The communication software and parallel environment of the IBM SP2

by M. Snir
P. Hochschild
D. D. Frye
K. J. Gildea

This paper describes the software available on the IBM SP2™ for parallel program development and execution. It presents the rationale for the design of the Message-Passing Library used on the SP2, outlines its current implementation, and gives information on performance. In addition, the paper describes the programming environment and the program development tools available for developing and executing parallel codes.

The IBM scalable POWERparallel system* 9076 SP2* was designed to run large-scale parallel applications efficiently. A critical aspect of parallel program support is communication: Efficient parallel computing requires high-bandwidth, low-latency interprocessor communication. The SP2 High-Performance Switch and adapter, described elsewhere in this issue,[1] provide hardware support for high-performance communication. In this paper, we describe the communication software that allows parallel applications to exploit the performance characteristic of the communication hardware.

The main parallel programming model supported by the SP2 is message passing: A set of tasks, each executing in its own address space, communicates via calls to the Message-Passing Library (MPL). This library was designed so as to provide programming convenience. For example, MPL supports a fairly extensive set of communication calls for collective communication. This capability alleviates the need to program in detail the communication for a scatter-gather or transpose operation. Conversely, MPL has a fairly small number of calls that can be implemented to map efficiently to the underlying communication hardware. In the next section we present the rationale for the design of MPL and describe the services provided by the main functions in MPL.

The implementation of a message-passing library offers a multitude of alternatives, especially on an architecture as rich as the SP2: some functions supported by microcode on the adapter and some by software on the computing processor; some functions executed in user space and some by kernel; trade-offs between more extensive use of buffering and data copying and more eager use of interrupts; "push" versus "pull" protocols; flow control; etc. We describe the current implementation of MPL, its performance, and the rationale for some of the decisions taken. This implementation achieves a performance that is close to the hardware limitations, although improvements are still possible.

Parallel code is more complex to develop than sequential code. To a large extent, the complexity is a reflection of the lower level of maturity of parallel software technology. Parallel languages and compilers have a shorter history and a much smaller investment than languages and compilers for uniprocessors. Beyond that, parallel computing is intrinsically harder than sequential computing. It requires the user to understand the performance impact of at least one added dimension of the execution model, namely parallelism. Therefore, it is important to provide users with a rich set of tools that will help them to understand the behavior of parallel code, debug it, and tune it. Such tools should provide information not only about the individual behavior of each process but also about the interactions and correlations between processes. In the last section of this paper we describe the Parallel Operating Environment available on the SP2. This environment includes facilities for compilation and parallel program submission, a source level parallel debugger, and a trace-driven performance visualization tool.

Message-passing libraries are the main programming interface used on distributed memory machines and networks of workstations.

Finally, we would like to emphasize that this paper does not provide an exhaustive description of the various programming models and services that can be used to develop parallel applications on the SP2. The SP2 is an open system that can be used in a variety of modes: processes can communicate using standard UNIX** interprocess communication mechanisms, such as Internet Protocol (IP), User Datagram Protocol (UDP), Transmission Control Protocol (TCP), and sockets; other message-passing libraries, such as PVM[2] (Parallel Virtual Machine), are supported; third-party vendors provide support for other parallel programming paradigms, such as LINDA[3] or High Performance FORTRAN;[4] various users of the SP1* and SP2 have implemented their own communication libraries or tools. In addition, the SP2 provides a variety of system services in support of parallel applications, such as parallel job management or parallel I/O operations.[5,6]

The Message-Passing Library

Message-passing libraries are the main programming interface used on distributed memory machines and networks of workstations. Such libraries support a programming model in which a parallel program consists of a number of tasks, each running a single execution thread within its own address space (i.e., executing a sequential FORTRAN or C program). Tasks communicate with explicit calls to a message-passing library.

At the time development started on the SP1 Parallel Operating Environment there was no accepted standard for such libraries: parallel system vendors and third-party software vendors supported proprietary, incompatible libraries.[7–9] (Since then, a standard Message-Passing Interface [MPI] has been designed by an open forum of vendors, customers, and researchers.[10] We plan to support this MPI in future software releases.) PVM[2] was the most popular public-domain library used on networks of workstations. However, PVM looked less attractive as an interface for a machine like the SP1. The design of PVM, which is optimized for IP communication in a networked environment, requires data copying operations that can be avoided with a library more directly targeted to an SP environment. Also, PVM (Version 2) was missing support for important functions, such as collective communication. Therefore, we decided to develop our own Message-Passing Interface for the SP1. This library has become known under two equally unimaginative names: External User Interface (EUI)[11] and Message-Passing Library (MPL).[12,13]

The design of EUI/MPL was influenced by several prototype systems and environments developed at IBM Research, in particular the Vulcan Operating Environment[14] and the Venus communication library.[15] The library was defined by a joint team at IBM Research and the IBM Highly Parallel Supercomputing Systems Laboratory (HPSSL).[11] The final specification of the library was completed in the summer of 1992. By the fall of 1992, the first implementations of EUI were operational on several platforms. These platforms included a prototype of the SP1 (demonstrated at the Supercomputing-92 conference) and clusters of RISC System/6000* work-

stations. The library is available as part of the SP1 and SP2 Parallel Operating Environment.

The design of EUI/MPL tried to balance several partially conflicting requirements. We wanted a library that would provide a convenient application programming interface. We wanted the library to be small, both for ease of use and for fast implementation, yet complete. We wanted nearly identical calls to be available both from FORTRAN and from C. We wanted calls that would be familiar to users of existing libraries. Most importantly, we wanted a library that can be well supported by the SP1 and SP2 hardware. In particular, we wanted a design that would allow the off-loading of communication to the adapter and allow overlap of communication and computation. This off-loading would not be possible on the SP1, as the High-Performance Switch adapter for the SP1 has no ability to move data on its own. However, we anticipated such ability in the High-Performance Switch adapter for the SP2.

We assumed that the main mechanism for allocating resources to parallel jobs would be space partitioning. In this model a parallel job has full control of a set of processors that is kept fixed during the run. This model is preferred by many users and simplifies resource allocation. A programming model in which a parallel job consists of a fixed number of tasks fits well with fixed physical partitions. It also simplifies the design of the communication subsystem and ultimately improves performance, because resources can be bound at load time, rather than dynamically. Therefore, we focused on supporting parallel applications that use a fixed number of tasks.

We ended up with a library that includes 33 functions. These functions provide three types of services:

- Task management for initialization, termination, and environment setting and querying
- Point-to-point communication for communication between pairs of tasks
- Collective communication for communication and synchronization operations that involve groups of tasks

The following sections describe these services in further detail.

Task management. Four routines are provided for task management and are described below.

The number *numtask* of tasks used by a parallel program (the partition size) is specified when the program is submitted for execution. During execution, each task is identified by an integer *taskid*, in the range 0 to numtask −1. The procedure MP_ENVIRON allows the user to find the total number of tasks and the taskid of the calling task. (We are using the FORTRAN names of the message-passing calls. For each FORTRAN procedure MP_xxx, there is a corresponding C function named mpc_xxx.)

Pairs of tasks communicate by issuing matching send and receive commands.

The procedures MP_TASK_QUERY and MP_TASK_SET are used to query system-dependent execution parameters and, respectively, to set user-definable execution parameters.

The procedure MP_STOPALL allows a task to abort execution of all tasks and is used for abnormal termination. Normal termination occurs when each task exits normally (e.g., via a STOP or END statement in FORTRAN).

Point-to-point communication. Pairs of tasks communicate by issuing matching send and receive commands. A send command specifies the location in memory of the data to be sent and the taskid of the destination. In addition, the send command includes a *type* parameter that can be used to identify the message. The receive command specifies where the message should be placed, as well as the desired source and type of message. Either or both of the source and type values may be a wild card on a receive (thus matching any message). Message truncation is supported, unless the receiving process is set to treat overflow as an error condition. Both blocking and nonblocking send and receive operations are supported in the MPL.

A *blocking send* call returns after the application buffer in the sending task is free to be reused; completion of this call does not imply that the message

has been received into the application buffer in the destination task. A *blocking receive* call returns after the receive operation completes and the message has been copied into the application buffer of the receiving task.

The two procedures MP_BSEND and MP_BRECV are used for blocking sends and receives. These two procedures support communication from a contiguous memory area, specified by an initial address and a byte length.

The two procedures MP_BVSEND and MP_BVRECV allow the user to send a message from noncontiguous memory (and, respectively, receive it in noncontiguous memory). The memory buffer is specified by four parameters: initial address, number of (contiguous) blocks, the size of each block, and the offset between successive blocks. Use of such a strided buffer can move an arbitrary submatrix of a two-dimensional program array with one communication call, in particular, a row of a FORTRAN array or a column of a C array. This method saves an additional memory-to-memory copy operation that would be otherwise required to pack the data into a contiguous buffer before they are sent and unpack the data when received.

A *nonblocking send* call MP_SEND just notifies the system that a message must be sent and returns without waiting for the message to be copied out of the user application buffer. As a result, the user must not overwrite the application buffer until the message has been copied by the system. Similarly, a *nonblocking receive* call MP_RECV just notifies the system that a particular application buffer is available for receiving a message and returns without waiting for the message to arrive. As a result, the user must check for the reception of the message before accessing it in the application buffer. To allow this, nonblocking send and receive calls return an integer *msgid* that identifies the pending communication operation and can be used to monitor its progress. The nonblocking procedure call MP_STATUS reports the status of a pending message. The call MP_WAIT blocks the calling task until the specified communication has completed.

The use of nonblocking communication calls results in code that is more complex, because each communication operation has to be coded as two separate calls, and care has to be taken not to access application buffers while they are accessed by the communication library. Nevertheless, this use usually improves performance. Nonblocking communications allow some overlap between computation and communication. More importantly, they allow better decoupling of sender and receiver. If a blocking send is executed ahead of the matching receive, either the sending task has to idle until the receive call occurs, or the outgoing message has to be copied and buffered. If a blocking receive is executed ahead of the matching send, the receiving task has to idle until the send occurs. When nonblocking communication is used, computation can continue until the wait call is executed. Thus, best performance is usually achieved by posting nonblocking send and receive calls as soon as possible (as soon as application buffers are ready) and posting wait calls as late as possible (just before the application buffers have to be reused).

Properties of point-to-point communication. Although blocking and nonblocking message-passing operations are common to many systems, the precise semantics of these constructs may vary in subtle and often undocumented ways. We describe below the main choices we made on the semantics of message passing in MPL and the rationale for these choices.

Order. Messages sent from a single source to a single destination are received in the order in which they were sent. More precisely, the following two properties are satisfied (see Reference 16 for a formal definition):

- A receive operation will receive a message only if there is no previously sent message from the same source that also matches the receive and has not yet been received.
- A receive operation will receive a message only if there is no previously executed (nonblocking) receive that is still pending with the same type and source parameters.

Order preserving reduces the amount of nondeterminism in the execution of parallel programs. For example, programs that do not use wild card receives will be deterministic. Order preserving simplifies programming and debugging and has no significant impact on performance on the SP2. Writing deterministic programs has the further advantage of decreasing the risk for deadlock.

Note that two posted receives may match the same message, even though they do not have the same type and source parameters: one of the receives

may use an explicit value, whereas the other uses a dontcare (the dontcare value specifies a wild card receive). In such a situation, there is no requirement in MPL that the message will be received by the earlier receive. Consider the following example:

```
void example1();
{
   int taskid, numtask, source, dest, type, msgid,
       msglen, nbytes, a[10], b[10];

   mpc_environ( &numtask, &taskid);

   source = 0;
   dest = 1;
   msglen = sizeof(a);

   if (taskid == source) {
        type = 1;
        mpc_bsend( a, msglen, dest, type);
        mpc_bsend( b, msglen, dest, type);

         } else {
            if (taskid == dest) {
               type = dontcare;
               mpc_recv( a, msglen, &source,
                         &type, &msgid);
               type = 1;
               mpc_brecv( b, msglen, &source,
                          &type, &nbytes);
               mpc_wait( &msgid, &nbytes);
            }
        }
    }
```

The first receive may receive either of the two messages sent, while the second receive receives the other message. In contrast, if both receive operations had used the same type (either 1 or dontcare), messages would be received in the order they were sent.

A stricter requirement would have a message received by the first posted receive that matches it, even if there were several matching receives that differ in their source or type parameters (because of dontcares). This requirement leads to perhaps more intuitive semantics, at the expense of more constraints on the algorithm used to match sends to receives. Since the difference is unlikely to affect many programs, MPL chose the easier-to-implement definition. Note, however, that MPI chose the second one. We can avoid this problem altogether by not using wild card receives—which will also improve performance (MPL implementations are optimized for code that does not use them).

Buffering. Any message-passing library has to cope with the limited amount of buffer space that can be made available to the library. Consider the following example:

```
void example2();
{
   int taskid, numtask, other, msglen, nbytes;
   char a[N], b[N];

   mpc_environ( &numtask, &taskid);

   msglen = sizeof(a);
   type = 1;

   if (taskid == 0) {
        other = 1;
        mpc_bsend( a, msglen, other, type);
        mpc_brecv( a, msglen, &other, &type,
                   &nbytes);

    } else {
       if (taskid == 1) {
        other = 0;
        mpc_bsend( a, msglen, other, type);
        mpc_brecv( a, msglen, &other, &type,
                   &nbytes);
       }
   }
}
```

Each task sends N bytes, using a blocking send, then receives the message sent by the other task. Neither of the receive operations can start before the preceding send operation completes. Thus, in order for this program to complete, it is necessary for at least N bytes to be copied and buffered. The success of this program depends on the amount of available buffering.

Messages can be buffered either in the memory of the sending node or in the memory of the receiving node, or both. (Other alternatives, such as buffering at a third node or buffering on disk, are theoretically possible but not practical.) When buffer space is exhausted, the sending task blocks. Thus, deadlock occurs if there is a cycle of tasks, each blocked while sending a message to the next task on the cycle. The previous program can lead to such a deadlock situation, with a cycle of length two.

The current high-performance implementation of MPL has a fixed amount of buffer space allocated to each pair of communicating tasks. This implementation leads to higher performance but requires the user to exercise some care in order to avoid deadlocks. The simplest deadlock avoidance policy is to make sure that communications occur in a consistent order at all tasks. That is, the communication calls executed by all tasks can be totally ordered in a sequence where sends occur before matching receives, and calls executed by any one task appear in the order in which they were issued by that task. A program that follows this rule will not deadlock, irrespective of the size of the messages and the amount of buffering available. Thus, deadlock can be avoided in the last example by reversing the order of send and receive at one task.

The use of nonblocking receives also helps. Early posting of nonblocking receives increases the likelihood that receives will precede sends, thus allowing the send operation to proceed without additional buffering. Deadlock can be avoided in the last example by having each task first post a nonblocking receive and then execute the send.

Collective communication. In parallel computations execution of a communication that collectively involves all tasks within a group is often required. Examples of such collective communications are broadcast, scatter, gather, and all-to-all communication. These patterns of communication are illustrated in Figure 1. In this figure we represent the global memory of all tasks that participate in the collective communication as a two-dimensional array, with task number being the vertical dimension and address within each task address space being the horizontal dimension. We illustrate the layout of data items in global memory before and after the communication.

We also need to perform reduction operations within groups, such as computing a global sum of variables stored one per task. MPL supports predefined and user-defined reduce operations. These operations can be used for reduction (results returned to one or all group members) and for scan operations.

Collective operations have limited usefulness, unless they can be applied to arbitrary, user-defined groups of tasks. One possible approach is to provide with each call an explicit list of group members. This approach leads to scaling problems and is not convenient to the programmer. Rather, MPL allows the programmer to define groups and refer to previously defined groups in collective communication calls. A *gid* (group id) argument is provided with each collective call to identify the group of participating tasks.

New groups can be defined, either by providing an explicit list of members or by splitting an existing group. The call `MP_PARTITION (parent_id, key, label, gid)` splits the group identified by `parent_id` into subgroups, one for each value of `label`. The processes are ordered within each subgroup according to the value of `key`.

Various management functions are provided for finding the size of a group, the rank of a task within a group, and vice versa. See Reference 17 for additional information on the design and implementation of the collective communication library.

MPL implementation

The Message-Passing Library is available on the SP2 in two different implementations, one on top of IP, and the other where message-passing calls are directly mapped, in user space, to the High-Performance Switch adapter. The user-space version has much better performance but can be used only by one task at each node. Thus, if several tasks timeshare the same node, the IP version is used.

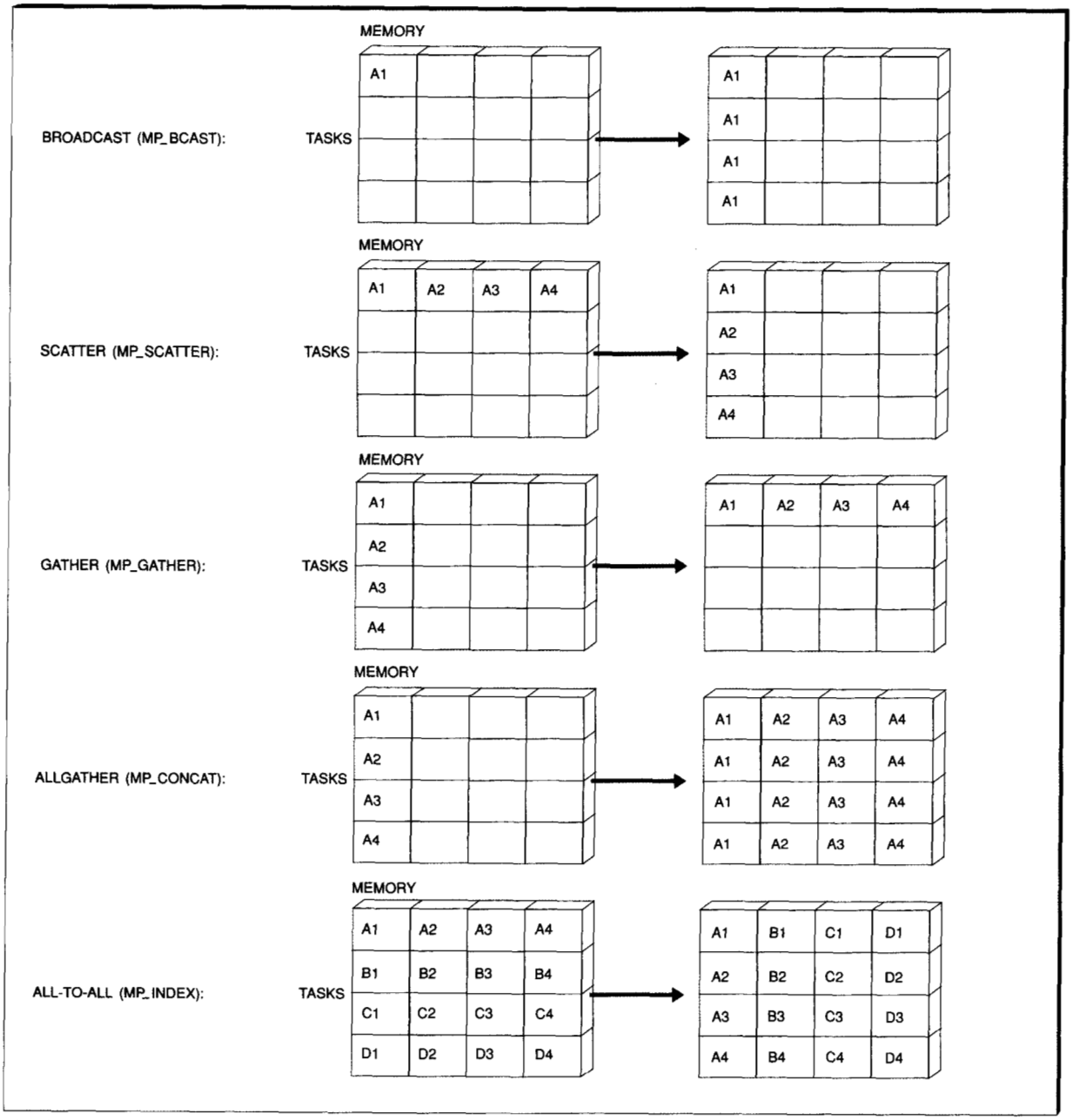
We shall discuss in detail the user-space implementation (US CSS, or user-space communication subsystem). The IP implementation shares the same top layer code, with the bottom layers replaced by IP.

The SP2 High-Performance Switch adapter is described in Reference 1 in this issue. It is Micro Channel*-attached and contains a DMA (direct memory access) engine for moving data on the Micro Channel. In addition, it contains an i860** microcontroller, memory, and a FIFO (first-in-first-out) interface to the switch. The adapter memory can be accessed by the main processor using programmed I/O (PIO).

The structure of the communication system was heavily influenced by the design of this adapter and by performance priorities.

A critical performance requirement was to increase the bandwidth (which was limited to 8.7 megabytes per second [MB/s] on the SP1). To do so, it is nec-

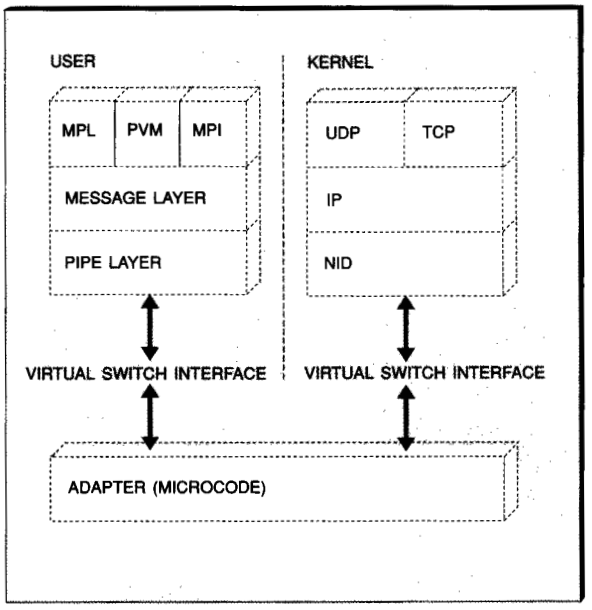
Figure 1 Collective communication operations



Adapted from D. Walker, "The Design of a Standard Message-Passing Interface for Distributed Memory Concurrent Computers," *Parallel Computing* **20**, No. 4, 657-673 (April 1994).

essary to transfer data over the Micro Channel using DMA, rather than PIO as on the SP1. This transfer requires data to be available in memory that is not paged or cached. Therefore, data to be sent are copied by the processor to a reserved buffer area and copied from there by the adapter using DMA; data received are copied by the adapter using DMA into a reserved buffer area, and copied from there by the processor to the user application space.

Figure 2 Structure of communication software



Latency was a second consideration, and processor communication overhead was a third consideration. Many codes are written in a loosely synchronous style, with computation phases followed by communication phases. In such codes there are few opportunities for overlapping computation and communication, and reduction of total communication time improves performance, even if processor overhead is increased.

The microcontroller used on the adapter is much slower than the main processor. Therefore, any offloading of protocol code from the processor to the adapter, although reducing the processor overhead for communication, increases the total communication time. We settled on a communication protocol where very few functions are executed on the adapter.

Communication between the processor and the adapter memory via the Micro Channel is slow. We therefore tried to minimize the number of accesses to adapter memory.

The communication hardware has no end-to-end flow control. Congestion at a destination causes saturation to occur at the switch that is connected to the destination and then to propagate backward in the network. This phenomenon of "tree saturation" can cause significant degradation in communication bandwidth for other destinations as well.[18] To avoid this problem (without tossing it back to the user), an end-to-end flow control protocol is needed.

The adapter has to support, at the same time, user-space communication for a task that uses the Message-Passing Library and IP communication for I/O and system services for that task, or other processes running on the same node. On the SP1, IP communication has to use an Ethernet connection when the adapter is used for user-space communication—thus reducing IP performance. This restriction is lifted on the SP2 where the adapter can be shared between the single task using user-space communication and any number of tasks using IP.

Finally, an important consideration resulting from the short development cycle was reuse of communication software that was developed for the SP1.

These considerations and constraints resulted in the implementation described below.

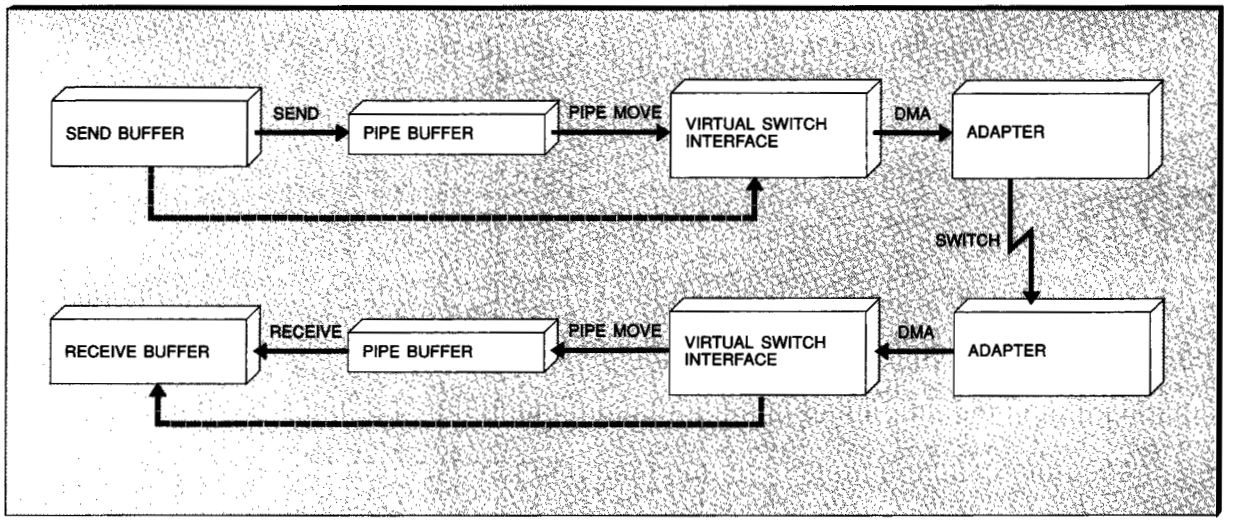
The overall structure of the communication software is shown in Figure 2. The left side of the diagram shows the communication stack for user-space communication.

The *message layer* supports a few simple, nonblocking message-passing calls. All MPL calls are mapped onto these calls. The collective communication layer is also implemented on top of these point-to-point message-passing calls.

The *pipe layer* maintains a separate bidirectional pipe for communicating with every other task. The pipes provide a reliable, flow-controlled, ordered stream of bytes between any pair of communicating processes. A variant of the sliding window protocol is used on each pipe. Acknowledgment packets with a time-out and retransmitting scheme are used to provide reliability. A token protocol is used on each pipe to make sure that data are sent on a pipe only when there is available space in the buffer at the other end of the pipe in order to avoid overflow.

The byte streams of outgoing pipes are multiplexed by the processor to a shared queue in the processor memory; conversely, the processor demultiplexes incoming data from a shared queue into the

Figure 3 Communication path from source to destination



incoming pipe buffers. This pair of shared queues, together with a few additional control registers, form a *virtual switch interface* in the processor memory. Data are moved between these queue buffers and the switch by the adapter, using high-bandwidth DMA. This data movement can occur simultaneously with computation done by the processor, allowing for some overlap between computation and communication.

The path traversed by a message from source to destination is illustrated in Figure 3. The transfer consists of the following steps:

- The message layer transfers the message data from the sender buffer to the pipe input buffer. It is a memory-to-memory copy operation that is executed by the sending processor.
- The pipe layer transfers the data from the pipe input buffer to the virtual switch interface. This memory-to-memory copy operation is also executed by the sending processor.
- The adapter DMA engine transfers the data from the virtual switch interface to the adapter.
- The sending adapter transfers the data, via the switch, to the receiving adapter.
- The DMA engine at the receiving adapter transfers the incoming data to the virtual switch interface.
- The pipe layer transfers the data from the virtual switch interface to the pipe output buffer.
- The message layer transfers the data from the pipe output buffer to the receive buffer.

When a message is sent that is larger than the pipe buffer, the communication subsystem attaches the user buffer in place of the pipe buffer. The pipe copy operation then moves data directly from the send buffer to the virtual switch interface, as indicated by the dashed line in Figure 3. The same bypass mechanism is used at the receiving node. Thus, the amount of data that is doubly copied in memory (to the pipe buffer, next to the virtual switch interface) is proportional to the number of messages sent, not to the total amount of data sent; likewise, the same applies to receives. The communication subsystem sends additional tokens to the sender to relax flow control when the receive pipe buffer is bypassed.

The same process executes, in one address space, the codes for the pipe layer, the message layer, and the user application. These codes can be seen as three coroutines that time-share the same process.

The message layer is invoked by the application code whenever an MPL call is executed. A send call results in data being copied from the application send buffer to a pipe input buffer; if no room is available in the pipe buffer, the send operation blocks. A receive call results in data being copied from a pipe output buffer to the application receive

buffer. If a matching message has not yet arrived, the receive is posted for later handling.

The message layer is invoked by the pipe layer whenever changes in the pipes allow pending communications to proceed. For example, when a new message arrives, the message layer is invoked in order to check whether a receive operation is already posted that matches this message. If such is found, the incoming data are copied from the pipe output buffer to the application receive buffer.

The pipe layer is invoked by the message layer whenever the latter executes. When invoked, the pipe layer services each active pipe; it copies data from the pipe input buffers to the virtual switch interface if tokens are available; it copies data from the virtual switch interface to the pipe output buffers and releases back tokens if data have arrived.

The pipe layer can also be activated by external interrupts from the adapter or by timer interrupts, as described in the next section. Such activation guarantees that progress occurs in data transfer even if the application code does not execute any MPL call.

The communication subsystem provides fair service to all active pipes. Thus, a long data transfer on one pipe will not block a transfer occurring on another pipe. If two different tasks concurrently send messages to the same destination, the destination can execute the matching receives in arbitrary order, without fear of deadlock. However, if a pipe buffer is full, no new messages can be received on that pipe until the pipe clears. In particular, if a task sends two successive messages to the same destination, and the first message fills the pipe buffer, the second message cannot be received ahead of the first.

The system maintains a list of active pipes to ensure that the overhead of polling pipes scales as the number of active pipes, not as the total number of pipes.

Another function provided by the adapter is translation of logical *taskids* used by the communication library to physical addresses. This translation allows the system to guarantee that user space communication will be contained within the allocated partition.

It is interesting to note that the software structure described above is very similar to an earlier implementation of MPL on the SP1 machine (MPL-p). The SP1 adapter did not support DMA, so the switch FIFO buffers in the adapter were directly exposed to the pipe layer. In the SP2, this implementation has been replaced by the virtual switch interface. Additional changes have been introduced because of the change in protection mechanisms and for performance tuning.

Polling versus interrupt. The pipe layer code is executed whenever an MPL function is invoked. In addition, this code is invoked periodically via timer interrupts in order to guarantee progress in the pipe layer, even if tasks are computing with no communication for a long period of time. Execution of the communication subsystem software can also be triggered by an interrupt from the adapter, when buffers have filled. Thus, if a nonblocking send is posted at a node, and a matching nonblocking receive is posted at another node, data will be moved from the sender buffer to the receiver buffer, even if neither node completes the communication operation by calls to `MP_WAIT` or `MP_STATUS`. Note, however, that if either sender or receiver is busy computing, the data transfer will be slow. If sender and receiver are not executing the communication code simultaneously, the amount of data that is sent on the pipe at each invocation of the communication subsystem will be limited by the number of currently available tokens. If the message is long, the data transfer is likely to complete only when both nodes block with a call to `MP_WAIT`.

MPL provides an alternative protocol whereby incoming packets arriving at a node that is computing cause an interrupt and the invocation of the communication library (the interrupt is disabled while the communication library executes). This action will, in general, cause earlier completion of data transfers, at the expense of additional overheads for interrupt handling.

Other communication subsystems. The High-Performance Switch can support IP communication at the same time it supports user-space communication. An AIX* (Advanced Interactive Executive*) network interface driver (NID) serves to connect the IP protocol to the adapter. NID supports IP packets with a Maximum Transmission Unit (MTU) of up to 64 kilobytes. The low-level protocol between the adapter and this driver is similar to the interrupt-driven protocol for user-space communication. However, the NID uses a separate virtual switch interface: The queues in system memory

and the registers in the adapter memory that are accessed by NID are mapped in kernel space. The adapter multiplexes or demultiplexes data between the separate virtual switch interfaces and the switch.

The message layer on top of which MPL is implemented can support other message-passing libraries. A product implementation of the communication functions of PVM is available.[19] In addition, a prototype implementation of MPI has been completed.[20] The MPI implementation required some changes in the message layer (e.g., to handle contexts); in some situations, it requires additional buffering in order to decouple communication occurring in different contexts. However, no changes were required in the pipe layer; the MPI prototype achieves performance virtually identical to MPL for basic communication functions.

Performance. The performance of the user-space implementation of MPL for both the SP1 hardware and the SP2 hardware is listed in Table 1. (The 370/TB0 numbers were measured for this paper by Fiona Sim.) All results shown are obtained with the SP2 software. The first row lists performance for the SP1 hardware: a 370 (POWER1*) processor and a TB0 adapter with no DMA function. The second row lists the performance for the SP1 (POWER1) nodes with the enhanced TB2 (SP2) adapter, which has DMA. The next two rows provide performance for the SP2, with the two types of POWER2* nodes: thin (390) and fat (590).

All numbers are measured application to application. Latency is measured by sending a zero-byte message round-trip between two nodes. It is calculated as half the round-trip time. Blocking sends and receives are used for the transfer. Point-to-point bandwidth is measured in the same fashion. The results quoted below are asymptotic bandwidth results (large messages) and include the startup time. With use of the SP1 software, there were large discontinuities in communication time as message size increased. With the SP2 software, the communication time is nearly linear as message size increases. Also reported is exchange bandwidth, which is the total bandwidth between two nodes that are engaged in simultaneously sending messages to each other utilizing the nonblocking message-passing calls. The bandwidth is measured in megabytes per second, and latency in microseconds (μs).

Table 1 MPL performance (user space)

Node Type	Switch Adapter	Latency (μs)	Bandwidth pt-to-pt (MB/s)	Bandwidth Exchange (MB/s)
370	TB0	37.6	8.7	8.8
370	TB2	55.0	31.2	34.7
Thin (390)	TB2	40.0	35.5	37.0
Fat (590)	TB2	39.0	35.5	48.2

The High-Performance Switch has a peak bandwidth of 40 MB/s in each direction, for a total of 80 MB/s, bidirectional, per node. On the SP2, the communication library achieves close to 90 percent of the unidirectional switch bandwidth and more than 60 percent of the bidirectional switch bandwidth. Thus, the unidirectional bandwidth is hardware bound, at the switch.

The Micro Channel bus imposes another constraint on peak communication bandwidth. The peak transfer bandwidth on the Micro Channel is 80 MB/s. Due to packet size and arbitration overhead on the bus, the Micro Channel can deliver at most 52 MB/s to the communication subsystem. Thus, the communication library achieves more than 90 percent of the Micro Channel bandwidth for exchanges and close to 70 percent of its bandwidth for unidirectional communication, on fat nodes. Therefore, the exchange bandwidth is hardware bound, at the Micro Channel, with fat nodes.

A third constraint on communication bandwidth is the processor (software) overhead for data transfer. The unidirectional bandwidth on the SP2 nodes is 96 percent of the exchange bandwidth for thin nodes. In contrast, the unidirectional bandwidth is only 73 percent of the exchange bandwidth on the fat nodes. The thin node exchange bandwidth of 37.0 MB/s is, essentially, the upper rate at which the communication software can transfer data on this processor. The processor executes 1.8 cycles per byte transferred. A higher rate would require support for larger packets or significant changes in protocol. Unidirectional transfers essentially saturate the thin node processor, with no spare compute power left. In contrast, a fat node executes less than 1.4 cycles per byte transferred. The improvement is almost entirely due to the higher memory bandwidth (factor of four or better) that allows us to copy data faster. Unidirectional transfers utilize less than 73 percent of the compute

Table 2 MPL performance (UDP/IP)

Node Type	Switch Adapter	Latency (μs)	Bandwidth pt-to-pt (MB/s)	Bandwidth Exchange (MB/s)
370	TB0	357.5	N/A	N/A
370	TB2	403.0	7.2	N/A
Thin (390)	TB2	319.0	8.3	N/A
Fat (590)	TB2	277.0	10.8	13.3

power of the fat node. The spare computation power could be used to achieve an overlap of more than 27 percent of computation and communication for well-structured computations.

A fourth constraint on communication bandwidth is the adapter and the microcode it executes. It seems, in fact, that the exchange bandwidth for fat nodes is limited by the adapter, rather than the processor, so that the excess compute power and possible computation-to-communication overlap is higher than suggested by our previous analysis.

It is worthwhile to observe that the new TB2 adapter boosts unidirectional bandwidth from 8.7 MB/s to 31.2 MB/s, even with no change in the SP1 compute nodes. This boost reflects the difference between PIO and DMA bandwidth on the Micro Channel. However, the latency increases from 37.6 microseconds to 55 microseconds when TB0 is replaced by TB2. This increase is no surprise since the communication pipeline is now longer (with DMA engines and the i860 controller). To the (software) delay occurring from the start of a send operation until data are in the (virtual) switch interface, we now add the (microcode) delay of the transfer from the virtual switch interface to the physical switch; likewise, the same applies on the receive end. The latency comes down to 39 (40) microseconds on the fat (thin) nodes, reflecting improvement in the execution time of the communication library between POWER1 and POWER2. The switch latency is less than one microsecond of this total.

The corresponding numbers for the IP version of MPL are listed in Table 2.

The difference in performance between user-space communication and IP communication mostly reflects the impact of IP software overheads. Although communication performance is much lower, the flexibility of sharing the adapter between multiple processes may lead to better overall system utilization for applications that do not have stringent communication requirements.

The latency and bandwidth for the current prototype implementation of MPI on the SP2 were measured on a system with thin nodes;[20] the latency and bandwidth numbers are virtually identical to the MPL numbers. At the time of writing, performance results for other libraries (such as IBM's implementation of PVM) are not available. It is expected that performance will be similar but presumably somewhat worse than for MPL, since optimization has been done primarily for MPL.

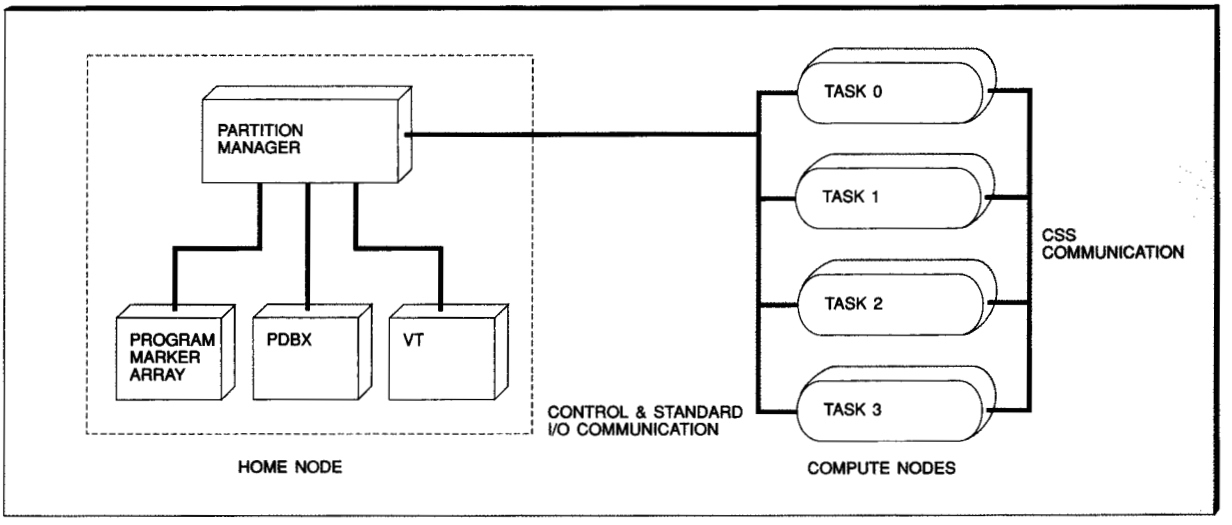
Parallel Operating Environment

The Parallel Operating Environment (POE) is used to control the execution of parallel programs. Its structure is illustrated in Figure 4. The execution of a parallel program is controlled by the Partition Manager (PM) process that runs on the home node. The home node can be any workstation that is connected by a local area network (LAN) to an SP2. The program will execute on remote nodes, which can be LAN-connected workstations or SP2 nodes. The user invokes various POE functions by submitting requests to PM. The various functions of POE can be invoked using either a line command interface or an AIXwindows* Parallel Desktop graphical user interface. In addition, a (batch) POE job can be submitted using LoadLeveler* (an IBM network batch queuing product[21]).

In order to run a parallel program using MPL, we need to compile it and link it to a message-passing library. A different Message-Passing Library is used for user-space communication (on dedicated SP2 nodes connected with the High-Performance Switch) and for IP communication. The Message-Passing Library can be linked statically or dynamically when the program is loaded.

The next step is to allocate a set of nodes that will run the parallel program. The mapping of tasks to physical nodes can be specified explicitly by the user, via a host file. Alternatively, the user may require a number of nonspecific nodes on an SP2. Nodes of an SP2 are divided into several pools (typically one for small systems), each managed by the Resource Manager (an IBM SP2 system software function that manages resources and controls access to them). The Partition Manager will interact with the Resource Manager to have the required

Figure 4 Parallel Operating Environment



number of nodes allocated. Nodes may be dedicated or shared by multiple tasks.

Next, the user specifies the executable program to be loaded on the allocated nodes. POE supports both a single-program, multiple-data (SPMD) model, where all tasks execute the same program, and a multiple-program, multiple-data (MPMD) model, where different tasks can execute different programs. In the latter case, the user can specify a different executable program for each task. Each task then starts executing its code asynchronously.

The Partition Manager also connects the standard I/O streams (stdin, stdout, stderr) to each task in the partition. Various mechanisms can be specified for demultiplexing input and multiplexing output. Stdin can be sent either to all tasks in a partition or to a unique, specified task. Stdout can contain the output of a unique, specified task, or the unordered merge of all task outputs, or ordered merge of the contents of the output buffers of all tasks. When multiple output streams are merged, the outputs can be prefixed by task numbers.

Tools. The Partition Manager can be used to activate and manage a variety of tools for program debugging and tuning. The simplest (and perhaps most convenient) tool for program debugging is print statements. However, it is quite tedious to decipher interleaved printouts of a large number of tasks. A simple "parallel output" interface is provided by the Program Marker Array (Figure 5). This array is a graphic display with a row associated with each task. Each row contains a number of "LEDs" (indicators, so-called because they are like light-emitting diodes) and additional space for text. Calls within the program can be used by each task to output a string on the text space of its row or to set the color of each LED. This can provide a simultaneous display of the state of each task during a parallel execution. The Program Marker Array is based on a research prototype developed by Dror Feitelson.[22]

Figure 5 The Program Marker Array

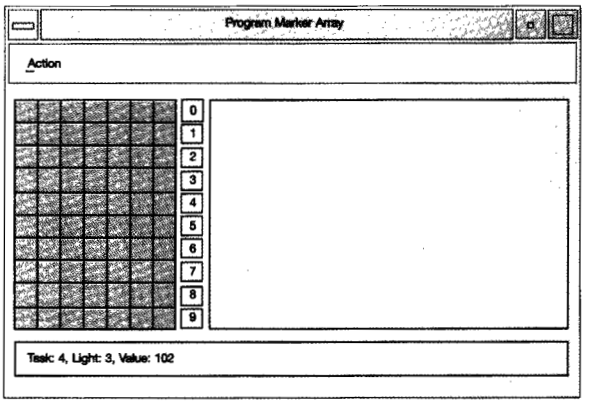


Figure 6 The xpdbx window

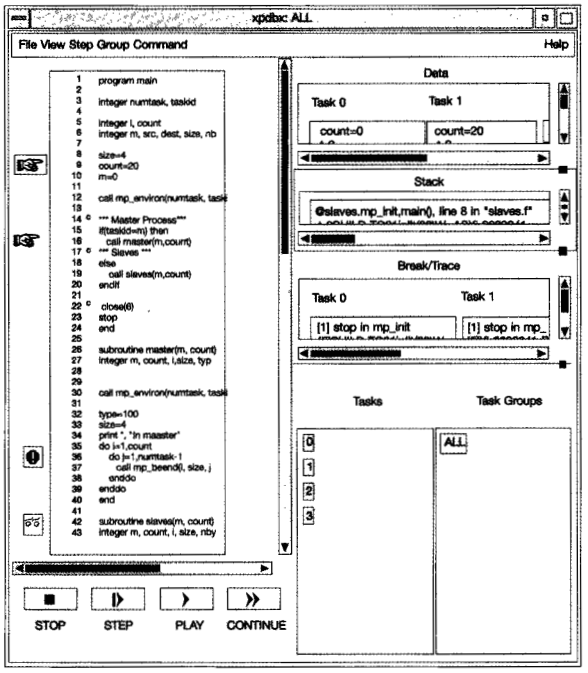
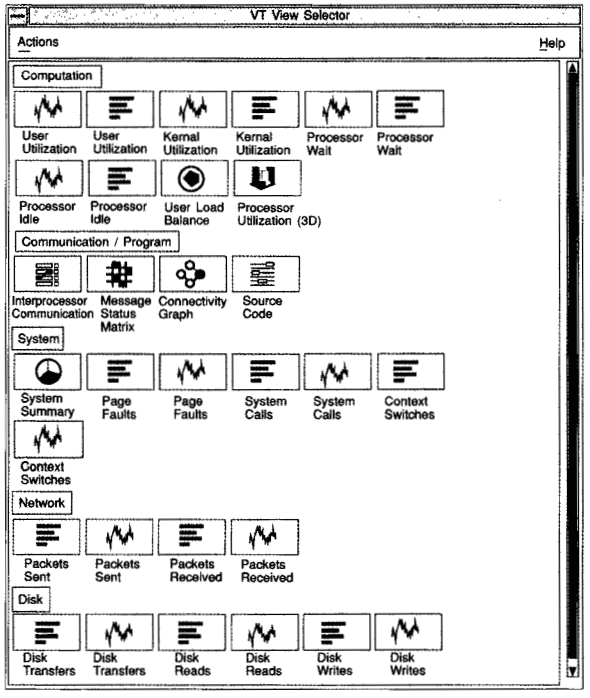


Figure 7 The view selector window



Source-level debugging of parallel codes is supported by *pdbx* (line interface) and *xpdbx* (X Windows System** interface). The pdbx function is a POE application that runs as a server task on the home node and as a number of client *dbx* tasks on the remote nodes. The pdbx function supports most of the dbx debugger functions. Breakpoints can be set for any task so as to halt execution when a source line is reached, or a variable changed, or a condition satisfied, or a procedure entered; tracepoints can be set at any task to print tracing information. In addition, pdbx allows the user to define groups of tasks and to execute dbx commands for all tasks in a group. Thus, we can set a breakpoint that will halt all tasks in a group at the same code line. Such commands are interpreted by the pdbx server, which sends corresponding dbx commands to the dbx clients, collects their output, and displays the information to the user. The xpdbx function provides a convenient, Motif**-based X Windows System interface for pdbx functions (Figure 6).

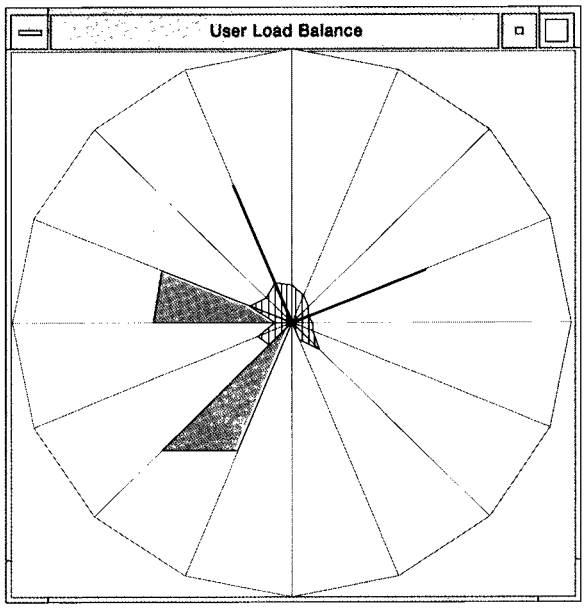
The Visualization Tool (VT) provides graphical views of performance characteristics of a parallel execution. A graphical interface is used to select active views (Figure 7).

VT can be used on line to monitor AIX kernel statistics such as CPU activity (kernel, user), disk activity, network TCP/IP activity, context switches, page faults, and system calls. Some of the information can be displayed in different views to highlight problems specific to parallel execution. Thus, a Kiviat diagram of (average and instantaneous) processor utilization presents a useful graphical display of load balance in a parallel application (Figure 8). VT can also be used postmortem to visualize a performance trace that was collected during execution. In this mode, we can display additional information on message-passing activity. We can display ongoing communications in a graph or matrix display, or in a streaming chart (Figure 9). We can also open a source code window.

A trace playback control window allows the user to move back and forth in execution time and adjust the playback speed (Figure 10).

Either an instantaneous or cumulative presentation can be selected for most of the views. Additional information can be obtained by clicking on the view. Thus, by clicking on one of the spokes of the load balance view, we get numerical values

Figure 8 Load balance view



of current times, instantaneous CPU utilization, and average CPU utilization for the processor displayed on that spoke. By clicking on the interprocessor communication view, we open a menu that allows a search for selected communication events.

VT is based on the program visualization tool developed by Doug Kimelman.[23]

Conclusion

The SP2 Parallel Operating Environment (POE) allows users to develop and execute on the SP2 parallel jobs that take advantage of the fast communication hardware. It allows the same codes to be developed and executed on a workstation or workstation cluster. POE has evolved in the transition from the SP1 to the SP2 and will continue to evolve in the future. As communication hardware changes, the communication subsystem software will change to take advantage of it. Future SP systems will provide more services to parallel applications and will allow more flexible resource allocation policies; POE will evolve to support those. POE will support new programming paradigms, such as High Performance FORTRAN. Finally, programming tools will continue to evolve so as to provide an increasingly user-friendly environment for program development.

Figure 9 Streaming chart of interprocessor communication

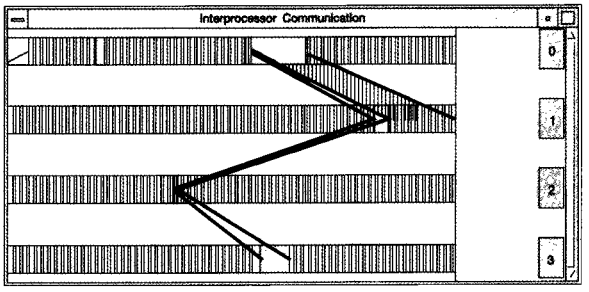
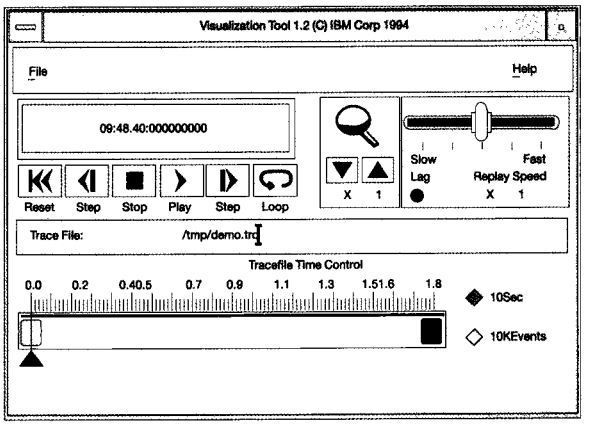


Figure 10 Trace playback control window



Acknowledgments

The design and implementation of the software referenced in this paper has been influenced by a large number of individuals, and it is impossible to correctly and fairly differentiate individual contributions. The names of some of the contributors are listed in References 1, 11, 14, 17, 20, 22, and 23. In addition, we wish to acknowledge the contributions of Don Grice, Pong Huang, Robert Straub, Wendy Cheng, Bill Tuel, Dave Reynolds, Kevin Reilly, Steve Hughes, Bob Dilly, Fiona Sim, and Mark Smith.

This list is by no means complete. All of the individual components referenced exist today because of significant contributions from IBM Research (at both the T. J. Watson Research Center and Almaden Research Center) and from the POWER Parallel Division. Some components exist

only because of the tightly knit joint efforts between the various organizations. We acknowledge all the individuals who have participated in the SP1 and SP2 projects.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of X/Open Co. Ltd., Intel Corporation, Massachusetts Institute of Technology, or Open Software Foundation, Inc.

Cited references

1. C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Grice, P. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker, "The SP2 High-Performance Switch," *IBM Systems Journal* **34**, No. 2, 185–204 (1995, this issue).
2. G. A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam, *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing*, The MIT Press, Cambridge, MA (1994).
3. N. Carriero and D. Gelernter, "LINDA in Context," *Communications of the ACM* **32**, No. 4, 444–458 (April 1989).
4. High Performance FORTRAN Forum, "High Performance FORTRAN Language Specification," *Scientific Programming* **2**, No. 1, 1–170 (1993).
5. T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, "SP2 System Architecture," *IBM Systems Journal* **34**, No. 2, 152–184 (1995, this issue).
6. P. F. Corbett, D. G. Feitelson, J.-P. Prost, G. S. Almasi, S. J. Baylor, A. S. Bolmarcich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. D. Herr, J. Kavaky, T. R. Morgan, and A. Zlotek, "Parallel File Systems for the IBM SP Computers," *IBM Systems Journal* **34**, No. 2, 222–248 (1995, this issue).
7. *nCUBE 2 Programmers Guide*, nCube Corporation, Foster City, CA (1990).
8. *Intel iPSC/860 Programmer's Reference Manual*, Intel Corporation, Beaverton, OR (1990).
9. *Express 3.0 Introductory Guide*, Parasoft Corporation, Pasadena, CA (1990).
10. Message-Passing Interface Forum, "MPI: A Message-Passing Interface Standard," *International Journal of Supercomputer Applications* **8**, No. 3/4, 165–414 (1994).
11. V. Bala, J. Bruck, R. Bryant, R. Cypher, P. de Jong, P. Elustondo, D. Frye, A. Ho, C-T. Ho, G. Irwin, S. Kipnis, R. Lawrence, and M. Snir, "The IBM External User Interface for Scalable Parallel Systems," *Parallel Computing* **20**, No. 4, 445–462 (April 1994).
12. *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference, Rel. 2*, SH26-7228, IBM Corporation (1994); available through IBM branch offices.
13. *IBM AIX Parallel Environment: Operation and Use, Rel. 2*, SH26-7230, IBM Corporation (1994); available through IBM branch offices.
14. B. G. Fitch and M. E. Giampapa, "The Vulcan Operating Environment: A Brief Overview and Status Report," *Proceedings of the 5th Workshop on Use of Parallel Processors in Meteorology*, European Centre for Medium-Range Weather Forecasts (November 1992).
15. V. Bala and S. Kipnis, "Process Groups: A Mechanism for the Coordination of and Communication Among Processes in the Venus Collective Communication Library," *Proceedings of the 7th International Parallel Processing Symposium*, IEEE (April 1993).
16. R. Cypher and E. Leu, "The Semantics of Blocking and Nonblocking Send and Receive Primitives," *Proceedings of the 8th International Parallel Processing Symposium* (April 1994), pp. 729–735.
17. V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C. Y. Ho, S. Kipnis, and M. Snir, "CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers," to appear in *IEEE Transactions on Parallel and Distributed Computing.*
18. G. F. Pfister and V. A. Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks," *Proceedings of the 1985 International Conference on Parallel Processing* (August 1985), pp. 790–795.
19. *IBM AIX PVMe User's Guide and Subroutine Reference, Rel. 1*, SH23-0019, IBM Corporation (1993); available through IBM branch offices.
20. H. Franke, P. Hochschild, P. Pattnaik, J-P. Prost, and M. Snir, "MPI on IBM SP1/SP2: Current Status and Future Directions," *Proceedings of the 2nd Workshop on Scalable Parallel Libraries* (October 1994), pp. 39–48.
21. *IBM LoadLeveler: User's Guide*, SH26-7226, IBM Corporation (November 1994); available through IBM branch offices.
22. D. Feitelson, "Terminal I/O for Massively Parallel Systems," *Proceedings of the Scalable High-Performance Computing Conference* (May 1994), pp. 263–270.
23. D. N. Kimelman and T. A. Ngo, "The RP3 Program Visualization Environment," *IBM Journal of Research and Development* **35**, No. 5/6, 635–652 (November 1991).

Accepted for publication January 5, 1995.

Marc Snir *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: snir@watson.ibm.com).* Dr. Snir is senior manager at the IBM Thomas J. Watson Research Center, where he leads research on scalable parallel software and on scalable parallel architectures. He recently led the Vulcan software effort and the initial design and prototyping of parallel software for the IBM SP1. He received a Ph.D. in mathematics from the Hebrew University of Jerusalem in 1979. He worked at New York University (NYU) on the NYU Ultracomputer project from 1980–1982 and worked at the Hebrew University of Jerusalem from 1982–1986. He has published on computational complexity, parallel algorithms, parallel architectures, interconnection networks, and parallel programming environments. He recently contributed to High Performance FORTRAN and to the Message-Passing Interface. Dr. Snir is a member of the IBM Academy of Technology, a senior member of IEEE, and a member of ACM and SIAM.

Peter Hochschild *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: phoch@watson.ibm.com).* Dr. Hochschild works in the areas of parallel hardware and software, and communication systems. He is the principal designer of the Vulcan switch and the EUIH prototype message-passing software for the IBM SP1 and SP2 machines. He received a Ph.D. in computer science from Stanford University.

Daniel D. Frye *IBM POWER Parallel Division, Highly Parallel Supercomputing Systems Laboratory, 522 South Road, Poughkeepsie, New York 12601–5400 (electronic mail: danielf@vnet.ibm.com).* Dr. Frye did graduate work at The Johns Hopkins University in theoretical atomic physics. He did postdoctoral work at the University of Virginia in computational atomic photoionization processes utilizing vector computers and at IBM Kingston in quantum chemistry of small molecules and in parallelization and vectorization of numerically intensive codes. At IBM he was responsible for parallelization of scientific applications on shared-memory and coupled shared-memory systems. He transferred to the Highly Parallel Supercomputing Systems Development Laboratory to work on parallel applications and do performance studies in a distributed memory environment. Dr. Frye continued working in the POWER Parallel Development Laboratory in Kingston on languages, architecture, and performance benchmarking for scalable, parallel RISC-based systems. He shifted to full-time work on software architecture for the SP2 before taking current responsibility as manager of the Software System Design group for the IBM SP series of supercomputers.

Kevin J. Gildea *IBM POWER Parallel Division, Highly Parallel Supercomputing Systems Laboratory, 522 South Road, Poughkeepsie, New York 12601–5400 (electronic mail: gildeak@vnet.ibm.com).* Dr. Gildea is the lead programmer for the communication subsystem software development team in the POWER Parallel Division. He joined IBM in Poughkeepsie, New York, in 1982 upon receiving a B.S. in computer science from the University of Scranton. He worked in Poughkeepsie until 1987 on the development of robotic and manufacturing control systems. In 1987, he became IBM's Resident Engineer in Rensselaer Polytechnic Institute's Center for Manufacturing Productivity, where he worked on the Computer Integrated Manufacturing Program. Dr. Gildea received M.S. and Ph.D. degrees in computer science from Rensselaer Polytechnic Institute and joined the POWER Parallel Division in 1992.

Reprint Order No. G321-5565.