An Event-driven Architecture for MPI Libraries

Supratik Majumder and Scott Rixner
Rice University
Houston, TX 77005
{supratik,rixner}@rice.edu

Vijay S. Pai
Purdue University
West Lafayette, IN 47907
vpai@purdue.edu

Abstract

Existing MPI libraries couple the progress of message transmission or reception with library invocations by the user application. Such coupling allows for simplicity of implementation, but may increase communication latency and waste CPU resources. This paper proposes the addition of an event-driven communication thread to make messaging progress in the library separately from the application thread, thus decoupling communication progress from library invocations by the application. The asynchronous event-thread allows messages to be sent and received concurrently with application execution. This technique dramatically improves the responsiveness of the library to network communication. Microbenchmark results show that the time spent waiting for non-blocking receives to complete can be significantly reduced or even eliminated entirely. Application performance as measured by the NAS benchmarks shows an average of 4.5% performance improvement, with a peak improvement of 9.2%.

1 Introduction

The Message Passing Interface (MPI) standard includes primitives for both blocking and non-blocking communication [10]. The latter are designed to enable overlap between an application's communication and computation, with the initiation of a message send or receive followed by independent computation. However, almost all existing MPI libraries (and all freely-available ones) only make progress on the transmission or reception of messages when the application calls into the library. Thus, if a message could not be sent to or received from another node at the time of the first non-blocking send or receive call, the actual communication could be arbitrarily delayed waiting for the application to re-enter the library through an MPI call. At such points, a progress engine attempts to make as much progress on each pending message as possible, but no communication progress takes place between library invocations.

Such MPI implementations lead to several problems. First, communication performance is tied to the rate at which an application makes MPI library calls. If the application calls the library too frequently, the progress engine wastes resources determining that there are no pending messages. If the application calls the library too rarely, communication latency increases, as no progress is made in between calls. Furthermore, such library implementations technically violate the MPI standard which stipulates that communication progress must be made even if no other calls are made by the sender or receiver to complete the send or receive, respectively[1].

This paper proposes the addition of an event-driven communication thread to actually make messaging progress in an MPI library, independent of whether or not the application layer enters the library again after the initial send or receive call. An asynchronous event thread allows messages to be sent and received concurrently with application execution. This can improve the responsiveness of an MPI library and improve application performance by decreasing message latency and increasing message bandwidth. An implementation of the event-driven communication model on top of Los Alamos MPI (LA-MPI) using TCP-based communication shows the value of this technique. In addition to providing low cost, portability, and well-studied reliability mechanisms, TCP is also tied in with efficient event notification mechanisms in standard operating systems (such as `select` and `poll` in all UNIX implementations, `epoll` in Linux, and `kqueue` in FreeBSD). This event-driven approach to MPI communication enables up to 9.2% perfor-

This work is supported in part by a donation from AMD and by the Department of Energy under Contract Nos. 03891-001-99-4G, 74837-001-03 49 and/or 86192-001-04 49 from Los Alamos National Laboratory.

[1] The MPI 1.1 progress rule specifically states: "A call to `MPI_WAIT` that completes a receive will eventually terminate and return if a matching send has been started, unless the send is satisfied by another receive. In particular, if the matching send is non-blocking, then the receive should complete even if no call is executed by the sender to complete the send. Similarly, a call to `MPI_WAIT` that completes a send will eventually return if a matching receive has been started, unless the receive is satisfied by another send, and even if no call is executed to complete the receive." [10]

mance improvement on 5 NAS benchmarks tested, with an average of 4.5% performance improvement.

This event-driven approach is an efficient technique for handling asynchronous I/O, such as network communication, and is commonly used in the network server domain [5, 21]. Network servers must efficiently handle a large number of incoming requests and outgoing responses to maximize the number of simultaneous clients they can support. To do so, they only execute event handlers in response to network events—available buffering for outgoing connections or available data on incoming connections. This approach enables network servers to handle large amounts of network traffic and, as this paper shows, can also apply to MPI libraries.

The rest of this paper proceeds as follows. Section 2 presents the architecture of the LA-MPI library, a representative MPI library that can use TCP as a message layer. Section 3 then shows how LA-MPI was modified to use an asynchronous event-driven communication thread and Section 4 presents the performance improvements achieved by these modifications. Section 5 then discusses related work and Section 6 concludes the paper.

2 Architecture of Los Alamos MPI Library

The Los Alamos MPI (LA-MPI) library is a high-performance, end-to-end, failure-tolerant MPI library developed at the Los Alamos National Laboratory [15]. The LA-MPI version 1.4.5 library implementation consists of three almost independent layers: the *MPI Interface Layer*, the *Memory and Message Layer* (MML), and the *Send and Receive Layer* (SRL). The interface layer provides the API for the MPI standard version 1.2 specification. The MML provides memory management, storage of message status information and support for concurrent and heterogeneous network interfaces. Finally, the SRL interfaces with the specific network hardware in the system and is responsible for sending and receiving messages over the network. LA-MPI supports a range of systems with varied interconnect hardware using independent SRL modules for each type of network.

All communication within LA-MPI is controlled by the progress engine, which is the mechanism responsible for making progress on pending requests in the library. The progress engine mainly belongs to the MML with helper routines in the SRL. The MML maintains a list of all pending messages—incomplete sends and receives—currently active in the library. Conceptually, the progress engine loops over all of these incomplete requests and attempts to send or receive as much of each message as possible. In order to make progress on each request, the engine invokes the appropriate send or receive routine within the SRL to access the operating system and/or the networking hardware. Most calls into the interface layer of the library also invoke the progress engine. On a blocking MPI call, the progress engine continues to loop through all pending requests until the specified request is completed and the blocking call can return. In contrast, on a non-blocking MPI call, the progress engine just loops through all of the pending requests once and then returns regardless of the state of any of the requests.

2.1 The TCP Path

Within the SRL, there are several different *path* implementations, corresponding to the supported network types. The TCP path supports TCP message communication over Ethernet using the socket interface to the operating system's network protocol stack. The TCP path utilizes events to manage communication. An event is any change in the state of the TCP sockets in use by the library. There are three types of events: read events, write events, and exception events. A read event occurs when there is incoming data on a socket, regardless of whether it is a connection request or a message itself. A write event occurs when there is space available in a socket for additional data to be sent. An exception event occurs when an exception condition occurs on the socket. Currently, the only exception event supported by the socket interface to the TCP/IP protocol stack is the notification of out-of-band data. The current LA-MPI library does not use any out-of-band communication, so exception events are only included to support possible future extensions.

The TCP path, like other paths supported by LA-MPI, provides its own progress routine which gets invoked by the library's main progress engine. The TCP path maintains three separate lists, one for each type of event, to keep track of all the events of interest. At various points of execution, the TCP path pushes these events of interest on to the respective event lists. In addition, callback routines are registered for each event. The progress mechanism of the TCP path uses these lists to inform the operating system that it would like to be notified when these events occur. The `select` system call is used for this purpose. `Select` takes three event arrays (read, write, and exception) and a timeout period as arguments. If any of the events of interest occurs, `select` returns with three new arrays of events of interest. If no events have occurred, `select` blocks until either an event of interest occurs or the timeout period is exceeded. In LA-MPI, these calls to `select` are non-blocking and return immediately. If any events of interest have occurred, they are each handled in turn by the appropriate callback routines and finally control is returned to the progress engine. The progress engine attempts to make progress on all active paths in the library in turn, and thus could invoke the TCP progress routine several times in the process of satisfying a blocking MPI request.

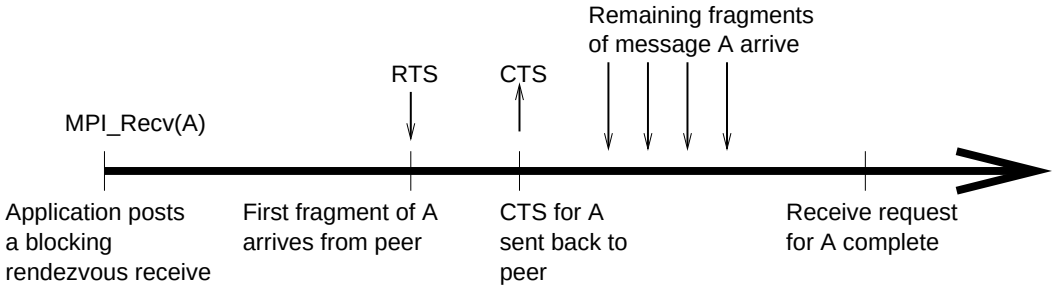

Figure 1. The *rendezvous* message transfer protocol at a LA-MPI receiver node

For communication in the TCP path, the library uses at least one bidirectional TCP connection between each pair of communicating nodes. These connections are not established during initialization. Rather, they are established only when there is an actual send request to a node that does not already have an open connection to the sending node. Once a connection is established, it remains open for future communication, unless an error on the socket causes the operating system to destroy it. In that case, a new connection would be established for future messages between those nodes.

2.2 Sending Messages

The TCP path starts the send process on a message by first fragmenting it—the first fragment is 16 KB in size, and all subsequent fragments are 64 KB. Each message fragment includes a header to allow the receiving node to identify the fragment. The fragments, along with their headers, are placed on a send queue within the library prior to being sent over the network. A write event for the appropriate connection is then added to the write event list to indicate that there is data in the send queue waiting to be sent. The MPI specification does not strictly require fragmentation, but it helps to limit the necessary buffering on the receiving node. If the message is small enough ($<$ 16 KB), it is sent as a single fragment directly to the destination. This is the *eager* protocol, as the sender does not request permission from the receiver to send the message. Larger messages are sent using the *rendezvous* protocol which involves a request-to-send (RTS) and clear-to-send (CTS) message exchange between the peers before the sender can send the whole message to the receiver. Figure 1 shows the steps of this *rendezvous* protocol at the receiver node. The sender initiates this protocol by sending the first fragment of a multi-fragment message, which also serves as the RTS message, to the receiver. As depicted in the figure, the receiver receives this message and if the corresponding MPI receive has already been posted, sends a CTS message back to the sender. The sender, on receiving the CTS message, sends the remaining fragments to the receiver to complete the send operation. On the receiver node, the reception of these fragments completes the receive operation.

When the TCP progress routine processes the write event (because there is space available in the socket buffer for writing), the first fragment is sent from the send queue and the write event is removed from the event list. This first fragment serves as a RTS message to the receiver. The sender then adds a read event for this connection to wait for the CTS message, which the receiver sends when it is ready to receive the rest of the message. Note that the use of RTS/CTS is not strictly necessary in TCP for flow control, as the socket buffer provides implicit flow control. If the receiver is not ready to receive the message, the receiver's socket buffer will fill up, and TCP flow control will stop the sender from sending further data. However, TCP does not allow the receiver to reorder messages within the same connection, so the library relies on RTS/CTS messages to achieve this. Note that TCP does allow the receiver to reorder messages from different senders trivially since they are using different connections. Once the *clear-to-send* has been received, the write event is placed back on the write event list until all fragments in the send queue have been sent. Depending on the size of the message and the size of the socket buffer, this could require numerous invocations of the appropriate callback routine.

The TCP path marks a send request as complete when all the fragments of that send message have been enqueued on the socket buffer. Since TCP guarantees delivery of the message to the receiver, the TCP path of LA-MPI itself does not provide any additional reliability features, instead relying purely on the reliability mechanisms provided by the TCP protocol. Completion of a send request signals the library to release all of the resources held for this message and to notify the application appropriately.

2.3 Receiving Messages

Message receives also use network events, as in the send case. However, there is an additional complication in that received messages can either be expected or unexpected, since the application may or may not have notified the library that it is waiting for a particular message (*posted* a

receive) before that message arrives. To handle this, the MML maintains two independent queues, the *Unexpected Receives Queue* (URQ) and the *Posted Receives Queue* (PRQ).

The receive handler of the TCP path is invoked through the progress engine of the library whenever there is incoming data on any of the established connections. All established connections are always active on the read event queue since messages may arrive at any time. The receive handler first reads the header of the incoming fragment from the socket buffer. Then, the messages on the PRQ are checked to find a match for the incoming fragment. If a match is found, the receive routine directly copies the data from the socket buffer into the user buffer posted by the application.

If the receive has not yet been posted, buffer space for the unexpected fragment is allocated and the fragment is added to the URQ. Subsequent receives posted by the application are first matched against fragments in the URQ before being posted on the PRQ. The library buffer used to allocate unexpected receives is freed when a match is found in the URQ and the data from the buffer has been copied into the user buffer. At that point, the fragment is also removed from the URQ.

For RTS messages, the receiver does not send a CTS to the sender until the corresponding receive has been posted by the application. So, if a match for an incoming fragment is found on the PRQ, the CTS is sent back immediately. Otherwise, the CTS is delayed until the application posts a receive that matches in the URQ. This ensures that at most one fragment of any unexpected message gets buffered in the library at a node. Again, this is only necessary to allow the receiver to easily reorder messages from a single sender, since TCP already provides flow control for unread messages. The receive message operation is completed when all the fragments of a message have been received and copied into the user buffer.

3 Event-driven MPI Communication

The LA-MPI library implementation executes the progress engine every time the application calls into the library, and only when the application calls into the library. Therefore, the execution of the progress engine is not related to the occurrence of network events. This can be wasteful if library calls occur more often than network events. In that case, the entire progress engine must execute despite the fact that there is nothing for it to do. The use of callback routines mitigates this cost to some extent, as these handlers will only be called when events do occur. However, every time the progress engine is executed, it incurs the overhead of calling into the operating system to check for events. On the other hand, since the progress engine is only invoked when the application calls into the library, messages cannot be sent or received until that time. This requires a delicate balancing act for the application programmer. If the library is called too frequently, resources will be wasted, and if the library is called too rarely, then messages will not be transferred promptly leading to unnecessary latency increases. From the programmer's perspective, the application should only be required to access the library when messaging functions are required, and not to allow the library to make progress on in-flight messages. Furthermore, the MPI standard specifies that the library should make progress on all pending messages (send/receive) regardless of whether the application is executing library code or not.

Since network events occur asynchronously to application library calls, it does not make sense to handle these events only during library invocations. This paper proposes a novel technique to handle network events within an MPI library autonomously—an event-driven progress engine. An event-driven progress engine decouples execution of the progress engine from library invocation and thus is able to handle asynchronous network events more efficiently. Such an event-driven progress engine also satisfies the MPI standard's stipulation that the MPI library should be able to progress pending requests at all times, regardless of whether or not the application is executing library code.

3.1 Event-driven Architecture

An event-driven software architecture consists of event-handling routines which only execute in response to events. Such an event-driven application, in its simplest form, consists of one tight loop, known as the event loop. The event loop's main function is to detect events and dispatch the appropriate event handler when they occur. In the absence of an event, the event loop simply blocks waiting for one. In that case, the occurrence of an event wakes up the event loop, which then dispatches the appropriate event handler.

For example, a web server is an inherently event-driven application. A web server's main function is to accept and respond to client requests, which arrive over the network. So, a web server only executes code when network events occur—either the arrival of a request or the availability of resources to send responses. In modern web servers, the event loop utilizes one of the operating system event-notification facilities, such as `select`, `kqueue`, or `epoll`, to detect network events [12, 19]. These facilities are centered around a system call that returns a list of pending events. If desired, the system call can block until such an event occurs. So, the event loop of a web server makes one of these blocking system calls to wait for an event. As soon as a client request arrives at the web server or resources free up to send a response to a previous request (both of which cause a network event), the operating system wakes up the event loop by returning from the system call. The event loop then dispatches the appropriate handler to respond to the event. When the event handler completes, it

returns control to the event loop, which either dispatches the next event handler if there were multiple events pending, or it blocks waiting for the next event.

An event-driven software architecture provides an efficient mechanism to deal with the latency and asynchrony inherent in network communication. This design eliminates the need for the application to query repeatedly about the occurrence of an event, making it possible for the application to instead accomplish other useful tasks while it waits for the event to occur. An added advantage is the scalability it offers in terms of the number of network events it can simultaneously monitor without undue performance degradation.

The LA-MPI library uses the notion of read and write events to wait for the availability of resources to send and receive message fragments, respectively. However, the occurrence of these events does not automatically trigger the library to send or receive message fragments. Instead, the library checks for these events only through its progress engine. Even after an interesting event has occurred, the library might not execute the handler until the next invocation of the library by the application, and subsequent execution of the progress engine. Thus, the LA-MPI library architecture is not strictly event-driven, since the tasks of sending or receiving fragments are not performed directly in response to the corresponding events.

3.2 An Event-driven MPI Library

An MPI application performs two main tasks: computation and communication. These tasks place different demands on the system, as computation is driven by the control flow of the program and communication is driven by asynchronous network events. An MPI library provides such applications with two important services. First, the functional interface of the library provides the application a mechanism to transfer messages among nodes. Second, the progress engine of the library performs the communication tasks that actually transfer those messages. Almost all existing MPI libraries combine these tasks by executing the progress engine only when the application invokes the functional interface of the library. This favors the computation portion of the application, as communication progress is only made when the application explicitly yields control to the MPI library, rather than when network events actually occur. This is a reasonable trade-off as it would be difficult to efficiently handle the computation portion of the application in an event-driven manner. However, the communication tasks of the MPI library are quite similar to the tasks of a web server and match the event-driven architecture model quite well.

This paper presents LA-MPI-ED, an event-driven version of the TCP path of LA-MPI. LA-MPI-ED separates the computation and communication tasks of an MPI application. This enables the communication portion of the library to use the event-driven model and the functional interface of the library to work synchronously with the computation portion of the application. The two tasks of the library occur in separate threads. The main thread (MT) executes the application and the functional interface of the library, as normal. The event thread (ET) executes the communication tasks of the library in an event-driven fashion. When sending messages, the main thread notifies the event thread that there is a new message to be sent, and the event thread sends it. When receiving messages, the event thread accepts the message and then notifies the main thread that it has arrived when the application invokes the library to receive the message. In this manner, the most efficient software model can be used for both components of the MPI application. This also facilitates greater concurrency between executing the MPI application and executing the progress engine of the MPI library.

The MT provides the synchronous aspects of an MPI library and executes the core of the library, including the application linked against the library. Thus, the MT is responsible for performing library tasks such as initialization, keeping track of pending messages, posting messages to be sent or received, etc. The event-driven progress engine of the library executes in the context of the ET and thus, can run concurrently and independent of the MT. The ET, by virtue of executing the progress engine, is responsible for handling all the asynchronous network events which affect the state of the library. The MT spawns the ET during library initialization and both threads exist until the end of execution of the MPI application. The ET monitors all connected sockets for incoming data (read events) and the sockets over which messages are being sent for available buffer space (write events) and is thus ultimately responsible for the transmission and reception of messages whenever resources become available. The threads communicate with each other through shared queues of data structures. Using light-weight threads, such as POSIX threads (pthreads), minimizes the overhead of thread switching and protected accesses to the shared data. The programming effort of sharing data between the two threads is also minimized since pthreads execute in the same application address space by design. The event-driven library can also support multi-threaded MPI applications with one event thread catering to all the threads of the application. This paper discusses the support for multi-threaded applications in the library in greater detail in Section 3.4.

Conceptually, the MT runs continuously until the application makes a blocking MPI function call like *MPI_Send*, *MPI_Recv* or *MPI_Wait*. A blocking MPI call signifies that the application needs to wait for the library to accomplish a certain task before it proceeds. In this case, the MT blocks on a condition variable using a pthread library call until

the pending request has been completed. The ET then gets scheduled by the thread scheduler for execution and wakes up the MT upon completion of the request. In contrast, the ET runs only when there is a network event of interest to the library. In the absence of an interesting event the ET is kept blocked on the `select` system call. The occurrence of an event causes the operating system to wake up the ET, which then invokes the appropriate event handler to process pending messages. In practice, if both threads have work to be done, the thread scheduler must allow them to share the system's resources. In that case, the responsiveness of the ET can be increased by running it at a higher priority than the MT.

The TCP path of LA-MPI already utilizes three event lists—read, write and except—to form the components of a blocking `select` call to retrieve events from the operating system. Thus, the notion of events in the TCP path complements the event-driven architecture of the progress engine very nicely. The event-driven approach in the design of the progress engine also provides an efficient means to exploit even more the event-centric design of the TCP path. Separating the handling of network events into the ET incurs minimal alterations to the structure and functionality of the LA-MPI TCP path. It still performs the tasks such as connection establishment, message transmission and message reception as before. The only difference with the event-driven progress engine is that execution of these tasks is now split between the MT and the ET, based on whether the task is performed synchronously by the library or happens as a result of an asynchronous network event. The event-driven nature of the progress engine significantly alters only the way the library interacts with the network. Thus, instead of having the progress engine poll for events every time it is invoked, the progress engine waits for the occurrence of an event and processes it immediately.

LA-MPI-ED, like LA-MPI, utilizes events to send or receive message fragments. However, as explained in Section 3.1, LA-MPI is not event-driven since the execution of the library is governed by the MPI application, and not by the occurrence of these events. On the other hand, in LA-MPI-ED, all message communication in the library is performed by the ET in direct (and immediate) response to such events. This makes communication in LA-MPI-ED truly event-driven in nature. The computation tasks of the application are still performed synchronously, and independent to communication, by the MT of the library.

The design of LA-MPI-ED relies heavily on the notion of events in the TCP path. Since the other paths within LA-MPI do not utilize network events, they do not easily fit into the event model. Messages transferred using other paths in the library currently revert to using the MT for all tasks, including communication. To enable specialized networking hardware, such as Quadrics or Myrinet, to utilize the event-driven model, those paths within LA-MPI, possibly including device firmware, would need to be rewritten to use network events for communication. While this is certainly possible and would likely improve communication performance, it is beyond the scope of this paper.

3.3 Example

Figure 2 illustrates the operational differences between the two library versions—LA-MPI and LA-MPI-ED. Subfigure (i) shows a simple MPI request sequence executing at a particular node. Subfigure (ii) shows the progression of this MPI request sequence in both versions of the library. For ease of understanding, the progression is shown on 3 different timelines. The topmost timeline shows the sequence of steps performed by the MPI application and events that are external to the library (such as arrival of the RTS message). The middle and the bottom timeline show the steps as they occur in LA-MPI and LA-MPI-ED respectively. As shown in the topmost timeline, the MPI application posts a *rendezvous* receive request for message A at time-step a. Then the application performs some computation tasks before invoking the MPI wait call at time-step c to wait for the arrival of message A. At some time-step b, between a and c, the node receives the RTS message from the peer node in the form of the first fragment of the posted receive.

As mentioned in Section 3.2, the event-driven progress engine only alters the way in which the library interacts with and responds to network events. The behavior of LA-MPI and LA-MPI-ED differs when the RTS message arrives from the peer node. The LA-MPI library takes no action in response to the RTS message arriving at the node until time-step d ($= c$), when the application makes the MPI_Wait call. At that point, the RTS is received and the CTS message sent to the peer node. After this, the library spins in the progress engine as it waits for the rest of the fragments of message A to arrive. The entire message is received at this node by time-step e. This completes the receive request along with the wait call and control is returned to the application.

With an event-driven asynchronous progress engine, the LA-MPI-ED library starts to process the RTS message immediately upon receiving it and sends the CTS back to the peer node at time-step f ($< c, d$). Note that at this time, the application (and MT) is still busy with its computation tasks, but the ET is free to handle incoming data. This enables the ET to perform communication concurrently with the computation being performed by the MT. Finally, at time-step g, the receive of message A is completed and the ET goes back to waiting for the next network event. The MT meanwhile finishes its computation and makes the wait call at time-step h ($= c, d; > g$). Since by this time the receive of message A is already complete, the wait call returns back to the application immediately. As a consequence, there is

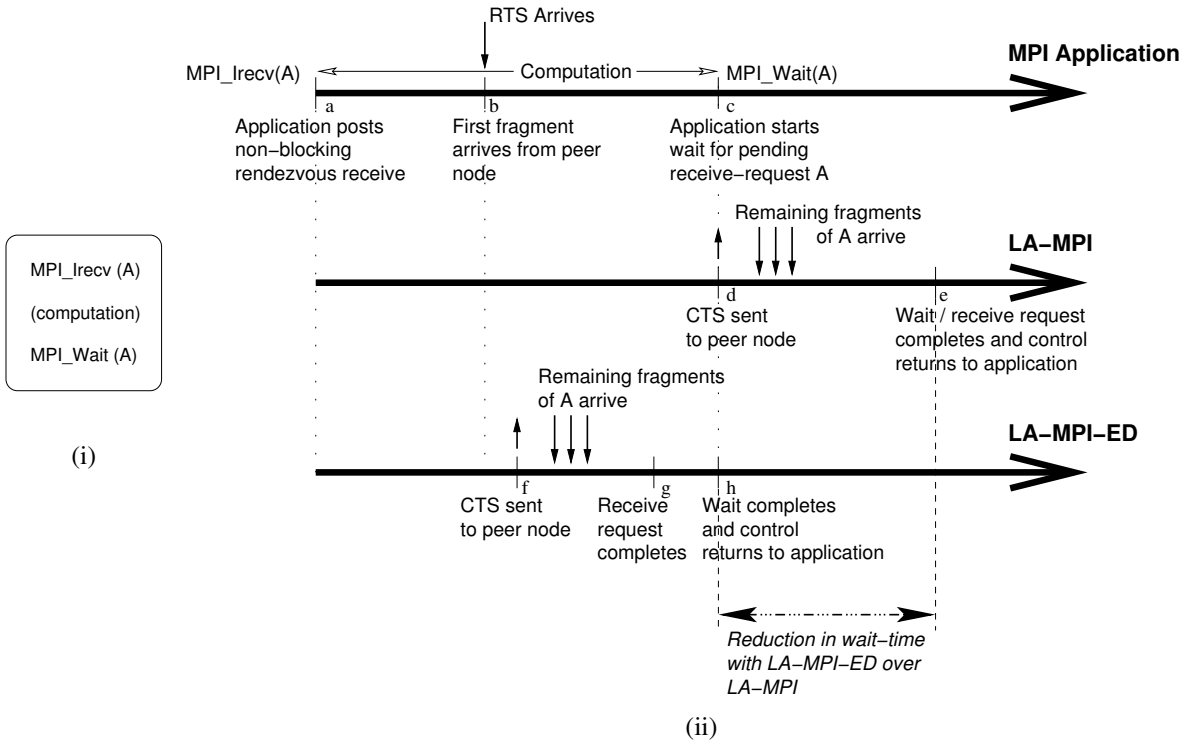

Figure 2. A simple MPI request sequence and its progression in LA-MPI and LA-MPI-ED libraries

a significant reduction in effective wait-time with the LA-MPI-ED library as pointed out in the figure.

Even though the example only illustrates the operational distinctions between the two versions of the library in performing a non-blocking receive, non-blocking sends also exhibit almost identical behavior. With the event-driven progress engine, the ET can continue sending fragments of a message without requiring the completion of the computation tasks of the MPI application.

The example presented in Figure 2 is relatively simple and is only intended to convey a hint about the behavior of an event-driven asynchronous progress engine. In practice, in LA-MPI-ED when computation and communication proceed concurrently, the net time spent for the computation task would increase because the execution of the progress engine would take away compute cycles from the application. As a result, the application would make the wait call later, so time-step *h* would shift further right along the timeline. However, as long as the overlap of computation and communication more than offsets the increase in effective computation time, this technique provides a performance benefit over the original LA-MPI library.

3.4 Performance Issues of LA-MPI-ED

The event-driven LA-MPI library effectively satisfies the objective of separating the synchronous and asynchronous aspects of the library and handling them independently. An event-driven progress engine is much more responsive to network events and can much more efficiently transmit and receive messages. As an added advantage the progress engine thread, which runs independent of and concurrent to the core of the library, is now able to process pending requests even in the absence of library invocations by the application.

The thread library does impose a limited amount of overhead for thread switching and thread management tasks. However, for a well designed MPI application, the improvement in performance due to the concurrency of computation and communication should offset this overhead. It also relieves the programmer from having to worry about enabling the library to make progress on outstanding messages.

MPI applications can also be very sensitive to message latency, especially for the really short messages which are frequently used to implement communication barriers. The extra latency introduced by thread switching on the send path can thus potentially hamper application performance. The event-driven LA-MPI library tries to mitigate this shortcoming of its threaded design by a simple optimization—the MT, after posting a message to be send, optimistically tries to send it, as well. If the socket buffer has enough space for the message, the message can be sent immediately without the use of the progress engine. This ensures that whenever there is available space on the socket buffer, which will be the case for most single fragment messages, the message

will be sent directly without incurring any thread switching overhead. The event thread will be used to send messages only when there is not enough space in the socket buffer to send the first fragment immediately, or for subsequent fragments of a multi-fragment message. This optimization effectively makes the send side performance of the event-driven and non-event-driven libraries equivalent.

The addition of an asynchronous event-management thread in the event-driven LA-MPI library does not alter the original library's support for multi-threaded MPI applications. Since different threads of a multi-threaded MPI application can be executing the progress engine simultaneously, the correct handling of message communication by the library requires special consideration. The original LA-MPI library ensures correctness by allowing access to message queues only one thread at a time. Thus, multiple threads executing the progress engine iterate through the lists of pending requests one after the other, which ensures their consistency across the different threads. Furthermore, progress on any particular message request is not tied to the thread that initiated this request. Hence, all pending requests in the library are progressed whenever any application thread invokes the progress engine. If there are multiple threads waiting for the completion of different requests simultaneously, the completion of any particular request only causes the corresponding thread to return to the application, while the other threads continue to wait in the progress engine. The event-driven LA-MPI library provides identical behavior for threaded MPI applications by having one event thread to progress requests initiated by all application threads. The only case which requires special consideration is when multiple application threads are sleeping during blocking MPI calls. Since the event thread does not know in advance which request a particular thread is waiting on, all sleeping threads are woken up by the event thread on completion of a request. Each thread re-checks the status of the request immediately after being woken up, and thus, only the thread whose request was completed resumes execution; the remaining threads return to sleep until the event thread awakens them the next time, repeating the process.

In addition to improving the efficiency and responsiveness of communication, an independent messaging thread can also improve the functionality of the MPI library. The event thread could also be used for such tasks as run-time performance monitoring, statistics gathering, and improved run-time library services. Furthermore, in the event of a dropped TCP connection, the ET could also reestablish the connection faster and further improve messaging performance.

4 Evaluation

This section presents an evaluation of the LA-MPI-ED library and compares it against the current version of the LA-MPI library. The evaluation is performed using both microbenchmarks and application benchmarks. The microbenchmarks measure the ping latency, the one-way communication bandwidth of the MPI library, and the ability of the MPI library to overlap communication with computation, all on Gigabit Ethernet. The application benchmarks are from the NAS set of parallel benchmarks and reflect real-world MPI computations [1].

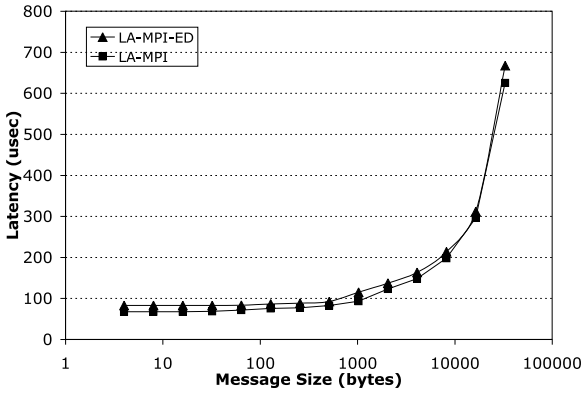

Figure 3. Comparision of ping-latency between LA-MPI and LA-MPI-ED

The evaluation is performed on a FreeBSD workstation cluster of up to 9 processing nodes. Each workstation has one AMD Athlon XP 2800+ processor, 1GB DDR SDRAM, and a 64bit/66Mhz PCI bus. Each of the nodes also has at least 20GB of hard drive capacity (none of the benchmarks are disk intensive). Each workstation also has one Intel Pro/1000 Gigabit Ethernet network adapter. The nodes are connected to each other through one 24-port Gigabit Ethernet switch. Each node runs an unmodified FreeBSD-4.7 operating system with various socket and networking parameters tuned in order to provide maximum TCP/IP performance. LA-MPI-ED uses the default POSIX thread library implementation in FreeBSD (`libc_r`).

4.1 Microbenchmark Performance

The ping latency of messages between two communicating MPI nodes is measured by a simple benchmark involving 2 nodes—node 0 sends a message of fixed size to node 1 and then receives the same message back from node 1. The ping latency between these two nodes is then half the total time elapsed in this two-way message transfer. The benchmark only uses the default blocking versions of sends and receives for all MPI communication. Figure 3 shows the ping latency for 4 B–32 KB messages, for both LA-MPI-ED and LA-MPI libraries. The figure shows that the ping latency of messages with LA-MPI-ED is always slightly higher than with LA-MPI. The difference in ping latencies between the two libraries is nearly constant at 15 μsec for

all message sizes between 4-byte to 16 KB; this difference stems from thread-switching overheads in LA-MPI-ED. Although this is a 23% increase for 4-byte message latency, it is only 5% for 16 KB messages. Beyond 16 KB, the ping latency for both library versions more than doubles as the messaging protocol shifts from *eager* to *rendezvous* (Section 2.2) and the number of message fragments increases from one to two. Again, because of thread-switching overhead on the receive path, the ping latency in LA-MPI-ED is slightly higher than in LA-MPI (approximately 42 μsec or 6% difference).

A separate microbenchmark is used to measure the unidirectional bandwidth of the MPI library using blocking sends and receives. This benchmark consists of one node sending a fixed size message to a receiver a given number of times; the receiver simply receives the messages. The sender and receiver use the default blocking versions of MPI send and receive routines, respectively. The bandwidth is measured by dividing the total number of bytes sent by the time elapsed to complete a given number of messages. Figure 4 shows the message bandwidths obtained on this benchmark by the LA-MPI and LA-MPI-ED libraries for message sizes ranging from 1 KB to 2 MB. Each data point is obtained by iterating over 1000 messages. LA-MPI always outperforms LA-MPI-ED, with performance gaps up to 22% at 1 KB messages and 19% at 32 KB messages (when the rendezvous protocol is first invoked), but with smaller gaps for larger messages within each sending protocol. Besides the thread library overhead, LA-MPI-ED suffers from a greater number of unexpected message receives, each of which results in extra memory copies. The LA-MPI library, which executes its progress engine synchronously with calls into the library, first posts the message receive and then reads the incoming message by invoking the progress engine. Even if the message arrives earlier, the library would only read the message from the socket at this time, and hence encounters all expected receives, which are read directly into the posted buffer. On the other hand, the LA-MPI-ED library receives a message as soon as it arrives up on the socket and thus invariably ends up receiving the message into a library buffer before the actual application buffer has been posted. The simplicity of the benchmark causes this drop in performance as a side effect to the increased responsiveness of LA-MPI-ED. Under the rendezvous protocol, only the first fragment of any message is received unexpected. Thus, the performance difference between LA-MPI and LA-MPI-ED drops with message size and is only 2% for 2 MB messages.

Figure 5 shows the CPU utilization, of both the sender and receiver, for the unidirectional bandwidth test with the LA-MPI and LA-MPI-ED libraries, respectively. The base LA-MPI library performs blocking MPI library calls by constantly spinning in the progress engine, repeatedly iterating over all pending requests and trying to make progress on all active paths in the library. This results in 100% CPU utilization with any MPI application as the progress engine remains active whenever the library (or application) is not performing other tasks. However, for comparing the effective CPU utilization (time spent by CPU doing useful work) of LA-MPI and LA-MPI-ED, the base library is modified suitably to block on the `select` system call while it waits for the completion of an incomplete request. Note that in general such a modification compromises library correctness, since it assumes that there is only one pending request active in the library at any given time. However, for the simple bandwidth benchmark employed here, this assumption is valid, and does not affect library functionality in any way. Figure 5 shows that the sending side, with either version of the library, has almost equal CPU utilization across the range of message sizes shown. At 32 KB message size, the sender's CPU utilization with the LA-MPI-ED library is about 4% below that of the LA-MPI library. At this message size the *rendezvous* protocol comes into effect. The receiver receives the first fragment as an unexpected fragment and delays the transmission of the CTS until the corresponding receive is posted. This causes lower CPU utilization on the sending side with the event-driven library as the library waits for the CTS message before proceeding with the remaining fragments. With increasing message sizes, the relative fraction of this additional waiting-time diminishes, resulting in converging CPU utilizations of the sender with either versions of the library. For the receiver, at small message sizes the thread context-switch overhead is substantial and results in considerably higher CPU utilization with the event-driven library. However, the effect of this additional overhead is largely marginalized with the *rendezvous* protocol, and consequently for message sizes of 32 KB and beyond the CPU utilization plots of the two libraries are almost coincident.

Figures 6 and 8 show the same bandwidth comparison between LA-MPI and LA-MPI-ED for the cases where all receives are expected and unexpected, respectively. Both of these graphs show that the performance of the two library versions is almost matched except at small message sizes. When message size is small, the thread library overhead is a higher percentage of the total communication time and causes a 8–10% drop in bandwidth performance with LA-MPI-ED. A comparison of the graphs also reveals that unexpected receives can cause up to 20% drop in library bandwidth as compared to expected receives. Figure 7 and 9 show the comparison of CPU utilization between LA-MPI and LA-MPI-ED, on both the sender and the receiver, for the above two variants of the basic unidirectional bandwidth test. These figures also show that the thread library overhead manifests itself most clearly for small sized messages at the receiving node. With increasing message sizes, the

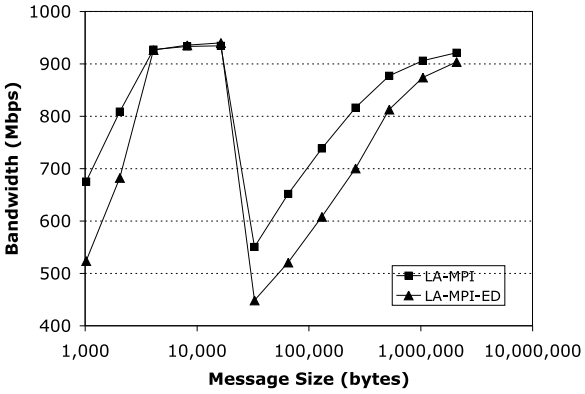

Figure 4. Comparision of unidirectional message bandwidth between LA-MPI and LA-MPI-ED

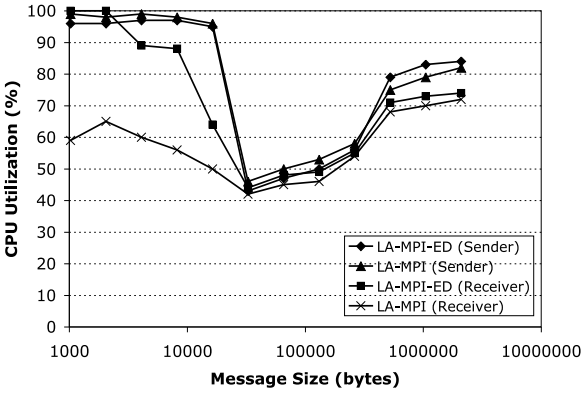

Figure 5. Comparision of CPU utilization for unidirectional bandwidth test between LA-MPI and LA-MPI-ED

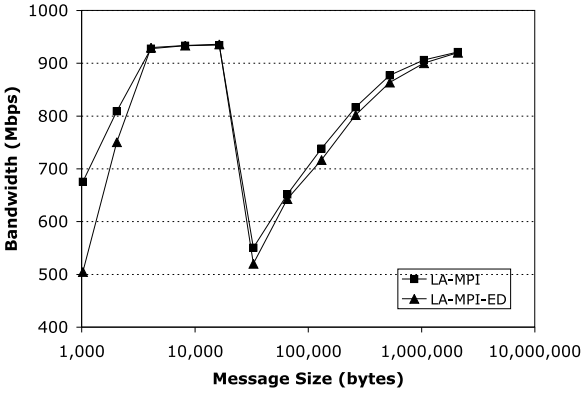

Figure 6. Comparision of unidirectional message bandwidth between LA-MPI and LA-MPI-ED with all expected messages at the receiver

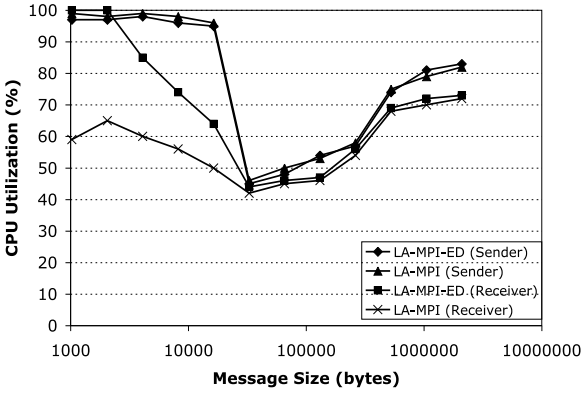

Figure 7. Comparision of CPU utilization for unidirectional bandwidth test (all expected receives) between LA-MPI and LA-MPI-ED

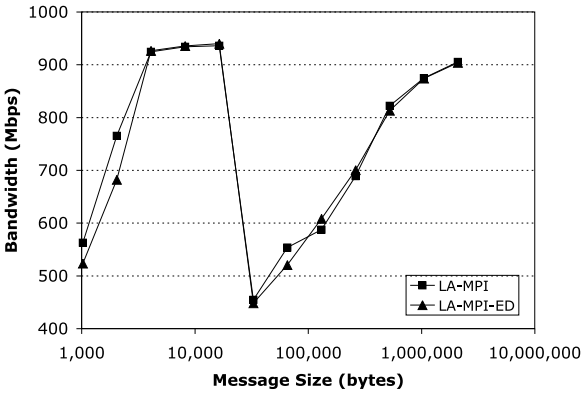

Figure 8. Comparision of unidirectional message bandwidth between LA-MPI and LA-MPI-ED with all unexpected messages at the receiver

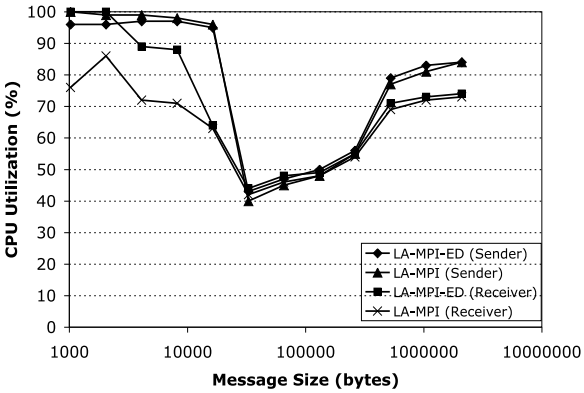

Figure 9. Comparision of CPU utilization for unidirectional bandwidth test (all unexpected receives) between LA-MPI and LA-MPI-ED

```
Sender
i = 0;
while (i++ < NUM_TESTS)
{
    MPI_Send(A);

    MPI_Recv(B);
}
```

```
Receiver
i = 0;
while (i++ < NUM_TESTS)
{
    MPI_Irecv(A);

    /* computation */

    MPI_Wait(A);
    MPI_Send(B);
}
```

Figure 10. Pseudo Code for the microbenchmark used to evaluate the advantages of LA-MPI-ED over LA-MPI

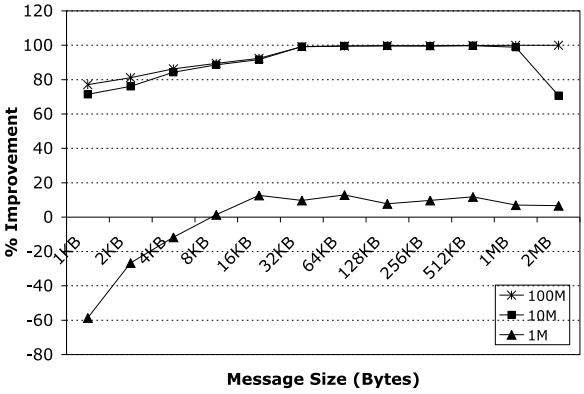

Figure 11. Percentage improvement in wait-time of the microbenchmark with the event-driven LA-MPI library

impact of this overhead on CPU utilization is increasingly reduced, and LA-MPI and LA-MPI-ED achieve almost the same CPU utilizations.

One of the main advantages of the event-driven LA-MPI library is its ability to overlap computation and communication, both of which are essential to real MPI applications. However, this advantage cannot be seen in microbenchmarks that only perform communication. Figure 10 presents pseudo code for a microbenchmark that highlights the benefits of LA-MPI-ED by carefully balancing computation and communication. The microbenchmark involves two MPI nodes—one sender and one receiver. The sender posts a message of user specified size to be sent to the receiver with the blocking MPI send message routine. The receiver posts the corresponding receive through a non-blocking MPI receive message call and then proceeds to perform computation. The amount of computation performed is again user specified and is in the form of integer increments. After the computation is completed, the receiver waits for the receive request to complete through the MPI wait call. Finally, it sends a 4 byte message back to the sender which allows the sender and the receiver to synchronize their iterations through the microbenchmark loop. Statistics are collected at runtime from the library at the receiver node. Specifically, it collects (i) the duration of the wait call (wait-time), and (ii) total time from the receive being posted to the completion of wait for that particular receive request (total-receive-time). Both the wait-time and total-receive-time values are averaged over the number of iterations of the microbenchmark.

Figure 11 shows the percentage change in wait-time with increasing message sizes for LA-MPI-ED over LA-MPI. The three plots shown in the graph are obtained for different computation amounts of 1 million increments (1M), 10 million increments (10M) and 100 million increments (100M). The 1M plot shows a degradation for small message sizes because the wire latency for these messages is itself comparable to the computation times. Thus, the message arrives at the receiver too late to effectively overlap with computation, exposing the thread switching overhead. However, with increasing message sizes, the communication time increases and message receive is able to overlap with computation more. This results in the reduction in wait-time for message sizes beyond 8 KB for LA-MPI-ED over LA-MPI. This trend continues until message size reaches 64 KB, at which point the overlap between communication and computation is maximized. Beyond this, the communication time is larger than the computation time, so the overlap between them gets limited to the amount of computation and the plot flattens itself out. For the 10M and 100M plots, the computation time is large enough to overlap almost all of the communication time. This results in almost complete elimination of the wait-times with LA-MPI-ED. However, even for the 10M plot there is a drop in the percentage improvement at the highest message size (2 MB). At this message size the communication time exceeds the computation time and hence the percentage wait-time improvement of LA-MPI-ED over LA-MPI falls. The 100M plot would show the same characteristic but for a much larger message, whose communication time would be comparable to the time required for 100 million increments.

Figure 12 shows the percentage improvement in the total-receive-time of LA-MPI-ED over LA-MPI, again as a function of message sizes. The three plots in the graph again correspond to varying amounts of computation. Again, as in Figure 11, the benefits of the event-driven library become apparent only when the amount of computation is high enough to overlap the communication time effectively. For a particular amount of computation, the percentage improvement of total-receive-time increases with increasing message sizes. However, with sufficiently large messages, the communication time eventually becomes a magnitude

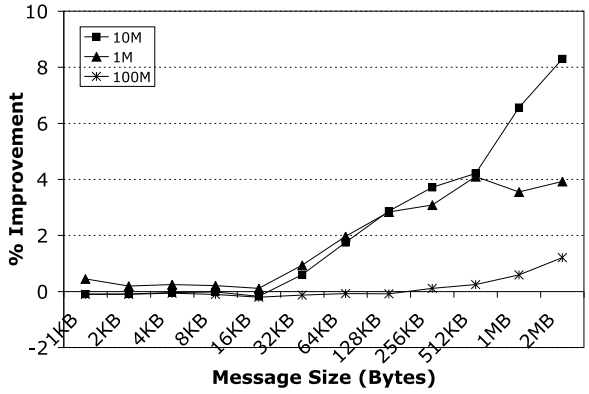

Figure 12. Percentage improvement in total-receive-time of the microbenchmark with the event-driven LA-MPI library

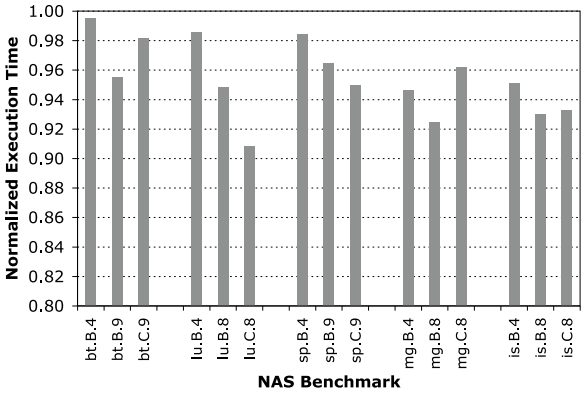

Figure 13. Execution times of NAS benchmarks with LA-MPI-ED relative to that with LA-MPI (scaled to 1)

greater than the computation time causing the plot to flatten out. This is seen for the 1M plot beyond 256 KB message size. The other plots do not show this behavior because the range of message sizes is never large enough to reach this stage. The 100M plot shows significantly lower improvement than the others as the computation time dominates and even a large reduction in the wait-time only has a very limited effect on the total-receive-time.

4.2 NAS Benchmarks

The NAS parallel benchmarks (NPB) is a set of 8 benchmarks, comprised of five application kernels and three simulated computation fluid dynamics (CFD) applications [1, 2]. This paper uses five benchmarks from the NPB version 2.2 suite—BT, SP, LU, IS and MG—to provide a thorough comparison of the event-driven LA-MPI library against the original version of the library. Among the benchmarks, the first three are simulated CFD applications and the latter two are smaller application kernels. The NPB version 2.2 suite supports up to three pre-compiled data-sets, A, B and C (in increasing order of size), for each benchmark. These benchmarks are run on a 4, 8 or 9 node cluster, depending upon whether the benchmark requires a squared or power-of-2 number of nodes. The 4-node experiments are all run with the mid-size data-set (B). The 8-node (or 9) experiments are run for both the B data-set and the larger C data-set.

Figure 13 compares the performance of the five NAS benchmarks achieved with LA-MPI-ED against LA-MPI. Each bar of the graph corresponds to one particular configuration of the benchmark and is named accordingly—the first part of name gives the benchmark name, the middle part conveys the data-set size used for that particular experiment and the last part conveys the number of participating MPI nodes for the experiment. Each bar of the graph depicts the normalized execution time of the corresponding benchmark using LA-MPI-ED relative to LA-MPI. Thus, a bar of length less than 1.0 indicates a performance gain with the event-driven library. Each data point for the graph is generated by averaging the execution times over 10 runs of the experiment.

Figure 13 shows that the benchmarks obtain varying degrees of performance improvement with the LA-MPI-ED library over LA-MPI. Each of the 15 benchmark configurations show an improvement with LA-MPI-ED over LA-MPI, ranging from 0.5% for BT.B.4 to 9.2% for LU.C.8, yielding an average speedup of 4.5%. Overall, the IS benchmark is the best performer with the event-driven library, showing an average speedup of 6.2%. The other 4 benchmarks achieve average speedups in the range of 2.3–5.6%. Among each individual benchmark, for the same data-set size, running the experiment over a larger cluster improves the relative performance of LA-MPI-ED to LA-MPI. Moving to a larger cluster for the same benchmark (with a certain data-set size) increases the communication component of the application with respect to the computation performed at any given node in the cluster. Thus, the event-driven library, with its more efficient communication mechanism, reaps greater benefits. On the other hand, going to a bigger data-set for the same benchmark, increases the computation component of the application more than the communication component. Thus, as shown in Figure 13, the relative performance improvement of LA-MPI-ED compared to LA-MPI sometimes increases and sometimes decreases when moving to the bigger data-set, depending on the particular application.

The results in Section 4.1 show that the benefits of the event-driven library depend on the amount of overlap between computation and communication. IS achieves greater benefits than the other NAS codes because its communication consists primarily of very large messages which are pre-posted before the benchmark goes into a computation phase, enabling effective overlap. While the other bench-

marks do not overlap communication with computation as much, they still do so to varying degrees, yielding noticeable speedups.

4.3 Discussion

The results of the above experiments indicate several situations under which the LA-MPI-ED library either outperforms or underperforms the base LA-MPI. LA-MPI-ED uses an event management thread to improve library responsiveness to incoming messages and opportunities to send messages. This structure leads to performance penalties from thread switching overheads and from a greater number of unexpected receives, a negative side effect of the increased responsiveness. These penalties were evident in the ping and bandwidth microbenchmarks, but were not substantial for sufficiently large messages. Further, these penalties were overcome by the opportunities to reduce message waiting time and thus improve performance in all the other tested codes, as the other codes used non-blocking receives and were structured to overlap communication and computation. Both asynchronous messaging and overlap of computation with communication have become standard tools of high-performance programmers; thus, the positive effects of LA-MPI-ED on the performance of the NAS benchmarks should be representative of the benefits of LA-MPI-ED in other well-tuned applications.

The performance bottlenecks in LA-MPI-ED, such as thread switching overhead and greater number of unexpected receives, may also be addressed by architectural improvements. For example, thread switching overhead between a small number of threads is largely eliminated in processors that include multiple thread contexts for hyper-threading or simultaneous multithreading [9, 18]. Although these systems would still see some inter-thread communication overheads, those should be low as the communication is only of metadata in shared memory regions that may be cached. Additionally, the problem of extra memory copies on unexpected receives can be resolved through the use of network interface cards that attempt to buffer data until an actual receive is posted. Such systems could then dispatch data directly from the network interface to the application buffer when needed. These and other architectural advances help to mitigate the downside of LA-MPI-ED while still maintaining the basic performance advantage of this library: using increased responsiveness to overlap computation and communication more effectively.

The current implementation of LA-MPI-ED only allows for TCP-based communication, as only TCP/IP is well-integrated with event notification in standard operating systems. In contrast, a more common approach to high-performance cluster computing is through bypassing the operating system and using specialized networks such as Myrinet or Quadrics [4, 22]. While these specialized networks once outperformed Ethernet, with the advent of 1–10 Gbps Ethernet, it is no longer clear that there is a significant performance benefit in network speed. Specialized user-level networks offer other advantages, such as reducing copies in data transmission, avoiding interrupts, and offloading message protocol processing onto a network interface. However, many of these same techniques can be integrated into commodity networks using TCP and Ethernet. For example, recent work looks to offload all or part of TCP processing onto an Ethernet network interface, reducing acknowledgment-related interrupts as a side-effect [3, 17]. Further, various techniques are available to eliminate extraneous copies for sends and posted receives [8, 20, 23]. Copy avoidance could be provided for unexpected receives through network interface support (as described above), while preposting library buffers to the operating system in conjunction with a system such as Fast Sockets would allow the reduction of one extra copy from the socket buffer to the library buffers [23]. When used together with the event-driven communication model described in this paper, copy avoidance, interrupt avoidance, and TCP offloading could help to make TCP a low-cost, portable, and reliable alternative to specialized networks.

5 Related Work

The success of specialized networks and user level messaging protocols has led to very little successful research in using TCP as a high-performance messaging protocol. One such research work has been TCP splintering, which focusses on the limitations of TCP as a communication medium in a cluster environment, especially those arising out of TCP's congestion control and flow control policies [14]. Other research in high-performance TCP has shown significant performance gains with optimizations such as zero-copy send/receive and offloading specific tasks like checksumming onto the NIC [11]. In practice, congestion control and flow control policies, however unnecessary in a cluster environment, become indispensable for reliable MPI communication. Utilizing TCP's inherent reliability mechanisms turn out to be less error prone and more efficient than providing the same at the library level. TCP/IP optimizations such as zero-copy send/receive or checksum offloading remain complementary to the MPI library enhancements proposed in this paper and can only help to further improve the performance of TCP based MPI communication.

Commonly used and publicly available MPI libraries, such as MPICH and LAM/MPI both use some form of a synchronous progress engine to handle pending requests [6, 16, 24]. Both of these libraries also support TCP messaging over Ethernet. The progress engine for TCP in both of these libraries uses a `select`-based polling approach similar to the technique employed in the original LA-MPI de-

sign. Hence these libraries also suffer from the same set of drawbacks that were identified with LA-MPI—specifically, progression of pending requests is dependent on library invocation rather than on the occurrence of corresponding network events. None of the research efforts on MPI communication using TCP utilizes the well-known techniques in the network server domain of event-driven software architecture and threading, despite the fact that most recent releases of common MPI libraries do incorporate the notion of events for messaging over TCP.

A significant amount of research has been done in exploiting task concurrency in applications using threads. This technique has been particularly popular for applications involving significant asynchronous activity such as web-servers. Threading however, imposes overheads such as cache and TLB misses, scheduling overheads and lock contention. The overheads start becoming particularly severe as the number of threads increases. Popular web servers such as Microsoft's IIS and Apache circumvent this problem by using a size bounded thread pool from which threads are utilized to process requests [13]. When the maximum number of threads are already in use, the server rejects additional requests. Since LA-MPI-ED uses only two threads, it is free from these scalability problems associated with a thread-based approach.

Event-driven software architecture is another technique that has been favored by web servers to maximize concurrency and handle the asynchronous behavior inherent in the application. This approach also avoids the scalability issues involved with threading. Several web-servers have been developed using the event-driven model to handle asynchronous network events and maximize web-server throughput. For example, Flash employs non-blocking threads not only to handle network events, but also to satisfy filesystem requests [21]. The Harvest web cache has an architecture very similar to to Flash but employs just one thread since most of its workload can be kept in main memory, and thus file accesses are ensured not to block [5, 7].

6 Conclusions

MPI libraries which tie communication progress to library invocations are inefficient and require the application to balance library invocations with messaging responsiveness. The addition of an event-driven communication thread enables communication progress to occur in response to network events, rather than when the MPI library is invoked. This paper evaluates an event-driven communication thread that utilizes TCP network events delivered by the operating system to process messages efficiently. This threaded implementation of the TCP path of the LA-MPI library dramatically improves communication responsiveness, significantly reducing the wait time for non-blocking receives, and even entirely eliminating it in certain cases. This results in an average of 4.5% performance improvement on 5 NAS benchmarks, with a peak improvement of 9.2%. Overlapping communication with computation is one step towards making TCP a competitive high-performance MPI communications protocol. The further addition of copy reduction and interrupt avoidance should make TCP over Ethernet a viable alternative to specialized networks for MPI communication.

References

[1] D. H. Bailey, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinki, R. S. Schreiber, H. D. Simon, V. Venkatakrishnam, and S. K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[2] D. H. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. NASA Technical Report NAS-95-020, NASA Ames Research Center, December 1995. http://www.nas.nasa.gov/Software/NPB/.

[3] H. Bilic, Y. Birk, I. Chirashnya, and Z. Machulsky. Deferred Segmentation for Wire-Speed Transmission of Large TCP Frames over Standard GbE Networks. In *Hot Interconnects IX*, pages 81–85, Aug. 2001.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE MICRO*, Feb 1995.

[5] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28(1–2):119–125, 1995.

[6] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrel. A heirarchial Internet object cache. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 153–163, Jan 1996.

[8] P. Druschel and L. L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP-14)*, pages 189–202, Dec. 1993.

[9] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stam, and D. M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, 17(5), 1997.

[10] T. M. P. I. Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[11] A. Gallatin, J. Chase, and K. Yocum. Trapeze/IP: TCP/IP at near-gigabit speeds. In *Proceedings of 1999 USENIX Technical Conference*, pages 109–120, June 1999.

[12] L. Gammo, T. Brecht, A. Shukla, and D. Pariag. Comparing and Evaluating epoll, select, and poll Event Mechanisms. In *Proceedings of the Ottawa Linux Symposium*, July 2004.

[13] D. Gaudet. Apache Performance Notes. http://httpd.apache.org/docs/misc/perf-tuning.html.

[14] P. E. Gilfeather and A. B. Maccabe. Making TCP Viable as a High Performance Computing Protocol. In *Proceedings of the LACSI Symposium*, 2002.

[15] 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 Terascacle Clusters. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing*, June 2002.

[16] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[17] Y. Hoskote, B. A. Bloechel, G. E. Dermer, V. Erraguntla, D. Finan, J. Howard, D. Klowden, S. G. Narendra, G. Ruhl, J. W. Tschanz, S. Vangal, V. Veeramachaneni, H. Wilson, J. Xu, and N. Borkar. A TCP Offload Accelerator for 10 Gb/s Ethernet in 90-nm CMOS. *IEEE Journal of Solid-State Circuits*, 38(11):1866–1875, Nov. 2003.

[18] K. Krewell. Intel's Hyper-Threading Takes Off. *Microprocessor Report*, 2002.

[19] J. Lemon. Kqueue: A generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.

[20] E. Nahum, T. Barzilai, and D. Kandlur. Performance Issues in WWW Servers. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 216–217, May 1999.

[21] V. S. Pai, P. Druschel, and W. Zwaenpoel. FLASH: An Efficient and Portable Web Server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[22] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The QUADRICS Network: High-Performance Clustering Technology. *IEEE MICRO*, Jan 2002.

[23] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-Performance Local Area Communication With Fast Sockets. In *Proceedings of the 1997 USENIX Technical Conference*, pages 257–274, Jan. 1997.

[24] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.