

Protocols and Strategies for Optimizing Performance of Remote Memory Operations on Clusters

Jarek Nieplocha
Pacific Northwest National Laboratory

Vinod Tipparaju

Amina Saify

Dhableswar Panda
Ohio State University

The paper describes software architecture for supporting remote memory operations on clusters equipped with high-performance networks such as Myrinet and Gigaset/Emulex cLAN. It presents protocols and strategies that bridge the gap between user-level API requirements and low-level network-specific interfaces such as GM and VIA. In particular, the issues of memory registration, management of network resources and memory consumption on the host, are discussed and solved to achieve an efficient implementation.

1. Introduction

Remote memory operations offer an intermediate programming model between message passing and shared memory. 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. Certain types of shared memory applications can be implemented using this approach. In some other cases, remote memory operations can be used as a high performance alternative to message passing. Many of such applications are characterized by irregular data structures, dynamic or unpredictable data access patterns. MPI-2 offers one version of remote memory operations with two particular flavors: “active” and “passive target” one-sided communication. Other versions are found in vendor specific interfaces such as LAPI on the IBM SP, RDMA on the Hitachi SR-8000, MPlib on Fujitsu VPP-5000, and in other portable interfaces such as ARMCI[6] or SHMEM[8]. Differences between these models can be significant in terms of progress rules and semantics, and they can affect performance. MPI-2 offers a model closely aligned with the traditional message passing and includes high-level concepts such as windows, epochs, and distinct progress rules for “passive-“ and “active” “target” communication. In ARMCI we are focusing on a low-level interface and simpler progress rules motivated by the h/w support for remote memory operations on existing MPP systems. The library is intended be used as a run-time system for other programming models such as Global Arrays [18], Co-Array Fortran [19] or UPC compilers, or even SHMEM-like library [7,8].

We are interested in exploiting the high-performance networks and protocols for optimizing remote memory copy on commodity clusters. The predominant high-

performance network for clusters is Myrinet. Others include Gigaset/Emulex cLAN, Dolphin SCI, or Quadrics Elan. cLAN network supports the industry standard Virtual Interface Architecture (VIA) interface, a protocol closely related and supported on next generation/emerging InfiniBand networks. VIA, GM, and Infiniband interfaces offer some support for remote memory operations. However, an important mismatch between user level programming interfaces and these network protocols relates to virtual memory. Neither Myrinet, nor cLAN, nor Infiniband networks are integrated with the virtual memory subsystem. The native remote memory copy on these networks can address only so called registered memory on both sides of the data transfer. Memory registration involves locking pages in physical memory and thus potentially deprives applications of the benefits of virtual memory. In addition, the amount of memory that can be registered/locked is usually limited. In some cases, for example GM on Solaris, registration of existing memory segments is not even supported. These constraints have a profound impact on the implementation strategies of user-level remote memory interfaces on such networks.

In this paper, we describe software architecture for supporting remote memory operations on clusters with networks such as Myrinet or cLAN. When combined with protocols and strategies for efficient management of network and host resources, this architecture can both deliver high performance and match network protocols with requirements of remote memory operations. The protocols and strategies address issues such as buffer memory consumption, management of GM tokens, dynamic memory registration, zero-copy data transfers and adaptive data streaming. For example, the adaptive data streaming technique bridges the performance gap between remote memory operations that target registered and those that use regular memory. Our approach relies on the standard unmodified system software and drivers for Myrinet and cLAN rather than on custom/alternative drivers and interfaces (e.g., AM [1], PM [2], BIP [3], and FM [4]) interfaces that replace the standard Myrinet Control Program (MCP) on the network interface card.

The paper makes several contributions to the field. First it presents a software architecture that supports efficiently a complete set of remote memory operations including remote copy, accumulate, locks, and atomic read-modify-write operations implemented on top of low-level messaging interfaces such as GM and VIA, and the standard operating system interfaces. 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 the context of standalone communication benchmarks. Finally, the paper offers some indications on what additional features would be desirable in network communication APIs to better support remote memory operations.

The paper is organized as follows. Section 2 discusses remote memory functionality. Section 3 presents software infrastructure for implementing communication based on VIA and GM. Experimental results are provided and discussed in Section 4, and conclusions are included in Section 5.

2. Remote memory operations

A minimum set of remote memory operations available in virtually all portable and vendor specific remote memory interfaces includes: remote memory copy (get/put) and synchronization operations. On systems where remote store operation (put) is nonblocking and/or the underlying network does not guarantee ordering (e.g., IBM SP switch), a fence operation is provided. Specific synchronization operations are available in remote memory libraries, and they vary widely. MPI-2 provides locks; Cray SHMEM and Hitachi RDMA offer atomic swap, Fujitsu MPLib supports semaphores, while IBM LAPI includes several flavors of atomic read-modify-write operations. ARMCI offers a fairly complete superset of these operations, with interfaces generalized to be portable across variety of platforms. On many systems with native remote memory copy including IBM SP, Cray T3E, Fujitsu VPP-5000, or Hitachi SR-8000, ARMCI is implemented as a thin layer on top of the native interfaces for the functionality supported by these interfaces and a thicker layer for the functionality that is not. In addition to the

MPP systems, the library is available on clusters of common Unix and Windows workstations/servers.

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. ARMCI supports up to eight stride levels corresponding to eight-dimensional arrays. The library provides three classes of operations (Table 1): 1) data transfer operations including put, get, and accumulate (operation also available in MPI-2 but not in any vendor specific remote memory interface); 2) synchronization operations—atomic read-modify-write, locks/mutex operations, and 3) operations for memory management, local and global fence, and error handling.

ARMCI only targets remote memory allocated via the provided memory allocator routine, `ARMCI_Malloc` (similar to `MPI_Win_malloc` in MPI-2). On shared memory systems including SMPs, this approach allows to allocate shared memory for the user data and consecutively map remote memory operations to direct memory references, thus achieving sub-microsecond latency and a full memory bandwidth [9].

3. Software architecture for remote memory operations on clusters

The most affordable network used in commodity clusters is Ethernet with TCP/UDP-IP sockets as the primary communication protocol. This protocol offers no explicit support for remote memory operations. The other networks used for cluster computing include Myrinet, cLAN, Quadrics, and SCI. The native communication protocols on these networks offer a variable level of support for remote memory operations from limited (Myrinet) to extensive (Quadrics). In order

Table 1: Remote memory operations in ARMCI

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

to provide a complete set of remote memory operations, the missing capabilities need to be implemented using operating system services and protocols.

In the following subsections, we briefly describe characteristics of two network protocols, then discuss communication protocols and strategies developed to achieve high performance of remote memory operations and at the same time minimize resource consumption. We consider clusters with two representative high-performance networks: Myrinet – for its popularity -- and cLAN for its h/w support for VIA, a protocol formulated by the PC industry leaders that will also be offered in the forthcoming Infiniband networks [17].

Target network protocols

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) protected, user-level access to the Myrinet; 2) reliable, ordered delivery of messages; 3) automatic recovery from transient network problems; 4) scalability to thousands of nodes; and 5) low host-CPU utilization. In addition to message passing, GM supports put operation. However, GM can only send messages from or receive messages into registered (DMA-able) memory. GM on Solaris does not support registration of memory that was allocated as non-DMA-able in the first place. In earlier versions of GM, registration of shared memory did not work on Linux.

Virtual Interface Architecture (VIA) is a high-performance communication layer for system area networks (SANs). Its design was strongly influenced by the academic research on low-overhead communication as well as experience with MPPs. Due to its widespread industry support (including Intel, Compaq and Microsoft) and connection to Infiniband, it is likely that VIA will become more widely adopted. VIA provides protected zero-copy data transfer, without requiring operating system kernel assistance. Both message passing and remote memory copies are available in VIA. However, only remote write (put) is mandatory, and for example on cLAN the optional remote memory read is not implemented. VIA requires that memory used in all the communication be registered by the application prior to communication so that it can be pinned to avoid page faults on transmission or reception of data.

The two protocols differ in several key respects. VIA is connection based while GM offers connectionless approach. When comparing to GM, VIA puts more responsibility on the user to do flow control. For every message sent there must always be a buffer available. GM would attempt to resend messages if the buffer is not available. For practical purposes, VIA requires at least one buffer preposed for every other process but it does not mandate message to match exactly the buffer size (can be smaller). Under GM, buffers must be posted

for entire range of messages as the message can only be delivered into a buffer that matches its size, but messages can be delivered into a buffer from any process. Neither GM nor VIA offer any support for remote synchronization/mutual exclusion operations and both protocols require memory registration. Some of these limitations can be addressed by layering a heavier-weight interface over VIA and GM, and they also have a profound impact on our system design.

Client-server architecture

To support a full set of remote memory operations on clusters with GM or VIA protocols our strategy relies on client server architecture. It is implemented by starting on each machine “server” thread(s) dedicated to remote-memory operations that are issued by the remote clients (user tasks). If the implementation of network protocols is not thread-safe, a heavyweight process can be used instead. The server thread upon receiving a request executes a handler function corresponding to the appropriate remote memory operation and, if needed, sends data back to the client.

The optimal number of server threads needed depends on several factors such as the number of processors and user tasks/processes per node, network throughput and the communication load and patterns in applications. For performance reasons on the current networks and hardware with low number of processors per SMP node, a single thread is appropriate. However, the number of threads is also related to the issue of how the memory used for RMA is allocated. In libraries that offer specific interfaces for memory allocation such as MPI-2 and ARMCI, one thread could suffice since their memory allocation operations can allocate shared memory. Otherwise, one thread for each user process would be required. As we show in this paper, a combination of server threads, network protocols and OS support for mutual exclusion is sufficient to implement a full set of remote memory operations and deliver high performance. With that architecture, special care is required to minimize resource consumption (memory, network bandwidth, CPU utilization) for the benefit of applications.

To prevent server thread/process in the absence of one-sided communication requests from consuming CPU resources needed by user processes, blocking wait rather

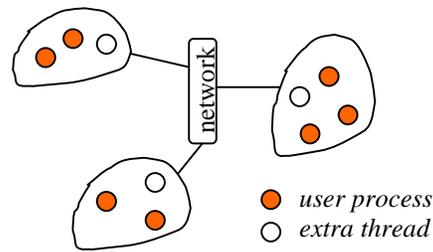


Figure 1: Extra threads on SMP cluster

than active polling of the network interfaces is appropriate. Both VIA and GM offer blocking communication calls that effectively block the calling thread until an associated communication event occurs. Although VIA offers both polling and blocking calls for completing data transfer operations, in the cLAN implementation their overhead differences are substantial as blocking calls involve interrupt processing. Therefore the blocking calls are used if extensive waiting periods are expected [10].

Memory Consumption

A conservative consumption of memory for internal buffers is critical for achieving implementation scalability. This is especially relevant for VIA, where in the server thread we need to prepost at least one buffer for every other remote client. The buffer must be as large as the maximum message that is limited by the MTU value of approximately 64KB on cLAN. Since the maximum size of the cLAN network is 128 nodes with possibly multiple client processes running on each node, to limit memory consumption we post only one buffer per every other client process and use auxiliary ACK message to notify client about availability of the buffer space. These ACK messages are not needed (are implicit) for requests such as get that bring the data back from server. We also use one extra buffer to quickly alternate it on the list of preposted buffers with the buffer that contains the current request data. That allows server to send ACK message to the client as soon as the new message arrives rather than after the current request is completed and the buffer becomes free. This improves performance in the pipelined implementation of put operations.

Under GM, buffers are not associated with particular remote client processes and the number of them is limited by the available receive tokens. Since we need to post buffers for entire range of expected messages, it is important to minimize buffer space consumption and provide sufficient number of buffers to match the application needs. The goal is to avoid a possibility of GM dropping and retransmitting messages due to the insufficient number of buffers available. We allow users to control the number of buffers for each message range at compile time to match it with the application communication patterns. There are two options: 1) uniform number of buffers per message range, 2) non-uniform number of buffers per range. For the uniform option, there are two sub options, which differ from each other in the number of buffers provided per range and hence in total memory consumption. The user can select the option that best matches the application.

Zero-copy data transfers

There are two techniques for addressing the requirement for registered memory in the network programming interfaces such as VIA and GM: 1) dynamic memory

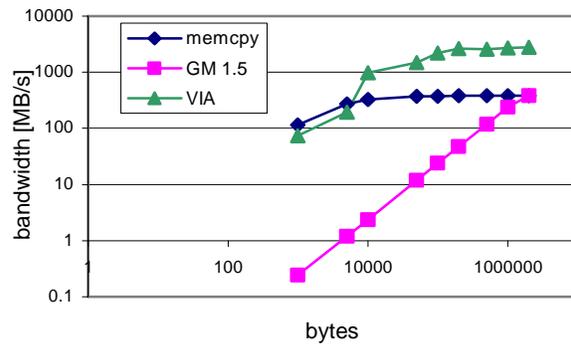


Figure 2: Performance of memory registration and copy

registration and deregistration as a part of the data transfer, and 2) by streaming data via preallocated registered memory buffers. The third alternative that requires user data to be placed in registered memory is not always feasible or desirable as, for example, it defeats the purpose of virtual memory.

The first technique is potentially more attractive as it provides zero-copy data transfers and eliminates the need for data copy present in the second one. However, it does not always lead to superior performance as the memory registration operations can be expensive. Figure 2 shows (on log-log scales) performance of memory registration operations (registration and deregistration calls combined) in the VIA and GM as compared to the bandwidth of the memory copy operation on Pentium III under Linux. The assembly-coded memcopy uses MMX registers, write combining, and prefetching instructions. In cLAN VIA and earlier versions of GM, the cost of deregistration is very small and does not depend on the number of pages. Under GM 1.4 and 1.5, deregistration of memory became more expensive and is a function of the number of memory pages involved.

In zero-copy data transfers, memory must be registered on both sides. As the cost of registration is not negligible, we overlap the memory registration on both sides, see Figure 3. A special acknowledgment flag on the server side (one for every client process) is used. For example in get operation, the client sends a request to the server before registering its memory buffer. After the registration is complete, client updates the flag by using a low-level put message (RDMA write in VIA and gm_directed_send in GM). The server after receiving

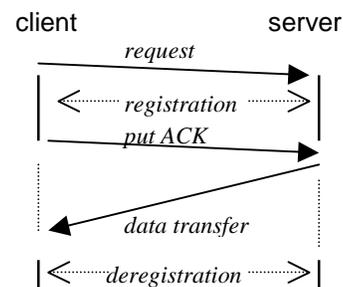


Figure 3: Get operation using dynamic memory registration with overlapping on client and server

the request, first registers its appropriate memory area and then waits until the flag is set which indicates that the client is ready to receive data.

Pipelined data streaming

Inspection of results in Figure 2 indicates that zero-copy protocols would not necessarily be always more competitive than streaming data through a preallocated and registered buffer (using memory copies). Even for VIA, where memory registration is very efficient, for small and medium requests memory copy is faster.

We also need to consider noncontiguous data transfers, strided and vector formats, which potentially involve multiple disjoint areas of memory. Even for large requests the data involved might reside on many partially occupied memory pages. Since, neither GM nor VIA offer memory registration interfaces to register collection of pages that correspond to disjoint memory areas, we would need to register pages individually. This increases the memory registration cost. To address these issues, we developed a data streaming technique based on adaptive pipelining. This approach relies on dividing the data into multiple chunks and exploits nonblocking communication operations: message send and receive on VIA and low-level put on GM, to overlap memory copies on client and sever side with data transmission. To improve performance for smaller requests the chunk size is adaptively chosen to maximize the concurrency between memory copies and data transmission operations on both sides involved in the data transfer. There are two versions of data streaming algorithm: one for put operations and one for get. They both work for contiguous and noncontiguous data. The put version is much simpler. It involves two buffers, one on server and one client side.

The baseline implementation of put requires three phases, see Figure 4: 1) A copy from the source data to the registered buffer, represented as ‘COPYS’ phase.

This copy is done in the chunks of the sizes of the buffer. 2) The actual data transmission phase done by gm_send_with_callback in GM and Vipl_send in VIA. The data from the message buffer at client is DMA’ed to a receive buffer on the server. ‘XMIT’ stands for an operation performed at the sender NIC to DMA the data and ‘RECV’ represents receiving the data into a server buffer. 3) The copy to destination memory location from the server buffer, represented as the ‘COPYR’.

In the pipelined version we overlap the ‘XMIT’ and the ‘COPYS’ phase as well as the ‘RECV’ and the ‘COPYR’ phase. The pipelined implementation requires multiple send and receive buffers. Hence the ‘COPYS’ phase 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. Since the ratio of time in the XMIT phase to the time taken in the COPYS phase is between one and three for most message sizes, two buffers suffice to efficiently fill the pipeline. Furthermore, the pipelined version is modified by rebalancing the size and ordering chunks to minimize the time needed to inject the data into the network, and maximize the overlapping.

The implementation of get operation is more complex than put. In addition to the baseline non-pipelined protocol, we developed a pipelined data streaming protocol that adapts to the message size by using variable packetization/buffer length. We found that the fixed size of the buffer does not provide optimal performance across the range of requests. Therefore, it is chosen at run-time depending on the size of the message to hide (at least partially) the memory copy costs for requests as small as 2KB. In get, large data requests are packetized twice: once to fit them into the 400KB registered buffer that is used for streaming and second time to divide them into multiple smaller chunks in order to overlap the memory copy and data

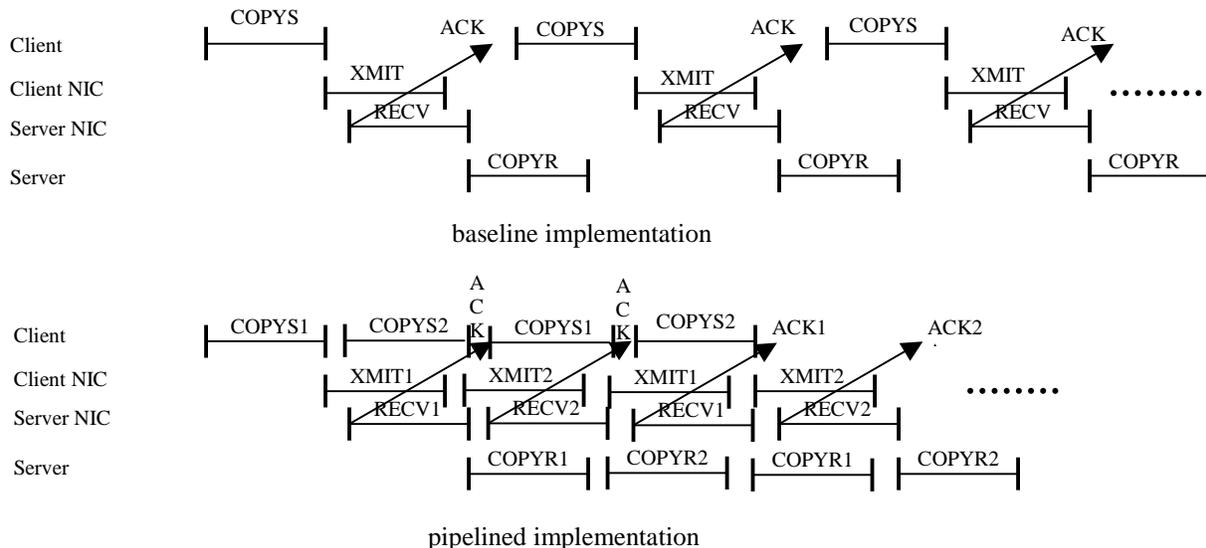


Figure 4: Baseline and pipelined implementation of put operation

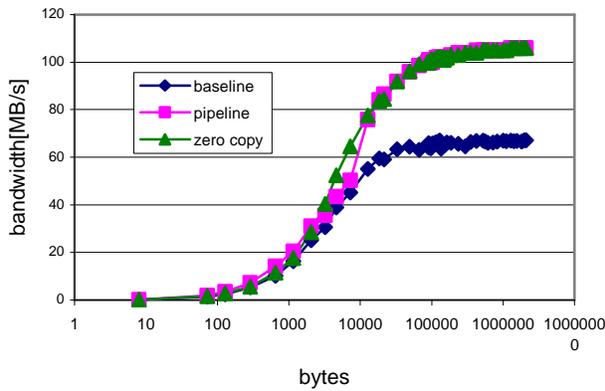


Figure 6: Performance of zero-copy, data streaming baseline and pipelined protocols for contiguous data on VIA

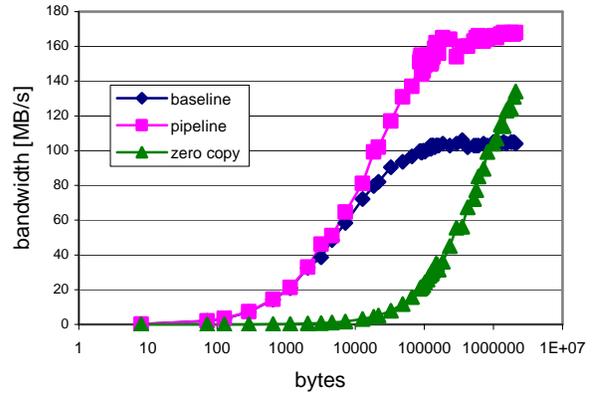


Figure 7: Performance of zero-copy, data streaming baseline and pipelined protocols for contiguous data on GM

become available. Under GM, we use a separate request message that when responded by the server indicates that the outstanding operations between the client and this server are completed. In order to reduce cost of this operation, we issue multiple requests to the servers that the calling process communicated with since the previous fence operation, and then wait for responses from all of them. In order to minimize contention in cases of multiple processes calling that operation at the same time we randomize the order requests sent to the servers.

4. Experimental Results

Our baseline configuration involved dual 1GHz Pentium III systems running Linux. One cluster used Myrinet-2000 network with GM 1.5 and the other employed the cLAN network with driver version 1.3. The MPI implementations on these systems are MPICH-GM 1.2.1..6 and MVICH 1.0a6.1. The cLAN network is rated at 125MB/s and Myrinet is rated at 250MB/s.

We used micro-benchmarks to measure performance of remote memory operations for both contiguous and strided data types. They rely on timing a series of calls and averaging the results. Some effort is made to assure that in repeated calls data is not in the cache. We

developed a closely related benchmark that uses same data reference patterns on top of MPI send/receive operations. For strided data, a user defined MPI datatype is used to define the data layouts. Our MPI benchmark is different from the traditional ping-pong tests by not reusing the same buffer and eliminating caching effects in repeated communication from/to the same buffer(s). The differences between one-sided protocols in remote memory operations and two-sided protocols in MPI send/receive communication are obvious and one can expect to see some differences in the performance for these protocols. In our paper, performance results are presented for both of them to show how effectively they exploit the network.

First, we discuss performance of contiguous get operation on cLAN and Myrinet, see Figures 6 and 7. They compare three protocols described in the previous section: zero-copy, baseline data streaming, and data streaming with adaptive pipelining. The results are closely related to the performance of 1) network, 2) memory registration and deregistration operations, and 3) memory copy. Since the registration operations in VIA are very efficient, the zero-copy protocol is most competitive for all but small messages. This is not the case on GM, where the zero-copy protocol performs

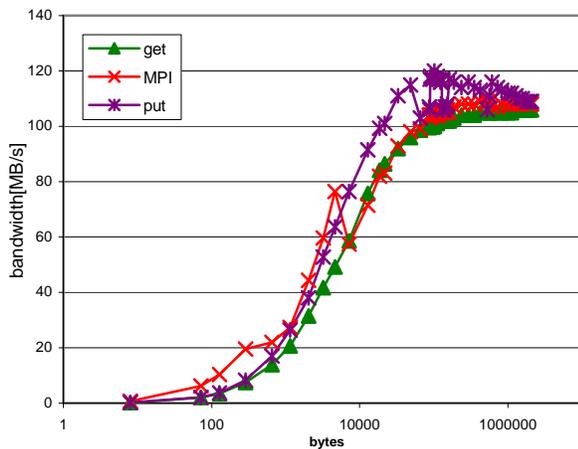


Figure 8: Performance of ARMCI get and put, and MPI send/receive operations for contiguous data on VIA

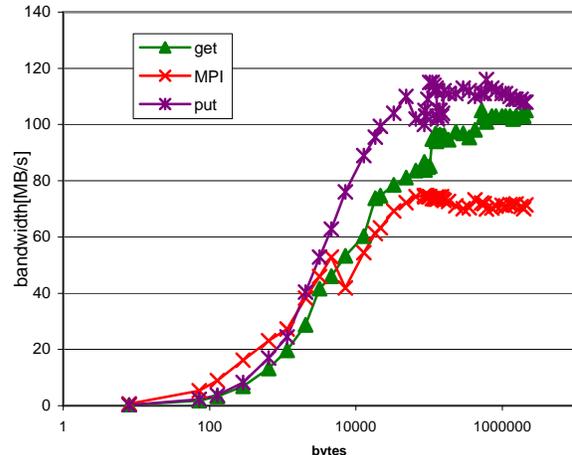


Figure 9: Performance of ARMCI get and put, and MPI send/receive operations for strided data on VIA

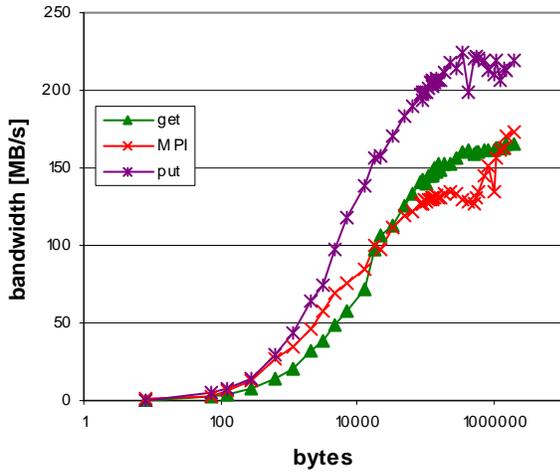


Figure 10: Performance of ARMCI get and put, and MPI send/receive for contiguous data on GM

rather poorly. Overall, the pipelined data streaming protocol is efficient both on GM and VIA. It is almost as good as zero-copy protocol on VIA and much better on GM. Based on these results it is clear that the most competitive protocol on GM is the pipelined data streaming, whereas on VIA a hybrid protocol should be used.

Figures 8-9 present performance of get, put and MPI send/receive operations on the cLAN VIA. We find get operation to be close in performance to MPI with exception for small messages where MPI performs better. The difference is due to the MPI implementation (MVICH) using multiple buffers per virtual interface. This approach minimizes or sometimes even avoids acknowledgment messages that occur in ARMCI implementation as a part of flow control algorithm that aims to minimize the buffer consumption for scalability reasons. In principle, there is no reason not to increase the number of buffers for small processor configurations in ARMCI, but we decided to leave this secondary (not relevant to larger configurations) optimization for later. Performance of the put operation is superior for medium and large requests.

In case of Myrinet, see Figures 10-11, we observe even wider performance advantage for the put operation. Despite targeting a regular unregistered memory and thanks to effective pipelining, contiguous put is able to achieve performance within 5% of that for GM alone for registered memory. The performance gap between strided and contiguous put operation on GM is much wider than on VIA. There are two reasons for that: 1) performance of our memory copy, highly optimized for Pentium-III, is much better for large contiguous messages than multiple small segments in strided format (for data segments larger than 2048 bytes a most efficient version based is enabled), and 2) the Myrinet-2000 is a faster network than cLAN and since it supports much larger messages than cLAN VIA we can use larger buffers that leads to improvement of the overall performance for the pipelined put protocol.

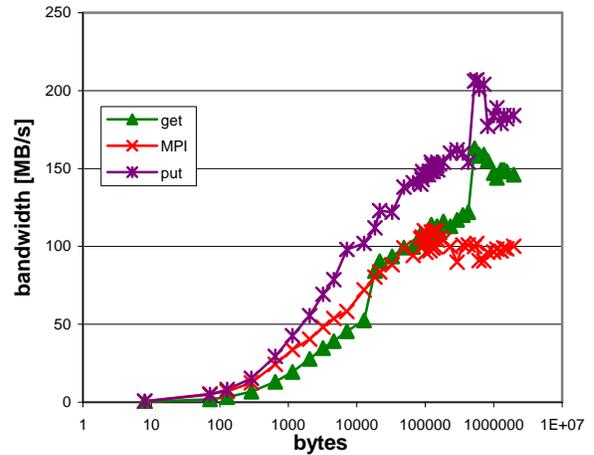


Figure 11: Performance of ARMCI get and put, and MPI send/receive for strided data on GM

Bandwidth of the get operation is also similar to that in MPI send/receive. In analyzing performance of MPI (MPICH-GM) we note that it uses a rather controversial technique of registering user buffers w/o actually deregistering them after completing the data transfer [15]. This is important as the cost of memory deregistration in GM is substantial. We do not feel that this technique is appropriate, at least in the context of remote memory operations, since it could lead to locking most of pages in user application in physical memory or even application failures due to disabling virtual memory and shortage of physical memory. For the put operation on VIA, the limited MTU of ~64KB on cLAN prevents that operation from achieving higher performance by employing larger buffers as done on GM. This limitation could be addressed by employing multiple buffers but at the cost of substantial increase of memory consumption as the multiple buffers would have to be added for each instance of via (remote process) per SMP node.

When comparing performance of contiguous and non-contiguous (strided) remote memory copy operations we find that the gap between them is smaller than under MPI. However, despite the lack of any additional memory copies or other overheads for noncontiguous data, in our case the gap has not been completely eliminated. In large part, this is due to the nonlinear performance of the memory copy operation that is implemented as a combination of three protocols, each of them enabled for certain data sizes. As the contiguous segment sizes are much smaller in the strided than contiguous case, the most efficient copy protocol is only enabled in strided requests starting at and exceeding 0.5MB. The performance difference in the memory copy protocols are shown in Figure 2; however, the log-log scale used in that graph does not fully expose them. Due to packetization effects in pipelined protocols, even larger requests usually depend on a combination of faster and slower memory copy protocols. Switching between the memory copy protocols is a major factor responsible for the performance discontinuities

Table2: Latency of remote memory operations

	SMP	Myrinet	cLAN
get	0.38 μ S	35.4 μ S	34.6 μ S
lock	1.1 μ S	35.6 μ S	35.2 μ S
unlock	0.8 μ S	1.9 μ S	1.6 μ S
RMW	0.8 μ S	41.3 μ S	35.2 μ S

(jaggedness in the graphs) in all our pipelined protocols.

The latency of remote memory operations (measured as transfer time for 8-byte data) is presented in Table 2. We also include performance within the SMP node where the shared memory protocols are used. The performance numbers for lock (uncontested mutex) and read-modify-write operations are similar to the cost of get operation that includes the cost of interrupt on the remote server process. The latency of unlock operation on Myrinet and cLAN only includes the cost to issue a request by the client as the full cost does not appear on the critical path and is hard to measure (is lower than for lock).

Figure 12 shows performance improvements due to concurrent processing in ARMCI_AllFence when called by a single and all processes in a parallel program. Up to three, five, and seven outstanding fence requests are issued to multiple servers before calling process waits for response. As the amount of processing on servers is proportional to the number of clients calling the operation, the improvement is lower for all processes making the call.

4. Related work

Remote memory copy, a subset of remote memory operations discussed, has been developed for clusters in the context of Active Message [1] and Fast Message [16] projects before. The adaptive data streaming protocols described in this paper offer an improvement over the protocols for put and get that were developed in

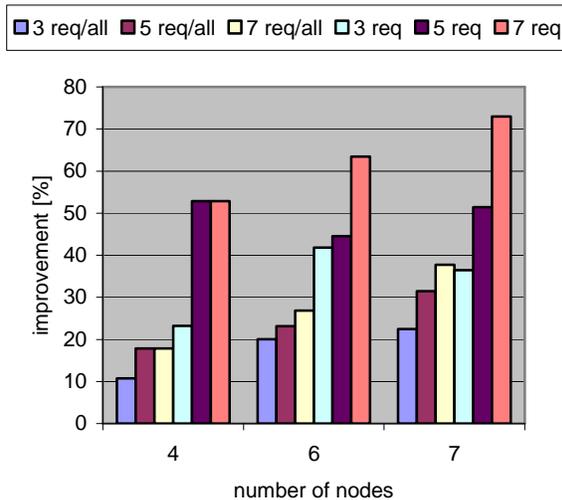


Figure 12: Performance improvement in ARMCI_Allfence for single and all processes calling

our previous work in the context of Myrinet-based clusters [14].

Performance of our implementation of remote memory copy operations (put/get) can be compared to results reported in other papers. In [12], performance of MPI-2 put and get operations is presented on a Linux cluster with cLAN. For example, for time for get operation with required fence operation (to complete the nonblocking get call in the MPI-2 active target model) for 1024 bytes was 262 μ S, and for 131072 was 3664.01 μ S. In our case the timings for get (ARMCI get is fully blocking thus fence operation is irrelevant) were 54.8 μ S and 12800 μ S.

The performance of ARMCI/GM also can be compared to the results of the HPVM/FM implementation of SHMEM [16]. On a dual CPU node with older Myrinet LANAi 7.3 and under Windows NT, 67MB/s bandwidth was achieved in `shmem_get`, and 70 MB/s in `shmem_put`. On the same generation of Myrinet the corresponding numbers for ARMCI under Linux are: 77 MB/s for get and 95 MB/s for put. HPVM exploited 1) the FM support for one-sided communication on the NIC (custom MCP) and 2) a dedicated CPU devoted to active polling rather than blocking like in our approach. ARMCI approach works with standard GM communication layer optimized for two sided protocols and does not require dedicating separate processor for handling communication [16]. The interrupt processing in ARMCI is responsible for the latency being two times higher than in the HPVM approach. However, our experience with applications [14] does not justify dedicating a CPU to further reduce the latency. However, replacing interrupt processing with polling on a dedicated CPU is straightforward to accomplish in the described software architecture. In practice, it only requires replacing blocking receive operation in the server thread by a polling version of that operation (available in GM and VIA).

5. Conclusions and Future Plans

This paper describes a software architecture, protocols and optimization strategies for implementing a full set of remote memory operations including get, put, accumulate, locks, fence, and atomic read-modify-write on SMP clusters that employ Myrinet or VIA-based networks. They deliver high performance and match capabilities of network protocols with requirements of remote memory operations and management efficiently network and host resources. Limitations of the underlying network protocols are presented along with techniques for overcoming them such as the adaptive pipelined data streaming and dynamic memory registration. Regarding the capabilities of network protocols, we found that the requirement for memory registration is the most significant obstacle to overcome. It would be useful to have memory registration interfaces able to handle disjoint memory areas that occur for noncontiguous data transfers. Implementation of remote memory operations would be greatly

simplified if NIC was able to handle unregistered memory. Our future plans include development of nonblocking versions of remote memory operations and a new fence operation. We will also investigate and adopt ARMCI for emerging Infiniband networks.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy (DOE) at Pacific Northwest National Laboratory (PNNL) and at Ohio State University. PNNL is operated for DOE by Battelle Memorial Institute. This work was supported by the Center for Programming Models for Scalable Parallel Computing and DoE-2000 ACTS project, both sponsored by the Mathematical, Information, and Computational Science Division of DOE's Office of Computational and Technology Research. The Molecular Science Computing Facility at PNNL and University of Buffalo provided the high-performance computational resources for this work.

References

1. T. von Eicken, D.E. Culler, S.C. Goldstein, K.E. Schauer. Active Messages: a Mechanism for Integrated Communication and Computation. 19th Int. Symp on Computer Architecture. 1992
2. H. Tezuka, A. Hori, Y. Ishikawa, M. Sato, PM: An operating system coordinated high performance communication library, High Performance Computing and Networking, Springer LNCS 1225, 1997.
3. Loïc Prylli and Bernard Tourancheau. BIP: a new protocol designed for high performance networking on Myrinet. In Workshop PC-NOW, IPPS/SPDP98, 1998.
4. S. Parkin, M. Luria, A. Chien, et. al., High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. 8th SIAM Conf. Parallel Processing for Scientific Computing (PP97); 1997.
5. Myricom, The GM Message Passing System, 10/16/1999.
6. J. Nieplocha, B. Carpenter, ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems, Proc. RTSPP IPPS/SDP'99, 1999.
7. K. Parzysek, J. Nieplocha and R. Kendall, A Generalized Portable SHMEM Library for High Performance Computing, Proc PDCS-2000, 2000.
8. R. Bariuso, Allan Knies, SHMEM's User's Guide, Eagan, MN; Cray Research, Inc., SN-2516, 1994.
9. J. Nieplocha, J. Ju, T.P. Straatsma, A multiprotocol communication support for the global address space programming model on the IBM SP, Proc. EuroPar-2000, Springer Verlag LNCS-1900, 2000.
10. D. Perkovic and P. J. Keleher. Responsiveness without Interrupts, The 13th International Conference on Supercomputing, June 1999.
11. J. Hsieh, T. Leng, V. Mashayekhi, R. Rooholamini, Architectural and Performance Evaluation of GigaNet and Myrinet Interconnects on Clusters of Small-Scale SMP Servers, Proc. SC2000. 2000.
12. M. Golebiewski, J. L. Träff. MPI-2 One-sided Communications on a Giganet SMP Cluster. In Recent Advances in Parallel Virtual Machine and Message Passing Interface. 8th European PVM/MPI Users' Group Meeting, vol 2131 of LNCS, 2001..
13. C. Wagner, F. Mueller, Token-based read/write locks for distributed mutual exclusion, Proc. EuroPar-2000, LNCS 1900. 2000.
14. J. Nieplocha, J. Ju, E. Apra, One sided communication on SMP clusters with Myrinet using the GM message-passing library, Proc CAC'01/IPDPS'01, 2001.
15. Myricom, Portable MPI Model Implementation over GM, ver 1.2.1, (file README-GM), June 23, 2000.
16. L. A. Giannini, A. Chien, A Software Architecture for Global Address space on communication on Clusters: Put/Get on Fast Messages, 7th Int. IEEE Symp on High Performance Distributed Computing, HPDC-7, 1998.
17. InfiniBand Trade Association. Infiniband trade association home page. <http://www.infinibandta.org>.
18. J. Nieplocha, RJ Harrison, and RJ Littlefield, Global Arrays: A portable `shared-memory' programming model for distributed memory computers. Proc. Supercomputing'94, pages 340-349, 1994.
19. R. Numrich, J.K. Reid, Co-Array Fortran for parallel programming. ACM Fortran Forum, 17(2):1-31, 1998.
20. W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Tech Report CCS-TR-99-157, Center for Computing Sciences, 1999.