

# Implementation of MPI on the Fujitsu AP1000: Technical Details

## Release 1.1

David Sitsky  
ANU-Fujitsu CAP Project  
Dept of Computer Science, Australian National University  
Canberra, A.C.T. 0200, Australia  
email: `David.Sitsky@cs.anu.edu.au`

September 21, 1994

## 1 Introduction

This document describes the design decisions regarding the implementation of MPI [1] on the Fujitsu AP1000 multicomputer. These decisions include how to map MPI onto the AP1000 and the implementation of groups, communicators, point-to-point operations, derived datatypes, collective communication and topologies. The main goal of the implementation was to provide an efficient MPI library with little change to the `sys.c7` (CellOS v1.4) kernel. The performance of the implementation is provided with some future directions nominated. This document assumes familiarity with the AP1000 and some of the CellOS functions.

## 2 MPI on AP1000 Host and Cells

The MPI standard only specifies message passing operations. It doesn't specify how an application should be invoked nor does it constrain how MPI processes are mapped onto a real physical machine. This leads to a number of design alternatives for mapping MPI on the AP1000.

- Closed or open system. In a closed system, all MPI processes in `MPI_COMM_WORLD` execute on the AP1000 using the host and the cells. In an open system, the AP1000 is part of a "parallel virtual machine" in a similar manner to PVM, where other MPI processes in `MPI_COMM_WORLD` may be located on other workstations or multicomputers.
- All cell `MPI_COMM_WORLD`. Allow `MPI_COMM_WORLD` to consist of all of the invoked cells, where cell id  $n$  is rank  $n$ . This should work nicely for those applications which require a hostless type computation. After initialisation of the system, the host process simply waits for all of the cells to terminate.
- Allow `MPI_COMM_WORLD` to consists of both the host and cell processes. Here the host process is rank 0 in `MPI_COMM_WORLD` and cellid  $n$  is rank  $n + 1$ . This provides the flexibility for those applications which exhibit a master-slave type structure.
- Single or multiple MPI processes per cell. A single process per cell allows a simple implementation and provides a straightforward interpretation of `MPI_COMM_WORLD` for both the host-cell and hostless configurations as given in the previous points. Multiple processes per cell can be executed as separate CellOS tasks.

A closed system is currently used for the implementation, mainly because of simplicity. There is no problem in principle to extend the current system to allow `MPI_COMM_WORLD` to span other machines. This could be achieved by either using the socket package with the `sys.anu` kernel or by reserving the host process to handle communication to and from the outside world in a similar fashion to the `APPVMDRIVER` host process from the AP1000 PVM port [3]. This however will require substantially more complicated code, since unlike PVM, the MPI standard does not specify any protocols for communication to other machines.

The current system only allows one MPI process running on each cell again for simplicity. The problem with allowing multiple processes per cell is that tasks under CellOS v1.4 have run to completion semantics which isn't desirable. Work on the CellOS kernel has allowed a true multitasking system which could be effectively used to provide multiple MPI processes running on each cell.

Both interpretations of `MPI_COMM_WORLD` for host-cells and all cells are important, and for this reason both modes are currently supported in the system. The mode which is used is determined at runtime by a switch to the `mpirun` script, which executes MPI programs.

The CellOS `ccreat` call allows the same executable to be downloaded to all of the cells at startup time providing a SPMD environment. This can be extended to run multiple executables as separate tasks to run on all of the cells if multiple processes per cell were supported. It is also possible to download certain executables to be run on specified cells using the `l_ccreat` call.

### 3 Groups

Since a closed system is used and only a single process can be executed on each cell, a group can be represented very easily by just recording a list of cellids. This is also applicable to the host process since it has a cellid of `0x1000` (see `cap2.c.h`).

The structure used to represent a group is illustrated below:

```
typedef struct {
    int refcount;                /* Number refs to this group */
    int cellid[MPID_MAX_NUMBER_PROCESSORS]; /* Rank -> cellid mapping */
    int rank[MPID_MAX_NUMBER_PROCESSORS]; /* Cellid -> rank mapping */
    int hostrank;                /* Rank of host, -1 if not in group */
    int size;                    /* Number members in group */
} MPID_Group;
```

The `refcount` field is used to avoid excessive copying of groups by keeping a count of the number of references to this group. When the `refcount` is 0, the group can be safely removed. The `MPID_MAX_NUMBER_PROCESSORS` value represents the maximum number of processes that can run on the machine. The `cellid` array provides a direct mapping of rank to cellid. The cellid of the rank 5 process in a group is obtained by evaluating `cellid[5]`. The `rank` array provides an inverse relation, where applying the array to a cellid will return its associated rank. This is required when doing a wildcard receive to quickly determine the rank of the sender.

This mechanism only works with the cells, since the host has a cellid of `0x1000`. For this reason, the `hostrank` field records the rank of the host process in the group, which has the value of -1 if the host is not present in the group. Finally the `size` field records the number of elements in the group.

Should the system be extended to support multiple processes running on each cell, the group structure can be suitably amended so that the mapping arrays reflect the identification of an MPI process by a taskid, cellid pair rather than just the cellid.

This structure would have to be extensively modified to support an open system, with a more general scheme to reference off-CAP MPI processes.

## 4 Communicators

Communicators basically represent a group of processes (two disjoint groups for intercommunicators) which operate under two unique contexts, one for point-to-point operations and the other for collective operations. There are several options for implementing contexts, they are listed below:

- Split the CellOS type field so that it holds both the MPI tag field and a context field. This causes problems since the maximum CellOS type value allowed is specified by `UMTYPE_MAX-1` which has the value of 65535 - a 16 bit value. The MPI standard specifies that the upper bound value for tags (`MPI_TAG_UB`) must be at least 32767 which is a 15 bit value. More than 1 bit is required to represent a context value so this approach is prohibited unless the standard is to be violated.
- Repack all messages so that the context value is contained in the first word of the message. This is fairly inefficient on the AP1000 since it has no vector send and receive operations, meaning messages must be copied into another buffer before being sent. Implementing the message receiving primitives would be quite complicated since the message queue would have to be traversed to find the appropriate messages that belonged to a specific context. Wildcard receiving within a given context is a good example of this.
- Send extra messages. Whenever a message is sent in context *c*, two messages are sent. The first identifies the context of the following message from the same sender, the second contains the actual message. This also greatly complicates the basic send and receive primitives and requires an extra message for every send or broadcast operation.
- Modify the CellOS system message headers to directly support contexts. This is the most efficient option and is the approach that has been taken. The domains package [4] provides this functionality and has been used in the implementation. This package also provides a selective broadcast operation which will be further discussed in Section 7 (Collective Communication).

The structure representing a communicator is given below:

```
typedef struct _MPID_Comm {
    MPID_Comm_type comm_type;      /* Type of communicator */
    int refcount;                  /* Number refs to this comm */
    int pt2pt_domain;              /* Domain for pt2pt ops */
    int collective_domain;         /* Domain for collective ops */
    MPI_Group remote_group;        /* Remote group */
    MPI_Group local_group;         /* Local group */
    int owner_rank;                /* Rank of owner */
    MPI_Errhandler error_handler; /* Error handling routine */
    SPLtree attribute_cache;        /* Attribute cache */
    MPID_TOPOLOGY topology;        /* Topology information */
} MPID_Comm;
```

The `comm_type` field is an enumerated type which specifies whether the communicator represents an intracommunicator or intercommunicator. The `refcount` field serves the same purpose as it does for the group structure. The `pt2pt_domain` and `collective_domain` fields represent the unique domains used for point-to-point and collective operations within this communicator. The `remote_group` and `local_group` fields represent the remote and local groups of the communicator (they are the same for an intracommunicator). The `owner_rank` field records what the process' rank is in this communicator. Other than this field, all of the other values are identical across all of the communicator members. The remaining fields record what the error handler, attribute cache and topology values are.

The domains package allows the use of any integer valued domain number ( $0 \dots 2^{32} - 1$ ), however a process can only be receiving from 300 domains at any given time. When MPI is initialised, each process creates a “domain stack”, which is a stack that contains 100 unique domain numbers across the entire machine. This stack is used when new domain numbers are determined for new communicators. For most of the communicator constructor functions, the rank 0 process in the new communicator determines the two new domain numbers to use by popping off two numbers from its domain stack. When a communicator is freed, the rank 0 process pushes these two domain numbers back onto its stack. During the execution of a communicator constructor function, these new domain numbers (along with other relevant information) are broadcasted to the other members of the new communicator so that they know which domains the new communicator will work under. The domain stack size of 100 per process should handle all but the most pathological of MPI programs.

When a process requires to receive messages in a particular domain, the `register_domain` system call must be invoked with this domain. Because of these semantics, it is necessary for all of the processes that are members of a new communicator to synchronise to insure they are all ready to receive messages in the new point-to-point and collective domains. This requirement is handled automatically within the implementation of all of the communicator constructor functions. Similarly the `deregister_domain` call is used to indicate that messages from the supplied domain are *not* to be received anymore. This is called when `MPI_Comm_free` is invoked.

## 5 Point-to-Point Operations

CellOS provides the routines `l_asend` and `l_arecv` as the basic send and receive operations. These routines work in conjunction with system buffers, meaning messages sent are first copied into a system buffer and then sent to the network using DMA. Similarly, messages arriving at a cell are placed into a system buffer and then copied into the user's buffer when requested. When linesending mode is enabled at runtime, the sending operation is changed by blocking the sender until the entire message from the user's buffer (or cache) has been directly sent onto the network, which avoids the extra memory copying operation to system buffers at the cost of blocking the user.

From this perspective, the sending mode without linesending appears to be nonblocking since the user's program may continue executing while the system does a DMA transfer from the system buffer onto the network, while a linesending send is blocking since the user's program is blocked until the message has been safely sent. However in both cases, after `l_asend` has returned, the user may safely reuse the buffer containing the message.

In MPI, nonblocking operations have the semantics that the user's buffer cannot be accessed until the nonblocking operation has completed. Functions are provided to indicate completion of nonblocking operations. A nonblocking send operation in MPI is envisaged to work like the linesending send operation, however the user's program should *not* be blocked while the message

is being sent onto the network. For this reason, both of the CellOS sending modes are classified under MPI as blocking send operations.

CellOS provides a form of active messages in the `put` and `get` primitives. This allows a cell to directly put or get strided data from another cell’s memory without any “handshaking” from the process running on that cell. These routines are nonblocking and there are mechanisms to determine when a `put` or `get` operation has completed.

The domains package provides the routines `dl_asend` and `dl_arecv` which work in a similar fashion to `l_asend` and `l_arecv`, except they work with an extra domain parameter. For this reason, the blocking routines `MPI_Send` and `MPI_Recv` are very efficient since they basically just call the associated domain communication routine.

Since CellOS v1.4 doesn’t provide any nonblocking send and receive operations which transfer directly from/to the user’s buffer, the `MPI_send` and `MPI_recv` operations are implemented using `MPI_Send`, `MPI_Recv` and `MPI_Probe`. The extensions present in the `sys.anu` kernel which provide striding-send and pre-receive could be used since they work directly from/to the user’s buffer without the extra copying to system buffers.

The `put` primitive can be used for the implementation of the synchronised sending mode (`MPI_Ssend`). In this mode, the sender must wait for the receiver to post a matching receive to this message. When the receiver receives a synchronised send request, a message is sent back to the sender indicating the address of the receiver’s buffer. The sender can then transfer the message using the `put` primitive thus avoiding an extra layer of copying both at the sender and the receiver side. Since the `put` call is nonblocking, the `MPI_Ssend` routine can also be readily implemented in this way. Note this technique could only be used efficiently for contiguous or strided buffers on the receiver side due to the functionality of the `put` call.

## 6 Derived Datatypes

The use of derived datatypes allows the specification of noncontiguous buffers. If special message hardware isn’t available, a noncontiguous message must be packed into a contiguous buffer before being sent. Similarly, receiving a message into a noncontiguous buffer will involve unpacking operations. This is because the standard requires that the “strides” in the user’s receiver buffer cannot be overwritten. This process can be seen in Figure 1 which illustrates a process packing a strided buffer which is sent to another process which unpacks the message into the receiver’s strided buffer. Again the use of the `sys.anu` striding-send and pre-receive functions could be used effectively here.

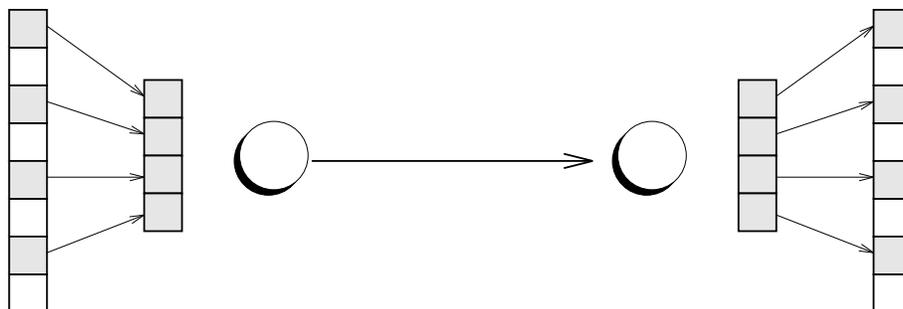


Figure 1: Sending and Receiving Strided Datatypes

The current system doesn’t use any of the `sys.anu` striding calls, however the system has been engineered to contain an extra layer of message passing operations which take an MPI

datatype as one of the parameters. All message passing operations within the library use these primitives. Currently, noncontiguous buffers are processed using packing and unpacking, however future releases which use hardware striding sends and receives can be implemented within these primitives, which the entire library will immediately benefit from.

Since all of the code for specifying and creating MPI datatypes is largely system independent, the code from the Argonne/Mississippi MPICH [2] distribution was used.

## 7 Collective Communication

The domains package provides a very important facility when implementing some of the collective operations in MPI. The selective broadcast operation (`dcbroadcast`) works like the `cbroadcast` operation, except the message is only delivered to those cells which have registered in the sent domain (ie the members of a communicator) using the `register_domain` system call.

This means a routine such as `MPI_Scatter` which usually requires some decomposition strategy for scattering an array amongst the group members of a communicator, can be implemented by simply selectively broadcasting the *entire* array to all of the group members, with each member extracting their appropriate piece from the array located in the system buffers. This results in a fast implementation which provides execution times *independent* of group size. The main disadvantage with this scheme is that memory for the scattered array must be allocated in the system buffers at each of the receivers. This process can be seen in Figure 2, where a six element array rooted at rank 3 is scattered to a six element communicator (the grey circles). Each member of this communicator copies the relevant piece from the array in a user's buffer. The operating system automatically discards the broadcasted message for all of the other processes in `MPI_COMM_WORLD`, since these processes are not a member of the six element communicator, and consequently haven't registered in any of its domains.

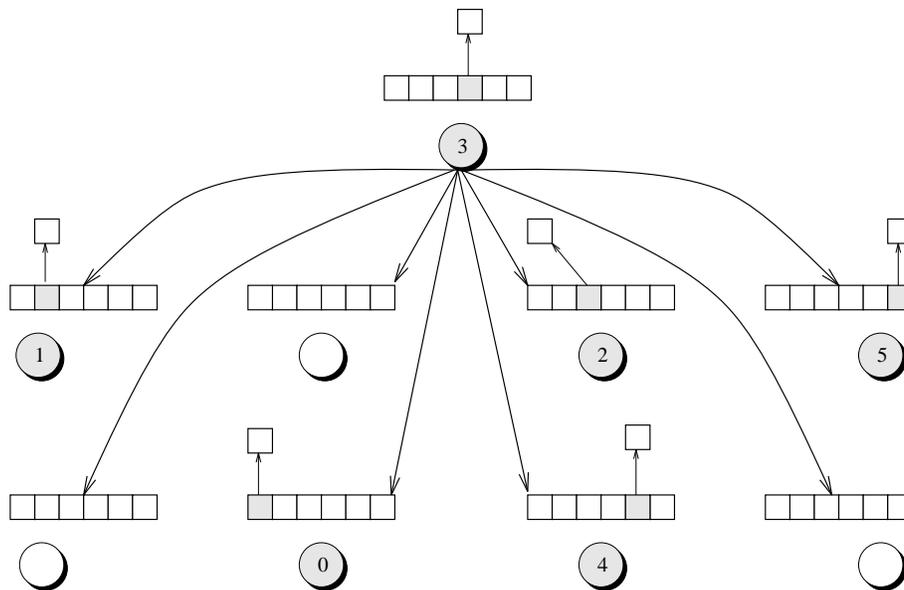


Figure 2: Implementation of `MPI_Scatter`

The collective routines `MPI_Barrier`, `MPI_Scatter`, `MPI_Scatterv`, `MPI_Allgather`, `MPI_Alltoall`, `MPI_Allgatherv`, `MPI_Alltoallv`, `MPI_Allreduce` and `MPI_Bcast` use the selective broadcast operation.

For small groups (ie less than 10), the use of the broadcast network may be inefficient due to its relatively large startup cost compared to point-to-point operations. For this reason, many of the collective operations are probably better implemented to work with point-to-point operations for small group sizes. This hasn't been implemented into the system, since it is expected that most applications won't involve collective communication over small groups.

There is also the issue of how "disruptive" the selective broadcast operation is when many communicators exist concurrently which are performing collective operations using the selective broadcast operation. This is addressed in Section 9 (Performance).

Some of the efficient collective CellOS routines are used when possible. For example in `MPI_Barrier`, the `sync` operation is used when the communicator consists of all of the cells and `hsync` when the communicator consists of the host plus all of the cells. In `MPI_Reduce` and `MPI_Allreduce`, the `xy` reduction routines are used when the communicator consists of all the cells and the user's input buffer is an integer, floating point or double precision array. The use of the CellOS scatter/gather routines `h_sscatter` and `h_rgather` haven't been exploited for the `MPI_Scatter` and `MPI_Gather` routines, but there is no reason why they can't be used. In fact since these routines can work with strided data, there is scope for applying these routines to strided MPI datatype buffers.

## 8 Topology

Due to the wormhole routing on the AP1000, distance is not a factor when considering latency between any two cells. For this reason, migrating processes under advice of the MPI topology functions is not likely to improve performance of a program significantly. Also network contention doesn't appear to be a significant performance detriment due to the relatively large software overheads of wormhole routing. Since no process migration is used, the topology code from the MPICH distribution was used with little change.

## 9 Performance

This section illustrates the performance of the MPI implementation. This includes a comparison of message passing and broadcast performance between the MPI implementation and native CellOS calls. This is followed by benchmarks of the `MPI_Bcast`, `MPI_Scatter`, `MPI_Barrier` and `MPI_Allreduce` collective functions. Finally a benchmark illustrating the disruptiveness of the selective broadcast operation is presented.

To benchmark a collective operation, each process measures how long it was involved and then the maximum, minimum and average time for all of the processes are calculated. This is necessary since not all of the processes execute the same code in the collective routines. To achieve sensible times, firstly all of the processes are synchronised (using the hardware S-net sync operation), a timer is started, the collective routine is invoked and then the timer is stopped. This is performed 10 to 100 times to obtain an average time. This time is then reported as the time taken to execute the collective routine for that process. The variance of these times were insignificant and are not shown on the graphs. All benchmarks were performed in the hostless `MPI_COMM_WORLD` mode.

The point-to-point benchmark is a basic ping-pong test, where cell A sends a double precision array to cell B, which copies it into a user buffer and sends an acknowledgement (zero size) message back. Cell A measures the time it takes to send the message and receive the acknowledgement. These times are illustrated in Figure 3 for different sized arrays (double is 8 bytes long). A rough estimate of the latency can be calculated by halving the intersection from the

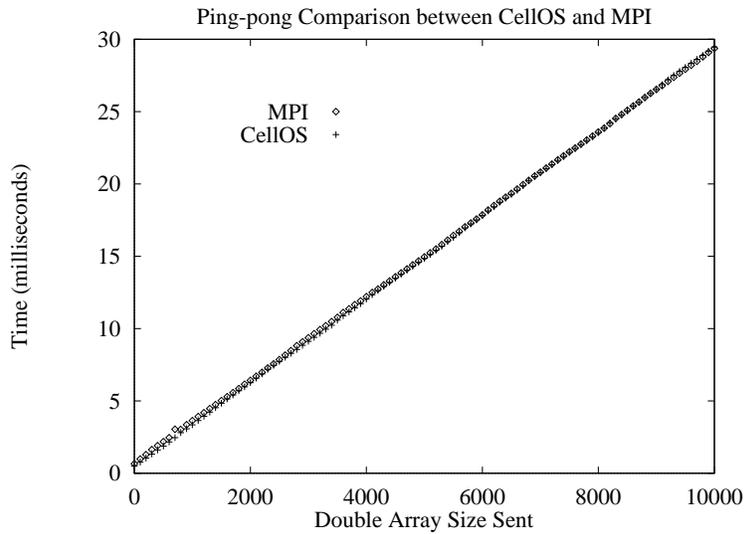


Figure 3: Point-to-Point Performance

graph, which shows Cellos to have 246  $\mu\text{sec}$  while MPI has 318  $\mu\text{sec}$ , which is 1.3 times longer. The transfer rates were measured to be 2.76 MBytes/sec for Cellos and 2.79 MBytes/sec for MPI, which is 1.01 time more. The marginally faster transfer rates for MPI could possibly be attributed to better memory alignment introduced by the domains package into the message header, resulting in faster memory copying for messages.

The aim of the first broadcast benchmark was to find the time difference for sending a double array to all cells on the machine when using Cellos and MPI. For clarity, only the maximum times are shown in Figure 4 which is approximately the same as the time for all of the receivers.

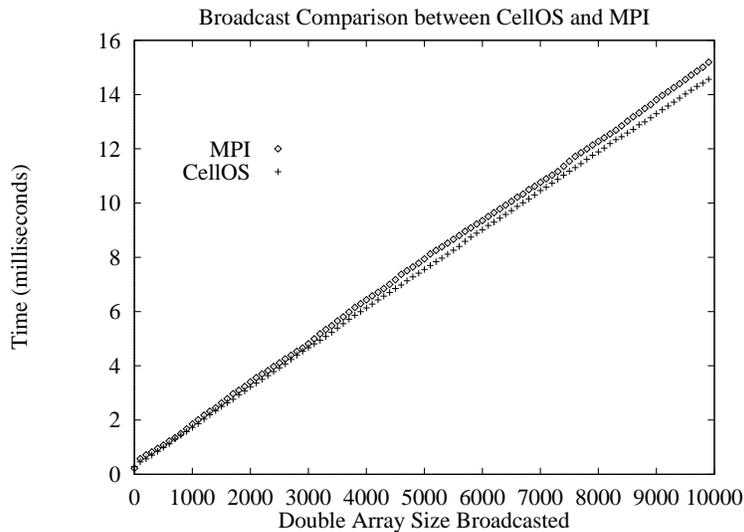


Figure 4: Broadcast Performance

The transfer rates were measured to be 5.49 MBytes/sec for MPI and 5.61 MBytes/sec for Cellos, which is 1.02 times greater. The maximum time difference between the two curves was measured as being 336  $\mu\text{sec}$  for 9990 doubles while the minimum was 53  $\mu\text{sec}$  for 100 doubles.

The next benchmark was to test how effective the MPI broadcast operation was for different

communicator group sizes when using the selective broadcast operation. The results can be seen in Figure 5. The minimum time shown on the graph is always the time taken by the root process

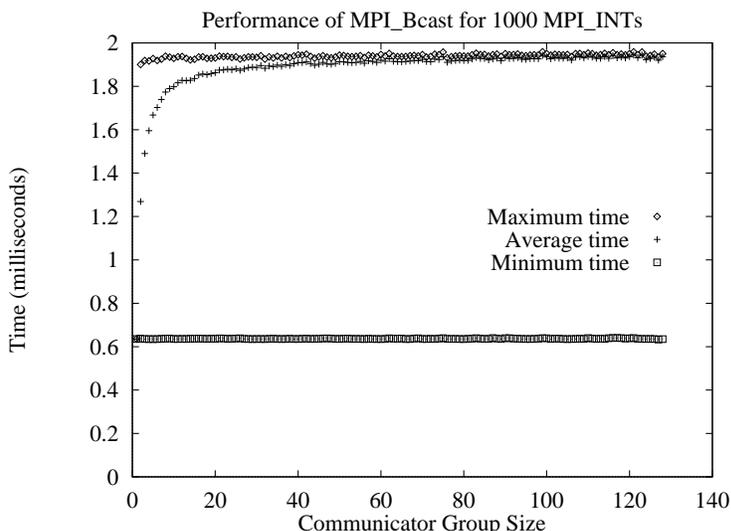


Figure 5: MPI\_Bcast Performance

or the process which sends the broadcasted message. The other processes have to wait for the message to arrive and copy it into the user’s buffer. The other process times are all close to the maximum time illustrated on the graph which explains the curve of the average time. The most important feature of the graph is that the operation is scalable: broadcast operations take the same time for a process regardless of the group size. This is precisely what we expected with the selective broadcast operation, which makes it an effective function for implementing some of the other collective routines.

The next benchmark illustrates the times for evenly scattering a 1280 integer array (integer is 4 bytes long) among different sized communicators. Given the behaviour of the selective broadcast operator previously shown, we expect this operator to also be scalable. This is indeed the case as shown in Figure 6, although the smaller sized communicators take slightly longer. This is because the processes in the smaller communicators receive larger pieces of the scattered array, meaning they have to perform more memory copying into the user’s buffer.

The next benchmark illustrates how the scatter operation scales for larger integer arrays. In this case, a single communicator was used which consisted of 128 cells. As it can be seen in Figure 7, the operation scales linearly, which can be attributed to the scalable aspects of the broadcast network. It should be noted that the “bends” in the graph occur at multiple bytes of the cell’s cache size (128KB).

The MPI\_Barrier collective routine is implemented by performing a zero data gather rooted to the rank zero process using logical trees followed by a selective broadcast. The tree has a branching factor of three, which was found to deliver the best performance. Since selective broadcast is a constant time operation for the same data, we expect the graph to exhibit a logarithmic curve since the number of steps in the gather operation is the height of the logical tree. This is confirmed in Figure 8. The small variance in times for all of the processes is due to the final selective broadcast operation which all processes except the sender is waiting for. Note the very fast time of 0.29 msec when the communicator size is 128, when the synchronisation hardware is used directly via the Cellos sync system call.

The next benchmark illustrates the time taken to call MPI\_Allreduce to sum an MPI\_INT.

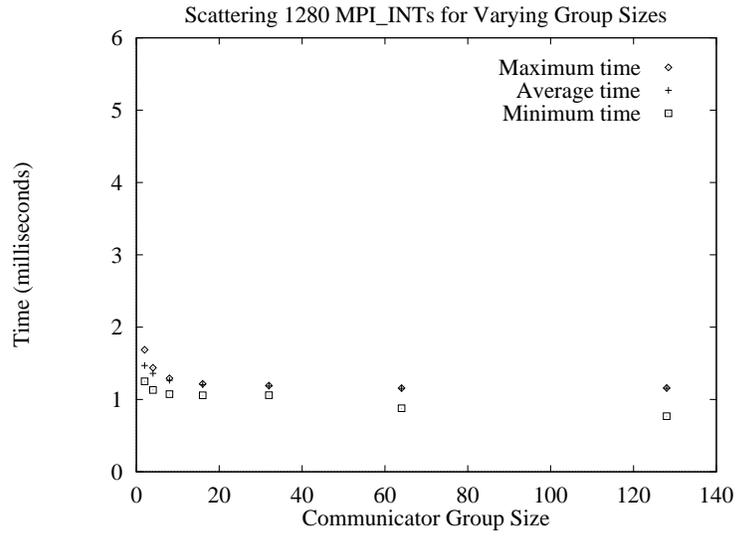


Figure 6: Performance of MPI\_Scatter for varying Group Sizes

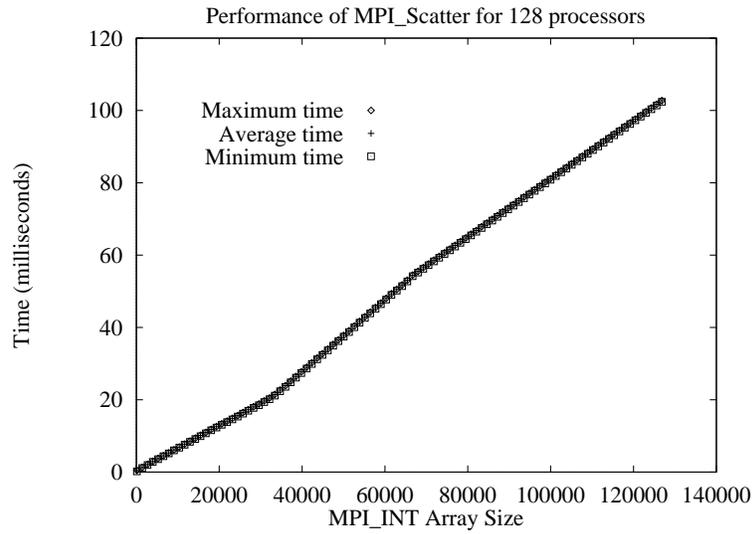


Figure 7: Performance of MPI\_Scatter for varying Array Sizes

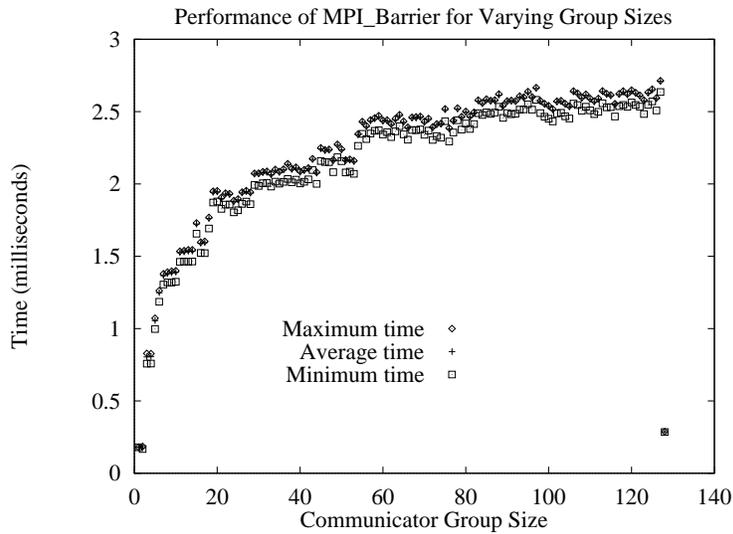


Figure 8: Performance of MPI\_Barrier

This routine is implemented by using logical reduction trees, and uses the CellOS `xy` routines whenever possible (when the communicator consists of all of the cells). The results can be seen in Figure 9. Each “step” in the graph represents the case when the logical reduction tree has

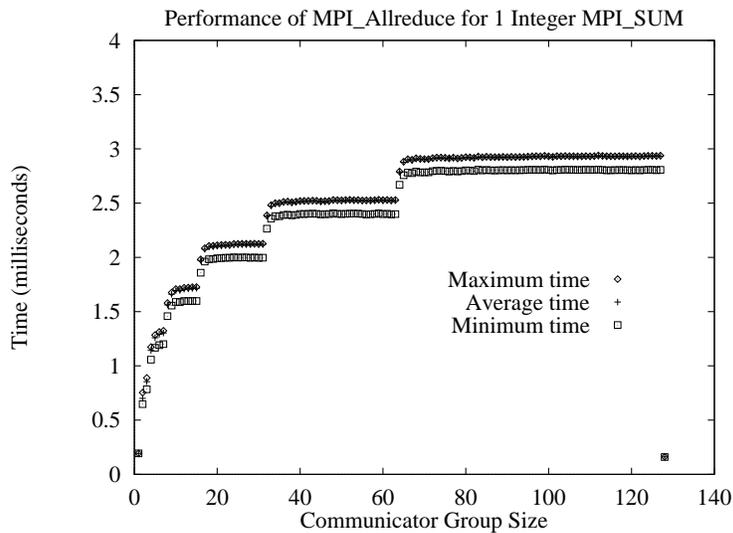


Figure 9: Performance of MPI\_Allreduce for varying Group Sizes

grown an extra level. Note the very fast time of 0.16 msec for a communicator of size 128 when the `xy_isum` call was used.

The next benchmark examines how the `MPI_Allreduce` function scales for larger input vectors when the communicator consisted of all the cells. The results can be seen in Figure 10. Since MPI reductions work on vectors and the CellOS `xy` calls work only on single numbers, it would be naive to continuously call an `xy` routine for each vector element from the user’s buffer for large vectors. It was determined that the best performance was obtained when using the `xy` routines for vectors of size 25 or less, and then to use reduction trees for larger sizes. There is little difference between the maximum and minimum times since `MPI_Allreduce` is implemented by

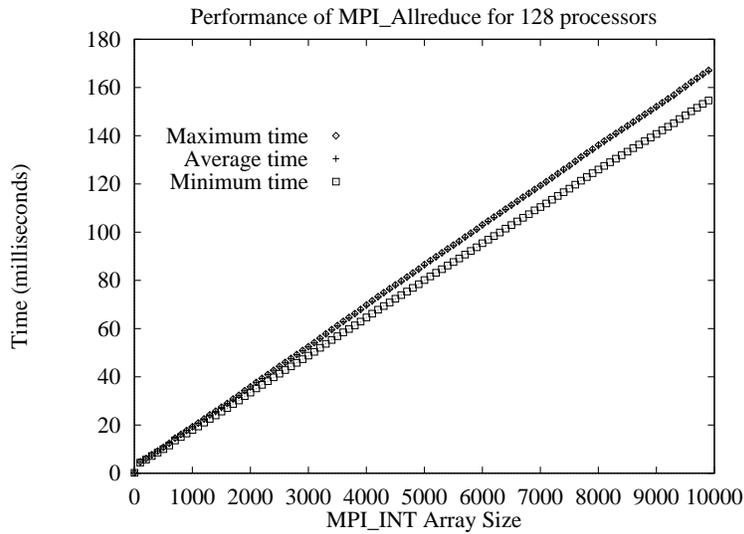


Figure 10: Performance of MPI\_Allreduce

calling `MPI_Reduce` and then invoking the selective broadcast operation to inform all members of the communicator of the result.

The final benchmark illustrates what happens when many groups use the selective broadcast operation at once. A number of two element communicators are created, all of the processes are then synchronised, a timer is started, each communicator performs an `MPI_Bcast` operation for 10,000 `MPI_INT`s and then the timer is stopped. Only the receiver times are measured since the `dcbroad` operation returns almost immediately for the sender. As can be seen in Figure 11, there is a slow average degradation in performance, with the maximum times doubling when

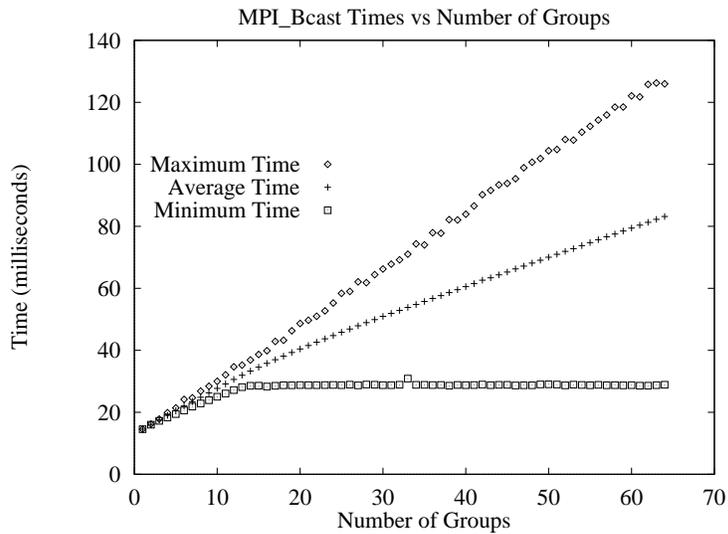


Figure 11: Interference of Selective Broadcast

the group size is doubled. As expected, after an initial number of groups, the minimum time stays constant, since this represents the receiver of the first communicator which acquired the broadcast network. Since it is not expected for an MPI application to require a large number of groups running concurrently and using the broadcast network at the same time, the utilisation

of the selective broadcast operation appears satisfactory.

All of the collective routines presented exhibit good performance. However it has been illustrated that the use of the very efficient CellOS routines which can only be applied to communicators consisting of all of the cells result in a significant performance gap between communicators consisting of a subgroup of the cells. It may be possible to rewrite these routines and modify the kernel to support these efficient operations for all groups in the same manner as the selective broadcast operation, but this is likely to be a nontrivial exercise.

## 10 Future Work

Future work to the implementation is to investigate the use of multiple processes per cell, to implement the `MPI_Ssend` calls using the `put` primitive, to use the `h_sscatter` and `h_rgather` routines to implement `MPI_Scatter` and `MPI_Gather` under special circumstances and to investigate the use of the `sys.anu` striding send and receive operation to implement the MPI nonblocking operations. Other work may include investigating the possibility of changing the system to be “open” and to allow the AP1000 to be part of a larger `MPI_COMM_WORLD`.

## References

- [1] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994. To appear in the International Journal of Supercomputing Applications, Volume 8, Number 3/4, 1994.
- [2] William Gropp and Ewing Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Technical Report MCS-P342-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1994. in preparation.
- [3] C. W. Johnson and D. Walsh. Porting the PVM distributed computing environment to the Fujitsu AP1000. In *Second Fujitsu Parallel Computing Workshop*, November 1993.
- [4] David Walsh, Peter Bailey, and Bradley M. Broom. Message domains: Efficient support for layered message passing software. Technical Report TR-CS-92-16, Department of Computer Science, Australian National University, 1992.