A Poly-Algorithm for Parallel Dense Matrix Multiplication on Two-Dimensional Process Grid Topologies*

Jin Li
Anthony Skjellum†

Department of Computer Science & NSF Engineering Research Center
Mississippi State University
Mississippi State, MS 39762

Robert D. Falgout

Center for Computational Sciences and Engineering, L-316
Lawrence Livermore National Laboratory
Livermore, CA 94550

September 1, 1993
Revised December 30, 1995

Abstract

In this paper, we present several new and generalized parallel dense matrix multiplication algorithms of the form $C = \alpha AB + \beta C$ on two-dimensional process grid topologies. These algorithms can deal with rectangular matrices distributed on rectangular grids. We classify these algorithms coherently into three categories according to the communication primitives used and thus we offer a taxonomy for this family of related algorithms.

All these algorithms are represented in the *data distribution independent* approach and thus do not require a specific data distribution for correctness. The *algorithmic compatibility* condition result shown here ensures the correctness of the matrix multiplication. We define and extend the *data distribution functions* and introduce *permutation compatibility* and *algorithmic compatibility*. We also discuss a permutation compatible data distribution (*modified virtual 2D data distribution*).

We conclude that no single algorithm always achieves the best performance on different matrix and grid shapes. A practical approach to resolve this dilemma is to use poly-algorithms. We analyze the characteristics of each of these matrix multiplication algorithms and provide initial heuristics for using the poly-algorithm.

All these matrix multiplication algorithms have been tested on the IBM SP2 system. The experimental results are presented in order to demonstrate their relative performance characteristics, motivating the combined value of the taxonomy and new algorithms introduced here.

---

*Revision and extension of our original paper entitled "The Multicomputer Toolbox Approach to Concurrent BLAS" submitted September 1, 1993.

†Address correspondence to: Mississippi State University, Engineering Research Center, PO Box 6176, Mississippi State, MS 39762. Tel: 601-325-8435. Email: tony@cs.msstate.edu.

A Poly-Algorithm for Parallel Dense Matrix Multiplication on Two-Dimensional Process Grid Topologies

1 Introduction

1.1 Parallel Dense Matrix Multiplication Algorithms

Many parallel algorithms have been previously proposed for multiplying dense matrices (such as [1, 5, 9, 13]). These algorithms are usually defined to operate on square matrices uniformly distributed on square process grids. Also, the distribution of the data is often driven by the algorithms. For instance, Mathur-Johnsson's algorithm [19] extends *Cannon*'s algorithm (the matrix C-stationary version) to deal with arbitrary matrices and grids, but the underlying data distribution is fixed (block linear data distribution). *PUMMA* [6] extends *Fox*'s algorithm to non-square grids but fixes on the block scattered data distribution. Furthermore, *PUMMA*'s authors only discuss their algorithm's behavior for square matrices. By way of contrast, *BiMMeR*'s *BMR* algorithm [16] uses a different approach to extend *Fox*'s algorithm to non-square grids. The data layout is flexible and the *virtual 2D torus wrap* data distribution is recommended [16]. At present, however, *BiMMeR*'s *BMR* algorithm only deals with square matrices. Furthermore, the algorithm proposed in [2] uses a different approach for parallel matrix multiplication. This algorithm only uses the *broadcast* communication primitive and tries to overlap the communication and the computation; *SUMMA* [25] evidently adopts the same idea with a slight difference in implementation.

We present several algorithms that apply to the general case of rectangular matrix multiplication on rectangular process grids of the form $C = \alpha AB + \beta C$ (shown in part in [11]). We classify the matrix multiplication algorithms into three categories according to the communication primitives used. Previously, some confusion on classes of algorithms has evidently persisted [12]. The first category is *Cannon*'s approach [5] or the *2D systolic* approach. We discuss three versions of *Cannon*'s algorithm: the C-stationary version, the B-stationary version, and the A-stationary version. The second category is *Fox*'s approach [13] or the *Broadcast-Multiply-Roll* approach, which has two orthogonal versions (the row and column versions). We present three algorithms: MM_3, MM_4, and MM_5. All these algorithms use different approaches to extend *Fox*'s algorithm to deal with non-square grids [12]. Specifically, MM_5 is a generalized version of *BiMMeR*'s *BMR* algorithm [16] and can deal with non-square matrices; MM_3 and MM_4 are completely new algorithms. The third category is the *Broadcast-Broadcast* approach and a new algorithm, *BB*, is detailed. All these algorithms use the *data distribution independent* [26, 27] approach so that the data distributions of matrices are flexible. However, the data distributions of matrices A, B, and C must satisfy the *algorithmic compatibility* [11, 12] requirement to ensure the correctness of these algorithms; this work assumes that arbitrary remapping is either space or time demanding and seeks to avoid such redistributions without further justification.

1.2 Design Methodology

The design of parallel dense matrix multiplication algorithms is based on the *Multicomputer Toolbox* approach [21]. Two key ideas underlying scalable programming are logical process grids and *data distribution independence* [26, 27]. A logical process grid, denoted here by $\mathcal{G}_{P\times Q}$, is a collection of processes (*e.g.*, one process per node of a mesh or hypercube), logically assigned a shape $P\times Q$. The processes are named (p, q) on this logical grid, where p (resp, q) ranges from $0 \ldots P-1$ (resp, $0 \ldots Q-1$). Such logical grids can be readily mapped to physical node topologies [20].

Hence, we are here striving to compare algorithms that minimize time to solution while avoiding extensive data remapping. Within this regime, an important concept of building parallel library is the role of poly-algorithms. Poly-algorithm[1] refers to the use of two or more algorithms to solve the same problem with a high level decision-making process determining which of a set of algorithms performs best in a given situation. As we perceive, no single parallel dense matrix multiplication algorithm always achieves the best performance when dealing with arbitrary grid shapes and arbitrary matrix shapes. One key issue of using a poly-algorithm is to determine under what kind of situations an algorithm achieves the best performance. Another key issue is to develop a common interface for these related algorithms. Here we concentrate on demonstrating the benefit of a poly-algorithmic matrix multiplication library by classifying algorithms and their performance. This motivates the need for investment in software design and performance modelling.

Finally, *data distribution independence* is an important design consideration for building scalable parallel libraries, so that applications do not have to redistribute data before and/or after calling a library. An explicit data redistribution is always expensive either in terms of speed (time) or additional memory requirements [4]. Furthermore, there is no convenient technology for efficient redistribution at present; such algorithms merit their own poly-algorithmic study, which we offer separately [18]. The data distribution functions provide a means to implement the *data distribution independence*. The *algorithmic compatibility* requirement ensures the parallel dense matrix multiplication algorithms to work correctly without prior remapping of one or more operands.

1.3 Implementation

The implementation of the parallel dense matrix multiplication algorithms described here exploits the object-based programming approach discussed in [20, 21] and is based on the MPI message passing interface [14]. For instance, The *LA_Mapping* object encapsulates the data distribution functions to support the *data distribution independence*. The *LA_Grid* object uses MPI topology constructors to create a 2-D process grid. The *LA_Distrib_2d* composes the *LA_Mapping* and the *LA_Grid* to provide a 2-D data distribution for a matrix. The *LA_Dmatrix* object encapsulates a dense matrix with a 2-D data distribution on a 2-D process grid. The *LA_Dmatrices_Mul* object encapsulates three matrices A, B, and C and an algorithm that performs the multiplication of the form $C = \alpha AB + \beta C$. This high level object provides the interface for our parallel dense matrix multiplication. Figure 1 shows the hierarchical structure of the objects just mentioned.

The communication primitives, such as *broadcast*, *shift*, *align*, and *slide*, are implemented using MPI point-to-point communication functions and collective communication functions. The BLAS

[1] We would like to thank Professor John Rice for introducing and discussing the poly-algorithm concept with us.

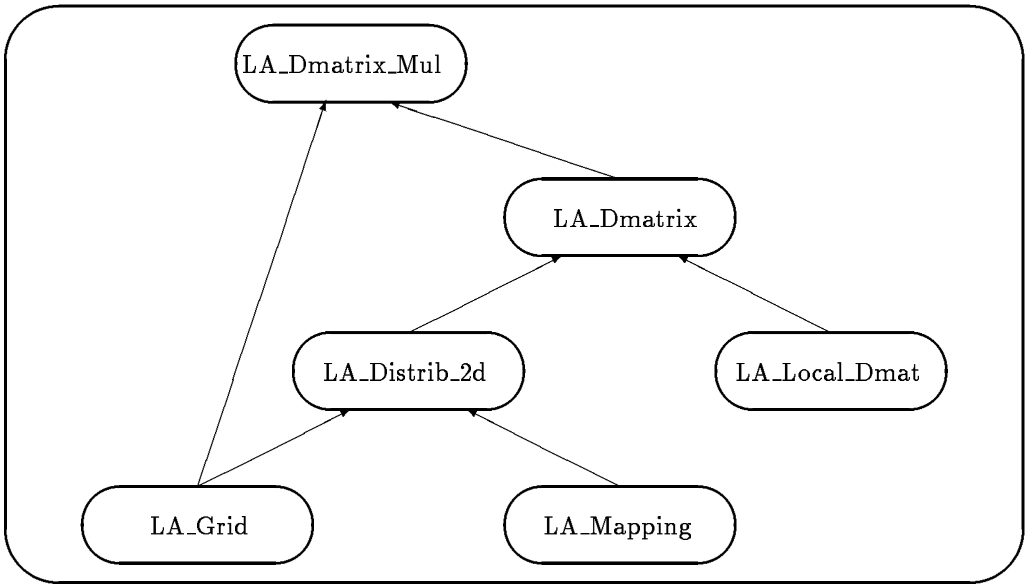

Figure 1: The hierarchical structure of the objects.

subroutine *DGEMM* is used as the local multiplication engine. Further consideration of software design merits separate discussion, which we offer separately in [22, 23] and elsewhere.

2 Permutation Compatible Data Distributions

In this section, we present the definition of *data distribution*. We also discuss two instances of data distributions (*i.e.*, (block) linear data distributions and (block) scattered data distributions with different strides) and give a general formula that can be used to derive these data distributions by selecting different parameters. Then we define the notion of *permutation compatibility*, which is important for subsequently defining the *algorithmic compatibility* requirement of our matrix multiplication algorithms. We discuss a permutation compatible data distribution (*i.e.*, *virtual 2D torus wrap data distribution* [15, 16]), which is used to distribute matrices on two-dimensional process grid topologies. Finally, we introduce a *modified virtual 2D data distribution* that can solve the potential load imbalance problem induced by the *virtual 2D torus wrap data distribution*.

2.1 Data Distribution Functions

Definition 1 (Data-Distribution Functions) *A data-distribution function μ maps three integers $\mu(I, P, M) \mapsto (p, i)$ where I, $0 \leq I < M$, is the global name of a coefficient, P is the number of processes among which all coefficients are to be partitioned, and M is the total number of coefficients. The pair (p, i) represents the process p $(0 \leq p < P)$ and local (process-p) name i of the coefficient. The inverse distribution function $\mu^{-1}(p, i, P, M) \mapsto I$ transforms the local name i back to the global coefficient name I. The cardinality function $\mu^{\sharp}(p, P, M) \mapsto m$ determines the number* *of coefficients m that belong to process p when M coefficients are distributed among P processes using data-distribution function μ.*

The "formal" definition of a data distribution may be found in either [20] or [21].

The (block) linear data distribution and the (block) scattered data distribution are two instances of data-distribution. We perform the mapping in two steps. First, we partition the global coefficients into R blocks. Each block may or may not be the same size, depending on whether R is a divisor of M, but we try to partition M into blocks as the same size as possible. The blocks may vary at most one element if we select the block size as $\lceil \frac{M}{R} \rceil$ or $\lfloor \frac{M}{R} \rfloor$. We choose to gather these blocks with slightly larger size in low numbered processes without loss of generality. The size of tth block is as follows:

$$b_t = \begin{cases} \lceil \frac{M}{R} \rceil & : \quad 0 \leq t < (M \bmod R) \\ \lfloor \frac{M}{R} \rfloor & : \quad (M \bmod R) \leq t < R. \end{cases} \tag{1}$$

Secondly, we assign each block to the processes consecutively. Successive blocks are assigned to different processes with a fixed stride s. The tth block is assigned to process p according to the following expression[2]:

$$p \;=\; \Big(s \times \big(t \bmod P\big) + \big(\frac{t}{\frac{P}{gcd(P,s)}} \bmod gcd(P,s)\big)\Big) \bmod P \tag{2}$$

where s $(1 \leq s < P)$ is the stride of panel space and gcd denotes the greatest common divisor function.

By selecting a different block size b or block number R and stride s in Equation 1 and 2, we can obtain various data distributions. Some of them are useful. For example, if we select $R = P$ and $s = 1$, we achieve a linear data distribution. Further, if we select $R = M$, we achieve a scattered data distribution with various strides s. If we select other value of R and s, we can achieve block scattered data distributions with different block sizes and strides.

2.2 Data Distributions on Two-Dimensional Topologies

Now, we consider the distribution of a matrix on a two-dimensional process grid. Consider an $M \times N$ matrix A, that is, M rows and N columns, mapped onto a process grid $\mathcal{G}_{P \times Q}$. Figure 2 shows a 9×8 matrix example. A $P \times Q$ process grid $\mathcal{G}_{P \times Q}$, that is, P rows and Q columns, is numbered as (p, q) where $p = 0 \ldots P-1, q = 0 \ldots Q-1$. Figure 3 shows a 4×3 process grid and how each process is numbered. We perform the data mapping on both dimensions. In the row dimension, we denote the mapping function as

$$\mu(I, P, M) \mapsto (p, i)$$

with different block number R_μ and stride s_μ. While in the column dimension, we denote the mapping function as

$$\nu(J, Q, N) \mapsto (q, j)$$

[2]This expression is a modified version of the formula defined by *BiMMeR* [16].

with different block number R_ν and stride s_ν. Figure 4 shows the distribution of a matrix A on a grid $\mathcal{G}_{\mathcal{P}\times\mathcal{Q}}$ with a scattered data distribution on the row dimension and a linear data distribution on the column dimension.

2.3 Permutation compatibility for matrix multiplication

Permutation compatibility is the following property between two data distributions:

Definition 2 (Permutation Compatibility) *Two distributions $\mu(\bullet, P, M)$ and $\nu(\bullet, Q, N)$ are permutation compatible if and only if $M = N$ and*

$$\begin{aligned} \Lambda_\mu^{-1}(\mu(I,P,M),P,M) &= \Lambda_\nu^{-1}(\nu(I,Q,M),Q,M) \\ \forall I &= 0,\ldots,M-1 \end{aligned}$$

where the associated linear distributions Λ_μ and Λ_ν have matching cardinalities:

$$\begin{aligned} \Lambda_\mu^\sharp(p,P,M) &= \mu^\sharp(p,P,M) \\ \Lambda_\nu^\sharp(p,P,M) &= \nu^\sharp(p,P,M) \\ &\forall p = 0,\ldots,P-1. \end{aligned}$$

An example of two permutation compatible distributions is given in Figure 5.

Now, consider the matrix multiplication algorithm $C = \alpha AB + \beta C$. Matrix A is an $M_A \times N_A$ matrix with a row distribution $\mu_A(\bullet, P, M_A)$ and a column distribution $\nu_A(\bullet, Q, N_A)$, matrix B is an $M_B \times N_B$ matrix with a row distribution $\mu_B(\bullet, P, M_B)$ and a column distribution $\nu_B(\bullet, Q, N_B)$, where $N_A = M_B$ and the data distributions ν_A and μ_B are permutation compatible. We now discuss the permutation compatibility problem between ν_A and μ_B with different data distributions on different grid shapes. Obviously, if ν_A and μ_B are permutation compatible, they must have the same block size r and the same stride s. Hence, we discuss distribution compatibility of two distributions with the same block size and the same stride:

1. if both the ν_A and the μ_B are linear data distributions, then they are obviously permutation compatible on an arbitrary process grid $\mathcal{G}_{\mathcal{P}\times\mathcal{Q}}$.

2. both the ν_A and the μ_B are scattered data distributions. If the process grid is square, ν_A and μ_B are permutation compatible. If the process grid is not square, ν_A and μ_B are not permutation compatible. Note that the problem is caused by the different values of P and Q in Equation 2.

3. both the ν_A and μ_B are block scattered data distributions. The same compatibility condition results as in the scattered data distribution situation.

The permutation imcompatibility of the (block) scattered distribution on non-square grid situations causes problem for the matrix multiplication of $C = \alpha AB + \beta C$, because the algorithm requires the permutation compatibility of ν_A and μ_B. This can be resolved grossly by data remapping, but our point is not to resort to remapping within the classification scheme; anyway, we wish to avoid temporaries and excess communication as mentioned in [4].

$$
\begin{pmatrix}
a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} \\
a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} \\
a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} \\
a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} \\
a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} \\
a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} \\
a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} \\
a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} \\
a_{8,0} & a_{8,1} & a_{8,2} & a_{8,3} & a_{8,4} & a_{8,5} & a_{8,6} & a_{8,7}
\end{pmatrix}
$$

Figure 2: A 9×8 matrix A.

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)
(3,0)	(3,1)	(3,2)

Figure 3: Coordinate numbering for a 4×3 process grid $\mathcal{G}_{\mathcal{P}\times\mathcal{Q}}$.

$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{4,0}$ $a_{4,1}$ $a_{4,2}$ $a_{8,0}$ $a_{8,1}$ $a_{8,2}$	$a_{0,3}$ $a_{0,4}$ $a_{0,5}$ $a_{4,3}$ $a_{4,4}$ $a_{4,5}$ $a_{8,3}$ $a_{8,4}$ $a_{8,5}$	$a_{0,6}$ $a_{0,7}$ $a_{4,6}$ $a_{4,7}$ $a_{8,6}$ $a_{8,7}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{5,0}$ $a_{5,1}$ $a_{5,2}$	$a_{1,3}$ $a_{1,4}$ $a_{1,5}$ $a_{5,3}$ $a_{5,4}$ $a_{5,5}$	$a_{1,6}$ $a_{1,7}$ $a_{5,6}$ $a_{5,7}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{6,0}$ $a_{6,1}$ $a_{6,2}$	$a_{2,3}$ $a_{2,4}$ $a_{2,5}$ $a_{6,3}$ $a_{6,4}$ $a_{6,5}$	$a_{2,6}$ $a_{2,7}$ $a_{6,6}$ $a_{6,7}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{7,0}$ $a_{7,1}$ $a_{7,2}$	$a_{3,3}$ $a_{3,4}$ $a_{3,5}$ $a_{7,3}$ $a_{7,4}$ $a_{7,5}$	$a_{3,6}$ $a_{3,7}$ $a_{7,6}$ $a_{7,7}$

Figure 4: The data distributions of matrix A on a process grid $\mathcal{G}_{\mathcal{P}\times\mathcal{Q}}$ with a linear data distribution on the column dimension and a scattered data distribution on the row dimension.

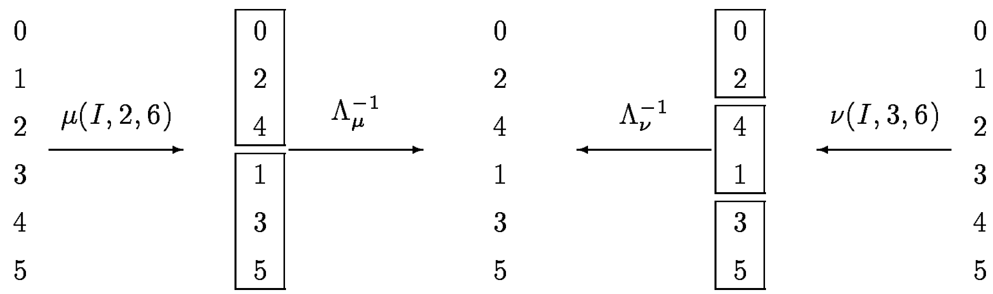

Figure 5: Example of permutation compatibility: two different data-distributions, $\mu(\bullet, 2, 6)$ and $\nu(\bullet, 3, 6)$, are permutation compatible.

2.4 The virtual 2-dimensional grid

For a non-square grid $\mathcal{G}_{\mathcal{P}\times\mathcal{Q}}$, we can view it as a $\alpha \times \alpha$ virtual grid [15, 16], where α is the least common multiple of P and Q. Then we distribute matrices on this $\alpha \times \alpha$ virtual grid [16]. We modify the Equation 2 as follows:

$$p \;=\; \left(s \times \left(t \bmod \alpha\right) + \left(\frac{t}{\frac{\alpha}{gcd(\alpha,s)}} \bmod gcd(\alpha, s)\right)\right) \bmod \alpha \tag{3}$$

If we choose the same block size and stride, ν_A and μ_B prove to be permutation compatible. Note that if we choose $s = \frac{\alpha}{P}$, we obtain the same scattered data distribution as that obtained by Equation 2, but with local storage differences (process-local permutations).

We achieve permutation compatibility of ν_A and μ_B by using the virtual grid, but we introduce another problem by doing so. Notice that when N_A (M_B) is not evenly divided by α, the remainder blocks are assigned to different virtual processes. However, these virtual processes may be in the same process, which causes more blocks on some processes than on other processes. Figure 6 illustrates the column distribution $\nu_A(\bullet, 6, 27)$ and the row distribution $\mu_B(\bullet, 4, 27)$. For the $\nu_A(\bullet, 6, 27)$ situation, process 0 has one more element than that on average. For the $\mu_B(\bullet, 4, 27)$ situation, process 0 has two more elements than that on average.

We can modify our distribution strategy to solve this problem and this is an important realization. After we use Equation 3, we actually obtain a new order for the global coefficients. We can then partition this new order of global coefficients into P parts so that each has $\lceil \frac{M}{P} \rceil$ or $\lfloor \frac{M}{P} \rfloor$ coefficients and consecutively assign them to the processes. We call this the *modified virtual 2D data distribution*. Figure 7 shows the data distributions using the modified strategy.

3 Taxonomy of Parallel Dense Matrix Multiplication Algorithms

In this section, we first define *algorithmic compatibility* for the matrix multiplication algorithms of the form $C = \alpha AB + \beta C$. We introduce the notation used in our pseudocode and define the communication primitives used by the matrix multiplication algorithms. Then, we present the

Process	Local Column Index					
0	0	12	24	1	13	25
1	2	14	26	3	15	
2	4	16	5	17		
3	6	18	7	19		
4	8	20	9	21		
5	10	22	11	23		

Process	Local Row Index								
0	0	12	24	1	13	25	2	14	26
1	3	15	4	16	5	17			
2	6	18	7	19	8	20			
3	9	21	10	22	11	23			

Figure 6: The column distribution $\nu_A(\bullet, 6, 27)$ and the row distribution $\mu_B(\bullet, 4, 27)$ using the *virtual 2D torus wrap data distribution*.

Process	Local Column Index				
0	0	12	24	1	13
1	25	2	14	26	3
2	15	4	16	5	17
3	6	18	7	19	
4	8	20	9	21	
5	10	22	11	23	

Process	Local Column Index						
0	0	12	24	1	13	25	2
1	14	26	3	15	4	16	5
2	17	6	18	7	19	8	20
3	9	21	10	22	11	23	

Figure 7: The column distribution $\nu_A(\bullet, 6, 27)$ and the row distribution $\mu_B(\bullet, 4, 27)$ using the *modified virtual 2D data distribution*.

matrix multiplication algorithms. We classify the matrix multiplication algorithms into three categories according to the communication primitives used. The first category is *Cannon*'s approach [5], which shifts two of the matrices A, B, and C. We discuss three versions of *Cannon*'s algorithm: C-stationary version, B-stationary version, and A-stationary version; usually only the C-stationary version is considered. The second category is *Fox*'s approach [13], which broadcasts one matrix and shifts the other; there are two orthogonal versions (the row version and the column version). In the row (resp, column) version, matrix A (resp, B) is broadcast and matrix B (resp, A) is shifted. Furthermore, we discuss two new algorithms: MM_3 and MM_4 and a generalized algorithm: MM_5. All these algorithms use different approaches to extend *Fox*'s algorithm to deal with a non-square grid; the MM_5 is a generalized version of *BiMMeR*'s *BMR* algorithm [16] (*BMR* only deals with square matrices). The third category is *Broadcast-Broadcast* approach, which broadcasts both matrix A and matrix B; for this category, we detail a new algorithm *BB*. Table 1 summarizes the taxonomy for our parallel dense matrix multiplication algorithms and other known algorithms that can be fit in the taxonomy.

3.1 Algorithmic Compatibility

Algorithmic compatibility is the property of *data distributions* of matrix A, matrix B, and matrix C that ensures an algorithm in our taxonomy can be used correctly. The algorithms of matrix

Table 1: Summary of the taxonomy for parallel dense matrix multiplication algorithms

Cannon's approach	*Fox*'s approach	*Broadcast-Broadcast* approach
Cannon C version	MM_3 *row* version	*BB* algorithm
Cannon A version	MM_3 *column* version	Algorithm presented in [2]
Cannon B version	MM_4 *row* version	*SUMMA* [25]
Mathur-Johnsson's algorithm [19]	MM_4 *column* version	Special case of 3D algorithm [3]
	MM_5 *row* version	
	MM_5 *column* version	
	PUMMA [6]	
	BiMMeR's *BMR* [16]	

$$\left(\begin{array}{ccc|ccc} a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ \hline a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ \hline a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \end{array}\right) \times \left(\begin{array}{cc|cc} b_{0,*} & b_{0,*} & b_{0,*} & b_{0,*} \\ b_{2,*} & b_{2,*} & b_{2,*} & b_{2,*} \\ \hline b_{4,*} & b_{4,*} & b_{4,*} & b_{4,*} \\ b_{1,*} & b_{1,*} & b_{1,*} & b_{1,*} \\ \hline b_{3,*} & b_{3,*} & b_{3,*} & b_{3,*} \\ b_{5,*} & b_{5,*} & b_{5,*} & b_{5,*} \end{array}\right)$$

Figure 8: Example of algorithm compatibility: multiplication of matrices $A_{8\times6}$ and $B_{6\times4}$, distributed compatibly on a 3×2 process grid. Coefficient subscripts (*i.e.*, $a_{I,J}$) are the global (I, J) indices.

multiplication of the form $C = \alpha AB + \beta C$ require the following *algorithmic compatibility*:

Definition 3 (Algorithmic Compatibility) *Matrices A, B, and C are compatible for the matrix multiplication algorithms if and only if the column distribution of A is permutation compatible with the row distribution of B, and the row and column distributions of C are identical to the row distribution of A and the column distribution of B, respectively.*

Figure 8 illustrates the definition in action for a 3×2 logical process grid and matrices A and B with shapes 8×6 and 6×4, respectively. If we view the multiplication, AB, as $(\mathcal{P}_A A \mathcal{Q}_A)(\mathcal{P}_B B \mathcal{Q}_B)$, where $\mathcal{P}_A$ (resp, $\mathcal{P}_B$) is a global permutation view of A's (resp, B's) row distribution, and $\mathcal{Q}_A$ (resp, $\mathcal{Q}_B$) is a global permutation view of A's (resp, B's) column distribution, then the compatibility definition above requires that $\mathcal{Q}_A = \mathcal{P}_B^T$. This means that

$$(\mathcal{P}_A A \mathcal{Q}_A)(\mathcal{P}_B B \mathcal{Q}_B) = \mathcal{P}_A (AB) \mathcal{Q}_B = \mathcal{P}_A C \mathcal{Q}_B,$$

so that C has A's row distribution and B's column distribution.

A principal goal of the design of the matrix multiplication algorithms discussed here is *data distribution independence*. These algorithms do not require a specific data distribution, such as a linear data distribution, a block data distribution, or a scattered data distribution. When the data distributions of matrices A, B, and C satisfy the *algorithmic compatibility* requirement, these algorithms can be used correctly. Otherwise, prior redistribution of data is needed to ensure the correct result, as we have mentioned above.

3.2 Notation of Pseudocode and Definition of Communication Primitives

Now, we introduce the notation and pseudocode used to represent the matrix multiplication algorithms. The notation $A_{p,q}$ is to represent the submatrix of A that is stored in process (p,q). Since all of the algorithms that follow are presented as being logically local to process (p,q), the objects within them do not need subscripts. Hence, the notation A within our pseudocode will mean $A_{p,q}$. Also within the pseudocode, the notation $A_{(i_1:i_1,j_1:j_2)}$ represents the submatrix of $A_{p,q}$ consisting of rows i_1 through i_2 and columns j_1 through j_2 (the indices here are local indices). The same notation applies for matrix B and matrix C.

The quantities m_A and n_A represent the local dimension of matrix A on process (p,q). m_A is determined by the row *cardinality function* $\mu^\sharp$ and n_A is determined by the column *cardinality function* $\nu^\sharp$ respectively. Similarly, m_B and n_B represent the local dimension of matrix B, and m_C and n_C represent the local dimension of matrix C.

The communication primitives used in the matrix multiplication algorithms are *broadcast*, *slide*, *align*, and *shift*. The *align* primitive actually consists of two primitives (*i.e.*, *slide* and *stride*). It first performs *slide* if necessary and then *stride* if necessary. All these communication primitives actually perform along one dimension of the grid (either the row dimension or the column dimension). Each includes a row version and a column version. Early versions of these are shown in [11, 12].

The collective communication primitive *broadcast* is conventional: one process sends identical data to all other processes in the same communication group. The *row broadcast*$_{(k)}$, where $0 \leq k < Q$, performs along the row dimension of grid $\mathcal{G}_{P\times Q}$. The definition of *row broadcast*$_{(k)}$ follows next.

Definition 4 (*row broadcast*$_{(k)}$) *Let D represent data distributed on a grid $\mathcal{G}_{P\times Q}$, and let $D_{p,q}$ be that part of D contained in process (p,q). let $D'_{p,q}$ be a temporary buffer of the same size as $D_{p,k}$ in process (p,q). Then, the collective primitive row broadcast$_{(k)}$ is defined by:*

$$\textit{row broadcast}_{(k)}\ D'_{p,q}\ :\ D_{p,k}\ \mapsto\ D'_{p,q};$$

for all $p : 0 \leq p < P$ and $q : 0 \leq q < Q$. k represents the column index of the process that is the root of the broadcast.

The *col broadcast*$_{(k)}$, where $0 \leq k < P$, is orthogonal to the *row broadcast*$_{(k)}$. It performs along the column dimension of the grid $\mathcal{G}_{P\times Q}$. k represents the row index of the process that is the root of the *broadcast*.

The collective communication primitive *stride*$_{(\pm i,\pm j)}$, where $0 \leq i < P$ and $0 \leq j < Q$, is defined as follows:

Definition 5 (*stride*$_{(\pm i,\pm j)}$) *Let D represent data distributed on a grid $\mathcal{G}_{P\times Q}$, and let $D_{p,q}$ be that* *part of D contained in process (p,q). Then, the collective primitive $stride_{(\pm i,\pm j)}$ is defined by:*

$$\begin{aligned} stride_{(\pm i,\pm j)}\ D_{p,q} &: D_{p,q} \mapsto D_{p',q'}; \\ p' &= (p \pm i) \bmod P, \\ q' &= (q \pm j) \bmod Q. \end{aligned}$$

The notation $stride_{(\pm i,\pm j)}$ D means $stride_{(\pm i,\pm j)}$ $D_{p,q}$ for all p, q. The sign of i and j denotes the direction of data movement. The "+" indicates the direction is from left to right along the column dimension and from up to down along the row dimension. The "-" indicates from right to left and from down to up respectively.

Thus, the row *stride* with stride j is denoted as $stride_{(0,\pm j)}$ and the column *stride* with stride i is denoted as $stride_{(\pm i,0)}$.

The collective communication primitive $shift_{(\pm i,\pm j)}$, where $i = 0$ *or* $i = 1$ and $j = 0$ *or* $j = 1$, is a special case of $stride_{(\pm i,\pm j)}$. So the row $shift_{(0,\pm 1)}$ is defined as $stride_{(0,\pm 1)}$ and the column $shift_{(\pm 1,0)}$ is defined as $stride_{(\pm 1,0)}$.

The collective communication primitive $slide_{(\pm i,\pm j)}$ is a collective data redistribution primitive, where $0 \le i < \mu^\sharp$ and $0 \le j < \nu^\sharp$.

Definition 6 ($slide_{(\pm i,\pm j)}$) *Let $D = \{d_{I,J}\}$ represent data distributed on a grid $\mathcal{G}_{P\times Q}$ with row and column distributions: $\mu_r(I,P,M)$ and $\mu_c(J,Q,N)$. Then, the collective primitive $slide_{(i,j)}$ D redistributes $d_{I,J}$ with row and column distributions:*

$$\begin{aligned} &\Lambda_{\mu_r}((I_{\mu_r} \pm i) \bmod M, P, M), \\ &\Lambda_{\mu_c}((J_{\mu_c} \pm j) \bmod N, Q, N), \end{aligned}$$

where Λ_{μ_r} and Λ_{μ_c} are associated linear distribution functions with matching cardinalities, and

$$\begin{aligned} I_{\mu_r} &= \Lambda^{-1}_{\mu_r}(\mu_r(I,P,M),P,M), \\ J_{\mu_c} &= \Lambda^{-1}_{\mu_c}(\mu_c(J,Q,N),Q,N). \end{aligned}$$

The sign of i and j denotes the direction of data movement. The "+" indicates the direction is from left to right along the column dimension and from up to down along the row dimension. The "-" indicates from right to left and from down to up respectively.

Thus, the row *slide* is denoted as $slide_{(0,\pm j)}$ and the column *slide* is denoted as $slide_{(\pm i,0)}$.

The collective communication primitive *align* consists of two steps of communication primitives: first *slide* and then *stride*. The *row* $align_{(\pm j,k)}$ is defined as:

$$row\ align_{(\pm j,k)} = \begin{cases} slide_{(0,-j)} + stride_{(0,-k)} \\ slide_{(0,j)} + stride_{(0,k)}. \end{cases}$$

The *col* $align_{(\pm i,k)}$ is defined as:

$$col\ align_{(\pm i,k)} = \begin{cases} slide_{(-i,0)} + stride_{(-k,0)} \\ slide_{(i,0)} + stride_{(k,0)}. \end{cases}$$

3.3 *Cannon*'s approach (the *2D-Systolic* approach)

The principle of *Cannon*'s approach is discussed in several works [5, 16, 17, 19]. Most of the discussion simply deals with a square grid and square matrices of the C-stationary version. Figure 9 gives the process-(p, q) pseudocode for *Cannon*'s algorithm of the matrix C-stationary version and Figure 10 gives an example for dealing with a non-square grid and non-square matrices.

The *align* primitives align the inner indices of local matrices A and B on each process so that the systolic multiplication can apply. There are two choices to implement the *align* primitive, when the global coefficients are not evenly distributed on each process. One approach maintains the property that the local matrices of A (resp, B) in the same process columns (resp, rows) have the same number of column (resp, row) coefficients. Referring to Figure 10b, the result of a row align of matrix A will be that $A_{(2:4,5:6)}$ is in process-(1,1) and $A_{(2:4,0:2)}$ is in process-(1,2). Another approach tries to move the local matrices as a whole. The result of this *align* approach is showed in Figure 10b. Notice that the local matrix sizes of matrix A on process-(1,1) and process-(1,2) are changed and the local matrix sizes of matrix B on process-(0,2) and process-(1,2) are changed accordingly. Although the above property is destroyed, the inner indices of local matrices A and B are still aligned. We prefer the later approach, because it has less overhead for preparing local data for alignment.

Two other variants of *Cannon*'s approach are the versions with either matrix B or matrix A held stationary, which we have not seen discussed elsewhere, except in passing [19]. The communication patterns of the two variants are a little bit different than that of the matrix C-stationary version. In the matrix B-stationary version, the matrix A will be shifted left and down in both the row and column dimensions and the matrix C will be shifted down in the column dimension. Figure 11 gives the process-(p, q) pseudocode for *Cannon*'s algorithm of the matrix B-stationary version and Figure 12 gives an example. The matrix A-stationary version is orthogonal in its data motion to the matrix B-stationary version, so is not shown explicitly. In the matrix A-stationary version, the matrix B will be shifted right and up in both the row and column dimensions and the matrix C will be shifted right in the row dimension.

The disadvantage of *Cannon*'s approach is the initial align and final realign of matrix A and matrix B. If we were chaining multiplication operations together or we were reusing matrix A and/or matrix B for several multiplication operations, we might want to suppress the final realign and thus the initial align for consequent multiplications. Such optimization will be considered in future research.

algorithm *CANNON_C [matrix C stationary]*

$C := \beta C;$	//scale matrix C
$(kA, jA) \leftarrow \nu_A(\mu_B^{-1}(p, 0, P, M_B), Q, N_A);$	//get col index of A matching row 0 of B
row align$_{(-jA,kA)}$ $A;$	//row align matrix A
$(kB, iB) \leftarrow \mu_B(\nu_A^{-1}(q, 0, Q, N_A), P, M_B);$	//get row index of B matching col 0 of A
col align$_{(-iB,kB)}$ $B;$	//column align matrix B
$a := b := 0;$	
$r_A := Q; \quad r_B := P;$	//initialization
while $((r_A > 0)$ **or** $(r_B > 0))$	
$r := \min\{(n_A - a), (m_B - b)\};$	//calculate the maximum # of inner indices
$C := C + \alpha A_{(:,a:a+r)} B_{(b:b+r,:)};$	//update local matrix C
$a := a + r; \quad b := b + r;$	
if $((a = n_A)$ **and** $(r_A > 0))$ **then**	//matrix A has used all of the data
row shift$_{(0,-1)}$ $A;$	//row shift matrix A
$r_A := r_A - 1; \quad a := 0;$	
end if	
if $((b = m_B)$ **and** $(r_B > 0))$ **then**	//matrix B has used all of the data
col shift$_{(-1,0)}$ $B;$	//column shift matrix B
$r_B := r_B - 1; \quad b := 0;$	
end if	
end while	
row align$_{(jA,kA)}$ $A;$	//restore initial data distribution of A
col align$_{(iB,kB)}$ $B;$	//restore initial data distribution of B

end algorithm

Figure 9: Pseudocode for *Cannon*'s algorithm of the matrix C-stationary version.

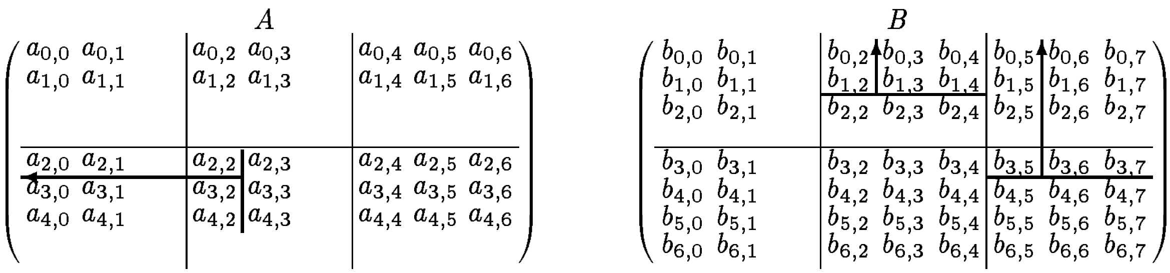

Figure 10a (*Cannon C* example): Initial data distributions of matrix A and matrix B. Matrix A will undergo *row align*$_{(0,0)}$ and *row align*$_{(-1,1)}$ on 0 and 1 rows respectively. Matrix B will undergo *col align*$_{(0,0)}$, *col align*$_{(-2,0)}$, and *col align*$_{(-1,1)}$ on 0, 1, and 2 columns respectively.

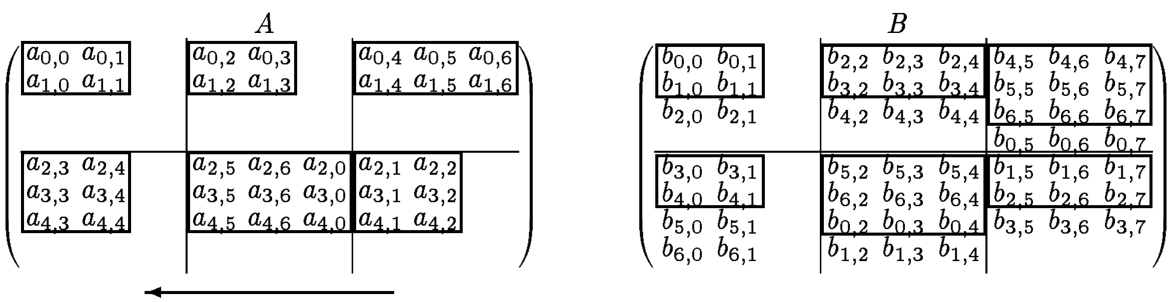

Figure 10b (*Cannon C* example): Data distributions of matrix A and matrix B after alignments. Each process computes as much of the matrix multiplication as can be done locally (data used in the multiplication are marked with solid boxes). Matrix A has used all of the data and will undergo *row shift*$_{(0,-1)}$.

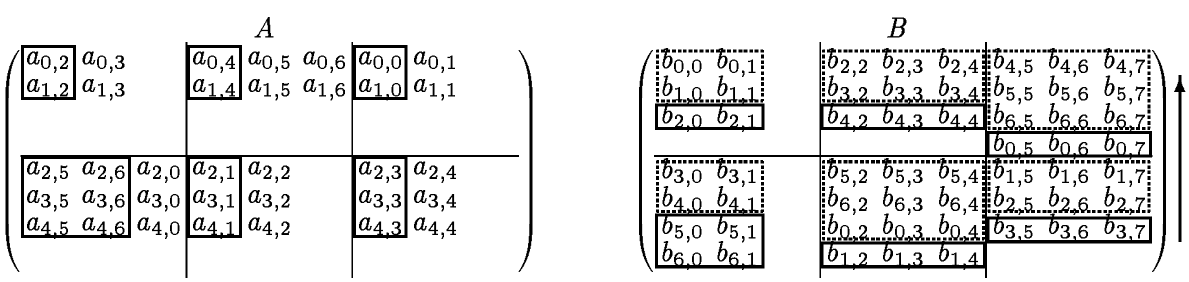

Figure 10c (*Cannon C* example): After matrix A undergoes *row shift*$_{(0,-1)}$, each process does a local matrix multiplication (data of matrix B previously used are marked with dashed boxes). Matrix B has used all of the data and will undergo *col shift*$_{(-1,0)}$.

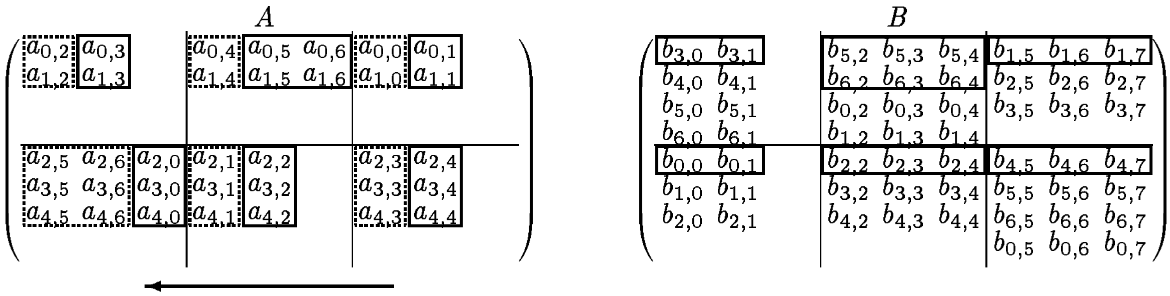

Figure 10d (*Cannon C* example): After matrix B undergoes *col shift*$_{(-1,0)}$, each process computes as much of the matrix multiplication as can be done locally. Matrix A has used all of the data and will undergo *row shift*$_{(0,-1)}$ again.

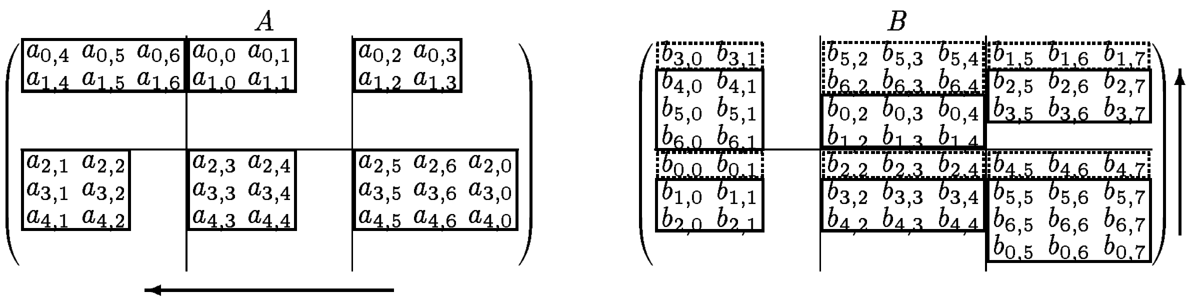

Figure 10e (*Cannon C* example): After local matrix multiplication, both matrices have used all of the data. Matrix A will undergo *row shift*$_{(0,-1)}$ and matrix B will undergo *col shift*$_{(-1,0)}$.

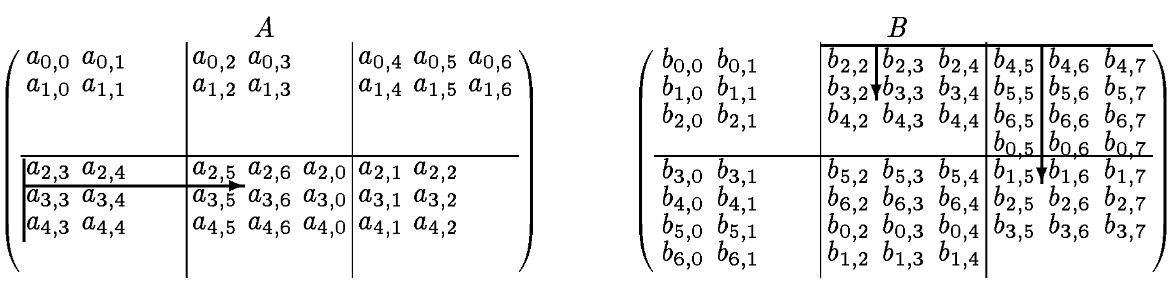

Figure 10f (*Cannon C* example): Matrix A has been shifted Q(here $Q = 3$) times and matrix B has been shifted P(here $P = 2$) times, the matrix multiplication finished. Matrix A undergoes *row align*$_{(1,1)}$ on row 1 and matrix B undergoes *col align*$_{(2,0)}$ and *col align*$_{(1,1)}$ on column 1 and 2 to restore the originial distribution.

algorithm *CANNON_B [matrix B stationary]*
    $C := \beta C;$ //scale matrix C
    $(kA, jA) \leftarrow \nu_A(\mu_B^{-1}(p, 0, P, M_B), Q, N_A);$ //get col index of A matching row 0 of B
    row align$_{(-jA,kA)}$ $A;$ //row align matrix A
    $(kB, iB) \leftarrow \mu_B(\nu_A^{-1}(q, 0, Q, N_A), P, M_B);$ //get row index of B matching col 0 of A
    col align$_{(-iB,kB)}$ $B;$ //column align matrix B
    $a := b := 0;$
    $r_A := Q; \quad r_B := P;$ //initialization
    while $((r_A > 0)$ **or** $(r_B > 0))$
        $r := \min\{(n_A - a), (m_B - b)\};$ //calculate the maximum # of inner indices
        $C := C + \alpha A_{(:,a:a+r)} B_{(b:b+r,:)};$ //update local matrix C
        $a := a + r; \quad b := b + r;$
        if $((b = m_B)$ **and** $(r_B > 0))$ **then** //matrix B has used all of the data
            col shift$_{(1,0)}$ $A;$ //column shift matrix A
            col shift$_{(1,0)}$ $C;$ //column shift matrix C
            $r_B := r_B - 1; \quad b := 0;$
        end if
        if $((a = n_A)$ **and** $(r_A > 0))$ **then** //matrix A has used all of the data
            row shift$_{(0,-1)}$ $A;$ //row shift matrix A
            $r_A := r_A - 1; \quad a := 0;$
        end if
    end while
    row align$_{(jA,kA)}$ $A;$ //restore initial data distribution of A
    col align$_{(iB,kB)}$ $B;$ //restore initial data distribution of B
end algorithm

Figure 11: Pseudocode for *Cannon*'s algorithm of the matrix B-stationary version.

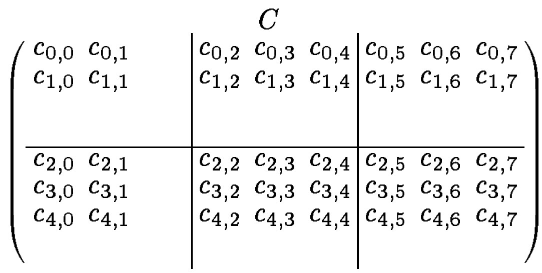

Figure 12a (*Cannon B* example): Initial data distribution of matrix C. The row and column data distributions of matrix C are the same as the row data distribution of matrix A and the column data distribution of matrix B respectively.

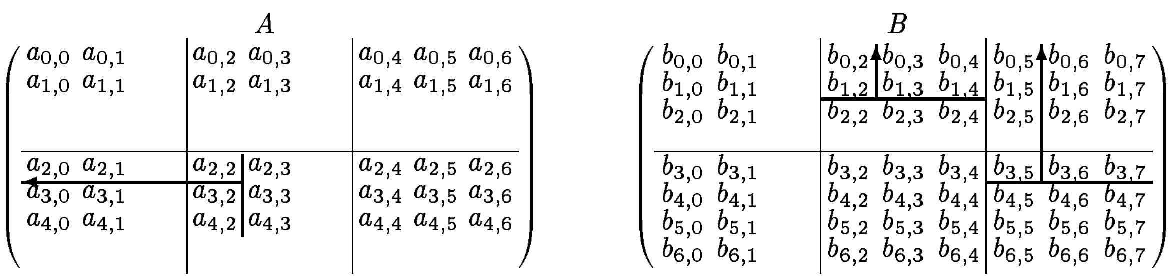

Figure 12b (*Cannon B* example): Initial data distributions of matrix A and matrix B. Matrix A will undergo *row align*$_{(0,0)}$ and *row align*$_{(-1,1)}$ on 0 and 1 rows respectively. Matrix B will undergo *col align*$_{(0,0)}$, *col align*$_{(-2,0)}$, and *col align*$_{(-1,1)}$ on 0, 1, and 2 columns respectively.

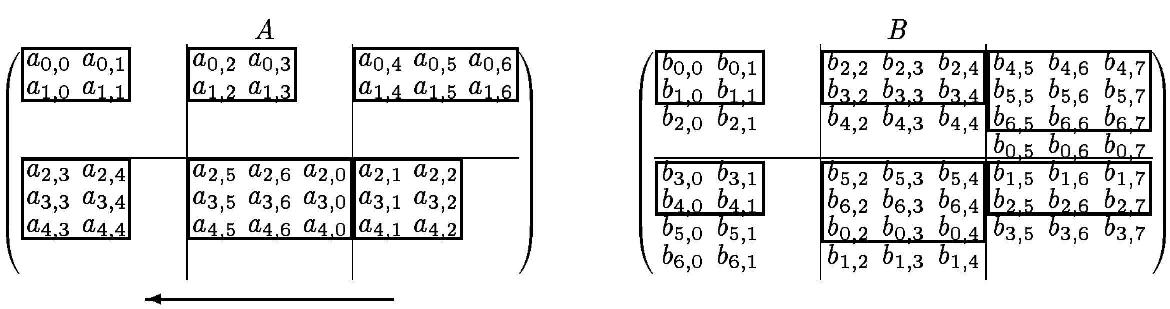

Figure 12c (*Cannon B* example): Data distributions of matrix A and matrix B after alignments. Each process computes as much of the matrix multiplication as can be done locally (data used in the matrix multiplication are marked with solid boxes). Matrix A has used all of the data and will undergo *row shift*$_{(0,-1)}$.

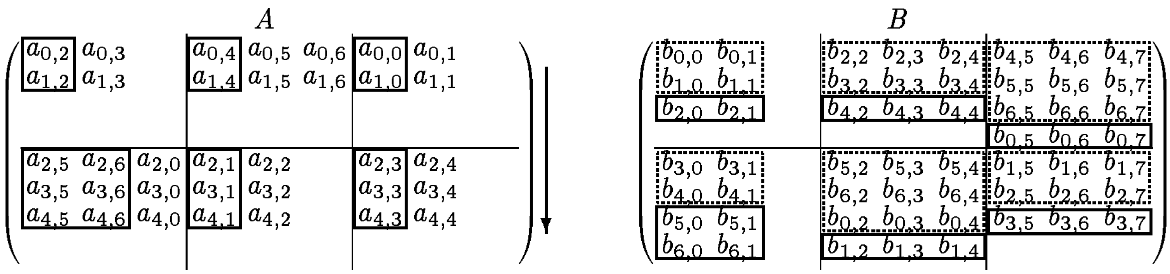

Figure 12d (*Cannon B* example): After matrix A undergoes *row shift*$_{(0,-1)}$, each process does a local matrix multiplication (data of matrix B already used are marked with dashed boxes). Matrix B has used all of the data, thus matrix A and matrix C will undergo *col shift*$_{(1,0)}$.

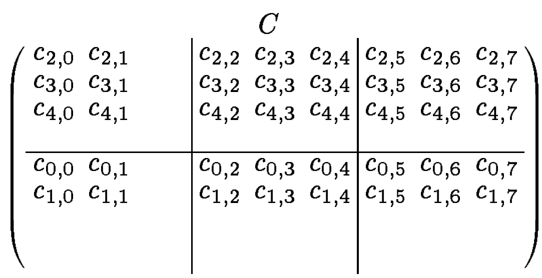

Figure 12e (*Cannon B* example): The data distribution of matrix C after *col shift*$_{(1,0)}$.

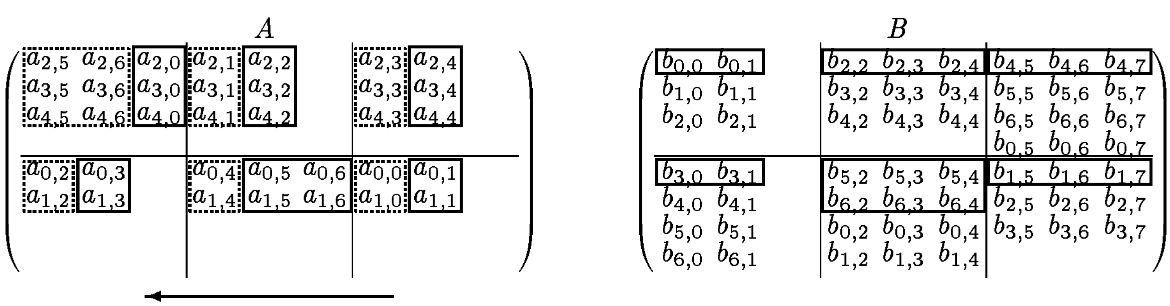

Figure 12f (*Cannon B* example): After matrix A and matrix C undergo *col shift*$_{(1,0)}$, each process does a local matrix multiplication (data of matrix A already used are marked with dashed boxes). Matrix A has used all of the data and will undergo *row shift*$_{(0,-1)}$.

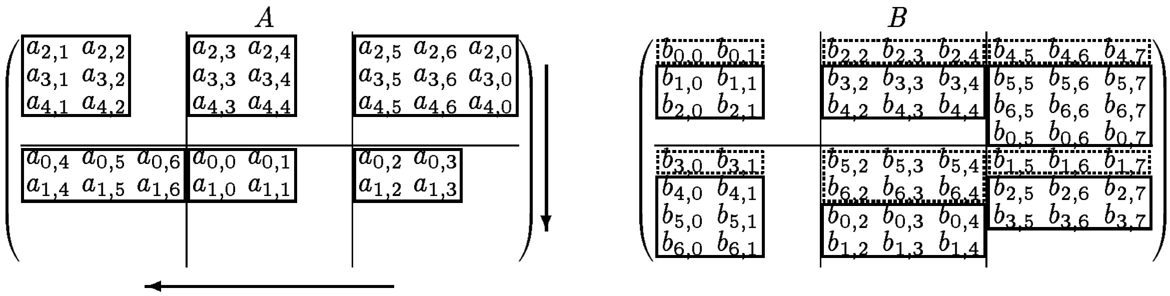

Figure 12g (*Cannon B* example): After matrix A undergoes *row shift*$_{(1,0)}$, each process does a local matrix multiplication. Matrix B has used all of the data, thus matrix A and matrix C will undergo *col shift*$_{(1,0)}$. Also matrix A has used all of the data and will undergo *row shift*$_{(0,-1)}$.

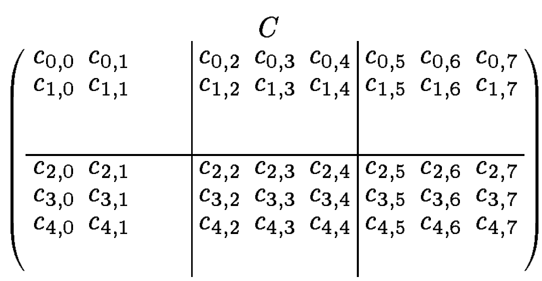

Figure 12h (*Cannon B* example): Matrix C restores the initial data distributions of matrix C after *col shift*$_{(1,0)}$ P(here $P = 2$) times.

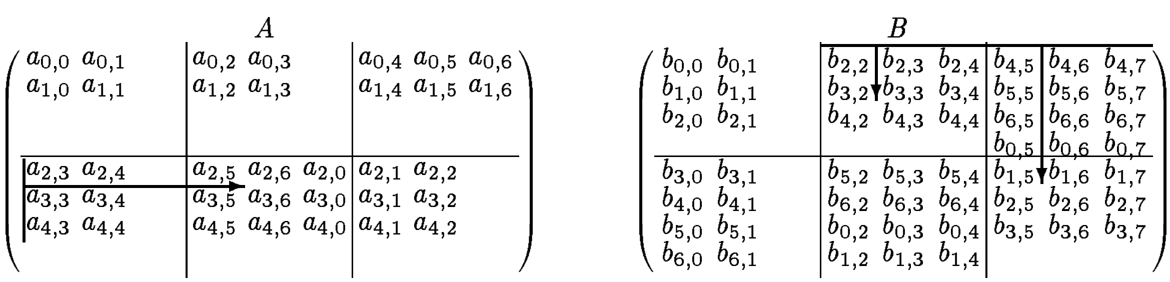

Figure 12i (*Cannon B* example): Matrix A has been row shifted Q(here $Q = 3$) times and column shifted P(here $P = 2$) times, the matrix multiplication finished. Matrix A undergoes *row align*$_{(1,1)}$ on row 1 and matrix B undergos *col align*$_{(2,0)}$ and *col align*$_{(1,1)}$ on column 1 and 2 to restore the originial distribution.

3.4 *Fox*'s approach (the *Broadcast-Multiply-Roll* approach)

Another well-known approach for parallel dense matrix multiplication is *Fox*'s algorithm [13]. *Fox*'s algorithm uses one-to-all broadcasts of blocks of matrix A in the row dimension and shifts of blocks of matrix B along the column dimension. Several different approaches are used to extend *Fox*'s algorithm to deal with non-square grids, such as *PUMMA* algorithm [6] and *BiMMeR*'s *BMR* algorithm [16]. Here we introduce two new algorithms (the MM_3 algorithm and the MM_4 algorithm). We also introduce the MM_5 algorithm that is a generalized version of *BiMMeR*'s *BMR* algorithm and can deal with non-square matrices. Each *MM* algorithm has two orthogonal versions (*i.e.*, row and column versions) but we only discuss the row versions here due to the limitation of space (pseudocodes of the column versions are provided in [12]).

Our first algorithm, MM_3 row version, is given by the process-(p, q) pseudocode in Figure 13 and is illustrated in Figure 14. The MM_3 algorithm is a direct extension of *Fox*'s algorithm to deal with non-square grids. Initially, each block of matrix A that locally matches the block of matrix B is selected for broadcast (Figure 14a). The remaining blocks of matrix A will be broadcast in the last additional broadcast step. Thus, there is one more broadcast step (total of $Q + 1$ broadcasts) of matrix A than that of *Fox*'s algorithm.

To achieve quasi-optimal performance of a parallel algorithm, one must balance the work load of each process in addition to reducing communication costs where possible. In order to balance both computation and communication in the above algorithm, it is evident that we must have that $A_{p,q}$ is $m_A \times n_A$ for all p, q, and that $B_{p,q}$ is $m_B \times n_B$ for all p, q. However, instantiations of algorithm MM_3 are generally loosely synchronous. As a result, the algorithm may display sub-optimal performance because of idle time accumulated by processes waiting for "not-yet-used" rows of B. For example, let $A_{12\times12}$ and $B_{12\times2}$ be linearly distributed on a 6×2 process grid, so that each $A_{p,q}$ is 2×6 and each $B_{p,q}$ is 2×1. Consider using MM_3 to multiply A and B, Figure 15 illustrates the affect of loose synchronization. We see that process rows are sometimes forced to wait for new components of B.

Algorithm MM_4 attempts to fix this synchronization problem by using a *slide* primitive at the initial stage to rearrange the local indices of matrices A and B. Algorithm MM_4 row version is given by the process-(p, q) pseudocode in Figure 16 and is illustrated in Figure 17.

```
algorithm MM_3 [row version]
    C := βC;                                      //scale matrix C
    (k,j) ← ν_A(μ_B^{-1}(p,0,P,M_B),Q,N_A);      //get col index of A matching row 0 of B
    S ← row broadcast_(k) A_(:,j:);               //row broadcast submatrix A at root k
    a := 0;   r_A := Q;
    b := 0;   r_B := P;                           //initialization
    while (r_B > 0)                               //repeat P times
        r := min{(n_S − a),(m_B − b)};            //calculate the maximum # of inner indices
        C := C + αS_(:,a:a+r) B_(b:b+r,:);        //update local matrix C
        a := a + r;   b := b + r;
        if (a = n_S) then                         //matrix A has used all of the data
            k := (k + 1) mod Q;                   //get next root
            if (r_A > 1) then                     //if it is not the last step
                S ← row broadcast_(k) A;          //row broadcast matrix A at root k
            else if (r_A = 1) then                //if it is the last step
                S ← row broadcast_(k) A_(:,:j−1); //row broadcast submatrix A at root k
            end if
            a := 0;   r_A := r_A − 1;
        end if
        if (b = m_B) then                         //matrix B has used all of the data
            col shift_(−1,0) B;                   //column shift matrix B
            b := 0;   r_B := r_B − 1;
        end if
    end while
end algorithm
```

Figure 13: Pseudocode for MM_3 algorithm row version.

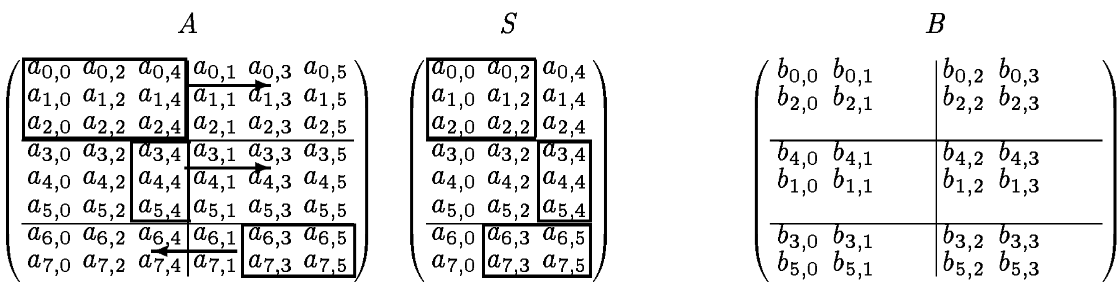

Figure 14a (MM_3 example): Processes $(0,0)$, $(1,0)$, and $(2,1)$ *row broadcast* their data. S is a temporary buffer storing the local matrix broadcast. Each process computes as much of the matrix multiplication as can be done locally (data in S used in the multiplies are marked with solid boxes).

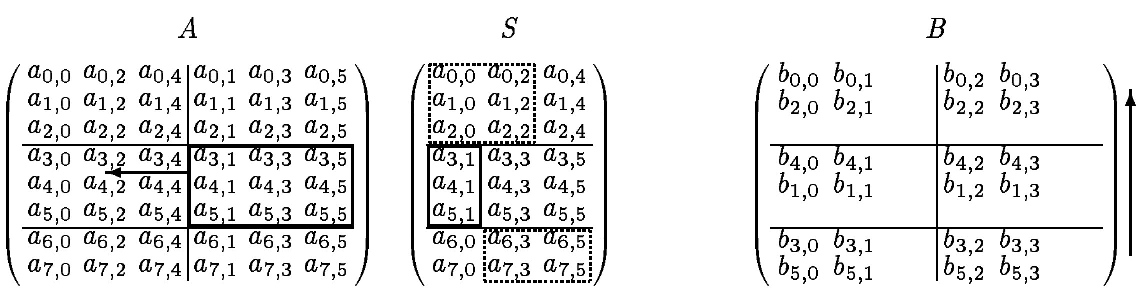

Figure 14b (MM_3 example): Process $(1,1)$ *row broadcasts* its data, then processes (1,q), where $q = 0, 1$, compute as much of the matrix multiplication as can be done locally (data in S already used are marked with dashed boxes). B then undergoes *col shift*$_{(-1,0)}$ (this is actually a loosely synchronous communication step).

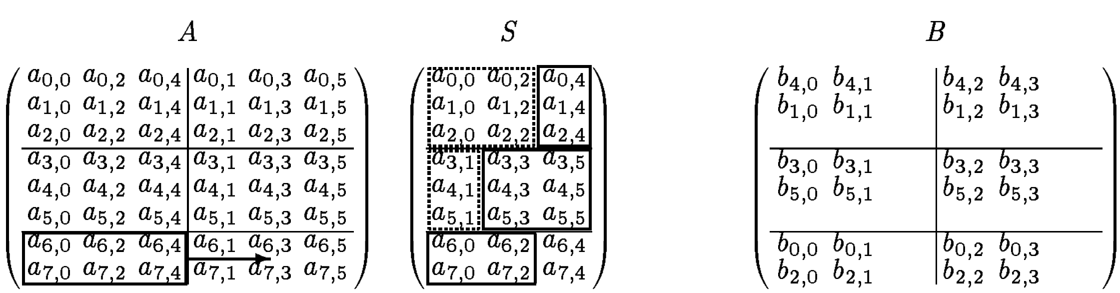

Figure 14c (MM_3 example): Process $(2,0)$ *row broadcasts* its data, then each process computes as much of the matrix multiplication as can be done locally.

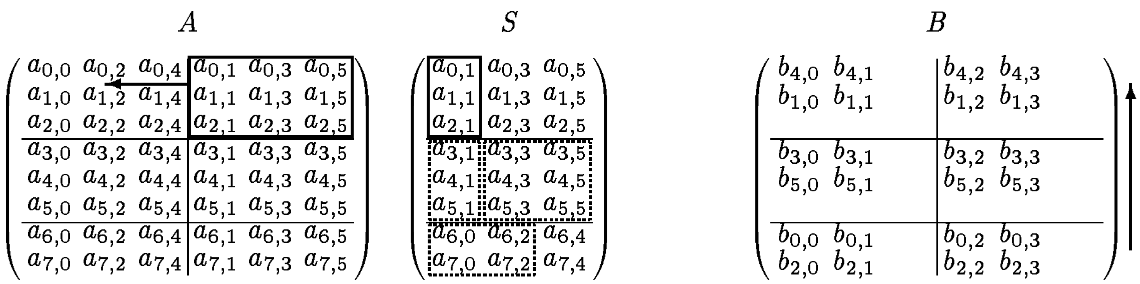

Figure 14d (MM_3 example): Process $(0,1)$ *row broadcasts* its data, then processes (0,q), where $q = 0, 1$, compute as much of the matrix multiplication as can be done locally. B then undergoes *col shift*$_{(-1,0)}$.

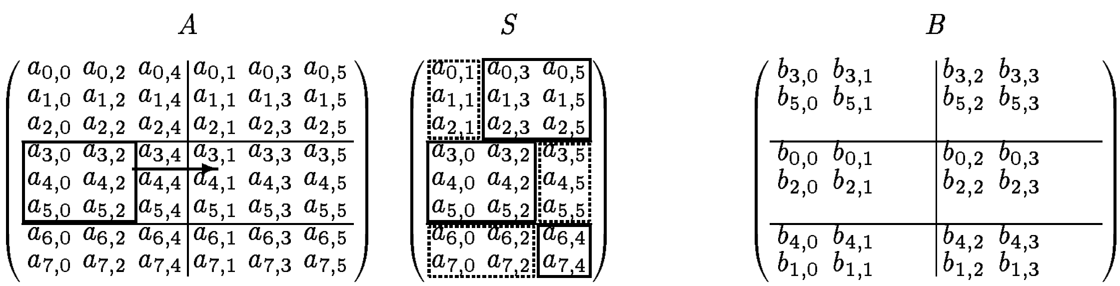

Figure 14e (MM_3 example): Process $(1,0)$ *row broadcasts* its data, then each process computes as much of the matrix multiplication as can be done locally.

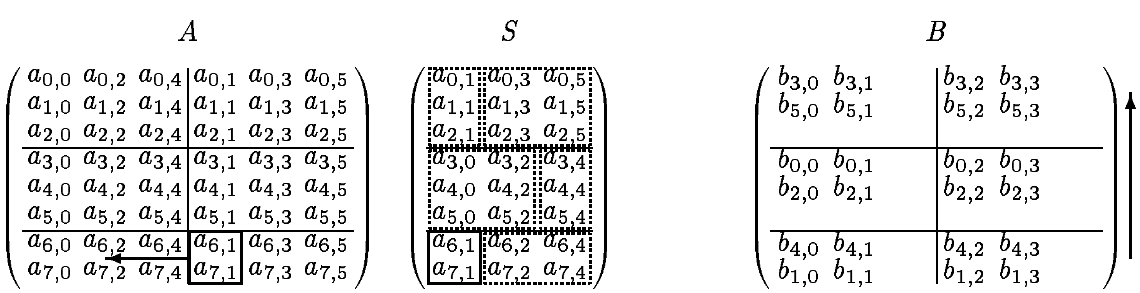

Figure 14f (MM_3 example): Process $(2,1)$ *row broadcasts* its data, processes $(2,q)$, where $q = 0, 1$, compute the remainder of the multiplies. Then, B undergoes *col shift*$_{(-1,0)}$, restoring the original distribution of matrix B.

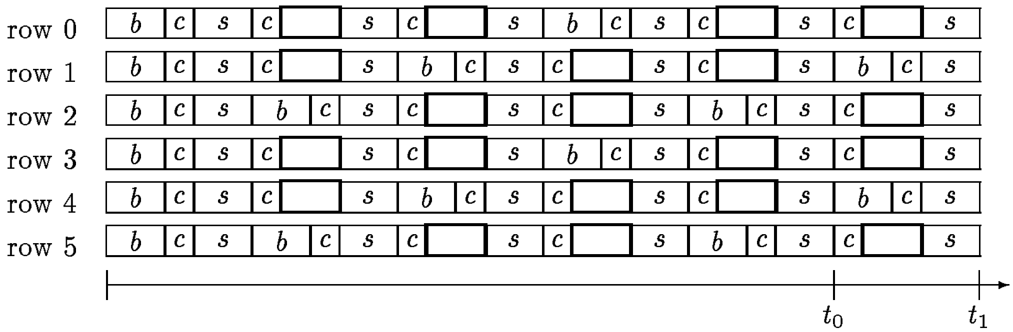

Figure 15: The qualitative affect of loose synchronization for algorithm MM_3 multiplying matrices $A_{12\times 12}$ and $B_{12\times 2}$, linearly distributed on a 6×2 process grid. The letters *b*, *c*, and *s* represent *broadcast*, *computation*, and *shift*, respectively. Time t_1 is required to complete the multiplication, whereas time t_0 would be required if the algorithm were synchronous.

algorithm MM_4 *[row version]*
  $C := \beta C;$ //scale matrix C
  $(k, j) \leftarrow \nu_A(\mu_B^{-1}(p, 0, P, M_B), Q, N_A);$ //get col index of A matching row 0 of B
  row slide$_{(0,-j)}$ $A;$ //initial row slide of matrix A
  $S \leftarrow$ *row broadcast*$_{(k)}$ $A;$ //row broadcast matrix A at root k
  $a := 0; \quad r_A := Q;$
  $b := 0; \quad r_B := P;$ //initialization
  while $(r_B > 0)$ //repeat P times
    $r := \min\{(n_S - a), (m_B - b)\};$ //calculate the maximum # of inner indices
    $C := C + \alpha S_{(:,a:a+r)} B_{(b:b+r,:)};$ //update local matrix C
    $a := a + r; \quad b := b + r;$
    if $(a = n_S)$ **then** //matrix A has used all of the data
      if $(r_A > 1)$ **then**
        $k := (k+1) \bmod Q;$ //get next root
        $S \leftarrow$ *row broadcast*$_{(k)}$ $A;$ //row broadcast matrix A at root k
      end if
      $a := 0; \quad r_A := r_A - 1;$
    end if
    if $(b = m_B)$ **then** //matrix B has used all of the data
      col shift$_{(-1,0)}$ $B;$ //column shift matrix B
      $b := 0; \quad r_B := r_B - 1;$
    end if
  end while
  slide$_{(0,j)}$ $A;$ //restore initial data distribution of A
end algorithm

Figure 16: Pseudocode for MM_4 algorithm row version.

Figure 17a (MM_4 example): Processes undergo *slide*$_{(0,-t)}$ A, where $t = 0$, 2, and 1 for process rows 0, 1, and 2, respectively.

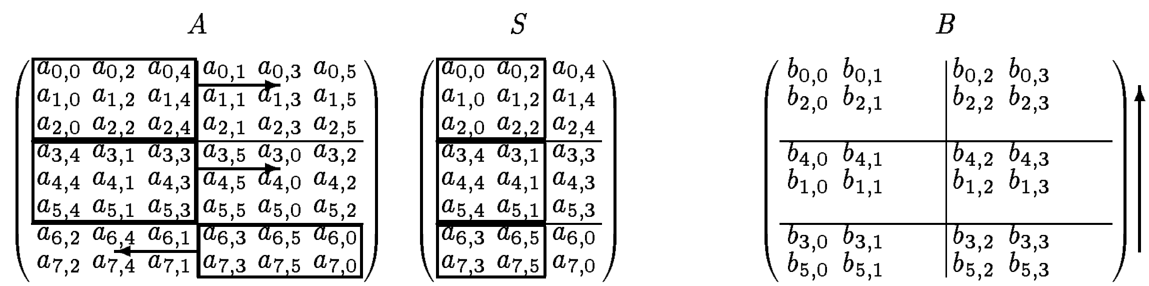

Figure 17b (MM_4 example): Processes $(0,0)$, $(1,0)$, and $(2,1)$ *row broadcast* their data. S is a temporary buffer storing the local matrix broadcast. Each process computes as much of the matrix multiplication as can be done locally. B then undergoes *col shift*$_{(-1,0)}$.

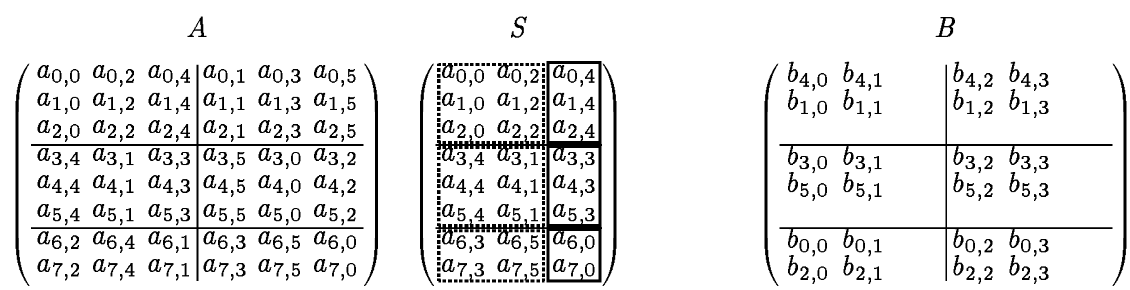

Figure 17c (MM_4 example): Each process computes as much of the matrix multiplication as can be done locally.

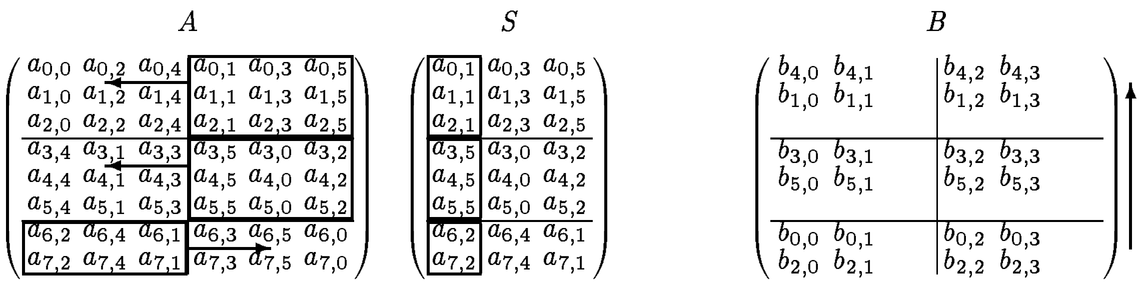

Figure 17d (MM_4 example): Processes $(0,1)$, $(1,1)$, and $(2,0)$ *row broadcast* their data, then each process computes as much of the matrix multiplication as can be done locally. B then undergoes *col shift*$_{(-1,0)}$.

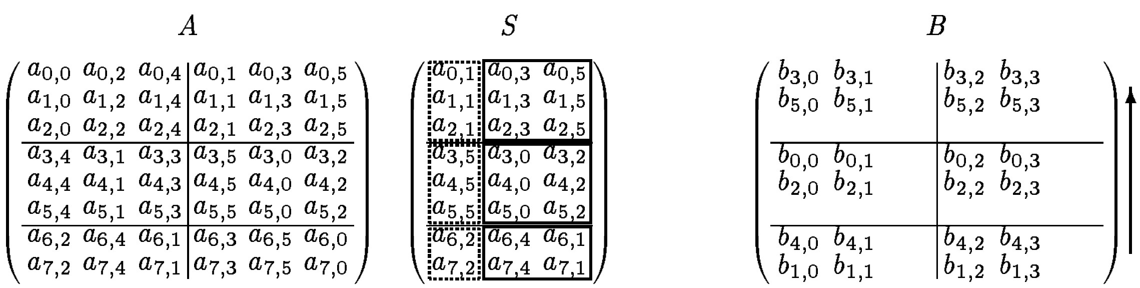

Figure 17e (MM_4 example): Each process computes as much of the matrix multiplication as can be done locally. B then undergoes *col shift*$_{(-1,0)}$.

Figure 17f (MM_4 example): Processes undergo *slide*$_{(0,t)}$ A to restore the originial distribution of matrix A.

The MM_5 (a generalization of *BiMMeR*'s *BMR* algorithm) uses another approach to extend *Fox*'s algorithm. It works to broadcast the exact amount of columns of matrix A that can be locally multiplied to matrix B [15, 16]. The number of columns of matrix A broadcast is limited by the number of rows of matrix B in each process row. There are exactly P steps of broadcast of matrix A. However, when the grid shape is $P > Q$, the columns of matrix A to be broadcast may lie in two processes. Thus, this broadcasting step will include two smaller broadcasts of matrix A. When the grid shape is $P < Q$, the columns of matrix A to be broadcast may lie in more than two processes. This broadcasting step is expensive. So the row version of MM_5 algorithm will be only applied for the $P \geq Q$ grid shape situation and the column version of MM_5 algorithm can only apply for the $Q \geq P$ grid shape situation. Algorithm MM_5 row version is given by the process-(p, q) pseudocode in Figure 18 and Figure 19 illustrates the behavior of MM_5 row version for dealing with a non-square grid and non-square matrices.

```
algorithm MM5 [row version P ≥ Q]
    C := βC;                                  //scale matrix C
    (k, j) ← ν_A(μ_B^{-1}(p, 0, P, M_B), Q, N_A);   //get col index of A matching row 0 of B
    r_B := P;                                 //initialization
    while (r_B > 0)                           //repeat P times
        if (j + m_B > n_A) then
            S(:,:m_B−j) ← row broadcast(k) A(:,j:);//two broadcasts at roots k and k + 1
            S(:,m_B:) ← row broadcast(k+1 mod Q) A(:,:m_B+j−n_A);
        else
            S ← row broadcast(k) A(:,j:);     //only one broadcast at root k
        C := C + αSB;                         //update local matrix C
        if (j + m_B ≥ n_A) then
            k := (k + 1) mod Q;               //get next root
            j := (j + m_B) mod n_A;           //get next submatrix A in the next root
        else
            j := j + m_B                      //get next submatrix A in the same root
        col shift(−1,0) B;                    //column shift matrix B
        r_B := r_B − 1;
    end while
end algorithm
```

Figure 18: Pseudocode for MM_5 algorithm row version.

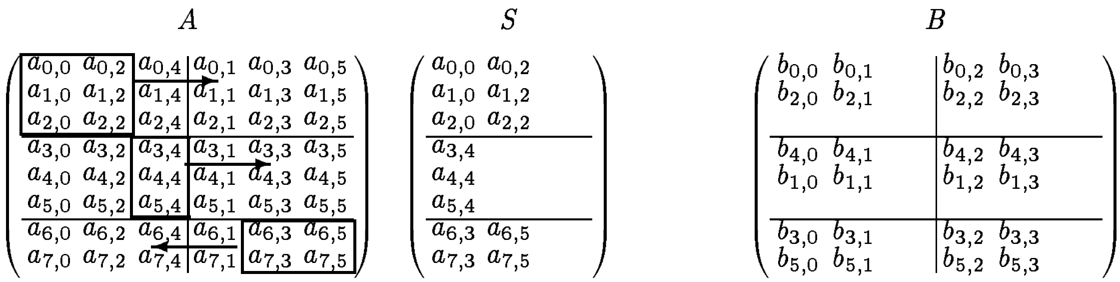

Figure 19a (MM_5 example): Processes $(0,0)$, $(1,0)$, and $(2,1)$ *row broadcast* their data. S is a temporary buffer storing the local matrix broadcast.

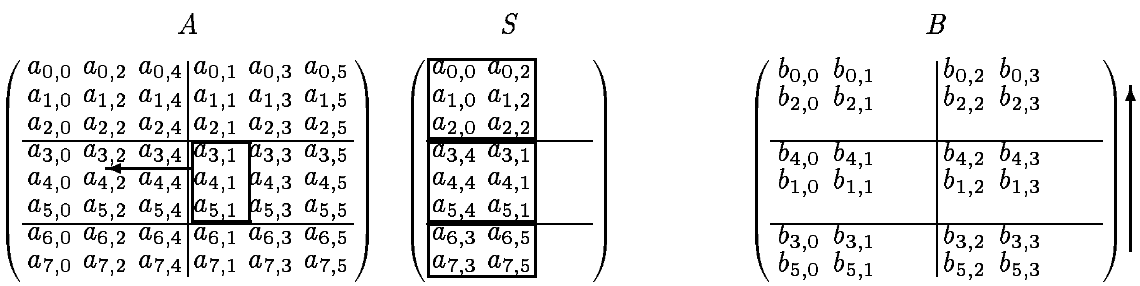

Figure 19b (MM_5 example): Process $(1,1)$ *row broadcasts* its data. Each process computes as much of the multiply as can be done locally (data in S used in multiplies are marked with solid boxes). Then B undergoes $shift_{(-1,0)}$.

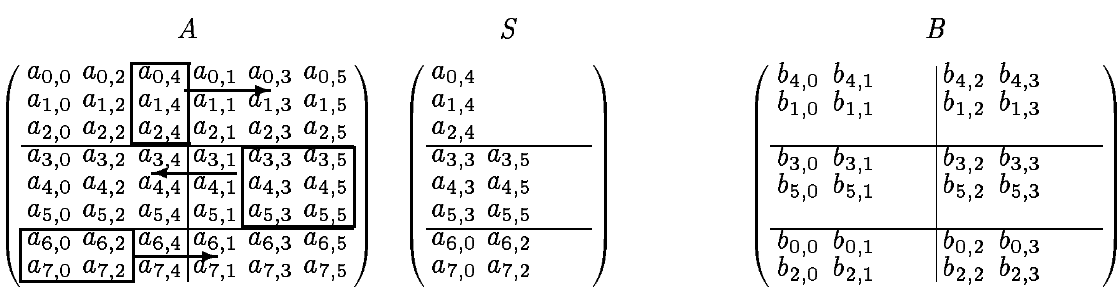

Figure 19c (MM_5 example): Processes $(0,0)$, $(1,1)$, and $(2,0)$ *row broadcast* their data.

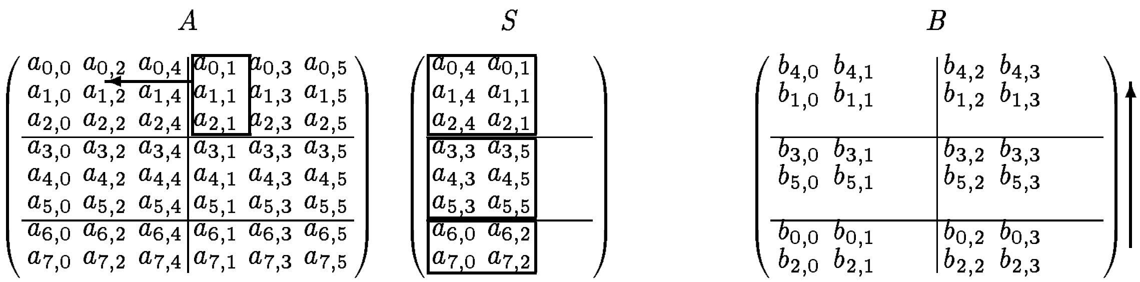

Figure 19d (MM_5 example): Process $(0,1)$ *row broadcasts* its data. Each process computes as much of the multiply as can be done locally. Then B undergoes $\textit{shift}_{(-1,0)}$.

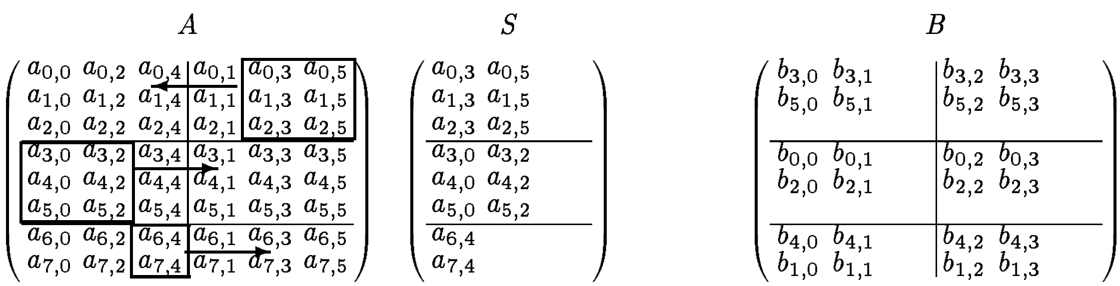

Figure 19e (MM_5 example): Processes $(0,1)$, $(1,0)$, and $(2,1)$ *row broadcast* their data.

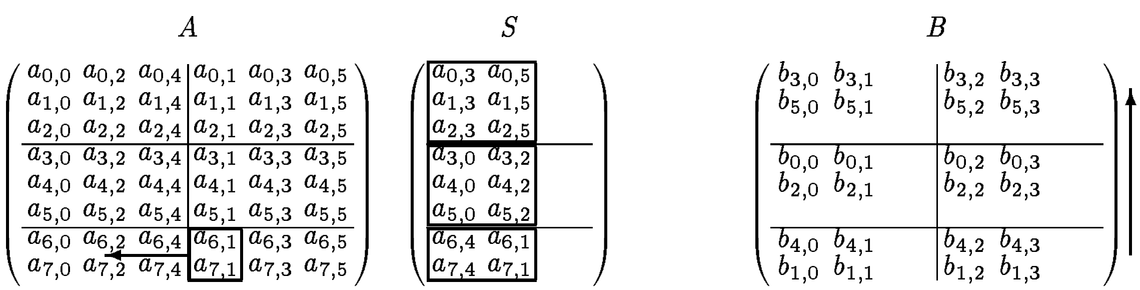

Figure 19f (MM_5 example): Process $(2,1)$ *row broadcasts* its data. Each process computes as much of the multiply as can be done locally. Then B undergoes $\textit{shift}_{(-1,0)}$, restoring the original distribution of matrix B.

3.5 The *Broadcast-Broadcast* approach

The *Broadcast-Broadcast* approach is a new approach for parallel dense matrix multiplication. The only communication primitive used is the *broadcast* primitive. The advantage of this approach is the simplicity of implementation. There is no initial data movement like the *align* of *Cannon*'s algorithms and the *slide* of the MM_4 algorithm and the implementation of *broadcast* is simple than that of *shift*. Also, both the computation and the communication in this algorithm are more synchronous. Our new algorithm *BB* is given by the process-(p, q) pseudocode in Figure 20 and is illustrated in Figure 21.

```
algorithm BB
    C := βC;                                  //scale matrix C
    r_A := r_B := 0;                          //initial roots for broadcast
    a := n_Â := 0;
    b := m_B̂ := 0;                            //initialization
    while ((r_A < Q) or (r_B < P))            //repeat max(P, Q) times
        if ((a = n_Â) and (r_A < Q)) then     //matrix A has used all of the data
            Â ← row broadcast_(r_A) A;        //row broadcast matrix A at root r_A
            r_A := r_A + 1;   a := 0;
        end if
        if ((b = m_B̂) and (r_B < P)) then     //matrix B has used all of the data
            B̂ ← col broadcast_(r_B) B;        //column broadcast matrix B at root r_B
            r_B := r_B + 1;   b := 0;
        end if
        r := min{(n_Â − a), (m_B̂ − b)};       //calculate the maximum # of inner indices
        C := C + αÂ_(:,a:a+r) B̂_(b:b+r,:);    //update local matrix C
        a := a + r;   b := b + r;
    end while
end algorithm
```

Figure 20: Pseudocode for *BB* algorithm.

Figure 21a (*BB* example): Initial data distributions of matrix A and matrix B. Processes $(0,0)$, $(1,0)$, and $(2,0)$ *row broadcast* matrix A and processes $(0,0)$ and $(0,1)$ *col broadcast* matrix B.

Figure 21b (*BB* example): Matrix $\hat{A}$ and matrix $\hat{B}$ are temporary buffers storing the local matrices broadcast. Each process computes as much of the matrix multiplication as can be done locally (data in $\hat{A}$ and $\hat{B}$ used for matrix multiplication are marked with solid boxes). Matrix B has used all of the data and will be broadcast.

Figure 21c (*BB* example): After processes $(1,0)$ and $(1,1)$ *col broadcast* matrix B, each process computes as much of the matrix multiplication as can be done locally (data in $\hat{A}$ already used are marked with dashed boxes). Matrix A has used all of the data and will be broadcast.

Figure 21d (BB example): After processes $(0,1)$, $(1,1)$, and $(2,1)$ *row broadcast* matrix A, each process computes as much of the matrix multiplication as can be done locally. Matrix B has used all of the data and will be broadcast.

Figure 21e (BB example): After processes $(2,0)$ and $(2,1)$ *col broadcast* matrix B, each process computes as much of the matrix multiplication as can be done locally. Matrix A row broadcasts Q (here $Q=2$) times and matrix B column broadcasts P (here $P=3$) times. The algorithm finished.

4 Theoretical Analysis

In this section, we present the communication cost analysis and the memory requirement analysis for the matrix multiplication algorithms. First, we analyze the communication cost for the communication primitives used by these algorithms. Then, we discuss the communication cost for each matrix multiplication algorithm and give the asymptotic analysis of the incremental memory requirement of each algorithm. Finally, we discuss the poly-algorithmic approach for parallel dense matrix multiplication.

The following communication cost analysis is based on a two-dimensional logical process grid topology. A logical process grid can be mapped to a set of physical processors [20]. Figure 22 gives an example of how to map a logical grid to a physical processor mesh. The purpose of the analysis is to provide a relatively simple theoretical performance model, although the physical processor topologies may be complicated and have distinct features. Here is the message-passing model. Let α be the *startup* time for a single message transmission. Let β^{-1} be the bandwidth in both the row and column dimensions of the logical process grid. Also, let c be the contention factor for both the row and column dimensions, say, $c = 1$ for a torus while $c = 2$ for a mesh.

Again, the process grid $\mathcal{G}_{\mathcal{P}\times\mathcal{Q}}$ has P rows and Q columns. Matrix A is of dimension $M \times K$, matrix B is of dimension $K \times N$, and thus matrix C is of $M \times N$. For the purpose of this analysis, we assume that matrices are evenly distributed on the grid $\mathcal{G}_{\mathcal{P}\times\mathcal{Q}}$, the local matrix size of matrix A is $\lceil \frac{M}{P} \times \frac{K}{Q} \rceil$, the local matrix size of matrix B is $\lceil \frac{K}{P} \times \frac{N}{Q} \rceil$, and the local matrix size of matrix C is $\lceil \frac{M}{P} \times \frac{N}{Q} \rceil$. We also assume that the matrix elements are represented using k bytes, such as $k = 8$ for double precision float point numbers.

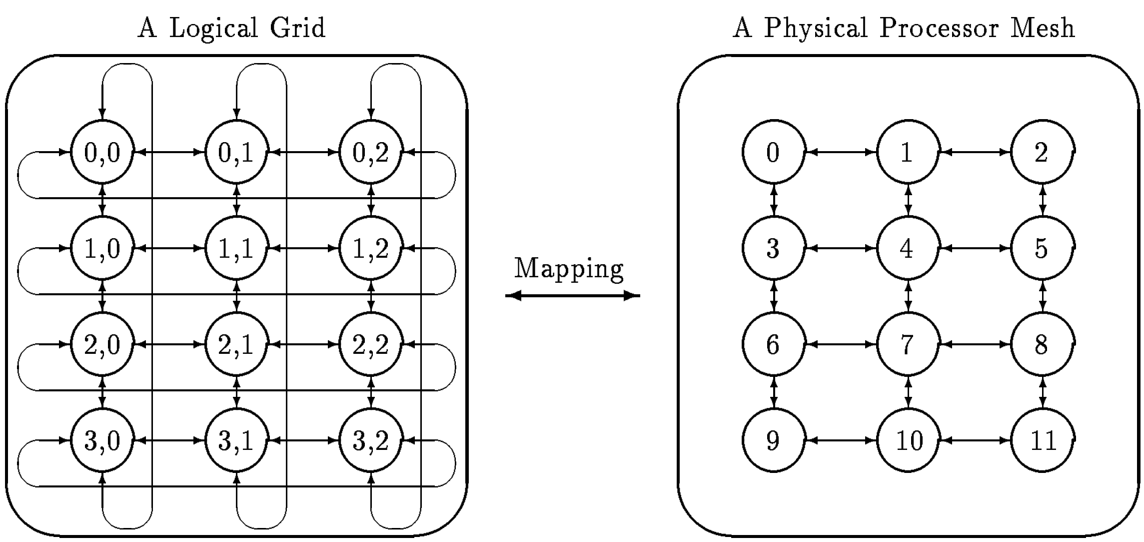

Figure 22: Conceptual mapping from a 2-dimensional logical process grid to a physical processor mesh.

4.1 Communication cost analysis for communication primitives

First, we discuss the cost of communication primitives (*i.e.*, *broadcast*, *slide*, *align*, and *shift*) based on one dimension (either row or column dimension) of the logical process grid; the same results apply to the other dimension. Suppose that there are p processes and message length is l bytes.

For the *broadcast* primitive, assuming a minimum spanning tree is used and thus there is no contention, the communication cost is modelled by:

$$T_{broadcast} = (\alpha + \beta l) \log p. \tag{4}$$

The *slide* primitive transfers part of the local data to the neighbor process, thus the maximum amount of message transferred is less than l. The maximum communication cost of *slide* primitive is approximately modelled by:

$$T_{slide} = \alpha + c\beta l, \tag{5}$$

where c will account for contention.

In the *stride* primitive of stride q, each process i sends the local data to process $(i + q) \bmod p$. Thus, the maximum communication cost of *stride* primitive is modelled by:

$$T_{stride} = \alpha + c\beta l, \tag{6}$$

where c will account for contention.

Combining Equation 5 and Equation 6, the maximum communication cost of *align* primitive is modelled by:

$$\begin{aligned} T_{align} &= T_{slide} + T_{stride} \\ &= 2(\alpha + c\beta l). \end{aligned} \tag{7}$$

In the *shift* primitive, each process transfers its local data to its neighbor process. The communication cost is modelled by:

$$T_{shift} = \alpha + c\beta l, \tag{8}$$

where c will account for contention.

4.2 Communication cost analysis for matrix multiplication algorithms

Assuming above models, we present the following analysis for the matrix multiplication algorithms.

4.2.1 *Cannon*'s approach (the *2D-Systolic* approach)

Consider the initial alignment of matrix A along the row dimension, the communication cost for the *align* primitive is as follows:

$$T_{A\_row\_align} = 2(\alpha + c\beta k \lceil \frac{MK}{PQ} \rceil). \tag{9}$$

Similarly, the initial alignment of matrix B takes:

$$T_{B\_col\_align} = 2(\alpha + c\beta k \lceil \frac{KN}{PQ} \rceil). \tag{10}$$

For the matrix C-stationary version, matrix A will be shifted along the row dimension and matrix B will be shifted along the column dimension. Each *row shift* primitive of matrix A takes $c(\alpha + \beta k \lceil \frac{MK}{PQ} \rceil)$ and there are Q such *shifts*. The total cost for *shift A* is modelled by:

$$T_{A\_row\_shift} = Q(\alpha + c\beta k \lceil \frac{MK}{PQ} \rceil). \tag{11}$$

The matrix B will be shifted P times in the column dimension, thus

$$T_{B\_col\_shift} = P(\alpha + c\beta k \lceil \frac{KN}{PQ} \rceil). \tag{12}$$

The total communication cost of matrix C-stationary version is, in combination, modelled as follows:

$$\begin{aligned} T_{C\_comm} &= 2(T_{A\_row\_align} + T_{B\_col\_align}) + T_{A\_row\_shift} + T_{B\_col\_shift} \\ &= (8 + P + Q)\alpha + c\beta k \lceil \frac{4MK + 4KN + QMK + PKN}{PQ} \rceil. \end{aligned} \tag{13}$$

For the matrix A-stationary version, matrix B will be shifted along both the row and column dimensions and matrix C will be shifted along the row dimension. The total communication cost of matrix A-stationary version is modelled as follows:

$$\begin{aligned} T_{A\_comm} &= 2(T_{A\_row\_align} + T_{B\_col\_align}) + T_{B\_row\_shift} + T_{B\_col\_shift} + T_{C\_row\_shift} \\ &= (8 + P + 2Q)\alpha + c\beta k \lceil \frac{4MK + 4KN + QKN + PKN + QMN}{PQ} \rceil, \end{aligned} \tag{14}$$

where

$$T_{B\_row\_shift} = Q(\alpha + c\beta k \lceil \frac{KN}{PQ} \rceil), \tag{15}$$

$$T_{B\_col\_shift} = P(\alpha + c\beta k \lceil \frac{KN}{PQ} \rceil), \tag{16}$$

$$T_{C\_row\_shift} = Q(\alpha + c\beta k \lceil \frac{MN}{PQ} \rceil). \tag{17}$$

For the matrix B-stationary version, matrix A will be shifted along both the row and column dimensions and matrix C will be shifted along the column dimension. The total communication cost of matrix B-stationary version is modelled as follows:

$$T_{B\_comm} = 2(T_{A\_row\_align} + T_{B\_col\_align}) + T_{A\_row\_shift} + T_{A\_col\_shift} + T_{C\_col\_shift}$$

$$= \quad (8+2P+Q)\alpha + c\beta k\left\lceil\frac{4MK+4KN+QMK+PMK+PMN}{PQ}\right\rceil, \qquad (18)$$

where

$$T_{A\_row\_shift} \quad = \quad Q(\alpha + c\beta k\left\lceil\frac{MK}{PQ}\right\rceil), \qquad (19)$$

$$T_{A\_col\_shift} \quad = \quad P(\alpha + c\beta k\left\lceil\frac{MK}{PQ}\right\rceil), \qquad (20)$$

$$T_{C\_col\_shift} \quad = \quad P(\alpha + c\beta k\left\lceil\frac{MN}{PQ}\right\rceil). \qquad (21)$$

Evidently, based on above formulas we can reduce the communication cost by choosing the appropriate version of *Cannon*'s algorithms that allows the matrix with the largest size to remain stationary. However, we should also consider the communication overhead introduced by versions A and B. In both cases, one matrix (either matrix A or matrix B) will be shifted in both dimensions. Intuitively, if matrix A has a really large size, we can leave matrix A remain stationary and vice verse.

4.2.2 *Fox*'s approach (the *Broadcast-Multiply-Roll* approach)

For algorithm MM_3 row version, the matrix A will be broadcast $Q+1$ times. In the first broadcast step, part of the local matrix will be broadcast and the remaining part will be broadcast in the last one. For the other $Q-1$ times of broadcast, the whole local matrix will be broadcast. The communication cost for the row broadcast of matrix A is modelled as follows:

$$T_{A\_row\_broadcast} = (Q+1)\log Q\alpha + \beta Q\log Qk\left\lceil\frac{MK}{PQ}\right\rceil. \qquad (22)$$

Matrix B will be shifted P times along the column dimension and the communication cost is modelled as follows:

$$T_{B\_col\_shift} = P(\alpha + c\beta k\left\lceil\frac{KN}{PQ}\right\rceil). \qquad (23)$$

The total communication cost is modelled by:

$$T_{MM3row\_comm} = (P + Q\log Q + \log Q)\alpha + c\beta k\left\lceil\frac{KN}{Q}\right\rceil + \beta k\left\lceil\frac{MK\log Q}{P}\right\rceil. \qquad (24)$$

Similarly, the communication cost of the column version of algorithm MM_3 is modelled by:

$$T_{MM3col\_comm} = (Q + P\log P + \log P)\alpha + c\beta k\left\lceil\frac{MK}{P}\right\rceil + \beta k\left\lceil\frac{KN\log P}{Q}\right\rceil. \qquad (25)$$

Algorithm MM_4 initializes matrix A with a *slide* primitive. The communication cost of the row version of algorithm MM_4 is modelled by:

$$T_{MM4row\_comm} \quad = \quad 2T_{A\_row\_slide} + T_{A\_row\_broadcast} + T_{B\_col\_shift}$$

$$= (2 + P + Q\log Q)\alpha + c\beta k\left\lceil\frac{2MK}{PQ}\right\rceil + c\beta k\left\lceil\frac{KN}{Q}\right\rceil + \beta k\left\lceil\frac{MK\log Q}{P}\right\rceil. \quad (26)$$

and the cost of the column version of MM_4 is modelled by:

$$\begin{aligned} T_{MM4col\_comm} &= 2T_{B\_col\_slide} + T_{B\_col\_broadcast} + T_{A\_row\_shift} \\ &= (2 + Q + P\log P)\alpha + c\beta k\left\lceil\frac{2KN}{PQ}\right\rceil + c\beta k\left\lceil\frac{MK}{P}\right\rceil + \beta k\left\lceil\frac{KN\log P}{Q}\right\rceil. \quad (27)\end{aligned}$$

For algorithms MM_5, the *row* version applies only to the situation of $P \geq Q$ as previously mentioned. It broadcasts the exact amount of columns of matrix A that can be locally multiplied to matrix B. Thus, there are exactly P steps of broadcast of matrix A. When $P > Q$, the columns of matrix A to be broadcast may lie in two processes. Thus, this step of *broadcast* will include two smaller *broadcasts*. It's hard to predict how many times this situation will occur, because it is dependent on the grid shape. In the best situation (*i.e.*, columns of A to be broadcast lie in one process, actually when P is a multiple of Q), the communication cost is modelled by:

$$T_{MM5row\_comm} = (P + P\log Q)\alpha + c\beta k\left\lceil\frac{KN}{Q}\right\rceil + \beta k\left\lceil\frac{MK\log Q}{P}\right\rceil \quad where\ P \geq Q. \quad (28)$$

For the *column* version of MM_5 algorithm, the communication cost is modelled by:

$$T_{MM5col\_comm} = (Q + Q\log P)\alpha + c\beta k\left\lceil\frac{MK}{P}\right\rceil + \beta k\left\lceil\frac{KN\log P}{Q}\right\rceil \quad where\ Q \geq P. \quad (29)$$

4.2.3 The *Broadcast-Broadcast* approach

The *BB* algorithm is quite simple. The only communication primitive is *broadcast*. There is total Q times of the row broadcast of matrix A and P times of the column broadcast of matrix B. The communication costs of broadcasts are modelled as follows:

$$T_{A\_row\_broadcast} = Q(\alpha + \beta\left\lceil\frac{MK}{PQ}\right\rceil)\log Q \quad (30)$$

and

$$T_{B\_col\_broadcast} = P(\alpha + \beta\left\lceil\frac{KN}{PQ}\right\rceil)\log P. \quad (31)$$

The total communication cost is modelled by:

$$\begin{aligned} T_{BB\_comm} &= T_{A\_row\_broadcast} + T_{B\_col\_broadcast} \\ &= (Q\log Q + P\log P)\alpha + \beta k\left\lceil\frac{MK\log Q}{P}\right\rceil + \beta k\left\lceil\frac{KN\log P}{Q}\right\rceil. \quad (32)\end{aligned}$$

Table 2 summarizes the communication cost of the parallel dense matrix multiplication algorithms based on above analysis.

Table 2: Communication Cost Models of Parallel Matrix Multiplication Algorithms	
Algorithm	Communication cost
Cannon C	$(8+P+Q)\alpha + c\beta k\left\lceil\frac{4MK+4KN+QMK+PKN}{PQ}\right\rceil$
Cannon A	$(8+P+2Q)\alpha + c\beta k\left\lceil\frac{4MK+4KN+QKN+PKN+QMN}{PQ}\right\rceil$
Cannon B	$(8+2P+Q)\alpha + c\beta k\left\lceil\frac{4MK+4KN+QMK+PMK+PMN}{PQ}\right\rceil$
MM_3 *row*	$(P+Q\log Q+\log Q)\alpha + c\beta k\left\lceil\frac{KN}{Q}\right\rceil + \beta k\left\lceil\frac{MK\log Q}{P}\right\rceil$
MM_3 *col*	$(Q+P\log P+\log P)\alpha + c\beta k\left\lceil\frac{MK}{P}\right\rceil + \beta k\left\lceil\frac{KN\log P}{Q}\right\rceil$
MM_4 *row*	$(2+P+Q\log Q)\alpha + c\beta k\left\lceil\frac{2MK}{PQ}\right\rceil + c\beta k\left\lceil\frac{KN}{Q}\right\rceil + \beta k\left\lceil\frac{MK\log Q}{P}\right\rceil$
MM_4 *col*	$(2+Q+P\log P)\alpha + c\beta k\left\lceil\frac{2KN}{PQ}\right\rceil + c\beta k\left\lceil\frac{MK}{P}\right\rceil + \beta k\left\lceil\frac{KN\log P}{Q}\right\rceil$
MM_5 *row*	$(P+P\log Q)\alpha + c\beta k\left\lceil\frac{KN}{Q}\right\rceil + \beta k\left\lceil\frac{MK\log Q}{P}\right\rceil$
MM_5 *col*	$(Q+Q\log P)\alpha + c\beta k\left\lceil\frac{MK}{P}\right\rceil + \beta k\left\lceil\frac{KN\log P}{Q}\right\rceil$
BB	$(Q\log Q+P\log P)\alpha + \beta k\left\lceil\frac{MK\log Q}{P}\right\rceil + \beta k\left\lceil\frac{KN\log P}{Q}\right\rceil$

4.3 Memory requirement analysis

The following memory requirement analysis does not include the memory initially used for storing matrices A, B, and C. It is the incremental memory (in bytes) required when using these algorithms.

4.3.1 *Cannon*'s approach (the *2D-Systolic* approach)

Cannon's algorithms are memory-efficient algorithms. During the local computation, no additional memory is required. If we consider the *shift* primitive for shifting matrices, a temporary receiving buffer is required. For the matrix C-stationary version, the incremental memory requirement is as follows:

$$\Theta(k\min(MK, KN)). \tag{33}$$

For the matrix A-stationary version, the incremental memory requirement is given by,

$$\Theta(k\min(KN, MN)) \tag{34}$$

and for the matrix B-stationary version, the incremental memory requirement is as follows:

$$\Theta(k\min(MK, MN)). \tag{35}$$

4.3.2 *Fox*'s approach (the *Broadcast-Multiply-Roll* approach)

Temporary memory is required to store the local matrix that is broadcast in either the row or column dimension. Thus the incremental memory requirement of MM_3 and MM_4 row versions is as follows:

$$\Theta(kMK). \tag{36}$$

The incremental memory requirement of MM_3 and MM_4 column versions is as follows:

$$\Theta(kKN). \tag{37}$$

The MM_5 requires less memory than MM_3 and MM_4. For the row version of MM_5, the incremental memory requirement is as follows:

$$\Theta(kMKQ/P) \quad where\ P \geq Q. \tag{38}$$

For the column version, the incremental memory requirement is as follows:

$$\Theta(kKNP/Q) \quad where\ Q \geq P. \tag{39}$$

4.3.3 The *Broadcast-Broadcast* approach

For the *BB* algorithm, temporary memory is required to store both the local matrix A and the local matrix B that are broadcast in the row and column dimensions respectively. Thus the incremental memory requirement is as follows:

$$\Theta(k(MK + KN)). \tag{40}$$

Table 3 summarizes the incremental memory requirements of the matrix multiplication algorithms based on previous analysis.

Table 3: Incremental Memory Requirements of Parallel Matrix Multiplication Algorithms

Algorithm	Incremental memory requirement (in bytes)
Cannon C	$\Theta(k\min(MK, KN))$
Cannon A	$\Theta(k\min(KN, MN))$
Cannon B	$\Theta(k\min(MK, MN))$
MM_3 *row*	$\Theta(kMK)$
MM_3 *col*	$\Theta(kKN)$
MM_4 *row*	$\Theta(kMK)$
MM_4 *col*	$\Theta(kKN)$
MM_5 *row*	$\Theta(kMKQ/P)$
MM_5 *col*	$\Theta(kKNP/Q)$
BB	$\Theta(k(MK + KN))$

4.4 Poly-algorithmic approach for parallel dense matrix multiplication

Evidence of the diverse memory costs and algorithmic performance indicates that no single algorithm discussed above always achieves the best performance on different matrix shapes and grid topologies. For *Cannon*'s approach, the initial alignment is expensive, but the advantage of *Cannon*'s approach is that we can choose an appropriate version to minimize the amount of data movement (which itself is a poly-algorithm). For example, if the size of matrix A is much larger than that of matrix B and C, we can use *Cannon*'s matrix A-stationary version that leaves matrix A remain stationary. *Fox*'s approach is suitable for extreme grid shapes. If the column dimension of the grid is much larger than the row dimension (*i.e.*, $P > Q$), the row version algorithms can be used. The *BB* algorithm works well when the process counts are small. This is because of the communication cost of *broadcast* primitive in both dimensions.

A poly-algorithm is a practical approach to maximize performance by marrying multiple algorithms. The poly-algorithmic approach refers to the use of a high level run time decision-making mechanism to select the best algorithm for a particular situation from a set of algorithms solving the same problem. There are several important issues related to the poly-algorithmic approach. First, the high level decision-making mechanism should be very simple without incurring additional overhead or it must be incurred once and be amortized over hundreds and thousands of uses to render the cost negligible. It should only evaluate the most important application parameters and make a quick decision. In the case of our matrix multiplication, the matrix size (*i.e.*, M, K, and N) and the grid shape (*i.e.*, P and Q) are only the parameters used. Secondly, a uniform interface should be defined for each algorithm. In our implementation, the *LA_Dmatrices_Mul* object encapsulates three matrices A, B, and C and a pointer pointing to an algorithm that can be dynamically linked to the *LA_Dmatrices_Mul* object according to the result of the decision-making. Furthermore, a set of heuristics that captures the characteristics of each algorithm should be provided for the run time decision-making. These heuristics should allow a less rigorous but quick selection. It is hard to provide a complete set of heuristics for each algorithm, because there are lots of factors that affect the performance of the parallel dense matrix multiplication. Here, we only provide some initial heuristics for our parallel dense matrix multiplication.

1. If the process counts are small, say $P < 5$ and $Q < 5$, the *BB* algorithm is selected.

2. If the size of matrix A (resp, B) is larger than the sum of the size of matrix B (resp, A) and the size of matrix C, *Cannon*'s A (resp, B) stationary version is selected.

3. If the grid is square, the MM_4 row/column versions is selected. If the grid shape is extreme and $P > Q$, the MM_5 row version is selected; otherwise, if $Q > P$, the MM_5 column version is selected.

Further optimization studies are indicated, but are beyond the scope of this paper. We also realize that the performance is not only the consideration of speed but also the consideration of space. The memory requirement of an algorithm sometimes is critical to an application. The incremental memory requirement models for our parallel dense matrix multiplication algorithms analyzed previously are also very important criteria for poly-algorithmic selection, which will be included in future research.

5 Performance Results

In this section, we present the experiment results of these parallel dense matrix multiplication algorithms on the IBM SP2 system at the Maui High Performance Computing Center (MHPCC). We carried out our experiment on IBM SP2 thin nodes using User Space communication subsystem (US). The execution was supported by the IBM's LoadLeveler environment. The program was compiled using the *mpcc* compiler with *O3* optimization option. We used the MPICH 1.0.11 version of the Portable MPI Model Implementation for message passing and the *DGEMM* subroutine of ESSL library for the local matrix multiplication.

The matrix multiplication is of the form $C = \alpha AB + \beta C$, where α and β are randomly selected double precision float numbers. Matrix elements are generated uniformly on the open interval $(-1, 1)$ (excluding 0) in double precision. The observed parallel runtime is measured on the slowest node of the two-dimensional grid and each algorithm is tested 8 times. Performance in gigaflops is computed as $2MKN10^{-9}/t$, where t is the average observed parallel runtime over 8 runs, for the multiplication of an $M \times K$ matrix by a $K \times N$ matrix.

5.1 Performance

We tested the above matrix multiplication algorithms on the situation of square matrices and square grids. First, we tested the actual performance of the local matrix multiplication engine *DGEMM*. Table 4 lists the actual performance in megaflops of *DGEMM* in one thin node of the IBM SP2 system for the problem size of 500×500, 800×800, 1100×1100, and 1400×1400. The actual performance of the kernel *DGEMM* will be considered as the upper bound performance of our parallel dense matrix multiplication algorithms. If we improve the kernel performance, we can consequently improve the performance of the parallel algorithms.

Table 4: Performance of Kernel *DGEMM*

Matrix Size	500^2	800^2	1100^2	1400^2
Mflops	208.04	204.34	206.62	205.03

Then, we tested the matrix multiplication algorithms with different problem sizes per node versus different process grids. Table 5a to j are the results for the problem size per node fixed at 500×500, 800×800, 1100×1100, and 1400×1400 versus grid shapes 2×2, 3×3, 5×5, 7×7, 9×9, and 11×11.

From the above results, we see that MM_3, MM_4, and MM_5 (both the row version and the column version) achieve almost the same performance. This is exactly what is expected according to previous analysis. In the situation of square matrices and a square grid, there is no synchronous problem for MM_3 algorithms, no initial *slides* for MM_4 algorithms, and no small broadcasts for MM_5 algorithms. For this situation, *Cannon*'s C-stationary version is slower than all the *MM* algorithms because of the initial *aligns*, which are expensive. *Cannon*'s A-stationary and B-stationary versions are even slower than *Cannon*'s C-stationary version because there are more data movement in the case of square matrices. *BB* algorithm is faster than other algorithms when process counts are small (*i.e.*, the 2×2 grid, the 3×3 grid, and the 5×5 grid). With the increase of processes, the performance degrades more quickly than others.

Figure 23 shows the performance of the *Cannon*'s C-stationary version as a function of problem size for different numbers of processes from 4 up to 121 processes. Figure 24 shows the performance of the MM_4 row version as a function of problem size for different numbers of processes from 4 up to 121 processes. Figure 25 shows the performance of the *BB* as a function of problem size for different numbers of processes from 4 up to 121 processes.

Figure 26, a 3-dimensional graph, compares the performance of *Cannon*'s C-stationary version, MM_4 row version, and *BB* algorithm in megaflops per node versus the number of processes and the local matrix size. For the algorithm MM_4 row version, the algorithm is not truly scalable because of the *broadcast*. However, since the cost of a *broadcast* grows like $\log p$, for most practical purposes the algorithm is acceptable. The performance of *BB* algorithm degrades more rapidly than that of *MM* algorithms with the increase of concurrency, since there are *broadcasts* in both the row and column dimensions.

Table 5a: Performance in Megaflops Per Node of Algorithm *BB*

Local Matrix Size	500^2	800^2	1100^2	1400^2
Grid Shape	Mflops/node	Mflops/node	Mflops/node	Mflops/node
2×2	148.24	163.92	176.40	176.58
3×3	135.43	153.87	166.84	171.15
5×5	110.98	133.17	147.48	156.71
7×7	108.31	130.23	146.28	154.80
9×9	93.31	116.86	132.87	142.54
11×11	91.57	115.05	130.98	140.29

Table 5b: Performance in Megaflops Per Node of *Cannon* *C*-stationary version

Local Matrix Size	500^2	800^2	1100^2	1400^2
Grid Shape	Mflops/node	Mflops/node	Mflops/node	Mflops/node
2×2	86.78	111.97	126.75	136.98
3×3	83.31	107.89	125.50	133.80
5×5	90.16	113.68	129.02	141.43
7×7	88.34	114.61	133.07	143.67
9×9	81.88	114.75	132.04	144.05
11×11	72.10	111.36	129.13	140.91

Table 5c: Performance in Megaflops Per Node of *Cannon* A-stationary version				
Local Matrix Size	500^2	800^2	1100^2	1400^2
Grid Shape	Mflops/node	Mflops/node	Mflops/node	Mflops/node
2×2	78.14	99.49	116.12	126.05
3×3	71.58	93.82	111.07	123.37
5×5	73.35	97.87	114.97	126.97
7×7	73.46	98.26	116.57	128.12
9×9	70.56	98.19	115.45	124.95
11×11	64.30	95.18	112.70	120.97

Table 5d: Performance in Megaflops Per Node of *Cannon* B-stationary version				
Local Matrix Size	500^2	800^2	1100^2	1400^2
Grid Shape	Mflops/node	Mflops/node	Mflops/node	Mflops/node
2×2	75.84	100.04	116.48	127.69
3×3	71.56	94.57	110.26	122.54
5×5	72.42	97.73	114.30	127.47
7×7	71.18	98.49	115.95	128.03
9×9	61.36	98.32	116.56	127.81
11×11	44.51	92.58	112.87	121.09

Table 5e: Performance in Megaflops Per Node of MM_3 row version				
Local Matrix Size	500^2	800^2	1100^2	1400^2
Grid Shape	Mflops/node	Mflops/node	Mflops/node	Mflops/node
2×2	134.70	152.60	165.96	169.78
3×3	121.38	141.45	156.99	161.68
5×5	111.74	133.69	146.23	156.37
7×7	104.17	126.79	142.53	152.48
9×9	101.41	124.61	140.82	149.96
11×11	96.55	120.39	136.54	144.39

Table 5f: Performance in Megaflops Per Node of MM_3 column version				
Local Matrix Size	500^2	800^2	1100^2	1400^2
Grid Shape	Mflops/node	Mflops/node	Mflops/node	Mflops/node
2×2	134.63	152.87	166.22	169.91
3×3	121.38	141.95	156.37	162.49
5×5	109.63	132.08	146.87	157.28
7×7	103.91	126.97	142.40	151.87
9×9	101.84	125.96	141.32	146.29
11×11	97.57	121.19	136.38	144.94

Table 5g: Performance in Megaflops Per Node of MM_4 row version				
Local Matrix Size	500^2	800^2	1100^2	1400^2
Grid Shape	Mflops/node	Mflops/node	Mflops/node	Mflops/node
2×2	133.69	152.69	166.05	169.78
3×3	120.91	141.95	156.69	162.13
5×5	111.81	132.89	146.73	156.78
7×7	103.57	125.94	142.52	151.78
9×9	101.32	124.56	141.75	148.88
11×11	97.53	120.65	135.99	143.45

Table 5h: Performance in Megaflops Per Node of MM_4 column version				
Local Matrix Size	500^2	800^2	1100^2	1400^2
Grid Shape	Mflops/node	Mflops/node	Mflops/node	Mflops/node
2×2	133.80	152.90	166.25	169.79
3×3	121.28	141.83	156.63	162.47
5×5	110.23	133.34	147.02	157.34
7×7	104.13	127.37	143.15	151.24
9×9	101.14	126.64	140.24	149.69
11×11	97.82	120.96	137.11	144.12

Table 5i: Performance in Megaflops Per Node of MM_5 row version				
Local Matrix Size	500^2	800^2	1100^2	1400^2
Grid Shape	Mflops/node	Mflops/node	Mflops/node	Mflops/node
2×2	133.65	152.93	165.97	170.93
3×3	121.34	142.02	156.04	163.44
5×5	113.18	134.74	148.60	158.93
7×7	106.09	127.42	142.70	153.82
9×9	101.79	125.59	141.10	149.85
11×11	99.46	122.81	138.36	143.98

Table 5j: Performance in Megaflops Per Node of MM_5 column version				
Local Matrix Size	500^2	800^2	1100^2	1400^2
Grid Shape	Mflops/node	Mflops/node	Mflops/node	Mflops/node
2×2	134.16	152.65	165.73	170.53
3×3	121.20	142.22	156.47	163.59
5×5	113.04	135.01	149.33	158.73
7×7	106.00	128.29	143.89	153.54
9×9	102.02	125.59	142.02	149.79
11×11	99.37	122.48	138.35	143.72

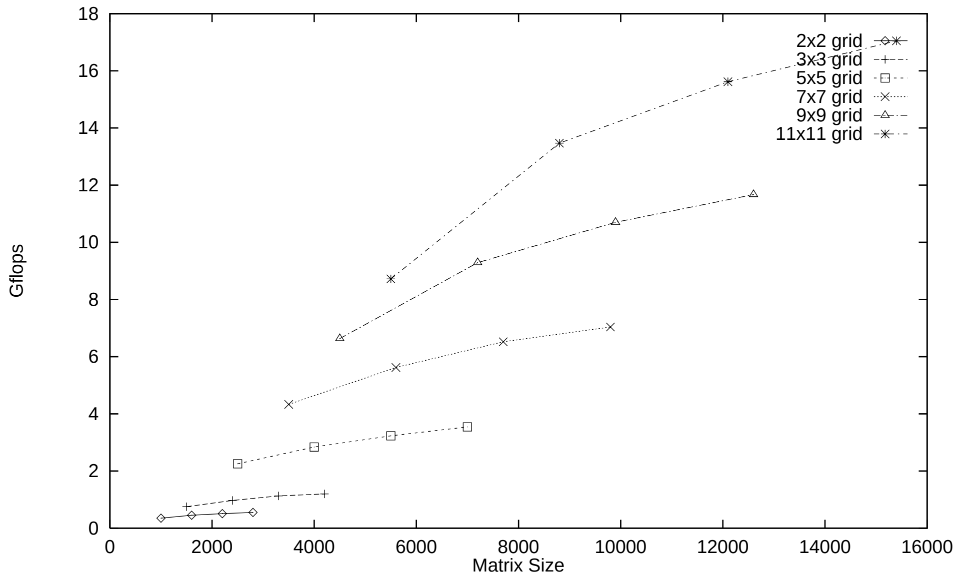

Figure 23: Performance results in gigaflops versus the number of processes and the size of matrix of *Cannon*'s C-stationary version.

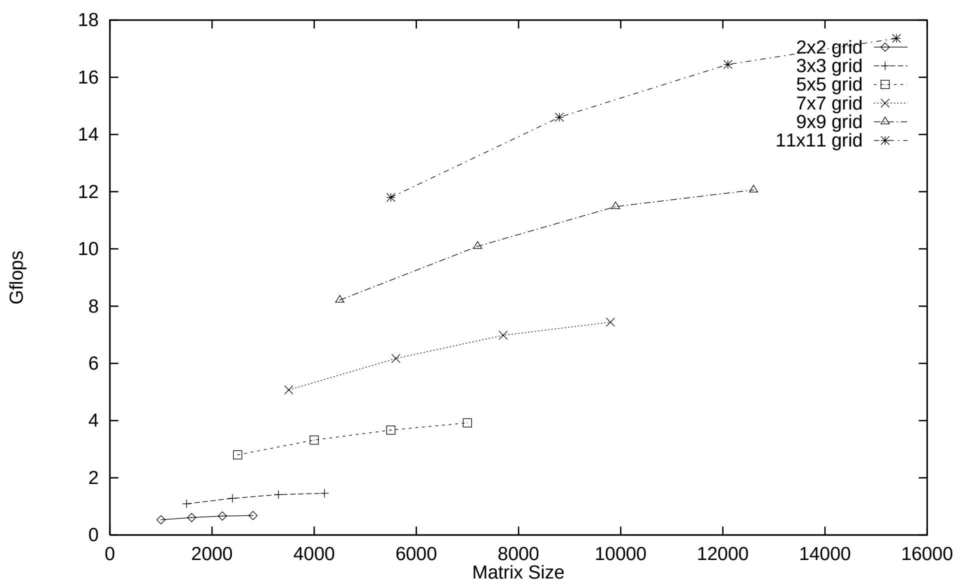

Figure 24: Performance results in gigaflops versus the number of processes and the size of matrix of Algorithms MM_4 row version.

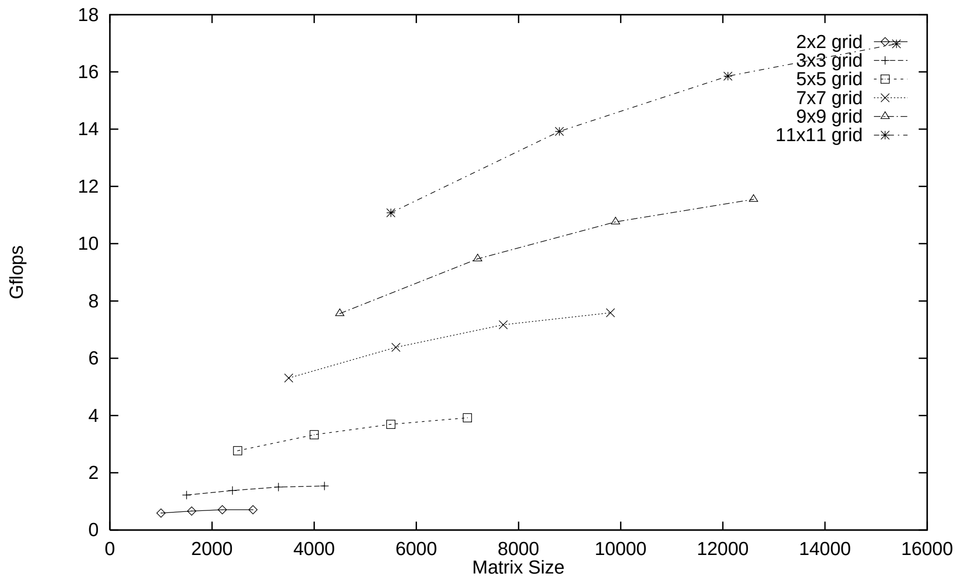

Figure 25: Performance results in gigaflops versus the number of processes and the size of matrix of Algorithms *BB*.

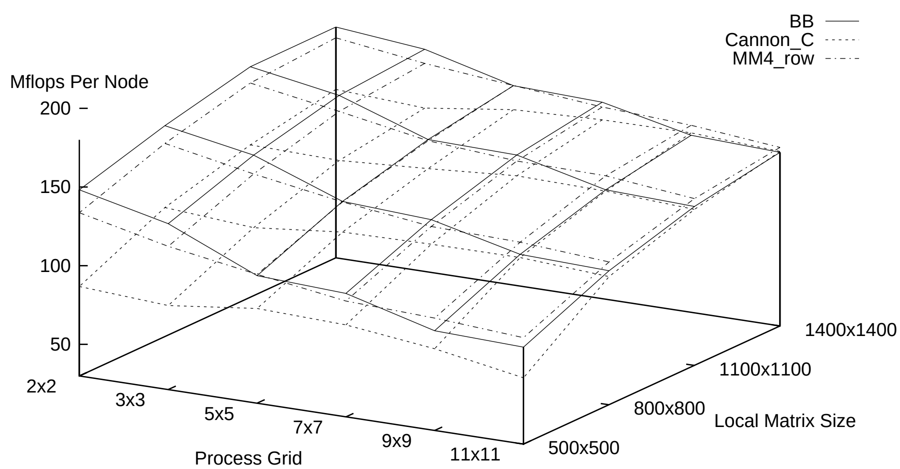

Figure 26: Performance results in megaflops per node versus the number of processes and the size of local matrix of Algorithms *BB*, *Cannon*'s C-stationary version, and MM_4 row version. Since the cost of a *broadcast* grows like $\log p$, the performance of the MM_4 row version drops more quickly than that of the *Cannon* C-stationary version with the increase of process counts. Furthermore, the performance of *BB* degrades even more rapidly than that of the MM_4 row version, because there are *broadcasts* in both the row and column dimensions.

5.2 Comparisons

We tested these matrix multiplication algorithms on different process grid topologies and different matrix shapes in order to compare the relative performance of each other. We chose three process grids. The first grid is a relatively small grid with $P = 4$ and $Q = 5$ (20 nodes). The ratio of P to Q is 0.80. The second grid is a relatively large grid with $P = 7$ and $Q = 9$ (63 nodes). The ratio of P to Q is approximately 0.78 and is almost the same as the first grid. The third grid has almost the same number of nodes as that of the second grid, but has a high aspect ratio shape with $P = 3$ and $Q = 20$ (60 nodes). The ratio of P to Q is 0.15 and Q is not a multiple of P.

We fix the inner dimension K to be 8000 and vary M and N to obtain five different test cases. The *case* 1 has $M = 500$ and $N = 4000$, thus the ratio of $M : K : N$ is $1 : 16 : 8$ and the matrix B has a much larger size than matrix A. The *case* 2 has $M = 1000$ and $N = 4000$ and the ratio of $M : K : N$ is $1 : 8 : 4$. The size of matrix B is four times larger than the size of matrix A. The *case* 3 has $M = 1000$ and $N = 1000$ and the ratio of $M : K : N$ is $1 : 8 : 1$. Thus matrix A and matrix B are of the same size. The *case* 4 is a symmetric case of the *case* 2 with $M = 4000$ and $N = 1000$. The *case* 5 is a symmetric case of the *case* 1 with $M = 4000$ and $N = 500$.

Table 6 lists the performance result on a 4×5 process grid (the parallel run time of the fastest algorithm in each test case is highlighted using bold letters). Because $P < Q$, we expect the column versions of MM algorithm are better than the row versions. However, in *cases* 1 and 2, the row versions of MM_3 and MM_4 are better than the column versions, because the difference of matrix size is more significant than that of this grid (the ratio of P to Q is 0.8). Overall, the BB algorithm performs better than other algorithms because the process counts are small in both dimensions and thus the broadcast is less expensive.

Table 7 lists the performance result on a 7×9 process grid (the parallel run time of the fastest algorithm in each test case is highlighted using bold letters). The performance of BB algorithm, comparing with that on the 4×5 grid case, drops with the increasing process counts in both dimensions. In *case* 1 and *case* 2, *Cannon*'s B-stationary version is the best, because the size of matrix B is much larger than the size of both matrix A and matrix C. *Cannon*'s B-stationary version minimizes the communication cost by leaving the largest matrix B stationary. In *case* 4 and *case* 5, *Cannon*'s A-stationary version is the best because of the same reason. The advantage of *Cannon*'s A-stationary version and B-stationary version will not present, when the process counts are small referring to the 4×5 grid case. Because in small process count case, the broadcast is less expensive and thus the MM algorithms perform well.

Table 8 lists the performance result on a 3×20 process grid (the parallel run time of the fastest algorithm in each test case is highlighted using bold letters). Although the total number of processes in this test case is almost the same as that of the 7×9 grid case, the ratio of grid shapes is quite different. The extreme shape grid situation causes a distinct performance result. In *case* 1 and *case* 2, the MM_5 column version is the best instead of the *Cannon*'s B-stationary version. The broadcast cost of the MM_5 column version is reduced because of the small P.

The distinct performance results of each algorithm for different matrix and grid shapes verified the conclusion of our theoretical performance analysis that no single algorithm always achieves the best performance when dealing with arbitrary matrices and grids. Hence, a poly-algorithm is needed because what is best changes.

Table 6: Performance Result in Parallel Run-Time (seconds) on a 4×5 Process Grid					
Matrix Ratio $M : K : N$	$1:16:8$	$1:8:4$	$1:8:1$	$4:8:1$	$8:16:1$
Algorithms	Run Time	Run Time	Run Time	Run Time	Run Time
MM_3 *row*	19.041	27.204	10.230	33.807	25.511
MM_3 *col*	19.630	28.014	9.901	29.823	21.456
MM_4 *row*	18.948	27.482	9.788	32.310	24.306
MM_4 *col*	23.034	30.299	9.913	30.031	20.883
MM_5 *col*	18.387	26.838	9.207	29.540	20.461
BB	**18.154**	**26.329**	**8.859**	**28.392**	**19.966**
Cannon C	27.515	36.947	13.072	39.159	30.501
Cannon A	40.643	48.260	13.871	31.627	20.087
Cannon B	19.257	31.201	14.049	49.518	41.359

Table 7: Performance Result in Parallel Run-Time (seconds) on a 7×9 Process Grid					
Matrix Ratio $M : K : N$	$1:16:8$	$1:8:4$	$1:8:1$	$4:8:1$	$8:16:1$
Algorithms	Run Time	Run Time	Run Time	Run Time	Run Time
MM_3 *row*	12.075	16.936	8.258	26.854	23.460
MM_3 *col*	16.680	19.742	7.681	19.046	15.194
MM_4 *row*	13.160	16.640	7.432	23.427	19.641
MM_4 *col*	16.079	19.046	7.037	19.496	14.142
MM_5 *col*	14.318	17.959	**6.592**	17.468	14.444
BB	12.218	16.438	6.876	21.013	17.264
Cannon C	17.763	21.326	8.172	22.606	19.449
Cannon A	28.961	32.994	9.044	**16.665**	**11.052**
Cannon B	**11.348**	**16.142**	9.148	31.435	29.933

Table 8: Performance Result in Parallel Run-Time (seconds) on a 3×20 Process Grid					
Matrix Ratio $M : K : N$	$1:16:8$	$1:8:4$	$1:8:1$	$4:8:1$	$8:16:1$
Algorithms	Run Time	Run Time	Run Time	Run Time	Run Time
MM_3 *row*	11.415	19.341	12.618	47.770	45.211
MM_3 *col*	19.051	24.500	10.325	30.755	27.533
MM_4 *row*	11.960	19.222	12.865	50.039	46.462
MM_4 *col*	11.183	15.526	8.120	28.821	25.996
MM_5 *col*	**7.685**	**13.118**	**7.995**	29.991	27.015
BB	9.864	17.426	13.012	46.439	43.616
Cannon C	14.925	20.846	10.590	34.106	30.600
Cannon A	42.102	43.972	12.128	**20.602**	**12.978**
Cannon B	12.118	18.182	10.866	37.119	33.987

6 Summary, Comments, and Future Work

In this paper, we presented several parallel dense matrix multiplication algorithms of the form $C = \alpha AB + \beta C$ on two-dimensional process grid topologies. We gave a taxonomy of these algorithms, which will help avoid confusion of future algorithms. We introduced *data distribution independent* approach and defined *data distribution functions*, *permutation compatibility*, and *algorithmic compatibility*. We analyzed the characteristics of each algorithm and provided initial heuristics for using the poly-algorithm. All these matrix multiplication algorithms were tested on the IBM SP2 system. The experimental results were presented to demonstrate their performance characteristics and the definite need for poly-algorithms.

One of the algorithms with potential advantage for large-size dense matrix multiplication is the Strassen's algorithm [24]. There are two levels of application of Strassen's approach. One (low level) is to use the Strassen's approach to replace *DGEMM* as local multiplication engine [10]. The other (high-level) is to develop a fully parallel Strassen's algorithms. The general parallel Strassen's algorithm that can deal with general cases of rectangular matrix multiplication on rectangular process grids deserves further research. Another approach for parallel dense matrix multiplication is to use a 3-dimensional process topology [3, 8]. The advantage of the 3-D algorithm is that it moves less data than the known 2-D algorithms [3] and thus achieves high performance. However, the 3-D algorithm [3] obviously requires more incremental memory to replicate local matrices than those 2-D algorithms presented in this paper do. So, there is a tradeoff of speed versus space and still a poly-algorithmic selection. We will continue investigating these algorithms and extending the poly-algorithmic set.

One of the key issues of using the poly-algorithm is to determine under what kind of situations an algorithm achieves the best performance. We will further investigate the characteristics of our parallel dense matrix multiplication algorithms and provide more detail heuristics for using the poly-algorithm. However, it is apparently clear that there can not be only one parallel matrix multiplication algorithm. The implications of this are several, and important both for this parallel algorithm for others built on it, and most likely for many others:

- an unabstracted interface to parallel matrix multiplication (traditional fat interface [7]) is insufficient, because it cannot easily capture multiple algorithms and heuristics without per-use overheads.

- one cannot standardize on a package that has no concept of poly-algorithms, and succeed in adhering to any single algorithm library, while attaining high performance over a range of problem environments (especially as space *or* time complexity may be more important to different applications).

- practical libraries must provide cost models and provide interface to help users select the right algorithm among several of the poly-algorithm, unless heuristics can be made quite tight and still be acceptably cheap; space/time tradeoffs must be acceptable too. This flexibility needs to include *data distribution independence*.

- this same type of study is clearly suggestive of studies for more complex parallel algorithms, where the computation to communication ratio is less favorable, and where it is even more likely that a variety of strategies will be needed.

So, performance requirements drive the use of abstracted (*e.g.*, object-based or object-oriented) formulations, which must also include, wherever possible performance models.

Acknowledgements

We express our gratitude to the reviewers of our initial paper. Our reviewers were quite careful in their review of our initial paper; we retrenched and strengthened this work because of their insistence that we demonstrate newness and relevance of matrix multiplication algorithms, as well as to compare with other known works. In fact, since our original submission, several new works (cited above) have continued to show the relevance for this earlier work, and the validity of presenting new algorithms, as well as a taxonomy for all the algorithms. We deferred most of the software technology discussing to other papers, but we have retained comments on the implications of our results on parallel libraries.

We acknowledge the Maui High Performance Computing Center where all the performance tests for the parallel dense matrix multiplication algorithms were conducted.

We also would like to thank the attendants of the BLAS Technical Workshop for their suggestions about the performance tests, when we presented our initial results at the Workshop in the University of Tennessee, November 14, 1995.

References

[1] M. Aboelaze, N. P. Chrisochoides, E. N. Houstis, and C. E. Houstis. The Parallelization of Level 2 and 3 BLAS Operations on Distributed Memory Machines. Technical Report CSD-TR-91-007, Purdue University, West Lafayette, IN, 1991.

[2] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A High-Performance Matrix-Multiplication Algorithm on a Distributed-Memory Parallel Computer Using Overlapped Communication. *IBM J. Res. Develop.*, 38(6):673–681, November 1994.

[3] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A 3-Dimensional Approach to Parallel Matrix Multiplication. *IBM J. Res. Develop.*, 39(5):1–8, September 1995.

[4] P. V. Bangalore. The Data-Distribution-Independent Approach to Scalable Parallel Libraries. Master's thesis, Mississippi State University, May 1995. Also available as ftp://cs.msstate.edu/pub/reports/bangalore_ms.ps.Z.

[5] L. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, Bozeman, MN, 1969.

[6] J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. *Concurrency: Practice and Experience*, 6:543–570, 1994.

[7] J. O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

[8] E. Dekel, D. Nassimi, and S. Sahni. Parallel Matrix and Graph Algorithms. *SIAM Journal on Computing*, 10(4):657–73, November 1981.

[9] J. W. Demmel, M. T. Heath, and H. A. van der Vorst. Parallel Numerical Linear Algebra. Technical Report CSD-92-703, U. C. Berkeley, Berkeley, CA, 1992. Computer Science Division.

[10] C. C. Douglas, M. Heroux, G. Slishman, and R. M. Smith. GEMMW: A Portable Level 3 BLAS Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm. *Journal of Computational Physics*, 110:1–10, 1994.

[11] R. D. Falgout, A. Skjellum, S. G. Smith, and C. H. Still. The Multicomputer Toolbox Approach to Concurrent BLAS and LACS. In J. Saltz, editor, *Proc. Scalable High Performance Computing Conf. (SHPCC)*, pages 121–128. IEEE Press, Los Alamitos, CA, 1992. Also available as LLNL Technical Report UCRL-JC-109775.

[12] R. D. Falgout, A. Skjellum, S. G. Smith, and C. H. Still. The Multicomputer Toolbox Approach to Concurrent BLAS. Technical Report MSU-CS-TR931001, Mississippi State University, Mississippi State, MS, October 1993. Department of Computer Science.

[13] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, Cambridge, Massachusetts, 1994.

[15] S. Huss-Lederman, E. M. Jacobson, and A. Tsao. Comparison of Scalable Parallel Matrix Multiplication Libraries. In Anthony Skjellum and Donna Reese, editors, *Proc. Scalable Parallel Libraries Conference*, pages 142–149. IEEE Computer Society Press, October 1993.

[16] S. Huss-Lederman, E. M. Jacobson, A. Tsao, and G. Zhang. Matrix Multiplication on the Intel Touchstone DELTA. *Concurrency: Practice and Experience*, 6:571–594, 1994.

[17] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Redwood City, California, 1994.

[18] J. Li, A. Skjellum, P. V. Bangalore, and P. J. Brennan. Poly-Algorithms for Remapping of Dense Arrays. Technical Report, Department of Computer Science, Mississippi State University, in preparation, 1996.

[19] K. K. Mathur and L. S. Johnsson. Multiplication of Matrices of Arbitrary Shape on a Data Parallel Computer. *Parallel Computing*, 20:919–951, 1994.

[20] A. Skjellum. *Concurrent Dynamic Simulation: Multicomputer Algorithms Research Applied to Ordinary Differential-Algebraic Process Systems in Chemical Engineering.* PhD thesis, Chemical Engineering, California Institute of Technology, May 1990.

[21] A. Skjellum and C. H. Baldwin. *The Multicomputer Toolbox:* Scalable Parallel Libraries for Large-Scale Concurrent Applications. Technical Report UCRL-JC-109251, Lawrence Livermore National Laboratory, December 1991.

[22] A. Skjellum and P. V. Bangalore. Driving Issues in Scalable Libraries: Poly-Algorithms, Data Distribution Independence, Redistribution, Local Storage Schemes. In David H. Bailey et al., editor, *Proc. of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 734–737. SIAM Press, Philadelphia, PA., February 1995.

[23] A. Skjellum, N. E. Doss, and P. V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna Reese, editors, *Proc. Scalable Parallel Libraries Conference*, pages 166–173. IEEE Computer Society Press, October 1993.

[24] V. Strassen. Gaussian Elimination is not Optimal. *Numer. Math.*, 13:354–356, 1969.

[25] R. A. van de Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. Technical Report TR-95-13, The University of Texas at Austin, Austin, TX, 1995. Department of Computer Science.

[26] E. F. van de Velde. Data Redistribution and Concurrency. *Parallel Computing*, 16:125–38, 1990.

[27] E. F. van de Velde and J. Lorenz. Adaptive Data Distribution for Concurrent Continuation. Technical Report CRPC-89-4, California Institute of Technology, 1989. Caltech/Rice Center for Research in Parallel Computation.