Cluster Computing 6, 115–124, 2003
© 2003 Kluwer Academic Publishers. Manufactured in The Netherlands.

One-Sided Communication on Clusters with Myrinet

JAREK NIEPLOCHA * and EDOARDO APRA
Pacific Northwest National Laboratory

JIALIN JU **
Tivoli Systems Inc.

VINOD TIPPARAJU
Ohio State University

Abstract. This paper describes an efficient implementation of one-sided communication on top of the GM low-level message-passing library for clusters with Myrinet. This approach is compatible with shared memory, exploits pipelining, nonblocking communication, and overlapping memory registration with memory copy to maximize the transfer rate. The paper addresses critical design issues faced on the commodity clusters and then describes possible solutions for matching the low-level network protocol with user-level interfaces. The performance implications of the design decisions are presented and discussed in context of a standalone communication benchmark as well as two applications. Finally, the paper offers some indications on what additional features would be desirable in a communication library like GM to better support one-sided communication.

Keywords: one-sided communication, remote memory operations, clusters, Myrinet

1. Introduction

In recent years, the Myrinet network has received much attention in the literature on high-performance communication. Multiple projects have focused on developing efficient messaging middleware (e.g., AM [1], PM [2], BIP [3], and HPVM FM [4]) by exploiting the programmable Network Interface Card (NIC) of Myrinet. The most common use of these interfaces has been support of MPI and sometimes other, internal research projects. The HPVM project used its FM system to implement MPI and other programming interfaces including two one-sided: the Cray SHMEM and Global Arrays. In the last few years, due to its good scalability and relatively moderate cost, Myrinet has become the primary network for building medium and large-scale clusters based on commodity processing nodes (e.g., Intel or Alpha Linux systems). To our best knowledge, with the exception of the NCSA Windows NT SuperCluster that operates in the HPVM environment, the majority of medium and large Myrinet-based clusters used in production mode rely on the GM message-passing library and the MPICH, with the MPICH GM channel provided by Myricom. GM is the Myrinet alternative to the low-level messaging middleware systems mentioned above [5]. With exception of an AM implementation on top of GM, these other interfaces replace the standard Myrinet Control Program (MCP) on the NIC thus are not compatible at run-time and cannot be installed with GM. GM has been designed to provide high bandwidth and very low latency message passing in the polling mode. In addition to message passing, GM also supports a one-sided put operation (gm_directed_send) between registered (pinned) local and remote memory areas.

In this work, we are interested in developing efficient one-sided communication on commodity clusters that use the Myrinet network and the standard Myricom software environment, which includes GM and MPICH/GM. One-sided communication supports the global-address-space programming model that for some applications is more appropriate than the MPI-1 two-sided message-passing communication. This model combines some advantages of shared memory, such as direct access to shared/global data, and the message-passing model, namely the control over locality and data distribution. One of the primary goals for the current work is to develop, distribute, and support a communication library compatible with the standard environment of the Myrinet clusters employed for production purposes. This consideration introduces some restrictions and offers certain advantages for the potential users. For example, interoperability with MPICH/GM is assumed and required. For practical reasons and the ease-of-use by ordinary non-expert users, our implementation attempts to minimize consumption of the GM ports in order avoid rebuilding and reinstalling Myrinet software to increase the number of available ports. We also consider the SMP environment and interaction of shared memory with the communication subsystem. This work is performed in context of a portable one-sided communication interface, ARMCI [6], developed as a portable run-time system for distributed array libraries and compilers. The primary operations in ARMCI include put, get, accumulate, and mutual ex-

* Corresponding author.
E-mail: jarek.nieplocha@pnl.gov
** The author did this work while at Pacific Northwest National Laboratory (PNNL).

Table 1
Remote operations description.

Operation	Description
ARMCI_Put, ARMCI_PutV, ARMCI_PutS	contiguous, vector and strided versions of put
ARMCI_Get, ARMCI_GetV, ARMCI_GetS	contiguous, vector and strided versions of get
ARMCI_Acc, ARMCI_AccV, ARMCI_AccS	contiguous, vector and strided versions of atomic accumulate
ARMCI_Fence	blocks until outstanding operations targeting specified process complete
ARMCI_AllFence	blocks until all outstanding operations issued by calling process complete
ARMCI_Rmw	atomic read-modify-write
ARMCI_Malloc	memory al locator, returns array of addresses for memory allocated by all processes
ARMCI_Free	frees memory allocated by ARMCI_Malloc
ARMCI_Lock, ARMCI_Unlock	mutex operations

clusion operations. The library is available on a variety of platforms including Massively Parallel Processors (MPP) and clusters.

The paper makes several contributions to the field. First it shows how to build an efficient implementation of one-sided communication on top of GM, a two-sided low-level message-passing library on Myrinet. It addresses critical design issues faced on the commodity SMP clusters and then describes possible solutions for matching the low-level network protocol and user-level programming model requirements. The performance implications of the design decisions are presented and analyzed in context of a standalone communication benchmark as well as two applications. Finally, it offers some indications on what additional features would be desirable in a communication library like GM to better support one-sided communication.

The paper is organized as follows. Section 2 describes ARMCI interface and model. Section 3 discusses the technical aspects of implementing one-sided communication based on GM. Experimental results are provided and discussed in section 4, and we present conclusions in section 5.

2. ARMCI

ARMCI was developed to support remote memory operations (see table 1) in the context of distributed array libraries and compiler run-time systems [6]. It was also used to implement other communication libraries such as a portable version of the Cray SHMEM library [7]. The ARMCI interface is portable and compatible with MPI (and on some platforms, it also was used with other message-passing libraries). ARMCI requires a message-passing library for the process startup and user environment initialization. The library is available on clusters of common Unix and Windows workstations/servers and MPP systems.

Compared to the well known Cray SHMEM one-sided interface [8], ARMCI places more focus on non-contiguous data transfers that correspond to data structures in scientific applications (e.g., sections of multi-dimensional dense or sparse arrays). Such transfers are optimized thanks to the non-contiguous data interfaces available in the ARMCI data transfer operations: multi-strided and generalized UNIX I/O vector interfaces. By default, ARMCI supports up to eight stride levels corresponding to eight-dimensional arrays. The library provides three classes of operations: (1) data transfer operations including put, get, and accumulate; (2) synchronization operations – local and global fence and atomic read-modify-write, mutex operations; and (3) operations for memory management and error handling.

ARMCI only supports communication that targets remote memory allocated via the provided memory allocator routine, ARMCI_Malloc, which is similar to MPI_Win_malloc in MPI-2. On shared memory systems including SMPs, this requirement allows the library to allocate shared memory for the user data and consecutively map one-sided operations to direct shared memory accesses, thus achieving sub-microsecond latency and a full memory bandwidth [9].

3. Technical approach

ARMCI progress rules, unlike one-sided "active" communication model in the MPI-2 or put/get operations in Generic Active Messages, declare its one-sided operation to be fully unilateral (i.e., they complete regardless of the actions taken by the remote process). In particular, polling (implicit when making a library call, or explicit by calling provided polling interface) in the application by remote process is not required for communication progress. Although polling is well suited for achieving very low-latency communication by avoiding the cost of interrupt processing, it only works well when remote processing is expecting communication and enters the polling loop before the message arrives [10]. Polling often leads to lower responsiveness and higher CPU consumption than the interrupt mode in context of real applications, but neither of these factors has any effect on latency results in the standard message-passing ping-pong benchmark that is often misused as a primary metric of the performance quality of a message-passing library implementation. For example, the polling implementation of MPICH on Myrinet offers several times better latency than the interrupt-based MPIPro on the Giganet cLAN, but it did not make the NAS benchmarks run any faster on the same Dell Linux cluster [11]. Similarly, on the IBM SP that offers polling and interrupt modes of operations in MPI, often the polling mode is used for running communication benchmarks while the interrupt mode, which pro-

duces a much higher latency [12], is more often used for running real applications. As we find explicit polling not compatible with the concept of one-sided communication, and for the reasons mentioned above, ARMCI does not offer or rely on a polling operation.

The default implementation of ARMCI for clusters of workstations with generic TCP/IP networking support exploits the standard sockets interface. An extra "data server" process (Unix) or thread (primarily on Windows) is forked on each cluster node to service one-sided communication requests from its clients (user MPI tasks). To prevent server thread/process in the absence of one-sided communication requests from consuming resources needed by the user processes, special care is needed to reduce CPU utilization by using blocking wait rather than active polling of the network interfaces. In the case of the TCP/IP protocol, "data server" exploits *select* call with a descriptor set that includes sockets dedicated to communication with every task on remote nodes. This system call blocks until data or a request is received on at least one of the sockets. The performance is consistent with the h/w and IP protocol limitations. For example, latency of the get operation on Linux/Intel varies from 160 μs (kernel 2.2.10smp) to 260 μs (kernel 2.2.14smp) on 100BaseT Ethernet, and is almost the same as the MPICH round-trip, message-passing latency for the same system configuration. The bandwidth on the 100BaseT Ethernet is 11.7 MB/s for both contiguous and strided operations. However, this level of performance does not meet requirements of many communication intensive applications, and often makes impossible for them to scale beyond a few nodes.

The TCP/IP implementation of ARMCI can be used on Myrinet as well, as this network, in addition to the GM interface, can support the standard IP protocols. The Myricom implementation of IP if enabled consumes one of the eight GM ports. There have been several implementations of the TCP/IP protocol on Myrinet in the last several years, most notably by the Trapeze project [13] for FreeBSD. Although, proof-of-concept implementations demonstrated that performance levels as good as 1 Gbit/s can be obtained, the production implementation distributed by Myricom for Linux is far from aproaching that level of performance. Apart from the nominal performance issues, the TCP/IP protocol characteristics [14] make it not the best choice for high-performance computing.

3.1. GM

GM is a low-level message-passing system for the Myrinet network [5]. The GM system includes a driver, the Myrinet-interface control program, a network mapping program, and the GM API, library, and header files. GM features include: (1) concurrent, protected, user-level access to the Myrinet interface; (2) reliable, ordered delivery of messages; (3) automatic mapping and route computation; (4) automatic recovery from transient network problems; (5) scalability to thousands of nodes; and (6) low host-CPU utilization. GM is a lightweight communication layer, and as such it has certain limitations including inability to send messages from or receive messages into non-DMA-able memory, and offers no support for gather or scatter operations. It is also unable to register shared memory under Linux. Moreover, memory registration operations in GM under Linux 2.2.X are very expensive relative for example to AIX. GM on Solaris does not even allow register local memory that was allocated as non-DMA-able in the first place. Some of these limitations can be addressed by layering a heavier-weight interface over GM, and they also have a profound impact on our system design.

3.2. Architecture of ARMCI over GM

We made the following design assumptions: (1) compatibility with standard unmodified version of MPICH/GM is required, (2) compatible with the System V shared memory is very desirable as it is required for effective SMP utilization [9], and (3) consumption of the GM ports by ARMCI should be minimized to avoid recompiling and reinstalling Myrinet s/w by ordinary non-expert users of the software.

The baseline GM port of ARMCI was derived from its TCP/IP implementation. As fork is not supported in the GM programs, an extra pthread is created on every cluster (uni- or multi-processor) node to play a role of "data server". The GM blocking receive operation used by that thread is well suited for developing an interrupt-driven implementation of one-sided communication. The GM driver through its interaction with the thread scheduler activates the blocked server thread upon arrival of the data. This thread at the initialization time opens two GM ports: one for receiving requests and one for sending a response. The flow control issues in GM make one port insufficient for that purpose as the server would not be able to respond to the current client request unless all other pending requests from other clients are received and stored in memory for further processing. In congested cases, depletion of memory on a node would be then possible. However, the number of GM ports used by ARMCI is independent of the number of MPI tasks on that node. We use the same GM ports allocated by MPICH to send ARMCI requests to a port opened by the remote "data server" thread. The port sharing technique works well since we do not use the GM receive operations to avoid handling GM messages sent to that port by the other protocol – MPI. Instead, a response from the server is delivered by directed send (put) that does not require GM receive operations (i.e., the data is directly placed in the client memory). In order to avoid deadlock, the GM tokens associated with send operations on the client side must be recovered by GM. That library assumes that tokens are relinquished when application calls `gm_unknown` operation while polling for arriving messages using the GM receive operations. However the ARMCI client code cannot use this approach for the reasons stated above. Instead, it relinquishes its send tokens indirectly by calling MPI_Iprobe while waiting for a response from the server. MPI_Iprobe checks the network interface for MPI messages received on the same port and, by doing so, unknowingly relinquishes tokens associated not only with MPI but also ARMCI messages. Although this

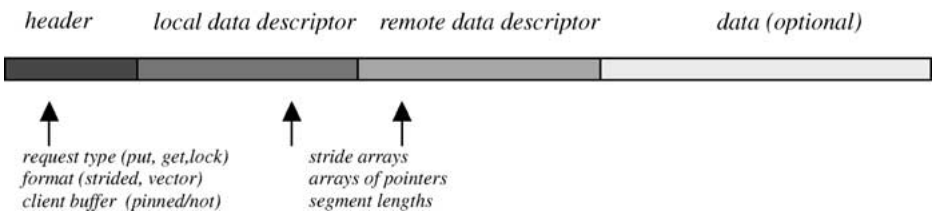


Figure 1. ARMCI request format for communication with server thread.

technique might first seem unsafe and dependent on an implementation of MPI, we believe that for progress and/or performance reasons any implementation of MPI_Iprobe would have to poll the network interface, thus effectively processing the GM tokens. We also found this trick to work reliably in practice.

The client communicates with the server by sending request messages in the format as shown in figure 1. The request messages contain one required component, a header describing the request, and depending on the request type, it might include a descriptor for the client buffer, a descriptor for remote memory that the request references, and data (e.g., in put operation). The optional descriptor for the client buffer has been introduced in the GM port to provide the server with the ability to respond directly to the client, thus avoiding any intermediate buffering.

As already mentioned, GM only supports data transfers from/to DMA-able memory. Such memory can be obtained with a GM memory allocation routine. Alternatively, on some systems (including Linux), GM allows registration of already-allocated memory. The zero-copy protocols in application level libraries require registration of the user memory, which underneath involves pinning memory. Since ARMCI applications can potentially use the entire physical memory (or more thanks to the virtual memory) in the system in context of one-sided communication, we cannot pin user data statically in ARMCI_Malloc. Pinning can only be done dynamically as a part of data transfer operations, and when these operations are complete, the memory must be unpinned. Unfortunately, memory registration and dereagistration on Linux through the GM driver is an expensive operation, see figure 4 in section 4. On our Dell Linux system, it is comparable in cost to memory copy thus its cost has to be considered in the protocol design. In addition, on Linux `gm_register` fails for the System V shared memory – according to Myricom due to a problem in the GM implementation. This constraint requires server to use an extra copy when sending shared memory data via a DMA-able buffer. On the client side, the user buffer is located most often in local memory; therefore, it can be pinned. Our baseline implementation for GM does not require memory registration and uses two DMA-able buffers allocated with GM by the server and client at the startup. The data must be transferred to and from these buffers, thus the corresponding two extra memory copies reduce the asymptotic transfer rate achievable in that version. This implementation scheme is used on systems such as Solaris where the registration is not possible. On systems where it is, ARMCI uses that scheme for short messages or messages involving multiple short contiguous data segments – GM unlike VIA does not support scatter/gather operations.

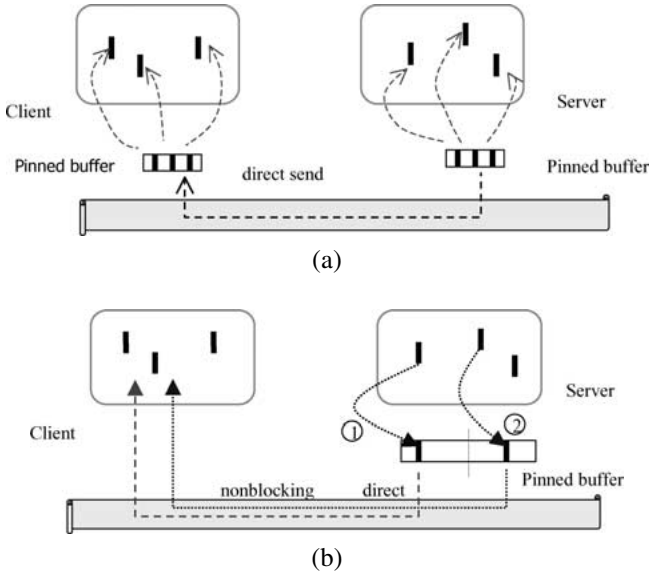
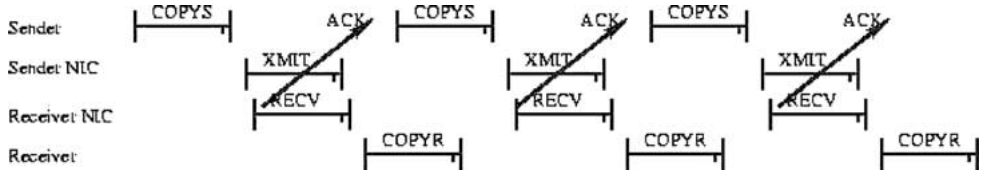


Figure 2. Baseline (a) and pipelined (b) implementation of multi-strided get.

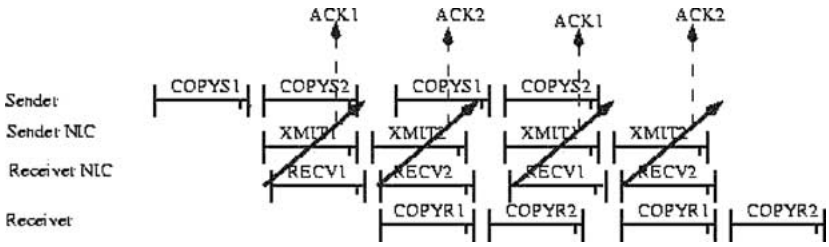
3.3. Get operation

We improved performance of the baseline implementation for medium and large messages by (1) pinning client buffer when possible, and (2) using pipelined nonblocking sends that overlap with the memory copy on the server side. The difference between the baseline and pipelined implementation is demonstrated for the multistrided get operation in figure 2. This implementation tries to pipeline and overlap memory copy operations with nonblocking communication using the GM directed send. For large data segments occuring in contiguous or strided requests, our implementation can increase the overlap level by splitting contiguous data segments into chunks to send them in the pipelined fashion.

In the next step, we attempted to address the registration cost of the client buffer. In the previous version, this operation was performed before the request was sent. At least a partial cost of this step can be removed from the critical path by overlapping it with the memory copy done by the server. We use an extra flag on the server side for the client to notify the server about the registration state of the user buffer. The notification is performed by a direct send (put). Client sends a request to the server and then calls `gm_register` for the user buffer. Before that operation is complete, the server can receive the request and start copying the first por-



The nonpipelined version of ARMCI's Put: COPYS – copy from second message send buffer; XMIT/RECV – the transmission and reception phases of the message; COPYR – copy from the provided receive buffer to destination.



Pipelined implementation of PUT: There are two buffers on the second side, COPYS1 indicates copy into one such buffer and COPYS2 into the second one. Similarly, one thereceived side, COPYR1 indicates copy into one of the two available buffers, COPYR2 indicates a copy into the second one.

Figure 3. Time diagram for the baseline and pipelined implementation of the Put operation.

tion of the requested data from shared memory into its pinned buffer. However, before the first portion is sent back to client, the server must check the status of the flag. Only when its value indicates that the client buffer registration is succesful can the server issue a GM directed send operation targeting that buffer. If the client notifies the server that registration was not succesful, processing of the current request is abandoned by the server. In that case, the client will send it one more time using a different approach that does require registration. In our previous paper [15], registration of client buffer was performed in one step. We refined this technique by splitting client registration into two steps – first involving only as much memory as the maximum size of the internal buffer size used by server (400 KB) and second step when the rest of the client buffer is registered. We use two acknowledgment notifications about progress of registration on the client side. This refinement improves the performance for larger requests by increasing the level of overlapping between operations performed by the client and server.

3.4. Put operation

The baseline implementation of ARMCI Put is comprised of three phases (see figure 3): (1) A data copy from the source data to the message send buffer (a pre-pinned, pre-allocated buffer, allocated when ARMCI is initialized), which we represent as 'COPYS' phase. This copy is done in the chunks of the sizes of the message send buffer. (2) The actual data transmission phase, where the gm_send_with_callback function is used to actually transmit data in the message send buffer to the destination. In this phase the data from the message send buffer at the source node is DMA'ed to a receive buffer at the destination node. This phase is represented as 'XMIT' and 'RECV' phase, where 'XMIT' stands for an operation performed at the sender NIC to DMA the data and 'RECV' is used to represent an operation performed at the receiver to receive the data into a receive buffer. (3) The copy to destination memory location from the receive buffer at the destination side, which is represented as the 'COPYR' phase.

We have designed and implemented a pipelined version of ARMCI Put operation essentially pipelining both the receiver and the sender side. In the pipeline, we overlap the 'XMIT' and the 'COPYS' phase as well as the 'RECV' and the 'COPYR' phase. The pipelined implementation has multiple send and receive buffers. Hence the 'COPYS' phase here is overlapped with the 'XMIT' and the 'COPYR' phase overlapped with 'RECV'. Instead of copying one chunk of data, transmitting it, and then waiting for an acknowledgement, we maintain a set of send buffers. Our analysis after timing the COPYS and the XMIT phase shows that the ratio of time taken in the XMIT phase to the time taken in the COPYS phase is between 1 and 3 for most message sizes. This implies that 2 buffers would be enough in most of the cases to efficiently fill the pipeline and optimize the pipelined algorithm performance.

4. Experimental results

The experiments were performed on a Linux (2.2.14smp kernel) cluster with Dell PowerEdge-1300 nodes, dual 500-MHz PIII, 512-MB main memory, and 512-kB, level-2 half-speed cache. The Myrinet network was based on the Universal, 64/32-bit, 66/33-MHz, Myrinet-SAN/PCI interface, M2M-PCI64A-2 cards, connected to the host by the 32-bit PCI interface. We used the GM message-passing library, version 1.2. In addition, to compare effectiveness of the GM port we also used the TCP/IP protocol over Myrinet and 100BaseT Ethernet. We could not run our experiments with the Trapeze TCP/IP drivers for Myrinet as they are not available for Linux.

Performance of memory copy, and registration and deregistration on this system is shown if figure 4. We note that the combined cost of registration and deregistration operations is a nonlinear function of the data size. The curve describing performance of memory copy is essentially linear past 4096 bytes, at which point our code is switching to an algorithm optimized for larger data sizes.

4.1. Communication performance

Figures 5 and 6 illustrate performance of the ARMCI get operation. In the tests, we attempted to avoid caching effects by using consecutive accesses that target different memory areas on remote side. In figure 5 we compare performance of the three schemes: baseline, pipelined, and pipelined with overlapped client memory registration for large messages. The two optimizations described in previous section increase the bandwidth to 78 MB/s, which is 19 MB/s over the baseline implementation. In order to better understand the efficiency of our approach we measured the rates of memory registration, memory copy, and GM bandwidth on our cluster. They were $b_{\text{registration}} = 411$ MB/s for the memory registration, $b_{\text{copy}} = 330$ MB/s for memory copy (for our optimized version of memcpy), and $b_{\text{GM}} = 101.8$ MB/s for GM. The GM one-way b/w was reported by the Myricom program `gm_allsize` for transfers using DMA-able buffers (the round trip b/w was reported by that program as 45.81 MB/s).

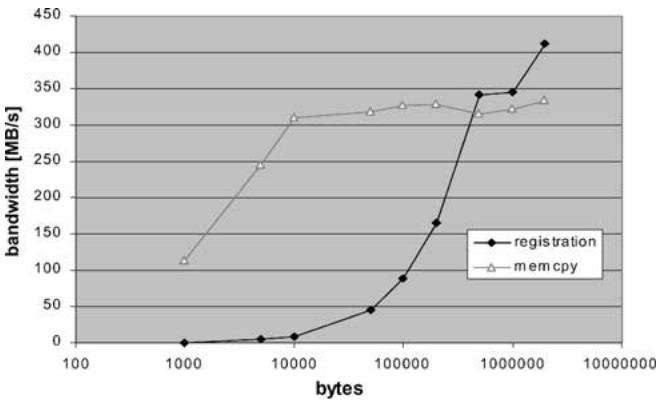


Figure 4. Combined cost of memory registration and deregistration compared to memory copy.

The performance model for the baseline implementation of the ARMCI get that does not use memory registration is

$$b_{\text{get}} = \frac{1}{2/b_{\text{copy}} + 1/b_{\text{GM}}} = 61.9 \text{ MB/s},$$

as the key time components on the critical path of the get operation are: two memory copies and the data transfer through the network. The transfer time for the request alone is negligible in context of large requests. The computed value is consistent with our experimental results for the baseline implementation. If registration of memory on both sides worked (i.e., on a uniprocessor system where ARMCI does not use shared memory) and we were able to register memory rather than copy the data the model would be

$$b_{\text{get}} = \frac{1}{2/b_{\text{registration}} + 1/b_{\text{GM}}} = 68.07 \text{ MB/s}.$$

For the full overlapping of memory registration on the client side and memory copy on the server side the model is

$$b_{\text{get}} = \frac{1}{1/\max(b_{\text{registration}}, b_{\text{copy}}) + 1/b_{\text{GM}}} = 77.7 \text{ MB/s}.$$

We obtain even a higher rate in the optimized implementation as ARMCI uses pipelined send that splits larger segments into chunks to partially overlap the copy operation with sending data through the network.

Figure 6 shows the ARMCI get performance of the optimized GM implementation over the TCP/IP implementation on Myrinet and Ethernet.

The latency of get operation was 75 μs for ARMCI/GM on our 500 MHz PIII Linux system. As get involves round-trip traffic across the network, this number could be compared with the 30 μs, round-trip latency for MPICH/GM. An important difference is that MPICH receives data in the polling

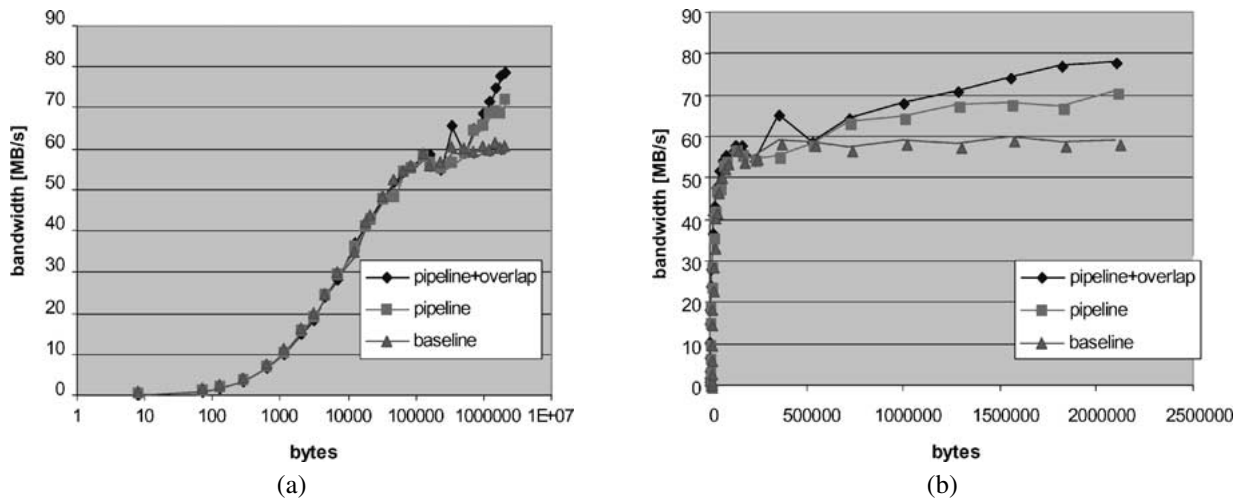


Figure 5. Performance of three implementations of get operation ((a) log and (b) linear scale).

mode with both sides actively participating in the data transfer, whereas ARMCI remote process in the benchmark is unaware of communication targeting its address space and the request is processed in the interrupt mode. In case of ARMCI, the latency number includes the cost of activating thread blocked in `gm_blocking_receive_no_spin`. On another Linux system with the 666 MHz EV67 Alpha CPU and the same Myrinet card, the latency is 45 μs. For another comparison, the ARMCI get latency on the Cray T3E is 5 μs, and on the IBM SP, it is 77 μs. Although the SHMEM interface (used by the ARMCI implementation) on the Cray T3E provides excellent latency, both the IBM SP LAPI and GM interfaces offer a better support for optimizing noncontiguous data transfers, which in part can improve the overall communication rate for applications that use such transfers. An example illustrating this point is given in figure 7.

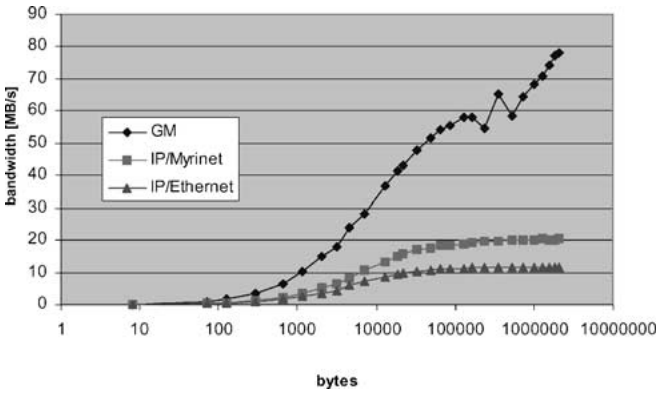


Figure 6. Performance of the ARMCI get operation using Myrinet and Ethernet.

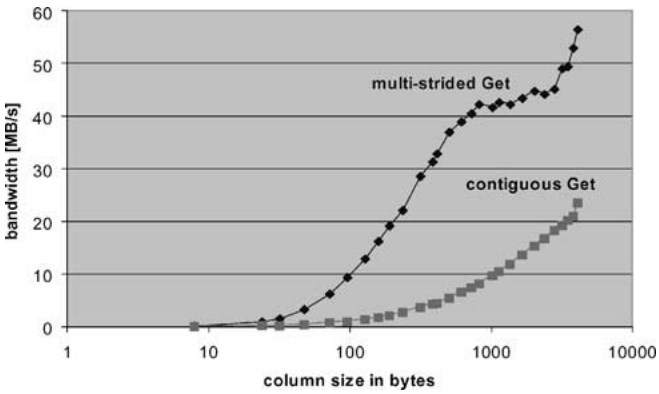


Figure 7. Bandwidth in multi-strided or series of contiguous (one column at a time) ARMCI get operations on GM for a square section of matrix.

We also tried compare get bandwidth with performance of the MPICH/GM message-passing communication. MPICH/GM similarly to ARMCI uses registration of user buffers to optimize performance for large requests. However, unlike ARMCI the Myricom channel device for MPICH by default does not deregister the user memory after communication is complete [16]. Therefore, repeated data transfers involving the same buffers do not incur the expensive memory registration cost which appears in every instance of ARMCI communication (where we do not make the assumption that the user data buffers would ever be reused). We found that for a single 2 MB message exchanged in the ping-pong test the MPICH/GM achieves 71.5 MB/s as compared to 78 MB/s in ARMCI get. However, by avoiding registration costs for next messages involving the same buffers, the average rate for 100 messages in a the ping-pong test grows to 82.76 MB/s.

Figure 8 illustrates performance of the put operation. In order to better understand the performance it should be noted that semantics of put operation allows an implementation to return when the user buffer can be reused. This does not necessarily imply that the data transfer completed on remote side. In particular since our implementation uses 400 KB internal registered buffer for sending data, put operation can return when data is copied out from the user to this internal buffer and a nonblocking GM send call is issued. For larger data sizes, our implemntation of put operation packetizes data in 400 KB chunks and sends them using one (baseline) or two (pipelined) buffers. We are not seeing the memcopy band-

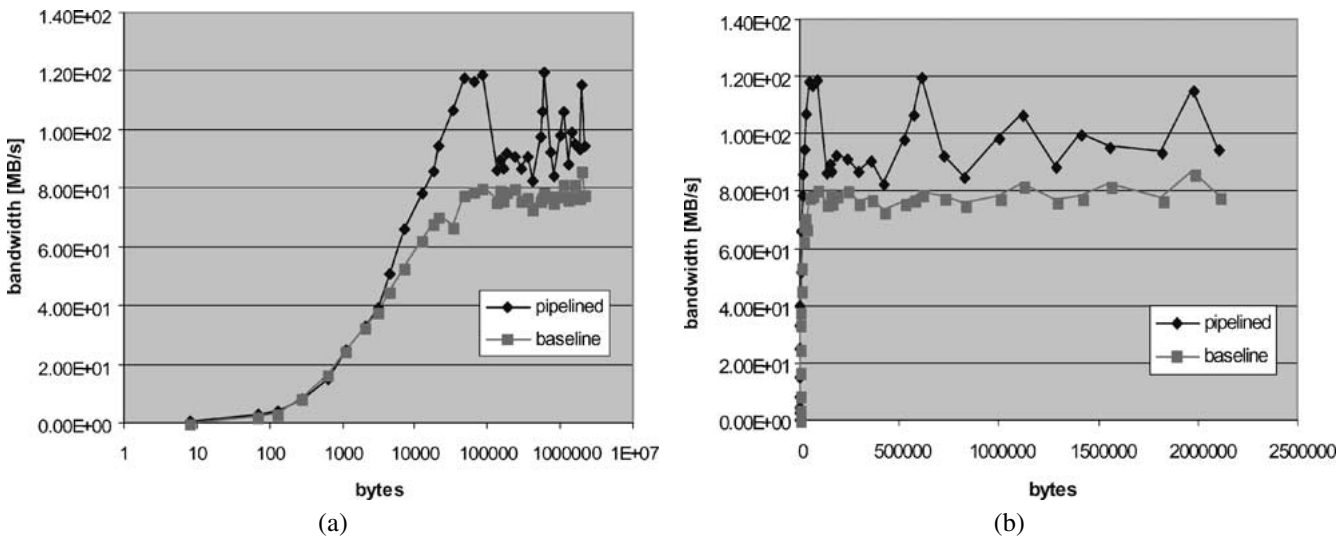


Figure 8. Performance of baseline and pipelined implementations of put ((a) log and (b) linear scale).

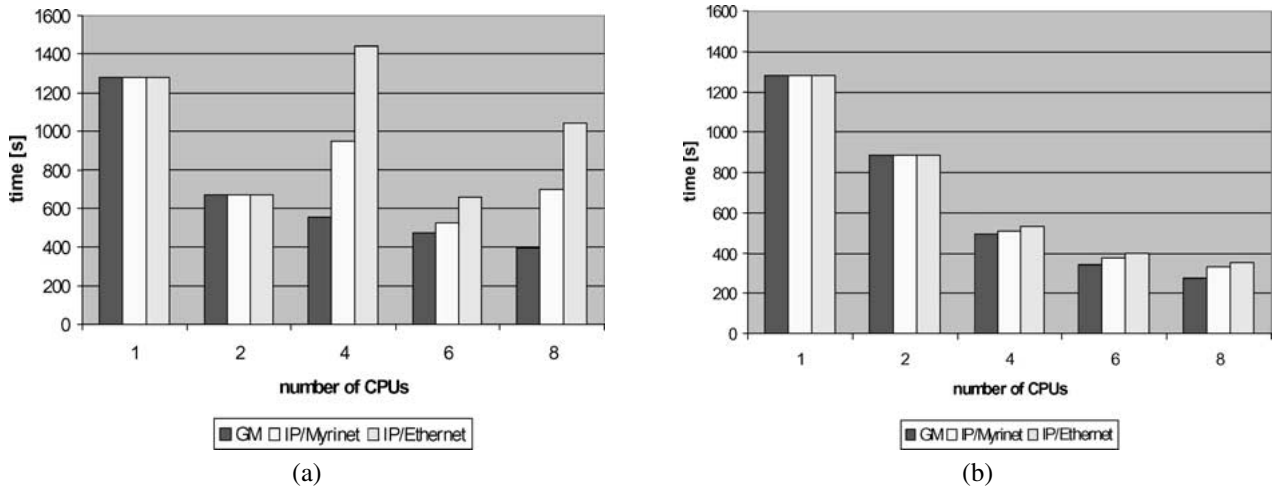


Figure 9. Performance of the SPLASH-2 LU benchmark with (a) *block-cyclic* and (b) *block* distributions using ARMCI implementations on top of GM and TCP/IP – Myrinet and 100BaseT Ethernet.

width for data sizes <400 KB since our performance test issues a series of put operations rather than just one and reports averaged timing results. The performance curve is not contiguous due to (1) the request packetization based on fixed size buffer(s); (2) the fact that when put returns the data in the last internal ARMCI buffer might be not fully transmitted (our flow control allows one outstanding nonblocking send operation to assure ordering and buffer space availability); and (3) the measurement is done for a series of puts and averaged. The latency for the put operation was measured as 28 μs by averaging time for 550 8-byte operations.

The performance of ARMCI/GM also can be compared to the results of the HPVM/FM implementation of SHMEM [17]. On a dual CPU node with the same generation of Myrinet and under Windows NT, the 26-μs latency and 67-MB/s bandwidth were achieved in `shmem_get`, while `shmem_put` achived 15-μs latency and 70 MB/s bandwidth. That implementation exploited (1) the FM support for one-sided communication on the NIC (MCP program), and (2) a dedicated CPU with an extra thread that uses active polling rather than blocking like in our approach. The paper recommended dedication of the second processor to the polling thread but did not report any application experiences with SHMEM to evaluate tradeoffs and merits of their design. While HPVM achieves lower latency, our approach delivers better bandwidth on the standard GM communication layer optimized for two sided protocols and does not require dedicating separate processor for handling communication.

4.1.1. Application study

The SPLASH-2 benchmark suite [19] is a set of parallel applications for use in the design and evaluation of shared-memory multiprocessing systems. We choose the LU program, which is one of the kernel programs from SPLASH-2, to evaluate the performance of our approach. The LU program factors a dense matrix into the product of a lower and an upper triangular matrix. The factorization uses blocking to exploit temporal locality w.r.t. individual sub-matrix elements. Originally designed to run on shared memory systems, this benchmark can only be used on a single SMP node. Due to the simplicity of the ARMCI progress rules and its memory consistency model, only minor modifications were needed for use in one-sided communication such as replacing *memcopy* with put/get operations. The computation requires transferring data blocks from the same row and column (for diagonal blocks). The original benchmark uses block cyclic distribution as a way of achieving load balancing. We also used block decomposition, as it had better locality of the data accesses thus reducing the amount of communication across the network. The blocks are distributed according to a block pattern, such that the block that needs to be transferred has a better chance to reside in a local memory or neighboring memory on the same node.

We used a matrix size of 3072 and a block size of 32 to study performance of the LU benchmark. For both block cyclic and block distributions, we tested three implementations: (1) GM over Myrinet, (2) TCP/IP over Myrinet, and (3) TCP/IP over 100baseT Ethernet. As all of them exploited shared memory in the same way, their single SMP results are identical. Since the block cyclic decomposition yields better load balancing the code scales better on a single SMP node. However, communication patterns are spread out across all the nodes. Consequently, the benchmark performance on multiple nodes is better with block decomposition. Performance of the benchmark with GM is always better than with the IP over Myrinet or Ethernet. For block-cyclic decomposition, a higher amount of communication through the networks leads to poor performance for TCP/IP over Myrinet as well as Ethernet.

We also tested performance of a real scientific application, NWChem [19] when solving a DFT (functional theory) $SiOSi_3$ benchmark. We run this test with the same h/w and s/w configuration as the LU benchmark. In addition, for ARMCI/GM we also run it on another Linux system equipped with the dual 800 MHz PIII nodes and Myrinet. In this case too, the GM version scaled pretty well from 1 to 16 processors. Surprisingly, for NWChem when using TCP/IP over Myrinet, the performance was much worse than

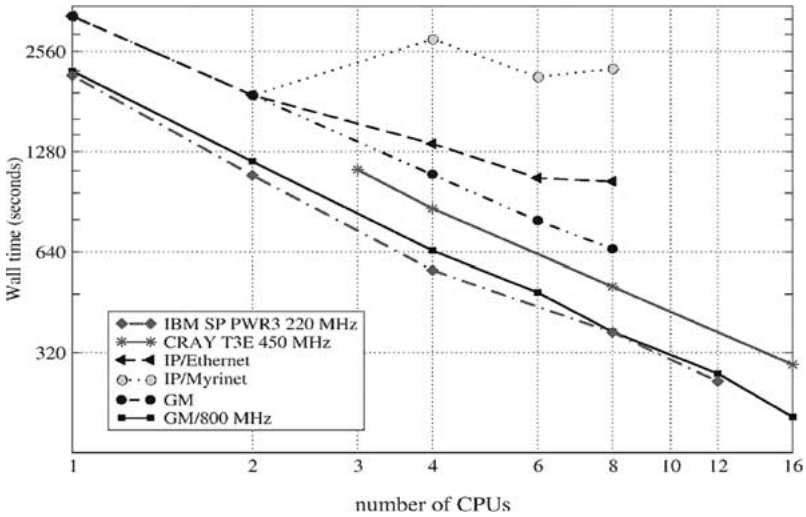


Figure 10. The DFT benchmark results on the Cray T3E, IBM SP (8-way SMP), and Linux cluster configurations with dual Pentium III: GM (500 and 800 MHz), IP over Myrinet, IP over Ethernet.

when using Ethernet. It seems that the driver implementation could be responsible for the unexpected performance drop-offs. Finally, as shown in figure 10, we compared scaling of this code on the two Myrinet clusters: the IBM SP with "Nighthawk-I" 8-way SMP nodes and the Cray T3E. On the IBM SP, only four tasks per node were used as the TB3MX switch allows only for that number of tasks to communicate in the user-space mode. The overall scaling of this application on the Linux/Myrinet is very close to that on the Cray T3E and the IBM SP. The IBM version scales better when the code executes within the single SMP; however, once multiple nodes are used the scaling becomes slightly worse than on the Myrinet cluster.

5. Conclusions and future work

We have described an efficient implementation of one-sided communication on top of GM and Pthreads for the Myrinet-based SMP clusters. This approach avoids polling, is compatible with shared memory, and exploits pipelining, non-blocking communication, and overlapping memory registration with memory copy to maximize the transfer rate. Our implementation uses the standard distribution of GM and is compatible with MPICH/GM. Performance of our two applications was very good, and the GM implementation allowed them to avoid performance anomalies of the standard TCP/IP version when running on Myrinet. We plan to evaluate feasibility and potential advantages of using other messaging systems such as FM, BIP, or PM for implementing ARMCI on Myrinet. Our experience brings up some suggestions for improving GM support for one-sided communication such as: ability to register shared memory, reducing cost associated with activating a thread blocked in GM upon a message arrival, or adding scatter capability to GM data transfer operations. However, in context of the current work we do not see the missing get operation in GM as a critical shortcoming – the usability issues related to the memory registration required by GM would limit the use of that capability directly anyway.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy (DOE) at Pacific Northwest National Laboratory (PNNL), which is operated by for DOE by Battelle. This work was supported under the DOE 2000 ACTS project sponsored by the Mathematical, Information, and Computational Science Division of DOE's Office of Computational and Technology Research. The Molecular Science Computing Facility in the Environmental Molecular Sciences Laboratory at PNNL and National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory provided the high-performance computational resources needed for the work.

References

[1] S. Lumetta, A. Mainwaring and D. Culler, Multi-protocol active messages on a cluster of SMP's, in: *Proceedings of Supercomputing '97* (1997).

[2] H. Tezuka, A. Hori, Y. Ishikawa and M. Sato, PM: An operating system coordinated high performance communication library, in: *High Performance Computing and Networking*, Lecture Notes in Computer Science, Vol. 1225 (Springer, 1997).

[3] L. Prylli and B. Tourancheau, BIP: A new protocol designed for high performance networking on Myrinet, in: *Workshop PC-NOW, IPPS/SPDP98* (Orlando, USA, 1998).

[4] S. Parkin, M. Lauria, A. Chien et al., High Performance Virtual Machines (HPVM): Clusters with supercomputing APIs and performance, in: *8th SIAM Conference on Parallel Processing for Scientific Computing (PP97)* (1997).

[5] Myricom, The GM Message Passing System (10/16/1999).

[6] J. Nieplocha and B. Carpenter, ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems, in: *Proc. RTSPP IPPS/SDP'99* (1999).

[7] K. Parzyszek, J. Nieplocha and R. Kendall, A generalized portable SHMEM library for high performance computing, in: *Proceedings Int. Conf. on Parallel and Distributed Computing and Systems, PDCS-2000* (2000).

[8] R. Bariuso and A. Knies, *SHMEM's User's Guide*, SN-2516, rev. 2.2 (Cray Research, Inc., Eagan, MN, 1994).

[9] J. Nieplocha, J. Ju and T.P. Straatsma, A multiprotocol communication support for the global address space programming model on the IBM SP, in: *Proc. EuroPar-2000*, Lecture Notes in Computer Science, Vol. 1900 (Springer, 2000).

[10] K. Langendoen, J. Romein, R. Bhoedjang and H. Bal, Integrating polling, interrupts, and thread management, in: *Proc. Frontiers'96* (1996).

[11] J. Hsieh, T. Leng, V. Mashayekhi and R. Rooholamini, Architectural and performance evaluation of GigaNet and Myrinet interconnects on clusters of small-scale SMP servers, in: *Proceedings SC2000*, http://www.sc2000.org/techpapr/papers/pap.pap294.pdf

[12] M. Banikazemi, R. Govindaraju, R. Blackmore and D.K. Panda, Implementing efficient MPI on LAPI for the IBM RS/600 SP systems: Experiences and performance evaluation, in: *Proc. IPPS'99* (1999).

[13] A. Gallatin, J. Chase and K. Yocum, Trapeze/IP: TCP/IP at near-gigabit speeds, in: *1999 USENIX Technical Conference*, June 1999.

[14] K. Moldeklev and P. Gunningberg, How a large ATM MTU causes deadlocks in TCP data transfers, IEEE/ACM Transactions on Networking 3(4) (August 1995).

[15] J. Nieplocha, J. Ju and E. Apra, One-sided communication on SMP clusters with Myrinet using the GM message-passing library, in: *Proc. CAC'01/IPDPS'01* (2001).

[16] Myricom, Portable MPI Model Implementation over GM, version 1.2.1 (file README-GM) (23 June, 2000).

[17] L.A. Giannini and A. Chien, A software architecture for Global address space on communication on clusters: Put/Get on Fast Messages, in: *7th Int. IEEE Symposium on High Performance Distributed Computing, HPDC-7* (1998).

[18] S.C. Woo, M.O. Ohara, E. Torrie, J.P. Singh and A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, in: *Proc. 22nd International Symposium on Computer Architecture* (1995).

[19] High Performance Computational Chemistry Group, NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.0, Pacific Northwest National Laboratory, Richland, WA (2000), http://www.emsl.pnl.gov:2080/docs/nwchem

Jarek Nieplocha is a technical group leader of the Scalable Computing Group in Computational Sciences and Mathematics directorate at Pacific Northwest National Laboratory. He is also a faculty member of the Computer Science Department at the Washigton State University. His interests are in parallel computing, high performance input/output and interprocessor communications. He received an R&D 100 Award and a patent in the area of network protocols.

Edoardo Apra is a senior research scientist in Molecular Sciences Software Group in Theory, Modeling and Simulation or Environmental Molecular Sciences Laboratory of Pacific Northwest National Laboratory. He received Ph.D. in Physical Chemistry from University of Turin in 1993.

Vinod Tipparaju is a Senior Research Scientist in Scalable Computing Group at Pacific Northwest National Laboratory. His reseach interests include Parallel and Distributed Computing and Cluster Communication Protocols. He has a Bachelors in computer science from Nagarjuna University, AP, India and a Masters in computer science from the Ohio State University, Columbus, OH, USA.