Optimizing Parallel Multiplication Operation for Rectangular and Transposed Matrices

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

Abstract

In many applications, matrix multiplication involves different shapes of matrices. The shape of the matrix can significantly impact the performance of matrix multiplication algorithm. This paper describes extensions of the SRUMMA parallel matrix multiplication algorithm [1] to improve performance of transpose and rectangular matrices. Our approach relies on a set of hybrid algorithms which are chosen based on the shape of matrices and transpose operator involved. The algorithm exploits performance characteristics of clusters and shared memory systems: it 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 and shared memory systems demonstrate consistent performance advantages over pdgemm from the ScaLAPACK parallel linear algebra package.

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. 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 and the commodity systems 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 the 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 underlying conceptual model of the architecture for which the SRUMMA (Shared and Remote-memory based Universal Matrix Multiplication Algorithm) 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 [1]. Remote memory access (RMA) is often be the fastest communication protocol available, especially when implemented in hardware as zero-copy RDMA 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*. SRUMMA 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., *RMA domain*).

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 our 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], it was shown [1] that for square matrices SRUMMA achieves 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, shared memory NUMA SGI Altix architecture as well as the Cray X1 with its partitioned shared memory architecture. In the current paper we extend the SRUMMA algorithm to handle efficiently rectangular and transposed matrices. With the current optimizations, the SRUMA algorithm is general, memory efficient, and able to deliver excellent performance and scalability on both clusters and scalable shared memory systems. 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 and SUMMA). For example on 128 processors of the SGI Altix in multiplication of transposed square matrices 4000x4000, SRUMMA achieves 15.18 times higher aggregate GFLOPs performance level than ScaLAPACK pdgemm. For rectangular matrices (m=4000 n=4000 k=1000) on the Linux cluster with Myrinet, SRUMMA outperformed ScaLAPACK pdgemm by 48.8%. In all cases, the same serial matrix multiplication was used.

The paper is organized as follows. The next section describes the SRUMMA algorithm, its efficiency model, and implementation. In Section 3, variations of the algorithm are described to handle efficiently transposed and rectangular matrices, and analysis of techniques for dealing with different matrix shapes is presented. 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. Finally, summary and conclusions are given in Section 5.

2. Description of SRUMMA

At the high level, 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.

```
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:           $C_{ij} = C_{ij} + A_{ik} \times B_{kj}$
6:        }
7:     }
8:  }
```

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

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 or 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.

2. 1 Baseline Efficiency Model

Consider a matrix multiplication operation C = AB, where the order of matrices A, B, and C is m x k, k x n, and m x n, respectively. Let us denote: t_w - data transfer time per element, t_s - latency (or startup cost), P - number of processors, p x q - process grid in two-dimensional fashion i.e., P = p x q, and assume (similarly to other papers [7, 30]) 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

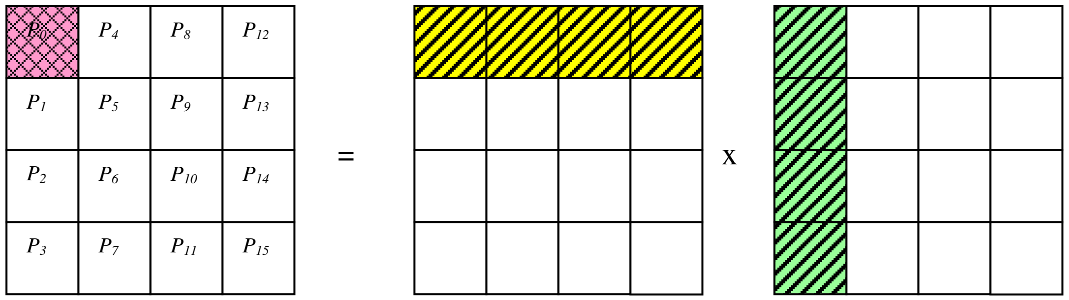


Figure 2: Matrix distribution example. In a 4 x 4 process grid, process P_0 needs blocks of matrix A from P_0, P_4, P_8, and P_{12}, and blocks of matrix B from P_0, P_1, P_2, and P_3.

in Figure 2. Each process owns a block of A, B and C matrix 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 and the time to *get* the blocks of matrices A and B.

T_{par_rma} = Computation Time (T_{comp}) + Communication Time (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, 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* $\frac{mk}{pq}$ } + {*latency/start-up cost*}

$$= \left(\left(\frac{mk}{pq}\right)t_w + t_s\right)q$$

Similarly, $T_{column\_comm} = \left(\left(\frac{nk}{pq}\right)t_w + t_s\right)p$

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

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

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

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

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

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

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

Overlapping communication with computation: 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 follows: $\omega = \left(1-\frac{T_{comp}}{T_{comm}}\right)$; if $\omega < 0$, $\omega = 0$

Introducing ω in (1), $T_{par\_rma} = \frac{N^3}{P} + \omega\left(2\frac{N^2}{P}t_w\right)\sqrt{P} + 2t_s\sqrt{P}$ (3)

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

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

2.2 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.

2.2.1 Cluster Version

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

1. Build a *list of tasks* corresponding to the block matrix multiplications in:

$$C_{ij} = \sum_{k=1}^{n_p} A_{ik} B_{kj} \tag{4}$$

   where, a task computes each of the $A_{ik}B_{kj}$ products.

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.

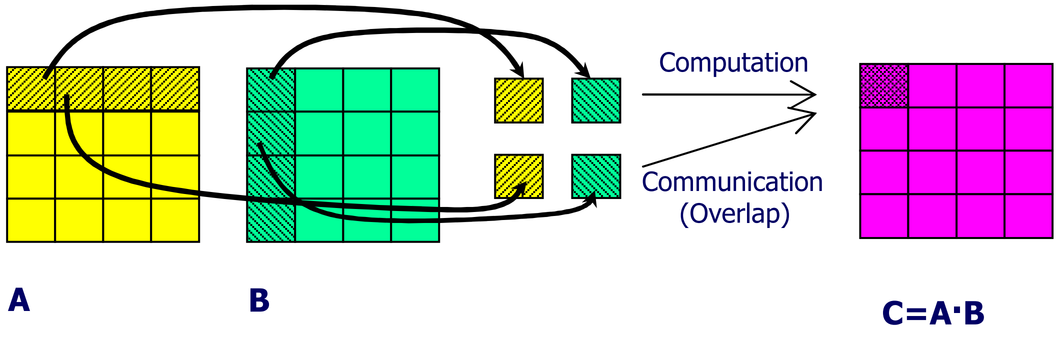


Figure 3: Overlapping communication with computation in matrix multiplication

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.

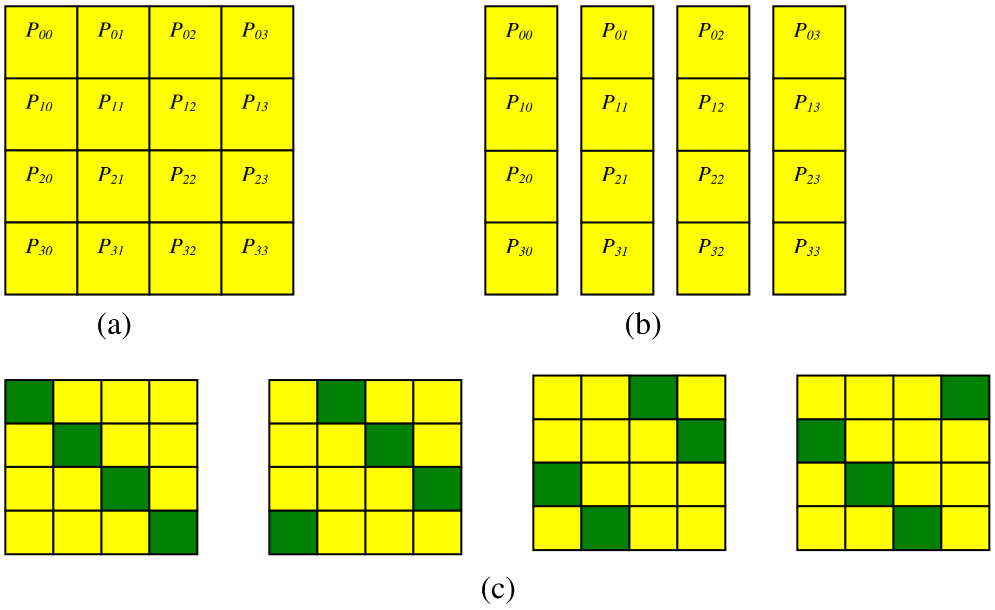

P_{00} P_{01} P_{02} P_{03}
P_{10} P_{11} P_{12} P_{13}
P_{20} P_{21} P_{22} P_{23}
P_{30} P_{31} P_{32} P_{33}
(a)
(b)
(c)

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

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 this indeed improves performance. For example, consider a matrix *A* that is distributed on a 4 x 4 processor grid (as shown in Figure 4a) on a 4-way SMP cluster, i.e., 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., as shown in Figure 4b. 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}, 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.

2.2.2 Shared Memory Version

The cluster algorithm running on a system with one shared memory communication domain reduces to shared memory version. However, this algorithm has two versions; 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.

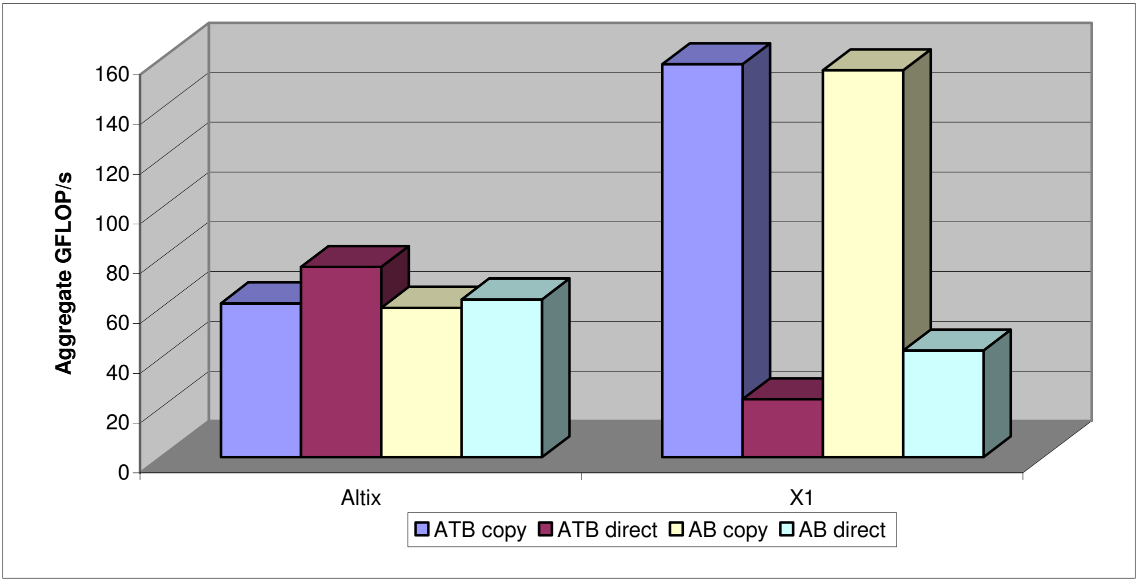


Figure 5: Performance of matrix multiplication (N=2000) on 16 processors using direct access and copy on the Cray X1 and the SGI Altix

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 versions actually increases for larger processor counts on the Altix).

2.3 Practical Implementation

Our current implementation of SRUMMA 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 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. On Linux clusters with Myrinet, to achieve maximum performance in the RMA communication 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.

3. Multiplication Kernels for Rectangular and Transposed Matrices

We developed several matrix multiplication kernels for achieving high performance for rectangular and transpose matrices. The appropriate algorithm is chosen based on the shapes of the matrices. For our analysis, we discuss the most representative and practical classes of rectangular and transpose matrix multiplication.

3.1 Strategies for Rectangular Matrices

For rectangular matrices, we select the most appropriate strategy based on the size and shape of the matrices. These strategies control how the load is distributed and how the parts of the matrices are accessed, i.e., they determine chunking of the data. SRUMMA supports three major chunk ordering schemes: first-order, second-order and third-order chunking. In first order chunking, a block of matrix A is decomposed column-wise into multiple chunks and a block of matrix B is decomposed row-wise as shown is Figure 6a. Let us assume the distribution in Figure 2. Process P_0 needs blocks of matrix A from P_0, P_4, P_8, and P_{12}, and blocks of matrix B from P_0, P_1, P_2, and P_3. In the first step, P_0 gets its local blocks and does the sequential *dgemm* locally. In the next step, P_0 gets a block of matrix A and a block of matrix B from remote processes P_4 and P_1 respectively. If the buffer size (size of the temporary buffers) is big enough to hold these matrices, then the local computation of *dgemm* is a one step process. If the buffer is not big enough, then these matrix blocks are divided into chunks. If matrix A and B has s chunks each, then the number of steps to compute this block locally is in O(s). In second order chunking scheme

(Figure 6b), the block of matrix A is decomposed row-wise panels and block of matrix B is decomposed column-wise panels. Here the number of steps to compute *dgemm* locally is O(s^2). In third-order chunking (Figure 6c), the blocks of matrix A and B are decomposed into square chunks. The number of steps to compute *dgemm* locally is O(s^3) in this scheme.

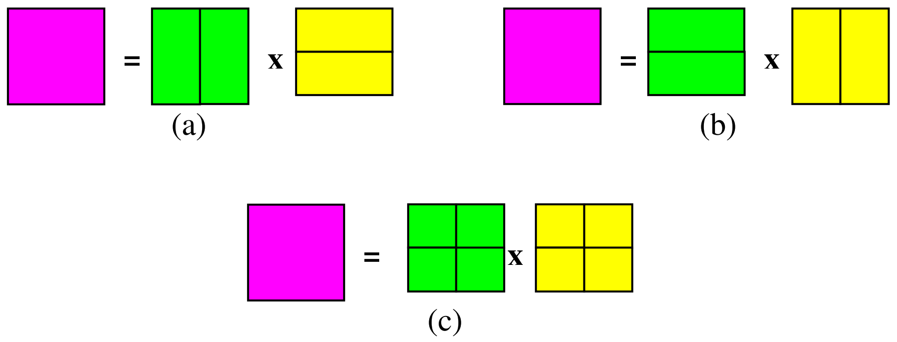


Figure 6: (a) First-order chunking. (b) Second-order chunking. (c) Third-order chunking

3.2 Rectangular Matrix Multiplication

The various cases of rectangular matrix multiplication are represented in Figure 7:

- *m* is small, *n* and *k* are large
- *k* is small, *m* and *n* are large
- *n* is small, *m* and *k* are large

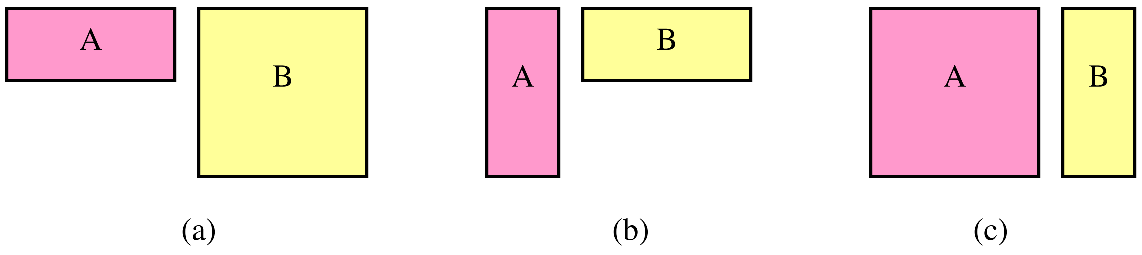


Figure 7: (a) represents case *i*, i.e. *m* is small, *n* and *k* are large. (b) *k* is small, *m* and *n* are large. (c) *n* is small, *m* and *k* are large.

***Case (i): m is small, n and k are large*:** For simplicity, let us assume the matrices *A*, *B* and *C* are distributed among the processes as shown in figure 8. In the RMA *get* operation, if the temporary buffer is not big enough to fit the chunks from *A* or *B*, then the chunks can be divided into multiple panels row-wise or column-wise as shown in figure 8a. In SRUMMA, a block distributed to a process is stored in column major order. Since n and k are large, the block corresponds to matrix B is decomposed into multiple panels column-wise, thus all the matrix elements that belong to a panel are stored in a contiguous fashion. Moreover, accessing row-wise panels of matrix A is relatively less expensive (as m is small) when compared to accessing row-wise panels of matrix B. Thus, the second-order chunking scheme can perform better here, as paneling of the matrix A and B correspond to the second-order chunks.

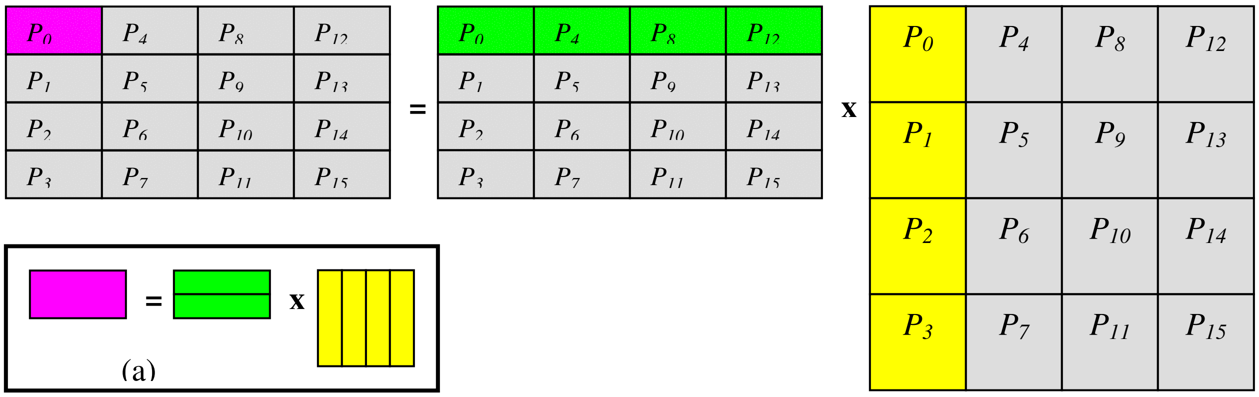

P_0 P_4 P_8 P_{12}
P_1 P_5 P_9 P_{13}
P_2 P_6 P_{10} P_{14}
P_3 P_7 P_{11} P_{15}
=
x
(a)

Figure 8: In a 4x4 process grid, process P_0 needs blocks of matrix A from P_0, P_4, P_8 and P_{12}, and blocks of matrix B from P_0, P_1, P_2 and P_3 to compute C locally. (a) second order chunking is used if the chunks are too big to fit in the buffer.

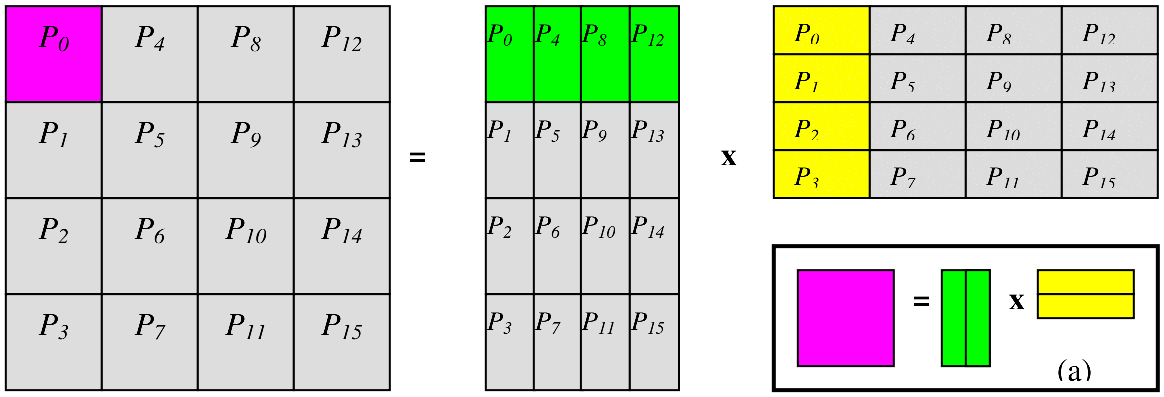
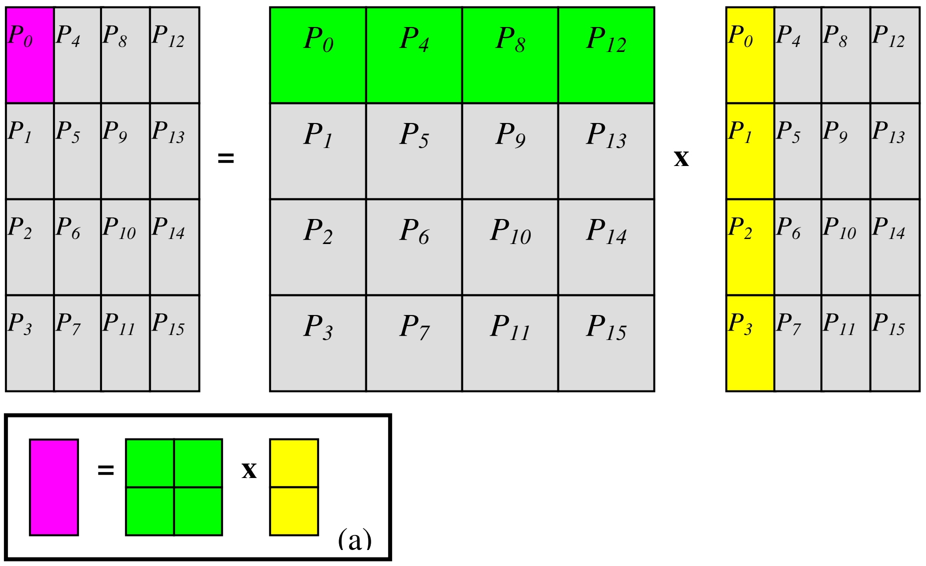

P_0 P_4 P_8 P_{12}
P_1 P_5 P_9 P_{13}
P_2 P_6 P_{10} P_{14}
P_3 P_7 P_{11} P_{15}
=
x
(a)

Figure 9: In a 4x4 process grid, process P_0 needs blocks of matrix A from P_0, P_4, P_8 and P_{12}, and blocks of matrix B from P_0, P_1, P_2 and P_3 to compute C locally. (a) First order chunking is used if the chunks are too big to fit in the buffer.

***Case (ii): k is small, m and n are large*:** Figures 9 illustrates this case. As matrices A and B are stored in column major order, and k is small and m is large, dividing matrix A column-wise performs better (Figure 9b).

***Case (iii): n is small, m and k are large*:** In this case, rectangular matrix multiplication is performed as illustrated in Figure 10. As m and n are large, it is better to use square shaped chunks for optimum performance as mentioned in section 2.1. In this case, decomposing the matrix into multiple rectangular panels (especially *A*) on networks that do not support good hardware support for noncontiguous data (i.e., scatter/gather) this results in multiple access to non-contiguous data, and so more expensive when compared to square chunks. Thus, this case uses the third order chunking scheme (Figure 10a).



P_0 P_4 P_8 P_{12}
P_1 P_5 P_9 P_{13}
P_2 P_6 P_{10} P_{14}
P_3 P_7 P_{11} P_{15}
=
x
(a)

Figure 10: In a 4x4 process grid, process P_0 needs blocks of matrix *A* from P_0, P_4, P_8 and P_{12}, and blocks of matrix *B* from P_0, P_1, P_2 and P_3 to compute *C* locally. (a) Third order chunking is used if the chunks are too big to fit in the buffer.

3.3 Transpose Matrix Multiplication Algorithm

There are three flavors of transpose matrix multiplication: 1) $C = A^T. B$ 2) $C = A. B^T$ and 3) $C = A^T. B^T$. The transposed algorithm is shown in Figure 11. Recall from Figure 2 that in order to compute a block of matrix *C* locally, process P_0 needs blocks of matrix A from P_0, P_4, P_8, and P_{12}, and blocks of matrix B from P_0, P_1, P_2, and P_3. Let us consider the case C = A^T. *B,* where *A* is transposed and *B* is not. In this case, process P_0 gets blocks of matrix A from P_0, P_1, P_1, and P_3, and blocks of matrix B from P_0, P_1, P_2, and P_3 as shown in figure 11. After getting the first set of blocks, process P_0 computes local transpose matrix multiply using the vendor optimized sequential BLAS library. Thus, SRUMMA transposes from process point of view rather than from matrix point of view (i.e. transposing the entire matrix, which is

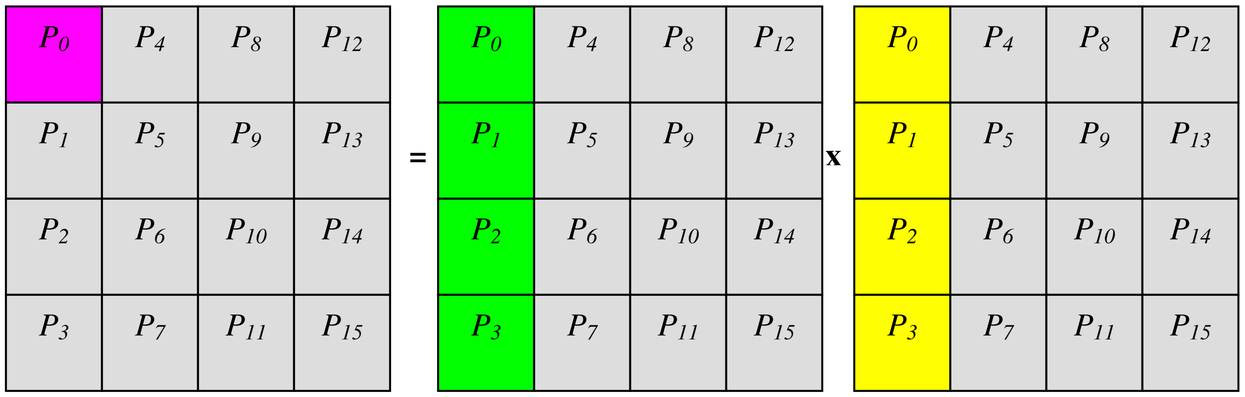

P_0 P_4 P_8 P_{12}
P_1 P_5 P_9 P_{13}
P_2 P_6 P_{10} P_{14}
P_3 P_7 P_{11} P_{15}
=
x

Figure 11: transpose matrix multiply $C = A^T. B$. In a 4x4 process grid, process P_0 needs blocks of matrix A from P_0, P_1, P_2 and P_3 (transpose), and blocks of matrix B from P_0, P_1, P_2 and P_3 to compute C locally.

expensive and leads to poor scalability).

4. Experimental Results

In the previous paper [1] we discussed the effectiveness of the SRUMMA for square matrices. In this section, we present how the optimizations for rectangular and transpose matrices perform in practice. The numerical experiments were conducted on the following platforms:

- Linux cluster based on dual 2.4-GHz Intel Xeon nodes and Myrinet-2000 network
- 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 PBLAS/ScaLAPACK Version 1.7, and SUMMA. However, to save space we are reporting ScaLAPACK results only, as it is the most commonly used parallel linear algebra library. Moreover, SUMMA is used in practice in ScaLAPACK/PBLAS [20]. The same *dgemm* (double precision serial matrix multiplication) routines from vendor optimized math library (-lscs for Altix, -lmkl for Xeon) were used in all three parallel algorithms. Optimum block sizes were chosen empirically for all matrix sizes and processor counts. Section 4.3 presents the performance of communication protocols including the potential for overlapping communication with communication.

4.1 Rectangular Matrices

As mentioned in Section 3.2, SRUMMA selects the appropriate matrix multiplication kernel based on the shape of the matrices. For example, SRUMMA selects the strategy with second order chunking for the matrices represented in Figure 7a. To validate our theoretical analysis, we experimentally evaluated the performance of all the cases of rectangular multiplication (in Section 3.2) with different matrix multiplication kernels. Matrices with larger dimension 4000 and smaller dimension 1000 are chosen. This medium size dimension (i.e. 4000) is chosen as computation pre-dominates communication in larger size matrices, and communication pre-dominates computation in smaller size matrices, for our selected

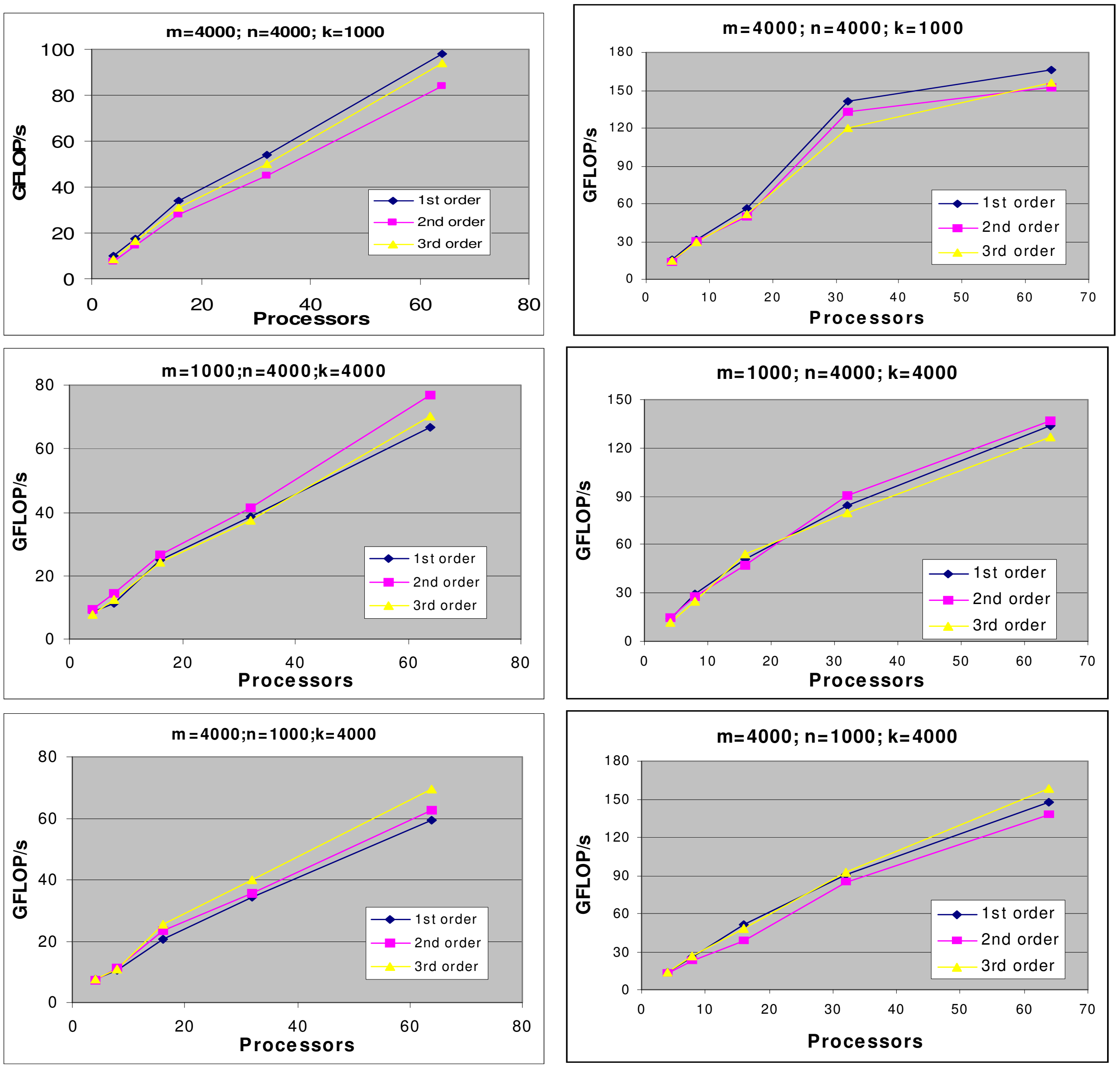


Figure 12: Performance of matrix multiplication with 1st, 2nd and 3rd order chunking for different matrix shapes on Linux cluster (left) and SGI Altix (right)

processor range (64 processors). Consider a matrix multiplication operation C = AB, where the order of matrices A, B, and C is *m* x *k*, *k* x *n*, and *m* x *n*, respectively. The performance of 1st, 2nd and 3rd order matrix multiplication kernels for various matrix shapes is shown in Figure 15. For example, second order matrix multiplication kernel in SRUMMA performs better for matrix multiplication of shape m=1000, n=4000 and k=4000 (Figure 12).

We also include performance results of SRUMMA and ScaLAPACK. 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 report results for the following cases:

i. m = 2000; n = 2000; k = 1000

ii. m = 1000; n = 1000; k = 2000

iii. m = 4000; n = 4000; k = 1000

iv. m = 1000; n = 1000; k = 4000

Figures 13 and 14 show the performance of SRUMMA and ScaLAPACK on a cluster (Linux) and a shared memory system (Altix), and indicate that SRUMMA performs better than ScaLAPACK in most of the cases. SRUMMA obtained a peak performance of 620 GFLOP/s (4.8 GFLOP/s per processor) on the SGI Altix with 128 processors, for case (i).

4.2 Transpose Matrices

We also measured the performance of matrix multiplication using SRUMMA and ScaLAPACK for transposed matrices. Three variants were considered: $C = A^TB$ (i.e., A is transposed, B is not), $C = AB^T$ and $C = A^TB^T$. $C = A^TB$, $C = AB^T$ and $C = A^TB^T$ are referred to as TN, NT, and TT, respectively in Figures 13 and 14. As shown in Figures 13 and 14, SRUMMA consistently outperformed ScaLAPACK/PBLAS *pdgemm* on clusters and shared memory systems in most of the 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 three 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.

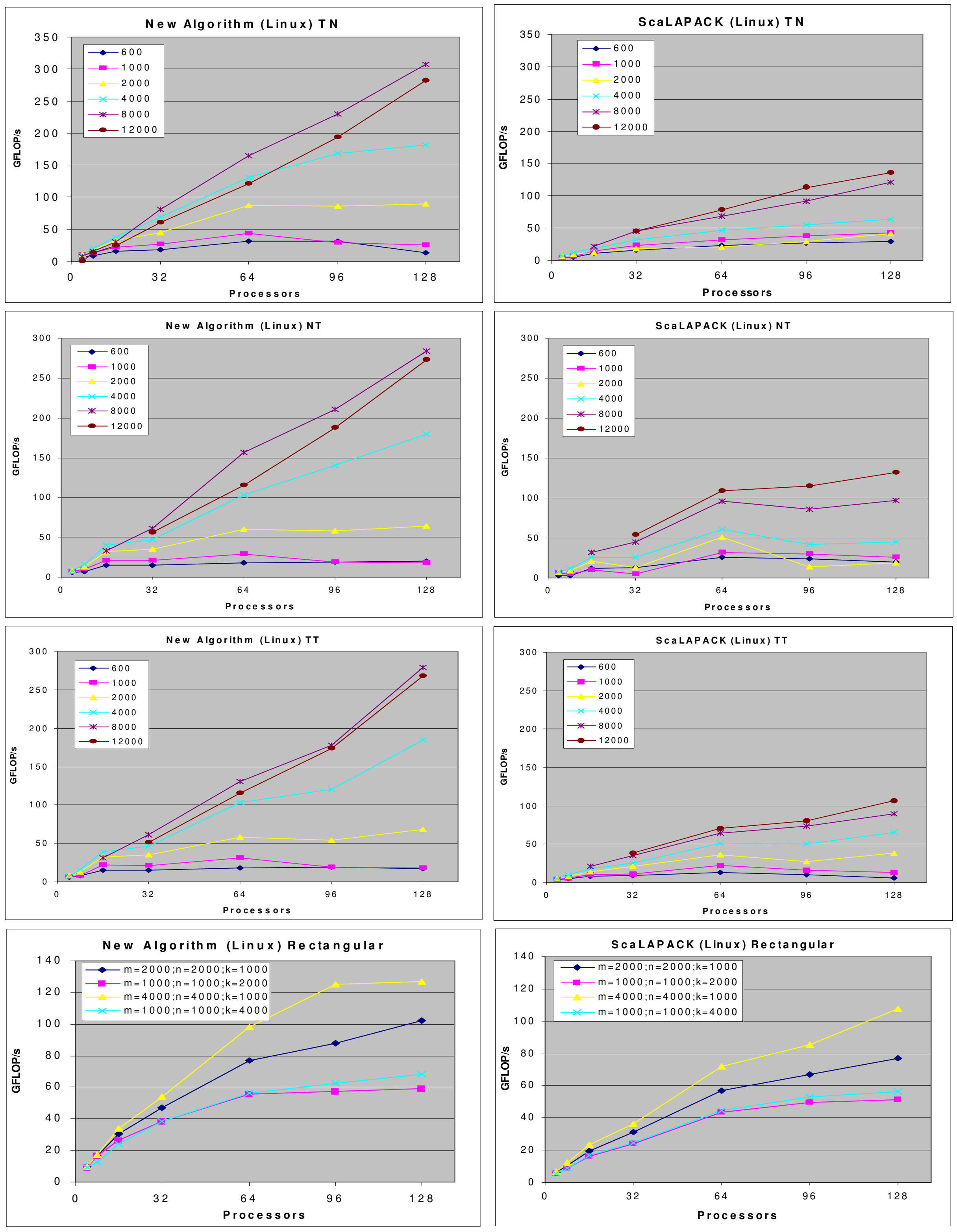


Figure 13: Performance of transpose cases (TN, NT, TT) and rectangular matrices on the Linux cluster (New Algorithm is SRUMMA).

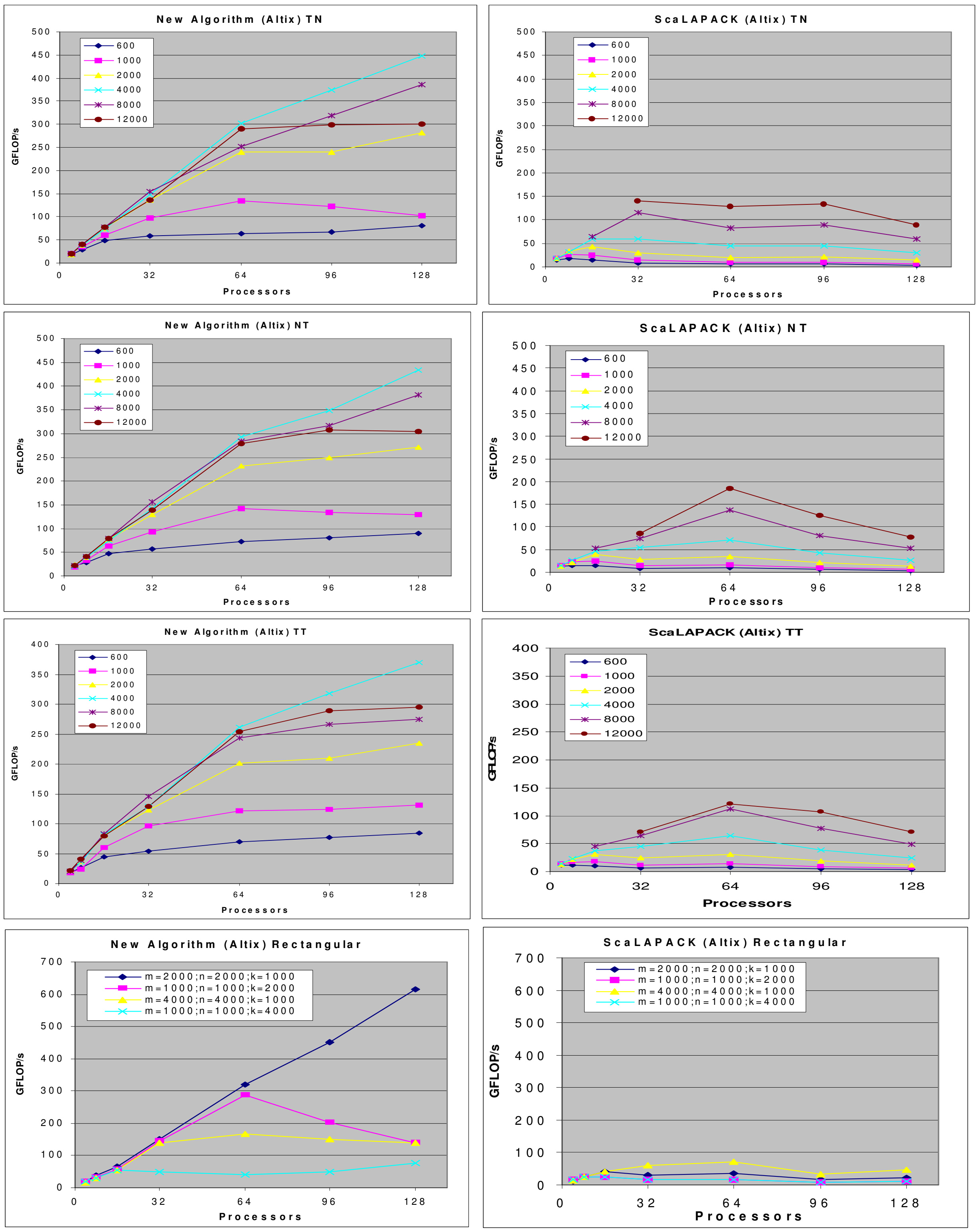


Figure 14: Performance of transpose cases (TN, NT, TT) and rectangular matrices on Altix.

The Linux results in Figure 14 indicate that 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.

4.3 Impact of communication protocols

First, we investigated the performance of MPI *send/receive* operations and the ARMCI *get* operation on the Linux cluster. SGI Altix is not included in this study as we directly access shared memory in our matrix multiplication. Since shared memory is the fastest communication protocol available on shared memory systems, it is not surprising that SRUMMA outperforms *pdgemm* algorithm implemented on top of message passing. However, to understand the performance differences between the two versions of matrix multiplication algorithm on clusters we performed several tests to measure the role of the underlying communication protocols.

Figure 16 shows that performance of these protocols is close, 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. Unlike the message-passing implementation of the matrix multiplication in ScaLAPACK, our algorithm is using *nonblocking RMA*, which offers a potential for overlapping communication with computations [38]. We measured this potential for ARMCI and MPI on our cluster platform (see Figure 15). In comparison to MPI non-blocking i*send/irecv*, ARMCI non-blocking *get* offers almost 99% overlap for medium- and larger-sized messages. Similarly to other studies [39, 40], 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]. The new matrix multiplication algorithm was tested when enabling and disabling the zero-copy implementation [29] of the ARMCI get operation. Figure 17 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.

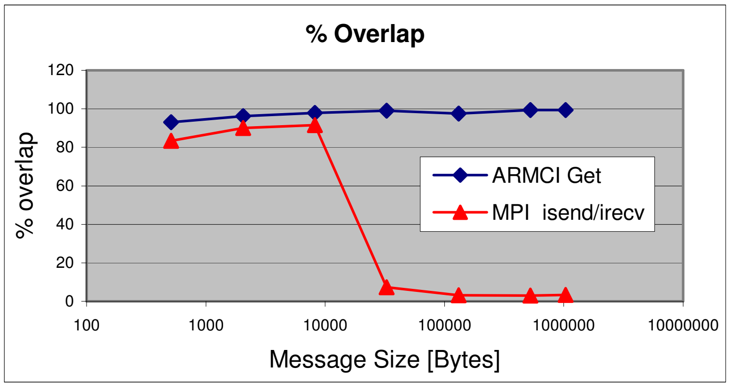
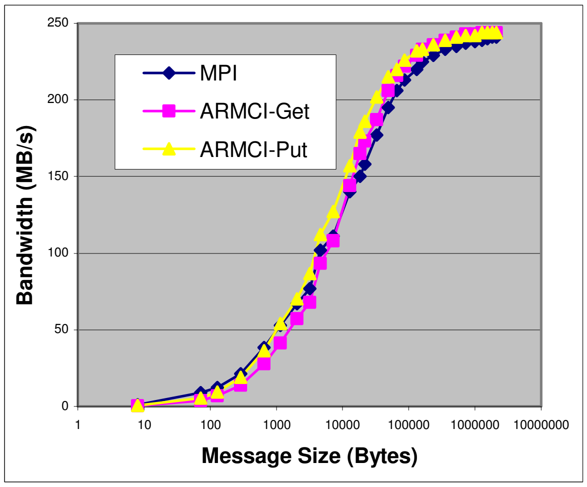


Figure 15: Degree of overlap as a function of message size in ARMCI and MPI on Linux cluster



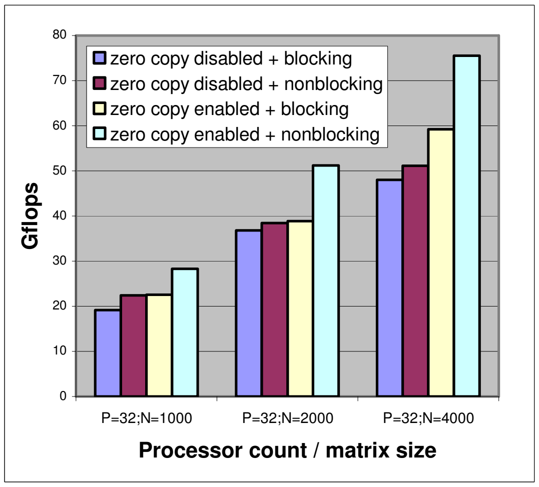


Figure 16: Performance of MPI and ARMCI_Get on Linux cluster (Myrinet Interconnect)

Figure 17: Performance of matrix multiplication on Linux Cluster (Myrinet Interconnect) with enabled or disabled zero-copy protocol

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:

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

$T_{par\_rma} = \frac{N^3}{P} + \zeta$, where $\zeta << \frac{N^3}{P}$ for medium/larger problem sizes.

5. Summary and Conclusions

This paper described SRUMMA parallel algorithm for a dense matrix multiplication with extensions to deal efficiently with rectangular and transposed matrices. 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 ScaLAPACK for transposed and rectangular matrices. Some of its best cases are presented in Table 1. Our plans for future work include multiplication of matrices with irregular distribution.

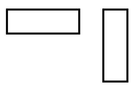
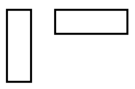
Matrix Size	**Processors**	**Case (Platform)**	**SRUMMA (GFLOP/s)**	**ScaLAPACK (GFLOP/s)**
600x600	128	$C=A^TB^T$(Linux)	16.64	6.4
4000x4000	128	$C=A^TB^T$ (Altix)	369	24.3
m=4000;n=4000; k=1000 (Rectangular)	128	 (Linux)	160	107.5
m=1000;n=1000; k=2000 (Rectangular)	64	 (Altix)	288	17.28

Table 1: SRUMMA best cases.

References

[1] M. Krishnan, J. Nieplocha, ”SRUMMA: A Matrix Multiplication Algorithm Suitable for Clusters and Scalable Shared Memory Systems”, to appear in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS’04),* Santa Fe, NM, 2004.

[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, 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”, in *Proceedings of the 1993 International Conference on Parallel Processing*, 1993.

[8] C. Lin and L.Snyder, “A matrix product algorithm and its comparative performance on hypercubes”, *Proc. Scalable High Performance Computing Conference SHPCC92*, 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 *Proceedings of the Scalable Parallel Libraries Conference*, IEEE Computer Society Press, 1994, pp. 142-149.

[11] C. T. Ho, S. L. Johnsson, A. Edelman, Matrix multiplication on hypercubes using full bandwidth and constant storage, *Proc. 6 Distributed Memory Computing Conference*. 1991.

[12] H. Gupta and P. Sadayappan, “Communication Efficient Matrix Multiplication on Hypercubes”, in *Proceedings of the Sixth ACM Symposium on Parallel Algorithms and Architectures*, 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), pp. 345–389, May 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) . 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, April 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*, Society for Industrial and Applied Mathematics, 1997, Philadelphia, PA.

[22] J. Choi, “A Fast Scalable Universal Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers”, in *Proceedings of the 11th International Parallel Processing Symposium (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, 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. Kendall, “Mixed Mode Matrix Multiplication”, in *Proceedings of IEEE International Conference on Cluster Computing (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, November, 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, in *Proceedings of 17th Annual ACM International Conference on Supercomputing*, San Francisco, June, 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 International Conference Cluster on Computing (CLUSTER’03)*, 2003.
[30] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, Addison Wesley, 2003.
[31] Cray Online documentation. 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, B. Carpenter, ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems, *Proc. 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”, in *Proceedings of the Communication Architecture for Clusters Workshop of IPDPS*, 2002.
[36] ORNL Evaluation of Early Systems Webpage. http://www.csm.ornl.gov/evaluation
[37] ORNL Tom Dunigan’s Evaluation of Early Systems Webpage. http://www.csm.ornl.gov/~dunigan/
[38] V. Tipparaju, M. Krishnan, J. Nieplocha, G. Santhanaraman, and D.K. Panda, “Exploiting Non-blocking Remote Memory Access Communication in Scientific Benchmarks”, *Proceedings of the HiPC,* 2003.
[39] B. Lawry, R. Wilson, A. B. Maccabe, and R. Brightwell, “COMB: A Portable Benchmark Suite for Assessing MPI Overlap”, *IEEE Cluster*, 2002.
[40] 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*, March 1999.