The IBM external user interface for scalable parallel systems

Vasanth Bala a, Jehoshua Bruck b, Raymond Bryant a, Robert Cypher b, Peter de Jong c, Pablo Elustondo b, D. Frye c, Alex Ho b, Ching-Tien Ho b, Gail Irwin a, Shlomo Kipnis a, Richard Lawrence a, Marc Snir a,⁎

a IBM Research Division, T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, USA
b IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA
c IBM Corporation, Highly Parallel Supercomputing Systems Laboratory, Neighborhood Road, Kingston, NY 12401, USA

(Received 16 May 1993; revised 16 August 1993)

Abstract

The IBM External User Interface (EUI) for scalable parallel systems is a parallel programming library designed for the IBM line of scalable parallel computers. The first computer in this line, the IBM 9076 SP1, was announced in February 1993. In essence, the EUI is a library of coordination and communication routines that can be invoked from within FORTRAN or C application programs. The EUI consists of four main components: task management routines, message passing routines, task group routines, and collective communication routines. This paper examines several aspects of the design and development of the EUI.

Key words: Message passing; IBM External User Interface (EUI); Parallel programming library; IBM 9076 SP1; FORTRAN; C

1. Introduction

The IBM External User Interface (EUI) for scalable parallel systems is an application programming interface that was designed for the IBM line of scalable
parallel computers. The first computer in this line, the IBM Scalable POWERparallel System 9076 SP1, was announced in February 1993. The design of the EUI is aimed at providing a scalable and efficient parallel programming environment over a wide range of parallel products from IBM. The EUI is a library of coordination and communication routines that can be invoked from within FORTRAN or C application programs.

Over the past several years, a large number of programming environments and communication libraries for parallel computers have been developed, including PVM [3], LINDA [5], PICL [9], PARMACS [11], Zipcode [13], Express [15], the NCUBE/2 library [12], the CM-5 library [14], and the iPSC/860 library [16]. The design of the EUI borrows some of the popular communication concepts that already exist in many of these libraries, and in addition, provides several novel operations. The primary goal in the design of the EUI was to provide a simple and efficient set of well-understood operations for coordination of and communication among processors.

Although the EUI was designed such that it could be used on several platforms, the main target of the initial release is a tightly-coupled distributed-memory MIMD parallel computer, such as the SP1. In such a system, an application that uses the EUI is able to gain control of a dedicated set of RS/6000 RISC processors that communicate via a high-speed interconnection network. The interconnection network in the SP1 provides node-to-node communication with performance that is roughly identical between any pair of nodes. Another platform for the EUI is a cluster of RS/6000 workstations, in which an application may be time-sharing the processors with other jobs and where communication between processors is not as predictable as in a tightly-coupled dedicated multicomputer. Finally, the EUI is also available in a time-shared environment on a single RS/6000 workstation. Such an environment is used primarily for initial application development.

Portable implementations of the EUI may use standard Unix communication protocols (such as IP) as their communication layer. Computers aimed at a more tightly coupled and performance-oriented market will use communication protocols that take advantage of high-speed communication networks and are tailored to the requirements of higher-level communication layers. This should provide opportunities to significantly improve performance. Several such implementations of the EUI on different platforms have been developed.

The design of the IBM External User Interface for scalable parallel systems was influenced by several prototype systems and environments developed at IBM Research and at the IBM Highly Parallel Supercomputing Systems Laboratory (HPSSL). Of particular interest are the Vulcan system [4,17], the Viper Operating Environment [7,10], and the Venus communication library [2].

Vulcan [4,17] is a prototype of a scalable, distributed-memory parallel computer based on Intel i860 compute-nodes and a low-latency, high-bandwidth, multistage interconnection network. The Vulcan system was designed and built at IBM T.J. Watson Research Center and aimed at exploring massively parallel computers. The Vulcan project started in 1988 and was successfully completed in early 1992.
The Vulcan prototype consists of 16 nodes connected by a multistage network of fast switches that is very similar to the high-performance network used in the SP1. Viper [7,10] is a light-weight operating system for Vulcan. Viper features task management facilities, multithreading, and point-to-point communication protocols. Viper was designed as a kernel for distributed-memory parallel computers.

Venus [2] is a prototype design of a collective communication library for massively parallel computers. Venus incorporates features such as Process Groups and Process Channels and supports coordination of and communication among processes in dynamic situations.

In January 1992, a joint effort by the authors of this paper started to define a programming interface for the line of parallel computers from the newly formed IBM HPSSL. This programming interface was termed the External User Interface or EUI. After several months of intense design of the various components of the EUI, a first draft of the design was sent to several external reviewers. The reviewers' comments were studied and the design of the EUI was modified accordingly. The final specification of the EUI was completed in the summer of 1992.

The implementations of the EUI followed shortly after freezing the specification. 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 Supercomputing-92) and clusters of RS/6000 workstations. The first release of the EUI on the SP1 occurred in early 1993.

During its design and implementation, the EUI had a major influence on the Message Passing Interface (MPI) [6] standardization effort that started in late 1992 and continued in 1993.

The rest of this paper is organized as follows. The next section presents definitions and provides an overview of the four basic components of the EUI. These components are then described in order: the task management routines in Section 3, the message passing routines in Section 4, the task group routines in Section 5, and the collective communication routines in Section 6. Section 7 describes future extensions of the EUI. Finally, some concluding remarks are included in Section 8.

2. Preliminaries

2.1. Terminology of the EUI

The terminology in the field of parallel computing is often confusing and many terms are overloaded. In the EUI, we use the following terms and definitions:

**Job:** A unit of work defined by a user to be done by the parallel system.

**Task:** An execution environment that includes an address space and allocated system resources. Messages are addressed to tasks. Currently, each task has a single thread of control.
**Partition**: A parallel execution environment for a job. A partition consists of a number of tasks, all of which are dedicated to the execution of a single job.  

**Group**: A user-defined collection of tasks that is a subset of a partition. Groups are used for coordination and communication operations that involve multiple tasks.  

**Node**: A processing unit consisting of an RS/6000 processor, its memory, a network adapter, and an optional disk.  

**Application buffer**: A location in a task’s address space that is used as the source or destination of a communication operation.

### 2.2. Overview of the EUI

The EUI consists of four main components. Here, we outline the functionality of each of these components. A more detailed description of each component is provided in a corresponding section.

- **Task management**: Routines for querying and controlling the parallel programming environment. Users can query things like the number of tasks defined for a job, the amount of memory in the system, the values of programming wildcards, and the legal types available for message-passing. The EUI has default settings for many things in the programming environment, but the user can change some of them.

- **Message-passing**: Constructs for blocking and non-blocking message-passing similar to those found in many distributed-memory parallel environments. In addition, there are routines for obtaining the status of non-blocking sends and receives.

- **Task groups**: The collective communication routines are designed to operate on groups of tasks, rather than on all tasks. This component contains routines that allow dynamic creation of such groups and querying and manipulating of existing groups.

- **Collective communication**: Routines designed to provide functions such as broadcast, reduce, scatter, gather, etc., in single commands, so that users need not code these operations with point-to-point message-passing.

### 2.3. Language binding

The EUI is implemented as a collection of C subroutines which are invoked from FORTRAN, C, and C++. From FORTRAN, the EUI is invoked as subprograms and arguments are passed by reference. From C, the EUI is invoked as functions, arguments that return a value are passed by reference, and arguments that do not return a value are passed by value. From C++, the EUI is invoked as C functions.

In FORTRAN, all user and system errors terminate the parallel job. In C and C++, user errors and some system errors are returned as negative return codes.
3. Task management

The EUI is designed for systems that support the execution of parallel programs in two main types of environments: dedicated partitions and time-shared partitions. A dedicated partition has the following properties:
(i) a parallel program has exclusive use of a dedicated set of nodes;
(ii) tasks are assigned to nodes on a one-to-one basis;
(iii) tasks cannot migrate between nodes; and
(iv) the user can control the partition.
A time-shared partition, on the other hand, has other properties:
(i) tasks may share the same physical nodes;
(ii) tasks may have different amounts of computing resources;
(iii) the system will attempt to allocate a roughly equal number of tasks to each node;
(iv) task migration between nodes is not supported.

The EUI environment in which parallel jobs execute has the following characteristics.
• When a job is submitted for execution, the number of tasks must be specified. The system will allocate the requested number of tasks and will load the code on the nodes that will execute the tasks. The tasks will begin executing asynchronously. The number of tasks is fixed throughout the execution of a job, that is, new tasks cannot be created.
• Each task is uniquely identified by a task id. These ids are integers ranging from zero to the number of tasks minus one. Tasks can obtain their ids and the total number of tasks in a job.
• Normal termination of a task occurs when it encounters the end of the program (such as a FORTRAN STOP.) Normal termination of one task does not stop other tasks.
• Termination of the job as a whole can be accomplished in one of three ways:
  (i) when all tasks have terminated normally,
  (ii) when some task has encountered a fatal error, the system stops all the tasks and cleans up the job, or
  (iii) upon processing an MP_STOPALL command. The MP_STOPALL command, executed from any task in the job, terminates all tasks in the job.

The main restriction in this model is that the number of tasks dedicated to run a parallel program is fixed at load time. Thus, there are no primitives to spawn new tasks at run time. This should not be viewed as overly restrictive, since tasks need not perform the same computations and since sufficient physical and virtual memory can be provided to support large executable files. On the contrary, given that each task has a unique task identifier, the user can have each task perform unique computations based on its task id.

3.1. List of task management primitives

• MP_ENVIRON: Obtain the number of tasks assigned to a job and the relative task number.
• **MP_STOPALL**: Terminate all tasks, cancel the parallel job, and return an abnormal termination condition. This should be used only for abnormal termination, since other tasks are stopped regardless of whether they have finished. This is, however, a protected stop that flushes all buffers and closes all files before the tasks are terminated.

• **MP_TASK_QUERY**: Obtain information on the current programming environment of a task.

• **MP_TASK_SET**: Set user-definable parameters for the calling task. For predictability, **MP_TASK_SET** should be called only at the beginning of a job and before invoking any communication.

### 3.2. Example

```c
C - Each task finds out its id and the total number of tasks.
call MP_ENVIRON (ntasks, taskid)
```

### 4. Message passing

Pairs of tasks communicate by issuing matching send and receive commands. In addition to the destination task and the message to be sent, the send command includes a **type** parameter that can be used to identify a message. The receive command specifies where the message should be placed, as well as the desired source and type of message. The source and/or the type values may be a wildcard (thus matching any message). Multiple messages, with the same type, sent from a single source to a single destination will arrive in the order in which they were sent. Messages sent from multiple sources to the same destination are non-interfering, in that the receiver can always receive the message from the specified source, even if messages from other sources arrived earlier. Message truncation (when the receiver's application buffer is smaller than the incoming message) is treated as an error condition unless the task has explicitly set message truncation to be valid.

Both blocking and non-blocking send and receive operations are supported in the EUI. A **blocking send** will only return when the application buffer in the sending task is free to be reused; however, completion of this call does not imply that the message has been received into the application buffer in the destination task. A **blocking receive** returns after the receive operation completes and the message has been copied into the application buffer of the receiving task.

A **non-blocking 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 **non-blocking receive** 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.

Two primitives are provided to let the user monitor the progress of the non-blocking sends and receives. One non-blocking function reports the status of a pending message and the other function blocks (waits) until the desired (incoming or outgoing) application buffer is available.

4.1. List of message passing primitives

The message-passing calls provided by the EUI are summarized below, with the blocking send and receive calls prefixed with B (for blocking).

- **MP_BSEND**: Send a message and block until the application buffer in the sending task is free for reuse.
- **MP_BRECVC**: Receive a message and block until the requested data is available in the application buffer in the receiving task.
- **MP_BSENDRECV**: Send a message and receive a message. Block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.
- **MP_SEND**: Request the system to send a message from a given application buffer and return immediately. A message id is returned for handling the pending message status.
- **MP_RECV**: Request the system to receive a message in a given application buffer and return immediately. A message id is returned for handling the pending message status.
- **MP_STATUS**: Check the status of a specified non-blocking send or receive operation.
- **MP_WAIT**: Block until a specified non-blocking send or receive operation completes. If the parameter **almsg** is used as a message id, the call blocks until all non-blocking operations complete. If the parameter **dontcare** is used, the call blocks until the next send or receive operation completes.

4.2. Deadlock

Users may write programs that reach deadlock. For example, a task may execute a blocking receive for a message with a type that is never sent to the receiving task; the receiver is then blocked forever. Program A below illustrates a deadlock condition: tasks are arranged in a logical ring. Each task sends a message to its right and receives a message from its left. The order of the send and receive operations was (mistakenly) inverted. Since blocking receives are used, all tasks block on the receive call and no message is ever sent.

C

C PROGRAM A

C Tasks perform a circular shift of data. Program deadlocks.
The deadlock problem of Program A has been fixed in program B below, by correcting the order of the send and receive operations. However, Program B exhibits a more subtle problem. Since a blocking send is used, a task cannot start executing the receive statement until its outgoing message has been copied out of its task memory. Depending on the communication protocol used, either the message is copied out of task memory into system buffer space, if space is available, or the message is transmitted directly to a task memory buffer in the destination task after the MP_BRECV is issued. In the first case, if the system has run out of system buffer space, the MP_BRECV will not be executed, and the program will deadlock. In the second case, the MP_BRECV will never execute, and the program will always deadlock. At least one message must be copied out of the sending task memory before the matching receive can execute. Thus, Program B may progress only if there is enough system buffer space to store at least one message. If, as is more likely, messages can be buffered only at the sending or at the receiving node, then this program will actually require enough space to store at least one message at a single node. Such programs, that can progress only if a sufficient amount of system buffer space is available, are called unsafe.

C
C PROGRAM B
C Tasks perform a circular shift of data. Program may deadlock.
C

call MP_ENVIRON (ntasks, taskid)
right = mod (taskid + 1, ntasks)
call MP_BSEND (outmsg, msglen, right, type)
call MP_BRECV (inmsg, msglen, source, type, nbytes)

The outcome of the execution of an unsafe program is implementation dependent. In particular, the success of such a program depends on the size of the messages sent, and on the amount of memory available for system buffers. An unsafe program may deadlock if there is not enough system buffer space. Program C below is a corrected version of the circular shift program: the blocking send has been replaced by a non-blocking send. The sender blocks, waiting for the send to complete, only after executing the receive. This program is safe with respect to buffer overflows, as it can complete even if no system buffer space is available. The outgoing message can be copied directly into the receiving task memory, after the receive call occurred.
We give a rule for writing programs that do not deadlock, regardless of the size of the system buffers. We call a program race-free, if each receive operation is always paired with the same send operation, regardless of the speeds of the tasks and the communication subsystem. Now consider an implementation of a race-free program without any system buffer space. Each send operation (or more formally, each MP_SEND call or MP_WAIT call which follows an MP_SEND call) will be forced to block until its unique matching receive operation is issued. Finally, there is one additional system resource that is consumed by (non-blocking) sends and receives, namely message identifiers. The user can assume that a very large number of message identifiers are available, but a pathological program which issues huge numbers of non-blocking sends and receives without matching wait calls could conceivably run out of message identifiers. As a result, a program is said to be safe if it is race-free, it does not run out of message identifiers, and it is free of deadlock, even when every send operation is forced to block until its matching receive operation is issued.

There is one other minor detail which was not specified in the above definition of a safe program. In the definition, it was stated that a call to MP_WAIT which follows a call to MP_SEND may block until its matching receive operation is issued. The detail which was not specified is what happens when the call to MP_WAIT uses the argument allmsg. In this case, the call to MP_WAIT should be viewed as being a sequence of calls to MP_WAIT for each of the pending send and receive operations, and no assumptions can be made about the order in which the pending send and receive operations appear in this sequence.

The example below illustrates that deadlocks may also be introduced because of the ability to select incoming messages by type. In Program D, task 1 consecutively sends a large number of messages with types 1,2,... to task 2, and task 2 receives these messages by type in reverse order. Because task 1 uses blocking sends and task 2 uses blocking receives, the program is unsafe as all outgoing messages (with the possible exception of the last) must be buffered in system space for the program to succeed. The program may deadlock, depending on the values of LARGE (the number of messages sent) and msglen (the message size).

C PROGRAM D
C Task 1 floods task 2 with a sequence of messages that are received in the reverse order by task 2. Program is unsafe.

C

```
call MP_ENVIRON (ntasks, taskid)
if (taskid .eq. 1) then
   do i = 1, LARGE
      call MP_BSEND (outmsg(i), msglen, 2, type(i))
   end do
else if (taskid .eq. 2) then
   do i = LARGE, I, -1
      call MP_BRECV (inmsg(i), msglen, 1, type(i), nbytes)
   end do
end if
```

However, if non-blocking receives are used, the only possible problem is running out of message identifiers. Provided that the value LARGE does not exceed the number of available message identifiers, Program E below is safe. Because all of the non-blocking receives will be issued, each of the blocking sends will have a matching receive and deadlock will not occur.

C

```
C PROGRAM E
C Task 1 floods task 2 with a sequence of messages that are received in the reverse order by task 2. Program is safe if number of msg ids C is not exceeded.
C

call MP_ENVIRON (ntasks, taskid)
if (taskid .eq. 1) then
   do i = 1, LARGE
      call MP_BSEND (outmsg(i), msglen, 2, type(i))
   end do
else if (taskid .eq. 2) then
   do i = LARGE, 1, -1
      call MP_BRECV (inmsg(i), msglen, 1, type(i), msgid(i))
   end do
   call MP_WAIT(allmsg, nbytes)
end if
```

5. Task groups

Conceptual collection of computational entities into logical groups arises naturally in the modeling of many physical, biological, and social processes and in the
use of many numerical techniques. The Task Group routines facilitate the creation and manipulation of logical task groups. Formally, a task group is an ordered set of tasks that has a system-wide unique identifier (called a ‘group id’ or ‘gid’). All of the collective communication routines operate within groups. All of the tasks that are created at the initialization of an application belong to the predefined group allgrp.

New groups can be created either explicitly by specifying the list of tasks that will form a group or by partitioning an existing group into subgroups. The first approach is accomplished by calling the MP_GROUP routine, and the second approach is accomplished by calling the MP_PARTITION routine. A call to MP_GROUP specifies a membership array of task ids glist (in the order in which they will appear in the group), the size of this group gsize, and a label to be associated with the group label. It returns as an argument the group identifier gid. If a task wishes to define a new group by calling MP_GROUP and supplying a membership array, then all other tasks specified in the membership array must issue a corresponding MP_GROUP call with the same values for the glist, gsize, and label parameters. Other tasks outside the group need not call MP_GROUP.

A call to MP_PARTITION specifies a label, a key, and a parent group identifier parent_gid, and it returns as an argument a group identifier gid. All of the tasks in the specified parent group must issue corresponding calls to MP_PARTITION. Those tasks that provide identical values for the label parameter are placed in the same group (that is, they receive the same gid value), and tasks that provide different values for the label parameter are placed in different groups. Thus, new groups are created that partition the parent group according to the values of the label parameters. The parent group itself remains unaffected by the call to MP_PARTITION.

5.1. List of task groups primitives

The Task Group routines in EUI are provided for creating logical groups and for manipulating and querying group information. The routines are summarized below:

• MP_GROUP: Create a task group by explicitly specifying the tasks participating in the group.
• MP_PARTITION: Partition an existing task group based on a locally supplied label.
• MP_GETSIZE: Get the size (the number of tasks) of an existing task group.
• MP_GETMEMBERS: Get the ordered array of task ids of an existing task group.
• MP_GETLABEL: Get the user-supplied label of an existing task group.
• MP_GETRANK: Get the rank of a task in a task group.
• MP_GETTASKID: Get the id of the task with a certain rank in a task group.

5.2. Example

The following code segment illustrates the simplicity and usefulness of group creation routines such as the MP_PARTITION. Assume that each task in the group
allgrp has been assigned a row_id and a col_id in a two-dimensional mesh topology.

c
C Create logical groups consisting of tasks with the same row_id, 
C and there is no preferred order of the tasks in the logical 
C groups.
C
key = 0
   call MP_PARTITION (allgrp, key, row_id, row_group_id)
c
C Create logical groups consisting of tasks with the same col_id. 
C Order of tasks in a group should be in ascending order of 
C row_id.
C
key = row_id
   call MP_PARTITION (allgrp, key, col_id, col_group_id)

Using the returned row_group_id or col_group_id as handles, collective 
communications can be performed within logical groups without the burden of 
enlisting the tasks explicitly. In some applications, enlisting the tasks for collective 
communications may be tedious but not difficult. In others, as in the simulation of 
dynamic systems or in event-driven simulations, it is both tedious and difficult to 
identify which tasks contain the data that needs to be communicated. The MP_PAR-
TITION routine provides an effective mechanism to construct such conceptual 
grouping easily.

6. Collective communication

The Collective Communication (CC) routines free the user from programming 
frequently needed operations involving multiple tasks (such as broadcast, reduce or 
gather) with the point-to-point message-passing routines. Furthermore, these rou-
tines can be highly optimized for performance. (Measured from application space 
to application space, the performance of the interconnection network is essentially 
independent of the relative positions of communicating processors, so the CC 
routines are not dependent on the exact set of tasks in the user's partition.)

All CC routines have a group identifier (gid) that identifies the group of tasks 
that participate in the collective operation. This identifier, in addition to specifying 
which tasks participate, also specifies the order of the tasks for those CC routines 
for which order is important. Whenever any task in a group calls a CC routine 
(such as MP_BCAST), then all tasks within the same group must participate in that 
operation by calling the same CC routine with the same gid. Furthermore, all the 
tasks that perform a CC routine must use consistent parameters. (For example, 
when calling MP_BCAST, every task must specify the same task as the source of the
Tasks that do not participate in a given CC routine are free to perform other CC routines, point-to-point communication, or local calculations while the given CC routine is in progress.

6.1. List of collective communication primitives

- **MP_SYNC**: Barrier synchronization in a group. Each task, when reaching the MP_SYNC call, blocks until all tasks in the group reach the corresponding MP_SYNC call.
- **MP_SHIFT**: Shift data up or down some number of tasks in the group.
- **MP_BCAST**: Single-source broadcast of a message from one task to all tasks in the group.
- **MP_SCATTER**: Distribute distinct messages from a single source task to each task in a group. In an N-task group, the application buffer of the source task consists of N equal-sized blocks. The ith block, for 0 ≤ i < N, is destined for the ith ranked task in the group.
- **MP_GATHER**: Gather distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MP_SCATTER.
- **MP CONCAT**: Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group.
- **MP_INDEX**: Index communication in a group, in which each task performs an MP_SCATTER operation. In an N-task group, the application buffer of each task consists of N blocks. In this operation, task i exchanges its jth block with the ith block of task j.
- **MP_REDUCE**: Apply an associative (not necessarily commutative) reduction function and place the result in one task. EUI predefines several reduction functions and users can define others.
- **MP_COMBINE**: Apply an associative (not necessarily commutative) reduction operation and place the result in all tasks in the group.
- **MP_PREFIX**: Parallel prefix operation. Given an ordered group of N tasks, p0, p1,...,pn−1, an associative operator ⊕, and a variable v_i at p_i, the resulting value for p_i is v_0 ⊕ v_1 ⊕ ... ⊕ v_i.

6.2. Example

Assume there are N × N tasks holding an image of N × N pixels initially. The goal is to scroll the image up to bring the pixel with the maximum gray level to the top row (in a minimum number of steps if there is more than one pixel with the same maximum gray level).

```c
C Initialization
C
msglen = 4
```
call MP_ENVIRON (ntasks, taskid)
C
Assume ntasks = N*N. Calculate the position of the task.
C
mycol = mod (taskid, N)
myrow = taskid/N
C
Find out the maximum gray level of the image:
C outmsg = mypixel, inmsg = maxlevel, and reduction-func = integer-maximum.
C
call MP_COMBINE (mypixel, maxlevel, msglen, i_vmax, allgrp)
C
A task with a maximum-level pixel assigns its 'mystep' to its row position.
C
if (mypixel .eq. maxlevel) then
  mystep = myrow
else
  mystep = N
end if
C
Find the minimum number of steps to scroll up a maximum in the top row:
C outmsg = mystep and inmsg = minstep.
C
call MP_COMBINE (mystep, minstep, msglen, i_vmin, allgrp)
C
Create one group for each column of the 2-dimensional structure with:
C key = 0, label = mycol, parent_group = allgrp, and resulting group = col_gid.
C
call MP_PARTITION (allgrp, 0, mycol, col_gid)
C
Use 'max_step' to scroll image up. (Fifth argument 0 for a circular shift.)
C
call MP_SHIFT (mynewpixel, mypixel, msglen, -minstep, 0, col_gid)

7. Future extensions

This section briefly discusses several advanced features of the EUI that are not yet implemented on the SP1. Research on these features is ongoing.
7.1. Probe and cancel operations

Two routines that enhance the message passing capabilities of the EUI are MP_PROBE and MP_CANCEL. An MP_PROBE routine would allow a user to check for incoming messages without actually posting receives. The user could then decide where to receive an incoming message. An MP_CANCEL routine would allow a user to cancel any pending messages of non-blocking send or receive operations. This would allow for cleanup and for freeing buffers in unexpected situations.

7.2. Communication channels

Channels are a method of providing high-performance communication for repeated transfers between pairs of tasks. Channels are particularly useful for domain decomposition problems where the pattern of communication between tasks is regular and repeated (for example, messages exchanged during every iteration between tasks logically arranged in a grid).

Channels can provide an efficient communication method because, when a channel is created, the user supplies additional information to the communication subsystem. This additional information includes:
(i) the pair of tasks that will use the channel for repeated message exchanges;
(ii) a specification of the minimum and maximum message sizes to be allowed on this channel; and
(iii) a specification of the maximum number of messages expected to be simultaneously present in the channel.

This information allows the communication subsystem to determine the amount of storage that should be reserved for channels.

Once a channel has been created, it is used by a separate set of send and receive routines (distinct from the EUI message-passing routines). These routines do not specify the sending or receiving task; instead they specify only the channel id associated with this channel. This is sufficient since a channel represents a point-to-point connection with the end points specified at channel creation time. This allows the user to code such commands as 'send to left' by referring to the channel id allocated for communication with the processor on the left.

The creation of channels may require the exchange of several messages between the two tasks and, hence, is a synchronous and expensive operation. Therefore, channels should not be used for one-time-only communication patterns. However, since storage allocation in the communication subsystem can be done at channel creation time and delivery mechanisms can be implemented that minimize flow control overheads on the channel, then for repeated message transfers the extra time spent in channel setup can easily be paid for in reduced communication costs.

7.3. Collective communication modes

The collective communication routines can run in two different modes of operation called barrier and non-barrier. When a routine is executed in barrier
mode, no task can complete its call to the routine until every task in the group has executed a (corresponding) call to the same routine. Therefore, when a routine is executed in barrier mode, there is a point in time at which all tasks in the group are simultaneously executing the routine. In contrast, when a routine is executed in non-barrier mode, each task blocks only until it has completed its role in the routine. As a result of this difference, tasks may be able to return more quickly from calls to non-barrier mode routines, resulting in better performance. However, this may lead to strange results, such as a message sent by a task after a collective communication call may be received by another task before it executes the matching collective communication call. Furthermore, because the role of each task is dependent on the particular way in which the routine is implemented, the behavior of non-barrier mode routines is implementation dependent. (See [1] for details.)

The barrier mode should be used when debugging code in order to verify correctness. Once the code has been shown to operate correctly in barrier mode, it is possible to change to non-barrier mode for performance reasons, provided that the barrier is not required for correctness. Finally, note that it is always possible to use MP_SYNC for explicit synchronization.

8. Concluding remarks

The design of a message passing library like EUI requires compromises between somewhat inconsistent goals: elegance of design, ease of implementation, and performance, especially in a tightly coupled environment of a dedicated partition. Below, we list a few of the issues we encountered.

Datatypes. The EUI library, like many others, treats messages as untyped streams of bytes: a send operation specifies the initial address and the length of the source buffer. The same send function is called to send integers, floating point numbers, or characters; the function is called with actual arguments of different types, thus violating the Fortran typing constraints. This constraint (all calls to the same function have actual arguments of the same type) is hardly, if ever, enforced by compilers. The alternative (a different send operation for each datatype) is very unappealing, for users and implementers. Since this problem can be solved in Fortran 90 by overloading and in C by using pointer casting, we decided to accept a nonstandard conforming implementation.

Asynchronous returns. When a non-blocking receive is invoked, some of the arguments may be modified by the system after the function has returned. This is error prone and violates the modularity expected from procedure calls. In fact, we ourselves wrote erroneous examples to illustrate non-blocking message passing in a first draft of EUI! Unfortunately, non-blocking message passing is essential for performance. One would hope that, in the future, the overlapping of computation and communication achieved by non-blocking message passing would be safely introduced by compilers or preprocessors, rather than done manually by a user.
System interfaces. In the current implementation of EUI, tasks are regular AIX processes that can use all system services. However, some system calls (e.g., fork) have limited use in a parallel environment like EUI; all system services introduce significant performance issues. The time to service a system call is significantly longer than the time required for interprocess communication. A system call delays not only the calling task, but also all other tasks in a partition. The same holds for page faults and other asynchronous system events. In the future, we expect to provide more control for paging and better ability for off-loading system calls to remote servers.

Debugging. Debugging and tuning of parallel codes is a nontrivial issue. Furthermore, the use of an interface like the EUI introduces the possibility for bugs that are difficult to trace, due to the timing of asynchronous events and the relative lack of protection. The parallel programming environment of the SP1 consists, in addition to the EUI, of several programming tools for debugging and for performance-tuning of EUI codes.

9. Acknowledgements

Many other people have contributed to the EUI in various ways. We thank Paul Bildzok, Wei-Hwan Chiang, Mark Giampapa, Magda Konstantinidou, Peter Levangia, and Dean Liberty for their assistance and helpful comments related to the EUI work. We also thank the many external reviewers of the EUI proposal including Jack Dongarra, Geoffrey Fox, Al Geist, Bill Gropp, Don Heller (and many others at Shell), Ewing Lusk, Bob Manchek, and Cherri Pancake. Finally, we would like to thank Zeev Barzilai, Joanne Martin, Dragutin Petkovic and many other people in IBM for their constant support.

10. References


