Implementing Efficient MPI on LAPI for IBM RS/6000 SP Systems: Experiences and Performance Evaluation

Mohammad Banikazemi† Rama K Govindaraju‡ Robert Blackmore‡
Dhabaleswar K Panda†

†Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210
Email: {banikaze, panda}@cis.ohio-state.edu

‡Communication Subsystems
IBM Power Parallel Systems
Poughkeepsie, NY 12601
Email: {ramag,blackmor}@us.ibm.com

Abstract

The IBM RS/6000 SP system is one of the most cost-effective commercially available high performance machines. IBM RS/6000 SP systems support the Message Passing Interface standard (MPI) and LAPI. LAPI is a low level, reliable and efficient one sided communication API library, implemented on IBM RS/6000 SP systems. This paper explains how the high performance of the LAPI library has been exploited in order to implement the MPI standard more efficiently than the existing MPI. It describes how to avoid unnecessary data copies at both the sending and receiving sides for such an implementation. The resolution of problems arising from the mismatches between the requirements of the MPI standard and the features of LAPI is discussed. As a result of this exercise, certain enhancements to LAPI are identified to enable an efficient implementation of MPI on LAPI. The performance of the new implementation of MPI is compared with that of the underlying LAPI itself. The latency (in polling and interrupt modes) and bandwidth of our new implementation is compared with that of the native MPI implementation on RS/6000 SP systems. The results indicate that the MPI implementation on LAPI performs comparably or better than the original MPI implementation in most cases. Improvements of up to 17.3% *in polling mode latency,* 35.75% *in interrupt mode latency, and* 20.9% *in bandwidth are obtained for certain message sizes. The implementation of MPI on top of LAPI also outperforms the native MPI implementation for the NAS Parallel Benchmarks. It should be noted that the implementation of MPI on top of LAPI is not a part of any IBM product and no assumptions should be made regarding its availability as a product.*

1 Introduction

The IBM RS/6000 SP[1] system [1, 9] (referred to as SP in the rest of this paper) is a general-purpose scalable parallel system based on a distributed-memory, message-passing architecture. Configurations ranging from 2-node systems to 128-node systems are available from IBM. The uniprocessor nodes are available with the latest Power2-Super (P2SC) microprocessors and the TB3 adapter. The SMP nodes are available with the 4 way, Power-PC 332MHz microprocessors and the TBMX adapter. The nodes are interconnected via a switch adapter to a high-performance, multistage, packet-switched network for interprocessor communication capable of delivering bi-directional data-transfer rate of up to 160 MB/s between each node pair. Each node contains its own copy of the standard AIX operating system and other standard RS/6000 system software.

IBM SP systems support several communication libraries like MPI [6], MPL and LAPI [4, 7]. MPL, an IBM designed interface, was the first message passing interface developed by IBM on SP systems. Subsequently, after MPI became a standard it was implemented by reusing some of the infrastructure of MPL. This reuse allowed for SP systems to provide an implementation of MPI quite rapidly, but also imposed some inherent constraints on the MPI implementation which are discussed in detail in Section 2. In 1997, the LAPI library interface was designed and implemented on SP systems. The primary design goal for LAPI was to define an architecture with semantics that would allow efficient implementation on the underlying hardware and firmware infrastructure provided by SP systems. LAPI is a user space library, which provides a one-sided communication model thereby avoiding the complexities associated with two-sided protocols (like message matching, ordering, etc.).

In this paper we describe the implementation of the MPI standard on top of LAPI (MPI-LAPI) to avoid some of the inherent performance constraints of the current implementation of MPI (native MPI) and to exploit the high performance of LAPI. There are some challenges involved in implementing a 2-sided protocol such as MPI on top of a 1-sided protocol such as LAPI. The major issue is finding the address of the receiving buffer. In 2-sided protocols, the sender does not have any information about the address of the receive buffer where the message should be copied into. There are some existing solutions to this problem. A temporary buffer can be used at the receiving side to store the message before the address of its destination is resolved. This solution incurs the cost of a data copy which increases the data transfer time and the protocol overhead especially for large messages. An alternative solution to this problem is using a rendezvous protocol, in which

[1] IBM, RS/6000, SP, AIX, Power-PC, and Power2-Super are trademarks or registered trademarks of the IBM Corporation in the United States or other countries or both.

in response to the request from the sender, the receiver provides the receive buffer address to the sender, and then the sender can send the message. In this method the unnecessary data copy (into a temporary buffer) is avoided, but the cost of roundtrip control messages for providing the receive buffer address to the sender impacts the performance (especially for small messages) considerably. The impact is increased latency and control traffic. It is therefore important that a more efficient method be used for resolving the receive buffer address. In this paper, we explain how the flexibility of the LAPI architecture is used to solve this problem in an efficient manner. Another challenge in implementing MPI on top of LAPI is to keep the cost of enforcing the semantics of MPI small so that the efficiency of LAPI is realized to the fullest. Another motivation behind our effort has been to provide better reuse by making LAPI the common transport layer for other communication libraries. Once again, it should be noted that MPI-LAPI is not a part of any IBM product and no assumptions should be made regarding its availability as a product.

This paper is organized as follows: In Section 2, we detail the different messaging layers in the current implementation of MPI. In Section 3, we present an overview of LAPI and its functionality. In Section 4, we discuss different MPI communication modes and show how these modes are supported by using LAPI. In Section 5, we discuss different strategies that are used to implement MPI on top of LAPI and the various changes we made to improve the performance of MPI-LAPI. Experimental results including latency, bandwidth, and benchmark performance are presented in Section 6. Related work is discussed in Section 7. In Section 8, we outline some of our conclusions.

2 The Native MPI Overview

The protocol stack for the current implementation of MPI on SP systems is shown in Figure 1a. This protocol stack consists of several layers. The functions of each of the layers is described briefly below:

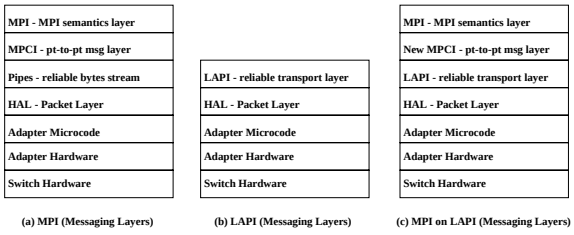

Figure 1. Protocol Stack Layering.

- The MPI layer enforces all MPI semantics. It breaks down all collective communication calls into a series of point-to-point message passing calls in MPCI (Message Passing Client Interface).
- The MPCI layer provides a point-to-point communication interface with message matching, buffering for early arrivals, etc. It sends data by copying data from the user buffer into the pipe buffers. The pipe layer then has responsibility for sending the data. Likewise data received by the pipe layer is matched, and if the corresponding receive has been posted, copied from the pipe buffers into the user buffer, otherwise the data is copied into an early arrival buffer (if the receive is not posted).
- The Pipes layer provides a reliable byte stream interface [8]. It ensures that data in the pipe buffers is reliably transmitted and received. This layer is also used to enforce ordering of packets at the receiving end pipe buffer if packets come out of order (the switch network has four routes between each pair of nodes and packets on some routes can take longer than other routes based on the switch congestion on the route). A sliding window flow control protocol is used. Reliability is enforced using an acknowledgement-retransmit mechanism.
- The HAL layer (packet layer, also referred to as the Hardware Abstraction Layer) provides a packet interface to the upper layers. Data from the pipe buffers are packetized in the HAL network send buffers and then injected into the switch network. Likewise packets arriving from the network are assembled in the HAL network receive buffers. The HAL network buffers are pinned down. The HAL layer handshakes with the adapter microcode to send/receive packets to/from the switch network.
- The Adapter DMAs the data from the HAL network send buffers onto the switch adapter and then injects the packet into the switch network. Likewise, packets arriving from the switch network into the switch adapter are DMAed onto the HAL network receive buffers.

The current MPI implementation, for the first and last 16K bytes of data, incurs a copy from the user buffer to the pipes buffer and from the pipe buffers to the HAL buffers for sending messages [8]. Similarly, received messages are first DMAed into HAL buffers and then copied into the pipe buffer. The extra copying of data is performed in order to simplify the communication protocol. These two extra data copies affect the performance of MPI. In the following sections we discuss LAPI (Fig. 1b) and explain how LAPI can replace the Pipes layer (Fig. 1c) in order to avoid the extra data copies and improve the performance of the message passing library.

3 LAPI Communication Model Overview

LAPI is a low level API designed to support efficient one-sided communication between tasks on SP systems [9]. The protocol stack of LAPI is shown in Figure 1b. An overview of the LAPI communication model (for LAPI_Amsend) is given in Figure 2 which has been captured from [7]. Different steps involved in LAPI communication functions are as follows. Each message is sent with a LAPI header, and possibly a user header (step 1). On arrival of the first packet of the message at the target machine, the header is parsed by a header handler (step 2) which is responsible for accomplishing three tasks (step 3). First, it must return the location of a data buffer where the message is to be assembled. Second, it may return a pointer to a completion handler function which is called when all the packets have arrived in the buffer location previously returned. Finally, if a completion handler function is provided,

it also returns a pointer to data which is passed to the completion handler. The completion handler is executed after the last packet of the message has been received and copied into a buffer (step 4). In general, three counters may be used so that a programmer may determine when it is safe to reuse buffers and to indicate completion of data transfer. The first counter (org_cntr) is the origin counter, located in the address space of the sending task. This counter is incremented when it is safe for the origin task to update the origin buffer. The second counter, located in the target task's address space, is the target counter (tgt_cntr). This counter is incremented after the message has arrived at the target task. The third counter, the completion counter (cmpl_cntr) is updated on completion of the message transfer. This completion counter is similar to the target counter except it is located in the origin task's address space.

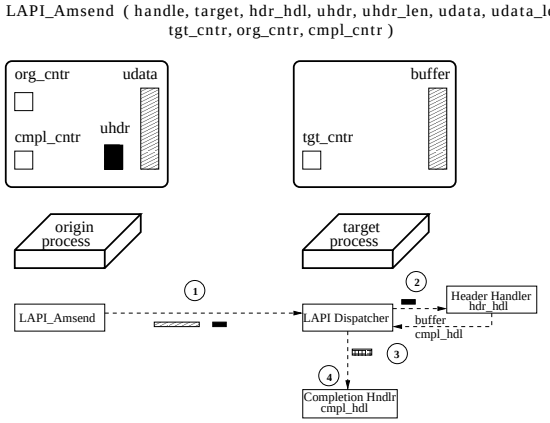

Figure 2. LAPI overview.

The use of LAPI functions may require that the origin task specify pointers to either functions or addresses in the target task address space. Once the address of the header handler has been determined, the sending process does not necessarily need to know the receive buffer address in the receiver address space since the header handler is responsible for returning the receive buffer address. The header handler may, for example, interpret the header data as a set of tags which, when matched with requests on the receiving side, may be used to determine the address of the receive buffer. As we shall see, this greatly simplifies the task of implementing a two sided communication protocol with a one sided infrastructure. To avoid deadlocks, LAPI functions cannot be called from header handlers. The completion handlers are executed on a separate thread and can make LAPI calls.

LAPI functions may be broadly broken into two classes of functions. The first of these are communication functions using the infrastructure described above. In addition to these communication functions, there are a number of utility function provided so that the communication functions may be effectively used. All the LAPI functions are shown in Table 1. For more information about LAPI we refer the reader to [7].

LAPI Function	Purpose
LAPI_Init	Initialize the LAPI subsystem
LAPI_Term	Terminate the LAPI subsystem
LAPI_Put	Data transfer function
LAPI_Get	Data transfer function
LAPI_Amsend	Active message send function
LAPI_Rmw	Synchronization read-modify-write
LAPI_Setcntr	Set the value of a counter
LAPI_Getcntr	Get the value of a counter
LAPI_Waitcntr	Wait for a counter to reach a value
LAPI_Address_init	Exchange addresses of interest
LAPI_Fence	Enforce ordering of messages
LAPI_Gfence	Enforce ordering of messages
LAPI_Qenv	Query the environment state
LAPI_Senv	Set the environment state

Table 1. LAPI Functions.

4 Supporting MPI on top of LAPI

The protocol stack used for the new MPI implementation is shown in Figure 1c. The PIPE layer is replaced by the LAPI layer. The MPCI layer used in this implementation is thinner than that of the native MPI implementation and does not include the interface with the PIPE layer. In this section, we first discuss different communication modes defined by MPI and then explain how the new MPCI layer has been designed and implemented to support MPI on top of LAPI.

The MPI standard defines four communication modes: *Standard*, *Synchronous*, *Buffered*, and *Ready* modes [6]. These four modes are usually implemented by using two internal protocols called *Eager* and *Rendezvous* protocols. The translation of the MPI communication modes into these internal protocols in our implementation is shown in Table 2. The Rendezvous protocol is used for large messages to avoid the potential buffer exhaustion caused by unexpected messages (whose receives have not been posted by the time they reach the destination). The value of Eager Limit can be set by the user and has a default value of 4096 bytes. This value can be set to smaller or larger values based on the size of the buffer available for storing unexpected messages.

In Eager protocol, messages are sent regardless of the state of the receiver. Arriving messages whose matching receives have not yet been posted are stored in a buffer called the *Early Arrival Buffer* until the corresponding receives get posted. If an arriving message finds its matching receive, the message is copied directly to the user buffer. In the Rendezvous protocol, a *Request_to_send* control message is first sent to the receiver which is acknowledged as soon as the matching receive gets posted. The message is sent to the receiver only after the arrival of this acknowledgment.

The blocking and nonblocking versions of the MPI communication modes have been defined in the MPI standard. In the blocking version, after a send operation, control returns to the application only after the user data buffer can be reused by the application. In the blocking version of the receive operation, control returns to the application only when the message has been completely received into the application buffer. In

MPI Communication Mode	Internal Protocol
Standard	if (size $<$ Eager Limit) eager else rendezvous
Ready	Eager
Synchronous	Rendezvous
Buffered	if (size $<$ Eager Limit) eager else rendezvous

Table 2. Translation of MPI communication modes to internal protocols.

the nonblocking version of send operations, control immediately returns to the user once the message has been submitted for transmission and it is the responsibility of the user to ensure safe reuse of its send buffer (by using MPI_WAIT or MPI_TEST operations). In the nonblocking version of receive, the receive is posted and control is returned to the user. It is the responsibility of the user to determine if the message has arrived. In the following sections we explain how the internal protocols and MPI communication modes are implemented by using LAPI.

4.1 Implementing the Internal Protocols

As mentioned in Section 3, LAPI provides one-sided operations such as LAPI_Put and LAPI_Get. LAPI also provides Active Message style operations through the LAPI_Amsend function. We decided to implement the MPI point-to-point operations on top of this LAPI active message infrastructure. The LAPI active message interface (LAPI_Amsend) function provides some enhancements to the active message semantics defined in GAM [10]. The LAPI_Amsend function allows the user to specify a header handler function to be executed at the target side once the first packet of the message arrives at the target. The header handler must return a buffer pointer to LAPI which tells LAPI where the packets of the message must be reassembled. The ability to not require the task making the LAPI_Amsend call specify the target address for the messages being sent, makes it ideally suited to be used as the basis for implementing MPI-LAPI. The header handler is used to process the message matching and early arrival semantics, thereby avoiding the need for an extra copy at the target side. The header handler also allows the user to specify a completion handler function to be executed after all the packets of the message have been copied into the target buffer. The completion handler therefore serves to allow the application to incorporate the arriving message into the ongoing computation. In our MPI implementation the completion handler serves to update local state of marking messages complete, and possibly sending control message back to the sender. The LAPI_Amsend therefore provides the hooks to allow applications to get control when the first packet of a message arrives and when the complete message has arrived at the target buffer, making it ideal to be used as a basis for implementing MPI-LAPI. In Sections 4.1.1 and 4.1.2, we explain how the Eager and Rendezvous protocols have been implemented.

4.1.1 Implementing the Eager Protocol

In the MPI-LAPI implementation, LAPI_Amsend is used to send the message to the receiver (Fig. 3a). The message descriptions (such as message TAG and Communicator) are encoded in the user header which is passed to the header handler (Fig. 3b). Using the message description, the posted "Receive Queue" (Receive_queue) is searched to see if a matching receive has already been posted. If such a receive has been posted, the address of the user buffer is returned to LAPI and LAPI assembles the data into the user buffer. It should be noted that LAPI will take care of out of order packets and copy the data into the correct offset in the user buffer. If the header handler doesn't find a matching receive, it will return the address of an "Early Arrival Buffer" (EA_buffer) for LAPI to assemble the message into. (The buffer space is allocated if needed.) The header handler also posts the arrival of the message into the "Early Arrival Queue" (EA_queue). If the message being received is a Ready-mode message and its matching receive has not yet been posted, a fatal error is raised and the job is terminated. If the matching receive is found, the header handler also sets the function Eager_cmpl_hdl to be executed as the completion handler. The completion handler is executed, when the whole message has been copied into the user buffer, and the corresponding receive is marked as complete (Fig. 3c). It should be noted that in order to make the description of the implementation more readable, we have omitted some of the required parameters of the LAPI functions from the outlines.

(a)
```
Function Eager_send
        LAPI_Amsend(eager_hdr_hdl, msg_description, msg)
end Eager_send
```
(b)
```
Function Eager_hdr_hdl(msg_description)
        if (matching_receive_posted(msg_description)) begin
            completion_handler = Eager_cmpl_hdl
            return (user_buffer)
        end else begin
            if  (Ready_Mode)
                Error_handler(Fatal, "Recv not posted")
            post msg_description in EA_queue
            completion_handler = NULL
            return (EA_buffer)
        endif
end Eager_hdr_hdl
```
(c)
```
Function Eager_cmpl_hdl(msg_description)
        Mark the recv as COMPLETE
end Eager_cmpl_hdl
```

Figure 3. Outline of the Eager protocol: (a) Eager send, (b) the header handler for the eager send and (c) the completion handler for the eager send.

4.1.2 Implementing the Rendezvous Protocol

The Rendezvous protocol is implemented in two steps. In the first step a request_to_send control message is sent to the receiver by using LAPI_Amsend (Fig. 4). The second step is executed when the acknowledgment of this message is received (indicating that the corresponding receive has been posted). The message is sent by using LAPI_Amsend the same way the message is transmitted in Eager protocol (Fig. 3a). In the next section, we explain how these protocols are employed to implement different communication modes as defined in the MPI standard.

```
(a)
    Function Request_to_send
        LAPI_Amsend(Request_to_send_hdr_hdl,
                    msg_description, NULL)
    end Request_to_send
(b)
    Function Request_to_send_hdr_hdl(msg_description)
        if (matching_receive posted(msg_description)) begin
            completion_handler = Request_to_send_cmpl_hdl
            return (NULL)
        end else begin
            post msg_description in EA queue
            completion_handler = NULL
            return (NULL)
        endif
    end Request_to_send_hdr_hdl
(c)
    Function Request_to_send_cmpl_hdl(msg_description)
        LAPI_Amsend(Request_to_send_acked_hdr_hdl,
                    msg_description, NULL)
    end Request_to_send_cmpl_hdl
```

Figure 4. Outline of the first phase of the Rendezvous protocol: (a) Request to Send, (b) The Header handler for the request to send and (c) the completion handler for the request to send.

4.2 Implementing the MPI Communication Modes

Standard-mode messages which are smaller than the Eager Limit and Ready-mode messages are sent by using the Eager protocol (Fig. 5). Depending on whether the send is blocking or not, a wait statement (LAPI_Waitcntr) might be used to ensure that the user buffer can be reused.

Standard-mode messages which are longer than the Eager Limit and Synchronous-mode messages are transmitted by using the 2-phase Rendezvous protocol. Figure 6 illustrates how these sends are implemented. In the non-blocking version, the second phase of the send is executed in the completion handler which is specified in the header handler corresponding to the active message sent for acknowledging the Request_to_send message as shown in Figure 7.

Buffered mode messages are transmitted using the same procedure as for sending nonblocking standard messages. The only difference is that messages are first copied into a user specified buffer (defined by MPI_Buffer_attach). The receiver informs the sender when the whole message has been received so that the sender can free the buffer used for transmitting the message (Figure 8).

Figure 9 shows how blocking and non-blocking receive operations are implemented. It should be noted that in response to a Request_to_send message, a LAPI_Amsend is used to acknowledge the request. When this acknowledgment is received at the sender side of the original communication, the entire message will be transmitted to the receiver. If the original send operation is a blocking send, the sender is blocked until the Request_to_send message is marked as acknowledged and the blocking send will send out the message. If the original message is a nonblocking send, the message is sent out in the completion handler specified in the header handler of Request_to_send_acked (Fig. 7).

```
Function StndShort_ready_send
    Eager_send
    if (blocking)
        Wait until Origin counter is set
end StndShort_ready_send
```

Figure 5. Outline of the standard send for messages shorter than the eager limit and the ready-mode send.

```
Function StndLong_sync_send
    Request_to_send
    if (blocking) begin
        Wait until request_to_send is acknowledged
        Eager_send
        Wait until Origin counter is set
    endif
end StndLong_sync_send
```

Figure 6. Outline of the standard send for messages longer than the eager limit and the synchronous-mode send.

5 Optimizing the MPI-LAPI Implementation

In this section we first discuss the performance of the base implementation of MPI-LAPI which is based on the description outlined in Section 4. After discussing the shortcomings of this implementation, we present two methods to improve the performance of MPI-LAPI.

5.1 Performance of the Base MPI-LAPI

We compared the performance of our base implementation with that of LAPI itself. We measured the time to send a number of messages (with a particular message size) from one node to another node. Each time the receiving node would send back a message of the same size, and the sender node will send a new message only after receiving a message from the receiver. The number of messages being sent back and forth was long enough to make the timer error negligible. The granularity of the timer was less than a microsecond. LAPI_Put and LAPI_Waitcntr were used to send the message and to wait for the reply, respectively. The time for the MPI-LAPI implementation was measured in a similar fashion. MPI_Send and MPI_Recv were the communication functions used for this experiment. It should be noted that in all cases, the Rendezvous protocol was used for messages larger than the Eager Limit ($4K$ bytes). Figure 10 shows the measured time for messages of different sizes. We observed that message transfer time of the MPI-LAPI implementation was too high to be attributed only to the cost of protocol processing like message matching which are required for the MPI implementation but not for the 1-sided LAPI primitives.

5.2 MPI-LAPI with Counters

Careful study of the design and profiling of the base implementation showed that the cost of thread context switching required from the header handler to the completion handler was the major source of increase in the data transfer time. It should be noted that in LAPI, completion handlers are executed on a separate thread (Section 3). To verify this hypothesis, we modified the design such that we do not require the execution

```
Function Request_to_send_acked_hdr_hdl
        if (blocking(msg_description))
            mark the request as acknowledged
        else
            completion_handler =
              Request_to_send_acked_cmpl_hdl
end Request_to_send_acked_hdr_hdl
Function Request_to_send_acked_cmpl_hdl
        Eager_send
end Request_to_send_acked_cmpl_hdl
```

Figure 7. Outline of receive for messages sent using the Rendezvous protocol.

```
Function Buffered_send
        Copy the msg to the attached buffer
        if (msg_size < EagerLimit)
            Eager_send
        else
            Request_to_send
end Buffered_send
```

Figure 8. Outline of the buffered-mode send.

of completion handlers. As described in Section 4, when the eager protocol is used, the only action taken in the completion handler is marking the message as completed (Fig. 3) such that the receive (or MPI_WAIT or MPI_TEST) can recognize the completion of the receipt of the message. LAPI provides a set of counters to signal the completion of LAPI operations. The target counter specified in LAPI_Amsend is updated (incremented by one) after the message is completely received (and the completion handler, if there exist any, has executed). We used this counter to indicate that message has been completely received. However, the address of this counter which resides at the receiving side of the operation should be specified at the sender side of the operation (where LAPI_Amsend is called). In order to take advantage of this feature, we modified the base implementation to use a set of counters whose addresses are exchanged among the participating MPI processes during initialization. By using these counters we avoided using the completion handler of messages sent through the eager protocol. We could not employ the same strategy for the first phase of the Rendezvous protocol. The reception of the Request_to_send control messages at the receiving side does not imply that the message can be sent. If the receive has not yet been posted, the sender cannot start sending the message even

```
Function Receive
        if (found_matching_msg(EA_queue, msg_description))
            if (request_to_send) begin
                LAPI_Amsend(Request_to_send_acked,
                            msg_description, NULL)
            endif
        else
            Post the receive in Receive_queue
        if (blocking)
            Wait until msg is marked as COMPLETE
end Receive
```

Figure 9. Outline of receive for messages sent by the Eager protocol.

though the Request_to_send message has been already received at the target. The time for the message transfer of this modified version is shown in Figure 10. As it can be observed, this implementation provided better performance for short messages (which are sent in Eager mode) compared to the base implementation. This experiment was solely performed to verify the correctness of our hypothesis.

5.3 MPI-LAPI Enhanced

The results in Figure 10, confirmed our hypothesis that the major source of overhead was the cost of context switching required for the execution of the completion handlers. We showed how we can avoid using completion handlers for messages which are sent in Eager mode. However, we still need to use completion handlers for larger messages (sent in Rendezvous mode). In order to avoid the high cost of context switching for all messages, we enhanced LAPI to include predefined completion handlers in the same context. In this modified version of LAPI, operations such as updating a local variable or a remote variable (which requires a LAPI function call), indicating the occurrence of certain events, were executed in the same context. The results of this version is shown in Figure 10. The time of this version of MPI-LAPI comes very close to that of the bare LAPI itself. The difference between the curves can be attributed to the cost of posting and matching receives required by MPI, and also the cost of locking and unlocking of the data structures used for these functions at the MPI level.

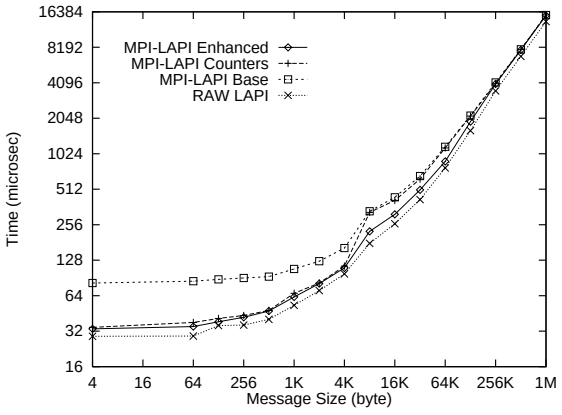

Figure 10. Comparison between the performance of raw LAPI and different versions of MPI-LAPI.

In the following section, we compare the latency and bandwidth of our MPI-LAPI Enhanced implementation with that of the native MPI implementation. We also explain the difference between the performance of these two implementations.

6 Performance Evaluation

In this section we first present a comparison between the native MPI and MPI-LAPI (the Enhanced version) in latency and bandwidth. Then, we compare the results obtained from running the NAS benchmarks using MPI-LAPI with those obtained from running NAS benchmarks using the native MPI. In all of our experiments we use a SP system with Power-PC

332MHz nodes and the TBMX adapter. The Eager Limit is set to $4K$ bytes for all experiments.

6.1 Latency and Bandwidth

We compared the performance of MPI-LAPI with that of the native MPI available on SP systems. The time for message transfer was measured by sending messages back and forth between two nodes as described in Section 5. The MPI primitives used for these experiments were MPI_Send and MPI_Recv. The eager limit for both systems was set to $4K$ bytes. To measure the bandwidth, we repeatedly sent messages out from one node to another node for a number of times and then waited for the last message to be acknowledged. We measure the time for sending these back to back messages and stop the timer when the acknowledgment of the last message is received. The number of messages being sent is large enough to make the time for transmission of the acknowledgment of the last message negligible in comparison with the total time. For this experiment we used MPI_Isend and MPI_Irecv primitives.

Figure 11 illustrates the time of MPI-LAPI and the native MPI for different message sizes. It can be observed that, the time of MPI-LAPI for very short messages is slightly higher than that of the native MPI. This increase is in part due to the extra parameter checking by LAPI, which unlike the internal Pipes interface is an exposed interface. The difference between the size of the packet headers in these two implementations is another factor which contributes to the slightly increased latency. The size of headers in the native MPI is 16 bytes, and the size of headers for MPI-LAPI is 48 bytes. It can be also observed that for messages larger than 256 bytes, the latency of MPI-LAPI becomes less than that of the native MPI. An improvement of up to 17.3% was measured. As mentioned earlier, unlike the native implementation of MPI, in the MPI-LAPI implementation messages are copied directly from the user buffer into the NIC buffer and vice versa. Avoiding the extra data copying helps improve the performance of the MPI-LAPI implementation.

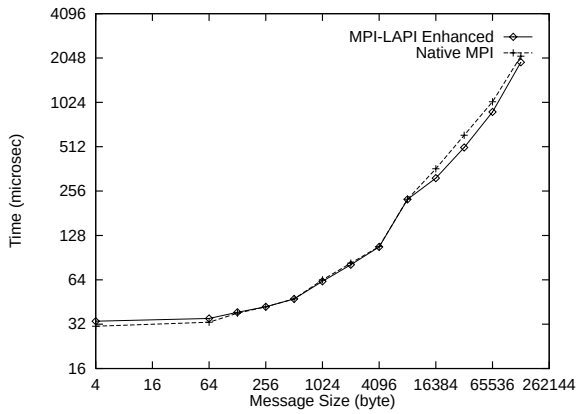

Figure 11. Comparison between the performance of the native MPI and MPI-LAPI.

The obtainable bandwidth of the native MPI and MPI-LAPI is shown in Figure 12. It can be seen that, for a wide range of message sizes, the bandwidth of MPI-LAPI is higher than that of the native MPI. For $64K$ byte messages, MPI-LAPI achieves a bandwidth of $83.35MB/sec$ which indicates a 20.9% improvement in comparison with the $68.93MB/sec$ bandwidth obtained by using the native MPI.

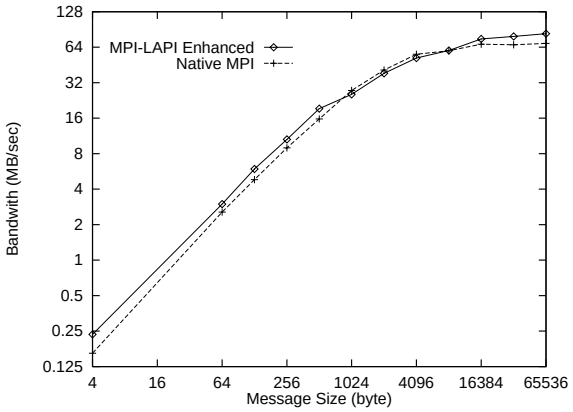

Figure 12. Comparison between the performance of the native MPI and MPI-LAPI.

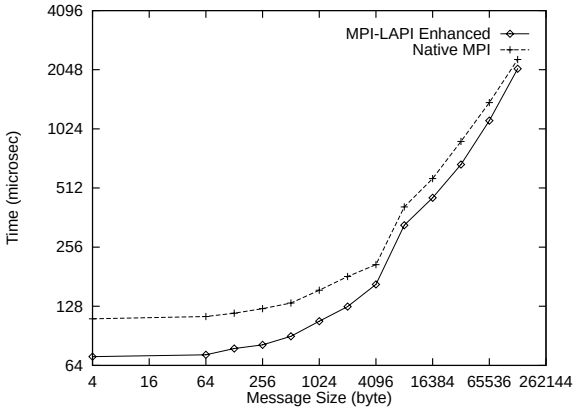

Figure 13. Comparison between the performance of the native MPI and MPI-LAPI in interrupt mode.

For measuring the time required for sending messages from one node to another node in interrupt mode, we used a method similar to the one used for measuring latency. The only difference was that the receiver would post the receive (using MPI_Irecv) and check the content of the receive buffer until the message has arrived. Then it would send back a message with the same size. The results of our measurements are shown in Figure 13. It can be seen that MPI-LAPI performs consistently and considerably better than the native MPI implementation. For short messages of 4 bytes an improvement of 35.75% is observed. The native MPI performs poorly in this experiment. The major reason behind the poor performance of the native MPI is the hysteresis scheme used in it. In the interrupt handler of the native MPI, the interrupt handler waits for a certain period of time to see if more packets are coming to avoid further interrupts. If more are coming then they increase the time the interrupt handler waits in the loop. The value of this waiting period can be set by the user. LAPI does not use any such hysteresis in its interrupt handler and thus, provides better performance.

6.2 NAS Benchmarks

In this section we present the execution times of programs from the NAS benchmark for the native MPI and MPI-LAPI. NAS Parallel Benchmarks (version 2.3) consist of eight benchmarks written in MPI. These benchmarks were used to evaluate the performance of our MPI implementation in a more realistic environment. We used the native implementation of MPI and MPI-LAPI to compare the execution times of these benchmarks on a four-node SP system. The benchmarks were executed several times. The best execution time for each application was recorded.

The MPI-LAPI performs consistently better than the native MPI. Improvements of 1.9%, 4.1%, 4.6%, 5.1% and 13.8% were obtained for LU, IS, CG, BT and FT benchmarks, respectively. The percentages of improvement for EP, MG, and SP were less than 1.0%.

7 Related Work

Previous work on implementing MPI on top of low-level one-sided communication interfaces include (a) the effort at Cornell in porting MPICH on top of their GAM (generic active message) implementation on SP systems [2], and (b) the effort at University of Illinois in porting MPICH on top of the FM (fast messages) communication interface on a workstation cluster connected with the Myrinet network [5]. In both cases the public domain version of MPI (MPICH [3]) has been the starting point of these implementations. In the MPI implementation on top of AM, short messages are copied into a retransmission buffer after they are injected into the network. Lost messages are retransmitted from the retransmission buffers. The retransmission buffers are freed when a corresponding acknowledged is received from the target. Short messages therefore require a copy at the sender side. The other problem is that for each pair of nodes in the system a buffer should be allocated which limits scalability of the protocol. MPI-LAPI implementation avoids these problems (which degrade the performance) by using the header handler feature of LAPI. Unlike MPI-LAPI, the implementation of MPI on AM described in [2] does not support packet arrival interrupts which impacts performance of applications with communication behavior that is asynchronous. In the implementation of MPI on top of FM [5], FM was modified to avoid extra copying at the sender side (gather) as well as the receive side(upcall). FM has been optimized for short messages. Therefore, for long messages (larger than $2K$ bytes) MPI-FM performs poorly in comparison with the native implementation of MPI on SP systems (Fig. 10 of [5]).

8 Conclusion Remarks

In this paper, we have presented how the MPI standard is implemented on top of LAPI for SP systems. The details of this implementation and the mismatches between the MPI standard requirements and LAPI functionality have been discussed. We have also shown how LAPI can be enhanced in order to make the MPI implementation more efficient. The flexibility provided by having header handlers and completion handlers makes it possible to avoid any unnecessary data copies. The performance of MPI-LAPI is shown to be very close to that of bare LAPI and the cost added because of the MPI standard semantics enforcement is shown to be minimal. MPI-LAPI performs comparably or better than the native MPI in terms of latency and bandwidth. We plan to implement MPI data types which have not been implemented yet.

Acknowledgements

We would like to thank several members of the CSS team: William Tuel and Robert Straub for their help in providing us details of the MPCI layer, and Dick Treumann for helping with the details of the MPI layer. We would also like to thank Kevin Gildea, M. T. Raghunath, Gautam Shah, Paul DiNicola, and Chulho Kim from the LAPI team for their input and the early discussion which helped motivate this project.

Disclaimer

MPI-LAPI is not a part of any IBM product and no assumptions should be made regarding its availability as a product. The performance results quoted in this paper are from measurements done in August of 1998 and the system is continuously being tuned for improved performance.

References

[1] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M.Snir. SP2 System Architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[2] C. Chang, G. Czajkowski, C. Hawblitzel, and T. V. Eicken. Low Latency Communication on the IBM RISC System/6000 SP. *Supercomputing 96*, 1996.

[3] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

[4] IBM. *PSSP Command and Technical Reference - LAPI Chapter*. IBM, 1997.

[5] M. Lauria and A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, pages 4–18, Jan 1997.

[6] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[7] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and Experience with LAPI - a New High-Performance Communication Library for the IBM RS/6000 SP. In *Proceedings of the International Parallel Processing Symposium*, pages 260–267, March 1998.

[8] M. Snir, P. Hochschild, D. D. Frye, and K. J. Gildea. The communication software and parallel environment of the IBM SP2. *IBM Systems Journal*, 34(2):205–221, 1995.

[9] C. B. Stunkel, D. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao. The SP1 High Performance Switch. In *Scalable High Performance Computing Conference*, pages 150–157, 1994.

[10] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266, 1992.