PROCEEDINGS OF THE SECOND WORKSHOP ON
ENVIRONMENTS AND
TOOLS FOR PARALLEL
SCIENTIFIC COMPUTING

Edited by Jack J. Dongarra
Bernard Tourancheau

S IAM
Abstract
A complete prototype implementation of MPI on the IBM Scalable Power PARALLEL 1 (SP1) is discussed. This implementation achieves essentially the same performance as the native EUI library, although MPI is much larger. The paper describes the implementation of EUI on SP1, the modifications required to implement MPI, initial performance measurements, and directions for future work.

1 EUI on SP1
1.1 SP1
The 9076 PowerPARALLEL System 1 (SP1) is a distributed memory multiprocessor that is produced by IBM [5]. Each SP1 node consists of an RS/6000 processor with up to 256 MByte memory and up to 2 GByte disk. Each node runs a full copy of the AIX UNIX operating system. Fast communication is provided by a multistage packet-switching network with a hardware capability for 40 Mbyte/s duplex transfers from each node, and a total latency which is below one microsecond. The effective performance achieved is much lower, due to limitations of the SP1 communication adapter. This is a passive device attached to the Microchannel bus and provides a simple interface consisting of FIFO buffers for incoming and outgoing packets and control registers. All communication protocol and data transfers are currently executed in software.

The AIX Parallel Environment [6] supports the EUI [1, 7] message passing library and provides tools for debugging and performance tuning of message-passing code. In order to achieve high performance, a user may have exclusive use of a partition of the system. Partitions may include an arbitrary subset of nodes. For good performance of parallel jobs, one typically dedicate each node in the partition to one computation process.

The EUI message-passing library is available in several implementations. For higher performance, a prototype implementation (called EUI-H) was developed where the entire communication stack executes in user space, as opposed to invoking a UNIX device driver. With this implementation, the communication adapter is dedicated to one process per node. Protection is provided by labeling each packet with a sufficiently large partition key.

1.2 System Architecture
The EUI-H communication software is structured as follows.

Packet Layer The packet layer provides software point-to-point packet transport facility among processors by direct interaction with the communication network. This layer creates packets, and inserts appropriate routing information into each of the generated packets. In
order to preserve the flexibility of choosing various routing strategies and simplify the error recovery, analogous to IP-UDP, this layer is not assumed to provide reliable transport or packet ordering. However this layer is expected to provide uncorrupted packets, performing checksum on packets if necessary.

**Pipe Layer**

The pipe layer is built on top of the packet layer and provides a reliable, flow controlled, byte-stream oriented communication layer. Each processor maintains a send and a receive buffer (called pipe) to every other processor in the parallel job. These buffers perform functions similar to UNIX pipes and sockets (that is how their name evolved), yet they shall not be confused with them, as they are placed in user space to avoid the cost of a UNIX kernel access.

Task of the pipe layer is a reliable data transport from the send pipe to the corresponding receive pipe. This layer uses the typical mechanisms, such as flow control, acknowledgments and retry after time-out etc. Flow control is achieved by associating tokens with the contents of the buffers. Tokens flow back when the message layer reads data from the receive pipe. Accordingly, the sending side is not permitted to enter new data into the send pipe if tokens are not available. The pipe layer provides the following non-blocking functional interface (where # denotes a specific pipe number).

- **TokenAvailable(#)**: tests how many bytes can be written to the pipe.
- **BytesReadable(#)**: tests how many bytes can be read from the pipe.
- **WritePipe(#)**: writes data into the pipe. The pipe maintains its internal status, i.e. buffer pointer, and takes care of wrap around etc.
- **ReadPipe(#)**: Reads data from a pipe, i.e. from its next available position.
- **KickPipes()**: Invokes the pipe scheduler.

The pipe scheduler effects the following activities. Most importantly, it has to drain the packets from the network to avoid network congestion. Since packets may arrive out of order, it must place the incoming data at the correct position in the pipe buffer and acknowledge the packet together with the appropriate number of tokens. When data arrives at the front of a receive buffer, the **ReadFromPipe** callback function is called to notify the message layer that data can be read. Similarly, if acknowledge packets arrive, by which new send pipe buffer space is freed, the **ShovelIntoPipe** callback function will be called to allow the message layer to insert more data into the pipe. Since either the data packets or acknowledge packets can be lost, the pipe scheduler will resend unacknowledged data after a certain time-out, and consequently the pipe layer must recognize duplicate packets and drop them.

The pipes are scheduled in a fair manner to access the underlying network. Sending a large message will not prevent the delivery of pending sends on other pipes.

The scheduler does not run as a separate thread. Rather, it is invoked by the message layer in almost every point-to-point communication request (using **KickPipes()**). Progress must be guaranteed even in the situation that the application executes communication unrelated code for long periods of time. This is achieved by periodically invoking the scheduler asynchronously in the background using UNIX signal handling. An interrupt driven version also exists, which calls the scheduler when packets arrive from the high performance switch.
simplify the error and maintain a send job. These buffers evolved), yet to avoid the cost of the corresponding acknowledgments with the contents in the receive pipe. Send pipe if tokens functional interface a don't care value is used in a receive call for source then EUI returns in the source argument the actual source of the message received; likewise, for tag. Thus, these two arguments are INOUT arguments. Such usage is error prone. Instead, MPI uses a separate structure argument status to return these values. This requires minimal changes in the implementation.

A more significant change introduced by MPI is the use of communicators. In EUI (as in most other message-passing libraries) the dest (destination) parameter is an absolute index that identifies the message destination. In contrast, in MPI, dest is the relative index of the destination within an ordered group of processes that is identified by the comm (communicator) argument. This mechanism provides important support for modular development of large codes and libraries: a module running on a subset of processes can use a local name space for its communication. This change introduces one additional level of indirection as an absolute address has to be derived from dest by lookup in the communicator member table.

The communicator argument (which is a handle to a communicator object) also identifies a communication context. Communications using different communicators occur in different "communication universes", and do not interfere with each other. To ensure this, messages carry an additional context field; exact matching on context is required at the receiving end. This context field can be seen as an extension of the tag field, except that it is extracted from the communicator object, and is always matched at the receive end. This requires minor changes in the implementation of sends and receives.

2 From EUI to MPI
The EUI message passing library consists of a small number (32) of functions for point-to-point communication, collective communication, and environmental inquiry and management. The basic point-to-point layer consists of nine functions: blocking and non-blocking sends and receives of contiguous data; blocking send and receive of strided data, and a combined send_and_receive function. Messages carry a tag (or type) and a receive can select messages by sender or by tag, with wild cards allowed for either or both.

We have recently completed a prototype implementation of the new MPI message passing library on SP1 (MPI-F). MPI [8] is the outcome of a standardization effort that involved about 60 people from 40 organization throughout 1993. The result is a library with close to 130 functions, and with richer functionality in the basic point-to-point communication functions. One might suspect this prevents an efficient implementation of MPI. Our experience dispels this suspicion.

The MPI point-to-point message passing layer is much richer than the EUI one (53 functions). Also, the functionality of the basic message-passing functions is somewhat different, requiring changes in the implementation. An overriding concern of our implementation has been to achieve the same level of performance as EUI achieves, for those basic communication functions that EUI supports.

Consider simple, non-blocking EUI C send and receive calls:

\[ \text{MP SEND}(\text{outmsg, msglen, dest, tag, msgid}); \]
\[ \text{MP RECV}(\text{inmsg, msglen, source, tag, msgid}); \]

This gets replaced, in MPI, by

\[ \text{MPI ISEND}(\text{outmsg, count, type, dest, tag, comm, msgid}); \]
\[ \text{MPI IRECV}(\text{inmsg, count, type, source, tag, comm, status, msgid}); \]

Both cases, the function returns 0 if successful, an error code, otherwise.

Some of the changes from EUI to MPI are merely syntactic convenience. For example, if a don't care value is used in a receive call for source then EUI returns in the source argument the actual source of the message received; likewise, for tag. Thus, these two arguments are INOUT arguments. Such usage is error prone. Instead, MPI uses a separate structure argument status to return these values. This requires minimal changes in the implementation.

The communicator argument (which is a handle to a communicator object) also identifies a communication context. Communications using different communicators occur in different "communication universes", and do not interfere with each other. To ensure this, messages carry an additional context field; exact matching on context is required at the receiving end. This context field can be seen as an extension of the tag field, except that it is extracted from the communicator object, and is always matched at the receive end. This requires minor changes in the implementation of sends and receives.
2.1 Buffering

More significant changes are required because of the different semantics of the basic send and receive functions. All message passing libraries have to cope with the limited amount of buffer space that can be made available to the library. If message production runs too much ahead of message consumption, then the system may run out of buffers, which will force to block senders, or to abort the program. If the first option is chosen, then program may deadlock. Thus, any message passing library has to assume that the user program is "well-behaved", to some extent, in its buffering requirements. Various trade-offs are possible in the implementation of a message passing library. Generally, an implementation that is more "lenient" to the user (requires looser coordination between producer and consumer) and buffers more aggressively, will use more memory and will have higher communication overheads, because of the need for additional memory-to-memory copies and dynamic buffer allocation.

The current EUI-H implementation is very strict (or restricted) in its buffer allocation policy: no buffering is provided, in addition to that available in the pipe layer. If the pipe between a and b is full, then no additional data can move into the pipe. A send from a to b blocks until a receive is posted at b for the message at the head of the pipe.

Many users seem to desire more buffering. More importantly, MPI specifies that communication with different communicators is non interfering. This becomes important if processes are multithreaded: blocked communication between two threads should not prevent communication in another context between two other threads of the same processes. Thus, the pipe between two processes need be fairly multiplexed by all contexts that use it.

To achieve this goal, we are providing additional buffering for early arriving messages at the receiver side. Short messages are sent eagerly, and are buffered by the receiver. Long messages use a rendezvous protocol: the sender first issue a "request-to-send". The receiver acknowledges this request when a matching receive is posted. At that point, the sender forwards the data. Thus, the amount of buffering that needs be provided is proportional to the number of messages sent before a receive is posted, not to their total size. The additional overhead for the rendezvous protocol is negligible for long messages.

In order to provide good buffer utilization, we dynamically allocate buffer space to early arriving messages from one shared pool. Note that malloc UNIX function cannot be used for that purpose: malloc is not a protected, atomic system call. On the other hand, the communication library can be invoked asynchronously by a UNIX signal to handle an incoming message. Instead, MPI-F uses its own "private" buffer management. This buffer management is also used for allocating space to the various objects that are created dynamically by MPI calls.

To determine the latency and bandwidth for MPI point-to-point communication we have measured the time of a simple Ping-Pong program using MPI_Send and MPI_Recv for contiguous messages of various sizes.

The resulting latency as a function of message size is shown in Figure 1. The latency of the MPI_Send/MPI_Recv pair is 29 µsec, for a zero byte message. The effective bandwidth is 8.6 MBytes/sec, for long messages. The additional functionality of MPI has not led to a significant change in performance, as compared to EUI. The reason is that no additional buffering occurs in the "best" case where receives are posted ahead of sends (no early arrivals). Essentially, MPI-F will provide additional buffering only in situations where EUI-H would block.
2.2 Derived datatypes

The EUI call argument msglen specifies the length of the message in bytes. This argument is replaced in MPI by the two arguments: type and count: type specifies the basic datatype of the message component (integer, real, etc.), whereas count specifies message length in multiples of this basic component. This simplifies the task of the programmer and provides better portability across machines with different sizes for the same basic type. More importantly, this allows data conversion for MPI implementations across heterogeneous systems.

Support of MPI types requires no significant overheads, for simple, predefined types in a homogeneous environment: it is merely that the count argument need be scaled by a factor that depends on the type argument. However, MPI also supports user-defined types. Such types basically specify a sequence of displacements (relative to the initial address of the communication buffer), and the basic datatype of the element at this displacement. New types are built by applying a variety of type constructors to previously defined datatypes (such as concatenation, replication with stride, replication with a sequence of user-provided displacements, etc.). Datatype definitions can be nested to an arbitrary depth. With such user-defined datatypes one can send or receive, with one call, a structure, or a submatrix of an array, or, indeed, an arbitrary collection of objects.

The communication operation needs to interpret the user-defined datatype, in order to gather the data from the communication buffer, or scatter it to the communication buffer. One approach is to prepare a “flattened” description of the datatype, as a sequence of displacement, blocklength pairs. In order to collect the data, one need merely to traverse sequentially this flat structure. The disadvantage of this approach is that the flattened datatype descriptor can be exponentially larger than the compact definition provided by the definition of that datatype (i.e., by the labeled directed acyclic graph that encodes the expression defining the datatype). Indeed, the flattened description can take a size proportional to the message size.

The alternative is to “evaluate” the expression that defines the datatype on the fly, using
a recursive traversal algorithm. This evaluation computes the sequence of displacements specified by the datatype expression and gets (puts) simultaneously the elements at these displacements from (into) the communication buffer.

A further degree of sophistication is needed, if one wishes to avoid copying and packing the entire message before it is sent out. One needs to be able to collect the "next" $n$ bytes of a communication buffer specified by such expression, and save the state of the traversal algorithm at that point; $n$ may be fixed (the size of a buffer), or variable (the amount of space currently available in the pipe).

We have implemented such on-line datatype parsing algorithm, so as to move data directly from the user communication buffer to the pipe, and vice-versa.

We show, in Figure 2, the bandwidth achieved for communication of a 3D char matrix $A[d_2][d_1][d_0]$ as a function of $d_0$ and $d_1$ ($d_2 = 16$ Meg/($d_1 + d_0$)). The matrix is transferred using a derived datatype with a 3-nested definition. As can be expected, for small $d_0$ the overhead is large. However, the graph shows that the bandwidth basically depends only on the size $d_0$ of the inner block and that 85% of the maximum achievable bandwidth is already achieved at $d_0 = 64$. I.e., the overhead of one transition in the datatype evaluation process is less than 15% if the "leaf" object has 64 bytes.

3 MPI libraries

One important feature of MPI is the support it provides for parallel SPMD libraries [10]. A parallel library function is executed collectively by a group of processes, each executing the same local procedure. These processes collectively invoke the function as each of them invokes the local procedure; they collectively return from the function execution as each of them returns from the local procedure.

The group of executing processes can be defined by an MPI communicator. The communicator may be statically predefined, if static scope is acceptable; e.g., if each process can belong to a unique executing group, and only one function invocation is active at any process at a time. It may be passed as a parameter at invocation, if dynamic scope is needed;
e.g., if each process may participate, at different times, in different executing groups, or different invocations may be active at a process at the same time, such as with recursive calls.

The communicator also provides a separate communication context for the invocation. This allows to start execution, in a loosely synchronous manner, without barrier synchronization: messages sent by a process after it invoked the library function will not be received by another process before it invoked the library function, and vice-versa. In the absence of a separate context, and without barrier synchronization, one could have a situation where process A that has not yet reached the function call, posts a wild card receive; process B, that has not reached the function call, sends a matching message to A; process C, that has already started the function execution, also sends a matching message to A. Rather than receiving the message from B, process A mistakenly receives the message from C. (This problem disappears if receive calls always specify a specific sender. However, such limitation may result in additional communication, in certain instances.) Similarly, no barrier synchronization is needed upon completion of the invocation. Furthermore, a point-to-point communication started by the sending process before the function invocation can be completed by the receiving process after the function execution has completed.

The executing processes may share additional information pertaining to the library execution, with the same lifetime as the executing group and the communication context that are defined by the communicator. E.g., static setup parameters, work place arrays, etc. One can attach this information to the same communicator, using the caching facility that is provided in MPI for communicators (see [8, Section 5.7]): a process can define a set of attribute keys; it can associate values with each key and each communicator.

3.1 MPI-IO
We illustrate the use of these features with an outline of an MPI-IO library that we are currently implementing. This library allows to perform I/O using a message-passing paradigm; it can be implemented on top of a regular (sequential) Unix file system, and can be easily modified to take advantage of a parallel file system, such as Vesta [2].

The paradigm provided by the MPI-IO library is that of an I/O server, that executes I/O calls on behalf of I/O clients. The I/O client is implemented by the MPI-IO client library that is linked with each executing process in a parallel MPI computation. The I/O server code can be implemented by a separate MPI process, that communicates via message-passing to the MPI clients, and has access to the file. It can also be implemented by a parallel I/O library executing on the client nodes. The former implementation is simpler; the latter can achieve better performance, especially when matched to a parallel file system. However, in either case, the semantics will be that of a parallel client, sequential server; parallelism on the server side is hidden from the user.

At any point in time, an open file is associated with a group of processes, the processes that can currently access the file. This group is represented by a communicator. Any MPI-IO call carries this communicator as a parameter: an OUT parameter for functions that open or close a file; an IN parameter for functions that read or write to the file. The obvious implementation is to cache the file descriptor with this communicator.

The communicator carries additional information on file connection parameters, that are set when a file is opened or reopened. In particular,

- A file can be opened in read-only, write-only or read-write mode.
- A file can be byte-oriented (C stream file) or record-oriented (Fortran file). In the
latter case, record size may be fixed or variable.

- File access may be sequential or direct.

- File accesses may be restricted to one record or span multiple records.

Additional parameters can be specified in the case of a parallel file.

An additional connection parameter specifies the sharing mode for file accesses. The three more interesting modes are

**private**: each process can access the file individually, and each process has its own file pointer; all file accesses are atomic.

**shared**: each process can access the file individually, and file accesses are atomic. The file pointer is shared, and all processes have a coherent value for it.

**collective**: all I/O accesses are collective calls executed by all processes in the group.

Open, Reopen and Close calls are collective calls executed by all processes in the communication group. This guarantees that all processes have a consistent view of the file status and connection parameters.

If I/O accesses are executed individually, then an I/O access is similar to a point-to-point communication between client and server: a Read call is semantically equivalent to a Receive from the server; a Write call is semantically equivalent to a Send to the server. A Read or Write call can be blocking, or nonblocking. In the latter case, the call behaves like a nonblocking MPI communication call: it returns a handle to a Request Object that can be later used to query for completion using an MPI call such as MPI_WAIT, or MPI_TEST, etc. This provides the functionality of synchronous and asynchronous I/O. A Read or Write call can use the same datatype facilities as MPI Send or Receive calls for specifying the data layout in the caller memory. This provides the functionality of an I/O vector list, and more.

Collective I/O calls are similar to collective MPI calls with a designated root, except that the root is the I/O server, rather than being one of the calling processes. Thus, a collective Read-Broadcast will read data from file and replicate it to all calling processes. A collective Read-Scatter will read data and break it into disjoint segments; a separate segment is sent to each calling process (as in MPI, there are two versions to this call; one where all segments have the same size, and one where they can differ in size). A collective Write-Gather will get a data segment from each process, and write the concatenation of these segments onto the file (here, too, there are two versions).

Both the definition and the implementation of the MPI-IO library is a fairly straightforward consequence of the MPI definition. Unix reduces all communications to file I/O. MPI, quite naturally, suggests a programming style where all communications, including file access, reduce to message passing, and where all object descriptors become adorned communicators.

### 4 Future extensions

Due to lack of time and/or lack of consensus, the MPI forum has not tackled some issues that are important for message-passing computation, in certain environments. Paramount among those, are dynamic process management and communication paradigms such as active messages or put and get. The following discussion is intended to show that such paradigms fit naturally as extensions to MPI.
4.1 Dynamic process management

MPI does not specify a mechanism for allocation of processes to processors; this is assumed to be external to MPI. In so much as process allocation does not affect the communication semantics, such approach is justified. Such is the case when the number of executing processes is fixed. The MPI initialization procedure will define the “universal” communicator MPI_COMM_WORLD to encompass all executing processes; each of these processes can be identified by its rank in MPI_COMM_WORLD.

New issues arise with dynamic process creation, where new processes can join an ongoing parallel computation. All communications in MPI occur within the universe of a predefined communication group, specified by a communicator. MPI assumes that changes in the communication environment of a process (the set of possible destinations for point-to-point communication, and of possible groups for collective communication) occur only as a result of calls executed by that process. Indeed, new communicators can be defined at a process only as a result of collective calls executed by groups that include this process; and existing communicators do not change value. Thus, if MPI communicator creating calls are synchronizing, then a process can anticipate at any point in time all possible contexts and sources for incoming messages. This is in contrast to PVM [4], where processes can join and leave groups or the global PVM communication universe asynchronously.

The advantage of the synchronous MPI model is that one gets a cleaner semantics, with less nondeterminism: a communicator always identifies a unique, well-defined set of processes. The validity of a destination in a send call does not depend on the timing of some asynchronous execution by another process. The implementation of the communication subsystem is simplified if one does not need to deal with messages from unknown sources, and error detection is improved. Also, no “out-of-band” communication is required for changes in the communication environment: all communicator creating operations can be implemented on top of the point-to-point calls.

It would seem, at first glance, that this synchronous model is not compatible with dynamic process creation. However, we shall show that we can preserve the synchronous aspect of MPI, in that changes in the communication environment of a process occur only as a result of explicit calls executed by that process, and processes need not cope with incoming messages from unknown sources.

Assume that new processes are generated by an “MPI_SPAWN” call executed by an individual process. The parameters of such call will include information on the number $n$ of spawned processes, the code to execute at these processes and, possibly, the locations of these processes. The newly spawned processes are not able to communicate with preexisting processes in the MPI universe, as these processes may not be aware of their spawning. However, they can be initialized so as to be able to communicate with their parent, and their siblings.

To do so, the call will include a communicator parameter (newcomm). The call returns at the callee a communicator for the group of processes that include the parent (rank 0 and the newly spawned children (ranks 1 to $n$). At each of the children MPI_COMM_WORLD is initialized to point to that same group of processes. Thus, each of the new processes starts executing in a communication universe that include its parent and its siblings.

At this point, a call to MPI_INTERCOMM_CREATE can be used to create an intercommunicator that consists of two groups: the initial old communication universe, defined by MPI_COMM_WORLD at the parent process, and the new communication universe, defined by newcomm at the parent process and by MPI_COMM_WORLD and the children processes. (An intercommunicator allows processes from one group to commu-
nicate with processes from another group, and vice-versa, using rank within the respective group to identify the communication partner; see [8, Section 5.6]. It is created by a call to `MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm)`, where `local_comm` is a communicator for the local group that includes the calling process, `local_leader` is a designated process in this group, `peer_comm` is a communicator that need be defined at the `local_leader` process, and `remote_leader` is the rank of the leader of the other group in `peer_comm`. In our case, the parent process will be both local and remote leader. The MPI specification requires that groups that are combined in an intercommunicator be disjoint but it is innocuous to relax this requirement so as to allow the local and remote leaders to be one and the same.

Once an intercommunicator is created, the two groups in the intercommunicator can be merged with a call to `MPI_INTERCOMM_MERGE` (a slight modification of the current definition is needed to accommodate the overlap between the two groups).

One may provide a shorter procedure, using functions that combine two, or all three of the steps outlined above. E.g., one could have a collective call that augments an existing communication group with newly spawned processes, in one call. And, of course, one cannot implement anymore the communicator generating calls used above on top of regular MPI point-to-point communication. Indeed, one need go outside MPI in order to establish the first communication channel between processes that were not set to communicate when MPI was initialized (e.g., because one or both did not exist yet). For this reason, it might be preferable to use syntactically distinct calls to establish the first communication channel between two processes. But the synchronous aspect of changes in the communication environment is preserved.

### 4.2 Message handlers

It is sometimes useful to be able to invoke a user defined handler in order to handle the incoming data at the receiving process. This gives more flexibility in the selection of a handling mechanism for incoming data; it leads to better performance on systems where most of message reception overhead is software executed by the receiving process [3]. A possible mechanism for such functionality is provided by a `receive&Call` function, as available in NX [9]. This is a `receive` call that has an additional parameter, a pointer to a function. The function is invoked after the receive completes. A possible MPI-like syntax for such (nonblocking) `receive&Call` can be

```c
MPI_RECEIVE&CALL(buf, count, datatype, source, tag, comm, handler)
```

(There is no need for a returned Request Object, as the call of the handler function signifies completion of the receive.)

This function achieves the goal of invoking an arbitrary handler, but does not allow to avoid the overhead of a regular receive. A variant, which may be more efficient on some systems, would be to have the incoming message passed as an argument to the handler: the function can be passed a pointer to the buffer where the incoming message is stored. A possible MPI-like syntax for this type of `receive&Call` can be

```c
MPI_RECEIVE&CALL(count, datatype, source, tag, comm, handler)
```

(same as before, except that no receive buffer is specified). It is not necessary to use one fixed handler interface (e.g., a fixed parameter list). The MPI library can be informed of the handler interface using a similar approach as that used in MPI for user-defined reduction functions.

In both message. If a new message is received, the first receive call will clear the message. The receive handler function will be called when the receive arrives; see Section 3.9 for details.

```c
MPI_RECEIVE(buf, count, datatype, source, tag, comm, handler)
```

The outcome of a `receive&Call` call can be used to determine which message has arrived. For a multithreaded application, this is particularly useful when the message is not ordered.

### 4.3 Get primitive

One disadvantage of the above `receive&Call` function is that the user must provide memory for receiving the message. A possible workaround is to use a receive function that allows the user to receive messages in a given memory location (e.g., local memory or a collective memory). One possible function is

```c
MPI_GET(buf, count, datatype, source, tag, comm, handler)
```

This function is similar to `receive&Call`, but the user must provide a memory location for receiving the message. The user can then use this memory location to store the received message.

For such a function, the user must provide a memory location for the message and a pointer to the buffer where the incoming message is stored. The MPI library can be informed of the handler interface using a similar approach as that used in MPI for user-defined reduction functions.
In both variants, a posted receive is “consumed” after it has handled one incoming message. If the handler code need to be continuously posted, then the handler code needs to post a new receive, when it gets invoked. Such approach is necessary for a Receive&Call of the first type: the new receive should be posted only after the receive buffer has been cleared. The second type of Receive&Call, does not have this problem; one can have a receive handler that is “continuously posted” and is invoked as soon as a matching message arrives; several invocations may proceed concurrently. This concept of a continuously posted receive handler matches MPI concept of persistent communication request (see [8, Section 3.9]). One would create such persistent handler by a call to

```c
MPI_RECEIVE&CALL_CREATE(count, datatype, source, tag, comm, handler, request)
```
dele the handler by a call to `MPI_REQUEST_FREE`.

A Receive&Call interface can be used to implement specific actions on message arrival. For example, such interface can be used to interact with a thread scheduler in a multithreaded environment. It can be used to implement the **Put** and **Get** functions, below. More generally, it can be used to provide the equivalent of a Remote Procedure Call interface.

### 4.3 Get and Put

One disadvantage of message-passing communication is that both sending and receiving processes must execute code in order to transfer data from the sender memory to the receiver memory. In many parallel computations data distribution is static but the data access pattern is data dependent. In such situation, a process that needs to access a remote value “knows” the location of this value; however, the process that has this value in its local memory may not “know” which process wishes to access this value. An expensive collective coordination protocol is needed in order to generate Sends to match the Receives [11].

A **Get** primitive, which allows a process to read data in a remote memory alleviates this problem. This, coupled with a **Put** primitive that allows to write data in a remote memory, provides a partial equivalent to a shared memory model: Any process can access any memory location. However, caching and coherence are under software control. A Receive&Call or an active message interface can be used to implement in software **Put** and **Get** primitives.

A **Get** can be added to MPI with a call that provides Receive parameters for the calling process, and Send parameters for the remote process. The syntax of a blocking **Get** might be

```c
MPI_GET(localbuf, localcount, localtype, remotebuf, remotecount, remotetype, source, comm)
```
The outcome of the execution of this call is as if the source node executed a Send with parameters `remotebuf`, `remotecount`, `remotetype` and this message was received with parameters `localbuf`, `localcount`, `localtype`. A **Put** function can be similarly defined, and so can nonblocking variants of these calls. Support of such functions require limited enhancements to an MPI implementation, assuming that a callback mechanism is available for handling unsolicited messages.

For such mechanism to be useful, one needs to be able to transfer across processes pointers and datatype objects. Note that pointers are not required to be portable – an address sent by a process will only be used to address a buffer at that process.
4.4 Interrupt vs polling

The EUI-H library allows the user to select at start-up time the preferred Receive mechanism: polling or interrupt. When polling is used, data is pulled out of the adapter only when the communication library is invoked by the user, with the following exception: to ensure progress, the communication subsystem is invoked periodically at timer interrupts. When interrupt is used, an incoming packet causes an interrupt that activates the communication library, unless the communication library is already executing. This increases the overhead for a Receive of zero length message to more than 200 μsec, because of the overheads associated with interrupt handling and invocation of the user-space library from the system. (However, the cost of signal handling is paid only once for a sequence of packets that arrive within a short period. Once invoked, the communication library keeps pulling data from the adapter, until it is quiescent.)

Consider a message-passing environment where processes are single-threaded. An incoming message that is buffered in the communication subsystem will not be processed by the receiver until a matching blocking Receive is executed, or until the Wait or Test that completes the matching non-blocking Receive is executed. Therefore, there is no reason to process an early arriving message, unless this is needed to “unclog” the communication subsystem. This justifies a polling implementation of Receives.

EUI-H provides only a small amount of buffering with each pipe. Thus, transfer of a sizeable message will not complete unless both sender and receiver are simultaneously executing the communication library. This means that, with polling, non-blocking communication will often behave like blocking communication, with data transfer occurring only when both processes execute the blocking Wait call. On the other hand, with interrupt, an incoming message will be moved to its destination in memory when it arrives, if a non-blocking Receive is already posted. In effect the “interrupt handler” provides the services one would expect from a smart adapter (or communication coprocessor): asynchronous transfer of data from the network into the receiver memory.

Immediate handling of incoming messages becomes even more important when processes are multi-threaded, or when message reception leads to the invocation of a handler. In both cases, message reception may cause a change of control. Users might take advantage of this arrival-driven scheduling mechanism in order to implement the equivalent of a Remote Procedure Call. The execution of a Get request is the simplest example of such RPC service. In such cases, sender and receiver are not in symmetric situations. While the receiver is busy doing useful work, the sender may idle, waiting for a reply or an action from the receiver. It may be advantageous to pay the cost of interrupt handling at the receiver, in order to unblock the sender.

We have found hard to predict which of the two methods will yield better performance, in practice. It may be interesting, in future systems, to allow users a finer grain control of the Receive mechanism. Note, however, that the issue becomes moot once processing of incoming messages is off-loaded to an intelligent communication coprocessor.

5 Conclusion

Our current experience with MPI indicates that, notwithstanding the large number of functions and options, basic communication can be implemented to be as fast in MPI as with simpler libraries, such as EUI. Furthermore, the added functionality proves useful, both in order to achieve better performance for more complex communication patterns and as support to parallel libraries. New functionality can be easily added to MPI, without
breaking the mold. One can hope that the rapid introduction of both public domain and proprietary implementations of MPI will speed up the development of industry-strength portable parallel libraries and application packages.

References


