

CRI/EPCC MPI for CRAY T3D

Kenneth Cameron, Lyndon J. Clarke, A. Gordon Smith¹

Edinburgh Parallel Computing Centre
The University of Edinburgh
James Clerk Maxwell Building
The King's Buildings
Mayfield Road
Edinburgh EH9 3JZ

Abstract

MPI is the standard interface for message passing in parallel systems. The standard was defined in an open collaborative forum involving about 60 people from 40 different organisations, and the finished version was completed in May 1994. EPCC and CRI have established a collaboration to develop and support a robust, high performance implementation of MPI for the T3D. The product was released from EPCC in May 1995. In this paper we describe the architecture of MPI for the T3D and present performance measurements of the implementation. It is shown that a significant part of the T3D capability is available through MPI, demonstrating that MPI is capable of delivering both performance and portability to applications.

1 Introduction

MPI [5] is the standard interface for message passing in parallel systems. The standard was defined in an open collaborative forum involving about 60 people from 40 different organisations. An initial draft of the standard was released at the *Supercomputing '93* Conference and the final version was completed in May 1994 [4].

Edinburgh Parallel Computing Centre (EPCC) and Cray Research, Incorporated (CRI) have established a collaboration to produce and support a robust, high performance implementation of MPI for the Cray T3D [1] multicomputer. The implementation took place during 1994 and the product was released from EPCC in May 1995. The software is distributed via the World Wide Web and further information can found at the URL:

<http://www.epcc.ed.ac.uk/t3dmpi/Product/>

In this paper we describe the architecture of the MPI implementation for the T3D. We

¹Email: smith@epcc.ed.ac.uk

also present performance characteristics of MPI on the T3D and relate these to the software architecture. In Section 2 we present a brief overview of aspects of the T3D system relevant to the MPI implementation. This is followed by a description of the architecture of the MPI implementation in Section 3. Performance characteristics of MPI on the T3D are presented and discussed in Section 4.

2 Overview of T3D architecture

The T3D is a distributed memory multicomputer based around a high performance three dimensional interconnect network. The six channels out of each interconnect node are able to simultaneously support hardware transfer rates of 300 MB/s. The cost of routing data between processors through interconnect nodes is essentially negligible. The penalty is two clock cycles per node traversed and one extra clock cycle to turn a corner.

Each interconnect node hosts two independent processing elements, consisting of a DEC Alpha 21064 processor and 64 MB of RAM. The processor is clocked at 150 MHz, supports 64 bit integer and IEEE floating point operations, and delivers a peak performance of 150 64 bit Mflop/s. The processor has an 8 KB direct mapped data cache and an 8 KB instruction cache (there is no second level cache).

The interconnect interface hardware of the T3D supports a single address space across the machine. This global address space means that in principle each processor has direct access to the memory of every other processor, allowing the total memory to be considered as a single object. In practice the machine appears as a shared-nothing architecture with a remote memory read/write capability provided as a software interface through the Shared Memory Access (SMA) library [2].

Remote memory read is performed using the function `shmem_get(tgt, src, n, pe)` which copies `n` words from source address `src` in the memory of the PE identified by `pe` to the target address `tgt` in the memory of the calling PE. Remote write is performed using the function `shmem_put(tgt, src, n, pe)` which copies `n` words from source address `src` in the memory of the calling PE to the target address `tgt` in the memory of the PE identified by `pe`. Note that the `shmem` calls are not consistent with the data cache, for example `shmem_put` does not change the value of cache data in the remote processor. The performance of `shmem_put` and `shmem_get` is an important aspect of the T3D architecture. The latency is 1-2 μ s. The bandwidth is 120 MB/s for `shmem_put` and 60 MB/s for `shmem_get`. The reason for this difference is that `shmem_get` consists of a request followed by a remote write by the remote PE to the local PE, whereas `shmem_put` simply consists of a remote write by the local PE to the remote PE.

The off-processor components of the PE also provide other facilities to support remote memory access programming. One of the most important of these for the MPI implementation is the atomic swap capability. The SMA library also provides a function `shmem_swap(tgt, value, pe)` which atomically reads the content of the word at target address `tgt` in the memory of the PE identified by `pe`, writes `value` into that address, and returns the read value. The other important capability is hardware support for barrier synchronisation of all processors in the user partition.

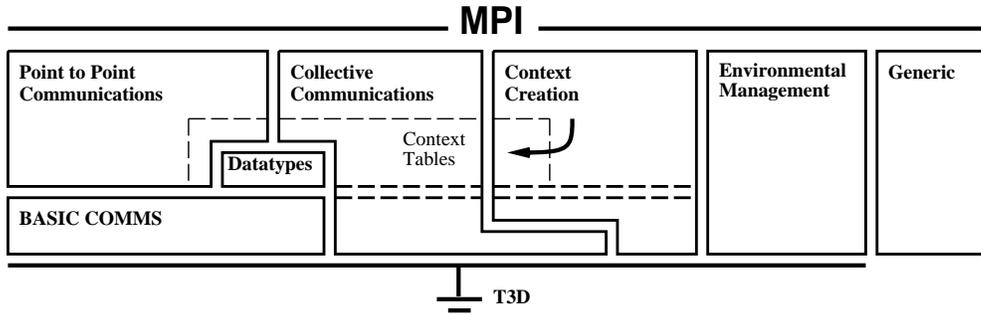


Figure 1: MPI for T3D module structure.

3 Architecture of MPI Implementation

In this section we present a brief description of the architecture of the implementation of MPI for the T3D multicomputer. We focus on the point-to-point and collective communications which were the major part of the implementation effort.

3.1 Overview

To provide competitive message-passing performance between CRAY T3D processes, depth of modularity is minimised. The flat-modular structure is illustrated in Figure 1.

The **BASIC COMMS** module provides internal services to transfer fixed-size and arbitrary-size blocks of data between PEs to modules that wish to perform inter-process communication. Transfer and mutual exclusion is implemented using shared memory operations. Since messages transferred by MPI communications can have arbitrary structure, the **Datatypes** module provides internal services for description, transformation and transfer of datatyped messages.

The **Generic** module comprises all external MPI services that have generic implementation; much of this code was based on a portable EPCC MPI implementation atop CHIMP [3]. The remaining modules group external MPI services by chapter. **Point-to-point** communications are implemented using the protocol and data transfer services of **BASIC COMMS**. For efficiency, **Collective** and **Context creation** communications use a dedicated mechanism based on shared memory transfer. **Environmental Management** makes use of system timers, UNICOS MAX signals and inquiry routines.

3.2 Point-to-point

The Point-to-point Communications chapter of the MPI standard provides an interface to pure message-passing functionality: transfer of data between a pair of processes as determined by the matching of a send and a receive operation. Auxiliary services are specified to declare the elemental types and layout of the data in a message; to query communication status; and to determine the effect of communications progress on program control.

Speculative transfer of communications parameters is necessary to manage the matching of send and receive operations. MPI send and receive operations specify data, a process, an integer tag value, and a communications context. A send and receive match when the processes correspond, and tag values and contexts are equal; this information is the minimum that must be transferred within a fixed-length control message, commonly known as an *envelope*. Following a match the data given by the send operation can be transferred from the source process into the data area given by the receive operation at the destination process.

The **BASIC COMMS** module provides the implementation of speculative transfer of envelopes, based on a shared-access queue for each process. Messages, and queue slots, correspond to data cache lines to facilitate cache coherence maintenance. Mutual exclusion on a process' queue is enforced using *atomic swap* functionality. A writer process obtains exclusive access to the remote queue when it acquires a valid queue index as the result of a swap; it can then write into the indexed slot using `shmem_put`, update the head index, and relinquish access using a second swap. Transfer of user message data can be driven by sender or receiver, via `shmem_put` and `shmem_get` respectively.

With MPI, as with most message-passing systems, it is possible that there may be a set of unmatched, or pending, communications. Relevant information from each communication event must be recorded and maintained in some data structure that encodes communications state. The semantics of MPI Point-to-point Communications require that the order in which local operations are made, and the order in which envelopes are received from a process, is maintained. This suggests that events be recorded within a system of queues.

Events are local if they relate to a user operation, or alien if they relate to the reception of an envelope that indicates an operation on another process. Local events are matched against alien events and vice-versa; similarly send events only match receive events these categories can be maintained separately. An event relating to a receive call can have *wildcard* values for source process or tag value, indicating that these properties are not relevant to a match with another event. In contrast, the communications contexts must be identical for two events to match. This means that events can be separated by context: communications state is represented by a set of event queues, characterised by $\{send, recv\} \times \{local, alien\} \times Context$. This division reduces the search space for a match.

The matching operation involves a traversal of the appropriate queue, comparing process and tag value with those of the candidate event; the first matching event encountered is the correct match. There are MPI operations, such as `MPI_Probe`, that query the communications state. Most, however, change the state and require that matching events are removed from their queue: the MPI communication relating to the match has been committed, though may not yet be complete. The event queues can be singly-linked as removal always immediately follows traversal, and can use a trailing pointer. If the match operation fails, the event is added to the opposite queue for the given communications context: the communications state is extended.

Since an MPI receive operation need not specify the source process, the send operation takes responsibility for communicating the send parameters to the receiver for matching. In this way the sender can invoke a number of end-to-end protocols according to message size and synchronisation requirements. These protocols are illustrated in Figure 2:

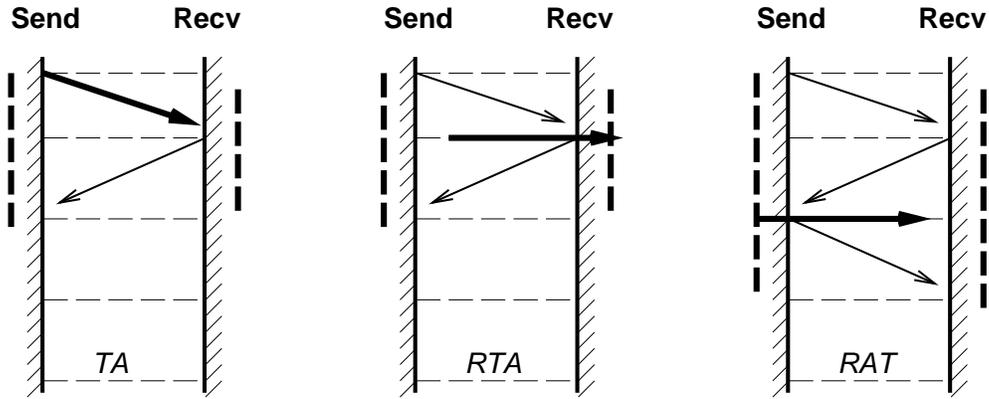


Figure 2: Point-to-point Data Transfer Protocols

Transfer (T) : When the message can be accommodated within the send envelope it is transferred immediately and is available to the receiver following a successful match.

Transfer-Acknowledge (TA) : This protocol extends the **T** protocol with an acknowledgement message, returned to the sender after a successful match. This provides synchronisation between sender and receiver.

Request-Transfer-Acknowledge (RTA) : This protocol has the receiver drive message data transfer, following a successful match. An acknowledgement message indicates completion of the get-based transfer to the sender.

Request-Acknowledge-Transfer (RAT) : This protocol has the sender drive message data transfer. Following a match, the receiver returns target address information to the sender within an acknowledgement message. The sender reacts to the acknowledgement by using put-based transfer. It is necessary for the sender to indicate completion of the transfer to the receiver using a reverse acknowledgement message.

The Point-to-point communications semantics define MPI behaviour with regard to system resource limitations. Accordingly, MPI provides a number of *send modes* each with different semantics in this area, manifested in the synchronisation of send operations with matching receives.

Standard mode relates to use of `MPI_Send` (and its Immediate and Persistent variants). The MPI standard does not commit as to whether this send mode allows a send operation to complete before a corresponding receive operation or not; behaviour is dependent on the MPI implementation. A popular interpretation is that the send completion be independent of the destination process as long as there is sufficient system buffering space (memory owned by MPI or by a communications subsystem); when this space is exhausted, send completion can be delayed for a matching receive. If the message can be accommodated within a protocol message, the **T** protocol is used; the system memory used is within the protocol message slot at the receiver. Otherwise, the message should be copied into memory owned by the MPI implementation so that the send can complete, allowing the user to modify the message without corrupting the transfer. The **RTA** protocol is invoked by the send operation, where the message copy is advertised

instead of the message specified by the send operation. Since the send has returned it is the receiver's responsibility to fetch the message copy using get-based transfer. The acknowledgement indicates that the sender process can re-use the buffer holding message copy. If there is no space available for holding a copy of the message, the sender can advertise the actual message, but must block until the acknowledgement is received.

Synchronous mode relates to use of `MPI_Ssend`. The semantics with respect to synchronisation with the receiver are clear: this send operation completes when a matching receive operation is started. Send and receive operations are active at the same time allowing in-place data transfer. If the message can be accommodated within a protocol message, the **TA** protocol is employed. Otherwise, the **RAT** protocol arranges put-based in-place data transfer, driven by the sender; cache coherency is maintained explicitly by the receiver when a protocol message indicating the end of transfer is obtained.

Buffered mode relates to use of `MPI_Bsend`. This send mode guarantees send completion that is independent of the corresponding receive operation. This uses exactly the same protocols and buffering discipline as Standard mode as long as there is buffer space for copying messages. If this space is exhausted this send mode will not block for an acknowledgement, instead generating an error. Another difference between this mode and Standard mode is that it may use buffer space provided by the user via `MPI_Buffer_attach` for holding message copies, in addition to space allocated by MPI.

Ready mode relates to use of `MPI_Rsend`. The behaviour of this mode is undefined if a send operation precedes a matching receive. There is no way to optimise this send mode on Cray T3D so it is identical to Standard mode.

Derived datatype construction and inquiry are generic operations in that they build and traverse tree data structures. The only part of these operations that is specific to the implementation of Point-to-point communications for Cray T3D, is classification of the layout of a datatype:

Contiguous : a message composed of this class of datatype can be described by the pair $(address, size)$.

Strided : a message composed of this class of datatype can be described by the 4-tuple $(address, numFragments, fragmentSize, strideSize)$.

Irregular : a message composed of this class of datatype cannot be described as a pair or 4-tuple.

Two forms of datatyped message transfer operations are used by all MPI communications corresponding to the **Contiguous** and **Strided** datatype classifications: they traverse a derived datatype tree transferring each contiguous fragment into a contiguous or strided target buffer, respectively. The transfer operation can be remote put or get, or a local copy. If the target buffer for a communication is classified as **Irregular** a contiguous system buffer must be provided as the new target. This means that MPI communications involving an irregular datatype may require that the message be copied during the transfer; if the message is buffered due to the send mode, then this transformation is not necessary.

Since the put and get transfer mechanisms on the Cray T3D have 4-byte granularity, MPI will copy messages and/or allocate receive buffer space when messages or buffers are misaligned. This is only occurs for misaligned or tightly-sized character messages.

3.3 Collective

MPI offers a rich set of collective functions which operate over an entire communicator. They allow commonly used communication patterns such as broadcast, scatter, gather and reduction to be encapsulated within a single call. Making these part of the standard has three clear advantages.

- Users are saved the effort of re-inventing these operations for each parallel application they write. This can result in reduced development time.
- The library developers can spend time choosing and implementing efficient algorithms tuned to particular hardware which all users can benefit from.
- The library developers will have access to lower level communication mechanisms (i.e. those used to develop the point to point communications) than the library users who would typically have to implement collective operations using the MPI point to point calls.

In order to allow for efficient implementations, MPI places a number of restrictions on the collective functionality. The most important of these, which have an impact on the MPI for T3D implementation are that communications must involve all processes within a communicator, all the collective operations are blocking and the user may make no assumptions about the synchronisation behaviour of the operations. The first of these restrictions allows all processors to know which other processors are taking part in an operation as soon as they enter it. The second and third allow ordering of the component operations of collective calls.

The implementation of collective communications within the MPI for T3D library uses a separate protocol mechanism from point to point communications. It is based on dedicated packet slots rather than the shared-access queue. Each processor provides a slot for every other processor. These have a number of advantages over the shared queue approach.

- As an individual slot is only ever accessed by one remote processor there is no need for the costly locking mechanism.
- Many collective operations require an all to one communication step. A shared queue can prove to be a bottleneck in such operations. With dedicated slots there is no such serialisation.
- The blocking nature of the collective calls allow the library to loop on a specific slot.

There are three operations used on a protocol slot. The first is a write operation which allows a process to send a protocol message to the slot reserved for it on a remote process. The second allows a local process to wait for the arrival of a protocol message on a specific slot. The third allows a local process to reset a slot ready for the next operation. Each process provides two slots for every other process. These are labeled the *up* and *down* slots. This naming reflects the primary underlying communication pattern which is tree based. It may also be thought of in terms of request/acknowledge.

The protocol slot mechanism allows processes to coordinate data transfer but the actual transfer is done using the same module as the point to point communication. It is

however abstracted slightly to hide buffering issues which are less complex than in point to point. All collective data transfers are driven by the sender using `shmem_put` and take the following form:

1. The receiver obtains a buffer description object given the address, size and MPI datatype of the receive buffer; this may involve allocation of a contiguous system buffer if the message layout is irregular. The receiver sends the buffer description through the protocol slot mechanism to the sender.
2. The sender takes the remote process number, the received buffer description and the address, size and datatype of the send buffer, and performs the transfer using the **BASIC COMMS** module. The sender sends an protocol message to the receiver indicating that the transfer has completed.
3. Once the receiver has received the completion protocol message it ensures the cache is in a consistent state and does any local copying and freeing required where the receive buffer was transformed in the first step.

We now present four collective operations that are indicative of those available in MPI, demonstrating how these make use of the lower level mechanisms described above.

Barrier The purpose of `MPI_Barrier` is to synchronise a group of processes. All the processes within the supplied communicator must make the call and until they all do, no process may leave the call.

The Cray T3D provides hardware support for barrier operations, accessed through the `barrier` function. However this requires all processors within the partition to make the call. When `MPI_Barrier` notes that a communicator spans the whole partition it makes use of the hardware barrier call, otherwise a software barrier is used. The software barrier uses a tree based algorithm. The initial implementation used a binary tree. Each child sends a protocol message up to its parent when it has received a protocol message from all of its children. When the root of the tree has received protocol messages from all its children, all processes must have made the barrier call. The root then sends an acknowledge message to each of its children. This message is propagated downward through the tree. Once a process has forwarded the acknowledge message to all its children it can exit the barrier call.

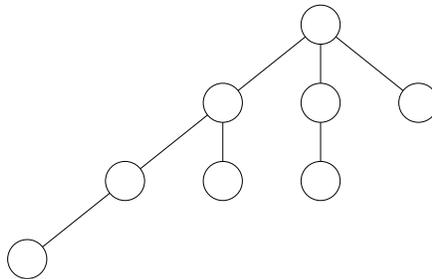


Figure 3: Collective Unbalanced Tree structure

It can be observed that using a binary tree, if all processes start the barrier at the same time, the root sits idle until the initial messages propagate up the tree. Also

once a process has forwarded the initial message it sits idle until it receives the acknowledgment. In addition, although the root exits as soon as the acknowledge broadcast begins the leaf processes must wait until the broadcast is complete. The tree used in the barrier algorithm has since been modified to reduce the amount of time processes spend idle, and reduce the time the last process spends in the barrier. The new tree structure is unbalanced and is illustrated in Figure 3. The result is that half the processes wait for an initial message from the other half. This is repeated for the subset which received a message, until only a single process is left. The single remaining process is the root of the tree and can send the acknowledge, which happens in exactly the reverse order to the initial message transfer, and the number of processes which have received the message then doubles on each time step.

Broadcast `MPI_Bcast` requires a buffer to be specified in each call, and a root nominated. The contents of the buffer at the root process will be copied to the buffers supplied by all the other processes. Broadcast uses a similar unbalanced tree to that used in the barrier operation, however it is necessary for the computation of parents and children to take account of the root nominated in the call (in the barrier call it is implicitly zero). The broadcast call makes use of the data transfer mechanism previously described. The broadcast operation has the form:

1. Create a buffer description (possibly allocating a temporary buffer).
2. Except at root, send the buffer description to parent and wait for an acknowledge packet.
3. Ensure the buffer is correct (no cache inconsistency, copy from temporary buffer if necessary).
4. For all children, get the remote buffer description and write the data into it. Send a confirmation of the transfer.

Reduce MPI provides a number of reduction operations, the most basic being `MPI_Reduce`.

In a reduction operation each process provides a buffer and the data in that buffer is combined with the other buffers using an operator supplied to the function. MPI has a number of predefined operators which work on the predefined basic datatypes. The user is also able to specify new operators to work on basic or user defined datatypes. The standard document suggests that predefined operators may be faster than user defined operators. In the MPI for T3D implementation, the built-in operations are simply pre-registered functions and have no speed advantage.

The `MPI_Reduce` function makes use of a binary tree, with buffers for the two children and the parent being combined before the result is forwarded. The binary tree is always rooted at the process with rank zero and the result forwarded to the nominated process if this is other than zero. This ensures the result is always the same regardless which process is nominated as the receiver of the final result.

All to All The initial implementation of the `MPI_Alltoall` function attempted to minimise the number of data transfer operations by using the algorithm represented in Figure 4 (wraparound communications are omitted for clarity). On each step the amount of data transferred is doubled, but there are only $O(\log_2 n)$ steps. This was expected to be the most efficient approach. However experimentation has shown

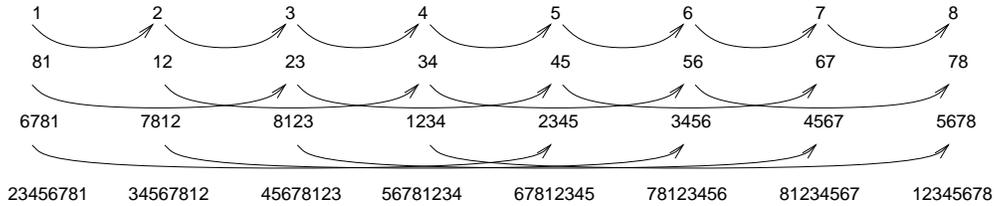


Figure 4: Initial All-to-All Algorithm structure

that the naïve approach of having every process send directly to every other process is faster. This is because the more sophisticated approach requires synchronisation at each step to ensure data is not forwarded until it has arrived. If the order of the communications in the naïve approach are arranged to ensure that two processes are not trying to send to the same process at any time the whole operation tends towards `shmem_put` bandwidth.

4 Performance Characteristics

In this section we present measurements of the performance of MPI on the T3D, and relate the results to the architecture of the implementation. We have measured the performance of point-to-point and collective communications using adaptations of standard benchmarks and performance measurements of our own design.

4.1 Point-to-point communication

The performance of MPI on the T3D has been measured using adaptations of the COMMS1 and COMMS2 benchmarks in the GENESIS [6] suite, written in C. Similar measurements of the performance of the CRI implementation of PVM for the T3D [8], have been obtained for comparison. Where possible the performance of MPI Standard and Synchronous modes are compared, as are traditional (`pk* send`, `recv upk*`) and single-call (`psend`, `precv`) PVM communications. The PVM data encoding `PvmDataRaw` was used.

The GENESIS COMMS1 benchmark involves pairs of processes reflecting a single message of fixed size and basic datatype back and forth. One process performs a send followed by a receive operation, where the other process performs a receive followed by a send. The time taken to complete a number of iterations of this loop is recorded. This divided by the number of iterations gives the average round-trip time; halving this quantity gives the average transfer time, or latency, for messages of the chosen size. End-to-end bandwidth is calculated by dividing message size by the latency. Figures 5 and 6 show the performance of MPI and PVM for our adaptation of COMMS1.

The transitions between protocols employed by MPI are clear on Figure 6. There is a sharp increase in latency for message over 24 bytes. This reflects the point at which message data can no longer be accommodated within protocol messages; this threshold marks the transition between **T** – or **TA** in Synchronous mode – and **RTA** protocol.

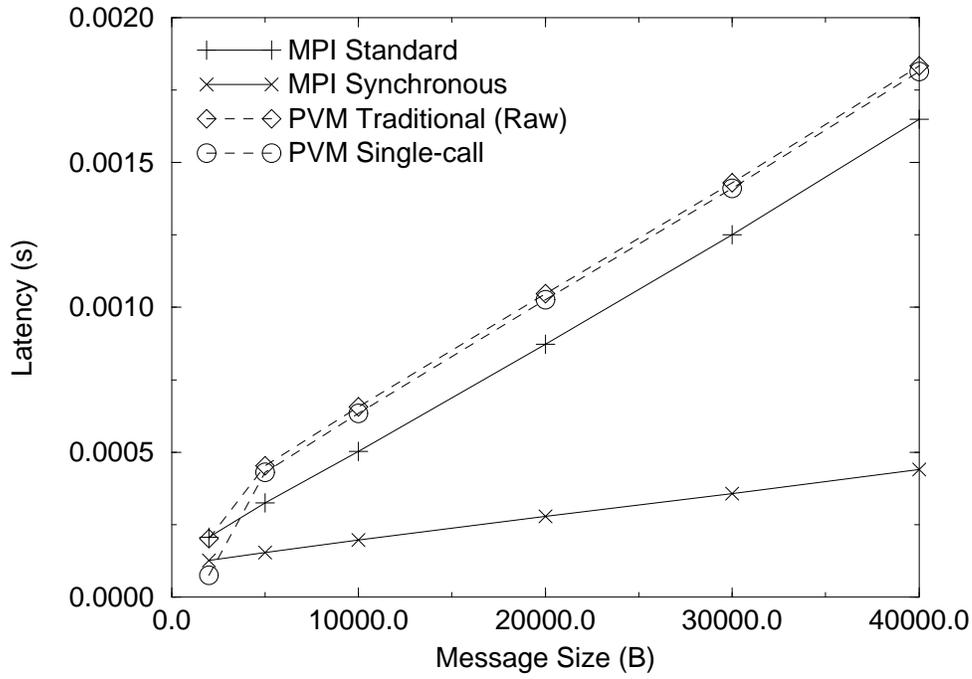


Figure 5: Performance of MPI and PVM on the T3D measured with COMMS1.

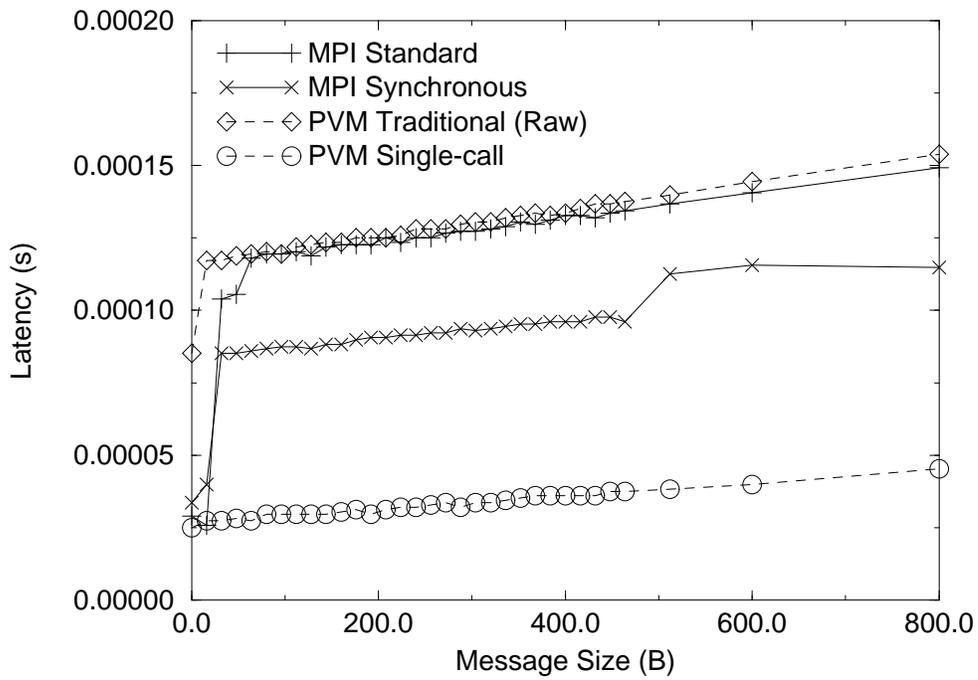


Figure 6: COMMS1 Performance of MPI and PVM on the T3D for small messages.

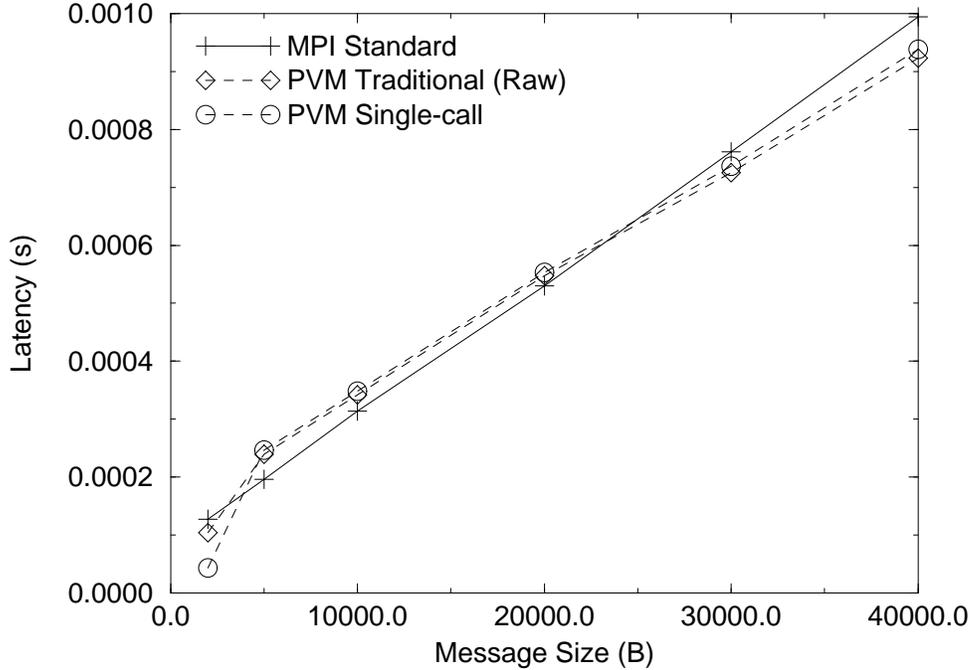


Figure 7: COMMS2 Performance of MPI and PVM on the T3D.

The **RTA** protocol is cheaper than **RAT**; however the asymptotic bandwidth obtained using **RTA** is that of remote memory get, half the put bandwidth possible with the **RAT** protocol. Synchronous mode can use either protocol and so has a second threshold where the put-based bandwidth dominates the extra cost of the **RAT** protocol. The transition from **RTA** to **RAT** protocol can be seen for MPI Synchronous mode at 512 bytes.

MPI Standard mode and traditional PVM communications require message buffering. This overhead is clear in Figure 6 as the gap between MPI Standard and Synchronous, and between PVM traditional and single-call communications. MPI is not alone in employing thresholds to control transfer; Figure 5 shows that the optimisation provided by single-call PVM only applies to messages under 2048 bytes.

Communications bandwidth is represented as the gradient of the latency incline. It is clear that MPI Standard mode and PVM tend to a similar steady bandwidth. Due to message buffering and the requirement that both MPI Standard and PVM send operations return independently of the receive operation, this is composed of message copy and remote memory get bandwidth. In contrast, MPI Synchronous achieves the far greater bandwidth obtained by remote memory put alone.

The GENESIS COMMS2 benchmark involves pairs of processes sending messages of fixed size and basic datatype simultaneously to one another. Both processes perform a send followed by a receive operation. This benchmark relies on an indirect model of communication, where the message is held by an intermediate agent or is buffered, allowing the send operation to return independently of a matching receive. For this reason MPI Synchronous mode is not used in this benchmark. Each iteration of this

loop involves a single transfer, rather than a two-transfer round-trip as for COMMS1, so the average iteration time gives the latency for the chosen message size. Figure 7 shows the performance of MPI and PVM for our adaptation of COMMS2.

It is clear that PVM is more effective at providing an indirect communications model as it achieves better bandwidth than MPI. Global state is implied by the PVM interface in having active send and receive buffers. In contrast, MPI requires per-transfer buffer management in order to provide secure non-blocking and persistent communications services. The extra cost of per-transfer buffer management explains the latency overhead and the small difference in steady bandwidth for MPI.

4.2 Collective communication

Benchmark results are provided for all the MPI collective calls described in Section 3.3. In addition to results for the MPI for T3D, results are also give for some equivalent PVM operations. All graphs show cost in μs against number of processes taking part. For rooted collective operations this cost is the time for all processes to complete the operation. All MPI results are marked with a square and all PVM results are marked with a circle.

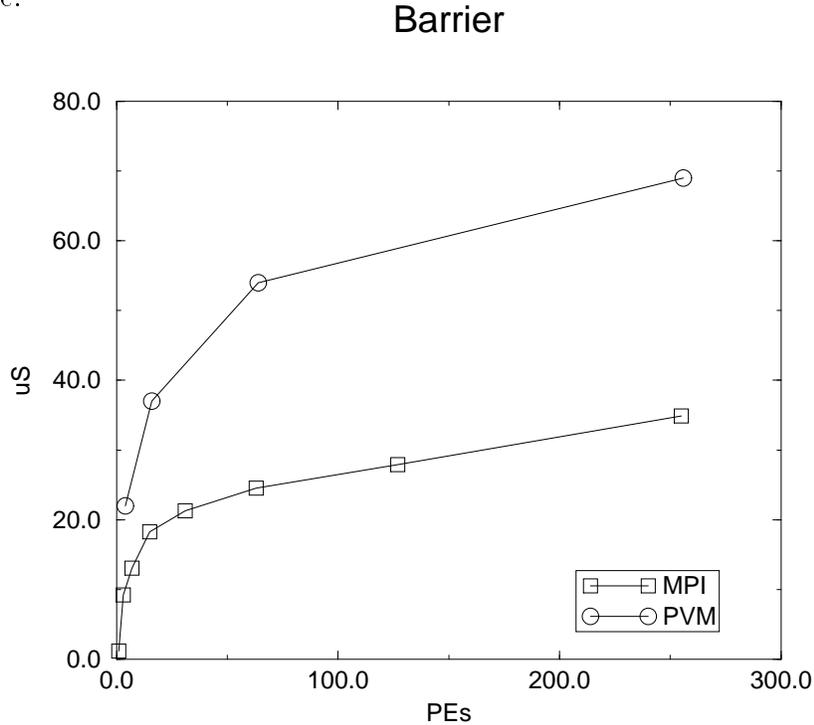


Figure 8: Performance of MPI and PVM Barrier on the T3D

Figure 8 shows the performance of the MPI and PVM barrier calls for varying numbers of processes. The MPI tests use a communicator one less than the power of two processes in a partition to avoid the use of the hardware barrier. Consistently, MPI has half of the cost of PVM.

Broadcast

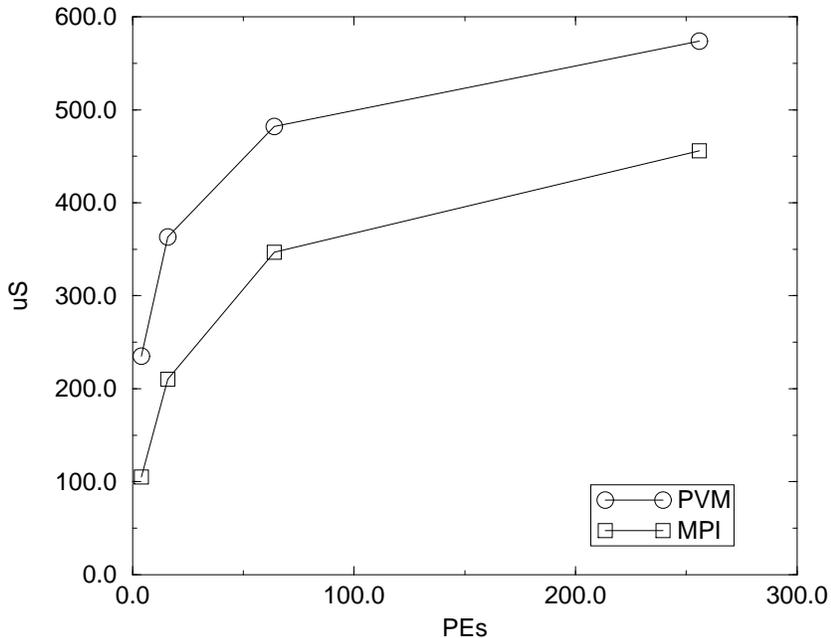


Figure 9: Performance of MPI and PVM Broadcast (2KByte) on the T3D

Figure 9 shows the cost of broadcasting a medium sized (2KByte) message across a varying number of processors. MPI performs consistently better than PVM. There appears to be a constant difference between the MPI and PVM results. This suggests that they are using a similar algorithm.

Figure 10 shows the cost of performing a vector sum (reduction) on an array of 128 integers (equivalent to 1 KByte). As the number of processes is doubled the number of vectors summed is also doubled. The operation cost increase is however a constant. This is to be expected as doubling the number only requires a single additional step (copy and sum).

Figure 11 shows the cost of performing an all to all exchange. Each processor sends a different 1 Kbyte block to every processor. The linear cost matches well with the approach of simply having each process send each of its outgoing blocks directly to their destination.

5 Further work

We are in the process of making enhancements and extensions to this software during 1995, and these will be incorporated into the software release from EPCC.

We are making a number of memory usage optimisations in the software, essentially pruning replication of data in different data structures. We are also providing the user with the ability to control the amount of memory used by the MPI software. This control will be

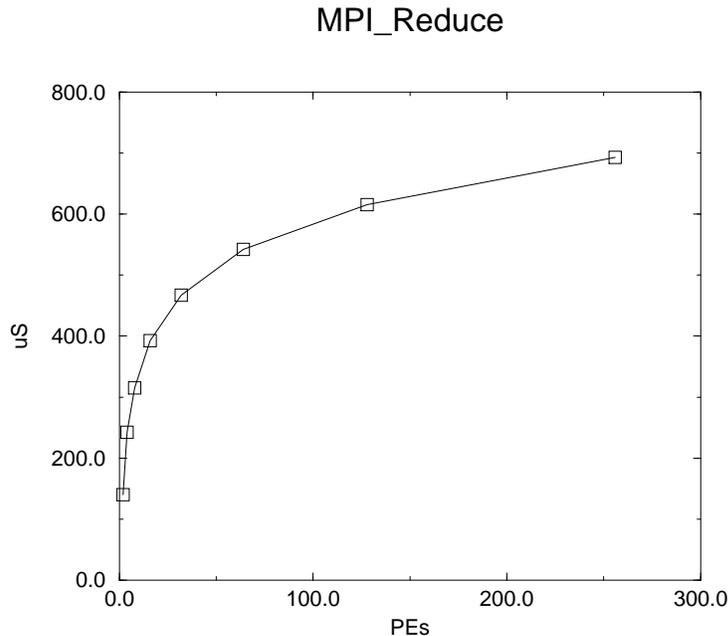


Figure 10: Performance of MPI Reduction (sum) on the T3D

exerted through UNICOS environment variables. `MPI_SM_POOL` will control the number of slots allocated in the shared-access protocol queue at each process. `MPI_BUFFER_MAX` will determine the maximum message length which will be buffered in system memory as part of Standard and Buffered mode point-to-point communications; larger messages will be transferred using a synchronous protocol. This variable can be used for performance optimisation in addition to memory optimisation. `MPI_BUFFER_TOTAL` will determine the maximum amount of system memory (from the heap) which will be used for standard-mode communications buffering.

We also plan to make performance enhancements in both point-to-point and collective communications. The memory-memory copy required to pack and unpack message data does not use `memcpy` since it has poor performance for the very common case of a user message which is word boundary aligned, contiguous, and a multiple of word length long; instead `shmem_put` from one area of local memory to another is used. There are better memory-memory copy methods available. The small message size behaviour can be improved by extending the idea of transferring very small messages in protocol. We could simply increase the possible size of a protocol message, in order to accommodate more user data. Another option is to implement an extended protocol consisting of multiple contiguous protocol packets, where the first packet contains protocol and subsequent packets contain user data. The second method has some advantages, for example the sender can transfer the user data without a memory-memory copy.

The MPI standard says that when a matching send and receive use incompatible data-types the result is an erroneous program. MPI implementations thus far have tended to ignore the problem of detecting datatype match errors, which removes the need to move datatype information between processors. This approach has been taken with MPI for the T3D, through a project carried out as part of the EPCC Summer Scholarship

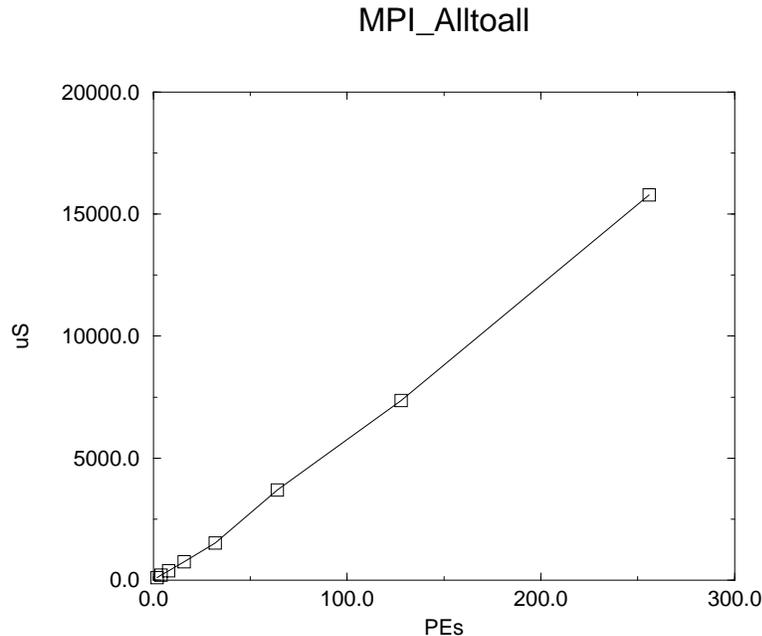


Figure 11: Performance of MPI All-to-All on the T3D

Programme [7]. This has provided a distributed type checking module for the implementation. The module uses a global datatype signature table, distributed across the PEs, to hold a minimised type signature for each datatype committed. All datatypes which share a compatible signature are assigned the same type signature identifier. The signature identifier is a single integer and can be packed into protocol messages. This allows the process which is responsible for transferring the data to know the signature of both the send and receive call, detect a datatype mismatch and raise an error. In principle, since the process performing the data transfer knows both type signatures, type coercion could be performed. The cost per message is a single integer compare as the type matching operation proper has been done at the `MPI_Type_commit` stage.

We are contributing to the second MPI Forum, MPI-2, and we expect to provide an early implementation of some of the MPI-2 features. We have a particular interest in single-sided communications, which have the potential to offer MPI users performance very close to `shmem_put` and `shmem_get` in portable programs.

6 Conclusions

CRI/EPCC MPI for Cray T3D demonstrates that it is possible to implement message-passing, through the MPI standard, with performance that closely matches hardware capabilities. Thus, Cray T3D users have access to a highly efficient and robust message-passing interface that provides a rich set of process-to-process and group-wide communications services, support for communicating libraries, and useful high-level functionality such as process topologies. MPI has already been implemented on a diverse range of MPP and workstation platforms, providing wide source code portability to T3D MPI

users. MPI for T3D is already widely used on many Cray T3D systems in the U.S. and Europe. MPI for T3D continues to be enhanced and developed by EPCC and continues to grow in use and popularity world-wide.

7 Acknowledgements

We express our gratitude to Peter Rigsbee for numerous useful discussions and guidance in use of low level interfaces for the T3D. The design and implementation of the MPI for Cray T3D software was managed by James G. Mills; we also thank R. Alasdair A. Bruce for helpful comments and encouragement.

References

- [1] CRAY T3D System Architecture. Technical Report HR-04033, Cray Research Inc., September 1993.
- [2] Ray Barriuso and Allan Knies. SHMEM User's Guide for C. Technical Report Revision 2.2, Cray Research Inc., August 1994.
- [3] R. Alasdair A. Bruce, James G. Mills, and A. Gordon Smith. CHIMP/MPI User Guide. Technical Report EPCC-KTP-CHIMP-V2-USER, Edinburgh Parallel Computing Centre, The University of Edinburgh, June 1994.
- [4] MPI Forum. MPI: A Message-Passing Interface Standard. Technical Report v1.0, University of Tennessee, May 1994.
- [5] MPI Forum. MPI: A Message-Passing Interface Standard. Technical Report v1.1, University of Tennessee, June 1995.
- [6] Ian Glendinning. The GENESIS Distributed-Memory Benchmark Suite, Release 3.0. Technical report, High Performance Computing Centre, University of Southampton, July 1994.
- [7] Jason Hunter. Datatype Checking in Cray T3D Native MPI. Technical Report EPCC-SS95-07, Edinburgh Parallel Computing Centre, September 1995.
- [8] Peter A. Rigsbee. CRAY T3D PVM Design & Implementation. Technical Report v1.1, Cray Research Inc., July 1994.