SRUMMA: A Matrix Multiplication Algorithm Suitable for Clusters and Scalable Shared Memory Systems

Manojkumar Krishnan and Jarek Nieplocha
Computational Sciences & Mathematics
Pacific Northwest National Laboratory
{Manojkumar.Krishnan, Jarek.Nieplocha}@pnl.gov

Abstract

This paper describes a novel parallel algorithm that implements a dense matrix multiplication operation with algorithmic efficiency equivalent to that of Cannon's algorithm. It is suitable for clusters and scalable shared memory systems. The current approach differs from the other parallel matrix multiplication algorithms by the explicit use of shared memory and remote memory access (RMA) communication rather than message passing. The experimental results on clusters (IBM SP, Linux-Myrinet) and shared memory systems (SGI Altix, Cray X1) demonstrate consistent performance advantages over pdgemm from the ScaLAPACK/PBBLAS suite, the leading implementation of the parallel matrix multiplication algorithms used today. In the best case on the SGI Altix, the new algorithm performs 20 times better than pdgemm for a matrix size of 1000 on 128 processors. The impact of zero-copy nonblocking RMA communications and shared memory communication on matrix multiplication performance on clusters are investigated.

1. Introduction

For many scientific applications, matrix multiplication is one of the most important linear algebra operations. 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 matrix multiplication interface in the open source Basic Linear Algebra Subroutines (BLAS) to deliver performance as close to the peak processor performance as possible. One of the significant innovations in this area was the recent discovery of a practical automatic performance tuning scheme for linear algebra operations, such as the matrix multiplication, in ATLAS [1] to maximize their performance for a given processor architecture as a part of the software installation process. Because the optimized matrix multiplication can be so efficient, computational scientists, when feasible, attempt to reformulate the mathematical description of their application in terms of matrix multiplications.

Parallel matrix multiplication has been investigated extensively in the last two decades [2-22]. There are different approaches for matrix-matrix multiplication: 1D-systolic [5], 2D-systolic [5], Cannon's algorithm [2], Fox's algorithm [3, 4], Berntsen's algorithm [6, 7], the transpose algorithm [8] and DNS algorithm [7, 14, 15]. Fox's algorithm was extended in PUMMA [16] and BiMMeR [17] using different data distribution formats. Agarwal et al. [18] developed another matrix multiplication algorithm that overlaps communication with computation. SUMMA [19] is closely related to Agarwal's approach, and is used in practice in pdgemm routine in PBLAS [20], which is one of the fundamental building blocks of ScaLAPACK [21]. DIMMA [22] is related to SUMMA but uses a different pipelined communication scheme for overlapping communication and computation.

In the earlier studies, researchers targeted their parallel implementations for massively parallel processor (MPP) architectures with uniprocessor computational nodes (e.g., Intel Touchstone Delta, Intel IPSC/860, nCUBE/2) 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. With the emergence of portable message-passing interfaces (PVM, and later MPI), the parallel matrix multiplication algorithms were implemented in a portable manner, distributed widely and used in applications.

The current architectures differ in several key aspects from the earlier MPP systems. Regardless of the processor architecture (e.g., commodity vector, or commodity RISC, EPIC, CISC microprocessors) to improve the cost-effectiveness of the overall system, both the high-end commercial designs (IBM SP, NEC SX-6, Hitachi SR-8000, Cray X1, SGI Altix) and the commodity systems (Beowulf clusters) employ as a building block Symmetric Multi-Processor (SMP) nodes connected with an interconnect network. All of these architectures have the hardware support for load/store communication within the underlying SMP nodes, and some extend the scope of that protocol to the entire machine (Cray X1, SGI Altix). Although the high-performance implementations of message passing can exploit shared memory internally, the performance is less competitive than direct loads and stores. Multiple studies have attempted to exploit the OpenMP shared memory programming model in parallel matrix multiplication, either as a standalone approach on scalable shared memory systems [23, 24] or as a hybrid OpenMP-MPI approach [25, 26] on SMP clusters. Overall,

the reported experiences in comparison to the pure MPI implementations were not encouraging.

The conceptual architectural model for which our algorithm was designed is a cluster of multiprocessor nodes connected with a network that supports remote memory access communication (put/get model) between the nodes. Remote memory access (RMA) is a simple communication model and, on modern systems, is often be the fastest communication protocol available, especially when implemented in hardware as zero-copy RMA write/read operations (e.g., Infiniband, Giganet, and Myrinet). RMA is often used to implement the point-to-point MPI send/receive calls [27, 28]. 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 computations) [29]. 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 between processors on the same SMP node, and nonblocking RMA operations to access parts of the matrix outside of the local shared memory domain (i.e., RMA domain). Note that the shared memory domain might not necessarily match the underlying SMP node configuration used as a hardware building block in many systems. For example, the entire 128-processor SGI Altix system available to us was used as a single shared memory domain, even though underneath it is implemented based on a 2-processor SMP configuration with processors sharing the memory in the module ("brick") and accessing the remainder of system memory through an interconnect network ("NUMAlink").

Implementing matrix multiplication 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. One difference between the OpenMP studies and the current approach is that instead of using a compiler-supported high-level shared memory model, we simply place the distributed matrices in shared memory and exercise full control over the data movement either through the use of explicit loads and stores or optimized block memory copies. In the comparison to the standard matrix multiplication interfaces pdgemm in ScaLAPACK [21] and SUMMA [19], the current algorithm achieved consistent and very competitive performance on four architectures used in the study. These were clusters based on 16-way (IBM SP) and 2-way (Linux/Xeon) nodes, and the shared memory NUMA SGI Altix architecture as well as the Cray X1 with its partitioned shared memory

architecture. In the experimental comparisons, the same optimized implementation of the serial matrix multiplication was used for all parallel matrix multiplication algorithms. The described algorithm is more general, memory efficient, and demonstrated excellent performance and scalability on all four platforms. In addition to describing the new algorithm, this paper demonstrates that the efficient implementation of the communication protocols plays a key role in the performance of matrix multiplication. The zero-copy and nonblocking characteristics of the communication protocols were found to be of critical importance for the performance of the matrix multiplication algorithm on clusters. In contrast to the OpenMP implementation of matrix multiplication on shared memory systems [23, 24], the direct use of shared memory produced excellent performance as compared to MPI (used in ScaLAPACK/PBBLAS pdgemm and SUMMA). For example, for a matrix size 2000x2000 on 128 processors of the Cray X1, ScaLAPACK (Cray optimized -lsci) produced 128 GFLOP/s, where as our algorithm performed at 922 GFLOP/s. In the best case on the SGI Altix, the new algorithm performs 20 times better than ScaLAPACK pdgemm for a matrix size of 1000x1000 on 128 processors.

The paper is organized as follows. The next section describes our algorithm and analyzes its efficiency model. In Section 3, practical implementations of the algorithm are presented for clusters and shared memory systems. Section 4 describes and analyzes performance results for the new algorithm and ScaLAPACK matrix multiplication as well as results for the communication operations used in the implementation. The impact of zero-copy and nonblocking communication on the matrix multiplication performance is demonstrated. Finally, summary and conclusions are given in Section 5.

2. New Algorithm – SRUMMA

At the high level, our algorithm, called SRUMMA (Shared and Remote-memory based Universal Matrix Multiplication Algorithm), follows the serial block-based matrix multiplication (see Figure 1) by assuming the regular block distribution of the matrices A, B, and C and adopting the "owner computes" rule with respect to blocks of the matrix C. Each process accesses the appropriate blocks of the matrices A and B to multiply them together with the result stored in the locally owned part of matrix C. The specific protocol used to access nonlocal blocks varies depending on whether they are located in the same or other shared memory domain as the current processor. In principle, the overall sequence of block matrix multiplications can be similar to that in Cannon's algorithm. However, unlike Cannon's algorithm, where skewed blocks of matrix A and B are shifted using message-passing to the logically neighboring processors,

our approach fetches these blocks independently, as needed, without requiring any coordination with the processors that own the matrix blocks. This is possible thanks to the use of RMA and shared memory access protocols. In addition, the specific sequence in which the block matrix multiplications are executed is determined dynamically at run time to more efficiently schedule and overlap communication with computations. The absence of sender-receiver synchronization/coordination (such in Cannon's algorithm) based on message passing makes the overall algorithm more asynchronous and thus more suited for the execution environments where the computational threads share a CPU with other processes and system daemons (e.g., on commodity clusters). This is because synchronization amplifies performance degradations due to the nonexclusive use of the processor by the application.

```
1:    for i=0 to s-1 {
2:        for j=0 to s-1 {
3:            Initialize all elements of C_ij to zero (optional)
4:            for k=0 to s-1 {
5:                C_ij = C_ij + A_ik×B_kj
6:            }
7:        }
8:    }
```

Figure 1: Block matrix multiplication for matrices N×N and block size N/s × N/s

2.1 Baseline Efficiency Model

Consider a matrix multiplication operation C = AB, where the order of matrices A, B, and C is $m \times k$, $k \times n$, and $m \times n$, respectively. Let us denote:

t_w - data transfer time per element
t_s - latency (or startup cost)
$p \times q$ - process grid in two-dimensional fashion
P - number of processors

and assume (as in [7, 30]) that the cost of the addition and multiplication floating point operation takes unit time (line 5 in Figure 1). For our analysis, we assume a two-dimensional matrix distributed as shown in Figure 2. Each process owns a block of the A, B and C matrices of size $\frac{m}{p} \times \frac{n}{q}$, $\frac{m}{p} \times \frac{k}{q}$ and $\frac{k}{p} \times \frac{n}{q}$, respectively. The sequential time T_{seq} of the matrix multiplication algorithm is N^3 (say, m=n=k=N). The parallel time T_{par_rma} is the sum of computation time (T_{comp}) and the time to *get* the blocks of matrices A and B (T_{comm}).

T_{comm} = (time to get row of matrix A blocks) + (time to get column of B blocks) = $T_{row\_comm} + T_{column\_comm}$

Using RMA protocols (e.g. get, put), each process *gets q* blocks of matrix A and *p* blocks of matrix B of size $\left(\frac{m}{p}\right)\left(\frac{k}{q}\right)$ and $\left(\frac{k}{p}\right)\left(\frac{n}{q}\right)$, respectively.

T_{row_comm} = {data transfer time of message size mk/pq} + {latency/start-up cost}

$$T_{row\_comm} = \left(\left(\frac{mk}{pq}\right)t_w + t_s\right)q \;;\; T_{column\_comm} = \left(\left(\frac{nk}{pq}\right)t_w + t_s\right)p$$

Figure 2: Matrix distribution example. In a 4 x 4 process grid, 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}.

$$T_{par\_rma} = \frac{mnk}{P} + \left(\left(\frac{mk}{pq}\right)t_w + t_s\right)q + \left(\left(\frac{kn}{pq}\right)t_w + t_s\right)p$$

For simplicity let us assume, $m=n=k=N$ and $p=q=\sqrt{P}$, then the above equation becomes,

$$T_{par\_rma} = \frac{N^3}{P} + 2\frac{N^2}{\sqrt{P}}t_w + 2t_s\sqrt{P} \qquad (1)$$

$$= O\left(\frac{N^3}{P}\right) + O\left(\frac{N^2}{\sqrt{P}}\right) + O\left(\sqrt{P}\right) \qquad (2)$$

For a network with sufficient bandwidth, t_s can be neglected as it is relatively small when compared to the total communication time. Therefore, the parallel efficiency (η) is

$$\eta = \text{Speedup}/P \approx \frac{1}{1 + \frac{2\sqrt{P}}{N}t_w} = \frac{1}{1 + O\left(\frac{\sqrt{P}}{N}\right)}$$

The isoefficiency function of this algorithm is O ($P^{3/2}$), which is the same as Cannon's algorithm [7, 19].

Overlapping communication with computations: When non-blocking RMA is used to transfer matrix blocks, the communication can be overlapped with computation, as shown in Figure 3.

The degree of overlapping, ω, is defined as: $\omega = \left(1 - \frac{T_{comp}}{T_{comm}}\right)$

If ω < 0, then ω = 0. Introducing ω in (1),

$$T_{par\_rma} = 2\left(\frac{N^3\alpha}{P} + \omega\left(\frac{N^2}{P}\beta\right)\sqrt{P} + \delta\sqrt{P}\right) \qquad (3)$$

When $T_{comp} >= T_{comm}$ (i.e., 100% overlap), equation (3)

reduces to, $T_{par\_rma} = \frac{N^3}{P} + 2t_s\sqrt{P} = O\left(\frac{N^3}{P}\right) + O\left(\sqrt{P}\right)$.

3 Implementation Considerations

To derive an efficient implementation of the matrix multiplication algorithm, we rely on the following assumptions: 1) the ability to overlap computation with the network communication on clusters is essential for latency hiding; 2) hardware-supported shared memory is the fastest protocol available on the shared memory architectures and SMP nodes of the current clusters ; 3) to avoid dependencies on the OpenMP interfaces and compiler technology, we need as much control over shared memory communication as possible; and 4) use of RMA is preferable to the send-receive model, as it makes the implementation simpler and potentially more efficient due to reduced synchronization. Based on these assumptions, we designed two instances of the matrix multiplication algorithm. We will first describe algorithm implementation for clusters composed of nodes with shared memory domains; then we will discuss special considerations for the scalable shared memory systems.

3.1 Cluster Version

For each processor p and corresponding matrix block C_{ij} held on that processor,

1. Build a *list of tasks* (where a task computes each of the $A_{ik}B_{kj}$ products.) corresponding to the block matrix

   multiplications in: $C_{ij} = \sum_{k=1}^{n_p} A_{ik} B_{kj}$ (4)

2. Reorder the *task list* according to the communication domains for processors at which the A_{ik}, B_{kj} are stored. 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. Another consideration in sorting the task list is to optimize the locality reference so that the currently held A_{ik} matrix block is used in consecutive matrix products before its copy is discarded and the corresponding buffer reused.

3. For each task on the list,
   - Issue a nonblocking get operation for the matrix block involved in the next task on the list if it is not on the same node.
   - Wait for the nonblocking get operation bringing A_{ik} and/or B_{kj} needed to execute the current task.
   - Call serial matrix multiplication *dgemm* that computes $A_{ik}B_{kj}$ and adds the result to the C_{ij} block.

4. There are two temporary buffers (*B1* and *B2*) used internally. One buffer is used for communication and the other buffer is used for computation as shown in Figure 3. At a given step, a processor receives data in *B2* while computing the data in *B1*. In the next step,

data received in *B2* is used for computation and *B1* is used for receiving data. Overlapping communication with computation is achieved in all steps, except first.

As a further refinement of the algorithm, as shown in Figure 4, the "diagonal shift" algorithm is used in Step 2 to sort the task list so that the communication pattern reduces the communication contention on clusters. We verified experimentally on the IBM SP that indeed this improves performance. For example, consider matrix *A* that is distributed on a 4 x 4 processor grid, on a 4-way SMP cluster. As shown in Figures 4a and 4b, node 1 has processors P_{00}, P_{10}, P_{20}, and P_{30}; node 2 has P_{01}, P_{11}, P_{21}, and P_{31}; etc.,. To compute its locally owned matrix *C*, a processor needs the corresponding rows and columns of matrix *A* and *B* respectively, as shown in Figure 3. i.e., processor 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}. If the diagonal shift algorithm is not used, processors P_{00}, P_{10}, P_{20}, and P_{30} get a block from P_{01}, P_{11}, P_{21}, and P_{31},

(a)   (b)

(c)
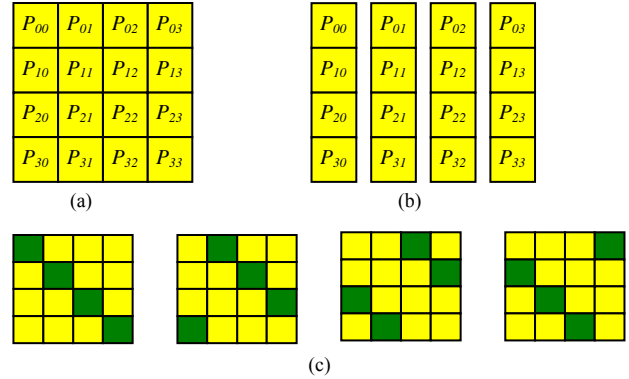
Figure 4: Pattern of getting blocks on a 4-way SMP cluster to reduce communication contention.

A   B   C
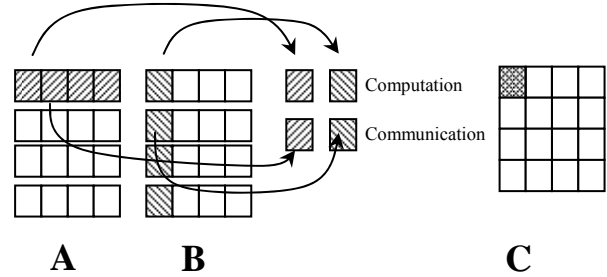
Figure 3. Using two sets of buffers to overlap communication and computation in matrix

respectively in the first step. Thus all the 4 processors are trying to share the bandwidth between node1 and node2. If the diagonal shift algorithm is used instead, then processors P_{00}, P_{10}, P_{20}, and P_{30} get a block from P_{00} (node1), P_{11} (node2), P_{22} (node3), and P_{33} (node4), respectively in the first step, thus reducing the contention. This algorithm performs better if there are more processors per node (e.g., 16-way IBM SP). Figure 4c represents the pattern of getting blocks by processors in node 1.

3.2 Shared Memory Version

The cluster algorithm running on a system with one shared memory communication domain reduces to the shared memory version. However, this algorithm has two flavors; the one used depends on whether remote shared memory is locally cacheable. For example, the Cray X1 with its partitioned shared memory supports load/store operations for its entire memory. The system is a cluster with four multi-stream processors (MSPs) on each node. A virtual memory address includes node number and address within that node. The memory on other nodes can be accessed with the load/store operations; however, it cannot be cached due to the memory coherency protocol [31]. Because the performance of the serial matrix multiplication depends critically on the effective cache utilization, on the Cray X1 we copy nonlocal blocks of matrices A and B to a local buffer before calling the serial matrix multiplication.

On the other hand, the SGI Altix is a shared memory system where shared memory data can be cached. Therefore, the matrix multiplication does not require explicit memory copies. Instead, the appropriate blocks of matrix A and B are passed directly to the serial matrix multiplication subroutine. A comparison between these two schemes for $C = A^T B$ and $C = AB$ on these two platforms is illustrated in Figure 5. The SGI Altix uses 1.5-GHz Intel Itanium-2 processors rated at 6 GFLOP/s whereas the Cray X1 processor is rated at 12.8 GFLOP/s. As expected, the copy-based version is faster than the direct access version on the Cray X1 and somewhat slower on the SGI Altix (the gap between these two algorithms actually increases for larger processor counts on the Altix).

3.3 Portable Implementation

Our current implementation of the matrix multiplication algorithm relies on the portable Aggregate Remote Memory Copy Interface (ARMCI) library [32] and, in particular, the memory allocation interface ARMCI_Malloc, nonblocking get operations, and the cluster configuration query interfaces [33]. The cluster configuration information provided by ARMCI enables the application at run time to determine which processors can communicate through shared memory. ARMCI_Malloc is a collective memory allocator that allocates shared memory on clusters or shared memory architectures (e.g., SGI Altix). This is accomplished using OS calls such as the System V *shmget*/*shmat*, with the exception of the Cray X1, where even memory allocated by *malloc* can be globally shared. ARMCI_Malloc returns pointers to the memory allocated for all the processors. Using the pointer values and cluster locality information, processors in the same shared memory domain can access the allocated memory directly through load/store operations or through the ARMCI communication calls. For example, the ARMCI get/put operations are implemented as a memory copy within the SMP node of a cluster. Thanks to the
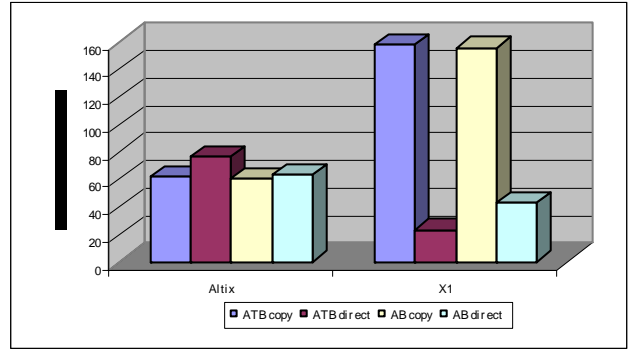
Figure 5: Performance of matrix multiplication (N=2000) on 16 processors using direct access and copy on the Cray X1 and the SGI Altix

ARMCI compatibility with MPI, the current implementation of the matrix multiplication routine could be used in normal MPI-based programs, provided that the distributed arrays are allocated using ARMCI_Malloc rather than, for example, the standard *malloc* call. This is not a significant restriction because in most applications, distributed arrays are created collectively anyway. To achieve maximum performance in the RMA communication on Linux clusters with Myrinet, ARMCI_Malloc internally attempts to register the memory used for the matrices with the Myrinet GM network interface driver. If registration is successful, it enables the direct use of efficient zero-copy communication through the GM Myrinet protocols. Otherwise, either copy-based or on-the-fly dynamic memory registration protocols are used [34, 35]. MPICH-GM registers user communication buffers to enable zero-copy data transfers as well; these registrations are transparent to the user. The zero-copy communication enables the network interface card (NIC) to complete the data transfers without involving the host CPU. This is important for ensuring progress in the nonblocking communication while the host CPU is involved in computations.

4. Experimental Study

To validate the effectiveness of the proposed algorithm, we ran numerical experiments on four platforms:

- Linux cluster based on dual 2.4-GHz Intel Xeon nodes and Myrinet-2000 network
- IBM SP based on 16-way 375-MHz Power-3 CPUs and colony switch at the National Energy Research Scientific Computing Center
- Cray X1 massively parallel vector supercomputer at Oak Ridge National Laboratory
- SGI Altix 3000, shared-memory NUMA system with 128 1.5-GHz Intel Itanium-2 CPUs at Pacific Northwest National Laboratory.

For the comparison, we used the *pdgemm* routine from ScaLAPACK Version 1.7, and SUMMA. However, to

save space we are reporting pdgemm results only, as it is the most commonly used parallel linear algebra library and the performance comparisons were similarly favorable with SUMMA. Moreover, SUMMA is used in practice in ScaLAPACK/PBLAS suite [20]. In all three parallel algorithms, the same *dgemm* (double precision serial matrix multiplication) routines from vendor optimized math library (-lsci for the X1, -lessl for the SP, -lscs for Altix, -lmkl for Xeon) were used. Optimum block sizes were chosen empirically for all matrix sizes and processor counts.

Figure 10 presents the performance differences for SRUMMA and ScaLAPACK *pdgemm* for square matrices; matrix ranks range from 600 to 12000 on all four platforms. In these tests, the matrices were not transposed. As shown in Figure 10, the new algorithm outperforms *pdgemm* and scales better, with the most profound gains noted on the two shared memory systems, Cray X1 and SGI Altix. This is due to the benefit shared memory communication offers on these architectures over message passing. As a best case on Altix, SRUMMA performs 20 times better than ScaLAPACK pdgemm for a matrix size of 1000 on 128 processors. However, memory bandwidth is a bottleneck for very large problem sizes (for e.g., problem size of 12000 on Altix) on shared memory systems. For the matrix size of 12000 on 128 CPUs we observe reduced scaling, most likely due to the fact that the larger test case increases the probability of the system daemons preempt the matrix multiplication code when all the processors on the Altix system were used.

Our ScaLAPACK results are consistent with, and even better than, the results reported earlier by other researchers [36]. For example, [36] reports 48 GFLOP/s on Altix and 112 GFLOP/s on Cray X1 to multiply matrices of size 8000x8000 using ScaLAPACK pdgemm. Our pdgemm runs gave even better performance; 96 GFLOP/s on Altix and 243 GFLOP/s on the Cray X1. This difference is due to the faster processor and memory configuration on Altix (we used Altix with 1.5 Ghz Itanium-2 CPUs, instead of the 1.3 GHz model in [36]), and recent improvements in the vendor math libraries.

The new algorithm performs better than ScaLAPACK on clusters as well. For example on the Linux cluster, it is faster by a factor of two for larger problem sizes, and by 20% to 40% in most of the cases. In most of the cases (especially on Linux), we were able to overlap 90% of the communication with computation.

Section 4.1 describes the impact of communication protocols to achieve this high degree of overlap. Since shared memory is the fastest communication protocol available on shared memory systems, it is not surprising that our algorithm outperforms ScaLAPACK algorithm implemented on top of message passing. However, to understand the performance differences between the two versions of the matrix multiplication algorithm on clusters

we performed several tests to measure the role of the underlying communication protocols.
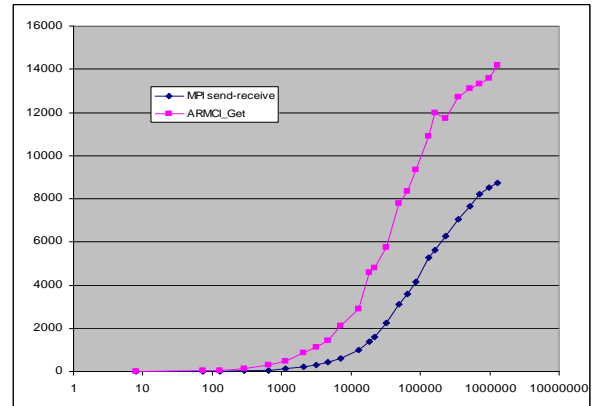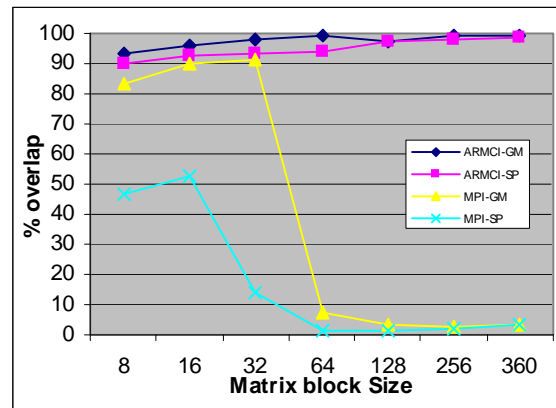

Figure 6: Bandwidth comparison on Cray X1

Figure 7: Potential degree of communication overlap on IBM SP and Linux cluster as a function of message-size

4.1 Impact of communication protocols

First, we investigated the performance of MPI *send/receive* operations and the ARMCI *get* operation (we did not include SGI Altix in this study due to the use of direct access protocol in our matrix multiplication). Figures 6 and 8 shows that performance of these RMA protocols is better than MPI, with the exception of the short message range (in principle, the *get* operation involves request and reply which understandably leads to a higher latency). The MPI timings correspond to half of the round-trip message exchange. In addition, the high cost of AIX interrupt processing in LAPI makes the latency of this protocol higher than in MPI send-receive operations that uses polling. We also measured performance of MPI_Get (MPI-2) on the IBM SP and found its performance to be relatively low as compared to the other two protocols.

Unlike the message-passing implementation of the matrix multiplication in ScaLAPACK pdgemm, our algorithm is using *nonblocking RMA*, which offers a potential for overlapping communication with computations [37]. We

measured this potential for ARMCI and MPI on our two cluster platforms (see Figure 7). In comparison to MPI non-blocking, ARMCI non-blocking *get* offers almost 99% overlap for medium- and larger-sized messages. Similarly to other studies [38, 39], we found the potential degree of overlapping MPI communication with computations to be less favorable: it sharply decreases after a certain message size (16Kb) as MPI switches to the Rendezvous protocol [29]. Therefore, the use of nonblocking protocols in SRUMMA is expected to be beneficial when the problem size and processor count is increased, due to a high degree of overlapping of communication with computations.

Our next communication-related test attempted to evaluate to what degree zero-copy RMA communication affects the performance of matrix multiplication. This communication approach allows the remote CPU to work on its own computations rather than be interrupted and involved in data copying on behalf of another processor. On the Myrinet with GM 1.X, ARMCI is implemented using the GM put operation and pthreads [35]. Unlike Myrinet GM, IBM LAPI is not a zero-copy protocol, thus we could only use Myrinet to investigate the role zero-copy protocols play in performance of the matrix multiplication. The new matrix multiplication algorithm was tested when enabling and disabling the zero-copy implementation [29] of the ARMCI get operation. Figure 9 shows that zero-copy protocol is very important for performance of the new algorithm. This test is performed on the Linux cluster with Myrinet, using:

- blocking and non-blocking communications, and
- zero-copy protocol disabled and enabled.

These results show that the performance benefit of using nonblocking communications is amplified when the zero-copy protocol is enabled. This is because, the remote host CPU cycles are not taken away when overlapping communication with computation since the NICs are able to transfer the data between the user buffers across the network. We were able to overlap more than 90% of the communication with computation, thus the degree of overlapping (ω) is less than 10%. Therefore, Equation 3 reduces to:

$$T_{par\_rma} = \frac{N^3}{P} + (0.1)\left(\frac{N^2}{P}t_w\right)\sqrt{P} + t_s\sqrt{P} \; = \; \frac{N^3}{P} + \zeta$$

where $\zeta \ll (N^3/P)$ for medium/larger problem sizes.

ARMCI_Get is implemented as a trivial wrapper to LAPI_Get on the IBM SP. Because neither LAPI nor IBM MPI is a zero-copy protocol, the remote host CPU involvement is required to copy the data from the user buffers to DMA buffers before NIC can transfer it. Therefore, the performance is not optimal. However, the IBM SP is based on 16-way nodes, thus our algorithm ends up using shared memory for a larger fraction of interprocessor communication than on the Linux cluster. This partly compensates for the lack of zero-copy communication on the IBM SP and makes the new matrix

multiplication algorithm work faster overall than ScaLAPACK. Based on the Myrinet experiments, we would expect our matrix multiplication to benefit from zero-copy protocols in LAPI, which IBM has already introduced in KLAPI, a kernel version of this library.
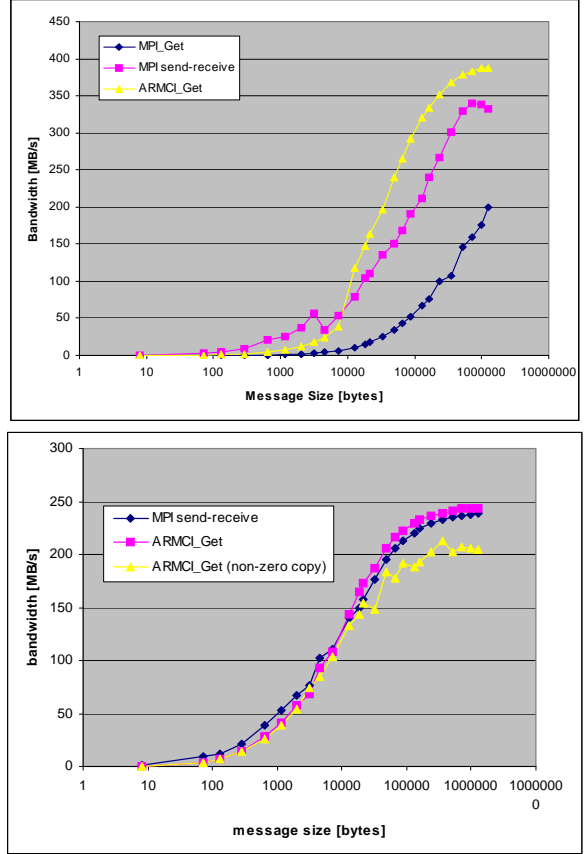

Figure 8: Performance of MPI and ARMCI_Get on IBM SP (top) and Myrinet (bottom)
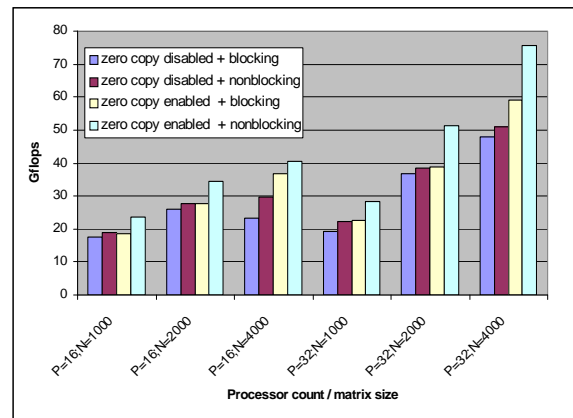

Figure 9: Performance of the matrix multiplication on Linux Cluster (Myrinet Interconnect) with enabled or disabled zero-copy protocol
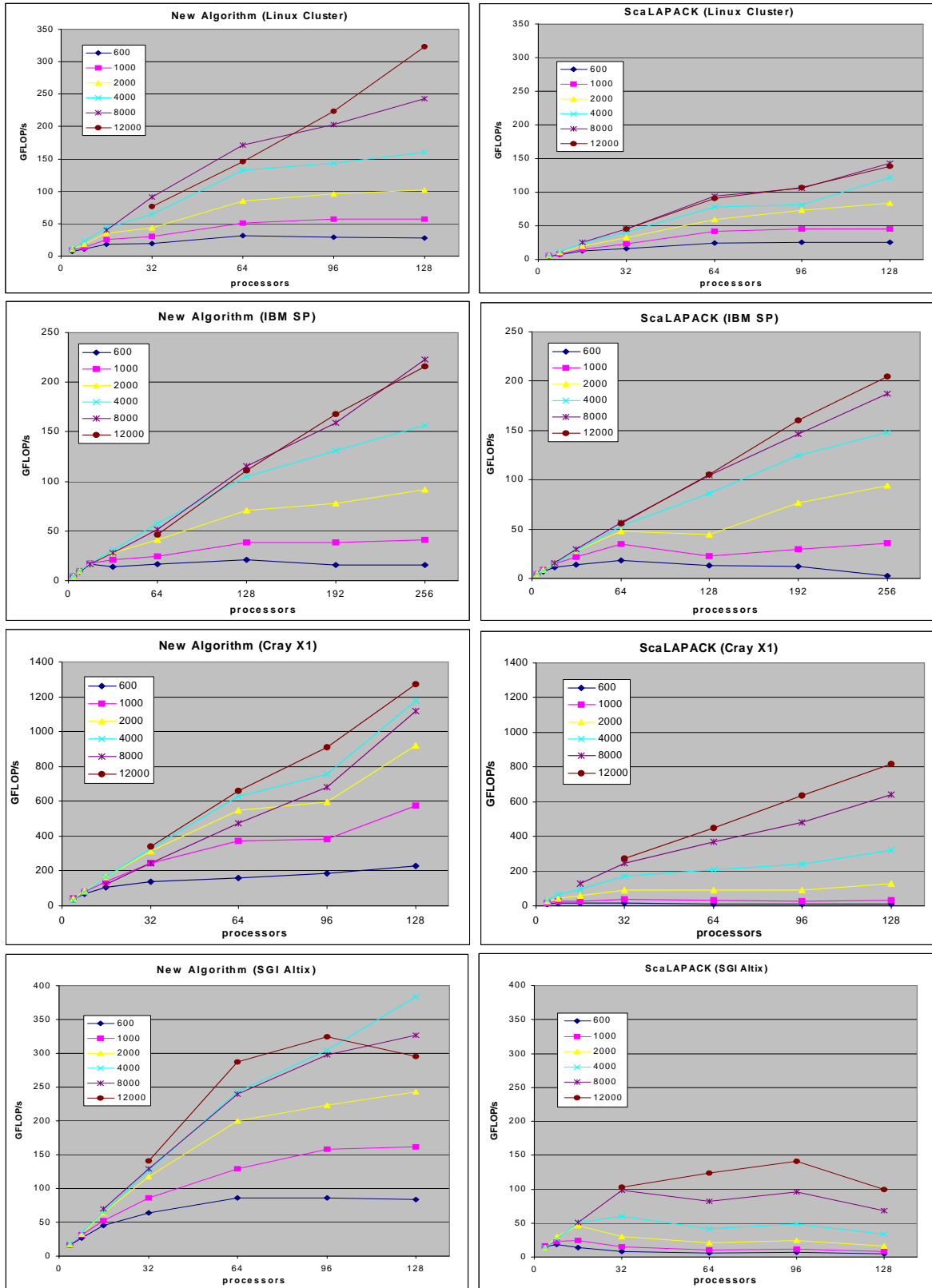
Figure 10: Performance of the new algorithm with ScaLAPACK pdgemm on different platforms

4.2 Transpose and Rectangular Matrices

We also measured the performance of matrix multiplication using SRUMMA and ScaLAPACK/PBBLAS pdgemm for transposed and rectangular matrices. Three variants were considered: $C = A^T B$ (i.e., A is transposed, B is not), $C = AB^T$ and $C = A^T B^T$. For rectangular matrices operation $C = AB$, where the order of matrices A, B, and C is m x k, k x n, and m x n, respectively. Several tests involving various sizes of matrices ranging in size from 600 to 8000 and of different shapes were conducted. Due to the space limitations, we cannot demonstrate all the performance result. However, the best cases are presented in Table 1 included in the next section. SRUMMA consistently outperformed pdgemm on clusters and shared memory systems in most of the cases. The observed performance is similar square and non-transposed cases. SRUMMA scaled well when the number of processors and/or the problem size was increased, thus proving the algorithm is cost-optimal. In all the cases, SRUMMA took advantage of the shared memory communication (in the case of shared memory systems), and zero-copy and non-blocking RMA protocols (in the case of the cluster), to achieve high performance. However, performance degrades for smaller matrices on larger processor counts. This is because, for small matrix sizes and larger processor counts, the potential for overlapping is limited. Moreover, for short message ranges, the *get* operation involves request and reply, which leads to a higher latency.
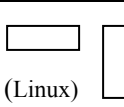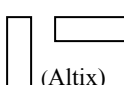
Matrix Size	CPUs	Case Platform	SRUMMA (GFLOP/s)	pdgemm (GFLOP/s)
4000x4000	128	C=AB (Altix)	384	33.9
2000x2000	128	C=AB (Cray X1)	922	128
12000x 12000	128	C=AB (Linux)	323.2	138.6
8000x8000	256	C=AB (IBM SP3)	223	186
600x600	128	C=$A^T B^T$ (Linux)	16.64	6.4
16000x 16000	128	C=$A^T B$ (IBM SP3)	108.9	77.4
4000x4000	128	C=$A^T B^T$ (Altix)	369	24.3
m=4000; n=4000; k=1000 Rectangular	128	(Linux)	160	107.5
m=1000; n=1000; k=2000 rectangular	64	(Altix)	288	17.28

Table 1: SRUMMA best cases.

5. Summary and Conclusions

This paper described a novel parallel algorithm for a dense matrix multiplication operation with algorithmic efficiency equivalent to that of Cannon's algorithm. Unlike the other leading parallel algorithms based on message passing, the current approach exploits shared memory and nonblocking remote memory access protocols on clusters and shared memory systems. Overall, the algorithm achieved consistent and substantial performance gains over the parallel matrix multiplication in the ScaLAPACK/PBBLAS suite. Some of its best cases are presented in Table 1. The experimental results on clusters indicate that the nonblocking zero-copy RMA communication plays an important role in the ability to overlap communication with computations and improve efficiency of the algorithm. On shared memory systems cache and direct access are instrumental in achieving high levels of performance, even though on the Cray X1 an additional memory copy might be involved.

References

1. R. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software (ATLAS)", *Supercomputing*'89.
2. L. E. Cannon, "A cellular computer to implement the Kalman Filter Algorithm", Ph.D. dissertation, Montana State University, 1969.
3. G. C. Fox, S. W. Otto, and A. J. G. Hey, "Matrix algorithms on a hypercube I: Matrix multiplication", *Parallel Computing*, vol. 4, pp. 17-31. 1987.
4. G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors*. vol. 1, Prentice Hall, 1988.
5. G.H. Golub and C.H Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989.
6. J. Berntsen, Communication efficient matrix multiplication on hypercubes, *Parallel Computing*, vol. 12, pp. 335-342, 1989.
7. A. Gupta and V. Kumar, "Scalability of Parallel Algorithms for Matrix Multiplication", *Proc. ICPP*, 1993
8. C. Lin and L.Snyder, "A matrix product algorithm and its comparative performance on hypercubes", in *Scalable High Performance Computing Conference*, 1992,
9. Q. Luo and J. B. Drake, "A Scalable Parallel Strassen's Matrix Multiply Algorithm for Distributed Memory Computers", http://citeseer.nj.nec.com/517382.html

10. S. Huss-Lederman, E. M. Jacobson, and A. Tsao, "Comparison of Scalable Parallel Matrix Multiplication Libraries," in *Scalable Parallel Libraries Conference*, IEEE Computer Society Press, 1994, pp. 142-149.

11. C. T. Ho, S. L. Johnsson and A. Edelman, "Matrix multiplication on hypercubes using full bandwidth and constant storage", in *Proceeding of the Sixth Distributed Memory Computing Conference*. 1991, pp. 447-451.

12. H. Gupta and P. Sadayappan, "Communication Efficient Matrix Multiplication on Hypercubes", in *Proc Sixth ACM SPAA*, 1994.

13. J. Li, A. Skjellum, and R. D. Falgout, "A Poly-Algorithm for Parallel Dense Matrix Multiplication on Two-Dimensional Process Grid Topologies," *Concurrency, Practice and Experience*, vol. 9(5), 1997.

14. E. Dekel, D. Nassimi, and S. Sahni, "Parallel matrix and graph algorithms", *SIAM Journal on Computing*, vol. 10, pp. 657-673, 1981.

15. S. Ranka and S. Sahni. Hypercube Algorithms for Image Processing and Pattern Recognition. Springer-Verlag, New York, NY, 1990.

16. J. Choi, J. Dongarra, and D. W. Walker, "PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers," *Concurrency: Practice and Experience*, vol. 6(7), pp. 543-570, 1994.

17. S. Huss-Lederman, E. Jacobson, A. Tsao, and G. Zhang, "Matrix Multiplication on the Intel Touchstone DELTA", *Concurrency: Practice and Experience*, vol. 6 (7) pp. 571-594. Oct 1994.

18. R. C. Agarwal, F. Gustavson, and M. Zubair, "A high performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication," *IBM J. of Research and Development*, vol. 38 (6), 1994.

19. R. van de Geijn, R. and J. Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," *Concurrency: Practice and Experience*, vol. 9(4), pp. 255–274, 1997.

20. J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and, R. C. Whaley, "A Proposal for a Set of Parallel Basic Linear Algebra Subprograms", University of Tennessee, Knoxville, Tech. Rep. CS-95-292, May 1995.

21. L. S. Blackford et. al., *ScaLAPACK Users' Guide*, SIAM, 1997, Philadelphia, PA.

22. J. Choi, "A Fast Scalable Universal Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers", in *Proc. IPPS '97*, 1997.

23. C. Addison and Y. Ren, "OpenMP Issues Arising in the Development of Parallel BLAS and LAPACK libraries", in *Proceedings EWOMP'01*. 2001.

24. G. R. Luecke and W. Lin, "Scalability and Performance of OpenMP and MPI on a 128-Processor SGI Origin 2000", *Concurrency and Computation: Practice and Experience*, vol. 13, pp 905-928. 2001.

25. M. Wu, S. Aluru, and R. A. Kendal, "Mixed Mode Matrix Multiplication", *IEEE CLUSTER'02*, 2002.

26. T. Betcke, "Performance analysis of various parallelization methods for BLAS3 routines on cluster architectures", John von Neumann-Instituts für Computing, Tech. *Rep. FZJ-ZAM-IB-2000-15*, Nov, 2000.

27. J. L. Träff, H. Ritzdorf, R. Hempel "The Implementation of MPI-2 One-Sided Communication for the NEC SX-5", in *Proceedings of Supercomputing*, 2000.

28. J. Liu, J. Wu, S. P. Kinis, P. Wyckoff, and D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand", in *Proc of 17th ACM International Conference on Supercomputing*, 2003.

29. J. Nieplocha, V. Tipparaju, M. Krishnan, G. Santhanaraman, and D.K. Panda,"Optimizing Mechanisms for Latency Tolerance in Remote Memory Access Communication on Clusters", *IEEE CLUSTER*, 2003.

30. A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, Addison Wesley, 2003.

31. Optimizing Applications on the Cray X1TM System. http://www.cray.com/craydoc/20/manuals/S-2315-50/html-S-2315-50/S-2315-50-toc.html

32. J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems", in *Proceedings of RTSPP IPPS/SDP*, 1999.

33. ARMCI Web page. http://www.emsl.pnl.gov/docs/parsoft/armci/

34. J. Nieplocha, V. Tipparaju, J. Ju, and E. Apra, "One-sided communication on Myrinet", *Cluster Computing*, vol. 6, pp. 115-124, 2003.

35. J. Nieplocha, V. Tipparaju, A. Saify, and D. Panda, "Protocols and Strategies for Optimizing Remote Memory Operations on Clusters", *Proc CAC/IPDPS'02.2002*.

36. ORNL Tom Dunigan's Evaluation of Early Systems Webpage. http://www.csm.ornl.gov/~dunigan/

37. V. Tipparaju, M. Krishnan, J. Nieplocha, G. Santhanaraman, and D.K. Panda, "Exploiting Non-blocking Remote Memory Access Communication in Scientific Benchmarks", *Proc. HiPC'2003*, 2003.

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

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