Building a High-performance Message-passing System for MEMORY CHANNEL Clusters

James V. Lawton
John J. Brosnan
Morgan P. Doyle
Seosamh D. Ó Riordáin
Timothy G. Reddin

The new MEMORY CHANNEL for PCI cluster interconnect technology developed by Digital (based on technology from Encore Computer Corporation) dramatically reduces the overhead involved in intermachine communication. Digital has designed a software system, the TruCluster MEMORY CHANNEL Software version 1.4 product, that provides fast user-level access to the MEMORY CHANNEL network and can be used to implement a form of distributed shared memory. Using this product, Digital has built a low-level message-passing system that reduces the communications latency in a MEMORY CHANNEL cluster to less than 10 microseconds. This system can, in turn, be used to easily build the communications libraries that programmers use to parallelize scientific codes. Digital has demonstrated the successful use of this message-passing system by developing implementations of two of the most popular of these libraries, Parallel Virtual Machine (PVM) and Message Passing Interface (MPI).

During the last few years, significant research and development has been undertaken in both academia and industry in an effort to reduce the cost of high-performance computing (HPC). The method most frequently used was to build parallel systems out of clusters of commodity workstations or servers that could be used as a virtual supercomputer.[1] The motivation for this work was the tremendous gains that have been achieved in reduced instruction set computer (RISC) microprocessor performance during the last decade. Indeed, processor performance in today's workstations and servers often exceeds that of the individual processors in a tightly coupled supercomputer. However, traditional local area network (LAN) performance has not kept pace with microprocessor performance. LANs, such as fiber distributed data interface (FDDI), offer reasonable bandwidth, since communication is generally carried out by means of traditional protocol stacks such as the user datagram protocol/internet protocol (UDP/IP) or the transmission control protocol/internet protocol (TCP/IP), but software overhead is a major factor in message-transfer time.[2] This software overhead is not reduced by building faster LAN network hardware. Rather, a new approach is needed—one that bypasses the protocol stack while preserving sequencing, error detection, and protection.

Much current research is devoted to reducing this communications overhead using specialized hardware and software. To this end, Digital has been working to make commercial Alpha clusters, descended from the original VAXcluster technology, available to scientific and technical users.[3,4] This cluster technology uses available commodity hardware and software to implement a high-performance communications subsystem.[5] The hardware interconnect that supports clustered operation is Encore Computer Corporation's patented MEMORY CHANNEL technology.[6] This interconnect provides a mechanism that allows the virtual address space of a process to be mapped so that a store instruction in one system is directly reflected in the physical memory of another system. We have developed software application programming interfaces (APIs) that provide user-level applications with this capability in a controlled and protected manner.

Data may then be transferred between the machines using simple memory read and write operations, with no software overhead, essentially utilizing the full performance of the hardware. This approach is similar to the one used in the Princeton SHRIMP project, where this process is described as Virtual Memory-Mapped Communication (VMMC). [7–10]

Figure 1 shows the relationship between the various components of our message-passing system. The first phase of our work involved designing a programming library and associated kernel components to provide protected, unprivileged access to the MEMORY CHANNEL network. Our objective in creating this library was to provide a facility much like the standard System V interprocess communication (IPC) shared memory functions available in UNIX implementations. Programmers could use the library to set up operations over the MEMORY CHANNEL interconnect, but they would not need to use the library functions for data transfer. In this way, performance could be maximized. This product, the TruCluster MEMORY CHANNEL Software, provides programmers with a simple, high-performance mechanism for building parallel systems.

TruCluster MEMORY CHANNEL Software delivers the performance available from the MEMORY CHANNEL network directly to user applications but requires a programming style that is different from that required for shared memory. This different programming style is necessary because of the different access characteristics between local memory and memory on a remote node connected through a MEMORY CHANNEL network. To make programming with the MEMORY CHANNEL technology relatively simple while continuing to deliver the hardware performance, we built a library of primitive communications functions. This system, called Universal Message Passing (UMP), hides the details of MEMORY CHANNEL operations from the programmer and operates seamlessly over two transports (initially): shared memory and the MEMORY CHANNEL interconnect. This allows seamless growth from a symmetric multiprocessor (SMP) to a full MEMORY CHANNEL cluster. Development can be done on a workstation, while production work is done on the cluster. The UMP layer was designed from the beginning with performance considerations in mind, particularly with respect to minimizing the overhead involved in sending small messages.

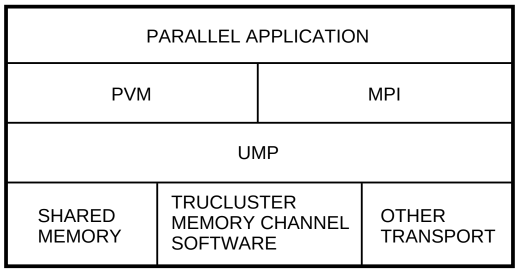


Figure 1
Message-passing System Architecture

Two distributed memory models are predominantly used in high-performance computing today:

1. Data parallel, which is used in High Performance Fortran (HPF).[11] With this model, the programmer uses parallel language constructs to indicate to the compiler how to distribute data and what operations should be performed on it. The problem is assumed to be regular so that the compiler can use one of a number of data distribution algorithms.
2. Message passing, which is used in Parallel Virtual Machine (PVM) and Message Passing Interface (MPI).[12–15] In this approach, all messaging is performed explicitly, so the application programmer determines the data distribution algorithm, making this approach more suitable for irregular problems.

It is not yet clear whether one of these approaches will predominate in the future or if both will continue to coexist. Digital has been working to provide competitive solutions for both approaches using MEMORY CHANNEL clusters. Digital's HPF work has been described in a previous issue of the *Journal*.[16,17] This paper is primarily concerned with message passing.

Building on the UMP layer, we constructed implementations of two common message-passing systems. The first, PVM, is a de facto standard for programmers who want to parallelize large scientific and technical applications. In addition to messaging functions, PVM also provides process control functions. The second, MPI, represents the efforts of a large group of academic and industrial users who are working together to specify a standard API for message passing. At this time, MPI does not provide any process control facilities. The performance of these PVM and MPI systems on MEMORY CHANNEL clusters exceeds that of the public-domain implementations.

MEMORY CHANNEL Overview

Encore's MEMORY CHANNEL technology is a high-performance network that implements a form of clusterwide shared virtual memory. In Digital's first implementation of this technology, it is a shared, 100-megabyte-per-second (MB/s) bus that provides a write-only path from a page of virtual address space on one node to a page of physical memory on another node (or multiple other nodes). The MEMORY CHANNEL network outperforms any traditional LAN technology that uses a bus topology. For example, a peak bandwidth of between 35 MB/s and 70 MB/s is possible with the current 32-bit peripheral component interconnect (PCI) MEMORY CHANNEL adapters, depending on the bandwidth of the I/O subsystem into which the adapter is plugged. Although the current MEMORY CHANNEL network is a shared bus, the plan for the next generation is to utilize a switched technology that will increase the aggregate bandwidth of the network beyond that of currently available switched LAN technologies. The latency (time to send a minimum-length message one way between two processes) is less than 5 microseconds (μs). The MEMORY CHANNEL network provides a communications medium with a low bit-error rate, on the order of 10^{-16}. The probability of undetected errors occurring is so small (on the order of the undetected error rate of CPUs and memory subsystems) that it is essentially negligible. A MEMORY CHANNEL cluster consists of one or more PCI MEMORY CHANNEL adapters on each node and a hub connecting up to eight nodes.

The MEMORY CHANNEL cluster supports a 512-MB global address space into which each adapter, under operating system control, can map regions of local virtual address space.[18] Figure 2 illustrates the MEMORY CHANNEL operation. Figure 2a shows transmission, and Figure 2b shows reception. A page table entry (PTE) is an entry in the system virtual-to-physical map that translates the virtual address of a page to the corresponding physical address. The MEMORY CHANNEL adapter contains a page control table (PCT) that indicates for each page of MEMORY CHANNEL global address space if that page is mapped locally and whether it is mapped for transmission or reception. Thus, to map a page of local virtual memory for transmission, all that is required is to

- Set up an entry in the system virtual-to-physical map to point to a page in the MEMORY CHANNEL adapter's PCI I/O address space window, which is directly mapped to the page in MEMORY CHANNEL space
- Enable the corresponding page entry in the PCT for transmission

Any write to the mapped virtual page will then result in a corresponding write to the MEMORY CHANNEL network.

To complete the circuit, the page of MEMORY CHANNEL space must be mapped to virtual memory on another node. This is accomplished on the other node by

- Making a page of physical memory nonpageable (wired)
- Creating a virtual region whose PTE points to the wired page
- Setting up the I/O direct memory access (DMA) scatter/gather map to point to the physical page
- Enabling the appropriate entry in the adapter's PCT for reception

Thus, when a MEMORY CHANNEL network packet is received that corresponds to the page that is mapped for reception, the data is transferred directly to the appropriate page of physical memory by the system's DMA engine. In addition, any cache lines that refer to the updated page are invalidated.

Subsequently, any writes to the mapped page of virtual memory on the first node result in corresponding writes to physical memory on the second node. This means that when a region in MEMORY CHANNEL space has been allocated and attached to a process, writes to that region are just simple stores to a process virtual address. The virtual address translates to a physical address that is mapped for transmission. Reads from that region are simply loads from a process virtual address, so the operating system is not involved in data transfer, with consequent reduction in overhead.

To use the MEMORY CHANNEL hardware, the operating system must provide certain basic services. Digital's cluster software includes a set of low-level primitives that can be used in the UNIX kernel. The functionality that these services provide includes

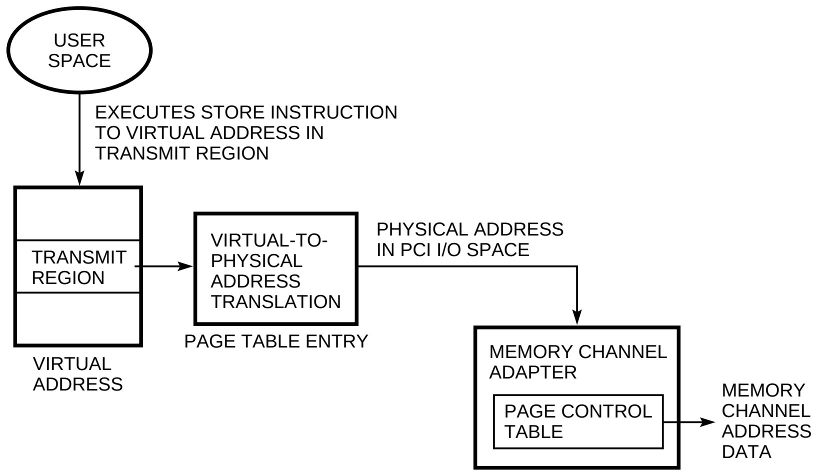
- Allocating and deallocating regions of MEMORY CHANNEL space for transmission or reception
- Allocating and deallocating cluster spinlocks
- Providing the capability to be notified when a page has been written (i.e., a notification channel)

TruCluster MEMORY CHANNEL Software

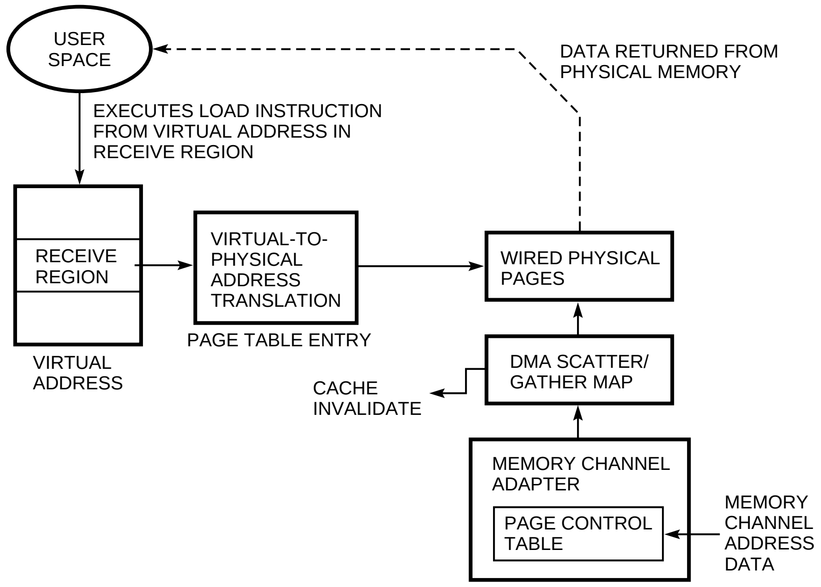
We designed the TruCluster MEMORY CHANNEL Software product to provide user-level access to the kernel functions that control the MEMORY CHANNEL hardware. The target audience for this technology is parallel software library builders and parallel compiler implementers. As shown in Figure 3, the product consists of two components layered on top of the kernel MEMORY CHANNEL functions:

1. A kernel subsystem that interfaces to the low-level kernel functions
2. A user-level API library

There were two choices in developing the product: provide simple user-level access to the basic functionality or build a more sophisticated system (e.g., a distributed shared memory [DSM] system). We chose to make a subset of the functionality of the operating system kernel primitives available to applications for two reasons. First, we did not initially know the degree of functionality required to provide generic user-level access to the MEMORY CHANNEL network for the long term. Second, the original purpose of the work was to give scientific and technical customers, rather than commercial cluster users, early access to the MEMORY CHANNEL network. As a result, the functionality we built into the product is


USER SPACE
EXECUTES STORE INSTRUCTION TO VIRTUAL ADDRESS IN TRANSMIT REGION
TRANSMIT REGION
VIRTUAL ADDRESS
VIRTUAL-TO-PHYSICAL ADDRESS TRANSLATION
PAGE TABLE ENTRY
PHYSICAL ADDRESS IN PCI I/O SPACE
MEMORY CHANNEL ADAPTER
PAGE CONTROL TABLE
MEMORY CHANNEL ADDRESS DATA

(a) Transmission


USER SPACE
DATA RETURNED FROM PHYSICAL MEMORY
EXECUTES LOAD INSTRUCTION FROM VIRTUAL ADDRESS IN RECEIVE REGION
RECEIVE REGION
VIRTUAL ADDRESS
VIRTUAL-TO-PHYSICAL ADDRESS TRANSLATION
PAGE TABLE ENTRY
WIRED PHYSICAL PAGES
DMA SCATTER/ GATHER MAP
CACHE INVALIDATE
MEMORY CHANNEL ADAPTER
PAGE CONTROL TABLE
MEMORY CHANNEL ADDRESS DATA

(b) Reception

Figure 2
MEMORY CHANNEL Operation

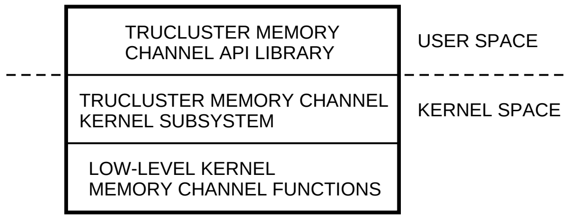

TRUCLUSTER MEMORY CHANNEL API LIBRARY
USER SPACE
TRUCLUSTER MEMORY CHANNEL KERNEL SUBSYSTEM
KERNEL SPACE
LOW-LEVEL KERNEL MEMORY CHANNEL FUNCTIONS

Figure 3
TruCluster MEMORY CHANNEL Software Architecture

a set of simple building blocks that are analogous to the System V IPC facility in most UNIX implementations. The advantage is that while a very simple interface is provided initially, the interface can later be extended as required, without losing compatibility with applications based on the initial implementation. Table 1 details the MEMORY CHANNEL API library functions that the product provides. An important feature to note is that when a MEMORY CHANNEL region is allocated using TruCluster MEMORY CHANNEL Software, a key is specified that uniquely identifies this region in the cluster. Other processes anywhere in the cluster can attach to the same region using the same key; the collection of keys provides a clusterwide namespace.

The MEMORY CHANNEL API library communicates with the kernel subsystem using kmodcall, a simple generic system call used to manage kernel subsystems. The library function constructs a command block containing the type of command (i.e.,

Table 1
TruCluster MEMORY CHANNEL API Library Functions

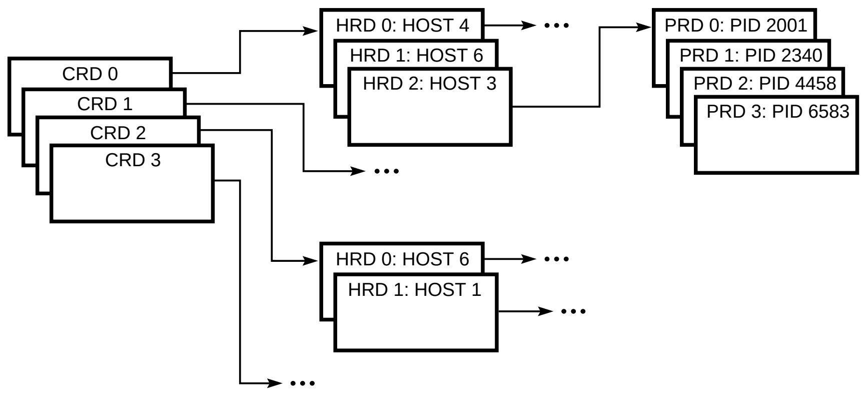
Function Name	Description
imc_asalloc	Allocates a region of MEMORY CHANNEL address space of a specified size and permissions and with a user-supplied key; the ability to specify a key allows other cluster processes to rendezvous at the same region. The function returns to the user a clusterwide ID for this region.
Imc_asattach	Attaches an allocated MEMORY CHANNEL region to a process virtual address space. A region can be attached for transmission or reception, and in shared or exclusive mode. The user can also request that the page be attached in loopback mode, i.e., any writes will be reflected back to the current node so that if an appropriate reception mapping is in effect, the result of the writes can be seen locally. The virtual address of the mapped region is assigned by the kernel and returned to the user.
Imc_asdetach	Detaches an allocated MEMORY CHANNEL region from a process virtual address space.
imc_asdealloc	Deallocates a region of MEMORY CHANNEL address space with a specified ID.
imc_lkalloc	Allocates a set of clusterwide spinlocks. The user can specify a key and the required permissions. Normally, if a spinlock set exists, then this function just returns the ID of that lock set; otherwise it creates the set. If the user specifies that creation is to be exclusive, then failure will result if the spinlock set exists already. In addition, by specifying the IMC_CREATOR flag, the first spinlock in the set will be acquired. These two features prevent the occurrence of races in the allocation of spinlock sets across the cluster.
imc_lkacquire	Acquires (locks) a spinlock in a specified spinlock set.
imc_lkrelease	Releases (unlocks) a spinlock in a specified spinlock set.
imc_lkdealloc	Deallocates a set of spinlocks.
imc_rderrcnt	Reads the clusterwide MEMORY CHANNEL error count and returns the value to the user. This value is not guaranteed to be up-to-date for all nodes in the cluster. It can be used to construct an application-specific error-detection scheme.
imc_ckerrcnt	Checks for outstanding MEMORY CHANNEL errors, i.e., errors that have not yet been reflected in the clusterwide MEMORY CHANNEL error count returned by imc_rderrcnt. This function checks each node in the cluster for any outstanding errors and updates the global error count accordingly.
imc_kill	Sends a UNIX signal to a specified process on another node in the cluster.
imc_gethosts	Returns the number of nodes currently in the cluster and their host names.

which library function has been called) and any parameters and sends it to the kernel subsystem using kmodcall. The kernel subsystem has a matching function for each of the library calls. When a command block is received, it is parsed and the appropriate function is called to service the request. All security and resource checks are performed inside the kernel.

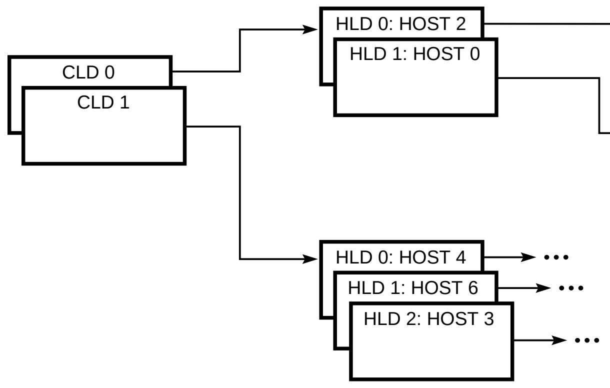
Figure 4 shows some of the data structures that the kernel services use. A clusterwide region of MEMORY CHANNEL space is allocated to store these management structures. This region contains a control structure and six linked lists of descriptors. The control structure manages MEMORY CHANNEL resources allocated using TruCluster MEMORY CHANNEL Software. Each region of MEMORY CHANNEL address space and each set of MEMORY CHANNEL spinlocks allocated using the product have a corresponding descriptor in the kernel data structure.

For each region of MEMORY CHANNEL address space allocated in the cluster, there is a cluster region descriptor (CRD) that contains information describing the region, including its clusterwide region identification number (ID), its size, key, permissions, creation time, and the UNIX user ID (UID) and group ID (GID) of the creating process. For an individual CRD, there is a host region descriptor (HRD) for each node that has the region mapped. This HRD contains the cluster ID of the node and other node-specific information. Finally, for a specific HRD, there is a process region descriptor (PRD) for each process on that node that is using the region. The PRD contains the UNIX process ID (PID) of the process that created the region and any process-specific information, such as virtual addresses.

Similarly, for each set of spinlocks allocated on the cluster there is a cluster lock descriptor (CLD) that contains information describing the spinlock set, including its clusterwide lock ID, the number of spinlocks in the set, the key, permissions, creation time, and the UID and GID of the creating process. For an individual CLD, there is a host lock descriptor (HLD) for each node that is using the spinlock set. The HLD contains the cluster ID of the node and other node-specific information about the spinlock set. For a specific HLD, there is a process lock descriptor (PLD) for each process on that node that is using the spinlock



(a) Regions



(b) Spinlocks

KEY:

CLD	CLUSTER LOCK DESCRIPTOR
CRD	CLUSTER REGION DESCRIPTOR
HLD	HOST LOCK DESCRIPTOR
HRD	HOST REGION DESCRIPTOR
PLD	PROCESS LOCK DESCRIPTOR
PRD	PROCESS REGION DESCRIPTOR

Figure 4
TruCluster MEMORY CHANNEL Kernel Data Structures

set. The PLD contains the PID of the process that created the spinlock set and any process-specific information about the spinlock set.

All these cluster data structures have pointers that cannot be updated atomically. In our implementation, they actually consist of two copies (old and new) and a toggle that indicates which of the two copies is valid. The toggle is switched from an old copy to a new copy only when the new copy is known to be consistent, so that failure of a cluster member while modifying the structures can be tolerated.

Figure 4a illustrates a hypothetical situation in which four regions of MEMORY CHANNEL space have been allocated on the cluster. The first region, with descriptor CRD 0, is mapped on three nodes: host 4, host 6, and host 3. The diagram also shows four processes on host 3 with the region mapped and lists the PID of each process. Figure 4b shows a similar situation for spinlocks. Two sets of spinlocks have been allocated. The first, with descriptor CLD 0, is mapped on two nodes of the cluster: host 2 and host 0. One process on each of these nodes is currently using the spinlock set.

Command Relay

The command relay is a kernel-level framework that enables the execution of a generic service routine on another node within the cluster. It functions as a simple kernel remote procedure call (RPC) mechanism based on fixed unidirectional message locations (mailboxes) and MEMORY CHANNEL notification channels to awaken the server kernel thread. Figure 5 shows the major components of the command relay and illustrates its operation between two hosts in a cluster. A client kernel thread on one host invoking a service and the corresponding server kernel thread on another cluster host communicate data using a defined bidirectional command/response block, known as a parameter structure. The client and server routines

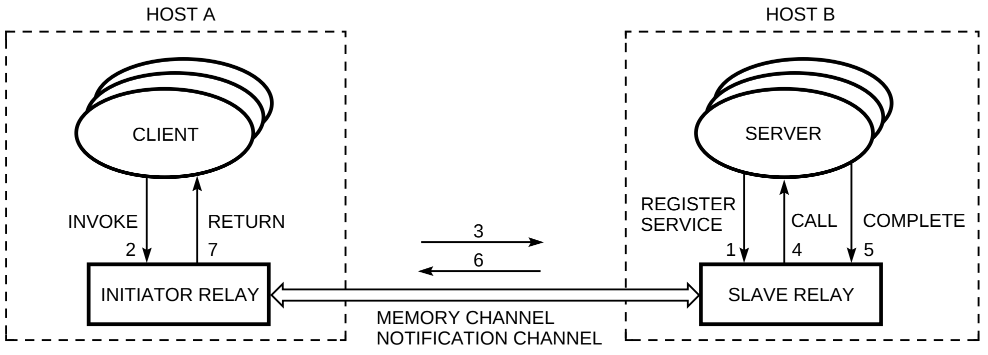


Figure 5
Command Relay Operation

must conform to this interface and must be reliable, i.e., they must always return to the caller. The server can call any kernel function. Server routines are registered (step 1 in Figure 5) using a clusterwide service ID. A kernel thread invoking a remote service passes a packed parameter structure to the command relay, together with a destination node ID and a service ID (step 2). This command relay then adds process credentials and builds a service protocol data unit (SPDU). Using a MEMORY CHANNEL notification channel, it signals the remote node and passes the SPDU by means of a mailbox in MEMORY CHANNEL space (step 3). The server parses the SPDU and calls the requested service function, passing it the parameter structure (step 4). When the service function completes (step 5), its return status and any data values are packed into an SPDU and placed into the mailbox, and the initiating relay is signaled (step 6). The initiator then unpacks the data from the SPDU and returns the appropriate status and values to the client kernel thread (step 7).

All calls to the command relay are synchronous and serialized. The invoking kernel thread blocks until the server returns. Requests to the command relay subsystem are treated on a first-come first-served basis, and calls to a busy relay block until the relay becomes free. Relays are automatically created between all nodes in the cluster.

The command relay mechanism makes it possible to send a UNIX signal to a process on another node within the MEMORY CHANNEL cluster. The imc_kill library function uses the command relay to invoke the registered kernel server routine for cluster signals on the remote node, which, in turn, calls the kernel kill function directly with the PID supplied.

Initial Coherency

When a process on a cluster member maps a region of MEMORY CHANNEL address space for both reception and transmission, any writes to the transmit region by that process are reflected as changes to the corresponding receive region. If another process on another cluster node subsequently maps the same region for reception, the contents of its receive region are indeterminate; i.e., the two processes do not have a coherent view of that region. This situation is known as the initial coherency problem. For an application developer, this problem makes it difficult to treat MEMORY CHANNEL address space as another form of shared memory. Applications can overcome this difficulty by using some form of start-up synchronization. However, all developers would have to implement these solutions separately. To increase the usability of TruCluster MEMORY CHANNEL Software, the design team decided to build in the ability to request coherent allocation of MEMORY CHANNEL address space across the cluster. Developers can specify this as an option in the call to imc_asalloc. As a result, a process can attach a MEMORY CHANNEL region for reception following any updates and still share a common view of the region with other processes in the cluster.

A special process, called the mapper, is used to provide the virtual address space to hold the coherent user space mappings. When the kernel subsystem receives a request for coherent allocation, it allocates the MEMORY CHANNEL region as normal and then maps the region for reception into the virtual address space of the mapper process. The command relay mechanism then causes all the other nodes in the cluster to allocate the same region and map it for reception into the address space of the mapper process on each node. Since multiple user-level processes on a node that attach a particular region for reception share the same physical memory, all updates to the region are seen by late-joining processes on any node in the cluster. If the requesting process exits, the region will still be allocated to the mapper, so that another allocation of the same region on that node will result in a coherent picture of that region. The region is fully deallocated (i.e., from all the mapper processes) when the last application process allocating the region either exits or explicitly deallocates the region.

Given the usefulness of coherent allocations, it may seem unusual that we made this feature an option rather than the default. There are several reasons for this. With coherent allocations, the associated physical memory becomes nonpageable on all nodes within the cluster, and, as such, it consumes physical resources. In addition, every outbound write to such a region results in an inbound write to the physical memory of each node in the cluster. For some application designs, it may be more desirable to create a region that is written by one node and only read by other nodes. Also, automatically reflecting all writes back to a node, as is done for coherent regions, consumes twice as much bandwidth on the PCI bus.

Late Join and Failure Resilience

To provide an operational environment in which nodes can join or leave the cluster at any time, the kernel subsystem needs to overcome a number of problems resulting from late join and node failure. In fact, the kernel subsystem is subject to the same difficulties of initial coherency as application-level processes. To manage user space allocations, late-joining nodes require a coherent view of the cluster data structures. Moreover, failure of an existing node can result in out-of-date or, even worse, corrupt data structures in the subsystem's control region. To contain the failure, corrupt data structures must be repaired.

Low-level kernel routines detect cluster membership change and wake up a management service thread on each node that performs operations local to that node. The first management service thread to acquire a specific spinlock is elected to manage clusterwide updates.

In the case of late join, the management service thread updates local state to reflect the new configuration. The thread that has been designated to manage clusterwide updates is responsible for providing the late-joining node with an up-to-date copy of the cluster data structures. When triggered by the new node, the thread retransmits the contents of the data structures so that the late-joining node has a fully up-to-date view of allocations and resource usage.

When a node fails, the thread elected to manage clusterwide updates must examine the entire management data structure and repair it appropriately. Repair is necessary when the failing node that is in the process of updating the global data structures has left these clusterwide updates in an unstable state. Repair is possible because all updates to global data structures use two copies of the structure (old and new, as described previously), which means that the structures can be reset easily to a stable state. If the failed node was not actively updating the data structures at the time of the failure, the management thread simply removes all resources allocated to the failed node.

Error Management

The MEMORY CHANNEL hardware provides a very low error rate, ordering guarantees, and an ability to detect remote error situations quickly, making it possible to construct simple error detection and recovery protocols. A kernel interrupt service routine detects cluster errors and updates an error counter that reflects the clusterwide error count. A low-level kernel routine returns the value of this counter. Due to timing considerations, it is not possible to guarantee that this count will be up-to-date with respect to possible errors on remote nodes. A low-level kernel routine that efficiently reads the error status of remote MEMORY CHANNEL adapters and detects unprocessed errors is provided. This routine uses a hardware feature, known as an ACK page, that is specifically designed to facilitate error detection. A write to such a page results in the error status of each MEMORY CHANNEL adapter being written to successive locations of the corresponding reception mapped region.

During development, we built simple interfaces to access these low-level routines, thereby allowing message-passing libraries to build in error management. Because the method of getting into and out of the kernel is a generic one, the overhead is high—approximately 30 μs. This compares poorly with the raw latency for short messages, which is less than 5 μs. To provide suitable performance, we reimplemented the functions to execute totally in user space. As a result, when an application reads the error count for the first time (using imc_rderrcnt), the kernel value of the error count is mapped for read-only access into the virtual address space of the process. Subsequent reads of the error count are then simply reads of a memory location. Similarly, when an application calls the check error service (using imc_ckerrcnt) for the first time, ACK pages are transparently mapped into the virtual address space of the process, and the error detection is performed at hardware speeds directly from user space. This has been measured at less than 5 μs.

The following sequence can be used to guarantee detection of intervening errors by the transmitter:

1. Save the error count.
2. Write the message.
3. Check the error count (using imc_ckerrcnt).

If the transmitter writes the saved error count at the end of the message, the message receiver can determine if any intervening errors have occurred by simply comparing the error count in the message with the current value using imc_rderrcnt. This is possible because of the sequencing guarantees built into the MEMORY CHANNEL network. Using imc_rderrcnt and imc_ckerrcnt, the programmer can build an appropriate error detection and/or recovery scheme that meets the performance requirements of the application.

Performance

The performance of TruCluster MEMORY CHANNEL Software on a pair of AlphaServer 4100 5/300 machines is presented in Table 2. These measurements were made using version 1.5 MEMORY CHANNEL adapters. The bandwidth (64 MB/s) and latency (2.9 μs) achieved using this system are essentially that of the hardware, since no system overhead is involved. The times required to perform the error-checking functions indicate that the overhead of calling imc_rderrcnt is much less than that of imc_ckerrcnt. This is because the latter has to synchronize with all other members of the cluster. Protocols that rely on receiver-only error detection (using imc_rderrcnt) will therefore have a lower overhead.

Programming with TruCluster MEMORY CHANNEL Software

The MEMORY CHANNEL network imposes some unique restrictions on the programmer. Since the network requires separate transmit and receive regions, any read-write memory location that is to be visible clusterwide must have two addresses: a read address and a write address. Attempts to read from a write address typically cause a segmentation violation. MEMORY CHANNEL address space can be used like shared memory. Unlike shared memory, though, its latency is visible to the programmer, who must consider latency effects when writing to a clusterwide location.

As an example of programming with TruCluster MEMORY CHANNEL Software, Figure 6 shows a simple program that implements a global counter, performs some work, and then decrements the global counter and exits. For the purposes of this example, assume that multiple copies of the program are run concurrently on different machines in a cluster. Such operation requires synchronization to ensure safe access to shared data in MEMORY CHANNEL space. The example program first allocates MEMORY CHANNEL regions for transmission and reception and attaches them to process virtual addresses. Next, a set of spinlocks is created (unless it already exists). The first copy of the program to create the spinlock set acquires the first lock in the set and initializes the global region, whereupon it releases the spinlock and continues. All other copies of the program wait in imc_lkacquire until the spinlock is released by the first copy. Each copy in turn acquires the lock itself, increments the process counter, and releases the lock. The copies then perform some work in parallel. When each program has finished its portion of the work, it decrements the global process counter (using the spinlock to control access again). Finally, the spinlock set and shared regions are deallocated. Several examples of code illustrating these topics are contained in the *TruCluster MEMORY CHANNEL Software Programmer's Manual.*[19] We have found that implementing a simple message-passing layer on top of TruCluster MEMORY CHANNEL Software is a more effective solution than programming directly with MEMORY CHANNEL regions, as described in the next section.

Several features described above were not initially present in the TruCluster MEMORY CHANNEL Software product. As a result of our experience implementing UMP and the higher PVM and MPI layers, we added the following features:

- Initial coherency
- Command relay
- Cluster signals
- User-level error checking

Table 2
TruCluster MEMORY CHANNEL Software Performance

Sustained bandwidth	64 MB/s
Latency	2.9 μs
Read error count (imc_rderrcnt)	<1 μs
Check error count (imc_ckerrcnt)	<5 μs

Universal Message Passing

The Universal Message Passing (UMP) library is designed to provide a foundation for implementing efficient message-passing systems on the MEMORY CHANNEL network. From the outset, we were aware that there would be a demand for PVM and MPI implementations and that other implementations might follow. We felt that it would be easier to construct high-performance message-passing systems if we provided a thin layer that could efficiently handle the restrictions that the MEMORY CHANNEL network imposes.

The goals in developing UMP were to

- Simplify the construction of message-passing systems utilizing the MEMORY CHANNEL network by hiding the details of the underlying communications transport (initially, shared memory or MEMORY CHANNEL).
- Optimize performance and exploit the low latency of the MEMORY CHANNEL network; the initial goal for latency over the MEMORY CHANNEL network using PVM was to achieve less than 30 μs.
- Ease the development of parallel message-passing libraries by providing a simple set of message-passing functions.
- Perform only basic communications; any more complex operations (e.g., process control) would be performed by a higher layer.
- Act as a convergence center for possible future interconnects.

```
extern long asm(const char *, ...);
#pragma intrinsic(asm)
#define mb() asm("mb")

#include <sys/types.h>
#include <sys/imc.h>

main ()
{
  int status, i, locks=4, temp, errors;
  imc_asid_t region_id;                                /* MC region ID */
  imc_lkid_t lock_id;                            /* MC spinlock set ID */
  typedef struct {                           /* Shared data structure */
    volatile int processes;
    volatile int pattern[2047];
  } shared_region;
  shared_region *region_read, *region_write;
  caddr_t read_ptr = 0, write_ptr = 0;

  /* Allocate a region of coherent MC address space and attach to */
  /* process VA */
  imc_asalloc(123, 8192, IMC_URW, IMC_COHERENT, &region_id);
  imc_asattach(region_id, IMC_TRANSMIT, IMC_SHARED, IMC_LOOPBACK, &write_ptr);
  imc_asattach(region_id, IMC_RECEIVE, IMC_SHARED, 0, &read_ptr);

  region_read = (shared_region *)write_ptr;
  region_write = (shared_region *)read_ptr;

  /* Allocate a set of spinlocks and atomically acquire the first lock */
  status = imc_lkalloc(456, &locks, IMC_LKU, IMC_CREATOR, &lock_id);
  errors = imc_rderrcnt();
  if (status == IMC_SUCCESS) {
     do {
       region_write->processes = 0;       /* Initialize the global region */
       for (i=0; i<2047; i++)
         region_write->pattern[i] = i;
       i--;
       mb();
     } while (imc_ckerrcnt(&errors) || region_read->pattern[i] != i) ;
    imc_lkrelease(lock_id, 0);
  } else if (status == IMC_EXISTS) {
    imc_lkalloc(456, &locks, IMC_LKU, 0, &lock_id);
    imc_lkacquire(lock_id, 0, 0, IMC_LOCKWAIT);
    temp = region_read->processes + 1;   /* Increment the process counter */
    errors = imc_rderrcnt();
    do {
      region_write->processes = temp;
      mb();
    } while (imc_ckerrcnt(&errors) || region_read->processes != temp) ;
    imc_lkrelease(lock_id, 0);
  }

       .
       .  (Body of program goes here)
       .

  /* clean up */
  imc_lkacquire(lock_id, 0, 0, IMC_LOCKWAIT);
  temp = region_read->processes - 1;     /* Decrement the process counter */
  errors = imc_rderrcnt();
  do {
    region_write->processes = temp;
    mb();
  } while (imc_ckerrcnt(&errors) || region_read->processes != temp) ;

  imc_lkrelease(lock_id, 0);
  imc_lkdealloc(lock_id);                        /* Deallocate spinlock set */
  imc_asdetach(region_id);                          /* Detach shared region */
  imc_asdealloc(region_id);                /* Deallocate MC address space */
}
```

Figure 6
Programming with TruCluster MEMORY CHANNEL Software

These goals placed some important constraints on the architecture of UMP, particularly with regard to performance. This meant that design decisions had to be constantly evaluated in terms of their performance impact. The initial design decision was to use a dedicated point-to-point circular buffer between every pair of processes. These buffers use producer and consumer indexes to control the reading and writing of buffer contents. The indexes can be modified only by the consumer and producer tasks and allow fully lockless operation of the buffers. Removing lock requirements eliminates not only the software costs associated with lock manipulation (in the initial implementation of TruCluster MEMORY CHANNEL Software, acquiring and releasing an uncontested spinlock takes approximately 130 μs and 120 μs, respectively) but also the impact on processor performance associated with Load-locked/Store-conditional instruction sequences.

Although this buffering style eliminates lock manipulation costs, it results in an exponential demand for storage and can limit scalability. If there are N processes communicating using this method, that implies N^2 buffers are required for full mesh communication. MEMORY CHANNEL address space is a relatively scarce resource that needs to be carefully husbanded. To manage the demand on cluster resources as fairly as possible, we decided to do the following:

- Allocate buffers sparsely, i.e., as required up to some default limit. Full N^2 allocation would still be possible if the user increased the number of buffers.
- Make the size of the buffers configurable.
- Use lock-controlled single-writer, multiple-reader buffers to handle both the overflow from the N^2 buffer and fast multicast. One of these buffers, called outbufs, would be assigned to each process using UMP upon initialization.

Note that while the channel buffers are logically point-to-point, they may be implemented physically as either point-to-point or broadcast. For example, in the first version of UMP, we used broadcast MEMORY CHANNEL mappings for the sake of simplicity. We are currently modifying UMP to use point-to-point MEMORY CHANNEL mappings, both to increase available bandwidth and to exploit a switched MEMORY CHANNEL network.

Figure 7 shows several tasks communicating in a cluster and illustrates how the two types of UMP buffers are used. Task 1 and task 2 are executing on node 1, while task 3 is executing on node 2. In the diagram, the channel buffers are located under the task in whose virtual address space they reside to indicate visually that they reside in the virtual address space of the destination task. In the figure, task 1 communicates

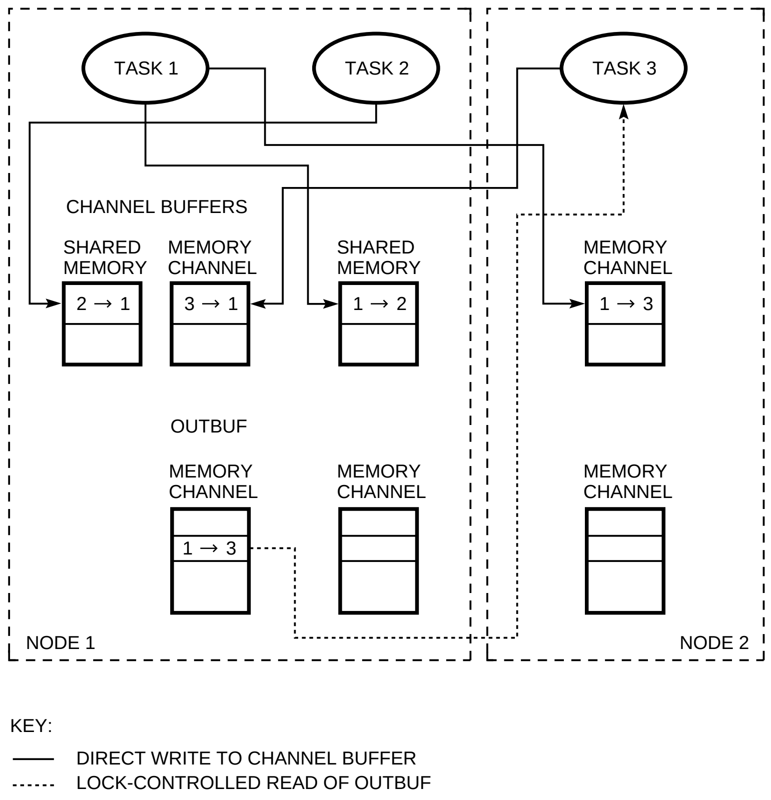


Figure 7
Cluster Communication Using UMP

with task 2 using UMP channel buffers in shared memory, shown as 1→2 and 2→1. Task 1 and task 3 communicate using UMP channel buffers in MEMORY CHANNEL space, shown as 1→3 and 3→1. Task 3 is reading a message from task 1 using an outbuf. The outbuf can be written only by task 1 but is mapped for transmission to all other cluster members. On node 2, the same region is mapped for reception. Access to each outbuf is controlled by a unique cluster spinlock.

Our rationale for taking this approach is that a short software path is more appropriate for small messages because overhead dominates message transfer time, whereas the overhead of lock manipulation is a small component of message transfer time for large messages. We felt that this approach helped to control the use of cluster resources and maintained the lowest possible latency for short messages yet still accommodated large messages. Note that outbufs are still fixed-size buffers but are generally configured to be much larger than the N^2 buffers.

This approach worked for PVM because its message transfer semantics make it acceptable to fail a message send request due to buffer space restrictions (e.g., if both the N^2 buffer and the outbuf are full). When we analyzed the requirements for MPI, however, we found that this approach was not possible. For this reason, we changed the design to use only the N^2 buffers. Instead of writing the message as a single operation, the message is streamed through the buffer in a series of fragments. Not only does this approach support arbitrarily large messages, but it also improves message bandwidth by allowing (and, for messages exceeding the available buffer capacity, requiring) the overlapped writing and reading of the message. Deadlock is avoided by using a background thread to write the message. Since overflow is now handled using the streaming N^2 buffers, outbufs were not necessary to achieve the required level of performance for large messages and were not implemented. Outbufs are retained in the design to provide fast multicast messaging, even though in the current implementation they are not yet supported.

Achieving the performance goals set for UMP was not easy. In addition to the buffer architecture described earlier, several other techniques were used.

- No syscalls were allowed anywhere in the UMP messaging functions, so UMP runs completely in user space.
- Calls to library routines and any expensive arithmetic operations were minimized.
- Global state was cached in local memory wherever possible.
- Careful attention was paid to data alignment issues, and all transfers are multiples of 32-bit data.

At the programmer's level, UMP operation is based on duplex point-to-point links called channels, which correspond to the N^2 buffers already described. A channel is a pair of unidirectional buffers used to provide two-way communication between a pair of process endpoints anywhere in the cluster. UMP provides functions to open a channel between a pair of tasks. While the resources are allocated by the first task to open the channel, the connection is not complete until the second task also opens the same channel. Once a channel has been opened by both sides, UMP functions can be used to send and receive messages on that channel. It is possible to direct UMP to use shared memory or MEMORY CHANNEL address space for the channel buffers, depending on the relative location of the associated processes. In addition, UMP provides a function to wait on any event (e.g., arrival of a message, creation or deletion of a channel). In total, UMP provides a dozen functions, which are listed in Table 3. Most of the functions relate to initialization, shutdown, and miscellaneous operations. Three functions establish the channel connection, and three functions perform all message communications.

UMP channels provide guaranteed error detection but not recovery. Through the use of TruCluster MEMORY CHANNEL Software error-checking routines, we were able to provide efficient error detection in UMP. We decided to let the higher layers implement error recovery. As a result, designers of higher layers can control the performance penalty they incur by specifying their own error recovery mechanisms, or, since reliability is high, can adapt a fail-on-error strategy.

Performance

UMP avoids any calls to the kernel and any copying of data across the kernel boundary. Messages are written directly into the reception buffer of the destination channel. Data is copied once from the user's buffer to physical memory on the destination node by the sending process. The receiving process then copies the data from local physical memory to the destination user's buffer. By comparison, the number of copies involved in a similar operation over a LAN using sockets is greater. In this case, the data has to be copied into the kernel, where the network driver uses DMA to copy it again into the memory of the network adapter. At this point the data is transmitted onto the LAN.

The first version of UMP used one large shared region of MEMORY CHANNEL space to contain its channel buffers and a broadcast mapping to transmit this simultaneously to all nodes in the cluster. This version of UMP also used loopback to reflect transmissions back to the corresponding receive region on the sending node, which resulted in a loss of available bandwidth. Using our AlphaServer 2100 4/190 development machines, we measured

Table 3
UMP API Functions

Function Name	Description
ump_init	Initializes UMP and allocates the necessary resources.
ump_exit	Shuts down UMP and deallocates any resources used by the calling process.
ump_open	Opens a duplex channel between two endpoints over a given transport (shared memory or MEMORY CHANNEL). Channel endpoints are identified by user-supplied, 64-bit integer handles.
ump_close	Closes a specified UMP channel, deallocating all resources assigned to that channel as necessary.
ump_listen	Registers an endpoint for a channel over a specified transport. This can be used by a server process to wait on connections from clients with unknown handles. This function returns immediately, but the channel is created only when another task opens the channel. This can be detected using ump_wait.
ump_wait	Waits for a UMP event to occur, either on one specified channel to this task or on all channels to this task.
ump_read	Reads a message from a specified channel.
ump_write	Writes a message to a specified channel. This function is blocking, i.e., it does not return until the complete message has been written to the channel.
ump_nbread	Starts reading a message from a channel, i.e., it returns as soon as a specified amount of the message has been received, but not necessarily all the message.
ump_nbwrite	Starts writing a message to a specified channel, i.e., it returns as soon as the write has started. A background thread will continue writing the message until it is completely transmitted.
ump_mcast	Writes a message to a specified list of channels.
ump_info	Returns UMP configuration and status information.

- Latency: 11 μs (MEMORY CHANNEL), 4 μs (shared memory)
- Bandwidth: 16 MB/s (MEMORY CHANNEL), 30 MB/s (shared memory)

To increase bandwidth, we modified UMP to use transmit-only regions for its channel buffers, thus eliminating loopback. The performance measured for the revised UMP using the same machines was

- Latency: 9 μs (MEMORY CHANNEL), 3 μs (shared memory)
- Bandwidth: 23 MB/s (MEMORY CHANNEL), 32 MB/s (shared memory)

Figure 8 shows the message transfer time and Figure 9 shows the bandwidth for various message sizes for the revised version of UMP using both blocking and non-blocking writes over shared memory and the MEMORY CHANNEL network. Using newer AlphaServer 4100 5/300 machines, which have a faster I/O subsystem than the older machines, and version 1.5 MEMORY CHANNEL adapters, the measured latency is 5.8 μs (MEMORY CHANNEL), 2 μs (shared memory). The peak bandwidth achieved is 61 MB/s (MEMORY CHANNEL), 75 MB/s (shared memory). In the non-blocking cases, the buffer size used was 256 kilobytes (KB) for shared memory and 32 KB for MEMORY CHANNEL. Further work is under way to improve the performance using shared memory as the transport. This work is aimed at eliminating the high-end falloff in bandwidth in the blocking case and the notch when the message size exceeds the buffer size in the nonblocking case. Note that these effects are not displayed in the MEMORY CHANNEL results.

Message-passing Libraries

Message-passing libraries provide the programmer with a set of facilities to build parallel applications. Typically, these services include the ability to send and receive a variety of data types to and from other peer processes in a variety of modes, as well as collective operations that span a set of peer processes. Other facilities may be provided in addition to the basic set, e.g., PVM provides functions for managing PVM processes (spawning, killing, signaling, etc.), whereas MPI (at least in its first revision, MPI-1) does not. PVM is probably the most widely used message-passing system. It has been available for approximately five years, and implementations are available for a wide variety of platforms. MPI is an emerging standard for message passing that is growing rapidly in popularity; many new applications are being written for it.

Parallel Virtual Machine

Parallel Virtual Machine (PVM) is supported on a wide variety of platforms, including supercomputers and networks of workstations (NOWs). PVM uses a variety of underlying communications methods: shared memory on multiprocessors, various native message-passing systems on massively parallel processors (MPPs), and UDP/IP or TCP/IP on NOWs. The large software overhead in the IP stacks makes it difficult to provide high-performance communications for

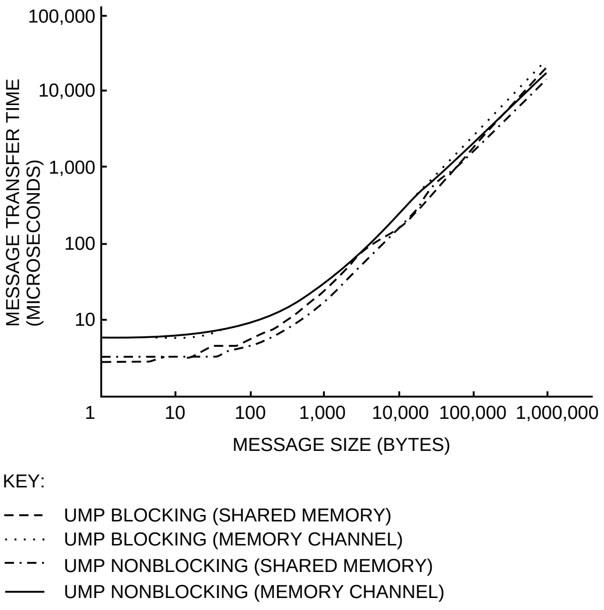


Figure 8
UMP Communications Performance: Message Transfer Time

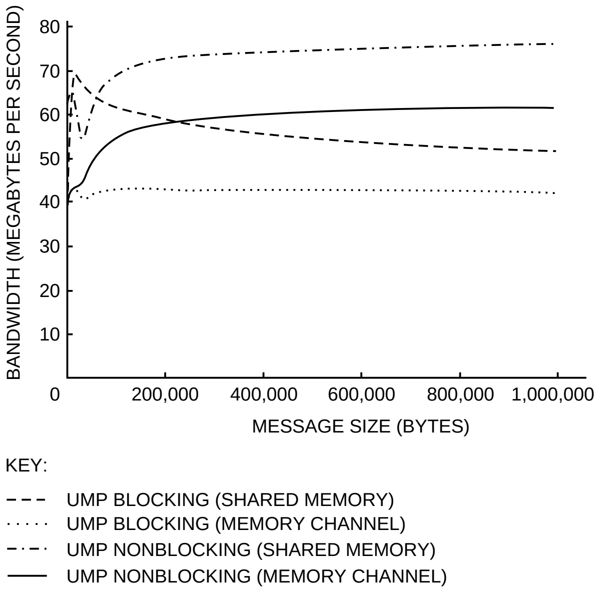


Figure 9
UMP Communications Performance: Bandwidth

PVM when using networks like Ethernet or FDDI. The high cost of communications for these systems means that only the more coarse-grained parallel applications have demonstrated performance improvements as a result of parallelization using PVM. Using the MEMORY CHANNEL cluster technology described earlier, we have implemented an optimized PVM that offers low latency and high-bandwidth communications. The PVM library and daemon use UMP to provide seamless communications over the MEMORY CHANNEL cluster.

When we began to develop PVM for MEMORY CHANNEL clusters, we had one overriding goal: to use the hardware performance the MEMORY CHANNEL interconnect offers to provide a PVM with industry-leading communications performance, specifically with regard to latency. Initially, we set a target latency for PVM of less than 15 μs using shared memory and less than 30 μs using the MEMORY CHANNEL transport.

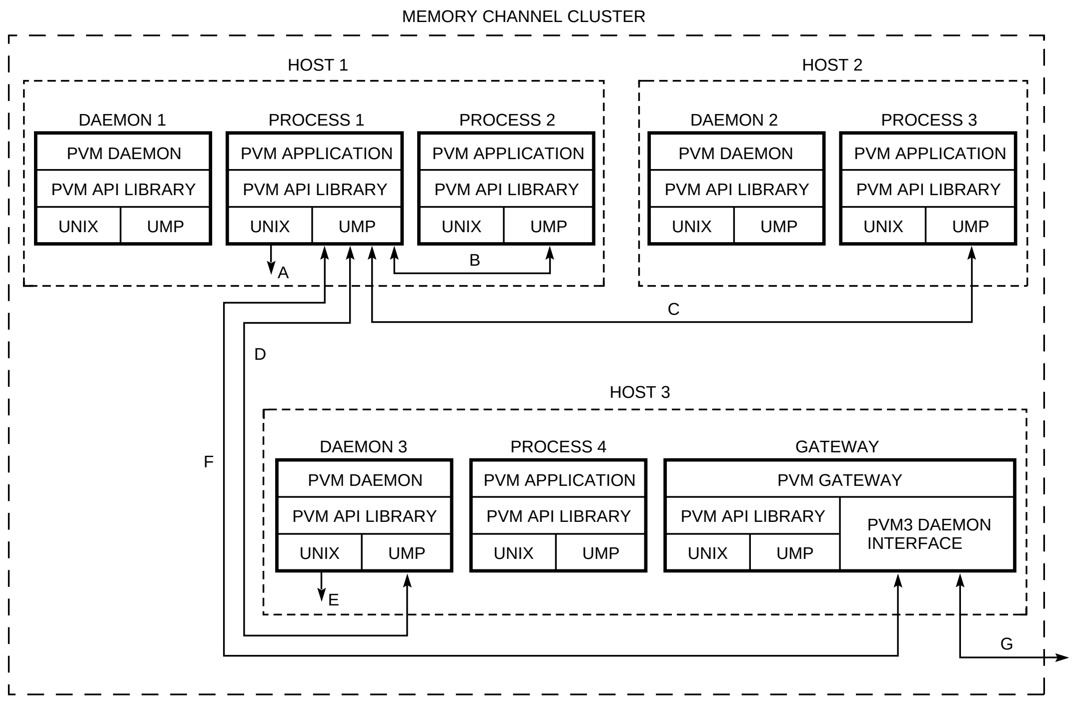
Our first task was to build a prototype using the public-domain PVM implementation. We used an early prototype of the MEMORY CHANNEL system jointly developed by Digital and Encore. The prototype had a hardware latency of 4 μs. We modified the shared-memory version of PVM to use the prototype hardware and achieved a PVM latency of 60 μs. Profiling and straightforward code analysis revealed that most of the overhead was caused by

- PVM's support for heterogeneity (i.e., external data representation [XDR] encoding)
- Messages being copied multiple times inside PVM
- A large number of function calls in the critical communications path
- Inefficient coding of the low-level data copy routines

Since we wanted to achieve the maximum possible performance available from the hardware, we decided to reimplement the PVM library, eliminating support for heterogeneity from the communications functions of PVM and focusing on maximum performance inside a Digital cluster.[20] Heterogeneity would then be supported by using a special PVM gateway process.

The overall architecture of the Digital PVM implementation is shown in Figure 10. To maximize performance, we decided that, wherever possible, an operation should be executed in-line rather than be requested from a remote task or daemon. This contrasts with PVM's traditional approach of relaying such requests to the PVM daemon for service. For example, when a PVM task starts, often it first calls pvm_mytid to request a unique task identifier (TID). Previously, this would have involved sending a message to a PVM daemon, which would then allocate a TID to the process and return another message. In our design, we could use global data structures in MEMORY CHANNEL space (e.g., the list of all PVM tasks and associated data). Now, for example, pvm_mytid simply involves acquiring a cluster lock on a global table, getting the new TID, and releasing the lock—all executed in-line by the calling process rather than a daemon. Executing PVM services in-line with the requesting process increases multiprocessing capability and eliminates daemon bottlenecks and associated delays.

We reimplemented the PVM library with the emphasis on performance rather than heterogeneity, although we plan to eventually allow interoperation with heterogeneous implementations of PVM using a special



KEY:

A A PVM application on host 1 performs local control functions using UNIX signals.
B A PVM application on host 1 communicates with another PVM task on the same host using UMP (via shared memory).
C A PVM application on host 1 communicates with another PVM task on a different host in the cluster (host 2) using UMP (via MEMORY CHANNEL).
D A PVM application on host 1 requires a control function (e.g., a signal) to be executed on another host in the cluster (host 3); it sends a request to a PVM daemon on host 3.
E The PVM daemon on host 3 executes the control function.
F A PVM application on host 1 sends a message to a PVM task on a host outside the MEMORY CHANNEL cluster; the message is routed to the PVM gateway task on host 3.
G The PVM gateway translates the cluster message into a form compatible with the external PVM implementation and forwards the message to the external task via IP sockets.

Figure 10
Digital PVM Architecture

gateway daemon. The PVM API library is a complete rewrite of the standard PVM version 3.3 API, with which full functional compatibility is maintained. Emphasis has been placed on optimizing the performance of the most frequently used code paths. In addition, all data structures and data transfers have been optimized for the Alpha architecture. As stated earlier, the amount of message passing between tasks and the local daemon has been minimized by performing most operations in-line and communicating with the daemon only when absolutely necessary. Intermediate buffers are used for copying data between the user buffers. This is necessary because of the semantics of PVM, which allow operations on buffer contents before and after a message has been sent. The one exception to this is pvm_psend; in this case, data is copied directly since the user is not allowed to modify the send buffer.

The purpose of our PVM daemon is different from that of the daemon in the standard PVM package. Our daemon is designed to relay commands between different nodes in the PVM cluster. It exists solely to perform remote execution of those commands that cannot be performed in-line by UNIX calls in the PVM API library or by directly manipulating global data structures. Commands to be executed on a remote node are sent to the daemon on that node, which then executes the command directly. Note that this removes a level of indirection that exists in standard PVM. Daemon-to-daemon communications are minimized. Since there is no master daemon, the PVM cluster has no single point of failure. All daemons are equal. When not in use, the daemon sleeps, being awakened as required by a signal from the calling task. For a local task, UNIX signals are used. If the task is on another node in the cluster, then MEMORY CHANNEL cluster signals are used. As a result, the daemon uses minimal cluster resources.

The PVM group or collective functions operate on a group of PVM tasks. For example: pvm_barrier synchronizes multiple PVM processes; pvm_bcast sends a message to all members of a particular group; pvm_scatter distributes an array to the members of a group; pvm_gather reassembles the array from the contributions of each of the group members, etc. The group functions are implemented separately from the other PVM messaging functions. They use a separate global structure (the group table) to manage PVM group data. Access to the group table is controlled by locks. Unlike other PVM implementations, there is no PVM group server, since all group operations can manipulate the group table directly.

Performance

Table 4 compares the communications latency achieved by various PVM implementations. As the table indicates, the latency between two machines with Digital PVM over a MEMORY CHANNEL transport is much less than the latency of the public-domain PVM implementation over shared memory, which validates our approach of removing support for heterogeneity from the critical performance paths. Figure 11 shows the message transfer time and Figure 12 shows the bandwidth for Digital PVM over shared memory and MEMORY CHANNEL transports for various message sizes. Two AlphaServer 4100 5/300 machines were used for these measurements. The peak bandwidth reached by Digital PVM is about 66 MB/s (shared memory) and 43 MB/s (MEMORY CHANNEL). By comparison, PVM 3.3.10 achieves a bandwidth of 24 MB/s (shared memory) and 3 MB/s (FDDI LAN). A version of PVM developed at Digital's Systems Research Center (SRC) using a specially adapted asynchronous transfer mode (ATM) driver achieved a latency of approximately 60 μs and a bandwidth of approximately 16 MB/s using the AN2 ATM LAN.[21] The performance results for PVM latency over the MEMORY CHANNEL transport given in Reference 6 were obtained using an earlier version of Digital PVM. Since those results were measured, latency has been halved, mostly due to improvements in UMP performance.

Figure 13 compares the performance of an unmodified PVM application using the public-domain PVM 3.3.7 implementation and Digital PVM version 1.0. The application is a parallel molecular modeling program. The bar chart shows the elapsed time for a variety of configurations. The application ran for 220 seconds on 2 two-processor SMP machines connected with FDDI. By replacing FDDI with a MEMORY CHANNEL network and PVM 3.3.7 with Digital PVM, we were able to speed up performance by a factor of approximately 3.4 for the same number of processors: the run time dropped from 220 seconds to 65 seconds. For comparison, we also ran the program on a four-processor SMP; the application completed in 64.5 seconds. This time was just marginally faster than the MEMORY CHANNEL configuration for the same number of processors, demonstrating that Digital PVM scales well from SMP to the MEMORY CHANNEL cluster. Finally, 2 four-processor SMP machines connected in a two-node MEMORY CHANNEL cluster ran the program in 38 seconds, demonstrating a speedup of 1.7.

Message Passing Interface

Message Passing Interface (MPI) is a message-passing standard developed by a large group of industrial and academic users. The standard contains a substantial number of functions (more than 120) and offers the same wide range of facilities that many earlier message-passing APIs provided. In fact, many parallel applications can be written using only six of the functions, but a correct implementation must provide the complete set. Argonne National Laboratory (ANL) has produced a reference implementation called MPICH.[22] This is a robust, clean implementation of the complete MPI-1 function set. In addition, it has isolated transport-specific components behind an abstract device interface (ADI).[23] The abstract device implements the communications-related functions and is further layered on what is called the channel device. The public domain version comes with channel implementations for a number of interconnects including shared memory, TCP/IP, and other proprietary interfaces. This version also includes a template for building a channel device, called the channel interface.[24] To build a channel device, the programmer must supply five functions:

1. Indicate if a control message is available on a control channel
2. Get a control message from a control channel
3. Send a control message to a control channel

Table 4
PVM Latency Comparison

PVM Implementation	Transport	Platform	Latency
PVM 3.3.9	Sockets FDDI	DEC 3000/800	400 μs
PVM 3.3.9	Shared Memory	AlphaServer 2100 4/233	60 μs
Digital PVM V1.0	MEMORY CHANNEL 1.0	AlphaServer 2100 4/233	11 μs
Digital PVM V1.0	MEMORY CHANNEL 1.5	AlphaServer 4100 5/300	8 μs
Digital PVM V1.0	Shared Memory	AlphaServer 2100 4/233	5 μs
Digital PVM V1.0	Shared Memory	AlphaServer 4100 5/300	4 μs
Digital PVM V1.0	Shared Memory	AlphaServer 8400 5/350	3 μs

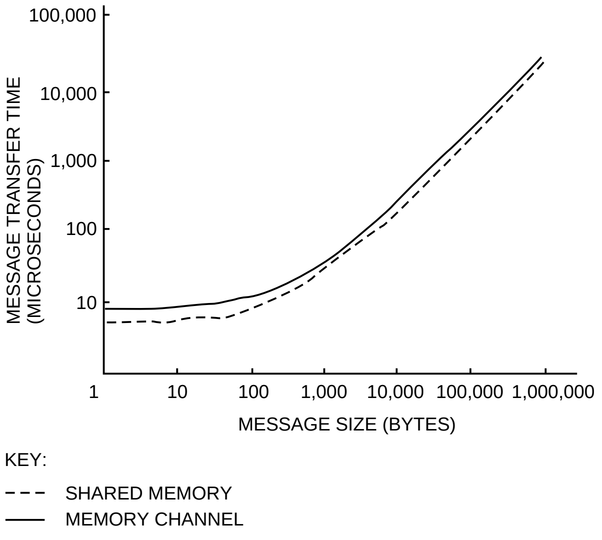


Figure 11
Digital PVM Communications Performance: Message Transfer Time

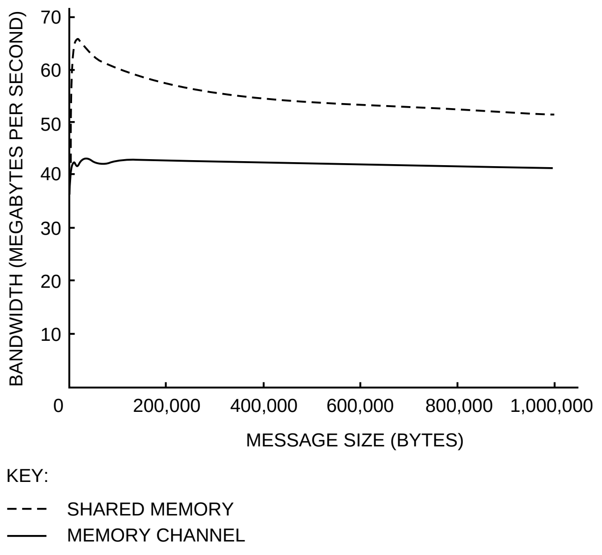


Figure 12
Digital PVM Communications Performance: Bandwidth

4. Receive data from a data channel
5. Send data to a data channel

These functions can all be implemented using the UMP functions ump_read, ump_write, and ump_wait described earlier. In addition, hooks are added to the channel initialization and shutdown code to call ump_init and ump_exit. This approach leaves the portable MPICH API library unchanged and attempts to deliver optimum performance. MPICH implements all its operations, point-to-point and collective, on the basic point-to-point services that the ADI provides.

Working with the Edinburgh Parallel Computing Centre (EPCC), we produced an early functional MPI prototype by building a channel device on UMP, as

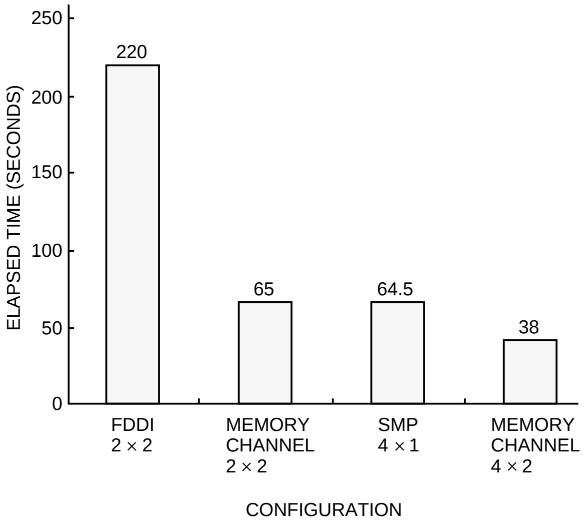


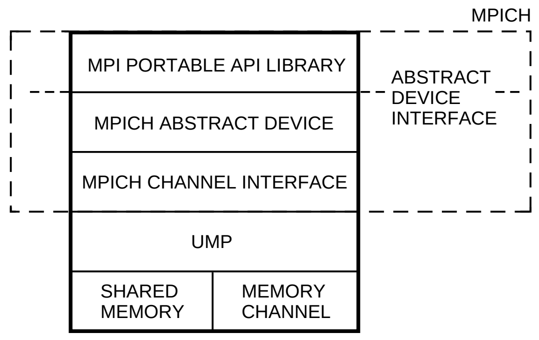
Figure 13
PVM Application Performance

shown in Figure 14a. This implementation demonstrated latencies of 12.5 μs (shared memory) and 29 μs (MEMORY CHANNEL), respectable performance for such a quick port of MPI for clusters. Furthermore, since this implementation uses UMP, it works transparently on shared memory and MEMORY CHANNEL. ADI channels typically support only one interconnect; multiple ADIs are not yet supported by MPICH. Unlike PVM, the semantics of MPI allow operation without an intermediate buffer, so that UMP buffers can be used directly.

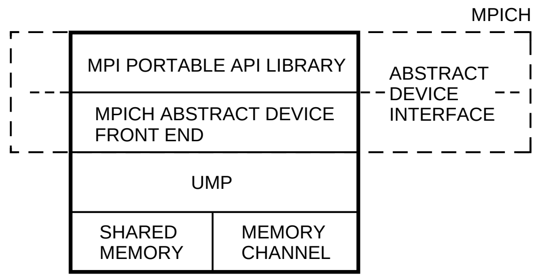
To further improve the performance of MPI on clusters, we eliminated the MPICH channel device and interfaced UMP directly to the ADI, as shown in Figure 14b. The abstract device incurs some performance penalty in its support for the channel device. In the UMP implementation, this is unnecessary as UMP already performs the function of hiding details of the transport mechanism. This implementation demonstrated latencies of 9.5 μs (shared memory) and 16 μs (MEMORY CHANNEL), using an Alpha cluster consisting of two AlphaServer 2100 4/233 machines connected by a MEMORY CHANNEL network.

Performance

Table 5 compares the communications latency achieved by MPICH and the Digital MPI implementation, using an Alpha cluster. Results are shown for both AlphaServer 2100 4/190 and AlphaServer 4100 5/300 machines connected by a MEMORY CHANNEL network. Figure 15 shows the message transfer time and Figure 16 shows the bandwidth of Digital MPI over shared memory and MEMORY CHANNEL transports for a variety of message sizes. A pair of AlphaServer 4100 5/300 machines were used for these measurements. Digital MPI reaches a peak bandwidth of about 64 MB/s using shared memory and 61 MB/s



(a) Initial Prototype



(b) Version 1.0 Implementation

Figure 14
Digital MPI Architecture

using MEMORY CHANNEL. By comparison, the unmodified MPICH achieves a peak bandwidth of 24 MB/s using shared memory and 5.5 MB/s using TCP/IP over an FDDI LAN.

Figure 17 shows the speedup demonstrated by an MPI application. The application is the Accelerated Strategic Computing Initiative (ASCI) benchmark SPPM, which solves a three-dimensional gas dynamics problem on a uniform Cartesian mesh.[25,26] The same code was run using both Digital MPI and MPICH using TCP/IP. The hardware configuration was a two-node MEMORY CHANNEL cluster of AlphaServer 8400 5/350 machines, each with six CPUs. Digital MPI used shared memory and MEMORY CHANNEL transports, whereas MPICH used the Ethernet LAN connecting the machines. The maximum speedup obtained using Digital MPI was approximately 7, whereas for MPICH the maximum speedup was approximately 1.6.

Future Work

We intend to continue refining the components described in this paper. The major change envisioned regarding the TruCluster MEMORY CHANNEL Software product is the addition of user-space spinlocks, which should significantly reduce the cost of acquiring a spinlock. We intend to increase the performance of UMP by making more efficient use of MEMORY CHANNEL in a number of ways: striping large messages over multiple adapters, supporting next-generation adapters, and using point-to-point mappings with a MEMORY CHANNEL switch. In addition, we plan to add outbufs to increase multicast message-passing performance. PVM enhancements planned include the addition of the gateway daemon to allow interoperation with other PVM implementations on external platforms. PVM will also be modified to use the UMP nonblocking write facility for arbitrarily large messages so that its performance matches that of MPI. Since the semantics of PVM force the use of an intermediate buffer, performance when using shared memory will be improved by passing pointers to a lock-controlled buffer for messages whose transfer time would exceed the overhead associated with a lock. We will continue to improve MPI performance by optimizing the UMP ADI for the MPICH implementation.

Summary

We have built a high-performance communications infrastructure for scientific applications that utilizes a new network technology to bypass the software overhead that limits the applicability of traditional networks. The performance of this system has been proven to be on a par with that of current supercomputer technology and has been achieved using commodity technology developed for Digital's commercial cluster products. The paper demonstrates the suitability of the MEMORY CHANNEL technology as a communications medium for scalable system development.

Table 5
MPI Latency Comparison

MPI Implementation	Transport	Platform	Latency
MPICH 1.0.10	Sockets FDDI	DEC 3000/800	350 μs
MPICH 1.0.10	Shared Memory	AlphaServer 2100 4/233	30 μs
Digital MPI V1.0	MEMORY CHANNEL 1.0	AlphaServer 2100 4/233	16 μs
Digital MPI V1.0	MEMORY CHANNEL 1.5	AlphaServer 4100 5/300	6.9 μs
Digital MPI V1.0	Shared Memory	AlphaServer 2100 4/233	9.5 μs
Digital MPI V1.0	Shared Memory	AlphaServer 4100 5/300	5.2 μs

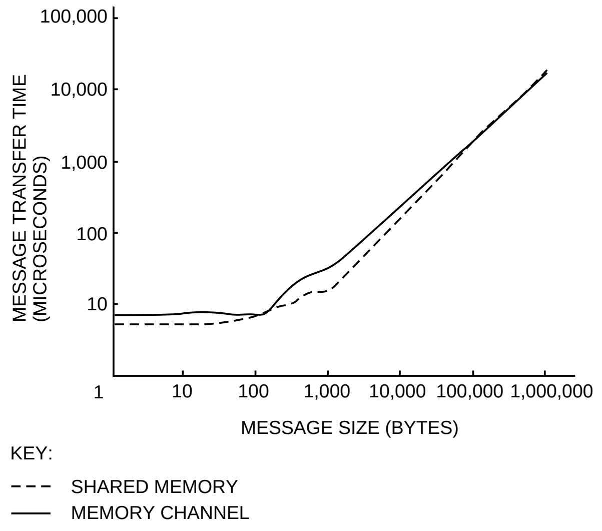


Figure 15
MPI Communications Performance: Message Transfer Time

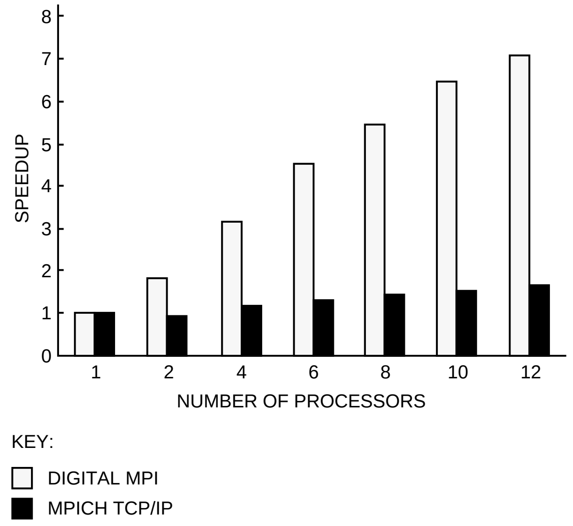


Figure 17
MPI Application Speedup

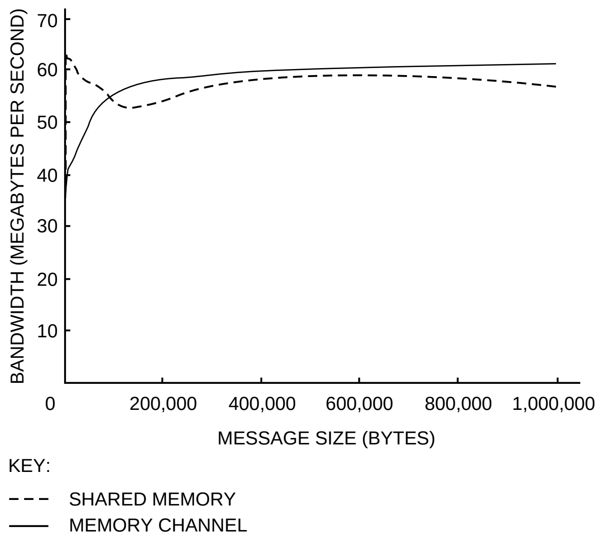


Figure 16
MPI Communications Performance: Bandwidth

Acknowledgments

The authors would like to acknowledge the following people for their contributions to this project: Gavan Duffy, whose testing made the TruCluster MEMORY CHANNEL Software a much more robust product; Liam Kelleher and Garret Taylor, who contributed some of the Digital PVM functionality; Wayne Cardoza and Brian Stevens of UNIX Engineering, who provided early access to and ongoing support of kernel MEMORY CHANNEL software; Rick Gillett and Mike Collins, who provided early MEMORY CHANNEL hardware; Richard Kaufmann, who gave us encouragement and support; and Lyndon Clarke and Kenneth Cameron at Edinburgh Parallel Computing Centre (EPCC), who modified MPICH to use UMP for Digital MPI.

References and Note

1. T. Anderson, D. Culler, and D. Patterson, "A Case for NOW (Network of Workstations)," *Proceedings of the Hot Interconnects II Symposium,* Palo Alto, Calif. (August 1994).

2. K. Keeton, T. Anderson, and D. Patterson, "LogP Quantified: The Case for Low-Overhead Local Area Networks," *Proceedings of the Hot Interconnects III Symposium,* Palo Alto, Calif. (August 1995).

3. R. Sites, ed., *Alpha Architecture Reference Manual* (Burlington, Mass.: Digital Press, Order No. EY-L520E-DP, 1992).

4. N. Kronenberg, H. Levy, and W. Strecker, "VAXclusters: A Closely Coupled Distributed System," *ACM Transactions on Computer Systems,* vol. 4, no. 2 (May 1986): 130–146.

5. W. Cardoza, F. Glover, and W. Snaman, Jr., "Design of the TruCluster Multicomputer System for the Digital UNIX Environment," *Digital Technical Journal,* vol. 8, no. 1 (1996): 5–17.

6. R. Gillett, "MEMORY CHANNEL Network for PCI: An Optimized Cluster Interconnect," *IEEE Micro* (February 1996):12–18.

7. M. Blumrich et al., "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," *Proceedings of the Twenty-first Annual International Symposium on Computer Architecture* (April 1994): 142–153.

8. M. Blumrich et al., "Two Virtual Memory Mapped Network Interface Designs," *Proceedings of the Hot Interconnects II Symposium,* Palo Alto, Calif. (August 1994): 134–142.

9. L. Iftode et al., "Improving Release-Consistent Shared Virtual Memory using Automatic Update," *Proceedings of the Second International Symposium on High-Performance Computer Architecture* (February 1996).

10. C. Dubnicki et al., "Software Support for Virtual Memory-Mapped Communication," *Proceedings of the Tenth International Parallel Processing Symposium* (April 1996).

11. High Performance Fortran Forum, "High Performance Fortran Language Specification," Version 1.0, *Scientific Programming,* vol. 2, no. 1 (1993).

12. A. Geist et al., *PVM 3 User's Guide and Reference Manual,* ORNL/TM-12187 (Oak Ridge, Tenn.: Oak Ridge National Laboratory, May 1994). Also available on-line at http://www.netlib.org/pvm3/ug.ps.

13. A. Geist et al., *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing* (Cambridge, Mass.: The MIT Press, 1994). Also available on-line at http://www.netlib.org/pvm3/book/pvm-book.html.

14. MPI Forum, "MPI: A Message Passing Interface Standard," *International Journal of Supercomputer Applications,* vol. 8, no. 3/4 (1994). Version 1.1 of this document is available on-line at http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html.

15. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface* (Cambridge, Mass.: The MIT Press, 1994).

16. J. Harris et al., "Compiling High Performance Fortran for Distributed-memory Systems," *Digital Technical Journal,* vol. 7, no. 3 (1995): 5–23.

17. E. Benson et al., "Design of Digital's Parallel Software Environment," *Digital Technical Journal*, vol. 7, no. 3, (1995): 24–38.

18. In the first implementations, the PCI MEMORY CHANNEL network adapter places a limit of 128 MB on the amount of MEMORY CHANNEL space that can be allocated.

19. *TruCluster MEMORY CHANNEL Software Programmer's Manual* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QTN4A-TE, 1996).

20. J. Brosnan, J. Lawton, and T. Reddin, "A High-Performance PVM for Alpha Clusters," *Proceedings of the Second European PVM Users' Group Meeting,* Lyons, France (September 1995).

21. M. Hausner, M. Burrows, and C. Thekkath, "Efficient Implementation of PVM on the AN2 ATM Network," *Proceedings of High-Performance Computing and Networking* (May 1995).

22. W. Gropp and N. Doss, "MPICH Model MPI Implementation Reference Manual," Draft Technical Report (Argonne, Ill.: Argonne National Laboratory, June 1995).

23. W. Gropp and E. Lusk, "MPICH ADI Implementation Reference Manual," Draft Technical Report (Argonne, Ill.: Argonne National Laboratory, October 1994).

24. W. Gropp and E. Lusk, "MPICH Working Note: Creating a New MPICH Device using the Channel Interface," Draft Technical Report (Argonne, Ill.: Argonne National Laboratory, June 1995).

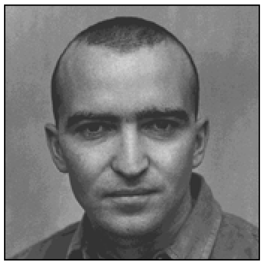
25. Accelerated Strategic Computing Initiative (ASCI), RFP Statement of Work C6939RFP6-3X, Los Alamos National Laboratory (LANL) (February 12, 1996). This document is also available on-line at http://www.llnl.gov/asci_rfp/asci-sow.html.

26. The ASCI SPPM Benchmark Code is available from Lawrence Livermore National Laboratory at http://www.llnl.gov/asci_benchmarks/asci/limited /ppm/asci_sppm.html.

Biographies

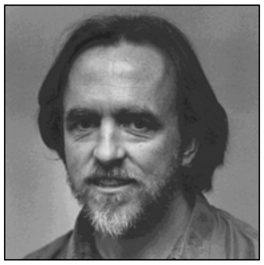
James V. Lawton
Jim Lawton joined Digital in 1986 and is a principal engineer in the Technical Computing Group. In his current position, he contributed to the design of Digital PVM and the UMP library and was responsible for implementing UMP and adding support for collective operations to Digital PVM. Before that, he worked on the characterization and optimization of customer scientific/technical benchmark codes and on various hardware and software design projects. Prior to coming to Digital, Jim contributed to the design of analog and digital motion control systems and sensors at the Inland Motor Division of Kollmorgen Corporation. Jim received a B.E. in electrical engineering (1982) and an M.Eng.Sc. (1985) from University College Cork, Ireland, where he wrote his thesis on the design of an electronic control system for variable reluctance motors. In addition to receiving the Hewlett-Packard (Ireland) Award for Innovation (1982), Jim holds one patent and has published several papers. He is a member of IEEE and ACM.

John J. Brosnan
John Brosnan is currently a principal engineer in the Technical Computing Group where he is project leader for Digital PVM. In prior positions at Digital, he was project leader for the High Performance Fortran test suite and a significant contributor to a variety of publishing technology products. John joined Digital after receiving his B.Eng. in electronic engineering in 1986 from the University of Limerick, Ireland. He received his M.Eng. in computer systems in 1994, also from the University of Limerick.



Morgan P. Doyle
In 1994, Morgan Doyle came to Digital to work on the High Performance Fortran test suite. Presently, he is an engineer in the Technical Computing Group. Early on, he contributed significantly to the design and development of the TruCluster MEMORY CHANNEL Software, and he is now responsible for its development. Morgan received his B.A.I. and B.A. in electronic engineering (1991) and his M.Sc. (1993) from Trinity College Dublin, Ireland.

Seosamh D. Ó Riordáin
Seosamh Ó Riordáin is an engineer in the Technical Computing Group where he is currently working on Digital MPI and on enhancements to the UMP library. Previously, he contributed to the design and implementation of the TruCluster MEMORY CHANNEL Software. Seosamh joined Digital after receiving his B.Sc. (1991) and M.Sc. (1993) in computer science from the University of Limerick, Ireland.



Timothy G. Reddin
A principal engineer in the Technical Computing Group, Timothy Reddin currently leads the team responsible for the TruCluster MEMORY CHANNEL Software, the UMP library, Digital PVM, and Digital MPI. Prior to coming to Digital in 1994, Tim worked for eight years as a systems designer at ICL High Performance Systems in the United Kingdom. He was responsible for the I/O architecture of the ICL Goldrush parallel database server, for which he holds two patents, and the design of an I/O and communications controller. Tim also worked at Raytheon on the data communications subsystem for the NEXRAD distributed real-time Doppler weather radar subsystem. Prior to that, he developed the software architecture for an integrated executive workstation while working at CPT Limited. After receiving his B.Sc. (with distinction, 1976) in computer science and mathematics from University College Dublin, Ireland, Tim joined the staff of University College Cork, where he was a systems programmer. Tim is a member of the British Computer Society and is a Chartered Engineer.