# Memory Efficient Parallel Matrix Multiplication Operation for Irregular Problems

Manojkumar Krishnan
Pacific Northwest National Laboratory
P.O.Box 999, K7-90
Richland, WA 99352
1-509-372-4206
manoj@pnl.gov

Jarek Nieplocha
Pacific Northwest National Laboratory
P.O.Box 999, K7-90
Richland, WA 99352
1-509-372-4469
Jarek.Nieplocha@pnl.gov

## ABSTRACT
Regular distributions for storing dense matrices on parallel systems are not always used in practice. In many scientific applicati RUMMA) [1] to handle irregularly distributed matrices. Our approach relies on a distribution independent algorithm that provides dynamic load balancing by exploiting data locality and achieves performance as good as the traditional approach which relies on temporary arrays with regular distribution, data redistribution, and matrix multiplication for regular matrices to handle the irregular case. The proposed algorithm is memory-efficient because temporary matrices are not needed. This feature is critical for systems like the IBM Blue Gene/L that offer very limited amount of memory per node. The experimental results demonstrate very good performance across the range of matrix distributions and problem sizes motivated by real applications.

## Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent Programming – *Parallel programming*.

## General Terms
Algorithms, Performance, Design (c) 2005 Association for Computing Machinery.

## Keywords
Parallel Matrix Multiplication, Parallel Linear Algebra, Irregular Distribution, SRUMMA, Remote Memory Access, Global Arrays, Parallel programming.

## 1. INTRODUCTION
Matrix multiplication is used in many areas of science and technology. In fact, for many scientific applications, it represents one of the most important linear algebra operations. Computer vendors have optimized the standard serial dense 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 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.


*CF'06*, May 3-5, 2006, Ischia, Italy.


In earlier studies [2-22], 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. Contemporary architectures differ in several key aspects from the earlier MPP systems. Regardless of the processor architecture, 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 a high-performance 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 performance results when compared 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 the fastest communication protocol available, especially when implemented in hardware as zero-copy RDMA write/read operations (e.g., Infiniband, Quadrics, and Myrinet). RMA is often used to implement point-to-point MPI send/receive calls [27,28]. To address the growing gap between processor and network speed, SRUMMA relies on nonblocking 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; that is, 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). Unlike the OpenMP studies [23, 24] that relied on a compiler-based and 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. A comparison to the standard matrix multiplication interface, pdgemm in PBLAS [20] and SUMMA [19], revealed that for square matrices with regular distributions, SRUMMA achieves consistent and very competitive performance on the four architectures used in the study [1]: 16-way (IBM SP) and 2-way (Linux/Xeon) nodes, SGI Altix, and the Cray X1 with its partitioned shared memory hardware. SUMMA [19] is also used in practice in the pdgemm routine in PBLAS [20], which is a building block of ScaLAPACK [21].

However, regular distributions of matrices on parallel systems are not always used in practice, and therefore, the parallel matrix multiplication algorithms designed for regular problems cannot be used directly. In many scientific applications, matrix distribution is based on the underlying physical problem which might involve variable block sizes on individual processors [41]. For example, in computational chemistry matrix distribution is chosen based on the basis set of the atoms in molecular systems to exploit data locality and maximize performance of the Fock construction algorithm [39] which is a key element of the self-consistent field (SCF) calculations. The irregular distribution of the matrix can significantly impact the performance of matrix multiplication operation and prevent users from using the available matrix multiplication algorithms directly. The standard approach for multiplying irregularly distributed matrices has been based on multiplying regularly distributed temporary arrays and requires data redistribution. In addition to the extra communication involved in the redistribution process, the main issue with this approach has been the *extra memory consumption required for the temporary arrays*. This issue is especially important for emerging massively parallel systems like the IBM Blue Gene/L, which are based on dense packaging and offer very limited memory expansion options. Addressing this limitation without compromising the performance is the primary goal of our work.

In this paper we generalize the SRUMMA algorithm to handle irregularly distributed matrices efficiently without relying on array temporaries and data redistribution. The main contribution of this work is the memory-efficient distribution-independent algorithm that delivers performance as good as that provided by the standard redistribution-based approach, and therefore is very well suited for large matrices and systems with memory constraints that cannot handle temporary arrays [42]. We also describe and compare the two approaches for irregularly distributed matrices: 1) redistribution of irregularly distributed matrices to the regular form followed by the regular matrix multiplication, 2) proposed distribution independent matrix multiplication based on logical blocking of the result matrix. The SRUMMA algorithm with proposed extensions is general, memory-efficient, and able to deliver excellent performance and scalability on modern systems.

The paper is organized as follows. Section 2 describes the SRUMMA algorithm, its efficiency model, and implementation. In Section 3, the distribution independent matrix multiplication algorithm is presented. Section 4 describes and analyzes performance results of SRUMMA matrix multiplication for various matrix distributions from two application areas as well as results for the communication operations used in the implementation. The paper is concluded in Section 5.

# 2. OVERVIEW OF THE BASELINE SRUMMA ALGORITHM

At the high level, SRUMMA follows the serial block-based matrix multiplication (see Figure 1) by assuming the regular block distribution of 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 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 another 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 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 as in Cannon's algorithm) based on message passing makes the overall algorithm more asynchronous and thus more suited for the execution environments in which the computational threads share a CPU with other processes and system daemons (e.g., on commodity clusters). This is because synchronization amplifies performance degradations resulting from the nonexclusive use of the processor by the application.

## 2.1 Efficiency Model

Consider a matrix multiplication operation C = AB in which the order of matrices A, B, and C is $m$ x $k$, $k$ x $n$, and $m$ x $n$, respectively. Let us 1) denote that $t_w$ is the data transfer time per element, $t_s$ is the latency (or startup cost), P is the number of processors, $p$ x $q$ is a process grid in two-dimensional fashion i.e., P = $p$ x $q$, and 2) assume (similarly to other papers [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.

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

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

Each process owns a block of A, B and C matrix of size $\frac{m}{p}\times\frac{k}{q}$, $\frac{k}{p}\times\frac{n}{q}$, and $\frac{m}{p}\times\frac{n}{q}$ 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 communication time to *get* the row and column blocks of matrices A and B ($T_{comm} = T_{row\_comm} + T_{column\_comm}$). 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} = $\left(\left(\frac{mk}{pq}\right)t_w + t_s\right)q$

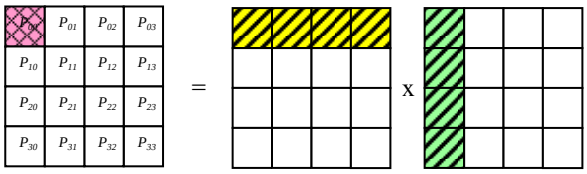


**Figure 2**. Matrix distribution example: In a 4 x 4 process grid, process P00 needs blocks of matrix A from P00, P01, P02, and P03, and blocks of matrix B from P00, P10, P20, and P30.

Similarly, $T_{column\_comm} = \left(\left(\frac{nk}{pq}\right)t_w + t_s\right)p$ ,thus, $T_{par\_rma} = \frac{mnk}{P} + \left(\left(\frac{mk}{pq}\right)t_w + t_s\right)q + \left(\left(\frac{kn}{pq}\right)t_w + t_s\right)p$

For simplicity, we assume $m=n=k=N$ *and* $p=q=\sqrt{P}$ , then

$$T_{par\_rma} = \mathrm{O}\left(\frac{N^3}{P}\right) + \mathrm{O}\left(\frac{N^2}{\sqrt{P}}\right) + \mathrm{O}\left(\sqrt{P}\right) \qquad (1)$$

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

The isoefficiency function of this algorithm is O ($P^{3/2}$), which is 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: ω = $\left(1-\frac{T_{comp}}{T_{comm}}\right)$ ; if ω < 0, ω = 0

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

If $T_{comp} >= T_{comm}$ (i.e., 100% overlap), (2) reduces to $T_{par\_rma} = \frac{N^3}{P} + 2t_s\sqrt{P} = \mathrm{O}\left(\frac{N^3}{P}\right) + \mathrm{O}\left(\sqrt{P}\right)$ (3)

## 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 because of the reduced synchronization cost. We will first describe the implementation of the algorithm for clusters; then we will discuss special considerations for the scalable shared memory systems.

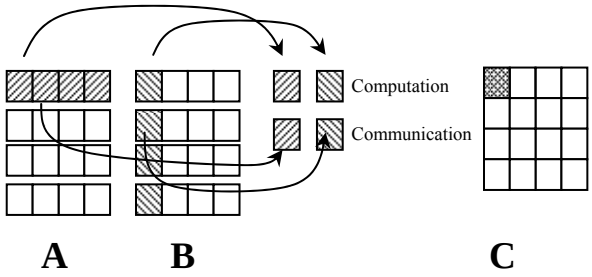


**Figure 3**. Using two sets of buffers to overlap communication and computation in matrix multiplication

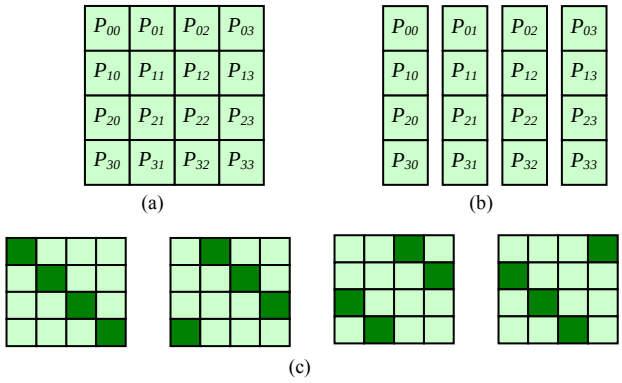


**Figure 4**. Pattern of getting blocks on a 4-way SMP cluster to reduce communication contention.

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: $C_{ij} = \sum_{k=1}^{n_p} A_{ik} B_{kj}$ 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 action is taken 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. Because 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 $B_{kj}$ to execute the current task.
   - Call serial matrix multiplication 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* are used for computation and *B1* is used for receiving data. Overlapping communication with computation is achieved in all but the first step.

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. For example, consider a matrix *A* that is distributed on a 4x4 processor grid (as shown in Figure 4a on a 4-way SMP cluster, that is, node 1 has processors $P_{00}$, $P_{10}$, $P_{20}$, *and* $P_{30}$; node 2 has processors $P_{01}$, $P_{11}$, $P_{21}$, *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 Figures 2 and 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 four processors are sharing the network 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 [1]. Figure 4c represents the pattern of getting blocks by processors in node 1.

The cluster algorithm running on a system with one shared memory communication domain reduces to a shared memory version. However, this algorithm has two versions, and the one used depends on whether remote shared memory is locally cacheable. For example, the Cray X1 the shared memory cannot be cached because of 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 in which shared memory data can be cached. The matrix multiplication does not require explicit memory copies: the appropriate blocks of matrix A and B are passed directly to the serial matrix multiplication subroutine.

The current implementation of SRUMMA relies on the portable ARMCI library [32,34,35] 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. 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, 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.

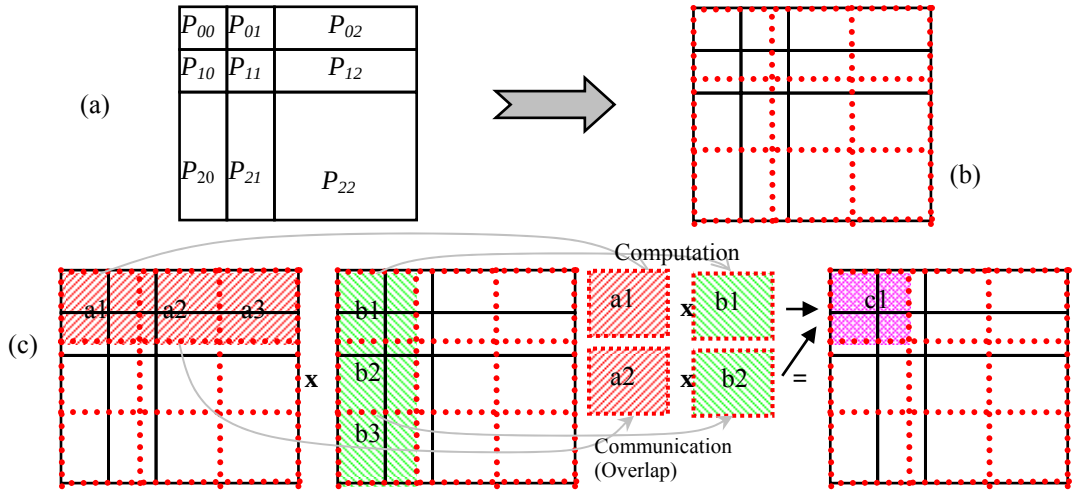


**Figure 5: (a)** Matrix distributed on a 3x3 process grid with uneven block sizes. **(b)** Matrix is logically partitioned so that it is load balanced. **(c)** Now each process works on its logical partition. Process $P_{00}$ gets the logical blocks *a1, a2, a3* and *b1, b2, b3* to compute its logically owned partition *c1*.

## 3. DISTRIBUTION INDEPENDENT ALGORITHM

In previous work, we extended SRUMMA to handle transpose and rectangular matrices [36]. In this paper, we propose a distribution-independent [42] algorithm to handle irregularly distributed matrices. The algorithm described in Section 2 is extended to accomplish this. To improve performance for irregular distributions, the current approach relies on two techniques 1) logical block partitioning and 2) exploiting data locality.

Regular distribution for distributing a matrix on parallel systems is not always used in practice. In many scientific applications, the matrix distribution is based on the underlying physical problem which might involve variable block sizes on individual processors [41]. This irregular distribution of matrices leads to load imbalance, a major performance degradation factor in many applications. The standard approach for multiplying irregularly distributed matrices has been based on multiplying regularly distributed temporary arrays and requires data redistribution. We will refer to this standard approach as *copy-based* approach. Our proposed algorithm addresses, not only the load imbalance, but also the following two major issues,

- extra communication involved in the redistribution process. In the best case (matrices are regular), the redistribution cost is zero, and in the worst case (a matrix resides in only one processor), the redistribution cost is maximum. Therefore, the redistribution cost is proportional to the degree of irregularity (i.e. distribution of a matrix among processors).
- extra memory consumption required for the temporary arrays. This issue is especially important for emerging massively parallel systems like the IBM Blue Gene/L, which are based on dense packaging and offer very limited memory expansion options. In the proposed algorithm, apart from the memory required to store the matrices, two temporary buffers of fixed size are used by each process in the communication step. This temporary buffer size is independent of the block size or matrix size. If the block size is too big to fit into the temporary buffer, then the blocks are sub-divided to fit into the temporary buffer. There are two temporary buffers used in the proposed algorithm because of the overlapping of communication and computation. Thus, the proposed algorithm for irregular matrix multiplication has similar memory requirements as regular SRUMMA [1]. For the sake of simplicity, and ease of explanation, we assume that the block size is the same as the buffer size, and we will refer these temporary buffers as blocks or logical blocks. In the *copy-based* algorithm, three temporary regular matrices (i.e. *A*, *B* and *C*) are created (i.e. O (N^2/P) extra memory than the proposed algorithm, where N is the matrix size and P is number of processors). The irregular matrices are redistributed [40,43] to these regular matrices, and performed regular SRUMMA matrix multiplication.

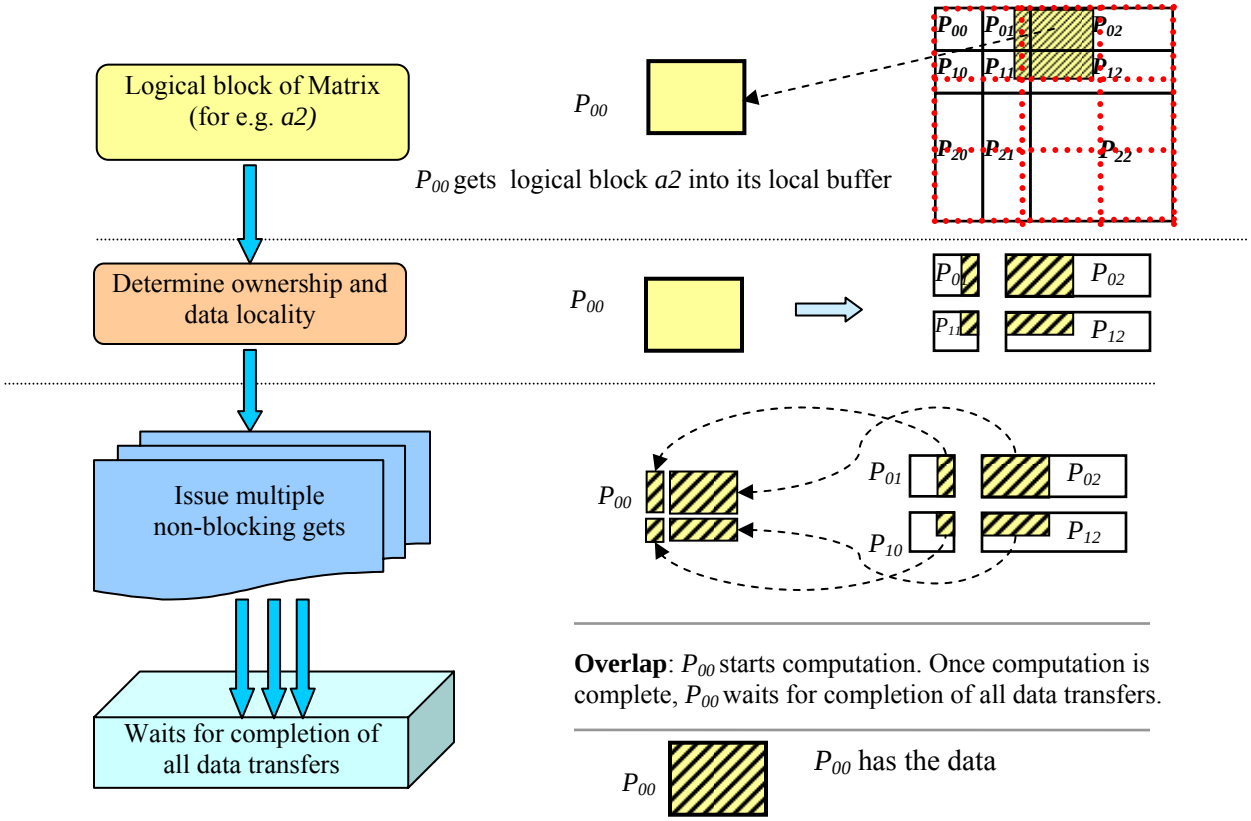


**Figure 6:** Process $P_{00}$ gets the logical chunk *a2* from processes $P_{01}$, $P_{02}$, $P_{11}$, and $P_{12}$

Consider a matrix multiplication operation C = AB, in which A, B, and C are distributed with variable block sizes across processors. Assuming the distribution of C matrix as shown in Figure 5a, each matrix is logically partitioned (Figure 5b) such that all the processors have almost the same logical block size, independent of the underlying distribution. As shown in Figure 5c, process $P_{00}$ gets the logical blocks *a1, a2, a3* and *b1, b2, b3* to compute *c1*. Similarly other processes attempt to compute their logically owned blocks. Each process is aware of the locality of all

the distributed matrices. Once the physical location of the logical block in the distributed/partitioned address space is determined, indices corresponding to where the logical block is located need to be determined. When this information is available, multiple non-blocking *get* calls are made, one for each remote destination that holds a part of the data. After all the calls are issued, they are waited upon until completed. By issuing all the calls first and then waiting on their completion, a significant amount of overlap can be achieved. For example, process $P_{00}$ gets the logical block *a2* from processes $P_{01}$, $P_{02}$, $P_{11}$, and $P_{12}$ by issuing four non-blocking calls to each of these processes while doing useful computation with the previously received blocks *a1 and b1*, and waiting for all the calls to complete as shown in Figures 5c and 6.

The algorithm attempts to hide most of the communication time by overlapping communication with computation. This applies to get operations that transfer blocks of A and B matrices as well as put operations that write the corresponding blocks of the result matrix C. The assignment of the logical blocks of matrices to individual processors is determined at run-time to achieve load balancing. To reduce communication contention on clusters we have to modify the diagonal shift algorithm (Section 2.2) to access logical matrix blocks that can be spread across multiple processors (Figure 6). The algorithm is applied to logical matrix block rather than physical processors.

# 4. EXPERIMENTAL RESULTS

The effectiveness of the SRUMMA for regularly distributed square matrices on multiple platforms has been discussed in [1]. In this section, we present and analyze the performance of the matrix multiplication operation for irregular problems. The numerical experiments were conducted on the following platforms at Pacific Northwest National Laboratory: 1) Linux cluster based on dual 1.5-GHz Intel Itanium-2 nodes and Quadrics QsNetII network (Elan4), 2) SGI Altix 3000, shared-memory NUMA system with 128 1.5-GHz Intel Itanium-2 CPUs. We used irregularly distributed matrices from computational chemistry and astrophysics applications. In addition to performance advantages of SRUMMA over *pdgemm* (ScalaPACK/PBLAS) for regularly distributed matrices as reported in [1], we also present results on the Linux/Quadrics cluster. Furthermore, we report performance of communication operations used by SRUMMA on that cluster.

## 4.1 Performance for regular distribution

In our previous work, was demonstrated that SRUMMA delivers superior performance over ScaLAPACK pdgemm on the IBM SP, SGI Altix, Cray X1 and Linux Xeon cluster with Myrinet. The work described in this paper uses the SGI Altix and a Linux cluster with a more recent processor (Itanium-2) and the latest Elan-4 Quadrics network. Figure 7 shows the performance of SRUMMA and ScaLAPACK/PBLAS *pdgemm* for regularly distributed matrix multiplication. For the comparison, we used the *pdgemm* routine from PBLAS/ScaLAPACK Version 1.7. SUMMA [19] is used in practice in ScaLAPACK/PBLAS [20]. The same *dgemm* (double precision serial matrix multiplication) routines from a vendor optimized math library (*mlib* from Hewlett Packard for IA64) were used in all three parallel algorithms. Optimum block sizes were chosen empirically for all matrix sizes and processor counts. Figure 7 shows that for this cluster configuration, similarly to other platforms [1], SRUMMA delivers competitive performance to PBLAS *pdgemm*.

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 [1]. However, to understand the performance of the matrix multiplication algorithm on the Linux cluster with Quadrics Elan-4 network we performed several tests to measure the role of the underlying communication protocols with respect to the overall performance model. Our algorithm uses nonblocking RMA communication, which in principle offers an excellent potential for overlapping communication with computations. An increasing amount of computation is gradually inserted between the initiating nonblocking get call and the wait completion call. At some point, the sum of the nonblocking call issue overhead and computation would exceed the idle CPU time, and hence the total benchmark running time would increase. This gives us the maximum possible overlap. Experimental results (Figure 8) indeed confirm that the non-blocking get offers almost 99% overlap for medium- and larger-sized messages on Quadrics which makes this operation very well suited for overlapping communication with computations. This validates the performance model described by Equation 3.

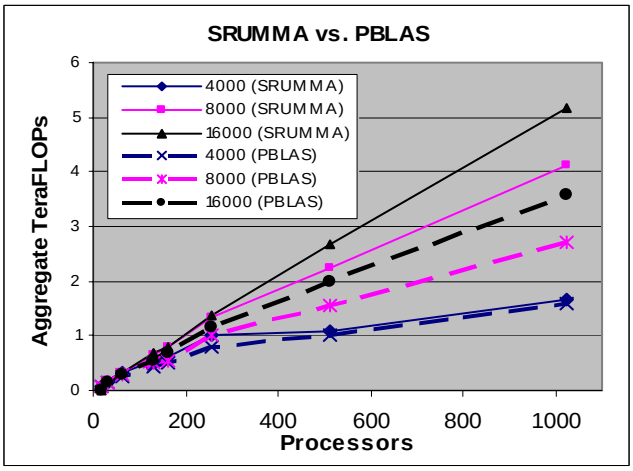


**Figure 7.** Performance of regular distributed matrix multiplication operation on the Linux64 cluster.

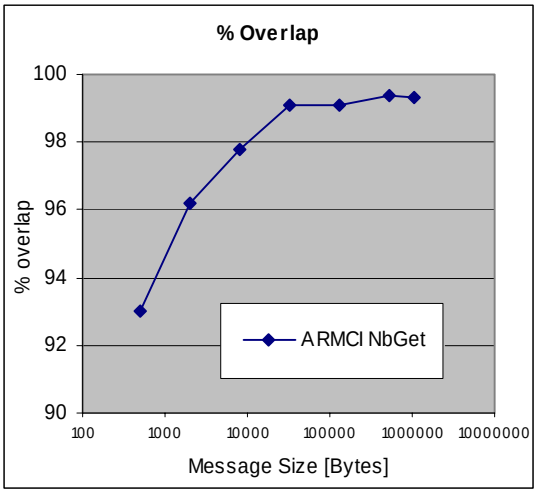


**Figure 8.** Degree of overlap as a function of message size in ARMCI on the Linux64 cluster.

## 4.2 Experimental results for irregular distribution

In many scientific applications, a matrix distribution is irregular because of the underlying physical problem, which involves variable block sizes on individual processors. To demonstrate the effectiveness of our proposed algorithm, we used irregularly distributed matrices in scientific applications, in particular computational chemistry applications and the N-body problem. In these applications, parallel matrix multiplication is one of the most important linear algebra operations and the distribution of the matrix among processors can significantly impact the performance of matrix multiplication algorithm.

We used these irregular matrices (Figure 9) and performed parallel matrix multiplication using the *proposed* distribution-independent algorithm (Section 3) and a *copy-based* algorithm. In the *copy-based* algorithm, we created three temporary regular matrices (i.e. *A*, *B* and *C*) and then redistributed (i.e. copy) the data [40,43] and performed regular SRUMMA matrix multiplication. There is a *redistribution cost* involved in the *copy-based* algorithm as the original matrices, which are irregularly distributed among processors, are redistributed to temporary regular matrices, which then are uniformly distributed among processors.

### *4.2.1 Computational Chemistry Example*

Fock matrices [37,38,39] are used in the distributed data SCF algorithm in massively parallel computational chemistry applications like NWChem [45]. They result in superior scaling for smaller molecules and very large systems. The Fock matrix is distributed by "atom blocks", that is, elements of the Fock F belonging to a given atom are all stored on the same processor to simplify the communication costs for the underlying physical problem [39]. Therefore, the Fock matrix should be stored in a square processor grid fashion because of the distribution nature of the Fock matrix. For example, a Fock matrix of size *m*x*m* can be only be distributed on a processor grid *p*x*q* (where $p = q$). However, this distribution nature of Fock matrix leads to load imbalance, and there is also potential load imbalance when $p \neq q$. A simple Fock matrix for $H_2O$ is shown in Figure 9a. For example, the processors that own Oxygen atom blocks have larger block size when compared to processors owning Hydrogen atom blocks (see Figure 9a). For our experimental purposes we used Fock matrices of various sizes: 1798x1798, 3880x3880 and 7096x7096 [38,39]. These are the matrices for human GTPase-activating protein (Figure 9b) [39].

Experiments were conducted on the Quadrics Linux cluster and the SGI Altix. Our experimental results (Figure 10) indicate that, the *proposed* algorithm, which is memory efficient because temporary matrices are not needed, is competitive with the *copy-based* algorithm. This is because of the redistribution cost (also shown in the Figure 10) in the *copy-based* algorithm, as the redistribution overhead (or communication overhead) is directly proportional to the number of processors and matrix size. The *proposed* algorithm uses O ($B^2$) memory (for temporary matrices) and copy-based uses O ($N^2/P$) + O ($B^2$) memory, where B is the block size. This is excluding memory required to store A, B and C matrices in *proposed* as well as *copy-based* algorithms. For example, let us consider $N$=3880 and $P$=16 processors, in the *proposed* algorithm, each process consumes only 3 MB extra memory (2*3*256*256*8, i.e. 2 temporary buffers, 3 matrices, 8 bytes for storing a *double* and block size as 256) and *copy-based* algorithm consumes approximately 25 MB extra memory (3*(3880*3880/16)*8 + 2*3*(256*256)*8).

Linux cluster results indicate that for small matrices, the *copy-based* algorithm seems to perform better because of the relatively low redistribution cost. For large matrices and large processor counts, the *proposed* algorithm outperformed the *copy-based* algorithm by at least 10% in most of the cases. This is due to the following reasons: (i) the *proposed* algorithm was able to overlap 90% of the communication with computation, (ii) higher redistribution cost in *copy-based* algorithm for larger problem sizes and processor counts. For example, on 240 processors, the *proposed* algorithm performed 20% and 16% better than the *copy-based* algorithm for matrix sizes 3880 and 7096 respectively. The percentage improvement is lower for matrix size 7096 when compared to 3880, because computation cost predominates communication cost for larger matrices. However, on a perfectly square processor grid (e.g., 16x16 grid; 256 processors), the *copy-based* algorithm performs well because the Fock matrix of human GTPase-activating protein is almost uniformly distributed among processors and the redistribution cost is low.

On the other hand, the SGI Altix is a shared memory NUMA system, in which shared memory data can be cached. Therefore, neither of the parallel matrix multiplication algorithms requires explicit communication. As the memory copy cost is relatively less when compared to the explicit communication (as in the Linux cluster), the proposed algorithm takes about the same time as the *copy-based* algorithm and shows only 5% improvement in some cases. However, on perfectly square processor counts (say, 64), the *copy-based* algorithm performs well because the matrix is

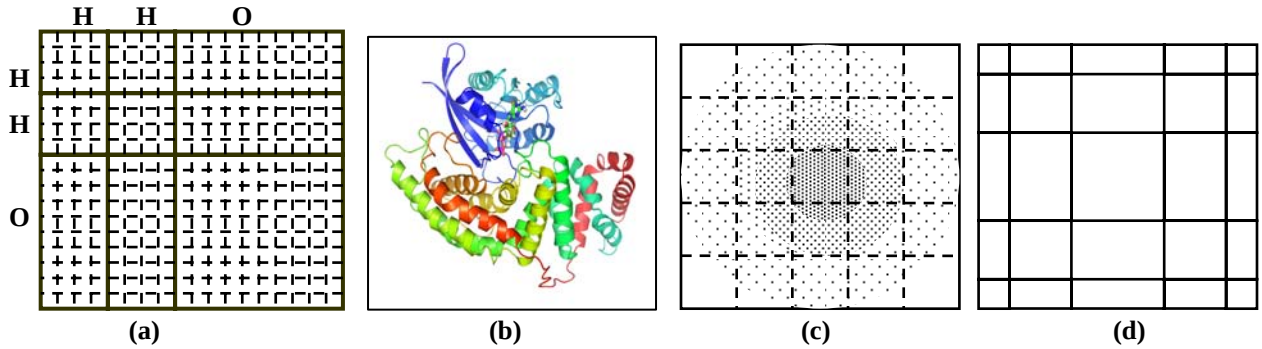


**Figure 9.** (a) Fock (Density) matrix for $H_2O$distributed on a 3x3 process grid [39]. Basis functions of each atom (assuming each Hydrogen Oxygen atom has 4 and 10 basis functions) should reside on the same processor because of the nature of the underlying physics problem. (b) GTPase-activating Protein (c,d) Gaussian distribution of the N-body particles in the computational domain and spatial decomposition for a 5x5 process grid

almost uniformly distributed among processors and the redistribution cost is low (local memory access is faster than remote memory copy in a NUMA system).

Although the *proposed* algorithm takes about the same time as the *copy-based* SRUMMA (only 10% improvement in most cases), the *proposed* algorithm is memory-efficient and the algorithm of choice for applications where it is not practical to redistribute matrices. Moreover, for a fixed problem size, when the processor count increases, the impact of the redistribution cost increases thus affecting scalability. However, the *proposed* algorithm scales well as long as there is enough computation to overlap communication. For example, a 10% communication overlap can result in a 10% improvement in communication cost.

### *4.2.2 N-body Problems:*

The *N*-body problem [44] is the problem of finding the motions of *N* bodies (particles), given the initial positions, masses, and velocities, using classical mechanics (i.e., Newton's law of gravity and Newton's laws of motion). The particles are distributed in a non-uniform way in the computational space. For our experimental purposes, we considered the standard Gaussian (or normal) distribution of particles with three different classes as shown in Figure 14. Class A is the most irregular case considered with more particles located toward the center of the computational domain. The spatial decomposition of the particles in a processor

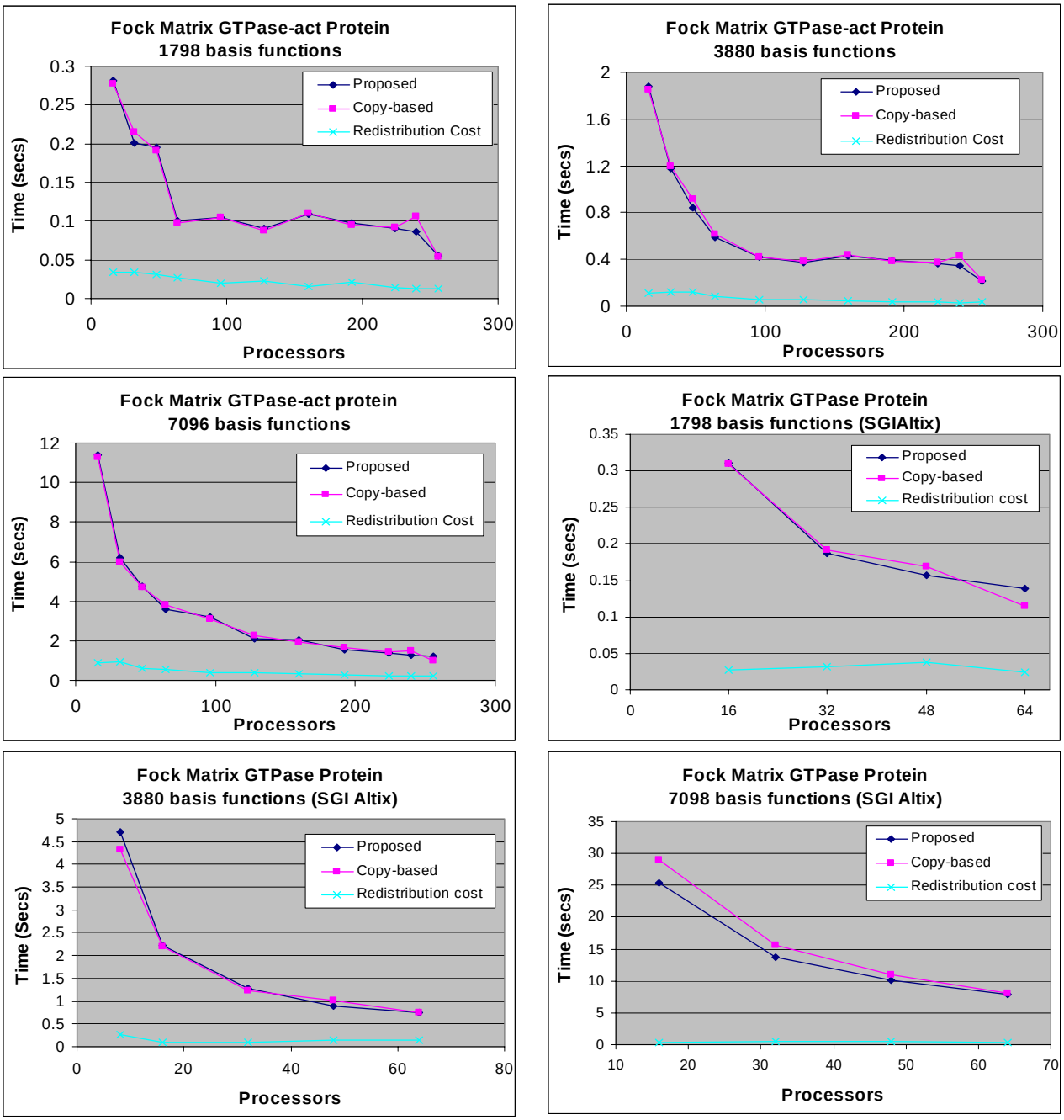


**Figure 10**. Performance of three versions of matrix multiplication for various matrix sizes/basis functions (1798, 3880 and 7096) corresponding to Fock matrix of Human GTPase-activating protein on the Linux cluster and SGI Altix.

grid results in an irregular distribution of particles among processors as shown in Figure 9c and 9d; therefore, the underlying matrices in this application are irregular. For example, the processors, which own a block from the center of the computational domain, have a much larger block size when compared to processors that own corner blocks (Figure 9d).

Experiments were conducted on the two platforms for all three classes of Gaussian distribution (Figure 14). We present results for square matrices of size 2048, 4096, and 8192. Although the performance results for the *proposed* algorithm are uniformly good for all the cases we studied, because of space limitations, we only include results for 1) all three problem sizes for class B, and 2) a medium problem size (4096) for classes A and C. Figures 11 and 12 illustrate that the *proposed* algorithm is competitive as well. The N-body results are similar to the Fock matrix example because the problems sizes and computational structure are almost same in both examples. For all the three classes of Gaussian distribution, the *proposed* algorithm consistently performed well on both platforms. The *Copy-based* algorithm performs somewhat better for perfectly square processor grid (e.g, 225, 256 processors), because of the relatively low redistribution cost. Linux cluster results illustrate that for larger processor counts and smaller problem sizes (e.g 2048), the *proposed* algorithm

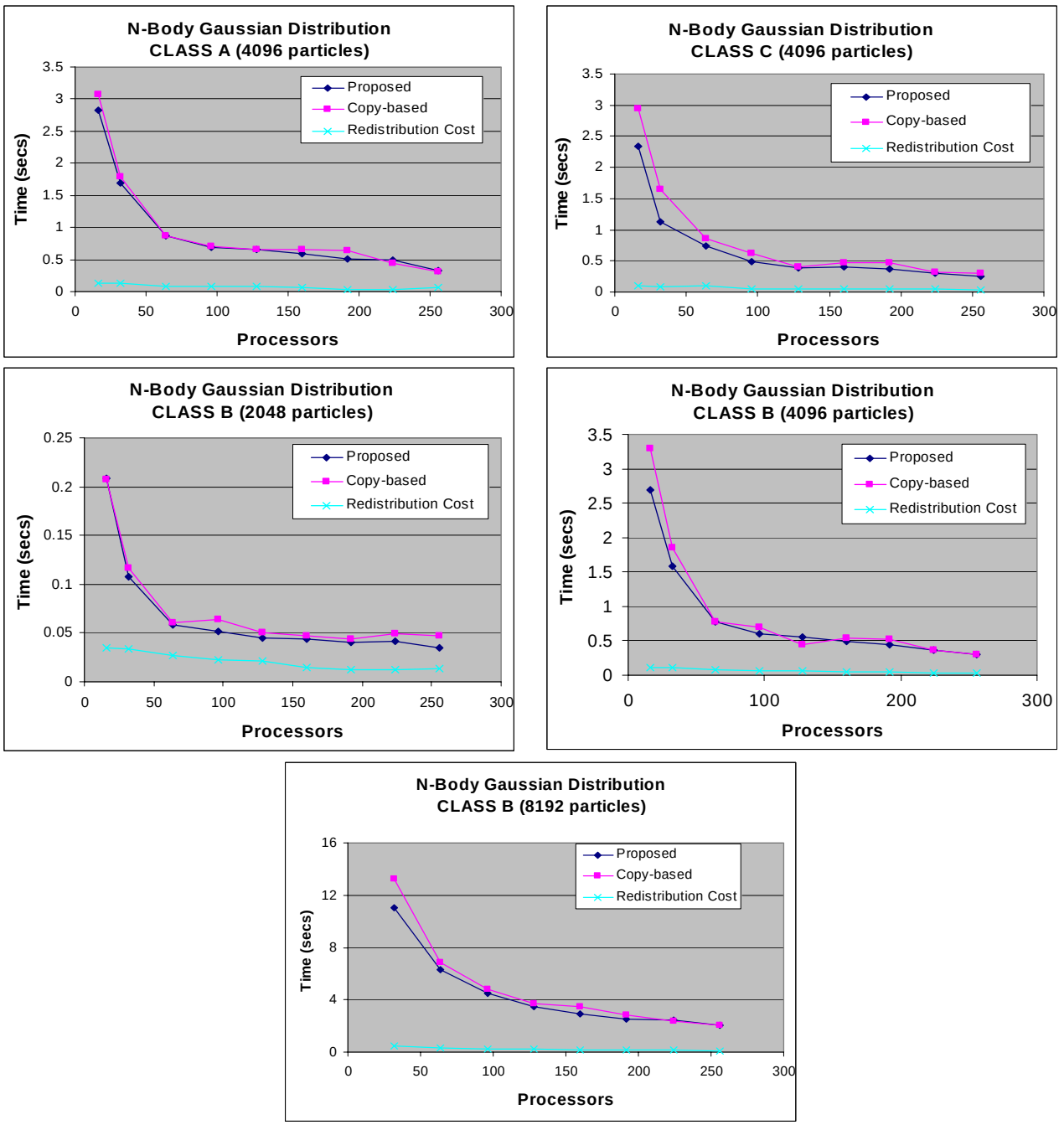


**Figure 11.** Performance of three versions of matrix multiplication for irregularly distributed matrices (Gaussian distribution for N-body problems) on the Linux cluster.

performed better than the *copy-based* because of the following reasons: (i) the *proposed* algorithm was able to overlap communication with computation, for example, a 10% communication overlap can result in a 10% improvement in communication cost; (ii) higher redistribution cost in *copy-based* algorithm when compared to the computation cost.

### *4.2.3 Degree of Irregularity*

The quantification on how the degree of irregularity impacts the performance of both algorithms for matrix size 4096 on Linux cluster (proposed and copy-based) is shown in Figure 13. Due to space limitations, we are only showing Linux cluster performance numbers. The SGI Altix results are similar to those in Figure 13. Class A is the most irregular case as shown in Figure 14 and matrices of class C are well-balanced compared to class A matrices. In Figure 14, degree of irregularity (peak of distribution function) is 2, 4 and 8 for class C, B, and A respectively (i.e. class A is 2 and 4 times irregular when compared to class B and C respectively). In spite of a huge variation in degree of irregularity, it is interesting to see that the performance of the proposed algorithm varies only 10-20% (see, Figure 13) for various classes of matrix distribution. The proposed algorithm scales well for various processor counts and classes of matrix distributions. For class C style matrices (low degree of irregularity), the copy-based algorithm performs slightly better than the proposed algorithm as the redistribution cost is less. For class A style matrices, the

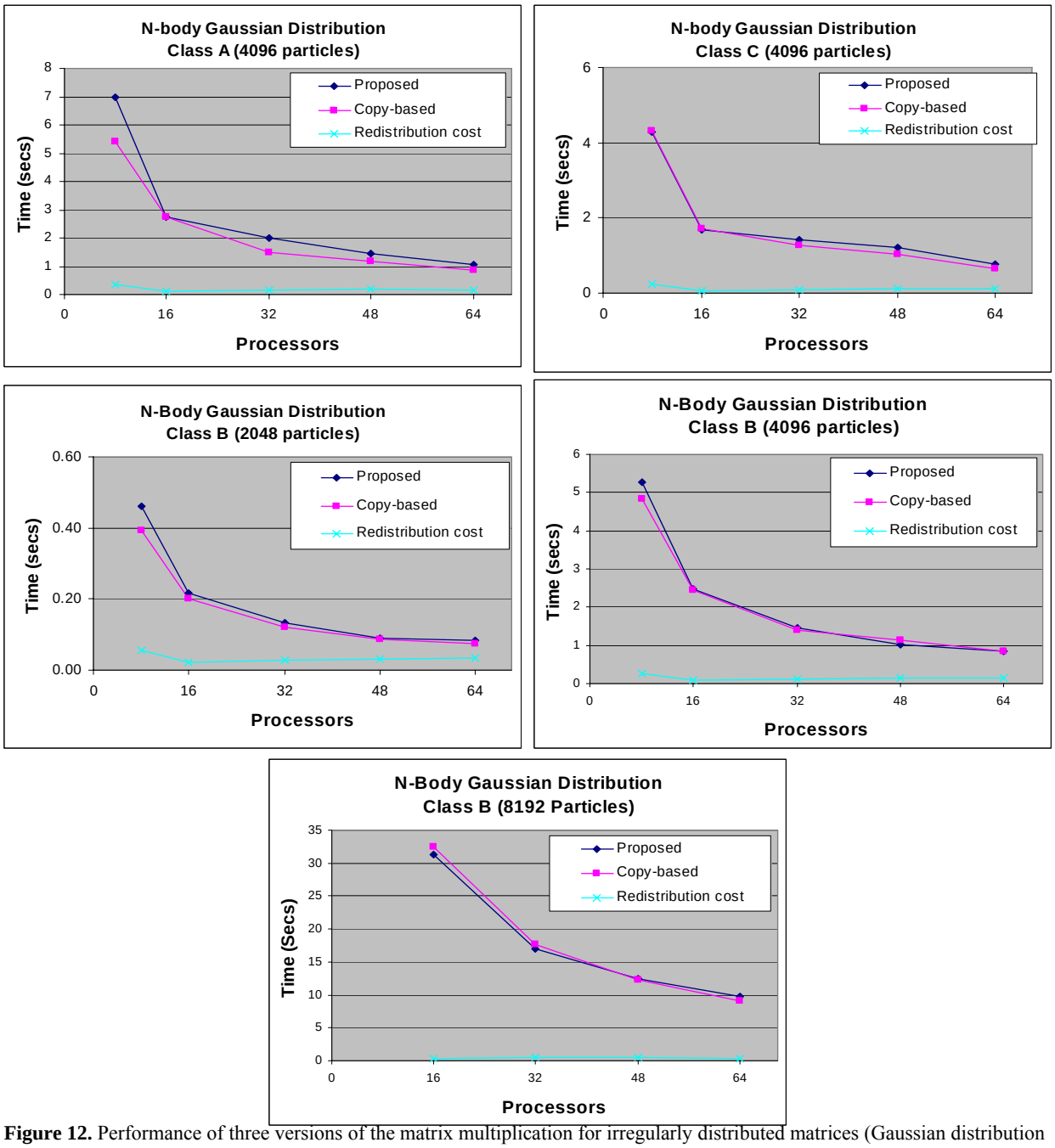


**Figure 12.** Performance of three versions of the matrix multiplication for irregularly distributed matrices (Gaussian distribution for N-body problems) on the SGI Altix.

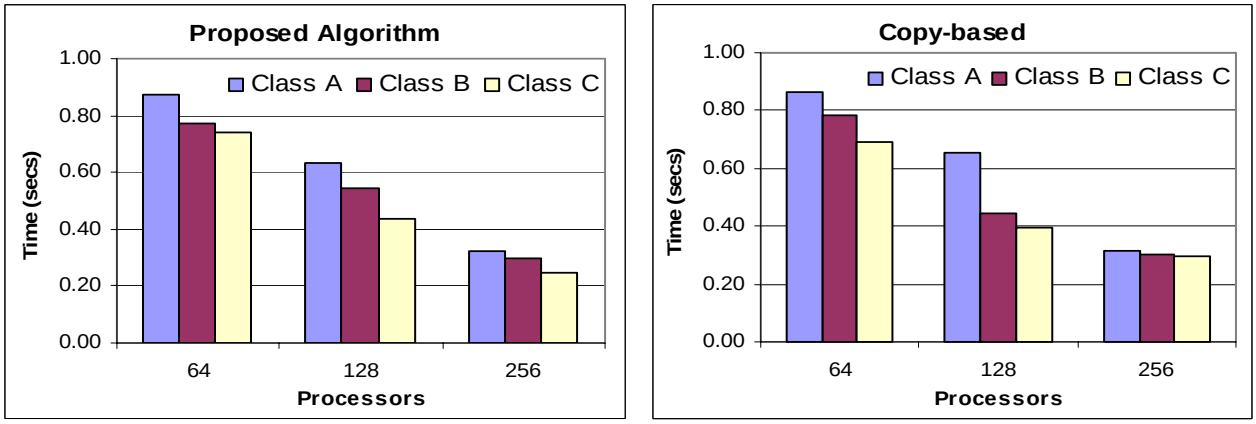


**Figure 13.** Degree of irregularity for matrix size 4096. Performance of the proposed and copy-based algorithm for various degrees of irregularity of matrix distribution on the Linux cluster.

redistribution cost is high and therefore, the proposed algorithm is competitive to the copy-based algorithm. For larger number of processors (say, 256 in our case), the performance of copy-based algorithm degrades because the redistribution cost predominates the computation cost.

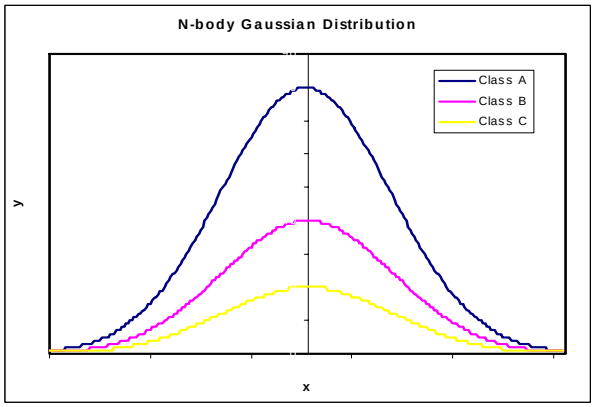


**Figure 14.** Three classes of N-body distribution used.

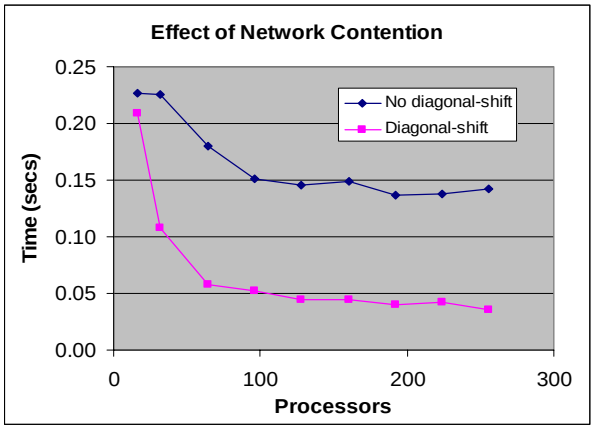


**Figure 15.** Reducing network contention on Linux cluster.

### *4.2.4 Avoidance of communication contention*

To demonstrate the effectiveness of diagonal-shift algorithm (Figure 4) for reducing communication contention we enabled and disabled the diagonal-shift algorithm in the *proposed* algorithm for an N-body class B problem (2048 particles). Figure 15 illustrates that the diagonal-shift algorithm in the *proposed* algorithm improves performance by effectively reducing network contention on clusters.

## 5. SUMMARY AND CONCLUSIONS

This paper describes a new dense matrix multiplication algorithm that can efficiently deal with irregularly distributed matrices while minimizing memory consumption. Generalized SRUMMA exploits shared memory and nonblocking RMA protocols on clusters and shared memory systems. The distribution independent algorithm delivers performance as good as the algorithm for regularly distributed matrices combined with the necessary data redistributions. However, because it avoids redistributions and the creation of temporary matrices, it is preferable in practice especially for problems with large matrices or when system memory constrains, on architectures such as the IBM Blue Gene/L, make creating temporary arrays and redistribution impractical.

## 6. REFERENCES

[1] M. Krishnan, J. Nieplocha, "SRUMMA: A Matrix Multiplication Algorithm Suitable for Clusters and Scalable Shared Memory Systems", *Proc. IPDPS'04,* 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, 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, 1989.

[7] A. Gupta and V. Kumar, "Scalability of Parallel Algorithms for Matrix Multiplication", ICPP'93, 1993.

[8] C. Lin and L.Snyder, “A matrix product algorithm and its comparative performance on hypercubes”, *SHPCC ‘92*.
[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*, 1994.
[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, 1997.
[14] E. Dekel, D. Nassimi, S. Sahni, Parallel matrix and graph algorithms, *SIAM J. Computing*, vol. 10, 1981.
[15] S. Ranka, S. Sahni. *Hypercube Algorithms for Image Processing and Pattern Recognition*. Springer-Verlag, 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), 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, M. Zubair, A high performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication, *IBM J. 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, 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 *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”, *Intl. Conf. Cluster Computing* ‘02.
[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, 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, *ACM Intl. 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’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, *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*, 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] M. Krishnan and J. Nieplocha, "Optimizing Parallel Multiplication Operation for Rectangular and Transposed Matrices," In Proceedings of 10th IEEE ICPADS. 2004.
[37] T.H. Dunning, Jr. J. Chem. Phys. 90, 1007 (1989).
[38] Y. Alexeev, M. Valiev, D. A. Dixon, T. L. Windus, "Ab initio study of catalytic GTP hydrolysis", J. of ACS, ‘04.
[39] I. Foster et al. "Toward High-Performance Computational Chemistry: I. Scalable Fock Matrix Construction Algorithms", Journal of computational chemistry, vol. 17, No. 1, 109-123, 1996.
[40] C. Hsu, Y. Chung, and C. Dow, Efficient Methods for Multi-Dimensional Array Redistribution, Journal of Supercomputing, 17, 23–46, 2000.
[41] C. Edwards, P. Geng, A. Patra, and R.Van De Geign, "Parallel Matrix Distributions: Have we been doing it all wrong?", TR-95-39, Department of Computer Sciences, University of Texas, Oct. 1995.
[42] Hyuk-Jae Lee, J.A.B. Fortes, Toward data distribution independent parallel matrix multiplication, IPDPS, 1995.
[43] S. D. Kaushik, C.-H. Huangl, R. W. Johnson2, and P. Sadayappan, "An Approach to Communication-Efficient Data Redistribution", Proc. Supercomputing’94, pp: 364-373,1994.
[44] Banicescu, Ioana and R. Lu, Experiences with Fractiling in N-Body Simulations, HPC Symposium, 1998.
[45] Kendall *et al*, “High Performance Computational Chemistry: an Overview of NWChem a Distributed Parallel Application", Computer Phys. Comm., 2000, 128, 260-283.