Optimizing Mechanisms for Latency Tolerance in Remote Memory Access Communication on Clusters

J. Nieplocha V. Tipparaju M. Krishnan
{j_nieplocha,vinod.tipparaju, manojkumar.krishnan}@pnl.gov
Pacific Northwest National Laboratory

G. Santhanaraman D.K. Panda
{gopal,panda}@cis.ohio-state.edu
Ohio State University

This paper describes the design and implementation of mechanisms for latency tolerance in the remote memory access communication on clusters equipped with high-performance networks such as Myrinet. It discusses strategies that bridge the gap between user-level requirements and network-specific communication interfaces while attempting to increase opportunities for latency hiding. Mechanisms for overlapping communication with computation and coalescing small messages (trading latency for bandwidth) are explored. The effectiveness of these techniques is evaluated using microbenchmarks and application kernels including the NAS parallel benchmark suite. The microbenchmark results showed a better degree of overlap for nonblocking operations in ARMCI as compared to MPI. Application results showed up 30% to 45% improvement over MPI on using nonblocking operations. The aggregation of small messages yielded performance improvement of up to 78% over non-aggregated communication.

1. Introduction

Despite the impressive progress in high performance interconnect technology achieved during the last decade, the gap between processor and interprocessor communication performance (especially with respect to latency) has been growing. For example, in 1990 on the NCUBE/2 massively parallel system employing a 1MFLOP/s processor, the message-passing latency was 80µs. Today, the 1GHz Itanium-2 processor is rated at 4GFLOP/s and is employed in Linux clusters connected with networks (e.g., Myrinet) that support ~10µs latency at the MPI layer. This growing gap is not specific to the commodity clusters. For example, the Cray X1 processor is rated at 12.8GFLOP/s (MSP mode), while the MPI latency is roughly the same as on the Pentium-4 based Linux clusters with Myrinet. Therefore, the growing gap between CPU and communication latency is a fundamental problem that requires attention in the design of all layers of communication protocol stacks as well as scalable parallel algorithms. Only by combining quality implementation of the communication interfaces with algorithms capable of exploiting available mechanisms for latency tolerance we can hope to address this issue.

Remote memory access (RMA) operations facilitate 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. On many modern platforms, RMA is directly supported by hardware and is the lowest-level and often most efficient communication paradigm available. In the context of this model, latency hiding can be accomplished through different techniques, including overlapping communication with computation [11] by the use of nonblocking communication (e.g., [5, 17]). Another technique is coalescing small put/get messages [13] into larger ones to eliminate startup cost [14] for as many messages as possible and to improve network utilization.

We are working on advancing Aggregate Remote Memory Copy Interface (ARMCI), a portable RMA library used as a part of the run-time system developed by the Center for Programming Models for Scalable Parallel Computing project (www.pmodels.org) sponsored by the U.S. Department of Energy. The current goal is to provide efficient communication capabilities that could be used for latency hiding and reducing communication overhead in language- and library- based programming models. The major contributions of this paper are 1) the design of efficient nonblocking RMA implementation allowing a high level of overlap between communication and computation; 2) the concept and development of aggregate handle interfaces for coalescing multiple small RMA messages; and 3) demonstration of effectiveness of the nonblocking communication and aggregation in the context of microbenchmarks and application kernels. For both nonblocking communication and aggregation, we investigate various design issues with respect to request handle data structure and management and the buffer management layers. For each of these issues, we analyze the trade-off between performance advantages while exploiting native communication protocols, opportunities for overlapping communication and computations, and portability across different platforms. Based on these trade-offs, we present efficient designs and implementation strategies for the new mechanisms.

The effectiveness of these mechanisms is evaluated across different platforms for microbenchmarks and application kernels. For example, the experimental results on Myrinet demonstrate that nonblocking operations in ARMCI offer a substantially better degree of overlapping communication with computation than MPI. The NAS MG benchmark using nonblocking ARMCI operations achieved 30% to 45% (class C) improvement over the reference MPI implementation. In addition, the aggregated handle nonblocking communication when incorporated into the sparse matrix vector multiplication improved performance by 78% over the version based on non-aggregated get communication on 32 Itanium-2 processors connected with Myrinet.

The paper is organized as follows: Section 2 describes the nonblocking and aggregated RMA functionality that is instrumental for reducing communication overhead. Section 3 discusses issues involved in designing portable and efficient implementation of these capabilities on modern networks. Section 4 presents experimental results that evaluate effectiveness of our design in the context of microbenchmarks as well as application kernels. Finally, conclusions are given in Section 5.

2. Nonblocking and Aggregated RMA Communication

Aggregate Remote Memory Copy Interface (ARMCI) [3] is a portable RMA communication library compatible with message-passing libraries such as MPI or PVM. It has been used for implementing distributed array libraries such as Global Arrays, other communication libraries such as Generalized Portable SHMEM [15], and compiler run-time systems such as PCRC Adlib [3] or the portable Co-Array Fortran compiler at Rice University. ARMCI offers an extensive set of functionality in the area of RMA communication: 1) data transfer operations; 2) atomic operations; 3) memory management and synchronization operations; and 4) locks. In scientific computing, applications often require transfers of noncontiguous data that corresponds to fragments of multidimensional arrays, sparse matrices, or other more complex data structures. With remote memory communication APIs that support only contiguous data transfers, it is necessary to transfer noncontiguous data using multiple communication operations. This often leads to inefficient network utilization and involves increased overhead. ARMCI, however, offers explicit noncontiguous data interfaces: strided and generalized I/O vector that allow description of the data layout so that it could, in principle, be transferred in a single message. Of course, the effectiveness of actual transfers depends on the ability of underlying networks to deal with noncontiguous data (e.g., scatter/gather operations). However, even when scatter/gather operations are not supported by the network, the ARMCI strided and vector operations take advantage of the information -- for example, at level of data packing/unpacking -- so that the overall number of messages and network packets is reduced. Although the explicit message aggregation accomplished through the use of strided and vector interfaces is an effective mechanism for reducing communication overhead, it does not exploit all the available opportunities for optimization. Our work focuses on developing techniques that could help for latency tolerance -- nonblocking RMA and implicit communication aggregation.

2.1 Nonblocking Operations

Nonblocking operations initiate a communication call and then return control to the application. The user who wishes to exploit nonblocking communication as a technique for latency hiding by overlapping communication with computation implicitly assumes that progress in communication can be made in a purely computational phase of the program execution when no communication calls are made. Unfortunately, that assumption is often not satisfied in practice -- the availability of nonblocking API does not guarantee that overlapping communication with computation is always possible [6]. Because the RMA model is simpler than MPI (e.g., does not involve message tag matching or dealing with early arrival of messages), in principle more opportunities for overlapping communication with computation are available. However, we found that these opportunities are not automatically exploited by deriving implementations of nonblocking APIs from their blocking counterparts. For example, the communication protocols used to optimize blocking transfers of data from non-registered memory by pipelined copy and network communication through a set of registered memory buffers [1] can achieve very good performance by tuning the message fragmentation in the pipeline [8]. However, the memory copy requires the active host CPU involvement and therefore reduces the potential for effective overlapping communication with computation. To increase the overlap, we expanded the use of direct (zero-copy) protocols on networks that require memory registration, such as Myrinet.

In ARMCI, a return from a nonblocking operation call indicates a mere initiation of the data transfer process, and the operation can be completed locally by making a call to the wait routine. Waiting on a nonblocking put or an accumulate operation ensures that data was injected into the network and the user buffer can be now be reused. Completing a get operation ensures that data has arrived into the user memory and is ready for use. A wait operation ensures only local completion. The library imposes a limit on the number of outstanding requests allowed (if necessary, it can transparently complete an old request and free up the resources for a new request).

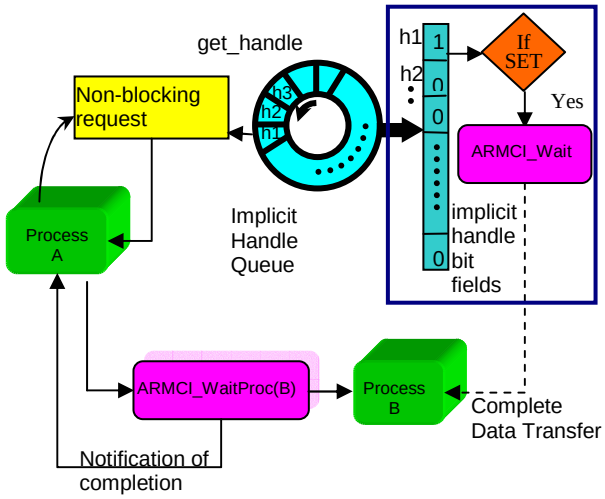

Figure 1: Nonblocking transfer with implicit handle

For performance reasons [12], ARMCI supports only a weak consistency for operations targeting remote memory. Unlike their blocking counterparts, the nonblocking operations are not ordered with respect to the destination. Performance is one reason; the other is that by ensuring ordering, we incur additional and possibly unnecessary overhead on applications that do not require ordered operations. When necessary, ordering can be done by calling a fence operation. The fence operation is provided to the user to confirm remote completion if needed.

2.2 Request handle

The request handle structure is central to the APIs associated with the latency hiding mechanisms in ARMCI. This opaque object is stored in the application memory and is used to 1) assign a unique identity to a nonblocking RMA operation, 2) facilitate aggregation of multiple operations, and 3) optionally store certain control information. Before the handle is used, it must be initialized with the ARMCI_INIT_HANDLE macro and can be reused after the associated nonblocking operation completes. The user passes a reference to a request handle structure. As a convenience to the user, a NULL value for the handle address can be specified. The library keeps track of these so-called "implicit handle requests" and assigns a handle to them from an internal pool of handles. This type of requests can be completed using either the wait operation associated with a particular remote processor (see Figure 1) or another wait operation to complete all pending implicit handle requests.

2.3 Implicit and Explicit Aggregation

Aggregation of requests is another mechanism for improving latency tolerance. Multiple nonblocking data transfer (put/get) requests can be aggregated into a single data transfer operation in order to improve the data transfer rate. Especially if there are multiple data transfer requests of small message sizes, aggregating those requests into a single large request reduces the latency, thus improving performance. This technique is unique in its ability to sustain high bandwidth utilization and enables high throughput. Each of these requests can be of a different size and independent of data type. The aggregate data transfer operation is independent also of the type of put/get operation; that is, it can be a combination of regular, strided, or vector put/get operations. There are two types of aggregation available: 1) explicit aggregation, where the multiple requests are combined by the user through the use of the strided or generalized I/O vector data descriptor, and 2) implicit aggregation, where the combining of individual requests is performed by ARMCI. The implicit aggregation involves the nonblocking request handle that is marked as "aggregate handle" using the ARMCI_SET_AGGREGATE_HANDLE macro.

Users can rely on a single aggregate handle to represent multiple requests. Any number of operations to/from the same processor can use the same aggregate handle. A wait on such a handle completes all the aggregated requests. For multiple small sends, aggregating is usually much faster and gives better performance. Figure 2 illustrates the aggregate data transfer. It shows that the descriptors of multiple put requests are stored in an aggregate buffer and, once the wait call is issued, the data transfer is completed.

3. Design and Implementation Approach

Designing a portable RMA communication layer involves addressing multiple issues: 1) the functionality must be implementable across a wide variety of platforms; 2) performance advantages of the native communication protocols must be exploited; 3) opportunities for

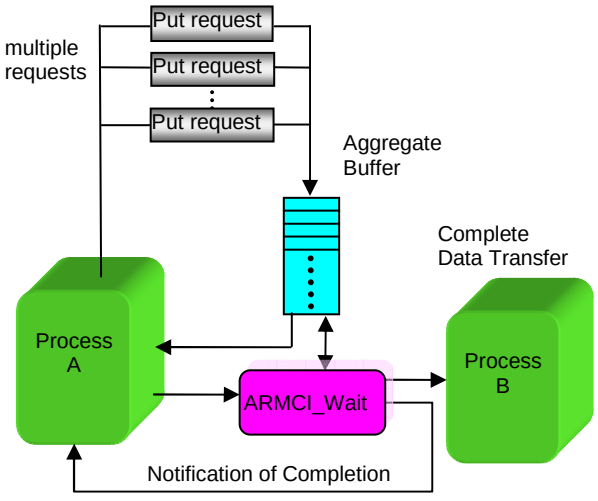

Figure 2: Implicit aggregate data transfer

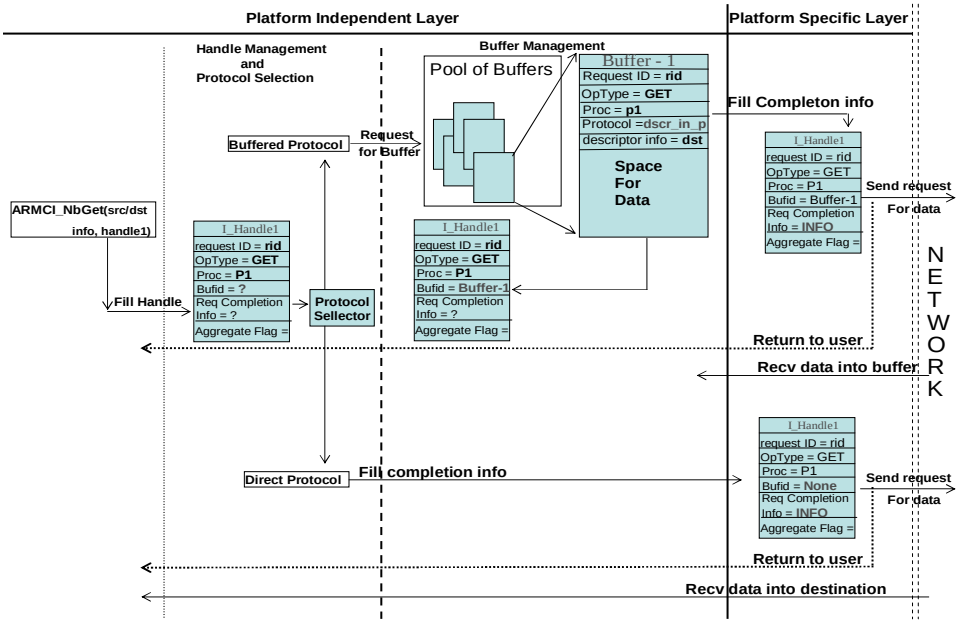

Figure 3: Nonblocking request transition

overlapping communication and computations should be provided; and 4) as much of the code as possible must be shared to minimize the maintenance efforts across different platforms. On networks like the IBM SP interconnect and Quadrics, the underlying RMA layer provides most of the required capabilities. Hence, on these systems, most of the nonblocking calls can be implemented as thin wrappers to the native protocols. We are referring to these protocols as *direct*. In the case of some networks, direct protocols are zero-copy (GM, VIA, Quadrics Elan), but others where the native communication interface involves copying the data (IBM LAPI) internally are not. Some networks like GM, VIA, and Infiniband require data to be transmitted from/to special memory. This can be accomplished either by 1) copying the data into a set of special registered/pinned buffers for transmission; 2) allocating registered memory for the user; or 3) by on-demand registration of the user's memory. ARMCI uses all three schemes, depending on the platform, operation type, or size of the data transfer. Protocols that use memory copy scheme are referred to as *buffered*. Although the goal is to generalize most of the design, doing so should not adversely affect the performance in cases where an underling network provides direct support.

Multiple requirements can be satisfied by a buffer management layer. First, on networks that allow data transfers between registered buffers, the data can be copied in, sent, received, and copied out from the internal set of buffers allocated in registered memory. In this manner, data can be transferred between nonregistered memory locations. Note that on-demand memory registration of user buffers might not always be available or can be very costly (e.g. GM) [1], [16]. Second, buffers are useful for packing/unpacking noncontiguous data transfers when the underlying network has support only for contiguous data transfers (for example, GM) [1]. Third, in the case of nonblocking communication, data descriptors might be required to be stored in persistent memory. For example, the data descriptors in the I/O vector format for IBM LAPI vector interfaces can be very large, and they must be saved in persistent memory until the request is completed.

One of the design goals is to make most of the handle management code and buffer management code platform-independent, thus making the architecture portable while avoid the unnecessary overhead. This is accomplished, as seen in Figure 3, by switching to a direct protocol when possible at the very beginning of the request processing. Interaction between the platform-independent layer and platform/network-specific layer is only to either inject the data into the network or check for the completion of an operation.

3.1 Handle Management

Every nonblocking call is associated with a nonblocking request handle. For explicit handle nonblocking calls and aggregate handle nonblocking calls, this handle is passed by the user as a parameter. An implicit handle call is associated with a handle from a static list of handles, maintained internally. The handle provided by the user is internally mapped to a data structure that in turn carries all the information required to identify and complete, or test completion of a nonblocking operation.

Because a common handle is used to represent a request on all platforms, for portability reasons it stores only the most generic information, including unique identifier of the request, the type of operation, and the remote processor number. Other fields include completion information required by the underlying network for request completion. For example, on IBM SP when using LAPI as the communication protocol, "Req completion information" field carries a LAPI counter that is updated by LAPI on request completion. For buffered client-server protocols [1] or protocols based on Active Messages, "bufid" field in the handle carries the identifier of the buffer used for this request. This field can be used also to indicate use of multiple buffers.

3.2 Communication buffers

The communication buffer is represented by a data structure that stores information about the associated request. In nonblocking operations, it also carries a unique request identifier for the request, see Figure 3. For the buffered implementation of the get operation, it stores the destination address for the data. For strided and vector operations, the destination information is represented by a more complex descriptor of variable size. The buffer data structure has a fixed space allocated to store destination data descriptors. For a larger descriptor, extra memory is allocated, and the corresponding address is stored in the buffer. That memory is freed when the operation associated with this buffer is completed. The "protocol" field in the buffer structure carries more detailed information. For example, the "protocol" field in the buffer management phase of Figure 4 carries the value "sdescr_in_p", which indicates that this buffer is being used for a strided data transfer and the destination data descriptor is in place (sdescr_in_p) inside the buffer data structure. This information is needed to complete a request.

ARMCI does not impose a limit on the number of outstanding operations. Hence, when the buffer management layer runs out of buffers, it completes an old request associated with a buffer currently in use to free a buffer. Because a request can be using more than one buffer, freeing a buffer might complete only a part of the request. A communication buffer is also freed as a part of the wait operation on the request using that buffer.

3.3 Nonblocking request processing

The goal of the design was to make the algorithms for operation processing as generic as possible without compromising the performance. In the first step, a *protocol selector* (Figure 3) decides which protocol to use for that particular request. If the protocol to be used is direct, then the request is sent via a thin wrapper directly to the platform-dependent code and is thereby injected into the network without any additional overhead. For example, the protocol selector for GM checks if the source and destination memories used in that request are in registered memory. If they are, then the protocol selector indicates that the request is a direct request, and it is thus sent directly to the platform-specific layer. The platform-specific layer, after updating the request completion information, injects the request in to the network. Similarly, with LAPI, depending on the operation and the size of data involved, the protocol selector decides if the request will go via the active message/buffered protocol or the direct protocol. The request handle passed by the user carries no information. Information is filled in and updated as the handle transits through the various phases of protocol processing, as shown in Figure 3. If the request needs a buffer, the information in the request handle is passed to the buffer management layer, where the information about the buffer associated with the request is recorded in the handle.

If there are no free buffers available as described in Section 3.2, a request associated with the least recently used buffer from the buffer pool is completed, and the buffer is reassigned. The buffer management phase fills the buffer with the information obtained from the request

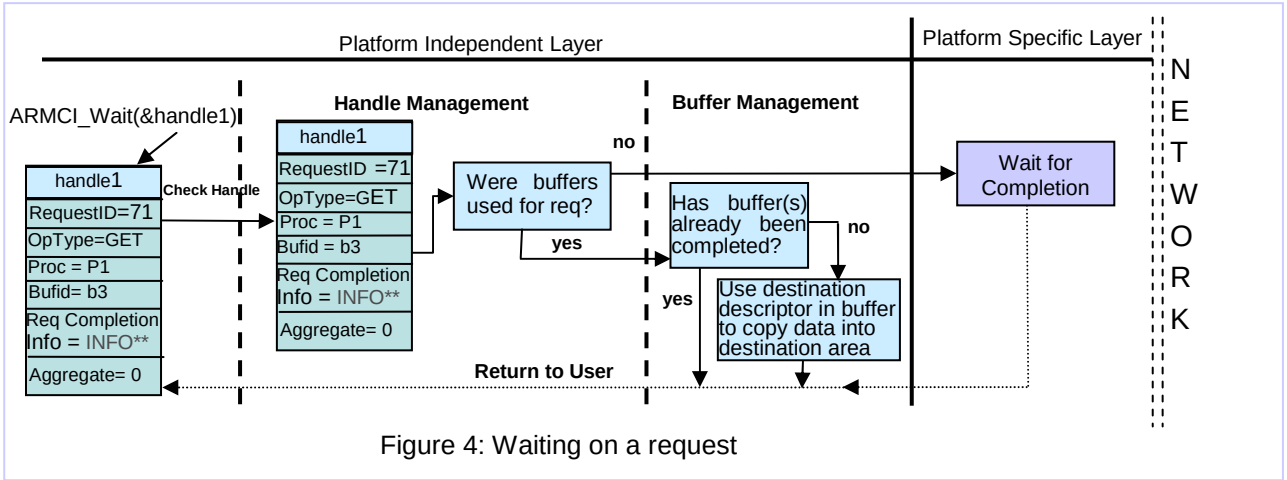

Figure 4: Waiting on a request

handle (e.g., request identifier) for this request. Correspondingly, the identifier of the buffer allocated for this request is filled in the request handle to represent the association between a request handle and a buffer used for it. When a wait operation is called on a request handle, the associated buffer(s) is identified.

The request completion information in the handle is also updated in the platform-specific layer. For direct protocol, the request bypasses the buffer management and directly transitions into the platform-specific layer, where the platform-specific request information inside the handle is updated. The control returns to the user program once the platform-specific layer issues the request.

3.4 Waiting on a request

The wait on a request handle completes the request. Whether the request used buffers or not can be determined by looking at the value stored in the bufid field of the request handle. In Figure 4, this is shown in the handle management phase. For the direct protocol, the platform-specific layer verifies request completion based on the information it stored in the "Req completion info" field. If buffers were used for the request (buffered protocol or for storing a data descriptor), then the buffer management layer checks to see if the buffers used for this request were completed already as a part of freeing resources. If they have not yet been completed, then the data from the buffer is copied into the appropriate destination based on the destination descriptor information stored in the buffer. To be able to verify if the data has already arrived in the buffer, the buffer management layer may check for data arrival via the platform-specific layer.

3.5 Aggregation

The implicit aggregation of data transfers is implemented using the generalized I/O vector operations available in ARMCI [3]. This interface enables the representation of a data transfer as a combination of multiple sets of equally sized contiguous data segments. When the first call involving aggregate nonblocking handle is executed, the library starts building a vector descriptor stored in one of the preallocated internal buffers. The actual data transfer takes place when the user calls wait operation or the buffer storing the vector descriptor fills up.

3.6 Optimizing Overhead and Overlap

As discussed in Section 2, the overhead introduced due to the additional processing and resource management incurred by a nonblocking call should be minimized. In our implementation, this goal is achieved in multiple ways:

- Before returning, all nonblocking operations always initiate data transfer so that the network interface card (NIC) can process a request while the host CPU is available to carry out the computations.
- When a nonblocking GET operation returns, either the buffered or direct protocols ensure that all the requested data will be received without explicit involvement of the host CPU. In the buffered protocol, the request is broken into pieces that fit the available buffer space. For very large buffered requests, some initial portion of the data might be received before the nonblocking operation returns.
- The direct protocol is switched to when possible, as described in section 3.3.
- The platform-specific protocols that involve extensive blocking time are avoided. For example, on the IBM SP for larger messages, the nonblocking vector get operation `LAPI_Getv` blocks for up to 90% of the data transfer time. Therefore, this operation is used by ARMCI for only the blocking operations. The nonblocking operations that need vector or strided format are implemented by executing `LAPI_Putv` (vector put) from the active message handler so that the nonblocking call returns to the application in the shortest time possible.

4. Performance Evaluation

The primary platform for the experiments was a Linux cluster with dual 2.4GHz Pentium-4 nodes and Myrinet-2000 (M3F-PCI64C-2 Myrinet interface) located at the State University of New York at Buffalo. It employs the most recent versions of GM (1.6.4) and MPICH-GM libraries provided by Myricom. The experiments included several microbenchmarks to evaluate different parameters of the communication operations. In addition, two of the NAS benchmarks and sparse matrix vector multiplication code were used to determine the effectiveness of nonblocking and aggregated communication in the application contexts.

4.1 Microbenchmarks

The motivation for the experiments described in this section was to demonstrate the performance of the implementation at the system level. The next section shows how much of these gains can be leveraged at the application level. Experiments discussed in the current section have been conducted for the nonblocking get operation since they explicitly demonstrate the overhead and overlap factors.

Overhead test

The first experiment demonstrates the efficiency of the implementation as compared with a base case GM implementation. For this purpose, a nonblocking operation is simulated at the GM level in the following fashion. The client issues a gm_send_with_callback (with

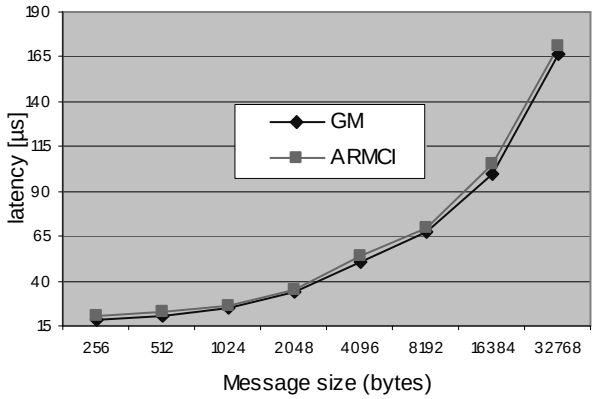

Figure 5: Comparison of latency of ARMCI get operation (nonblocking get followed by Wait) with GM version

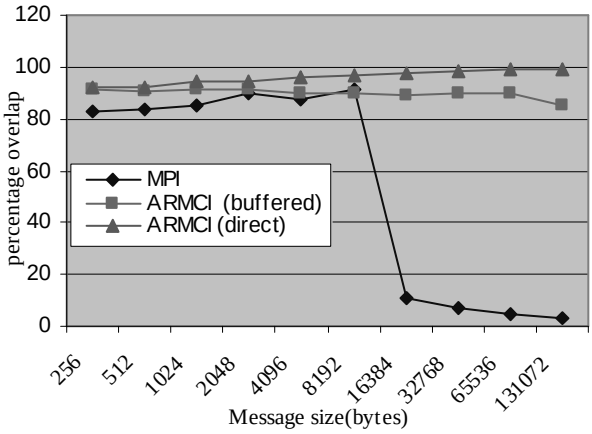

Figure 6: Percentage computation overlap for increasing message sizes for MPI and ARMCI direct and server- based protocols

the details of the required data) and then polls on a flag set when the data reaches this node. On the other end, the server does a GM_receive, processes the request, and issues the RDMA put operation with the data using the gm_directed_send_with_callback function. The ARMCI layer is actually built on this basic scheme to implement the nonblocking get. This experiment tries to evaluate the efficiency of the implementation. Figure 5 shows the latency at the base GM and ARMCI levels. The timings have been averaged over 1000 iterations. They show that the ARMCI layer adds very little overhead to the base level and thus provides a very efficient interface to the applications.

Overlap Test

The second experiment deals with overlapping communication with computation, and it was performed in the context of ARMCI and MPI. In the ARMCI version, the computation is incorporated in the program in the form of a delay. Increasing computation is gradually inserted between the initiating nonblocking get call and the wait completion call. As we keep increasing the computation, at some point the sum of the nonblocking call issue overhead and computation would exceed the idle CPU time, so the total benchmark running time would increase. This point gives us the maximum possible overlap. We performed this experiment on two nodes, with one node issuing the nonblocking get for data located on the other and then waiting for the transfer to be completed in the ARMCI_Wait call. The timings were averaged over 1000 iterations. We have developed versions of this microbenchmark for direct and buffered protocols. We also implemented an MPI version of the above benchmark because our motivation was to compare the overlap in ARMCI and in the MPI nonblocking send/receive operations. In MPI, if the node needs a portion of data from another node, it sends a request and waits on a nonblocking receive for the response. We can overlap the time duration between these two calls with computation. We measured the computation overlap for both the ARMCI and MPI versions of the benchmark, and results are plotted in Figure 6. The percentage overlap is measured as the amount of time of a nonblocking (data transfer) call that can be overlapped with useful computation without increasing the overall benchmark time.

We observe that ARMCI offers a higher level of overlap than MPI. The buffered protocol is able to achieve about 90% overlap. For large messages, this percentage drops because of time involved in copying to the destination buffer. In the direct protocol, we are able to overlap almost the entire time (greater than 99%). The exception (1%) was the time involved in issuing the nonblocking get. The MPI version does reasonably well up to message size 16kb. At 16kb and beyond, the MPI implementation switches to the rendezvous protocol. This has a serious impact on the computation overlap because the handshake involved in the protocol occurs in MPI_Wait. Consequently, the only part that can be overlapped is till the receipt of 'request to send' and not until the actual data transfer is completed. Several studies have tried to analyze and benchmark the MPI overlap. Paper [7] describes some experiments to analyze and test the overlap for basic asynchronous MPI calls for MPI implementations on different platforms. Another related paper [8] provides a portable benchmark suite for assessing overlap. One of the methods relies on polling to advance progress, and they use another metric ('availability') for their assessment. Our benchmark does not introduce additional MPI library calls for making progress, looks for plain overlap performance gains, and is probably more representative of how real applications use nonblocking communication.

Aggregation Test

A simple benchmark test was written to compare the performance of regular, vector, and implicit aggregate

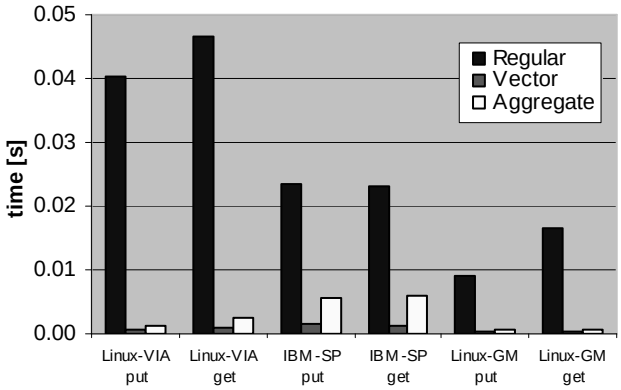

Figure 7: Performance of aggregate, vector, and regular (multiple calls) operations in transfer of 1000 doubles

put/get operations. In the test, 1000 values of type double- located in nonadjacent memory locations were transferred using 1000 calls to regular put and get calls, with and without enabling aggregation. Also, 1000 doubles were stored in a vector descriptor and transferred in a single vector operation to show that aggregation of 1000 calls is as good as a single vector call. The experiments were performed on the 500MHz Pentium-III Linux cluster with Giganet cLAN (VIA) at PNNL, a Power-3 IBM-SP at NERSC, and a 2.4GHz Pentium-4 Linux cluster with Myrinet at SUNY Buffalo. Figure 7 shows that enabling aggregation significantly outperforms the regular put/get operation. The figure also indicates that aggregating multiple data transfer requests performs almost as well as a *single* vector operation.

4.2 Sparse Matrix-Vector Multiplication

Sparse matrix-vector multiplication is one of the common computational kernels, for example in solving linear systems using conjugate gradient method. It is described as $Ax = b$, where A is an nxn nonsingular sparse matrix, b is an n-dimensional vector, and x is an n-dimensional vector of unknowns. In this benchmark, one of the sparse matrices (Figure 8a) from the Harwell-Boeing collection is used [9] to test the matrix-vector multiplication. The sparse matrix size is 41092 and has 1683902 (~.1%) non-zero elements. The experiments were conducted on the Linux cluster (dual node, 1GHz Itanium-2, Myrinet-2000 interconnect) at PNNL. Sparse matrix-vector multiplication was done with aggregation enabled and disabled. The sparse matrix and the vector are distributed among processors. Instead of gathering the entire vector, each process caches the vector elements corresponding to the non-zero element columns of its locally owned part of the matrix. When aggregation is enabled, all the *get* calls corresponding to a single processor are aggregated into a single request, thus reducing the overall latency and improving the data transfer rate. Figure 8b shows that aggregation outperforms the regular put/get version of the code in all cases and scales well. Figure 9 provides

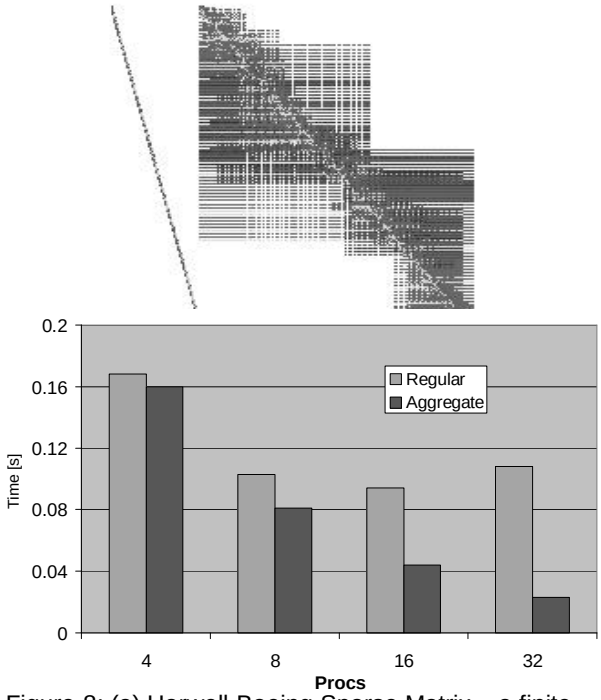

Figure-8: (a) Harwell-Boeing Sparse Matrix – a finite element problem. (b) Sparse matrix-vector

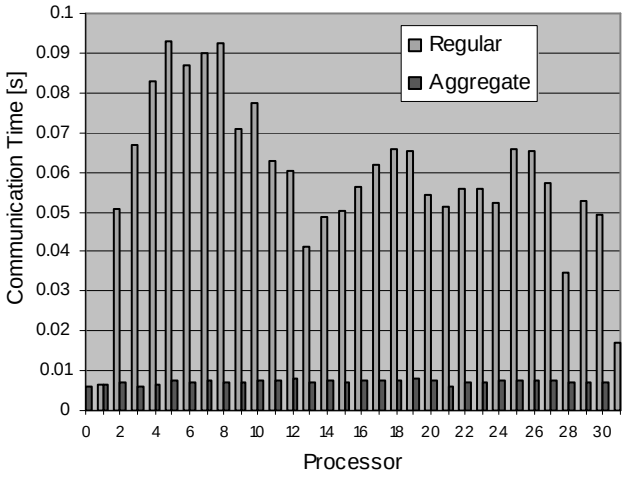

Figure 9: Time spent on communication on each processor in the sparse matrix-vector multiplication in the 32-processor case

explanation of the performance gaps resulting from the large and nonuniform communication overhead effectively addressed by aggregation.

4.3 NAS benchmarks

The Numerical Aerodynamic Simulation (NAS) parallel benchmarks (NPB) are a set of programs designed at NASA. Our starting point was NPB 2.3 [4]

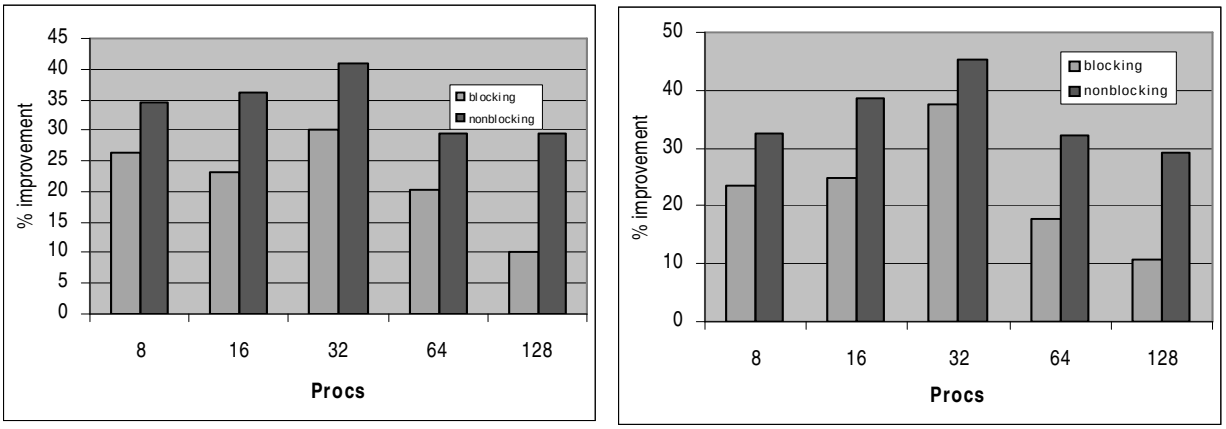

Figure 10: Performance improvement in NAS MG for class B (left) and class C (right)

implementation written in MPI and distributed by NASA. We modified two of the five NAS kernels, MultiGrid (MG) and Conjugate Gradient (CG), to replace point-to-point blocking and nonblocking message-passing communication calls with first blocking and then nonblocking RMA communication. This is just a mere replacement of the point-to-point message passing communications part of the current message-passing version of CG and MG NAS kernels using ARMCI RMA blocking and nonblocking operations [18]. Other benchmarks (e.g., FFT, IS) rely on collective communication thus limiting appropriateness of RMA (point-to-point) communication without reformulating the underlying mathematical algorithms. We ran our MG tests for classes A, B, and C. They are three production-grade problem sizes for the MG benchmark: Class A (grid 256x256x256, 4 iterations), Class B (grid 256x256x256, 20 iterations), and Class C (grid 512x512x512, 20 iterations).

For Class A, a smaller problem size with the fewest iterations, the ARMCI blocking code outperforms the reference MPI implementation by 7% to 30%. ARMCI nonblocking version achieves an additional improvement of 10% to 23% over the ARMCI blocking implementation and a 28% to 46% improvement over the reference MPI implementation. Most of the improvement achieved over the blocking implementation is just by mere issue of the update in the next dimension while working on the current one. For Class B, with the same problem size as class A but more iterations, ARMCI blocking implementation outperforms MPI by 10% to 37% (see Figure 10 (left)). The ARMCI nonblocking implementation achieves an additional improvement of 5% to 20% over the blocking version and shows a 30% to 45% improvement over the reference MPI implementation. For Class C, the ARMCI blocking implementation outperforms MPI by 10% to 32%. ARMCI nonblocking implementation achieves an additional improvement of 2% to 21% over the blocking implementation and shows a 30% to 40% improvement over MPI. Since coarser levels of multi grid do not carry enough work to hide all the communication, for small processor configurations any improvement achieved by using a nonblocking over blocking API is limited. With an increased processor count for the fixed problem size, the improvement is amplified.

Due to the synchronous nature of data transfers in the CG algorithm, the performance improvement over MPI, although consistent is rather limited (see Figure 11). As expected, the main source of performance improvement is the increased efficiency of RMA operations over the message passing (e.g., due to overheads associated with tag-matching, early message arrival that MPI must do). However, the nonblocking RMA offers an additional performance improvement. For example, for 128 processors, it exceeds 10% over MPI.

7. Summary and Conclusions

This paper describes design and implementation of mechanisms for latency tolerance in the context of remote memory access communication on clusters equipped with high-performance networks. They include nonblocking RMA communication and aggregation of small messages. The design maximizes the potential for overlapping communication with computations and minimizes the overhead while preserving the portability. The experimental results showed that nonblocking operations in ARMCI show a better degree of overlap in comparison to MPI. Application kernels using nonblocking ARMCI operations showed 30%-45% improvement over MPI. In addition, aggregation has been shown to be an effective technique for latency tolerance, giving as much as 78% improvement over non-aggregated communication.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy (DOE) at Pacific Northwest National Laboratory (PNNL) and Ohio State University.

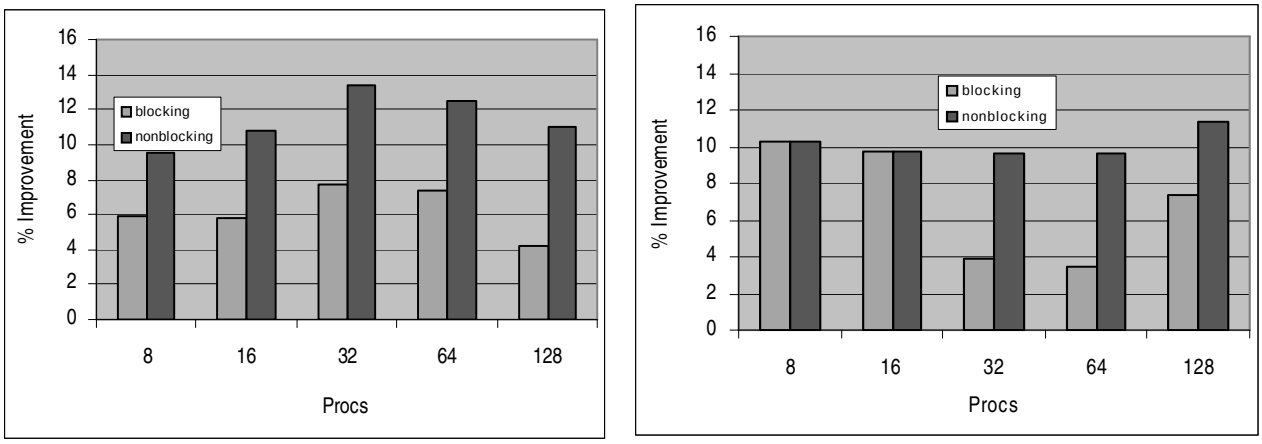

Figure 11: Performance improvement in NAS CG class B (left) and class C (right)

PNNL is operated for DOE by Battelle. This work was supported by the Center for Programming Models for Scalable Parallel Computing project sponsored by the MICS/ASCR program in the DOE Office of Science.

References

1) J. Nieplocha, V. Tipparaju, A. Saify, D. Panda, Protocols and Strategies for Optimizing Remote Memory Operations on Clusters, CAC/IPDPS, 2002.

2) J. Nieplocha, V. Tipparaju, J. Ju, and E. Apra, "One-sided communication on Myrinet", Cluster Computing, 6, 115-124, 2003.

3) J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems", Proc. RTSPP IPPS/SDP, 1999.

4) D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, The NAS parallel benchmarks, RNR-94-007, NASA 1994.

5) S. B. Baden and S. J. Fink, "Communication overlap in multi-tier parallel algorithms", Proceedings of Supercomputing, Orlando, FL, November 1998.

6) J. B. White and S. W. Bova, "Where's the overlap? Overlapping communication and computation in several popular MPI implementations", Proceedings of the Third MPI Developers' and Users' Conference, March 1999.

7) B. Lawry, R. Wilson, A. B. Maccabe, and R. Brightwell, "COMB: A Portable Benchmark Suite for Assessing MPI Overlap", IEEE Cluster, 2002.

8) R. Y. Wang, A. Krishnamurthy, R. P. Martin, T. E. Anderson, and D. E. Culler, Modeling and Optimizing Communication Pipelines, ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 1998.

9) T. Davis, University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices, NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996

10) Afsahi, N. Dimoppolous, Hiding Communication Latency in Reconfigrable Message-Passing Environments, Proc. IPPS/SPDP 1999.

11) V. Strumpen, T. L. Casavant, Exploiting communication latency hiding for parallel network computing: model and analysis", Proc. PDS'94, 1994.

12) S. Kim and A. V. Veidenbaum, The effect of limited network bandwidth and its utilization by latency hiding techniques in large-scale shared memory systems, Proc. PACT'97, 1997.

13) D. Pham and C. Albrecht, "Optimizing Message Aggregation for Parallel Simulation on High Performance Clusters", 7th Intern. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1999.

14) Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick, "An evaluation of current high-performance networks", in Proc. 17th IPDPS, 2003.

15) K. Parzyszek, J. Nieplocha, and R.A. Kendall, "A generalized portable SHMEM library for high performance computing", Proc. PDCS, 2000.

16) Bell and D. Bonachea, "A New DMA Registration Strategy for Pinning-Based High Performance Networks", Proc. CAC'03, 2003.

17) E. Culler, A. Dusseau, S. Goldstein, A. Krishna-murthy, S. Lumetta, T. Eicken, K. Yelick, Parallel programming in Split C, Proc. Supercomputing, 1993.

18) V. Tipparaju, M. Kumar, J. Nieplocha, Santhanaraman, D.K. Panda, Exploiting nonblocking remote memory access communication in scientific benchmarks, Proc. HiPC'03, 2003.