Design and Evaluation of Nemesis: a Scalable, Low-Latency, Message-Passing Communication Subsystem

Darius Buntinas, Guillaume Mercier, William Gropp

To cite this version:

Darius Buntinas, Guillaume Mercier, William Gropp. Design and Evaluation of Nemesis: a Scalable, Low-Latency, Message-Passing Communication Subsystem. Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06), May 2006, Singapour, Singapore. pp.521-530, 10.1109/CCGRID.2006.31 . hal-00344350

HAL Id: hal-00344350
https://hal.archives-ouvertes.fr/hal-00344350

Submitted on 4 Dec 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem *

Darius Buntinas Guillaume Mercier William Gropp

Mathematics and Computer Science Division
Argonne National Laboratory
email: {buntinas, mercierg, gropp}@mcs.anl.gov

Abstract

This paper presents a new low-level communication subsystem called Nemesis. Nemesis has been designed and implemented to be scalable and efficient both in the intranode communication context using shared-memory and in the internode communication case using high-performance networks and is natively multimethod-enabled. Nemesis has been integrated in MPICH2 as a CH3 channel and delivers better performance than other dedicated communication channels in MPICH2. Furthermore, the resulting MPICH2 architecture outperforms other MPI implementations in point-to-point benchmarks.

1 Introduction

Computationally intensive applications, such as simulations and modeling, require a large amount of computing power that can solely be delivered by parallel architectures. Efficiently programming parallel machines remains a difficult challenge because of the variety of available hardware: Massively parallel processors, SMPs, ccNUMA machines, and clusters all fall into this category. The design and development of dedicated environments, tools, and libraries are therefore the cornerstone of the successful exploitation of this kind of architecture. These technologies should be portable and yet be able to narrow the gap between the hardware's capabilities and the actual performance delivered to the application. The Message Passing Interface standard has been defined to address these portability and performance issues. The performance level, however, remains largely implementation-dependent. Hence, implementing the MPI specification in a fast and efficient fashion remains the focus of attention of many research groups. Moreover, other key characteristics besides performance must be taken into account in order to reflect the hardware trends. Scalability, communication hierarchy (for instance, intranode vs internode communication, high-performance networks vs regular networks) as well as the development of multinetwork environments should also be considered and raise new issues. Indeed, multirail systems or architectures featuring several different networks such as the IBM BlueGene/L bring new challenges for MPI developers. Open source MPI projects under development include YAMPII [14], Open MPI [7], and MPICH2 [10]. In particular, the goal of MPICH2 is to provide an efficient MPI-2 implementation for massively parallel processors, SMPs, and clusters (small- and large-scale alike).

A careful analysis of several MPICH2 communication channels underscores the fact that the performance level could be dramatically enhanced. We therefore designed and implemented a new communication subsystem, called Nemesis, the goal of which is to address these performance issues while being scalable and natively supporting multimethod communication. This communication subsystem can be a stand-alone piece of software, but it is a relevant target to be the basis of higher-level programming tools. As such, it has been successfully integrated within MPICH2 as an CH3 channel and yields very good performance.

Section 2 explains the design of our new communication subsystem. Section 3 describes some implementation issues we addressed that are crucial for achieving good performance. Section 4 presents our performance evaluation of Nemesis. We show comparisons between the MPICH2 channel implementation of Nemesis and other MPICH2 channels and other MPI implementations. Section 5 concludes this paper and discusses future work.

2 Design of the Nemesis Communication Subsystem

We designed Nemesis to be a high-performance communication subsystem for MPICH2 [10]. Our primary design goals, in order of priority, were scalability, high-performance intranode communication, high-performance internode communication, and multimethod internode communication. We ranked the design goals in order of priority to help us resolve conflicts between these goals as they arise. In this section we describe our design by first giving a general overview, then giving more detail on the basic queue mechanism and network modules.

Our design started with a shared-memory queue for scalable and efficient intranode message passing. To this we added network modules that interface with the queue mechanism, providing a unified method for sending and receiving messages. We designed the network modules around the queue mechanism, rather than the other way around, in order to get the best performance for intranode communication. This design also allows us to add multiple network modules to support multimethod internode communication.

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

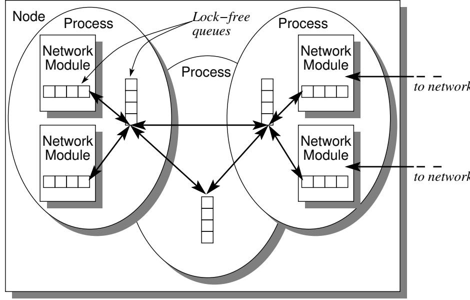

Figure 1. Block diagram of Nemesis design showing one node with three processes.

Figure 1 shows the basic design. The figure depicts three processes in a node. The processes communicate using lock-free queues. The network modules are used to communicate with processes on other nodes. Messages received by the network modules are queued on the process's lock-free queue. In this way the process only has to poll one queue to receive messages from any process on any node.

We address two shared-memory models. A small-scale shared-memory model, such as an SMP workstation running Linux, is characterized by each node having a relatively small number of processors and a software environment in which the processes create and attach to shared-memory regions in order to be able to communicate using shared memory. A large-scale shared-memory model, such as a ccNUMA machine, is characterized by each node having a relatively large number of processors and a software environment in which the processes access the memory of other processes without the memory having to be in a special shared-memory region or having to attach to the memory region first.

2.1 Intranode Communication

Intranode communication in Nemesis is performed by using lock-free queues in shared memory. Our design goal here was for each process to have a single receive queue that it needs to poll, rather than having a separate receive queue to poll for each remote process. Having only one queue to poll makes the design very scalable.

Our design comprises three variations. Each process has a *receive queue*; and, depending on the variation, there is either one *free queue* per process or one global *free queue*. Figure 2 shows how a messages is sent by one process and received by another. The figure shows the design variation where free queues are located at sending processes. The sending process (1) dequeues a queue element from the free

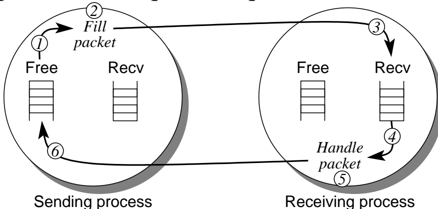

Figure 2. Block diagram of send and receive using shared-memory queues with free queues located at sender.

```
Enqueue (queue, element) {
  prev = SWAP (queue->tail, element);
  if (prev == NULL)
    queue->head = element;
  else
    prev->next = element;
}
Dequeue (queue, &element) {
  element = queue->head;
  if (element->next != NULL)
    queue->head = element->next;
  else {
    queue->head = NULL;
    old = CAS (queue->tail, element, NULL);
    if (old != element) {
      while (element->next == NULL)
        SKIP;
      queue->head = element->next;
} } }
```

Figure 3. Lock-free algorithm, using atomic swap (`SWAP`) and compare-and-swap (`CAS`) operations.

queue, (2) fills the queue element with the message, and (3) enqueues it on the receiving process's receive queue. The receiving process then (4) dequeues the element from its receive queue, (5) handles the message, and (6) enqueues the queue element back onto the same free queue where it had originally been dequeued.

In the small-scale memory model, the receive and free queues and their elements all have to reside in shared-memory regions accessible by all of the processes. This makes adding queues and elements dynamically difficult, so the queues and elements have to be allocated at initialization time. For the large-scale memory model, however, processes need not specifically attach to a region of memory to access it, queues and elements can easily be added dynamically.

2.1.1 Design Variations

The three variations of the design change the location of the free queue. In the first variation, there is one global free queue from which every process will dequeue elements. In the second variation, each process has a free queue that other processes will dequeue from when sending to it. In the third variation, each process has a free queue that it will dequeue from when sending messages to other processes. For a more detailed discussion of the advantages and disadvantages of the variations, see [5].

2.1.2 Implementation

Our current implementation of Nemesis uses the third design variation. This design variation uses the same type of lock-free queues for both the receive queue and the free queue, namely, a lock-free queue that allows multiple enqueuers and a single dequeuer concurrently.

The lock-free queue algorithm we use is similar to the MCS lock [8] using swap and compare-and-swap atomic operations. The pseudo-code for the algorithm is given in Fig. 3. The queue has a head and a tail. To enqueue an element, the process atomically swaps the pointer to the element with the value of the tail of the queue. If the previous value of the tail was NULL, indicating that the queue was empty, the process sets the head to point to the element also. If the queue was not empty, the process sets the *next* pointer of the previous element to point to the new element.

To check whether the queue is empty, a process can simply check whether the head of the queue is NULL. To de-

queue an element, the process gets a pointer to the element at the head of the queue. If that element has a non-NULL *next* pointer, the process sets the head of the queue to point to that next element, and the dequeue operation is complete. If the *next* pointer is NULL, however, then we have to deal with a potential race condition. It's possible that another process has started enqueuing an element and has performed the swap of the queue tail but has not yet set the *next* pointer of the element we are dequeuing. In this case the dequeuing process sets the queue head to NULL, then performs a compare-and-swap on the queue tail pointer, swapping it with NULL only if the queue tail is still pointing to the element that is being dequeued. If the compare-and-swap succeeds in swapping the queue tail pointer, then no other process is trying to enqueue, and the dequeue operation is complete. If it does not succeed, another process is performing an enqueue operation, so the dequeuing process waits, polling on the *next* pointer, until the enqueuing process finishes enqueuing and sets the *next* pointer of the element being dequeued. The dequeuing process then completes the dequeue operation by setting the queue head to point to the newly enqueued element. The implementation of the lock-free queue algorithm proved to be very efficient: only six instructions are required for an enqueue operation, and only 11 instructions are required for a dequeue operation.

We currently have an implementation of Nemesis on a Linux ia32 platform. In this implementation, one process on each node allocates a shared-memory region that holds the free and receive queues of all the processes. The other processes map this shared-memory region into their address space. Each process keeps an array of pointers to the receive and free queues of all the other processes. Because the region might not be mapped into the same address on all the processes, the pointers into this region are implemented as offsets rather than absolute pointers. Using offsets means that accessing a pointer requires performing an extra translation, but the impact on the overall performance is negligible.

2.2 Internode Communication

To the shared-memory intranode communication mechanism described in the previous section, we add internode communication using network modules. The interface between a process and its network module is the same as between two processes on the same node: Each network module has a *send* queue, which is analogous to a process's receive queue, and a free queue. Messages to be sent over the network are queued on the network module's send queue using elements from the process's free queue. The network module dequeues messages from its send queue and transmits them over the network. Messages received from the network are queued on the process's receive queue using elements from the network module's free queue.

2.2.1 Implementation

In the same way that a process keeps an array of pointers to the receive and free queues of the other processes on the same node, it also has entries in this array for processes on remote nodes, except that the receive and free queue pointers for a remote node point to the send and free queues of the network module. The network module determines which remote process to send the message to by examining the header of the message. In this way, the code for a process to send intra- and internode messages is the same, making the critical path very efficient: The process dequeues and fills a free queue element, dereferences the pointer to the receive queue for the target process, then enqueues it on that queue. The only difference is that in the internode case, when the process gets a pointer to the "receive" queue, it is a pointer to the network module's *send* queue. Similarly, because the network module enqueues received messages on the process's receive queue, receiving messages from processes on remote nodes is the same as receiving messages from processes on the same node.

The network's free queue entries must be allocated in the shared-memory region, even though the entries will be used only within the same process. The reason is that when the network module queues an entry on the process's receive queue, if another process tries to enqueue an element after it, that process will need to be able to set the *next* pointer of the network module's element.

In order for the network module to make progress sending messages on its send queue and receiving messages into the process's receive queue, the process periodically calls the network module's poll function.

When using user-level OS-bypass networks that require registration of send and receive buffers, memory copies can be avoided in the network module when sending messages by registering all of the process's free queue entries. Hence, when the process enqueues a message on the network's send queue, the network module can transmit the message directly from the queue entry, rather than copying it to a preregistered send buffer. Similarly, memory copies can be avoided when a message is received, by registering all of the network module's free queue entries, then dequeuing the entries and posting them as receive buffers with the network. Hence, incoming messages are received directly into the queue entries, which can then be queued on the process's receive queue. As queue entries are returned to the network module's free queue, they are again posted as receive buffers with the network.

For user-level networks that require specially registered send and receive buffers but cannot use this method, for example if the network doesn't support the registration of shared memory, memory copies can still be avoided when receiving a message. An incoming message is received into one of the network module's receive buffers; then the queue entry that is queued on the process's receive queue includes a pointer to the receive buffer containing the message, rather than the message itself. Receive buffers are reused when the process returns the queue element referencing the buffer to the network's free queue. Note that this method requires the process to take special action when sending through a network module to a process on a different node, adding more complexity to the critical path.

Avoiding a copy on the send side may be more difficult. If the data is contiguous, then the send queue entry would contain a pointer to the data along with the message header. The network module would then copy the header and the data to the registered send buffer. Rather than performing the memory copy from the user buffer into the queue entry, the data is copied from the user buffer into the network module's registered send buffer. However, the data to be sent may be arranged as a complex noncontiguous datatype. Two main issues must be addressed in handling this case. First, adding the ability to handle datatypes to the network module would increase its complexity, and second, the description of the datatype may be larger than the size of the

queue entry. Our current implementation would simply perform an additional copy in this case. We intend to examine ways of optimizing this in the future.

2.2.2 Multiple Network Support

Supporting multiple network modules allows one network, for instance GM, to be used for intracluster communication and another network, for instance TCP, to be used for intercluster communication. Because of the modularity of our design, additional network modules can be easily added. All that is needed to support multiple network modules is to have the process call the poll function for each module and to set the pointers appropriately in the process's arrays of receive and free queue pointers.

Networks that require registration of send and receive buffers most likely will not allow different networks to register the same page, so only one network will be able to register all of the queue elements. We refer to this network as the *primary* network. The other networks may have to use the alternative mechanism described above, using pointers to receive buffers and performing an additional copy on the send. These are the *secondary* networks. Note that not all secondary networks need to perform extra copies, only those that require registration of send and receive buffers. TCP, for example, can send and receive directly to and from queue and so does not need to perform extra copies.

When the networks are arranged hierarchically, where one network is being used for intracluster communication and the other for intercluster communication, generally the intracluster network will be the primary network and the intercluster network the secondary network. The intercluster network will most likely have higher latencies because it is a lower-performance network or because of the distance to the other cluster, and so will most likely not benefit as much as the primary network will from the ability to reduce copies.

2.3 Remote Memory Access

Nemesis also supports remote memory access (RMA) operations. An RMA operation allows one process, the initiator process, to access variables stored in the address space of another process, the target process. RMA operations are often called one-sided operations because, the initiator process supplies all the arguments of the operation. Nemesis currently supports *Get* and *Put* RMA operations on contiguous and noncontiguous data. Noncontiguous data is described by using an array of address and length pairs.

2.3.1 Design

There is one interface for performing RMA operations using shared memory and another using networks. The shared-memory interface is based on the concept of a *window*. A window is a region of a process's memory in which other processes can perform RMA operations. When a process wants to allocate memory that may be used for shared-memory RMA, it allocates a window. Before another process can perform an RMA operation on the window, the window description must be sent to that process, which then *attaches* to that window.

For RMA operations performed over networks, any memory to be used in a RMA operation must be *registered*. The registration operation returns a *key* for the registered region. This key is sent to any process that will perform RMA operations on that memory. The initiator process then uses the key as a parameter to any RMA operations on that memory region.

The window-based RMA operations are blocking operations: When the function returns the operation has completed. The network-based RMA operations are nonblocking. The RMA operations are passed a pointer to a completion counter, which is set to a value greater than zero when the operation is initiated. When the operation is complete, Nemesis will set this value to zero. The application is required to call a Nemesis communication function or the poll function in order to guarantee that the nonblocking RMA operations complete. Depending on the network and the implementation of the network module, the target process may also have to call Nemesis communication functions or poll in order to guarantee that the operation completes.

2.3.2 Implementation

In our implementation on a Linux ia32 platform, when a process allocates a window, a shared-memory file is created and mapped into the process address space. The window description contains the name of this file as well as the address of this region in the process's address space. When another process attaches to this window, it maps the corresponding file into its address space. Because the initiator process may not map the region at the same address as the target process, the address of an object in the window may not be the same at the initiator as at the target. For this reason, the RMA interface specifies that addressees passed to RMA functions should be addresses in the target's address space. In fact the initiator does not need to know where the window has been mapped into its address space. The RMA operation at the initiator process will translate the target pointer into the address in the initiator's address space before performing the operation.

Some networks, such as InfiniBand [1], require memory keys from the target process to be passed to the initiator process, while others, such as GM [11], do not. The Nemesis memory registration key can be used by the network module to pass such information. If the network does not require the use of keys, then the key description that is sent from the target to the initiator can be empty. The RMA operations are implemented by using the network's native RMA operations.

The Nemesis registration function will register the memory only with the primary network. For secondary networks that require memory registration and networks that do not support RMA operations, the RMA operations are performed by using the message queue mechanism.

3 Optimizing Nemesis

We applied several optimizations to the basic design in order to improve performance. We first focus on intranode communication and then describe internode communication mechanisms that we implemented in Nemesis.

3.1 Reducing L2 Cache Misses

L2 cache misses are unavoidable when processes on different processors are accessing the same memory locations. Because Nemesis is designed around the shared-memory message queues, and L2 cache misses are very costly (over 400 cycles, or 200 ns on a dual 2 GHz Xeon node), it is critical to eliminate unnecessary L2 cache misses when accessing the head and tail pointers of the queues.

A process must access both the head and tail of the queue when it is enqueuing an element on an empty queue or when

it is dequeuing an element that is the last element in the queue. In these cases, if the head and tail were in the same cache line, only one L2 cache miss would be encountered. If the queue has more elements in it, however, then the enqueuer only needs to access the tail, and the dequeuer only needs to access the head. If the head and tail were in the same cache line, then there would be L2 misses encountered as a result of false sharing each time a process enqueues an element after another has been dequeued from the same queue, and vice versa. In this case it would be better if the head and tail were in separate cache lines.

Our solution is to put the head and tail in the same cache line and have a *shadow* head pointer in a separate cache line. The shadow head is initialized to NULL. The dequeuer uses the shadow head in place of the real head except when the shadow head is NULL, meaning that the queue has become empty. If the shadow head is NULL when the dequeuer tries to dequeue, it checks the value of the real head. If the real head is not NULL, meaning that an element has been enqueued on the queue since the last time the queue became empty, the dequeuer initializes its shadow head to the value of the real head and sets the real head to NULL. In this way, only one L2 cache miss is encountered when enqueuing onto an empty queue or dequeuing from a queue with one element. And because the tail and shadow head are in separate cache lines, there are no L2 cache misses from false sharing.

We found that using a shadow head pointer reduced one-way latency by about 200 ns on a dual 2 GHz Xeon node.

3.2 Bypassing Queues for Intranode Communication

Latency can be further improved by bypassing the queue mechanism using *fastboxes*. A fastbox is simply a single buffer with a full/empty flag. If the fastbox is empty, a process will put the message into the fastbox and set the flag to *full*, rather than use the queue. The receiver will first check whether the fastbox is full before checking the queue. After the receiver processes the message, it sets the flag to *empty*. There is one fastbox per pair of processes on a node.

While this approach isn't a scalable for large shared-memory machines, it can be used for clusters of small SMP machines, which make up many modern clusters. Even for large shared-memory machines, fastboxes can be used only between nearest neighbors or can be created dynamically between processes depending on the communication pattern.

One problem introduced with fastboxes is that while there is only one queue to poll on, adding fastboxes means that the receiver has to check multiple fastboxes. We address this problem by noting that in message-passing systems, the application is required to specify the sender when requesting to receive a message. Nemesis provides a function to specify which processes to expect messages from. The middleware library can use this function whenever the application posts a receive. Nemesis will then poll those fastboxes. To handle *wildcard* receives, where an application will receive a message from any process, Nemesis will poll every fastbox, but less frequently, to minimize the impact of the additional polling on receiving other messages.

Another issue is message ordering. There are now two paths between a sender and a receiver: the queue and the fastbox. A sender may have enqueued a message on the queue and sent the next message in the fastbox. Before using the fastbox, the sender has no way of knowing whether the receiver has received the message in the queue. Similarly, before checking the fastbox, the receiver has no way of knowing whether there are any messages somewhere in the queue from that particular receiver. If the receiver checks the fastbox before it checks the queue, it will receive the messages in the wrong order. This situation is handled in Nemesis by using sequence numbers. When the receiver checks the fastbox, if the sequence number of the message in the fastbox is not the expected one, it knows there is at least one message sent before this one on the queue, and it leaves the message in the fastbox. On the other hand, when the receiver checks the queue and the sequence number of the message is not the expected one, it knows that the expected message is in the fastbox, and it checks there.

We found that when fastboxes are used, the one-way latency is reduced by about 500 ns on our testbed.

3.3 Improving Memory Copies

The standard libc `memcpy()` routine did not perform optimally in our tests. In [4], we analyzed several memory copy functions for the Linux ia32 architecture and found that using the assembly string copy functions was optimal for messages up to 2 KB and using a memory copy using MMX instructions was optimal for larger messages. The MMX copy function increased the bandwidth by around 400 MBps on our testbed.

3.4 Bypassing the Message Queues for Internode communication

We bypass the queues on the send side in order to save on instructions and latency. Instead of enqueuing a message in the receive queue and then calling the progress routine that has to dequeue it to send it, if the network module's send queue is empty, we directly send the message. Our original design goal was to simplify the critical path for shared-memory and avoid having to check whether a messages was an intra- or internode message. However, we found that the added cost of performing this check on intranode messages was negligible, while the benefit to internode messages was about 500 ns using the GM network module and several microseconds when using TCP.

3.5 Optimizing the TCP Network Module

In the TCP module, we receive a Nemesis packet by first receiving the Nemesis header, from which we can determine the size of the rest of the packet; then we receive the rest of the packet. This approach allows us to receive a single packet directly into a free queue element. The downside is that two `read` system calls are required to receive each packet. We optimize by receiving the header, plus some small amount of the data in the first call to `read`. Then we check the header for the packet length, and only if we haven't received the entire packet do we call `read` to receive the rest of the packet. This way, small packets can be received by using one `read` call. If the packet was smaller than the amount we initially received, we have received part of the next packet. In this case, we copy the portion of the next packet to another free queue element.

When implementing the Nemesis channel for MPICH2, we set the additional amount received to be the MPICH2 header length (currently 32 bytes), plus an additional amount of the payload (currently 16 bytes). We note that nonblocking operation are used on both the sending and the receiving sides: We don't switch back and forth between

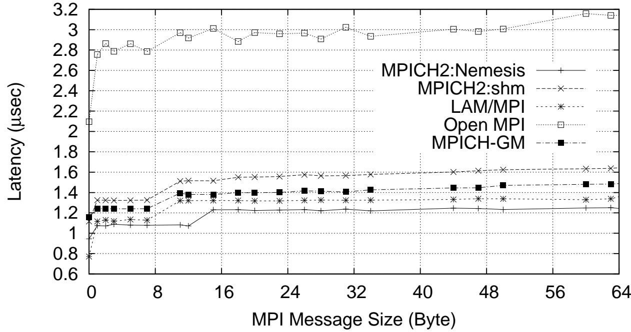

Figure 4. MPI Intranode Communication Latency

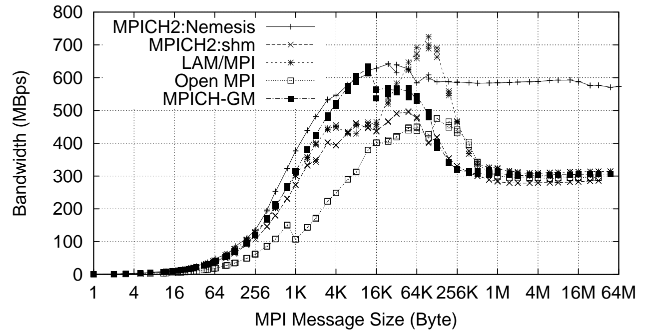

Figure 5. MPI Intranode Communication Bandwidth

blocking and nonblocking modes as in LAM/MPI for instance.

4 Performance Evaluation of Nemesis

In this section, we present preliminary results obtained with the Nemesis communication subsystem. To compare Nemesis to other communication subsystems, we implemented an MPICH2 CH3 channel using Nemesis. We then compared the performance of the Nemesis channel to other MPICH2 channels at the MPI-level. The other channels we compared were the *shm* channel for shared-memory (intranode communication), the *sock* channel for TCP, and the *GASNet* channel for Myrinet. We also compared the resulting new MPICH2 software stack with other MPI implementations in order to demonstrate the relevance of our approach. We measured latency and bandwidth for both intra- and internode communication. In the latter case, we evaluated the Nemesis performance on two high-performance networks: Myrinet, with the GM [11] interface, and gigabit Ethernet, with the TCP protocol. The other MPI implementations considered were Open MPI v1.0 [7], MPICH-GM [9], LAM/MPI [6, 13], and YAMPII [14]. MPICH2 was compiled with the `-enable-fast` option that disables the error checking in MPICH2 code making it faster. All tests were performed by using dual-SMP 2 GHz Xeon nodes with 4 GB of memory. The operating system was Linux with a 2.6.10 kernel. For the interconnects, we used a Myrinet 2000 [2] "PCI64C" NIC connected to a 32-port switch using the GM [11] message-passing system, version 2.0.21. The NIC was installed in a 64-bit 66 MHz PCI slot. We also used an Intel 82544GC Gigabit Ethernet Controller installed in the same type of PCI slot. In this paper we consider one megabyte as 1024×1024 bytes.

4.1 Intranode Communication Performance

When an MPI application is executing, the fastest communication is expected to happen between processes belonging to the same physical node, since shared-memory performance is still quite a bit higher than that of current high-performance networks. Achieving the fastest intranode communication is therefore crucial. For comparisons with other MPICH2 CH3 channels, we chose to test Nemesis against the *shared-memory* (shm) channel, which is the fastest device for intranode communication in the current MPICH2 release. We also compared our software with other MPI implementations: LAM/MPI, Open MPI, and MPICH-GM. MPICH-GM was included in this test because its intranode communication mechanisms are fast and indeed relevant for our study. YAMPII was excluded because it doesn't support fast intranode communication: in the current release, it is handled through the loop-back capabilities of the Ethernet NIC. We used Netpipe [12] to perform the latency and bandwidth tests.

4.1.1 Latency and Bandwidth

Figure 4 presents the MPI latency for messages smaller than 64 bytes. The first comparison that we make is between the shm and Nemesis channels and shows that Nemesis is roughly 25% faster than shm. Comparisons with other MPI implementations are also favorable to us: aside from the 0 byte message case, where LAM/MPI is the most efficient, MPICH2 using Nemesis offers the best latency. The next closest implementation to MPICH2:Nemesis is LAM/MPI, followed by MPICH-GM and then Open MPI. We note that Open MPI's latency is twice as high compared to that of the three other solutions.

Bandwidth curves are shown in Fig. 5. For messages smaller than 48 KB, MPICH2:Nemesis offers the best bandwidth and even outperforms the efficient mechanisms implemented within MPICH-GM. LAM/MPI yields better results than MPICH2:Nemesis for messages between 48 and 192 KB, but this result can't be sustained for larger messages because it is caused by the amount of L2 cache available on the node. However, this performance gap between LAM/MPI and MPICH2:Nemesis is limited and not very large, about 15% at most. The situation is quite different for very large messages (i.e., more than 192 KB) because the bandwidth of MPICH2:Nemesis is almost twice that of other MPI implementations. We managed to sustain such a result that is influenced by the optimized memcopy routine employed (see Section 3.3) along with careful tuning of the MPICH2:Nemesis software stack.

4.1.2 Instruction Count

To assess performance accurately, we include the instruction count for intranode communication. We present only the comparison between the Nemesis and shm MPICH2 channels so we can make fair comparisons and show the improvement allowed by our new communication subsystem, since the upper layers are similar in both cases. Our test is restricted to a simple case: the number of instructions needed to perform a blocking send and a blocking receive on the `MPI_COMM_WORLD` communicator. In this case, the message is 8 bytes long (a double on a 32-bit machine). To measure these counts, we used the PAPI [3] software library

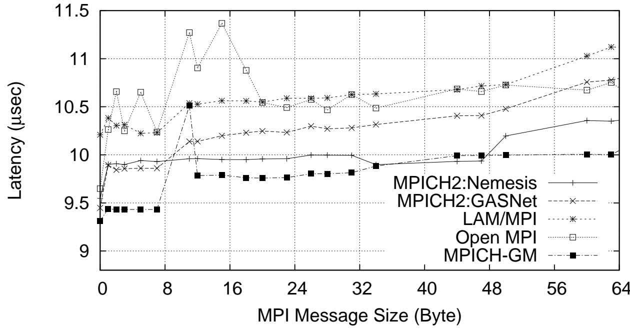

Figure 6. MPI Internode Communication Latency (Myrinet/GM)

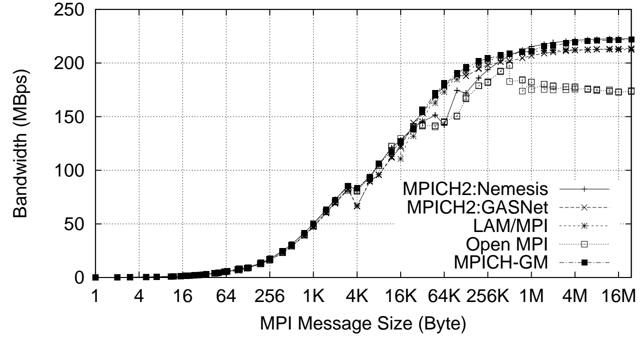

Figure 7. MPI Internode Communication Bandwidth (Myrinet/GM)

that offers a convenient interface to gather such results. Our test program consists of two processes exchanging the double for 10,000 iterations. Before a call to `MPI_Recv` is issued, a call to the `usleep` routine is performed in order to make sure that the message is actually available. By doing so, we eliminate the instructions that are used to perform polling, which is variable.

The following table shows the results obtained:

Instruction count	Shm	Nemesis	Improvement
`MPI_Send`	446	278	37%
`MPI_Recv`	1233	815	34%
Total	1679	1093	35%

This data confirms that sending and receiving messages with MPI are two very different operations performance-wise. Comparison between the shm and the Nemesis channels shows that we reduced the number of instructions by roughly 35%. We also note that sending an 8-byte message with MPICH2:Nemesis is fast, taking less than 300 instructions.

4.2 Internode Communication Performance

In this section, we evaluate communication involving network transfers. Even though we carefully designed the Nemesis communication subsystem to be highly efficient in the intranode case, the internode communication has also received considerable attention. Hence we can present a new solution tailored to a wide range of architectures. We have developed two network modules for Nemesis: the TCP module and the GM/Myrinet module (our Myrinet hardware does not support the MX software). The results of both modules are presented in the following sections. We plan to develop other modules, to take advantage of newer interfaces or new hardware, for instance InfiniBand or Quadrics.

4.2.1 Performance over Myrinet

The first high-performance network we tested was Myrinet. New networking technologies might have emerged in the past few years, but Myrinet still remains a popular interconnect for cluster building. Figures 6 and 7 show the comparison between Nemesis and GASNet channels, as well as LAM/MPI, Open MPI, and MPICH-GM. The best latency is offered by MPICH-GM (except for 10-byte messages, where a peak happens that we cannot explain), followed by MPICH2 using the Nemesis channel and then the GASNet channel. We note that the GASNet channel's performance for messages smaller than 8 bytes is slightly better than that of Nemesis. Open MPI's latency is larger than MPICH2:GASNet but eventually outperforms this channel for messages larger than 56 bytes. LAM/MPI's latency is better than Open MPI's for small non-null messages and then lags a little bit behind for messages larger than 18 bytes.

As for bandwidth, the Nemesis channel performs well for messages smaller than 24 KB and offers the best bandwidth, but all implementations show similar results. For messages between 24 and 512 KB, both Open MPI and MPICH:Nemesis yield a lower bandwidth than the other software tested. However, MPICH2:Nemesis performs slightly better than Open MPI, and the drop in performance indicates the threshold size that is used to switch between eager and rendezvous protocol (that is, 64 KB). The other MPI implementations (LAM/MPI, MPICH2:GASNet, and MPICH-GM) have similar results. For messages larger than 512 KB, MPICH2:Nemesis has higher bandwidth than any other software, including MPICH-GM. We can actually see three groups for performance: MPICH2:Nemesis and MPICH-GM which perform better than MPICH2:GASNet and LAM/MPI, and Open MPI, which doesn't manage to sustain its performance for messages larger than 512 KB (after this size, the bandwidth drops and stabilizes around 180 MBps).

4.2.2 Performance over Gigabit Ethernet

The last performance comparison was performed over gigabit Ethernet. Figure 8 shows the latencies for both the Nemesis and sock channels for MPICH2, as well as the performance of LAM/MPI, Open MPI, and YAMPII. We can see that the best results are achieved by MPICH2:Nemesis and LAM/MPI, which feature latencies that are very close. They are followed by MPICH2 using the sock channel and YAMPII. The Open MPI results are surprising because there is a large gap with the other software. Indeed, Open MPI offers a latency that is 50% higher than LAM/MPI or MPICH2:Nemesis.

In Fig. 9 we see that the Nemesis channel delivers the best bandwidth, along with LAM/MPI and MPICH2:sock for messages smaller than 4 KB. YAMPII and Open MPI are a little bit behind. As in the Myrinet case, we can see that medium-sized messages (i.e., messages between 4 and 128 KB) is where MPICH2:Nemesis yields its worst performance: while still on par with other implementations, the Nemesis channel is less efficient than other software. For large messages, however, the situation is different. MPICH2:Nemesis and Open MPI both offer similar

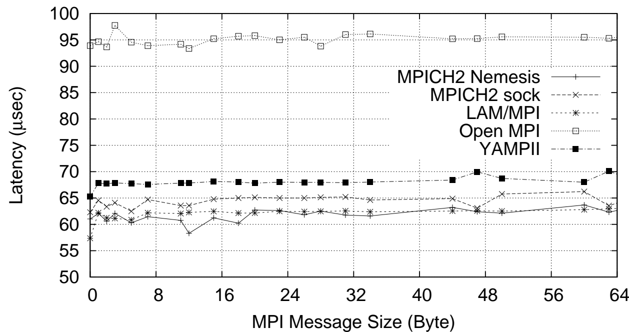

Figure 8. MPI Internode Communication Latency (Gigabit Ethernet/TCP)

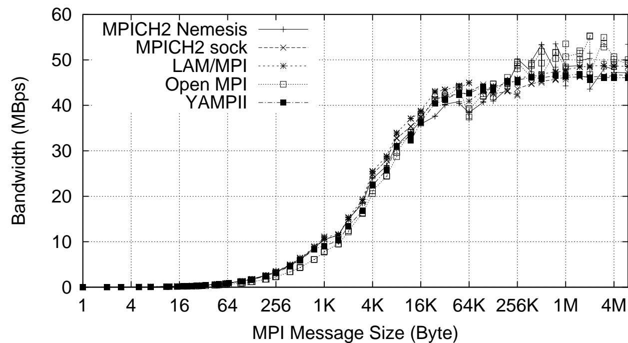

Figure 9. MPI Internode Communication Bandwidth (Gigabit Ethernet/TCP)

(and irregular) bandwidth results. They are followed closely by LAM/MPI, YAMPII, and MPICH2:sock, but these three implementations show more regular results.

5 Conclusion and Future Work

In this paper, we presented Nemesis, a new message-passing communication subsystem that is designed to be low-latency and scalable. We presented its original design, based on a novel use of scalable and efficient lock-free queues, that have enqueue and dequeue costs of 6 and 11 instructions, respectively. This communication subsystem has been integrated into MPICH2 as an CH3 channel. Our performance evaluation shows that Nemesis effectively delivers low latency for small messages and high bandwidth for large messages, both in intranode and internode communication cases. We performed comparisons between MPICH2 implemented using Nemesis and other MPI implementations, and found that MPICH2 based on Nemesis has better intranode latency for non-zero messages than all other MPI implementations, and better intranode bandwidth for messages larger than 256 KB than all other MPI implementations. For internode performance, MPICH2 based on Nemesis performs comparably to, if not better than, the best of the other MPI implementations over GM and TCP.

Since we worked only at the channel level and left the ADI3 layer untouched, there is still room for improvement, and we are currently working on a device version of Nemesis with a new progress engine. Indeed, the results are promising because the instruction count for performing an `MPI_Recv` operation is down to 300 instructions (roughly 60% less instructions compared to the count of Nemesis as a channel), and we have reduced the latency by more than 20% (we achieve submicrosecond latency for messages up to 64 bytes for instance). Nemesis has been designed as a scalable channel in order to meet the requirements of large-scale clusters, and shared-memory machines. Fault tolerance is a central issue in this context, and we have started to develop and integrate checkpointing and restart mechanisms within Nemesis. We also plan to perform more significant tests with real computing kernels, such as HPL or NAS.

We would like to assess the performance of our RMA interface and evaluate the performance improvement at the MPI level when exploiting this part of the Nemesis interface. We also would like to explore Nemesis as a communication subsystem for environments other than MPI. GAS languages could be an interesting and relevant target for such work.

Acknowledgments

We thank Ewing Lusk, Robert Ross and Rajeev Thakur for their invaluable help and comments.

References

[1] InfiniBand Trade Association. InfiniBand architecture specification, volume 1, release 1.0. http://www.infinibandta.com.

[2] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. Seizovic, and W. Su. Myrinet - A gigabit per second local area network. In *IEEE Micro*, pages 29–36, February 1995.

[3] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference, Monterey, California*, 1999.

[4] Darius Buntinas, Guillaume Mercier, and William Gropp. Data transfers between processes in an SMP system: Performance study and application to MPI. Technical Report ANL/MCS-P1306-1105, Argonne National Laboratory, 2005. Submitted to International Parallel and Distributed Processing Symposium (IPDPS) 2006.

[5] Darius Buntinas, Guillaume Mercier, and William Gropp. The design and evaluation of Nemesis, a scalable low-latency message-passing communication subsystem. Technical Report ANL/MCS-TM-292, Argonne National Laboratory, 2005.

[6] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[7] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[8] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memeory multiprocessors. *ACM Computer Architecture News*, 9(1):21–65, Feb 1991.

[9] MPICH-GM. http://www.myri.com/scs/.

[10] MPICH2. http://www.mcs.anl.gov/mpi/

[11] Myricom. Myricom GM Myrinet software and documentation. http://www.myri.com/scs/GM/doc/gm_toc.html, 2000.

[12] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. Netpipe: A network protocol independent performace evaluator. In *Proceedings of Internation Conference on Intelligent InformationManagement and Systems*, 1996.

[13] Jeffrey M. Squyres and Andrew Lumsdaine. A component architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[14] YAMPII. http://www.il.is.s.u-tokyo.ac.jp/yampii/.

The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract No. W-31-109-ENG-38 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.