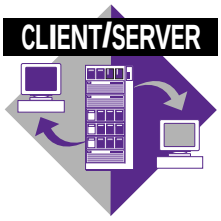
Message Passing on the RS/6000 SP



By Xianneng Shen and David Klepacki

This article is the second in a series about parallel programming models on the RS/6000 SP system. The focus is on the message-passing model and its features. In particular, unilateral (one-sided) communication is compared and contrasted to bilateral (two-sided) communication.

The first article[1] in this series briefly introduced the IBM RS/6000 SP architecture and its parallel programming models. Basic functions of the Message Passing Interface (MPI) standard were presented in another recent article[2]. Here, we discuss additional aspects of the message passing programming model as implemented on the RS/6000 SP.

Message-passing programming on the RS/6000 SP is performed with the IBM Parallel Environment (PE) product for AIX. PE allows users to develop, profile, debug, analyze, tune, trace, and execute parallel application programs.

PE supports two data communication protocols: the TCP/IP protocol and the User Space protocol. The TCP/IP protocol is the industry standard most commonly used in internetworking computers. The User Space Protocol (USP) is unique to the RS/6000 SP and provides a high-bandwidth and low-latency communication path to applications running over the high-performance SP switch network. Hence, USP is only meaningful if there is a high-performance SP switch available in the system (the SP switch is an optional hardware component of an RS/6000 SP).

USP allows applications to take full advantage of the RS/6000 SP switch communication network and offers the highest performance. However, this protocol cannot be used by more than one process per node (that is, per switch adapter) at a given time. On the other hand, the TCP/IP protocol has a different set of limitations. It can support multiple parallel applications running simultaneously on the same set of processing nodes, but at lower communication performance relative to using USP.

The MPI interface is a parallel programming library that specifies the functions and subroutine calls to be used from within the Fortran and C languages. Its specification is an international standard with a goal of not compromising efficiency, portability, and functionality for any given computer architecture or vendor. The IBM implementation of the MPI interface constitutes a library component of the PE product. It is important to remember that the protocols used for interprocessor communication are not part of the MPI specification. The IBM PE and its USP design are features that allow MPI-based applications to take full

[1]Klepacki, David and Shen, Xianneng. "Parallel Programming Models on the IBM RS/6000 SP." *AIXpert*, September 1997.
[2]Shen, Xianneng; Ho, Eddie; and Hammill, Mike. "Message Passing Interface for RS/6000 SP."*AIXpert*, March 1997.

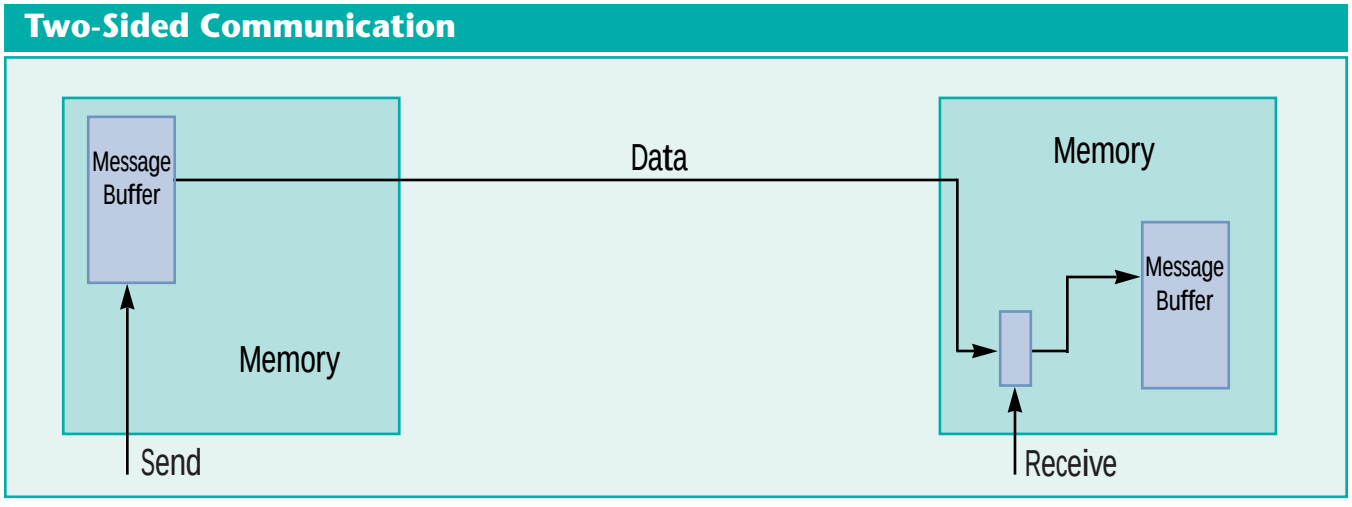


Figure 1. Send and receive

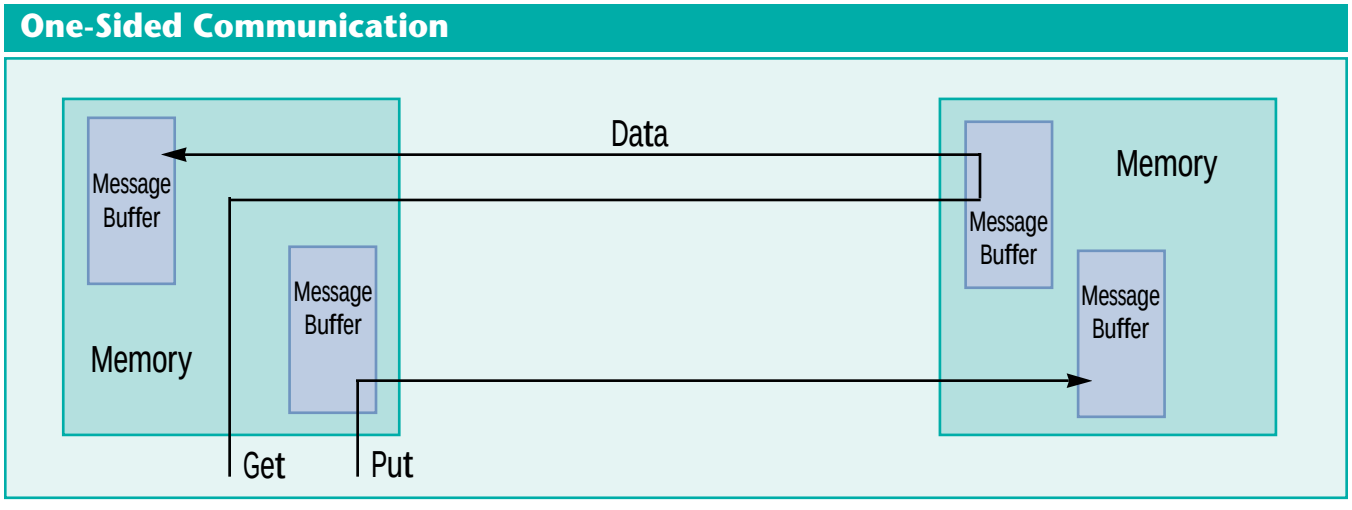


Figure 2. Put and get (one-sided communication)

advantage of the hardware capabilities of the RS/6000 SP.

MPI Enhancement

The message-passing programming model is simply described as a collection of processes that communicate among themselves using messages (that is, a sequence of bytes). The communication can be either unilateral (one-sided) or bilateral (two-sided). Bilateral communication consists of one process which initiates an operation (for example, `send`) and another process which completes the communication with a complementary operation (for example, `receive`). In fact, bilateral message passing is often referred to as a "send-receive" model (see Figure 1). In contrast, unilateral communication does not require a second process to take a complementary action. It is often referred to as a "put-get" model, whereby a `put` operation pushes messages and a `get` operation pulls messages to or from a processor's memory (see Figure 2).

In principle, a message-passing program requires nothing beyond the use of the basic `send-receive` or `put-get` functions. However, the MPI standard offers a rich set

of programming features to enhance the power and ease-of-use of message passing. In particular, a set of collective operations is defined which can be applied to user-defined groups of processes. These operations include various data movement operations (for example, broadcast, scatter, gather, all-to-all, all-to-one) as well as collective computation operations (that is, "global" sum, minimum, maximum, and so on). Additional features of MPI include virtual process topologies (such as graphs and Cartesian grids), asynchronous communication modes, and derived datatypes. Altogether, there are approximately 150 functions defined in the current MPI specification.

The MPI specification is now in its second release and is appropriately referred to as MPI-2. MPI-2 incorporates many new functions and enhancements based upon the experience gained from the first specification. The additional features can be classified into four areas: dynamic task management, unilateral communication, a parallel I/O interface, and miscellaneous functions.

Dynamic task management allows a parallel application to alter the number of concurrent processes during the course of its execution. In MPI-1, the number of concurrent processes is fixed at runtime. With MPI-2 on the RS/6000 SP, scalability can be controlled since the number of actual nodes can change to accommodate the change in requested processes. This feature allows independent MPI jobs to interact concurrently as well.

The unilateral communication features of the MPI-2 are very similar to the `put-get` semantics described above with the additional caveat that synchronization is explicitly required.

The parallel I/O interface specification, aptly called MPI-IO, introduces a portable method of efficiently working with various parallel file systems. It allows multiple processes to collectively operate on parallel files in a seamless fashion, and be able to take full advantage of MPI-related efficiencies (for example, MPI-derived datatypes).

Lastly, the miscellaneous category of the MPI-2 includes features such as non-blocking collective communication operations and the handling of external interfaces.

> ***The message-passing programming model is a collection of processes that communicate among themselves using messages, that is, a sequence of bytes.***

IBM Unilateral Communication

In addition to MPI, IBM offers a separate library for unilateral communication, called the Low-level Application Programming Interface (LAPI). LAPI is completely independent of MPI, but can coexist with MPI in the same application. Its purpose is to provide a high-performance alternative to the specification given by MPI-2. This performance improvement can be obtained by relaxing some of the MPI-2 requirements that fully support the MPI message-passing semantics. For portability purposes, the programmer can easily substitute the corresponding MPI-2 functions. LAPI is designed for use by libraries and power programmers for whom performance is more important than code portability.

The LAPI library provides `PUT` and `GET` functions and a general active message function that allows programmers to supply extensions by means of additions to the message notification handlers. LAPI has the following advantages over similar programming interfaces (like MPI):

- **Performance:** LAPI is designed to especially provide low latency on short messages, low interrupt latency, and high bandwidth.
- **Flexibility:** LAPI provides a more primitive interface (than either MPI or TCP/IP) to the SP switch that coexists with the standard communications protocols.

- **Extendibility:** LAPI supports programmer-defined handlers that are invoked when a message arrives. Programmers can customize LAPI to their specific environments.

Some other general characteristics of LAPI include the following:

- Reliability (LAPI provides guaranteed delivery of messages. Errors not directly related to the application are not propagated back to the application.)
- Flow control
- Support for large messages
- Non-blocking calls
- Interrupt and polling modes
- Efficient exploitation of switch function
- By default, ordering of messages is not guaranteed.

LAPI functions (see Figure 3) can be divided into three categories:

1. A basic active message infrastructure that allows programmers to install a set of handlers that is invoked and executed in the address space of a target process on behalf of the process originating the active message. This interface has enhanced capabilities for user-written handlers that allow decoupling of the tasks for data transfer, incorporating incoming data into ongoing computation and synchronization. This generic interface allows programmers to customize the LAPI function to their unique environments.

2. A set of defined functions that is complete enough to satisfy the requirements of most programmers. These defined functions make LAPI more usable and, at the same time, lend themselves to efficient implementation because their syntax and semantics are well-known.

Operations	Functions
Setup functions	`LAPI_Init and LAPI_Term`
Active Message function	`LAPI_Amsend`
Data Transfer functions	`LAPI_Put, LAPI_Get`
Synchronizing functions	`LAPI_Rmw`
Signaling functions	`LAPI_Setcntr, LAPI_Waitcntr, LAPI_Getcntr`
Ordering functions	`LAPI_Fence, LAPI_Gfence`
Address Exchange function	`LAPI_Address_init`
Environment functions	`LAPI_Qenv, LAPI_Senv`

Figure 3. LAPI functions

3. A set of control functions for the initialization and eventual orderly shutdown of LAPI and to query the state of the LAPI subsystem.

Conclusion

A few words should be said about thread-based library implementations. With the advent of Symmetric Multiprocessor (SMP) nodes on the RS/6000 SP (that is, high nodes), it becomes necessary to support threaded application code with libraries that are thread safe. This means that one thread of execution will not interfere with the operation of another thread. For instance, when dynamically allocating storage with `malloc()`, a non-thread-safe library could not guarantee that two threads would not end up allocating the same storage locations and writing over one another; a thread-safe library prevents this from happening.

Both LAPI and the IBM implementation of MPI provide thread-safe libraries. However, a thread-safe library does not guarantee a thread-safe application, and the programmer must still apply thread-safe techniques when developing parallel programs.

Acknowledgments

The authors would like to thank the following developers who took the time to explain the richness of their work: Paul DiNicola, Rama Govindaraju, Chulho Kim, Gautam

Shah, Jamshed Mirza, Carl Bender, Kevin Gildea, and Richard Treumann.

Xianneng Shen, *IBM Corporation, 522 South Road, Poughkeepsie, NY 12601. E-mail: xshen@us.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.*

David Klepacki, *IBM Corporation, 522 South Road, Poughkeepsie, NY 12601. Dr. Klepacki has been working in IBM's POWERparallel Systems Group since its beginning in 1991 as a computational physicist and scientific applications specialist with emphasis on performance benchmarking. Today, in addition to his technical endeavors, he also manages the parallel software tools segment for the technical marketing branch of the RS/6000 Division. Dr. Klepacki's current interests include performance programming, scalable parallel algorithms, scalable I/O, and portable high-performance computing tools. He holds a PhD in Theoretical Nuclear Physics from Purdue University as well as an MS in Electrical Engineering from Syracuse University.*