Message Passing Interface for RS/6000 SP

By Xianneng Shen, Eddie Ho, and Mike Hammill

Message Passing Interface (MPI) provides an efficient, portable, standardized interface to implement parallel programs for RS/6000 SP. This is a different programming model than the single OS environment for AIX. This article covers the basic concepts of MPI, provides some sample programs, and discusses MPI point-to-point and collective communication routines.

An RS/6000® Scalable POWERParallel™ (SP) system, a scalable parallel computer, is the high-end system of the RS/6000 product family, which ranges from laptop, desktop, workgroup server, enterprise server, to super computer using the share-nothing architecture. AIX® is the operating system used across the entire product family.

AIX provides binary compatibility for applications running on a single SP node and applications running on the rest of the RS/6000 family. The SP is a shared-nothing system of separated, full-function nodes connected by a high-performance switching network. Each node is a stand-alone RS/6000 workstation running AIX. An explicit message-passing mechanism allows communication between nodes.

The SP provides both traditional Internet Protocol (IP) and user space communication protocols. Because the user application has direct access to the SP switch adapter fabric, the user space protocol is very fast. Currently only one user application process is allowed per processor at one time in the user space protocol. A Message Passing Interface (MPI) code can be run using either IP or user space protocol on a set of uniprocessor nodes since the Symmetric Multiprocessor (SMP) node does not yet have MPI support. In this article, the terms "processor" and "node" are interchangeable.

MPI Overview

MPI is a message-passing interface that is formally specified, portable, and based on standards. The primary objective of MPI is to give programmers an efficient, portable, standard message-passing library with rich functionality. The MPI standard, which uses the most attractive features of several existing message-passing systems, was introduced at Supercomputing '93 and finalized in May 1994. Software developers have now begun implementing this standard on many platforms.

There are many reasons to use MPI, but the four most important considerations are standardization, portability, performance, and richness.

Figure 1 illustrates the basic node-to-node communication paradigm.

Basic MPI Programming and Concepts

As in other message-passing environments, you can insert calls to MPI routines in either C or FORTRAN code. Six basic MPI routines can be found in most MPI code:

- Initialize MPI by calling `MPI_INIT` at the beginning of an application
- Terminate MPI by calling `MPI_FINALIZE` at the end of the application
- Identify a particular process in your application by calling `MPI_COMM_RANK`
- Determine the total number of processors for the application by calling `MPI_COMM_SIZE`
- Send messages by calling `MPI_SEND`
- Receive messages by calling `MPI_RECV`

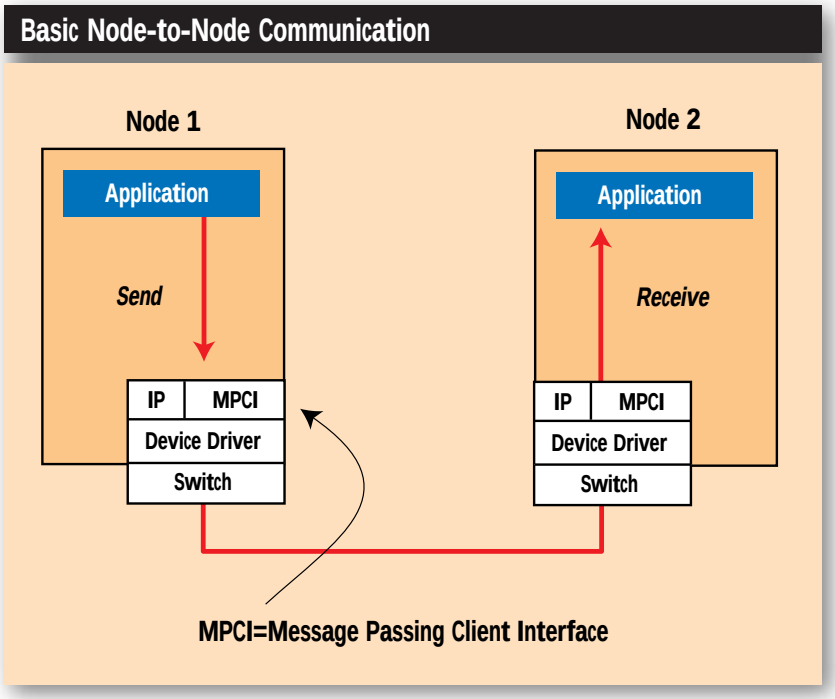

Figure 1. Node-to-node communication

```
#include <stddef.h>
#include <stdlib.h>
#include "mpi.h"
main(int argc, char **argv )
{
        char message[20];
        int i,rank, size, type=99;
        MPI_Status status;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        if(rank == 0) {
           strcpy(message, "Hello, world");
           for (I=1; i<size; I++)
           MPI_Send(message, 13, MPI_CHAR, I,
               type, MPI_COMM_WORLD);
        }
        else
          MPI_Recv(message, 20, MPI_CHAR, 0,
               type, MPI_COMM_WORLD, &status);

        printf( "Message from node =%d : %.13s\n", rank,message);
        MPI_Finalize();
}
```

Figure 2. Basic MPI routines

Figure 1 shows the six basic MPI routines described above.

In Figure 2, process zero (rank == 0) sends a message to the rest of the processes (rank != 0) in the communication world (described below). After receiving the message sent by processor 0, each processor prints its rank and the message `hello world`.

This simple example illustrates some fundamental MPI concepts including the MPI message. An MPI message contains the message and the message envelope. The message consists of the data, count, and datatype; the message envelope includes source (or destination) process rank (integer), message tag (integer), and communicator.

Communication context and the process group represent the minimum information found in a communicator. The communication context can be viewed as a hidden system tag for extra protection. The process group is a set of processes whose members are identified by their rank.

MPI Point-to-Point Communication

The most elementary form of message-passing communication involves two processes: one passing a message to the other. Although there are several ways that this might happen in hardware, logically, the communication is point-to-point: one process calls a `send` routine and the other calls a `receive` routine.

MPI Blocking Send / Receive

As with most existing message-passing systems today, MPI provides blocking as well as nonblocking send and receive. Figure 3 shows a simple send and receive flow of large messages between two nodes. Figure 4 shows the programming syntax of blocking `send`.

The blocking `send` call does not return until the message has been safely stored away so that the sender can freely reuse the send buffer. Figure 5 shows the syntax of the blocking `receive` routine.

The blocking `receive` does not return until the message has been stored in the receive buffer.

Nonblocking Send and Receive

Performance on many systems can be improved by overlapping communication and computation. One

way to achieve that is to use nonblocking communication.

A nonblocking `send` call initiates the send operation, but does not complete it. The nonblocking `send race` call will return before the message is copied from the send buffer. A separate call `mpi_wait` or `mpi_test` is needed to complete the communication—to verify that the data was copied from the send buffer.

Figure 6 shows an example message flow for nonblocking communications. Figure 7 shows the nonblocking send syntax.

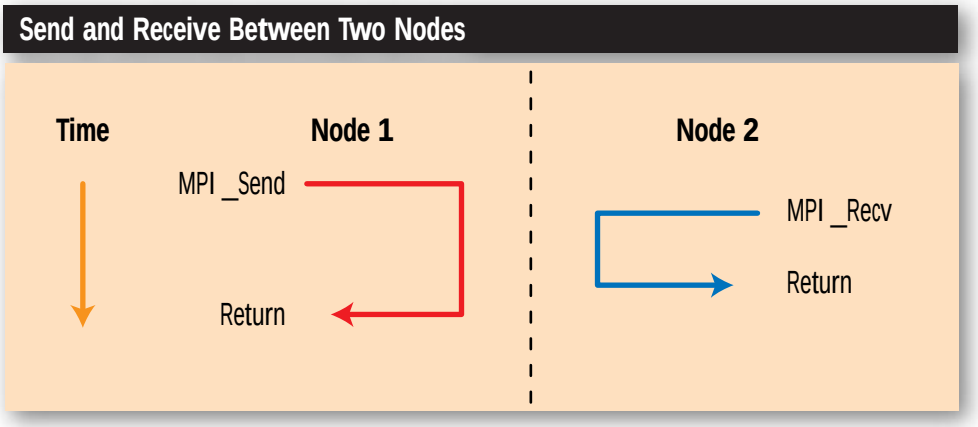

Figure 3. Send and receive between two nodes

```
MPI_Send(void* buf, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm);
```

`buf`	initial address of the send buffer (IN)
`count`	number of items in the send buffer (IN)
`datatype`	datatype of each send buffer item (handle) (IN)
`dest`	rank of the destination process in comm (IN)
`tag`	message tag (integer) (IN)
`comm`	communicator (handle) (IN)

Figure 4. Blocking send syntax

```
MPI_Recv(void* buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm,
        MPI_Status *status);
```

`buf`	initial address of the receive buffer (OUT)
`count`	number of items to be received (integer) (IN)
`datatype`	datatype of each receive buffer item (handle) (IN)
`source`	rank of the source process in comm or `MPI_ANY_SOURCE` (integer) (IN)
`tag`	message tag or `MPI_ANY_TAG` (integer) (IN)
`comm`	communicator (handle) (IN)
`status`	status object (status) (OUT)

Figure 5. Blocking receive syntax

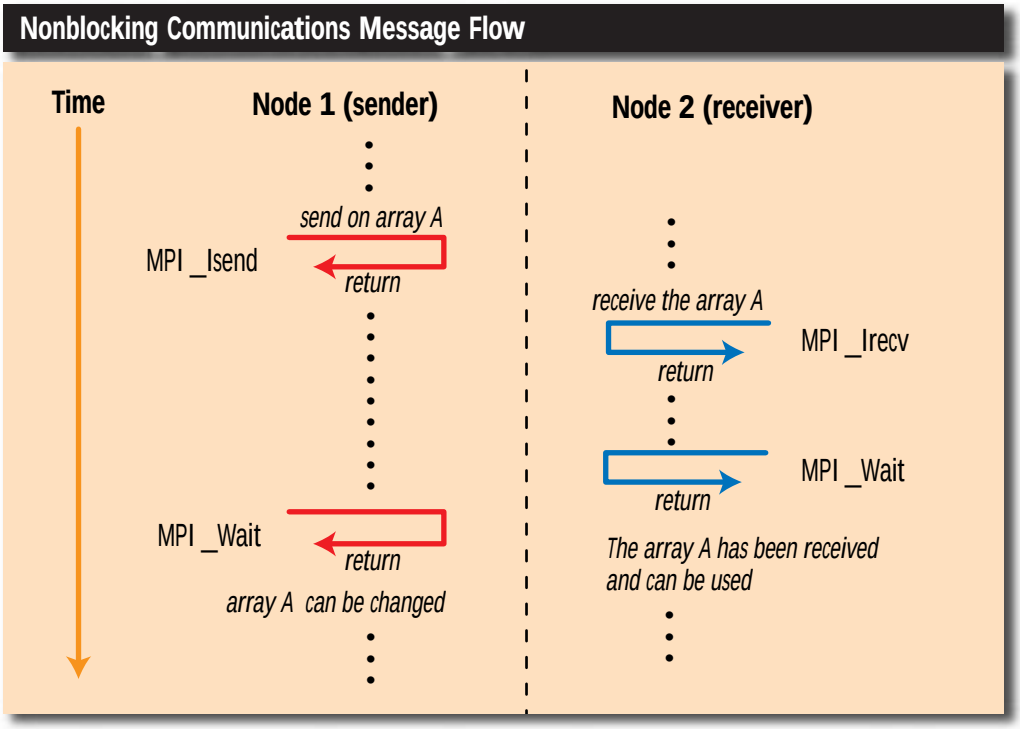

Figure 6. Nonblocking message flow

```
MPI_Isend(void* buf, int count, MPI_Datatype datatype,
     int dest, int tag, MPI_Comm comm,
     MPI_Request *request);
```

buf	initial address of the send buffer (IN)
count	number of items in the send buffer (IN)
datatype	datatype of each send buffer item (handle) (IN)
dest	rank of the destination process in comm (IN)
tag	message tag (integer) (IN)
comm	communicator (handle) (IN)
request	communication request (handle) (OUT)

Figure 7. Nonblocking send syntax

```
MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm,
        MPI_Request *request);
```

buf	initial address of the receive buffer (OUT)
count	number of items to be received (integer) (IN)
datatype	datatype of each receive buffer item (handle) (IN)
source	rank of the source process in comm or MPI_ANY_SOURCE (integer) (IN)
tag	message tag or MPI_ANY_TAG (integer) (IN)
comm	communicator (handle) (IN)

Figure 8. Nonblocking receive syntax

If the nonblocking `send` is called, the system may start copying data from the send buffer. The sender should not access any part of the send buffer after a nonblocking send operation is called until `send` completes.

Similarly, a nonblocking `receive` call initiates the receive operation, but does not complete it. The call will return before a message is stored in the receive buffer. An `MPI_Wait` call must complete the receive operation. Figure 8 shows the nonblocking receive syntax.

The receive buffer stores count consecutive elements of the type specified by datatype, starting at the address in buf. The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, in the receive buffer.

To check the status of a nonblocking `send` or `receive`, call `MPI_Test` or to complete a nonblocking send or receive, call `MPI_Wait`.

Collective Communication

Collective communication involves all the group processes. You can build your own collective communication routines, but it may involve a lot of work and may not be efficient. Although other message-passing libraries provide some collective communication calls, none provides a set of collective communication routines as complete and robust as those provided by MPI.

MPI collective communication can be divided into three subsets:

- Barrier synchronizations
- Data movements
- Reduction computation

Barrier Synchronizations

In almost all parallel applications, explicit or implicit synchronization is required. As with other message-passing libraries, MPI provides a function call, `MPI_Barrier` (`MPI_Comm comm`), to synchronize all processes within a communicator.

Data Movements

MPI provides a set of useful data movement routines, such as `broadcast`, `gather`, `scatter`, and `alltoall`. In many cases, one process needs to send (`broadcast`) some data to all the processes under the same communicator. Each process must call `MPI_Bcast` (`void* buf`, `int count`, `MPI_Datatype datatype`, `int root`, `MPI_Comm comm`), where the process sending the data is called root. Figure 9 shows a broadcast operation for a 4-node system.

If you have an array scattered throughout all processes in the group, and you want to collect each piece of the array into a specified process, the function to use is `GATHER`. MPI provides `MPI_GATHER`. The `MPI_SCATTER` is the counterpart of `MPI_GATHER`. Figure 10 shows the scatter and gather operation; Figure 11 shows the syntax.

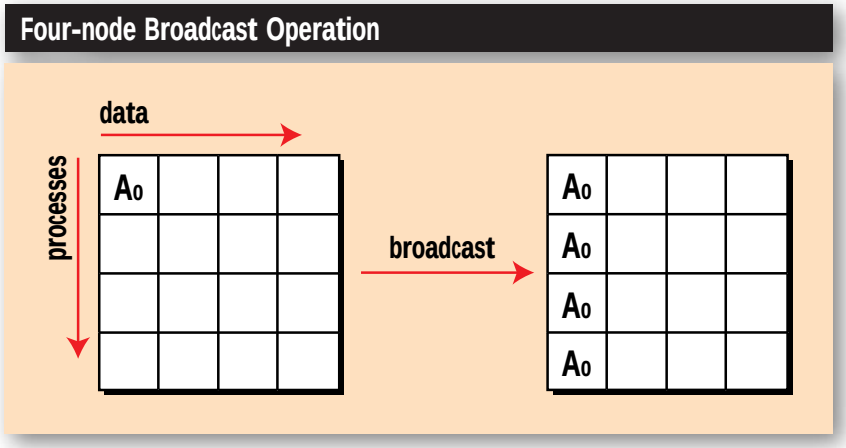

Figure 9. Broadcast operation for a 4-node system

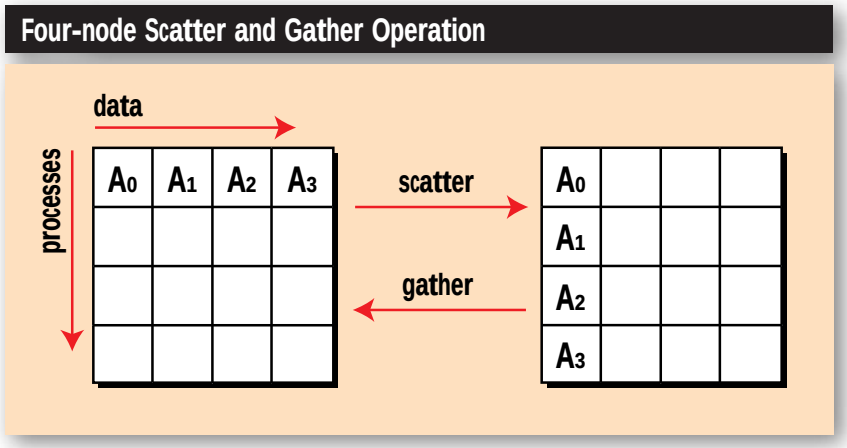

Figure 10. Scatter and gather operation for a 4-node system with four data types

```
int MPI_Gather(void* sbuf, int scount, MPI_Datatype sdatatype, void* rbuf,
   int rcount, MPI_Datatype rdatatype, int root, MPI_Comm comm)

int MPI_Scatter(void* sbuf, int scount, MPI_Datatype sdatatype, void* rbuf,
   int rcount, MPI_Datatype rdatatype, int root, MPI_Comm comm)
```

Figure 11. Scatter and gather syntax

MPI provides a vector version of MPI_Gather and MPI_Scatter routines: MPI_Gatherv and MPI_Scatterv.

An MPI_ALLTOALL call is helpful in applications such as matrix transpose and FFT. Figure 13 shows its syntax.

Reduction Routines

One of the most useful collective operations is global reductions, or combine operations. The partial result in each process in the group is combined into one specified process or all the processes using some desired basic functions. Figure 14 shows the form of each of the two global reduction primitives.

MPI has predefined operations, such as MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, and MPI_MINLOC. See the MPI standard documentation for these operations.

A scan, or prefix-reduction operation, performs partial reductions on distributed data. The scan operation returns the reduction of the data in the send buffers of nodes ranked 0,1,2,...,*n* in the receive buffer of the node ranked *n*. Figure 15 shows the syntax.

Conclusion

The message-passing programming model closely matches the SP shared-nothing architecture. The

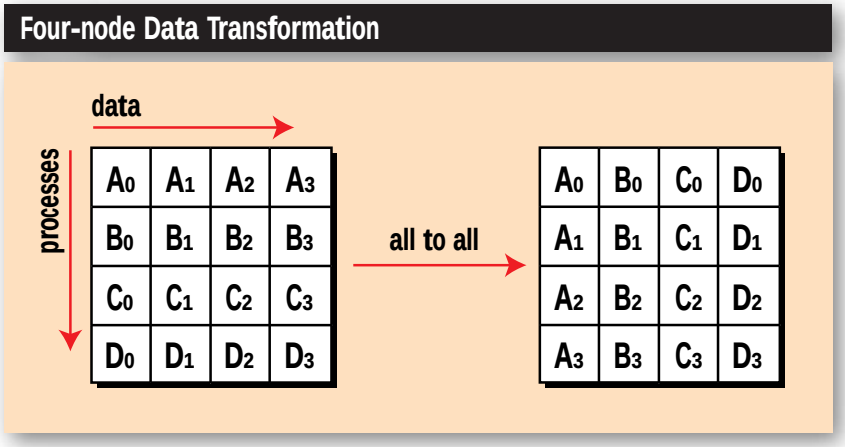

Figure 12. Data transformation on a 4-node system with multiple data types

```
int MPI_Alltoall(void* sbuf, int scount, MPI_Datatype sdatatype,
   void* rbuf, int rcount, MPI_Datatype rdatatype, MPI_Comm comm)
```

Figure 13. Syntax for a data transformation

```
int MPI_Reduce(void* sbuf, void* rbuf, int count, MPI_Datatype datatype,
 MPI_Op op, int root, MPI_Comm comm)

int MPI_Allreduce(void* sbuf, void* rbuf, int count, MPI_Datatype datatype,
   MPI_Op op, MPI_Comm comm)
```

Figure 14. Syntax for a prefix-reduction operation

```
 int MPI_Scan(void* sbuf, void* rbuf, int count, MPI_Datatype
    datatype, MPI_Op op, MPI_Comm comm)
```

Figure 15. MPI scan

message-passing program running on an SP system can achieve the highest possible performance. MPI becomes the choice of the message-passing library because of its standardization, portability, high performance, and rich functionally. Moreover, MPI continues to be improved as more functions are added. However, MPI will be backward compatible, which will enable an MPI application program developed using MPI today to run on an MPI implementation in the future.

Reference

IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference Version 2, Release 1 (GC23-3894-00).

Message Passing Interface Forum. MPI: A Message Passing Interface Standard. June 12, 1995.

IBM Parallel Environment for AIX: Operation and Use Version 2.1.0. (GC23-3891-00).

Gropp, William; Lusk, Ewing; and Skjellum, Anthony. *Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

Xianneng Shen, *IBM Corporation, 522 South Road, Poughkeepsie, NY 12601. E-mail: xshen@vnet.ibm.com. Dr. Shen is a senior marketing support representative in the RS/6000 Executive Briefing Center. He has a BS and an MS in Electrical Engineering from the University of Electronic Science and Technology of China, an MS in Computer Engineering from Syracuse University, and PhD in Electrical Engineering from Syracuse University.*

Eddie Ho, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. E-mail: eddieho@vnet.ibm.com. Mr. Ho is a programming consultant in the RS/6000 Executive Briefing Center. He has a BS in Computer Science from the University of Wisconsin and an MS in Computer Science from North Dakota State University.*

Mike Hammill, *Cornell Theory Center, Ithaca, NY 14853. E-mail: mhammill@tc.cornell.edu. Mr. Hammill is the academic outreach coordinator. Mr. Hammill has a BS in Computer Science, a BA in Mathematics, and a BA in Philosophy from the University of Iowa.*