Scaling Linear Algebra Kernels using Remote Memory Access

Manojkumar Krishnan
High Performance Computing
Pacific Northwest National Laboratory
Richland, WA, USA
Email: manoj@pnl.gov

Robert R. Lewis
School of EECS
Washington State University
Richland, WA, USA
Email: bobl@tricity.wsu.edu

Abhinav Vishnu
High Performance Computing
Pacific Northwest National Laboratory
Richland, WA, USA
Email: abhinav.vishnu@pnl.gov

***Abstract*—This paper describes the scalability of linear algebra kernels based on remote memory access approach. The current approach differs from the other linear algebra algorithms by the explicit use of shared memory and remote memory access (RMA) communication rather than message passing. It is suitable for clusters and scalable shared memory systems. The experimental results on large scale systems (Linux-Infiniband cluster, Cray XT) demonstrate consistent performance advantages over ScaLAPACK suite, the leading implementation of parallel linear algebra algorithms used today. For example, on a Cray XT4 for a matrix size of 102400, our RMA-based matrix multiplication achieved over 55 teraflops while ScaLAPACK's `pdgemm` measured close to 42 teraflops on 10000 processes.**

I. Introduction

Linear algebra kernels are among the most important tools in scientific computing. Computational scientists, when feasible, attempt to reformulate the mathematical description of their application in terms of linear algebra operations (e.g. matrix multiplication, solution of linear equations) as they can be performed efficiently on modern architectures. By adopting a variety of techniques such as prefetching or blocking to exploit the characteristics of the memory hierarchy in current architectures, computer vendors have optimized the standard serial linear algebra kernels in the open source Basic Linear Algebra Subroutines (BLAS, [1]) and Linear Algebra PACKage (LAPACK, [2]) packages to deliver performance as close to the peak processor performance as possible.

Optimized parallel linear algebra kernels play a key role in achieving scalability and performance for several scientific applications. However, most parallel linear algebra implementations used in today's high end systems are implemented on top of point-to-point message passing operations such as the Message Passing Interface (MPI, [3]) and collective communications.

RDMA (Remote Direct Memory Access) and shared-memory are the lowest-level (closest to hardware) and highest-performing communication protocols on current high-end systems. MPI implementations use these protocols, however this adds overhead. There is a clear imbalance between advancement in parallel programming model research and the use of message passing in linear algebra kernels.

Earlier researchers targeted their parallel linear algebra implementations (e.g. PBLAS in ScaLAPACK [4]) for massively parallel processor (MPP) architectures on which message passing was the highest-performance and typically the only communication protocol available. In particular, these algorithms relied on optimized broadcasts or send-receive operations.

Current and future architectures differ in several key aspects from the earlier MPP systems. Regardless of the processor architecture, both high-end systems (IBM BlueGene, Cray XT4) and commodity systems (clusters) employ a building block of multi-core (or upcoming manycore) or symmetric multi-processor (SMP) nodes connected with an interconnect network. All of these architectures have hardware support for load/store communication within the underlying SMP nodes. Although high-performance implementations of message passing can exploit shared memory internally, their performance is less competitive than direct loads and stores.

This paper discusses the performance and scalability of two popular parallel linear algebra kernels - matrix multiplication and LU factorization. The conceptual architectural model for which our algorithms were designed is a cluster of multiprocessor nodes connected with a network that supports remote memory access (RMA) communication (a.k.a. the "put/get" model) between the nodes. Our implementation uses the fastest communication model possible (i.e. shared memory within an SMP node and RMA between nodes).

RMA is a simple communication model and, on modern systems, is often the fastest communication protocol available, especially when implemented in hardware as zero-copy RDMA write/read operations (e.g., OpenIB/Infiniband [5], Portals/Seastar [6], DCMF/Bluegene [7]). RMA is often used to implement the point-to-point MPI send/receive calls [8], [9]. To address the historically growing gap between the processor and network speed, our implementation relies on the availability of the nonblocking mode of RMA operation as the primary latency hiding mechanism (through overlapping communication with computation) [10]. In addition, each cluster node is assumed to provide efficient load/store operations that allow direct access to the data.

In other words, a node of the cluster represents a shared memory communication domain. Our algorithm is explicitly aware of the task mapping to shared memory domains i.e., it is written to use shared memory to access parts of the matrix held on processors within the domain of which the

given processor is a part, and nonblocking RMA operations to access parts of the matrix outside of the local shared memory domain (i.e., the RMA domain).

Our RMA-based parallel linear algebra algorithms (`dgemm` and LU factorization) achieved consistent and very competitive performance on two large scale systems (Infiniband Linux cluster and Cray XT4). They are more general and memory efficient, and they demonstrated excellent performance and scalability on both platforms. For example, on a Cray XT4 with 10000 processes and a matrix size of 102400, our RMA-based matrix multiplication achieved over 55 teraflops, while ScaLAPACK's `pdgemm` measured close to 42 teraflops.

The paper is organized as follows. The next section describes our RMA approach using ARMCI. Section 3 presents the design and implementation of our RMA-based linear algebra algorithms for clusters and shared memory systems. Section 4 describes and analyzes performance results for the RMA-based algorithms and ScaLAPACK. Finally, summary and conclusions are given in Section 5.

II. Remote Memory Access - ARMCI Approach

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 [11]. On many modern platforms, RMA is directly supported by hardware and is the lowest level and often most efficient communication paradigm available [12]. An important difference over the MPI-1 message-passing model is that RMA does not require an explicit receive operation and thus offers increased asynchrony of data transfers (see Figure 1).

In our implementations, we used a portable RMA interface called Aggregate Remote Memory Copy Interface (ARMCI) [13] for communication and Global Arrays (GA) [14] for providing global views of distributed arrays. ARMCI can be used for developing applications and for creating a communication layer for libraries and compiler runtime systems, especially for the global address space languages/libraries (e.g., Global Arrays).

MPI-2 RMA [15] offers a model closely aligned with traditional message passing (MPI-1). It includes high-level concepts such as windows, epochs, and distinct progress rules for passive and active target communication. This model lacks support for classical shared-memory operation and is clearly a mismatch with hardware evolution (e.g. multi and many cores). Moreover, MPI-2 RMA is not optimal for implementing global address space languages due to excessive synchronization and its complex progress rules [16]. For the above mentioned reasons, MPI-2 RMA is not widely-used by programmers. The MPI Forum [3] is planning to address these limitations of MPI-2 RMA, as a part of the recently formed MPI-3 RMA initiative [17].

ARMCI exploits native network communication interfaces (e.g. InfiniBand/OpenIB, Seastar/Portals/, Bluegene/DCMF, etc.) and system resources (such as shared memory) to achieve the best possible performance of the remote memory access/one-sided communication. It exploits high-performance network protocols on high end systems such as DCMF on BlueGene and Portals on Cray XT4. ARMCI is also used to implement several parallel programming models such as Co-Array Fortran [18], Global Arrays [14] or GPSHMEM [19]. Due to the widespread popularity of MPI, ARMCI is designed to be compatible with it. ARMCI offers an extensive set of functionality in the area of RMA communication: data transfer operations (put/get/accumulate), atomic operations, memory management and synchronization operations, and locks.

In scientific computing, applications often require transfers of noncontiguous data that correspond to fragments of multidimensional arrays, sparse matrices, or other more complex data structures. The noncontiguous interfaces are available in ARMCI to address these needs. ARMCI offers blocking and nonblocking data transfer operations.

ARMCI relies on simpler progress rules, which are motivated by hardware support for RMA operations on the current architectures [20]. The progress rules in ARMCI follow those of the Cray SHMEM library. Unlike many other portable RMA interfaces, such as the one-sided "active" communication model in MPI-2, or put/get operations in Generic Active Messages [21], GASNet [22], or Data Movement and Control Substrate (DMCS) [23], ARMCI guarantees that its one-sided operations are fully unilateral (i.e., completed regardless of the actions taken by the remote process). In particular, polling the application by the remote process (implicit when making a library call, or explicit by calling the provided polling interface) is not required for communication progress. In this respect, ARMCI is similar to the MPI-2 "passive target" and the vendor-specific RMA interfaces.

III. RMA-based Matrix Multiplication and LU Factorization

In this section, we will describe the parallel implementation of RMA-based `dgemm` (dense matrix-matrix multiplication) and dense LU factorization. Our implementation effectively uses the following techniques and protocols to achieve the best possible performance:

1) zero-copy protocol,
2) network latency hiding through effective non-blocking communication,
3) explicit control of data locality and task mapping (e.g., exploiting shared memory within SMP nodes and

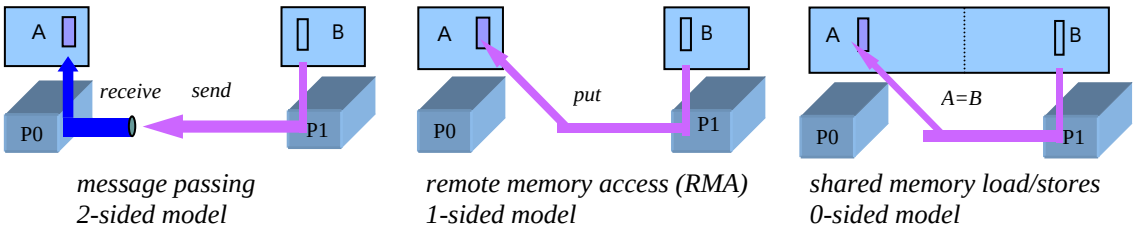

Figure 1. Taxonomy of Communication Models

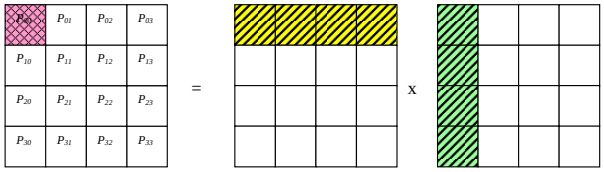

Figure 2. Matrix Distribution for Matrix Multiplication $C = AB$. In a 4×4 process grid, to perform matrix-matrix multiply, process P_{00} needs blocks of matrix A from P_{00}, P_{01}, P_{02}, and P_{03}, and blocks of matrix B from P_{00}, P_{10}, P_{20}, and P_{30}.

RMA across nodes), and

4) reducing communication contention in access to the data.

Implementing matrix multiplication and LU directly on top of shared and remote memory access communication helps us optimize the algorithm with a finer level of control over data movement and hence achieve better performance.

Our matrix multiplication is an extended version of SRUMMA [24]. SRUMMA is a block-based matrix multiply. The disadvantage with SRUMMA is as follows: if the block size is less that problem size per process (i.e. in Figure 2, the size of matrix owned by process P_{00} is greater than block size), then SRUMMA relies on optimized non-contiguous RMA operations. On most networks, contiguous RMA data transfer is faster (in terms of both latency and bandwidth) than non-contiguous data transfer. Also, on some systems (IBM BlueGene/DCMF), there is no native support in DCMF for non-contoguous RMA data transfer. To overcome this limitation, we extended SRUMMA to support block-cyclic distribution. Thanks to block-cyclic distribution, a process can own more than one block, and therefore these blocks can easily map to the block size. Since these blocks are contiguous in memory, we can get the highest performance during data transfer.

A. Parallel Dense Matrix Multiply Algorithm

Consider a matrix multiplication operation $C = AB$, where A, B, and C have the same distribution of data across processes (Figure 2). Our algorithm is based on owner-compute rule: The owner of a block in C performs the `dgemm` for that block. As shown in Figure 2, to perform a matrix-matrix multiply, process P_{00} needs blocks of A owned by processes P_{00}, P_{01}, P_{02}, and P_{03}, and blocks of matrix B owned by P_{00}, P_{10}, P_{20}, and P_{30}. In our algorithm, P_{00} first gets the blocks in A and B owned by P_{00} itself and computes part of C. In the next step, P_{00} gets the blocks in A and B owned by P_{01} and P_{10}, respectively, and accumulates with the C. In a similar fashion, all other processes compute the matrix multiply.

The parallel algorithm to perform RMA-based `dgemm` is as follows:

1) Create global arrays A, B, and C based on block cyclic distribution for the input matrix size. Since A, B, and C are allocated using global arrays, this memory is registered with the network (if required by the network).
2) Generate a list of tasks to perform matrix multiplication on the block(s) owned by a process.
3) Reorder the tasklist in a such a way that the tasks that involve matrix blocks stored in the shared memory domain of the current processor are moved to the beginning of the list. This is done to ensure overlap of computations and nonblocking communication required to bring matrix blocks from other cluster nodes to compute the other tasks on the list. Since the tasks at the beginning of the list use data accessible directly, we do not have to wait to start the pipeline.
4) For each task in the list,
   - Issue a non-blocking get operation for the matrix blocks involved in the next task on the list. Since we always fetch the entire block (as the extended SRUMMA is based on block-cyclic distribution), it is contiguous in memory. The local buffer used in the RMA get operation to get the remote data is also registered with the network in advance (allocated using `ARMCI_Malloc_local()`). Since both the source and destination buffers are contiguous and registered with the network, we can

do zero copy non-blocking RDMA transfers.
- Wait for the non-blocking get operation correspoding to the current task.
- Perform a serial `dgemm`.

As outlined above, this algorithm relies heavily on contiguous zero-copy non-blocking RMA operations and does not rely on `send`/`recv` or collective operations (such as broadcast).

B. Parallel LU Algorithm

Consider a matrix A, whose LU factorization needs to be computed. We create a global array A, by distributing the data onto a two-dimensional grid of processes according to the ScaLAPACK-style block-cyclic scheme to ensure good load balance as well as the scalability of the algorithm. Let us assume the block size is $b \times b$. At each iteration of the loop, a panel of b columns is factorized, and the trailing submatrix is updated. For the sake of simplicity, we have disabled partial pivoting as we make sure the input matrix is diagonally dominant.

1) panel factorization
   - The owner of A_{00} performs a local LU solve: $A_{00} \rightarrow L_{00}, U_{00}$
   - The owner of A_{00} does an RMA `Put()` followed by RMA `Put_Notify()` to both the row and column of processes. Since our data is contiguous, and both the source and destination buffers are registered, we can perform a zero-copy non-blocking RDMA `Put()` here. The source process initiates the `Put()` and returns immediately. It does not wait for remote completion as that is taken care of by RMA `Put_Notify()` which notifies the remote process regarding completion.
   - The remaining column processes wait on the notify and, when received, call `dtrsm()` to perform the panel factorization $A_{10} \rightarrow L_{10}$.
   - The remaining row processes wait on the notify and, when received, call `dtrsm()` to perform the panel factorization $A_{01} \rightarrow U_{01}$.
   - All column and row processes do an RMA `Put()` followed by a `Put_Notify()` to the rest of the processes that own the trailing submatrix.
2) matrix-matrix multiply
   This is owner-compute rule:

$$A'_{11} = A_{11} - L_{10}U_{01}$$

   The processes that own the blocks in the trailing submatrix (A_{11}) wait for the `Put_Notify()` to get L_{10} and U_{01}. Then the trailing submatrix is updated, which includes parallel matrix mutiply as outlined in Section III-A. The processes do not need to synchronize after this step, and they can proceed with the next iteration.

For an efficient implementation of the LU algorithm, we rely on the following assumptions:

- zero-copy RMA `Put()` and `Put_Notify()`,
- the ability to overlap computation within the network communication on clusters is essential for latency hiding,
- hardware-supported shared memory is the fastest protocol available on the shared memory architectures and SMP nodes of the current clusters,
- to avoid dependencies on the OpenMP interfaces and compiler technology, we need as much control over shared memory communication as possible, and
- use of RMA is preferable to the send-receive model, as it makes the implementation simpler and potentially more efficient due to reduced synchronization.

IV. Performance Results

To demonstrate the performance and scalability (strong and weak scaling) of our RMA-based parallel kernels (`dgemm` and LU), we ran numerical experiments with various problem (matrix) sizes on the following platforms:

- Linux cluster based on dual socket, 64-bit, Quad-core AMD 2.2 GHz Opteron processors and Infiniband network at Pacific Northwest national Laboratory [25]
- Cray XT4 at Oak Ridge National Laboratory [26]

We performed both strong and weak scaling experiments. In strong scaling, we fixed the matrix size and measured how the time varies with processor counts. In weak scaling, we fixed the matrix size `per processor` and measured how the time with varies with processor counts. On larger problem sizes, computation cost predominates communication as these parallel kernels have $O(N^3)$ computations and $O(N^2)$ communications. Keeping this in mind, we carefully selected problem sizes (matrix size N from 2K to 100K) such that the communication cost is a reasonable proportion when compared to the computation cost.

We will call our RMA-based dense matrix-matrix multiply and LU factorization kernels "`RMA-dgemm`" and "`RMA-LU`" respectively. We compared our kernels with the `pdgemm()` (parallel matrix multiply) and `pdgetrf()` (parallel LU factorization) kernels from ScaLAPACK. As mentioned in the previous section, for simplicity we assumed diagonally dominant matrices and therefore disabled pivoting in LU factorization. We also manually disabled pivoting in ScaLAPACK's LU factorization as there is no direct method to perform LU without partial pivoting in ScaLAPACK. Optimum block sizes were chosed empirically for all matrix sizes and processor counts.

Figure 3 presents intranode performance of RMA-based kernels and ScaLAPACK. As expected, our kernels achieve superior performance as they exploit direct load/store communication within the underlying SMP nodes. Although the high-performance implementations of message passing can

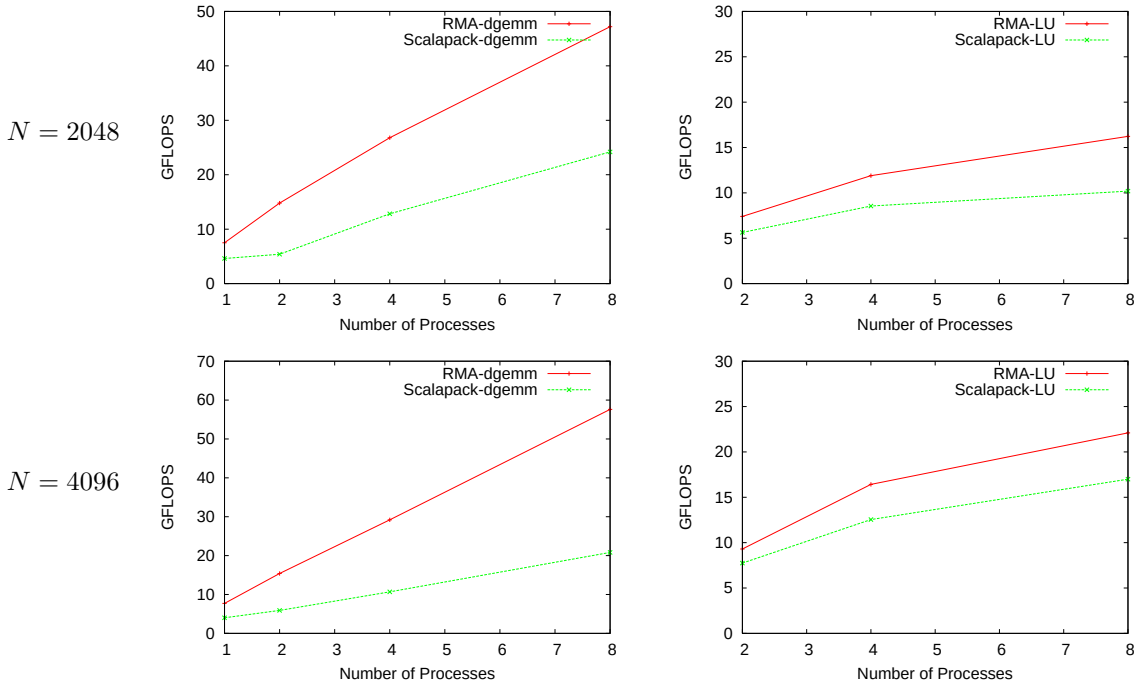

Figure 3. Intranode Performance of `dgemm` (left) and `LU` (right) Algorithms (RMA-based and ScaLAPACK

exploit shared memory internally, the performance is less competitive than direct loads and stores. In the best case, for the matrix size of $N = 4096$, `RMA-dgemm` achieves close to 58 GFlops when compared to ScaLAPACK `dgemm` (20 GFlops) on 8 cores.

To validate the effectivennes of our RMA-based parallel kernels, we ran numerical experiments with various problem sizes ranging from 4K to 64K square matrices. Figures 4 and 5 illustrate the scalability and performance of `RMA-dgemm` and `RMA-LU` respectively. RMA-based kernels consistently outperform MPI-based ScaLAPACK kernels in most of the cases. From Figure 4, `RMA-dgemm` scales well on both the platforms up to 8192 processes. For this study, we chose relatively smaller problem sizes for LU as the impact of communication can be seen (e.g. scaling of $N = 8000$ up to 1600 procs). For larger problem sizes, RMA-LU and ScaLAPACK-LU are comparable. Moreover, since parallel LU algorithm is synchronous in nature `RMA-LU` performs close to ScaLAPACK. However, parallel matrix multiply is more asynchronous, i.e. no coordination between processes are required during computation and therefore RMA model is a natural fit. This is one of the reasons for smaller performance gap in LU, when compared to parallel matrix multiply kernel.

Table I presents the break down source of performance improvements into contributions from 3 main protocols (i.e. zero-copy, non-blocking, and network contention handling) on 1024 processors for a matrix size of 16384. It shows that zero copy is very imporant for performance of RMA-based dgemm. The best possible performance is achieved when all the protocols (including zero copy, non-blocking, and network contention handling) are enabled in the algorithm.

Protocol	Aggregate `TFLOPS`
Zero copy and non-blocking disabled	3.5
Zero copy disabled + non-blocking	4.1
Zero copy + non-blocking disabled	4.9
Zero copy + non-blocking	5.9
Zero copy + non-blocking + n/w contention	6.2

Table I
PERFORMANCE OF RMA-BASED `DGEMM` ON LINUX INFINIBAND CLUSTER USING ZERO-COPY, NON-BLOCKING, AND NETWORK CONTENTION HANDLING PROTOCOLS

Figure 6 presents the weak scaling performance of `dgemm` on both platforms. In this experimental study, we fixed the problem size per process to be $N = 1024$ and ran up to 10000 processes. We noticed that the solution time is constant for both RMA-based and ScaLAPACK kernels, however `RMA-dgemm`'s solution time is twice as fast as ScaLAPACK's `pdgemm()`. Moreover, RMA-dgemm measured close to 60 TFLOPS on 10000 processes when compared to ScaLAPACK (42 TFLOPS), a 43% improvement.

V. CONCLUSION

This paper described and evaluated the design and implementation of two parallel linear algebra kernels based on the RMA programming model for clusters based on multicore processors and shared memory systems. We demonstrated

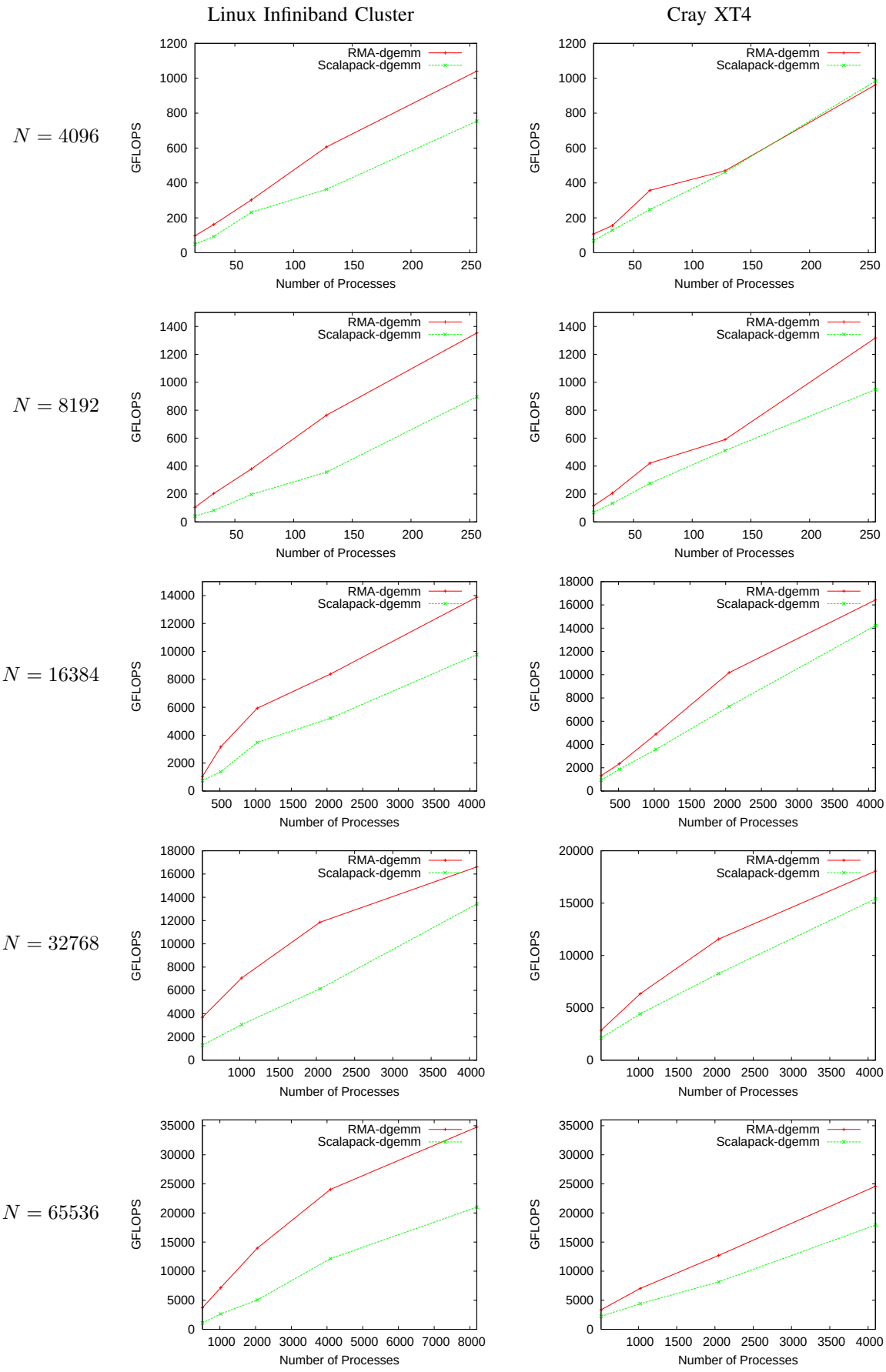

Figure 4. Performance of `RMA-dgemm` vs. ScaLAPACK-dgemm.

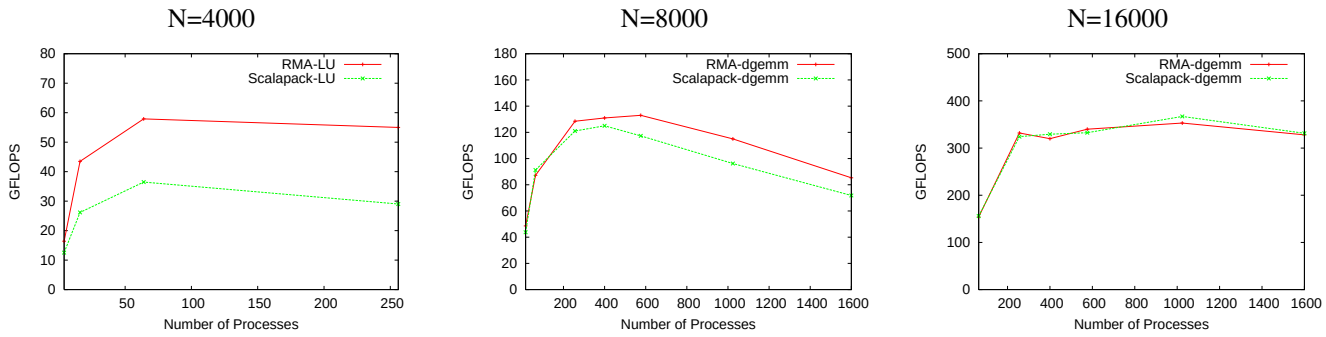

Figure 5. Performance of RMA-based LU vs. ScaLAPACK on Linux Infiniband Cluster

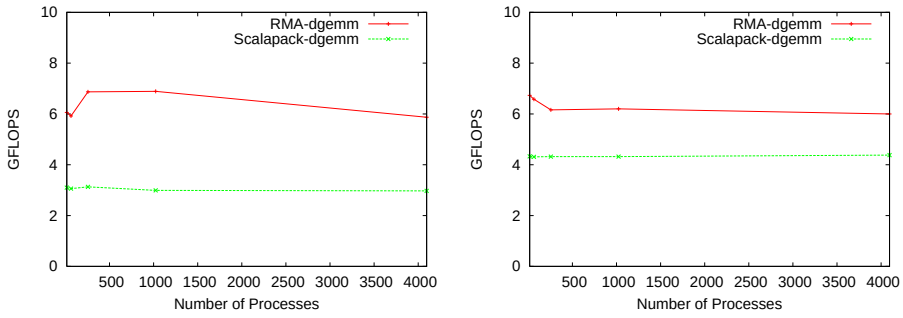

Figure 6. Performance (Weak Scaling) of `RMA-dgemm` vs. ScaLAPACK-dgemm on Linux Infiniband Cluster (left) and Cray XT4 (right)

the strong and weak scaling characteristics of the RMA-based kernels. The experimental results validates the performance and scalability of our algorithm in comparison to the ScaLAPACK suite.

In the future, we intend to explore the possibilty of optimizing the algorithm using RMA-based non-blocking collectives instead of RMA `Put/Get` (which is point-to-point). Our RMA-based collectives will be relatively asynchronous when compared to the MPI-based collectives, which are synchronous.

REFERENCES

[1] J. Dongarra, "Basic Linear Algebra Subprograms Technical Forum Standard," *International Journal of High Performance Applications and Supercomputing*, vol. 16, no. 1, 2002.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide*, 3rd ed. Society of Industrial and Applied Mathematics, 22 Aug. 1999.

[3] MPI Forum. MPI documents page. [Online]. Available: http://www.mpi-forum.org/docs/docs.html

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[5] OpenFabrics Organization. Openfabrics enterprise distribution (ofed) infiniband software stack. [Online]. Available: http://www.openfabrics.org/

[6] R. Brightwell, K. T. Pedretti, K. D. Underwood, and T. Hudson, "Seastar interconnect: Balanced bandwidth for scalable performance," *IEEE Micro*, vol. 26, no. 3, pp. 41–57, 2006.

[7] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, "The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 94–103.

[8] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, "High Performance RDMA-based MPI Implementation over InfiniBand," in *International Conference on SuperComputing*, 2003, pp. 295–304.

[9] J. L. Träff, H. Ritzdorf, and R. Hempel, "The implementation of mpi-2 one-sided communication for the nec sx-5." in *SC*, 2000.

[10] J. Nieplocha, V. Tipparaju, M. Krishnan, G. Santhanaraman, and D. Panda, "Optimizing mechanisms for latency tolerance in remote memory access communication on clusters," in *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, 2003, pp. 138–147, tY - CONF.

[11] C. Guiang, K. Milfeld, and A. Purkayastha, "Remote memory operations of Linux clusters: expressiveness and efficiency of

current implementation," in *3rd LCI International Conference on Linux Clusters*, 2003.

[12] J. Nieplocha, V. Tipparaju, A. Saify, and D. K. Panda, "Protocols and strategies for optimizing performance of remote memory operations on clusters," in *Communication Architecture for Clusters (CAC'02) Workshop, held in conjunction with IPDPS '02*, 2002, pp. 164 – 173.

[13] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. Panda, "High Performance Remote Memory Access Comunications: The ARMCI Approach," *International Journal of High Performance Computing and Applications*, vol. 20, no. 2, 2006.

[14] J. Nieplocha, B. Palmer, V. Tipparaju, M. K. ishnan, H. Trease, and E. Apra, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit," *International Journal of High Performance Computinga and Applications*, vol. 20, no. 2, 2006.

[15] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. L. Lusk, W. Saphir, T. Skjellum, and M. Snir, "MPI-2: Extending the message-passing interface," in *Euro-Par, Vol. I*, 1996, pp. 128–135. [Online]. Available: citeseer.ist.psu.edu/geist96mpi.html

[16] D. Bonachea and J. Duell, "Problems with using mpi 1.1 and 2.0 as compilation targets for parallel language implementations," *Int. J. High Perform. Comput. Netw.*, vol. 1, no. 1-3, pp. 91–99, 2004.

[17] V. Tipparaju, W. Gropp, H. Ritzdorf, R. Thakur, and J. L. Traff, "Investigating high performance rma interfaces for the mpi-3 standard," in *ICPP '09: Proceedings of the 2009 International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 293–300.

[18] R. W. Numrich and J. K. Reid, "Co-Array Fortran for parallel programming," *ACM Fortran Forum*, vol. 17, no. 2, pp. 1–31, August 1998.

[19] K. Parzyszek, J. Nieplocha, and R. A. Kendall, "Generalized portable shmem library for high performance computing," in *IASTED Parallel and Distributed Computing and Systems*, M. Guizani and X. Shen, Eds. Las Vegas, Nevada: IASTED, 2000, pp. 401–406.

[20] K. M, J. Nieplocha, and V. Tipparaju, ""Extending the MPI-2 One-sided Communication Model," in *Intl Conference on High Performance Computing Asia*, 2009.

[21] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *International Symposium on Computer Architecture*, 1992, pp. 256–266.

[22] D. Bonachea, "GASNet Specification, v1.1," October 2002.

[23] N. Chrisochoides, I. Kodukula, and K. Pingali, "Data movement and control substrate for parallel scientific computing," in *of Lecture Notes in Computer Science*. Springer-Verlag, 1997, pp. 77–101.

[24] M. Krishnan and J. Nieplocha, "SRUMMA: a matrix multiplication algorithm suitable for clusters and scalable shared memory systems," in *Parallel and Distributed Processing Symposium*, 2004, pp. 70–79, tY - CONF.

[25] Molecular Science Computing Facility at Pacific Northwest National Laboratory. EMSL: Capabilities: Molecular Science Computing. [Online]. Available: http://www.emsl.pnl.gov/capabilities/computing/

[26] National Center for Computational Sciences. National Center for Computational Sciences >> Jaguar. [Online]. Available: http://www.nccs.gov/computing-resources/jaguar/