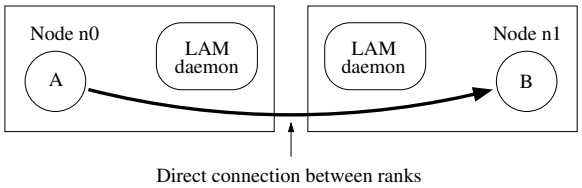

Fig. 8. The c2c RPI design. The LAM daemons are not used for user communication. The three RPIs that are currently included in the LAM/MPI distribution use internet domain sockets for off-node communication.

software) for speed, this does not need to be the case. A sample c2c RPI connection scheme is shown in Fig. 8.

LAM/MPI currently includes three c2c RPI implementations:
1. **tcp**: The TCP RPI uses internet domain sockets between ranks in MPI_COMM_WORLD. TCP is LAM's default RPI.
2. **uysv**: The USYSV RPI is the same as the TCP RPI, except that shared memory is used for communication between ranks on the same node. Spin locks are used to lock the shared memory between ranks.
3. **usysv**: The SYSV RPI is the same as the USYSV RPI, except that SYSV semaphores are used for locking the shared memory between ranks. The blocking nature of semaphores can give higher performance than USYSV on uniprocessor machines.

LAM/MPI currently does not allow more than one c2c RPI to used simultaneously. As such, the specific c2c RPI must be selected when LAM/MPI is configured. The choice of lamd vs. c2c RPI can be made at run time with the `mpirun` command.

IV. Implementation of IMPI in LAM/MPI

Aside from some debugging and maintenance, implementing IMPI was the first large-scale project in LAM/MPI that we attempted. As such, the current package is actually a third generation implementation of the IMPI standard.[4] This implementation of IMPI was written in C++ in order to take advantage of some basic object constructs as well as make use of the Standard Template Library (STL) [16], [17].

The following overall design goals were specified by the LAM team when implementing IMPI:
1. A separate daemon (the `impid`) will implement the role of the IMPI client and host.
2. There will only be one `impid` per IMPI job; subsetting into multiple hosts will not be supported.
3. The `impid` must be transparent to the user; it will automatically be started, die gracefully when the program finishes, and be able to be aborted with the `lamclean` command.

[4]The first two implementations represented a steep learning curve about the LAM/MPI infrastructure by the Notre Dame LAM team. These two implementations generally barely worked, and resulted in design changes for the next generation.

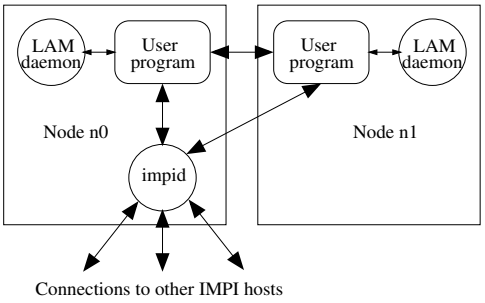

Fig. 9. The `impid` acts as a separate process in the IMPI job. Each rank in the local LAM opens a connection to the `impid` using the current RPI.

4. An IMPI job must be able to use any of the four RPIs.

A. *The* `impid`

In order to achieve true asynchronous communication and to maximize bandwidth, the IMPI host and client was implemented as a separate daemon: the `impid`. The `impid` acts as the IMPI client during startup and shutdown, and as the IMPI host during the rest of the job. After the startup, each local rank makes a connection to the `impid` using the current RPI. This connection will be used for communication with any rank that is not in the local LAM. Fig. 9 shows how the `impid` fits into a LAM/MPI user job.

A.1 Separate Daemon

Having the IMPI host in a separate process space – similar to the rationale for using the main LAM daemon – allows for some degree of communication progress independent of the user program. That is, a separate thread of control can progress message queues as well as provide buffering for local and remote messages. In one sense, the `impid` acts as a post office for messages originating from local ranks that are destined for remote IMPI hosts (and vice versa). Just as a snail mail letter reaches its destination after being dropped off at a post office without the sender knowing or caring how it gets there, so too the `impid` takes care of communication with remote IMPI hosts. In many cases, for example, a user process can MPI_SEND a message and continue processing long before the message reaches its destination. This can allow for true overlap of communication and computation – this is especially important since IMPI communication is expected to be slow.

Additionally, since LAM/MPI provides a persistent run-time environment for MPI programs, multiple IMPI jobs can be running in the same universe simultaneously. Having a separate process for the `impid` (potentially on different nodes) not only segregates communication from multiple IMPI jobs, it simplified the implementation since no additional logic was necessary to determine which IMPI job a particular message belongs to.

A.2 Startup Procedure

To further reduce complexity and take advantage of the infrastructure already in LAM/MPI, the `impid` was implemented as an "almost MPI" process. By using MPI for the majority of message passing, the `impid` automatically uses underlying data conversion, message fragmenting and flow control, the LAM progress engine, etc.

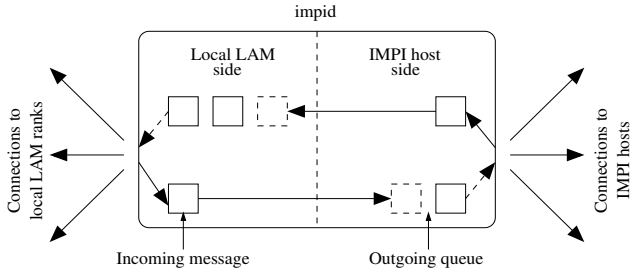

Fig. 10. The `impid` is split into two halves; one for receiving and processing messages from local LAM/MPI ranks, the other for receiving and processing messages from remote IMPI hosts. A message that is received on one side is queued for transmission on the other side.

That is, the `impid` calls MPI_INIT, uses different flavors of MPI_SEND/MPI_RECV to communicate with local ranks, and eventually calls MPI_FINALIZE. Indeed, the `impid` is initially launched via MPI_COMM_SPAWN from within MPI_INIT of the user's program. The `impid` is an "almost MPI" process, however, because it was impossible to avoid circumventing some of the normal MPI mechanisms in certain cases.

The user must specify two command line arguments on the `mpirun` command line: the IP name or address of the IMPI server and the rank number of this IMPI client. These arguments are passed to the `impid` from rank 0 in MPI_COMM_WORLD. The `impid` then connects to the IMPI server and performs the startup negotiation sequence as described above. Once the negotiation has completed, the `impid` creates a socket to each other IMPI host, creating a fully-connected topology of hosts.

Since the `impid` was started via MPI_COMM_SPAWN, it has an intercommunicator containing the local LAM ranks. This intercommunicator is merged into an intracommunicator so that the IMPI startup negotiation data can be broadcast to all ranks. When the ranks receive the negotiation data, MPI_COMM_WORLD is formed, MPI_INIT returns, and the MPI portion of the user program begins.

A.3 Design

The IMPI host in the `impid` is split in two halves. One side communicates with the local LAM/MPI, the other communicates with other IMPI hosts. When a message is received on one side, the message buffer and destination information is transferred to the other side where it is queued up to be sent. Each side of the `impid` follows a complex state machine to process an incoming message; both sides can receive multiple types of messages, each of which require different (but related) handling. Fig. 10 shows the basic division in the `impid`.

Ideally, it would have been possible to assign multiple threads to each side of the `impid`. For example, one thread could receive incoming messages while another could progress the outgoing queues. This design allows for a blocking event-driven semantics; threads that are waiting for something to happen will block until they are needed. Unfortunately, since the underlying LAM infrastructure is not thread safe, implementing this design was not possible. Instead, both sides of the `impid` must be polled for activity. A side effect of only having one thread of control is that the `impid` must never block; only non-blocking communications can be used, except in situations where it is known that a blocking communication will complete immediately.

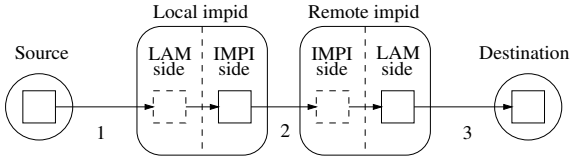

Fig. 11. Path of a short message from one IMPI proc to another. 1) The message is sent from the source proc to the `impid`. It immediately crosses from the LAM side to the IMPI side, where 2) it eventually gets sent to the remote `impid`. The message immediately crosses to the LAM side, where 3) it gets sent to the destination local LAM rank.

Since the `impid` is likely to be running on the same node as a user process, it is vital that the `impid` take as few CPU cycles as possible. Although unfortunately locked into a polling model, the `impid` reduces its overall activity by using the `poll(2)` system call with a non-zero timeout. That is, the `impid` allows itself to be blocked, thereby guaranteeing CPU cycles for the user program. Pseudocode for the main `impid` polling loop is shown below:

```
while (job_is_running) {
  // Check for activity on IMPI side
  while(poll(hosts, nhosts, timeout) > 0)
    cleanup_finished_sends();
  // Check for activity on LAM side
  do {
    MPI_Testany(npending, requests,
      &index, &flag, MPI_STATUS_IGNORE);
    if (flag)
      cleanup_finished_request(index);
  } while(flag);
}
```

Notice that there is no "sending" code evident in the above pseudocode. A convenient side effect of having the event-driven model is that sends are always triggered by receives. For example, when a short non-synchronous message is received from the LAM side, it is immediately placed in the outgoing host queue.[5] Placing a message in the queue triggers an attempt to progress the queue according to current flow control values (see Sec. II-C.1). The queue will be progressed as much as possible, after which the polling loop will continue.

As another example, it is possible that the queue will not have drained before flow control values indicate that sending must stop. Eventually, the remote host will send an acknowledgment indicating "ok to send more packets". When this acknowledgement is received, it will trigger an attempt to progress the send queue again.

A.4 General Operation

Fig. 11 shows the process for sending and receiving a short message between two instances of LAM/MPI. Two instances of LAM are shown both for simplicity; the diagram for connecting LAM/MPI to another IMPI implementation is similar.

Long and synchronous messages use a similar, but more complicated, protocol. Fig. 12 shows the process for sending and receiving a long message. Except for the addition of the ACK sent

[5] Unless the queue is empty and flow control values indicate that it is permissible to send, in which case it is sent immediately. This is similar to the short-circuit optimization that is used in the MPI layer.

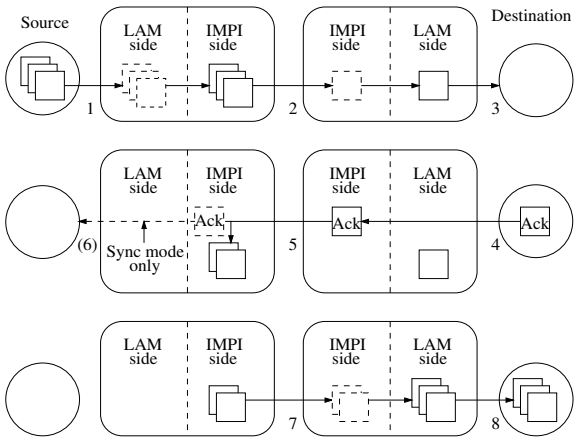

Fig. 12. Path of a long message from one IMPI proc to another. 1) The message is sent from the source proc to the `impid`, and is queued on the IMPI side. 2) The first packet is sent to the remote `impid`, where it is buffered on the LAM side. 3) A "ping" message is sent to the destination rank. 4) When the ping is acknowledged, 5) an ACK is send from the remote `impid` to the local `impid` who 6) sends a "ping" back to the sender (for synchronous mode sends only), and 7) sends the remaining packets of the message to the remote `impid`. 8) The remote `impid` then sends the entire message to the waiting destination.

back to the source (step 6), the process for a long synchronous message is the same. The process for a short synchronous message is slightly different: the message is sent to the local LAM rank in step 3 (instead of just a ping), and steps 7 and 8 are not necessary.

A.5 Local LAM Side

On the local LAM side of the `impid`, MPI is used to communicate with the local LAM/MPI ranks. The `impid` posts persistent receives for the following kinds of messages:

- **"Lamgiappe" headers**: Lamgiappe headers are described more in Sec. IV-B. This is step 1 in Figs. 11 and 12.
- **Synchronization ACK**: A LAM rank will send a synchronization ACK back to the `impid` after it receives a "ping" message from the `impid` indicating that a long or synchronous mode message is being received. This is step 4 in Fig. 12.
- **Abort**: If a local LAM rank invokes MPI_ABORT, the `impid` will exit immediately. The other ranks in the local LAM will be killed by their respective LAM daemons.
- **Finalization**: This message is sent to the `impid` when a LAM rank invokes MPI_FINALIZE.

The array of requests given to MPI_TESTANY in the main polling loop not only contains the requests for the persistent receives listed above, it also includes requests from any pending sends to local LAM ranks. Hence, in addition to making progress on incoming messages, the MPI_TESTANY also makes progress on outgoing messages.

A.6 IMPI Side

The IMPI side of the `impid` can receive the following kinds of packets from other IMPI hosts:

- **Sync data**: This packet type is used to send the first packet of long and synchronous mode messages (step 2 in Fig. 12). For short messages, the message is immediately sent to the local LAM rank, and a placeholder is inserted into the "waiting for ACK" list on the LAM side. When the ACK is received from the local LAM (step 4 in Fig. 12), an ACK is returned to the remote IMPI host (step 5 in Fig. 12). Similarly, for long messages, a ping is sent to the local LAM rank, and an entry is inserted in the "waiting for ACK" list. When the ACK is received, an ACK is sent to the remote IMPI host. The remaining packets are sent using the Data packet type (step 7 in Fig. 12).
- **Data**: This packet type is used to send short messages as well as the remaining packets of a long message (step 2 in Fig. 11 and step 7 in Fig. 12).
- **Sync ACK**: These ACKs (step 5 in Fig. 12) trigger sending the remaining packets of a long message, and the sending of an ACK back to the local LAM rank for synchronous messages.
- **Flow control**: Flow control packets decrease the "unacknowledged" packet count by `ackmark`.
- **Cancellation**: Acknowledgements of cancel requests will be sent from remote IMPI hosts. However, the `impid` will currently not receive such ACKs, because LAM does not support the cancellation of sent messages. Hence, LAM will never request a remote IMPI host to cancel a message.
- **Finalization**: When a remote IMPI host closes down, it must send a notice to each other IMPI host. This prevents hosts from interpreting the close of the connecting socket as an error.

B. MPI Layer Hooks

In order to enable IMPI to use any of the existing RPIs, it was decided to implement IMPI hooks in the MPI layer. Most of the hooks deal with the manipulation *shadow requests*. A shadow request is supplementary, system-generated request that is linked to a user-generated request.

For example, when sending a message to the `impid`, it is necessary to first tell the `impid` information about the message that will be sent, such as the message size, datatype, destination rank, etc. This meta information is packaged in a "lamgiappe" header and sent to the `impid`. The real message is sent immediately following the lamgiappe. Each of these two sends generates a request; they are both marked as "mandatory" and linked together so that the overall MPI_SEND will not complete until both requests complete.

The ability to link one or more shadow requests to any user-generated request enabled many of the hidden aspects of IMPI to be performed in the MPI layer. Most of the queue manipulation code of LAM is contained in the various flavors of MPI_TEST and MPI_WAIT. Enabling shadow requests entailed rewriting much of this code.

B.1 Redirected Send Requests

In order to effect seamless communication with ranks on other IMPI hosts, it is necessary to intercept messages bound for remote ranks and redirect them to the `impid`. For example, when MPI_SEND is used to send a message to a rank on a different IMPI host, the message must be redirected to the `impid` instead. That is, when the send request is initially created, LAM determines that the message is bound for a remote IMPI proc and switches the destination to the `impid`.

Before the user message is sent, a lamgiappe header is formed and sent to the `impid`. When the `impid` receives the lamgiappe, it allocates a buffer for the incoming message and calls

MPI_RECV to receive the real message. Note that it is safe to use the blocking MPI_RECV call because the header is always immediately followed by its corresponding data message.

For non-blocking and persistent sends, all flavors of MPI_TEST and MPI_WAIT will not indicate that the request has finished until the header and user data have both been sent.

Synchronous mode sends, however, post an additional shadow request – a receive from the `impid`. MPI mandates that synchronous mode sends do not return until the destination starts to receive the message. The long message protocol in IMPI described in Sec. II-C.2 conveniently handles this case – synchronous mode messages are sent as long messages. When the synchronization ACK is returned by the remote IMPI host (step 5 in Fig. 12), the local host will send a "ping" message to the sending rank (step 6 in Fig. 12), letting it know that the synchronous send has completed.

B.2 Redirected Receive Requests

There is no way to know ahead of time whether a message received from the `impid` will be synchronous or not. Indeed, the MPI API does not provide a mechanism for determining if a received message was the result of a synchronous send. Even though the underlying implementation has this information, it is buried deep within the LAM progression engine (it is actually in lowest regions of the three c2c RPIs), and is difficult to propagate up to the MPI layer. But the MPI layer needs to know if the send was synchronous or not in order to potentially send an ACK back to the `impid` (step 4 in Fig. 12).

Since the decision to send an ACK (or not) must be made in the MPI layer in the user's program, it is not possible to piggyback the "synchronous" flag on the main message data without potentially corrupting the user data, or causing an incidental memory copy. Hence, the synchronous flag must be contained in an additional message.

One potential solution to this problem is to send a query to the `impid` asking if the received message was synchronous. However, this "active query" model would entail sending a query every time a message is received from the `impid`, this could create undue latency and bandwidth overhead.

Hence, a passive solution was used. The `impid` sends the additional ping message only if the message was synchronous. This ping will always precede the main message; the guaranteed ordering allows the MPI layer to determine if the message was synchronous. That is, a shadow receive is posted on every receive request that will receive the ping from the `impid` (step 3 in Fig. 12). If the ping request completes, the message was synchronous, and it triggers an ACK to be sent to the `impid` (which is, itself, a shadow request linked to the ping request). If the ping request does not complete by the time the main message request completes, the message was not synchronous, and the ping request is canceled.

B.3 MPI_BARRIER

IMPI mandates an MPI_BARRIER inside of MPI_FINALIZE. This entailed adding much of the infrastructure for IMPI collective algorithms to LAM. Since the IMPI collectives are implemented on top of point-to-point functionality, the majority of LAM's collective infrastructure had to do with segmenting communicators into local and remote groups. Any communicator in LAM that contains ranks on a different IMPI host now has a *shadow communicator*. The shadow communicator has a unique context ID and contains only the local LAM ranks from the real communicator.

Shadow communicators are used for the local phases of collective operations. For example, Fig. 4 shows that an MPI_BARRIER on a communicator with ranks from multiple IMPI hosts has two local phases and one global phase. With shadow communicators created and maintained elsewhere in LAM, MPI_BARRIER is implemented as:

```
// Local phase
MPI_Barrier(shadow_communicator);
// Global phase
// [...IMPI-mandated algorithm...]
// Local phase
MPI_Barrier(shadow_communicator);
```

C. Shutdown Sequence

As mentioned previously, the call to MPI_FINALIZE triggers a message to be sent to the `impid` indicating that it is shutting down. When all of the local LAM ranks have called MPI_FINALIZE, the `impid` will quit the main polling loop. The IMPI client code in the `impid` will send a message to the IMPI server indicating that it is shutting down, and then call MPI_FINALIZE itself before exiting.

D. Limitations of Implementation

The current implementation of IMPI in LAM/MPI does not allow any MPI collectives other than MPI_BARRIER. While the infrastructure for the remainder of the data-passing collectives is already in place, implementing the collective communicator constructors (e.g., MPI_COMM_SPLIT) will require significant modifications to the current design. It was realized late in the process that the `impid` contains many assumptions that the communicator being used to pass messages is MPI_COMM_WORLD. This will need to change; the `impid` will probably need to become aware of all communicators that are created, at least in the local LAM.

Canceling sent messages is a difficult problem, especially in a distributed environment. As such, LAM/MPI does not support MPI_CANCEL on send requests. Since LAM itself does not support this functionality, there was little point in including cancel support in the IMPI functionality, either.

V. Results

The LAM/MPI implementation of IMPI passes all the point-to-point tests of the NIST IMPI conformance tester. Hence, when vendors make IMPI implementations available, LAM should be able to interoperate with them.

One use for IMPI (especially while LAM/MPI is the only MPI implementation that has IMPI support) is for running parallel jobs over WAN distances. IMPI is useful in this situation because a low number of sockets are used, thereby lowering the possibility of network disruption (for example).

To measure the overhead of IMPI protocol, a standard ping-pong test was conducted between machines at the University of Notre Dame in Indiana and Lawrence Berkeley National Lab in California. A 400Mhz Intel Pentium-II machine running Linux

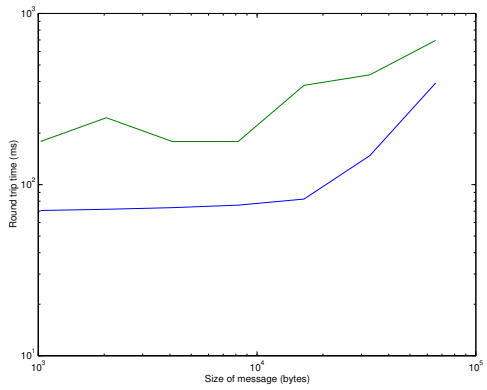

Fig. 13. Ping-pong timing results using IMPI and MPI.

2.2.12 was used at each end. Each machine was unloaded and located on a 100Mbps local switched network. The measured available bandwidth between Notre Dame and LBL at the time of the test was 2.3Mbps.

Figure 13 shows timing results comparing various sized ping-pongs between Notre Dame and LBNL using IMPI and using a single LAM spanning both hosts. As expected, the IMPI results reflect the overhead in the three-hop design of IMPI. These results should not be taken as an indictment of IMPI (or this particular implementation), however. The value of IMPI is in allowing highly-tuned implementations of MPI to work together. In some cases, having IMPI available will be the difference between being able to conduct an experiment or not. In other cases, the proper performance comparison to make will be to compare vendor-tuned implementations communicating over IMPI with a portable implementation (such as LAM) running on all hosts. To be able to conduct such experiments awaits the delivery of at least one vendor implementation of IMPI. Naturally, we hope that this particular implementation will give encouragement and impetus to the vendors to do so.

VI. Conclusions

LAM/MPI has proved that IMPI works, and several vendors on the IMPI Steering Committee have voted in favor of the proposed IMPI standard. Having IMPI-enabled vendor-tuned MPI implementations will enable not only larger, and potentially more efficient MPI jobs, it also joins MPI with the growing field of geographically distant parallel computing research – the study of linking distant resources into a single parallel resource.

More information about LAM/MPI, as well as the software package is available from `http://www.mpi.nd.edu/lam/`.

A. Future Work

LAM/MPI has some limitations in its implementation of IMPI which need to be addressed:

- Finish implementing the data-passing collectives, such as MPI_SCATTER, MPI_GATHER, etc.
- Re-design the `impid` allowing communicator constructors.
- Optimize the collectives; the pseudocode algorithms in the IMPI standard are only a first attempt, and use well-known algorithms. More research is needed in this area.
- Re-structure LAM/MPI to use generic multi-protocols; re-design the RPI to be used in a rank-pairwise manner such that multiple RPIs can be used in a single program.

VII. Acknowledgements

We would like to acknowledge the help and guidance from Raja Daoud and Nick Nevin during this project. M. D. McNally wrote the IMPI server and helped with debugging. Kinis L. Meyer contributed to the design and implementation. Finally, this paper was written while the first author was a guest at Lawrence Berkeley Lab in the Future Technologies Group, led by Bill Saphir.

References

[1] Message Passing Interface Forum, "MPI: A Message Passing Interface," in *Proc. of Supercomputing '93*, pp. 878–883, IEEE Computer Society Press, November 1993.

[2] G. Burns, R. Daoud, and J. Vaigl, "LAM: An open cluster environment for MPI," in *Proceedings of Supercomputing Symposium '94* (J. W. Ross, ed.), pp. 379–386, University of Toronto, 1994.

[3] N. E. Doss, W. Gropp, E. Lusk, and A. Skjellum, "An initial implementation of MPI," Tech. Rep. MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993.

[4] IMPI Steering Committee, "IMPI - interoperable message-passing interface," tech. rep., NIST, `http://impi.nist.gov/IMPI/`, 1999.

[5] M. McNally, J. M. Squyres, and A. Lumsdaine, *A Freeware Implementation of the IMPI Server*. University of Notre Dame, `http://www.mpi.nd.edu/research/impi/`, 1999.

[6] G. E. Fagg and J. J. Dongarra, "PVMPI: An integration of the PVM and MPI systems," tech. rep., University of Tennessee, 1994.

[7] G. E. Fagg, J. J. Dongara, and A. Geist, "Heterogeneous MPI application interoperation and process management under PVMPI," in *EuroPVM/MPI'97*, 1997.

[8] G. E. Fagg and K. S. London, "MPI inter-connection and control," Tech. Rep. 98-42, Corps of Engineers Waterways Experiment Station Major Shared Resource Center, 1998.

[9] F.-C. Cheng, P. Vaughan, D. Reese, and A. Skjellum, *The Unify System*. NSF Engineering Research Center, Mississippi State University, September 1994. Version 0.9.2.

[10] Message Passing Interface Forum, "MPI-2," July 1997. http://www.mpi-forum.org/.

[11] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir, "MPI-2: Extending the message-passing interface," in *Euro-Par '96 Parallel Processing* (L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, eds.), no. 1123 in Lecture Notes in Computer Science, pp. 128–135, Springer Verlag, 1996.

[12] G. D. Burns, "The local area multicomputer," in *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, ACM Press, March 1989.

[13] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pp. 23–32, ACM Press, May 1999.

[14] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pp. 182–189, IEEE Computer Society Press, February 1999.

[15] L. Team, "Porting the LAM 6.3 communication layer," Tech. Rep. TR 00-01, University of Notre Dame, August 1999. `http://www.mpi.nd.edu/lam/download/`.

[16] A. Stepanov, "The Standard Template Library — how do you build an algorithm that is both generic and efficient?," *Byte Magazine*, vol. 20, Oct. 1995.

[17] A. A. Stepanov and M. Lee, "The Standard Template Library," Tech. Rep. X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.