Architecture of LA-MPI, a Network-Fault-Tolerant MPI*

Rob T. Aulwes, David J. Daniel,
Nehal N. Desai, Richard L. Graham,
L. Dean Risinger, Mark A. Taylor
and Timothy S. Woodall
Los Alamos National Laboratory
Advanced Computing Laboratory
MS-B287, P. O. Box 1663
Los Alamos NM 87545, USA

Mitchel W. Sukalski
High Performance Computing
and Networking Research
Sandia National Laboratories
P. O. Box 969, Mail Stop 9915
Livermore CA 94551-0969, USA

Abstract

We discuss the unique architectural elements of the Los Alamos Message Passing Interface (LA-MPI), a high-performance, network-fault-tolerant, thread-safe MPI library. LA-MPI is designed for use on terascale clusters which are inherently unreliable due to their sheer number of system components and trade-offs between cost and performance. We examine in detail the design concepts used to implement LA-MPI. These include reliability features, such as application-level checksumming, message retransmission, and automatic message re-routing. Other key performance enhancing features, such as concurrent message routing over multiple, diverse network adapters and protocols, and communication-specific optimizations (e.g., shared memory) are examined.

1 Introduction

LA-MPI [7, 2] is an implementation of the Message Passing Interface (MPI) [14, 15] motivated by a growing need for fault tolerance at the software level in large high-performance-computing (HPC) systems.

This need arises from a growing concern with the reliability of processors, communication networks, and system infrastructure in large clusters. The manufacturing tolerances to which such components conform may be inadequate to guarantee error-free execution [18] of an application, given the length of a typical application run and the very large number of individual systems that are aggregated into a cluster. For example, a network device may have an error rate which is perfectly acceptable for a desktop system, but not in a cluster of thousands of nodes, which must run error free for many days or even months to complete a scientific calculation. Notice that the crucial quantity is the product of component count and application run time. For applications with very large run times – a data-processing application on a satellite for example – fault tolerance is a concern even for small clusters.

There have been several research efforts that incorporate network and process fault tolerance into message passing systems, each addressing the problem from a different perspective. Following Bosilca [3], we divide messaging-fault-tolerance approaches into three categories depending on where in the software stack they are implemented: the application, transport and data-link levels. Any software implementation may address fault tolerance at one or more of these levels, with the choice implicitly reflecting assumptions about the type and probability of failures.

One of the first efforts to incorporate fault tolerance into MPI was CoCheck tuMPI [22] from Technischen University Munich, which addresses fault tolerance at the application level. CoCheck used the Condor [10] library to checkpoint and then if necessary restart and roll-back the MPI job. This system's main drawback was the need to checkpoint the entire application, which could be prohibitively expensive in terms of time and scalability for large applications (like those that would run on a terascale cluster). Another effort, Starfish MPI [1], is similar in operation to CoCheck, and also operates at the application level. However, Starfish uses its own

systems to checkpoint jobs, and does not rely on a flush message protocol to handle communications. Starfish uses "atomic" group communications protocols based on the Ensemble system [6]. A third application level approach is the FT-MPI [5] effort from the University of Tennessee-Knoxville. FT-MPI handles fault tolerance at the MPI communicator level, and lets the application developer decide what course of action they wish to take. The application may decide to shrink, rebuild or abort the communicator depending on the type of fault. More recently the widely used LAM/MPI [9] implementation has released a module that implements checkpoint/restart facilities.

LA-MPI addresses fault tolerance at all levels, with our primary efforts directed at the data-link and transport levels. Currently LA-MPI (a) reliably delivers messages in the presence of I/O bus, network card and wire-transmission errors; (b) survives network card and path failures (when the operating system survives) and guarantees delivery of in-flight messages after such a failure; (c) supports the concurrent use of multiple types of network interface; and (d) implements message striping across multiple heterogeneous network interfaces, and striping of message fragments across multiple homogeneous network interfaces. In future we do intend to explore application-level fault tolerance (i.e., process fault tolerance), but believe that a high-performance, network-fault-tolerant messaging system is a necessary first step. LA-MPI gives the MPI application developer a guarantee of end-to-end network fault tolerance and provides a firm foundation for reliable applications.

This paper gives a detailed description of the architecture of LA-MPI, focusing on its fault-tolerant features. We believe high performance is a goal of equal importance to fault tolerance, so we also present some benchmark results indicating the performance impact of implementing reliable message passing.

2 Architecture

Figure 1 shows the basic architectural elements of the LA-MPI library. LA-MPI provides MPI version 1.2 and some version 2 services by layering MPI routines on a basic set of User Level Messaging (ULM) interface primitives. ULM was designed to be a high-performance, fault-tolerant, reliable messaging subsystem capable of supporting any number of MPI implementations, and of being extended to support different messaging models (e.g., put/get one-sided communications, etc.).

ULM itself consists of two layers: the Memory and Message Layer (MML) and the Send and Receive Layer (SRL). The MML provides message manage-
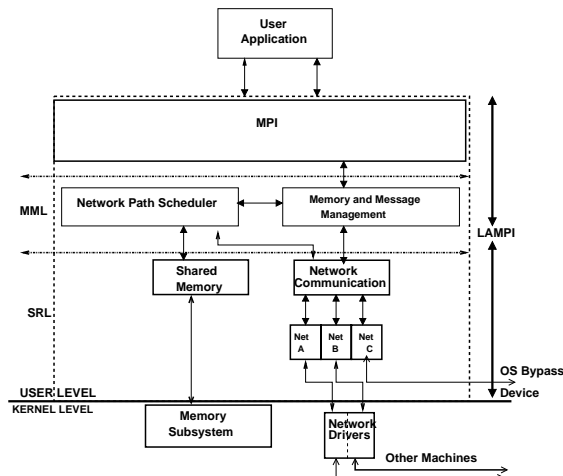

Figure 1. LA-MPI architecture.

ment services including message routing (i.e., network path selection), message tag matching, buffer allocation (for uniform and non-uniform memory access machines (NUMA)), message retransmission, and message status tracking. The SRL provides message transmission and reception over shared memory and different network adapters. Each network type (Quadrics Elan3, Myrinet GM, etc.), or path, manages its own resources, and implements its own flow control and resource exhaustion schemes. All elements of ULM are designed to be non-blocking and thread-safe in operation.

Besides the basic architectural elements of the library, LA-MPI also implements a run-time system consisting of an executable, mpirun, and the MPI library for process spawning, standard I/O handling, job control, and network topology wire-up. To start a MPI job, LA-MPI spawns the user executable as a single process on each machine, specified either explicitly or through a resource reservation system such as Platform's LSF [20]. This single process upon calling MPI_Init() participates in local network resource discovery, which is shared globally as needed. After this first phase of network wire-up, this process forks itself to create all of the desired MPI processes. As a performance enhancement, forking is handled by default as a tree of processes. A second post-fork phase of network wire-up is then initiated and performed for those paths that require unique information from each MPI process (e.g., unique UDP ports for each MPI process). At the end of this second phase, all processes synchronize through a global barrier. On architectures in which LA-MPI handles standard I/O redirection and/or job control itself, the original process never exits MPI_Init(), and instead dae-

monizes itself to handle I/O redirection, signal handling, and job termination.

2.1 MPI

By layering MPI on ULM, most MPI routines can be implemented in three phases: argument checking, dispatch to the appropriate ULM routines, and error code and status translation, if necessary. A function dispatch table provides a means of selecting different implementations of collective operations at run-time, to provide optimal performance in different environments. This basic layering approach allows modularity, thereby facilitating code maintenance, while minimizing the impact upon performance-critical services. Compile- and run-time flags control whether MPI parameters are validated before use, and the extra performance gained can be balanced against the users' confidence in the correctness of their MPI-dependent application code.

MPI communicators and groups are shared between the MPI and ULM layers. They have been augmented to provide functionality not strictly required by MPI semantics, but needed for internal operation and performance optimizations (e.g., precomputed and cached information to reduce the execution time of repeated code). For example, the group object has extra maps, such as a mapping of process rank to local process rank (a unique rank between 0 and n-1 for n local processes on a given machine) for faster access into local arrays of lists. Also, precomputed trees are associated with each group to enhance the performance of certain collectives (i.e. gather/scatter operations between multiple machines). ulm_get_info() and a number of other ancillary functions are used by the MPI layer to access information in these ULM objects.

MPI data types are stored as a "flattened" array of (offset, size, sequential_offset) structures. Contiguous offsets are merged into a single entry, and a data type with contiguous, in-order offsets will be represented by a single structure. This flattened representation is highly efficient for data packing and unpacking. It is also relatively memory-efficient for many typical data types, but it does not support a single MPI job over machines with different byte ordering (i.e., a mixture of big-endian and little-endian machines), which is rare in most cluster environments.

Constructor information from MPI type calls, such as MPI_Type_struct(), is saved for MPI_Type_get_envelope() and MPI_Type_get_contents() (MPI version 2) functionality. This information is traversed recursively as needed for MPI_Get_elements(). Reference counting is implemented for all data types to avoid the
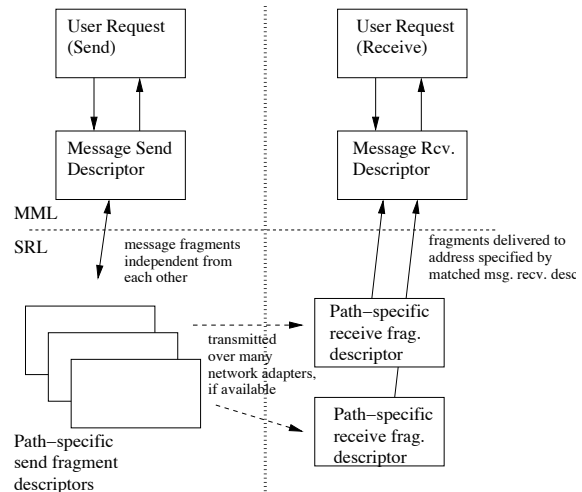

Figure 2. Descriptor relationships.

unnecessary overhead of creating duplicates of existing data types for MPI_Type_get_contents().

2.2 Memory and message layer

The Memory and Message Layer (MML) is the common code between MPI functionality and network transport-specific code (i.e., the SRL). It consists of a set of common abstractions to ensure reliable message delivery and matching while minimizing the overhead of memory management of LA-MPI data structures and buffers. Opaque MPI message requests are implemented as pointers to MML message descriptors, which are used to track sends and receives (figure 2).

The message descriptors contain MPI send and receive parameters such as the message tag, source or destination process, and communicator. The descriptors also maintain status information and a flag that indicates whether the operation has completed. In addition, the send descriptor contains all of the necessary information to send a message reliably, including lists of message-fragment descriptors (see section below). The receive descriptor is used for matching message fragments with a particular message, and also keeps track of the number of bytes received and discarded due to a too small receive buffer.

LA-MPI transmits messages in fragments whose size is determined by the underlying SRL network transport (or path, as described later in section 2.2.2). Each network transport handles its own flow control to prevent resource exhaustion, and minimizes the memory footprint of the library as a whole. Message fragmentation also allows LA-MPI to transmit a message simultane-

ously over different physical interfaces for a given network transport. For example, on LANL machines made up of multiple SGI Origin 2000 SMPs interconnected by a switched network with 2 to 13 HIPPI-800 adapters each, LA-MPI can effectively transfer an MPI message over all 2 to 13 adapters simultaneously for greater bandwidth.

Fragments are tracked by a set of message-fragment descriptors which are associated with a message descriptor. Each message send descriptor manages two lists of fragment descriptors: `FragsToSend` for fragments whose resources still need further allocation, and `FragsToAck` for sent fragments that are awaiting acknowledgment of successful delivery from a peer process. The message send descriptor itself is stored on one of two similarly paired lists: an incomplete list for send descriptors that still need further processing for all fragments to be sent, and an unacknowledged list for send descriptors awaiting peer acknowledgment.

Message progress from incomplete to unacknowledged for fragments and all message descriptors is made via calls to `ulm_make_progress()`. These calls are embedded in MPI operations such as wait and test, and other potentially blocking calls such as `MPI_Barrier()`. Currently, LA-MPI does not use a separate thread for making progress asynchronously, but uses polling in various library entry points. The indiscriminate use of threads can lead to an oversubscription of processors, which creates resource contention. Polling prevents this cache and context switching contention with the application's main code, but does require special design considerations to handle the uncertainties of MPI scheduling. In particular, the retransmission scheme must be designed not to overrun receiver resources simply because the remote process is occupied with non-MPI processing.

The MML message and message-fragment descriptors are maintained in arrays of free lists. Each free list may be constructed with memory policies to allocate machine memory close to a given process rank's processor and its local memory. This architectural consideration is extremely important for good performance on large NUMA machines, such as SGI's Origin 2000 and 3000 machines. Memory allocation for long-lived objects is managed with sub-allocators that subdivide large chunks of memory, allocated via `malloc()` or `mmap()`, into fixed elements whose addresses are quickly allocated and released by the use of a simple stack [12]. If the memory is private to a process, then new elements can be created by allocating another chunk of memory. If the memory is anonymous shared memory (allocated from a common ancestor and hence growable only with local process synchronization), then re-

source exhaustion is handled with simple retry logic until a maximum number of retries has been reached. Experience on a number of platforms has shown that the sub-allocator approach is significantly faster than a general-purpose allocator, such as `malloc()`. The same strategy is used for SRL objects (see section 2.3).

In the case of MPI persistent send requests, several message descriptors can be generated due to MPI's local send completion semantics (which are necessary for low latency). These must be managed concurrently by the MML.

2.2.1 Fragment retransmission and check-summing

Unlike many MPI libraries that consider all underlying communication perfectly reliable, LA-MPI optionally supports sender-side retransmission of messages by assigning each fragment a timeout for acknowledgment and by periodically checking a list of unacknowledged fragments for message send descriptors that have exceeded their timeout. This retransmission scheme is appropriate for low-error-rate environments, typical of most clusters. Each network transport is responsible for arranging to retransmit the necessary fragments. Each fragment's retransmission time is calculated using a truncated exponential back-off scheme; this avoids resource exhaustion at a receiving process that is busy doing non-MPI computation. Fragments that must be retransmitted are moved from the `FragsToAck` list to the `FragsToSend` list, and the associated message send descriptor is placed on the incomplete list.

Each network transport is also responsible for providing a main-memory-to-main-memory 32-bit additive checksum or 32-bit cyclic redundancy code (CRC), if it is needed. This checksum/CRC protects against network and I/O bus corruption, and is generated at the same time data is copied, if at all possible. By delaying checksumming to avoid wasting memory bandwidth, a received fragment is not necessarily a deliverable, or uncorrupted, fragment.

Several MML generic features aid in the implementation of this retransmission and checksumming scheme. Every fragment is assigned a range from a sequence of monotonically increasing 64-bit numbers (sequence numbers) for each (sender, receiver) pair of processes. These sequence numbers are recorded by the receiving process in a special object, `SeqTrackingList`, as an ordered set of non-contiguous ranges of sequence numbers; these lists use internal hint pointers to exploit any temporal locality in accessing these lists to minimize access overhead. The receiver maintains two `SeqTrackingList` lists for each peer with which it communicates to distinguish between fragments that
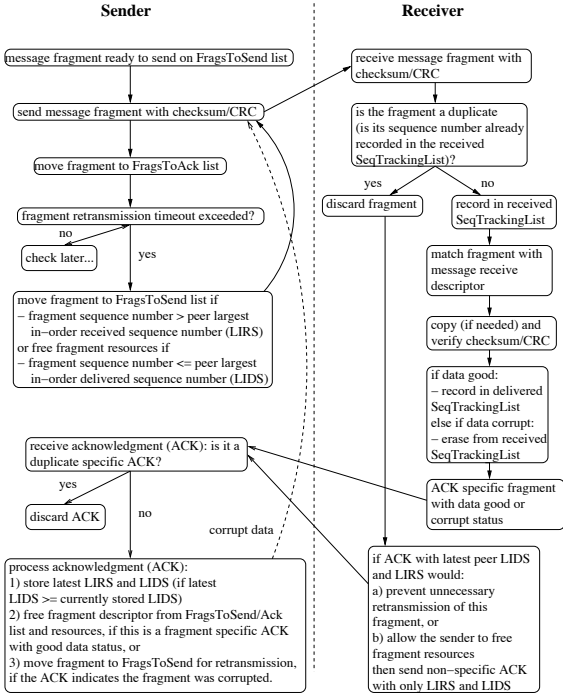
Sender   **Receiver**

message fragment ready to send on FragsToSend list

receive message fragment with checksum/CRC

send message fragment with checksum/CRC

is the fragment a duplicate (is its sequence number already recorded in the received SeqTrackingList)?

move fragment to FragsToAck list

yes → discard fragment

no → record in received SeqTrackingList

fragment retransmission timeout exceeded?

no → check later...

yes

match fragment with message receive descriptor

move fragment to FragsToSend list if
– fragment sequence number > peer largest in–order received sequence number (LIRS)
or free fragment resources if
– fragment sequence number <= peer largest in–order delivered sequence number (LIDS)

copy (if needed) and verify checksum/CRC

if data good:
– record in delivered SeqTrackingList
else if data corrupt:
– erase from received SeqTrackingList

receive acknowledgment (ACK): is it a duplicate specific ACK?

yes → discard ACK

no

corrupt data

ACK specific fragment with data good or corrupt status

process acknowledgment (ACK):
1) store latest LIRS and LIDS (if latest LIDS >= currently stored LIDS)
2) free fragment descriptor from FragsToSend/Ack list and resources, if this is a fragment specific ACK with good data status, or
3) move fragment to FragsToSend for retransmission, if the ACK indicates the fragment was corrupted.

if ACK with latest peer LIDS and LIRS would:
a) prevent unnecessary retransmission of this fragment, or
b) allow the sender to free fragment resources
then send non–specific ACK with only LIRS and LIDS

Figure 3. Retransmission and check-summing.

have been received, and those that have been received and delivered successfully (i.e., no data corruption). Duplicate fragments are easily detected by checking the received fragment's sequence number against the received `SeqTrackingList`.

Upon processing fragment acknowledgments from a receiver, a sender will store two special values that are carried in every acknowledgment: the largest in-order peer received sequence number (LIRS), and the largest in-order peer delivered sequence number (LIDS). The LIRS is used to increase the timeout between the retransmission of fragments that have been received, but whose data integrity has not been checked yet; it may increase or decrease over time, depending upon transmission and I/O bus errors. The LIDS is used to free any fragments whose acknowledgment was lost. The LIDS is always less than or equal to the LIRS. Figure 3 shows the interaction of these sequence numbers, the retransmission scheme, and checksumming.

While this approach exhibits many similarities to the internet's transmission control protocol (TCP/IP), there are several key differences. TCP provides end-to-end reliability and flow control at the network level via a sliding window protocol that limits the number of unacknowledged segments or fragments that are in-flight in the network to the window size advertised by the receiver. Unacknowledged segments are retransmitted by the sender on a timeout computed based on an estimate of round-trip time. In the event of network congestion, the sender must decrease the window size and increase retransmission timeouts to avoid flooding the network with retransmissions. The reduced window size and delay in acknowledgments throttle the application reducing its effective bandwidth. As congestion in the network improves, the sender will incrementally adjust these parameters until the full bandwidth is restored or additional segments are lost, at which point the cycle is repeated.

In contrast, LA-MPI can bypass much of the complexity of the TCP approach due to the nature of the targeted clusters. Main-memory-to-main-memory reliability is provided via the MML, which optionally handles checksumming and retransmission on a per-fragment basis. Flow control is delegated to the lower-level SRL, which employs path-specific features (e.g. Quadrics events, shared memory) to provide maximum utilization of each respective network path without overloading the receiver. The relatively low error rate of the targeted clusters and the ability to separate acknowledgments from flow control allows the retransmission timers to be configured at relatively long intervals without adversely affecting the network bandwidth. The overall approach both improves the reliability and performance over TCP while providing comparable performance to other MPI implementations which do not provide end-to-end reliability.

2.2.2 Network path interactions

The network path object is an abstraction of lower-level network transports and devices that are available to LA-MPI. Each path can represent a single network adapter, or a set of common adapters, or even a common protocol over many different network adapters. LA-MPI currently implements paths for the user datagram protocol [21] (UDP over all IP-enabled devices), shared memory, Myrinet GM [17], Quadrics Elan3 [19] remote direct memory access (RDMA), and HIPPI-800 operating system bypass (SGI IRIX only). Paths for Myrinet MX and Infiniband [8] (Mellanox HCA Verbs [13]) are in development. In all of our current paths except UDP/IP, which treats multiple network adapters as a single Internet Protocol "device", multiple network adapters are used by a single path instantiation, if they exist on the machine. By fragmenting messages and sending different fragments over different network adapters, LA-MPI takes advantage of all available bandwidth.

Each path provides a common set of services to the library (via virtual methods):

- to store/retrieve a handle to a path: `bindToContainer()`, `getHandle()`, and `getContainer()`;

- to control a path's status and query remote process reachability: `isActive()`, `activate()`, `deactivate()`, and `canReach()`;

- to store/retrieve information about a path: `getInfo()`, and `setInfo()`;

- to control the ownership of a MPI message: `bind()`, and `unbind()`;

- to initialize and send/resend a bound message: `init()`, `send()`, `retransmitP()`, and `resend()`;

- to query a message's send status: `sendDone()`;

- to receive message fragments from a path: `receive()`; and

- to check whether control messages (non-MPI message data) need sending and to send those control messages: `needsPush()`, and `push()`.

Each MPI communicator contains a bind method that associates a message to a particular path. This function can be manipulated on a per-communicator basis via a get/set ULM interface. The default function implements a static hierarchy of network paths based on the nominal bandwidth; if there is only one path available, the function returns immediately with that one.

At startup, each path does basic initialization, and registers with a global `pathContainer`. During registration the `pathContainer` queries the path to determine which processes it can reach via that path. One method, `paths()`, is used to find all of the currently active paths to a given process, and another, `allPaths()`, is used by `ulm_make_progress()` to poll all active paths for received data.

Figure 4 illustrates the interactions between send messages and paths. Using these abstractions, LA-MPI has the necessary infrastructure to support automatic network fail-over (currently in development) in the face of general and specific network failures, including network adapter, switch, and link failures. Each path sets its own policy for determining when a path failure has occurred, and signals that a message needs to be sent over another path (i.e., rebound to another path) by returning a path failure error code. If a path manages multiple network adapters, and another managed adapter can reach the receiving process, then the path is responsible for silently retransmitting the affected traffic over that adapter without signaling failure.
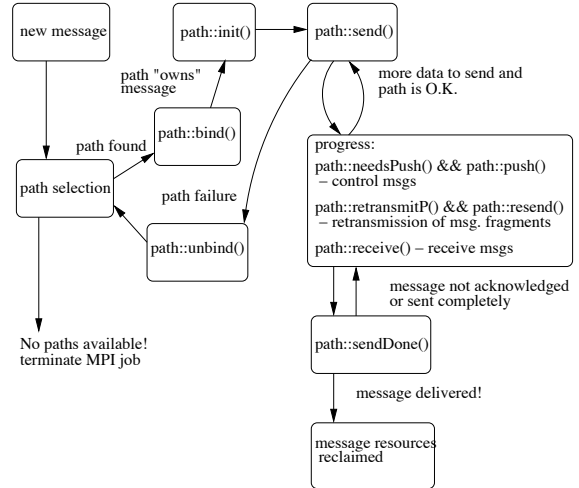

Figure 4. Message-path interactions.

2.2.3 Message tag matching

Because LA-MPI supports the concurrent use of multiple networks, MPI message-tag matching is implemented independently of the underlying networks. In order to preserve MPI semantics, messages between any two processes must be matched by the receiver in the order that they are posted by the sender. LA-MPI accomplishes this with a message sequence number that is sent with all fragments of a message. This sequence number increases monotonically, and is unique between a given sending and receiving process. Since fragments can be received out of order over different network adapters, tag matching can be accomplished with any of a message's fragments. Each receiving process tracks the next expected sequence number for a given sending process. While this sequence number prevents matching messages out of order from the sending process' perspective, message receive descriptors are kept in a list in the order they are posted for faster processing.

When a message fragment is received, its message sequence number is compared to the next expected sequence number. If a fragment's message sequence number is greater than the expected sequence number, then the fragment has been received ahead of sequence and is stored on a special array of lists that are indexed by the sending process' rank.

If a fragment's message sequence number is equal to the expected sequence number, then this is the first fragment received for this message, and the fragment is matched against the next appropriate message receive descriptor. If a match is not made, then the fragment is stored on a special ok-to-match array of lists that is indexed by the sending process' rank. If a match is made,
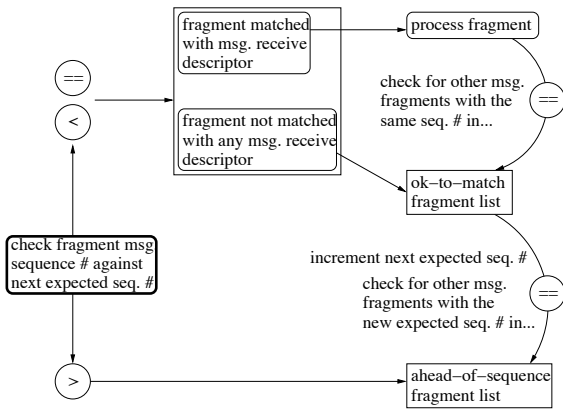
Figure 5. Message tag matching.

then the fragment is processed (copied, checksum/CRC verified, and an ACK generated, as needed), and any fragments of this same message on the ok-to-match list are also processed. The matched message receive descriptor is moved to a special list for matched descriptors. In either case, the receiver then increments the next expected sequence number, and checks the ahead-of-sequence fragment list for a possible match to the new expected sequence number.

Finally, if a fragment's message sequence number is less than the next expected sequence number, then the fragment must belong to a message whose first fragment has already arrived. This fragment is processed in almost the same way as the sequence equality case. If a match was already made by checking the special matched descriptor list, then the fragment is processed. If a match has not been made, then the fragment is appended to the ok-to-match list, along with the first fragment and any other fragments that have previously arrived. Figure 5 illustrates the entire matching process.

In multi-threaded operation LA-MPI uses an array of receive locks, indexed by the sending process rank, to prevent race conditions from the simultaneous posting of message receive descriptors (e.g., MPI_Irecv()), and processing of received fragments. In processing non-wild-card receive requests, only threads trying to receive messages from the same sending process must contend for the same locks. Wild-card requests, however, require the acquisition of all receive locks; the resultant performance degradation is a design trade-off against the performance gained for non-wild-card requests.

2.3 Send and receive layer

The Send and Receive Layer (SRL) consists of multiple network path implementations including a highly optimized shared memory communication path. Each path is independent and can be optionally disabled at compile- or run-time.

2.3.1 Shared memory communication

LA-MPI's shared memory path uses shared memory for the message send descriptor, the message fragment descriptors and data. This means that the receiving process receives 32- or 64-bit pointers to these control structures, and does not incur the overhead of allocating and managing receive fragments. The only cost of data transmission is remote access to another processor's local memory, which can be considerable on large NUMA machines, especially with cache coherency. However, LA-MPI manages the control structures to ensure that on NUMA machines they are allocated close to the sending process' processor. This ensures that at least one of the processes has minimum latency to the shared memory.

One of the key shared memory path structures is a two-dimensional array of first-in-first-out (FIFO) lists indexed by the sending and receiving processes' ranks. This array of FIFOs is constructed to use memory near the sending process, and is used to transmit the 32- or 64-bit addresses of fragment descriptors to a receiving process. Since each FIFO has only one writer, namely the sending process, cache invalidation thrashing caused by multiple processors writing to the same memory is avoided. This procedure is lock-free (for single-threaded applications) and gives very good performance. Since the FIFOs are of fixed size, overflow can occur and is handled using an ordered linked list.

The first fragment descriptor of a message (SMPFragDesc_t) differs from all other fragment descriptors (SMPSecondFragDesc_t) in that it contains all of the information needed to match the message to a posted receive descriptor at the receiving process. Once the first fragment is matched, further matching is avoided by the receiving process setting a pointer to its message receive descriptor in the first fragment descriptor and the message send descriptor. Any following fragments can then be placed directly on the receiver's SMPMatchedFrags list. If a match has yet to be made, they are placed on a fragsReadyToSend list in the message send descriptor so that they can be processed later by the receiver.

Since all messages need at least one fragment descriptor, allocation costs are minimized by creating message send descriptors and first fragment descriptors as adjacent, paired objects. They are then allocated as a single object. As a further optimization, the first fragment descriptor can be created, and its address transmitted via the fixed FIFO (and an overflow list) to a receiver before the sender actually copies message data into shared

7

memory. The receiver's cost of matching the first fragment to a message receive descriptor can be hidden by the time required for the sender to copy data into shared memory. This is a performance gain for all messages greater than zero bytes in length.

Similar shared memory communication methods have been implemented the MP_Lite message-passing library [16].

2.3.2 Network communication

Each network path has its own design considerations balancing performance, scalability, and memory usage. In this section we briefly discuss interesting design points for different network transports and devices.

LA-MPI's user datagram protocol (UDP/IP) path implementation uses two UDP sockets per MPI process. To support full connectivity between N processes, LA-MPI needs 2N UDP connectionless sockets, as opposed to N(N-1)/2 connection-oriented sockets. This scalability becomes very important in terascale distributed computing environments with thousands of MPI processes, and was a major factor in choosing UDP/IP over connection-oriented TCP/IP.

The UDP/IP implementation uses its two sockets for different types of traffic. One (the short message socket) is used for eager sending of small messages, the first fragment of a multi-fragment message, and control messages such as fragment acknowledgments. The other socket (the long message socket) is used for fragments of multi-fragment messages that have already been matched. A multi-fragment message's first fragment is sent to the receiving process' short message socket. When it is matched and processed, an acknowledgment is generated and sent back to the sending process. Upon receiving the acknowledgment, the sender then sends the remaining fragments to the receiver's long message socket. This protocol ensures that any traffic on the long message socket can be processed immediately without additional user-level buffering that must be maintained for an indefinite period of time.

All UDP/IP message fragments have a fixed MPI header that is peeked at first (i.e., copied but not deleted from kernel buffers), so that if the fragment's contents represent contiguous MPI data types, then the data can be placed from kernel memory to the proper user address via one `recvmsg()` call.

LA-MPI's Quadrics Elan3 RDMA implementation uses a radically different approach from the UDP/IP implementation. The Elan3 library provides facilities to create chained DMAs (batches of DMAs performed sequentially by the Elan3 processor), and support for two types of RDMA writes into a remote process' 32-bit address space: a queue DMA that delivers a payload up to
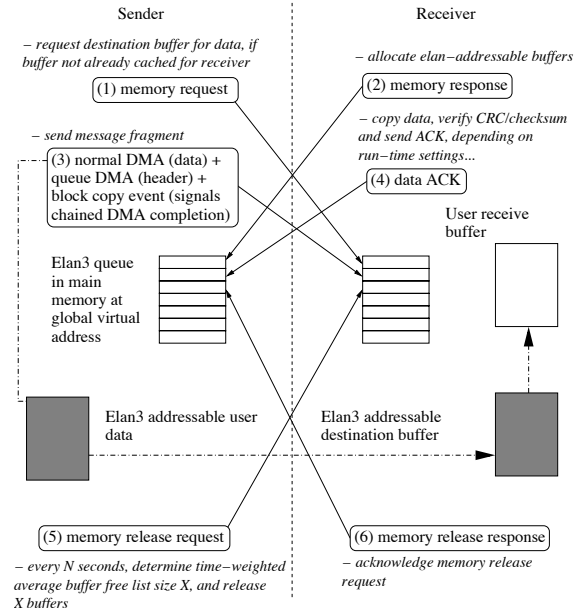

Figure 6. Quadrics Elan3 send and receive implementation.

320 bytes to a remote queue whose access is coordinated between the network adapter and the main processors; and a normal DMA that delivers an arbitrary payload to a particular virtual memory address of a remote process. A block copy event copies the contents of an Elan adapter memory region to another Elan-addressable region, usually in main memory; by chaining a block copy event at the end of a set of RDMA writes, a local process can be notified that all of the previous RDMA operations have completed. This is extremely useful, if the ACK protocol has been disabled by run-time settings, or if the process is trying to detect path failure.

LA-MPI utilizes both of these types of DMAs by sending fragments' header information (with 52 bytes of immediate data payload) and all control messages via queue DMAs with a queue element size of 128 bytes, and by sending larger fragments via normal DMAs to Elan3 addressable memory that has been allocated using a memory request/response protocol. Fragments with 53 bytes of data or more are sent using chained DMAs, with the data sent first using a normal DMA and the header information sent next using a queue DMA. A special copy event that copies a known value into an event block in main memory terminates the chain, and is used to signal the completion of the chained DMA locally.

Each process allocates a single queue per network adapter, or rail in Quadrics parlance, at the same virtual memory address. By sending all control informa-

tion to one of these queues, LA-MPI minimizes polling overhead. The Elan3 adapter only supports access into a 32-bit virtual memory address space; on 64-bit architectures the Elan3 adapter can only access a static portion of the entire process address space. Since not all memory is guaranteed to be addressable, a special set of control messages is used between a (sender, receiver) pair of processes to request elan-addressable memory buffers. The sender caches these addresses for later use, and uses them in a last-in-first-out (LIFO) fashion to reduce the chances of a page fault. The sender maintains a time-weighted average free list size of these cached addresses, and lazily releases the average number of buffers on the free list to the receiver process every couple of minutes. Figure 6 illustrates the basic send and receive mechanisms.

3 Performance

In this section we present benchmark results that characterize the performance of LA-MPI on a variety of computer architectures.

In table 1 we give "ping-pong" performance results from two systems currently of interest to us: (a) an alpha/Tru64 system consisting of HP/Compaq ES45 4-way nodes with 1.25 GHz alpha ev68 processors, and a "dual rail" Quadrics Elan/Elite interconnect; and (b) an i686/Linux system composed of 2 GHz dual Xeon nodes with 2 Myrinet 2000 cards.

Also included in table 1 are results for LA-MPI's UDP/IP path run over Elan/IP, and Myrinet GM/IP. These numbers give an indication of the very large cost associated with a complete IP implementation, and why LA-MPI uses the light-weight checksum/retransmission protocol described in section 2.2.1. Note that over ethernet LA-MPI's UDP/IP path gives very similar performance to the TCP/IP performance of other MPI implementations.

For the Quadrics path on alpha we quote the results with and without data integrity guaranteed. As can be seen the impact of reliability on performance is relatively small, increasing latency by about a third and reducing bandwidth by less than 10 %.

LA-MPI excels in the bandwidth benchmarks, for two reasons. Firstly, on-host traffic is handled by the shared memory path which has a much higher bandwidth than the Elan devices. Secondly, on systems with multiple network devices per node (for example, two "rails" of Quadrics Elan/Elite network), LA-MPI highly efficiently sends message fragments along both rails. This differs from some other message-passing libraries which use multiple network devices by allocating a device to all communication between a particular process pair, a strategy that works well for random communication patterns but not for pairwise exchange patterns.

4 Summary and future work

With the rise of terascale distributed computing environments consisting of thousands of processors and network adapters, the need for fault tolerant software has become critical to their successful use. Negligible component error and failure rates in small to medium size clusters are no longer negligible in these large clusters, due to their complexity, sheer number of components, and amount of data transferred. LA-MPI addresses the network-related challenges of this environment by providing a production-quality, reliable, high-performance Message Passing Interface (MPI) library for applications capable of (a) surviving network and I/O bus data corruption and loss, and (b) surviving network hardware and software failure if other connectivity is available. In this paper, we have presented an overview of LA-MPI's design and implementation.

LA-MPI is currently available as open source software under an LGPL license. It currently runs on Linux (i686 and Alpha processors), HP's Tru64 (Alpha only), SGI's IRIX 6.5 (MIPS), Apple's Mac OS X (PowerPC), and Microsofts Windows (using Cygwin). It supports shared memory, UDP/IP, Quadrics Elan3 RDMA, HIPPI-800 OS bypass (IRIX only), and Myrinet GM paths, and work is progressing on Myrinet MX and Infiniband (Mellanox HCA Verbs) communications paths. LA-MPI supports job spawning and control with Platform LSF, Quadrics RMS, Bproc [4], and standard BSD `rsh`. Please send email to `lampi-support@lanl.gov` for more information [11].

Future development will address

- the implementation of a fault-tolerant, scalable, administrative network for job control, standard I/O redirection, and MPI wire-up;

- the implementation of process fault-tolerance in the face of multiple process failures;

- one-sided communication;

- collective communication optimizations;

- the implementation of dynamic topology reconfiguration and addition of MPI processes to support dynamic process migration and MPI-2 dynamic processes.

System	Path	Latency (μs)	Bandwidth (MB/s)
alpha	Shared Memory	2.93	935
alpha	Quadrics/Elan (1 NIC)	11.23 (8.39)	257 (273)
alpha	Quadrics/Elan (2 NICs)	11.37 (8.43)	438 (468)
alpha	Quadrics/Elan (UDP/IP)	156	67
i686	UDP/IP gigE	125.1	91
i686	Shared Memory	3.09	455
i686	Myrinet/GM (1 NIC)	11.91	241
i686	Myrinet/GM (2 NICs)	12.26	403
i686	Myrinet/GM (UDP/IP)	125.2	94
i686	UDP/IP gigE	125.1	91

Table 1. Zero-byte latency and peak point-to-point bandwidth for various LA-MPI paths. For the Quadrics path, we also give (in parentheses) the performance numbers with reliability (guaranteed data integrity) turned off (a run-time option).

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, 1999.

[2] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, and M. W. Sukalski. LA-MPI: The design and implementation of a network-fault-tolerant MPI for terascale clusters. Technical Report LA-UR-03-0939, Los Alamos National Laboratory, 2003.

[3] G. Bosilca, A. Bouteiller, F. Cappello, S. Djailali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes.

[4] Cluster Research Team, Advanced Computing Laboratory, Los Alamos National Laboratory. *http://public.lanl.gov/cluster/index.html*.

[5] G. Fagg and a Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *EuroPVM/ MPI User's Group Meeting 2000, Springer-Verlag, Berlin, Germany, 2000*, 2000.

[6] G. E. Fagg, K. Moore, and J. J. Dongarra. Scalable Networked Information Processing Environment (SNIPE). *Future Generation Computer Systems*, 15(5–6):595–605, 1999.

[7] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. In *Proceedings of the 16th international conference on Supercomputing*, pages 77–83. ACM Press, 2002.

[8] Infiniband Trade Association. *http://www.infinibandta.org/home*.

[9] LAM/MPI, Progressive Technology Labs, Indiana University. *http://www.lam-mpi.org/*.

[10] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *8th International Conference on Distributed Computing System*, pages 108–111. IEEE Computer Society Press, 1988.

[11] Los Alamos Message Passing Interface, Advanced Computing Laboratory, Los Alamos National Laboratory. *http://www.ccs.lanl.gov/ccs1/projects/la-mpi*.

[12] M. McKusick and M. Karels. Design of a general purpose memory allocator for the 4.3 BSD Unix kernel. In *Proceedings of the Summer 1988 USENIX Conference*, pages 295–303. USENIX Association, 1998.

[13] Mellanox Technologies, Inc. *http://www.mellanox.com/*.

[14] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. Technical report, 1994.

[15] Message Passing Interface Forum. MPI-2.0: Extensions to the Message-Passing Interface. Technical report, 1997.

[16] MP_Lite, Ames Laboratory. *http://cmp.ameslab.gov/cmp/ercap.old/MP_Lite.html*.

[17] Myricom, Inc. *http://www.myri.com/*.

[18] C. Partridge, J. Hughes, and J. Stone. Performance of checksums and CRCs over real data. *Computer Communication Review*, v. 25 n. 4:68–76, 1995.

[19] F. Petrini, W.-C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, v. 22 n. 1:46–57, 2002.

[20] Platform Computing, Inc. *http://www.platform.com/*.

[21] J. Postel. RFC 768: User Datagram Protocol, Aug. 1980.

[22] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.