UNIVERSITÉ DU QUÉBEC

THÈSE PRÉSENTÉE À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DU DOCTORAT EN GÉNIE ÉLECTRIQUE

PAR
HUSSAM HUSSEIN ABU AZAB

ALGORITHME INNOVANT POUR LE TRAITEMENT PARALLÈLE BASÉ SUR L'INDÉPENDANCE DES TÂCHES ET LA DÉCOMPOSITION DES DONNÉES

JANVIER 2017

UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

DOCTORAT EN GÉNIE ÉLECTRIQUE (PH.D.)

Programme offert par l'Université du Québec à Trois-Rivières

ALGORITHME INNOVANT POUR LE TRAITEMENT PARALLÈLE BASÉ SUR L'INDÉPENDANCE DES TÂCHES ET LA DÉCOMPOSITION DES DONNÉES

PAR

HUSSAM HUSSEIN ABU AZAB

Adel Omar Dahmane, directeur de recherche	Université du Québec à Trois-Rivières
Sousso Kelouwani, président du jury	Université du Québec à Trois-Rivières
Habib Hamam, codirecteur de recherche	Université de Moncton
Ahmed Lakhssassi, évaluateur	Université du Québec en Outaouais
Rachid Beguenane, évaluateur externe	Collège militaire royal du Canada

Thèse soutenue le 15 décembre 2016

Abstract

In this thesis, a novel framework for parallel processing is introduced. The main aim is to consider the modern processors architecture and to reduce the communication time among the processors of the parallel environment.

Several parallel algorithms have been developed since more than four decades; all of it takes the same mode of data decomposing and parallel processing. These algorithms suffer from the same drawbacks at different levels, which could be summarized that these algorithms consume too much time in communication among processors because of high data dependencies, on the other hand, communication time increases gradually as number of processors increases, also, as number of blocks of the decomposed data increases; sometime, communication time exceeds computation time in case of huge data to be parallel processed, which is the case of parallel matrix multiplication. On the other hand, all previous algorithms do not utilize the advances in the modern processors architecture.

Matrices multiplication has been used as benchmark problem for all parallel algorithms since it is one of the most fundamental numerical problem in science and engineering; starting by daily database transactions, meteorological forecasts, oceanography, astrophysics, fluid mechanics, nuclear engineering, chemical engineering, robotics and artificial intelligence, detection of petroleum and minerals, geological detection, medical research and the military, communication and telecommunication, analyzing DNA material, Simulating earthquakes, data mining and image processing.

In this thesis, new parallel matrix multiplication algorithm has been developed under the novel framework which implies generating independent tasks among processors, to reduce the communication time among processors to zero and to utilize the modern processors architecture in term of the availability of the cache mem. The new algorithm utilized 97% of processing power in place, against maximum of 25% of processing power for previous algorithms.

On the hand, new data decomposition technique has been developed for the problem where generating independent tasks is impossible, like solving Laplace equation, to reduce the communication cost. The new decomposition technique utilized 55% of processing power in place, against maximum of 30% of processing power for 2 Dimensions decomposition technique.

Foreword

I dedicate this work first of all to my parents Hussein and Inaam, who partied the nights on my upbringing.

I dedicate this work to my life partner and so my PhD partner .... Rana

I foreword my work to the scientist of Math, to Al-Khwārizmī, Abū Ja'far Muhammad Ibn Mūsā, Pythagoras (Πυθαγόρας), Isaac Newton, Archimedes, René Descartes, and Alan Mathison Turing, father of computer science.

I foreword this work to my daughter Leen, and my sons Hussein, Yazan, and Muhammad.

I would acknowledge proudly my supervisor Prof. Adel Omar Dahmane, for his unlimited support through my PhD march, and I would thank all who supported me by all means.

Table of Contents

List of Tables

List of Figures

List of Symbols

f = number of arithmetic operations units

t_f = time per arithmetic operation << t_c (time for communication)

c = number of communication units

q = f / c average number of flops per communication access.

Speedup $S_p \ = \ q \ (t_f / t_c)$

nproc = number of processors

nblock = number of blocks

Chapter 1 -Introduction

The need for vast computing power in so many fields like forecasting the weather, analyzing DNA material, simulating earthquakes etc. has led for looking for parallel and distributed computing, in the light of the limited speed of the classic computers and processing power due to the physical constraints preventing frequency scaling. On the other hand, the physical limits been achieved at the hardware level in processors industry, which leads for opening the doors to design parallel processors in the mid of 1980's by introducing the parallel processing and networks of computers [1-8]. By the 1990's, the Single Instruction Multiple Data (SIMD) technology show up, and later multi-core platforms in the mainstream industry such as multi-core general purpose architectures (CPUs) and Graphics Processing Units (GPUs) where several cores working in parallel inside the processor chip [4, 9, 10- 12]. New processors show vast computing power [4, 6, 13-15]. Multi-core i7 CPU is the most updated parallel CPU produced by Intel at PC level; While latest NVidia Graphics Processing Unit has 1536 core at its VGA card Tesla K10, which has two GPUs, which implies 1536 × 2 = 3,072 processors running in parallel [16], which is needed for applications of seismic, image, signal processing, video analytics. This implies that the future software development must support multi-core processors, which is parallel processing.

Parallel processing has opened an era to have super computing power at the cost of several PCs. So, connecting several PCs into a grid could be utilized as one single supercomputer by the help of certain algorithm to manage the distributing of the load among the active connected PCs. Well, parallel processing did not stop here, but parallel processing extends to include computers and super computers connected over internet, where the load could be distributed over connected super computers to have enormous parallel processing power.

All parallel algorithms until the moment depended in decomposing the data of a problem into blocks and perform the functionality on it, in parallel mode taking in consideration the data and the functional dependency among the data.

Data Decomposition in general having two modes, one dimension (1D), where the data will be set of strips, where each processor will process one single strip at a time; and the other mode is two dimensions decomposing (2D), where the data will be set of blocks, and each processor will process single block at a time.

In addition, existing parallel algorithms did not address and utilize the parallel capabilities of the new processors, like multicore and other enhancement like cache memory and wide address bus of 64 bit. All parallel algorithms have kept looking at the processor on the old architecture design. For example, they are considering the grain applications, where the application will be divided into the most simple functions, so each processor will process these simple functions and getting the next afterword; while all these algorithms are not looking at high capabilities of the new processors, where it can process more than a function at a time, dual core processors processes two functions a time, while Intel Xeon Phi processor has 61 cores, which implies it processes 61 functions at a time. Existing parallel algorithms sends one single function to each processor regardless number of cores it has. In addition, the new processors architecture has different levels of cache memory which allows for more data upload capacity to reduce the access time of RAM, so cache memory leads to faster processing for complex functions which need huge chunk of data; Existing parallel algorithms do not consider the availability of cache memory and keep send simple math functions to each processor, which is a bad exploitation of modern architecture processors.

In this thesis, new technique for parallel processing will be introduced, which will overcome the drawbacks of the previous algorithms. The technique is called ITPMMA algorithm, it depends on dividing the problem into set of tasks, which implies the data will not be decomposed in ITPMMA algorithm, and instead the problem will be decomposed into independent sets of operations, where each processor will execute independent operations and will upload what data it needs. In addition, ITPMMA algorithm will utilize the capabilities of the parallel and multi-core processors, which is absent in the existing parallel algorithms.

1.1. Motivation

Numerical problems consume a lot of processing resources which leads to utilize the parallel processing architecture. Since 1969, parallel algorithms start showing up, [17-21]. Parallel Matrix Multiplication algorithms were one of the earliest parallel processing algorithms that appeared since then. So, since 1969 for homogenous clusters like Systolic algorithm [22], Cannon's algorithm [23], Fox and Otto's algorithm [24], PUMMA (Parallel Universal Matrix Multiplication) [25], SUMMA (Scalable Universal Matrix Multiplication) [26] and DIMMA (Distribution Independent Matrix Multiplication) [27]. All these algorithms had been designed for distributed memory platforms, and most of them use the popular ScaLAPACK library [28, 29], which includes a highly-tuned, very efficient routine targeted to two-dimensional processor grids.

PUMMA algorithm maximizes the reuse of the data that have been hold in the upper levels of the memory hierarchy (registers, cache, and /or local memory) [16]. PUMMA, which had been developed in 1994, did not address the time consumption by exchanging intermediate results between processors.

SUMMA algorithm, been developed in 1997. It has introduced the pipelining in PUMMA to maximize reuse of data. In addition, SUMMA reformulated the blocking method in terms of matrix-matrix multiplications instead of matrix-vector multiplications, which reduced the communication overhead [26]. In general, SUMMA did not address the time consumption of the communication between processors. In addition, SUMMA did not address the cache memory of the modern processors.

On the same year, 1997, Choi [27] has developed DIMMA algorithm. "The algorithm introduced two new ideas: modified pipelined communication scheme to overlap computation and communication effectively; and to exploit the least common multiple (LCM) block concept to obtain the maximum performance of the sequential BLAS – Basic Linear Algebra Subprograms – routine in each processor" [27]. But still, DIMMA did not address the huge time consumed on communication between processors to exchange the intermediate results.

In 2005, NGUYEN et al. [29] combined the use of Fast Multipole Method (FMM) algorithms and the parallel matrix multiplication algorithms, which gave remarkable results. Nevertheless, the algorithm still suffers data dependency and high communication cost among the processors. Moreover, the algorithm does not address heterogeneous environments.

In 2006, Pedram et al. [30], have developed high-performance parallel hardware engine for matrix power, matrix multiplication, and matrix inversion, based on distributed memory. They have used Block-Striped Decomposition (BSD) algorithm directly to implement the algorithm. There was obvious drawback related to processors' speed up efficiency. The algorithm reduces memory bandwidth by taking advantage of reuse data, which results in an increase in data dependencies.

On 2008 James Demmel developed a new algorithm to minimize the gap between computation and communication speed, which continues to widen [31]. The performance of sparse iterative solvers was the aim of this algorithm, where it produced speedup of over three times of serial algorithm. In fact, the increasing gap between computation and communication speed, is one of the main points to be addressed by reducing the communication between processors as much as possible. The algorithm still suffers data dependency and communication; especially for large matrices sizes.

In 2008 Cai and Wei [32] developed new matrix mapping scheme to multiply two vectors, a vector and a matrix, and two matrices which can only be applied to optical transpose interconnection system (OTIS-Mesh), not to general OTIS architecture, to reduce communication time. They have achieved some improvements compared to Cannon algorithm, but it was expensive in term of hardware cost. In addition, the algorithm did not add any new value in term of algorithm design.

In 2009, Sotiropoulos and Papaefstathiou implement BSD algorithm using FPGA device [33]. There was no achievement in terms of reducing data dependencies and communication cost.

In 2012, Nathalie Revol and Philippe Théveny developed new algorithm, called "Parallel Implementation of Interval Matrix Multiplication" to address the implementation of the product of two dense matrices on multicore architectures [34]. The algorithm produced accurate results but it fails to utilize the new features of the multicore architectures processors, as it has targeted in advance.

In 2013 Jian-Hua Zheng [35] proposed new technique based in data reuse. It suffers from a lot of data dependency and high communication cost.

In 2014, another decomposition technique called Square-Corner instead of Block Rectangle partition shapes to reduce the communication time has been proposed in [36]. The research was limited to only three heterogeneous processors. For some cases, they have reported less communication time and therefore showed a performance improvement.

Also, in 2014 Khalid Hasanov [37] introduced hierarchy communication scheme to reduce the communication cost to SUMMA algorithm. Although achieved some better performance, pre ITPMMA algorithm drawbacks like data dependency and communication cost are still there. Moreover, this algorithm is for homogenous environment.

Other algorithms have been designed later on to enhance the process of matrix multiplication and to reduce the processing time. In 2014, Tania Malik et al. [38] proposed new network topology to decrease communication time among the processors. The algorithm suffers from more data dependencies between the processors. The major drawback of all previous parallel algorithms developed until now need homogenous processor architecture, and never addressed heterogeneous processors, except NGUYEN et al. [29], which conclude very negative results, so, by executing these algorithms in a grid of several PCs – heterogeneous environment – would have a lot of incompatible latency factors.

Having homogenous environment, all parallel algorithms, either for parallel matrix multiplication as we will see in this thesis, or for any numerical problem else, all existing algorithms suffer from classic drawbacks, like:

1. The optimal size of the block of the decomposed matrices.
2. The communication time of the exchanged messages among the processors, which is proportional to number of processors and number of the blocks of the decomposed matrices.
3. Data and functional dependency between the processors.
4. Poor load balance especially with non-square matrices.

1.2. Originality

In this thesis, a new novel framework for parallel processing has been developed to add the following new values:

1. Reducing the communication time among the parallel processors to ZERO.
2. No processor becomes idle or in hold, waiting other processors output until the parallel operations over.
3. Eliminating the need for a certain topology of processors.

1.3. Objectives

Thesis objectives to develop a novel framework for parallel processing which implies reconstruct the parallel problem into independent tasks to:

1. Reduce the idle time of the processors to increase the efficiency, and this to be achieved by:
    a. Proper load balance among the processors.
    b. Utilizing the modern processor architecture capabilities in term of multicore and cache memories of different levels.
2. Reduce the communication time among the processors, by eliminating the data dependencies by reconstructing the parallel problem into set of independent tasks, to reduce the communication time among processors to zero.

On the other hand, for numerical problems where reconstructing the parallel problem into independent tasks is not within hands, like solving Laplace equation in parallel, a new data decomposing technique is developed, to reduce the communication time among the processors, and reduce idle time of the processors, to increase the efficiency.

1.4. Methodology

To satisfy the objectives of this thesis, I am to follow the procedure below:

1. Review literature of parallel matrix multiplication and data decomposition techniques.
2. Study in details different parallel algorithms and data decomposition techniques to find out its drawbacks.

3. Develop common diagram to setup new framework for parallel processing to address the drawbacks of the previous algorithms.
4. Develop new parallel matrix multiplication algorithm and new data decomposition technique within the new framework.
5. Test the new framework on small environment of 4 and 16 processors.
6. Test the new framework on the advanced supercomputer CLUMEQ.
7. Compare the results of the performance of the new framework with benchmark algorithms like Cannon algorithm and Fox algorithm.

1.5. Thesis Organization

This thesis is organized into six chapters. Chapter I is an introduction for the thesis to show the motivation, originality, objectives of the research and the methodology.

Chapter II, titled "Matrix Multiplication", I will discuss the common Parallel Matrix Multiplication Algorithms in terms of performance, which includes speed up, time complexity, load balancing, data dependencies.

Chapter III, titled "ITPMMA algorithm for Parallel Matrix Multiplication (STMMA)", presents the proposed algorithm followed by the results and comparisons with parallel matrix multiplication algorithms.

Chapter IV, titled "Analysis and Results", presents the experimental results which showed that the ITPMMA algorithm achieved significant runtime performance on CLUMEQ supercomputer and also a considerable performance compared by other algorithms like

Cannon Algorithm and Fox Algorithm. In these experiments I have used up to 128 processors in parallel, and about 40000 matrices size.

Chapter V, titled "The clustered 1 Dimension decomposition technique" presents the new data decomposition technique, developing parallel solution for Laplace equation using Gauss-Seidel iterative method, and test the results and compare it with same of 1 dimension and 2 dimensions data decomposition techniques.

Finally, Chapter VI presents the conclusion and explores the opportunities for the future that build upon the work presented in this dissertation.

Chapter 2 -Matrix Multiplication

2.1.Introduction

In this chapter I will address the matrix multiplication problem in serial and parallel mode. This problem will be used to describe the efficiency of the parallel algorithms; in fact, matrix multiplication has been used as benchmark problem for parallel processing algorithms for several reasons:

1. It is easily scaled problem for a wide range of performance; its size grows like N^3 for matrices of order N.
2. It has two nested loops plus the outer loop, total of three loops; the most inner loop consists at least a single multiply and add operation. Where the loops can be parallelized or to achieve high performance.
3. Computation independency, where the calculation of each element in the result matrix is independent of all the other elements.
4. Data independence, where the number and type of operations to be carried out are independent of the data type of the multiplied matrices.
5. Different data types like short and long integers and short and long floating-point precision can be used, which promises for different levels of tests. Also it allows for utilizing of the capabilities of the new processors, which has high cache memories, by

multiplying long type matrices, which needs high size of cache memory for intermediate results.

6. Number of floating point operations (flops) easily is calculated.
7. Matrix multiplication is used as bench mark problem to test the performance of so many processors, like Intel, as shown in the figure 2-1, below.

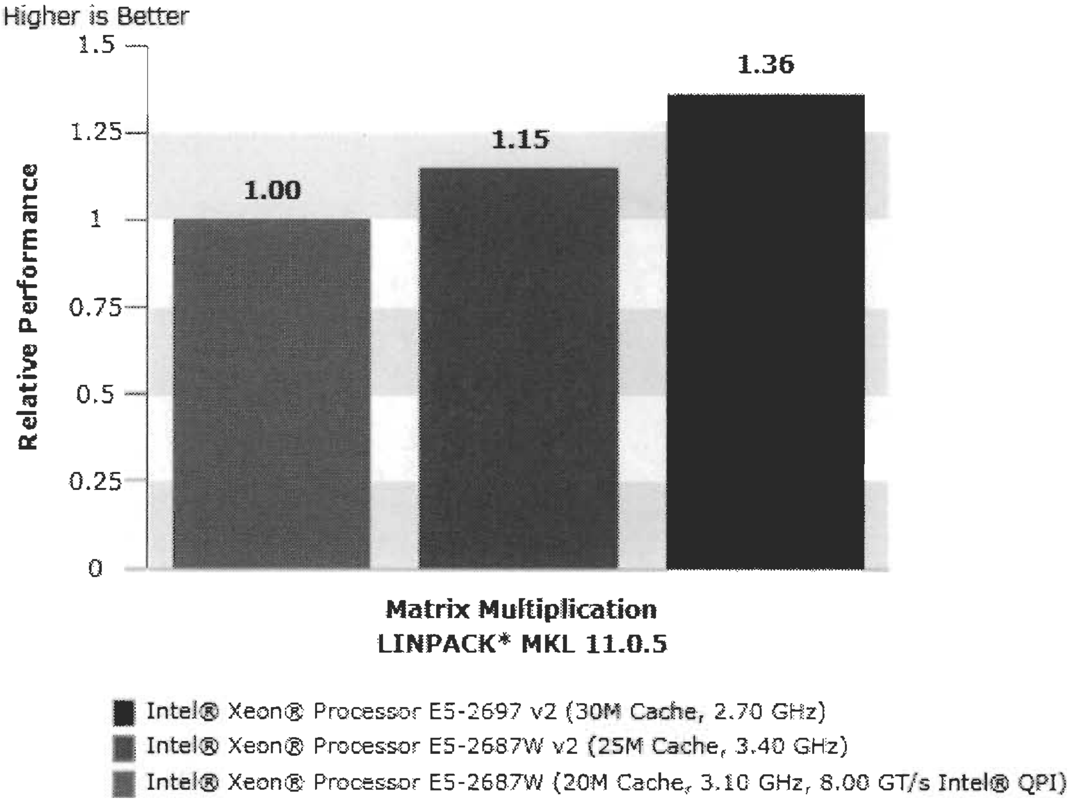


Figure 2-1 Matrix Multiplication problem as benchmark problem for Intel processor performance [39].

In this chapter, some important aspects of matrix multiplication will be addressed, together with the outstanding characteristics of the already developed parallelization algorithms, like cannon and fox algorithms.

2.2. Matrix Multiplication Definition

It is defined between two matrices only if the number of columns of the first matrix is the same as the number of rows of the second matrix. If A is an i-by-k matrix and B is a k-by-j matrix, then their product AB is an i-by-j matrix, is denoted by C_{ij}=A_{ik} × B_{kj}, which be given by

$$C_{ij} = \sum\nolimits_{k=1}^{n} A_{i,k} \times B_{k,j} \qquad \text{Eq. 21}$$

And it is calculated like

$$\begin{pmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ C_{31} & C_{32} & C_{33} & C_{34} \\ C_{41} & C_{42} & C_{43} & C_{14} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{41} & A_{44} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{pmatrix} \text{Eq. 2-2}$$

So,

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} + A_{13} \times B_{31} + A_{14} \times B_{41} \qquad \text{Eq. 2-3}$$

Which is different than BA, which is denoted by D_{ij}= B_{kj} × A_{ik}, given by

$$D_{ij} \quad = \sum\nolimits_{k=1}^{n} B_{k,j} \times A_{i,k} \qquad \text{Eq. 2-4}$$

So,

$$D_{11} = B_{11} \ \times A_{11} \times +B_{12} \ \times A_{21} + B_{13} \times A_{31} + B_{14} \times A_{41} \qquad \text{Eq. 2-5}$$

Which implies matrix multiplication is not commutative; that is, AB is not equal to BA. The complexity of matrix multiplication, if carried out naively, is $O(N^3)$, where N is the size of product matrix C. So for the Matrix multiplication of 4×4 size, the complexity is $O(N^3) = O(4^3)$= 64 operations, or the timing of 64 operations.

In 1969, Volker Strassen has developed Strassen's algorithm, has used mapping of bilinear combinations to reduce complexity to $O(n^{\log 2(7)})$ (approximately $O(n^{2.807\ldots})$). The algorithm is limited to square matrix multiplication which is considered as a main drawback.

In 1990, another matrix multiplication algorithm developed by Don Coppersmith and S. Winograd. The algorithm has complexity of $O(n^{2.3755})$. [40]

In 2010, Andrew Stothers gave an improvement to the algorithm, $O(n^{2.3736})$ [41]. In 2011, Virginia Williams combined a mathematical short-cut from Stothers' paper with her own insights and automated optimization on computers, improving the bound to $O(n^{2.3727})$ [42].

2.3. Serial Matrix Multiplication Algorithm

2.3.1. Serial Algorithm

The serial algorithm for multiplying two matrices is taking the form:

```
for (i = 0; i < n; i++)
     for (j = 0; i < n; j++)
          c[i][j] = 0;
          for (k = 0; k < n; k++)
               c[i][j] += a[i][k] * b[k][j]
          end for
     end for
end for
```

with the complexity $O(N^3) = O(4^3)$= 64 operations, or the timing of 64 operations.

2.3.2. Strassen's algorithm

In 1969, Professor V. Strassen [36] developed new algorithm known by his name, Strassen's algorithm, where the complexity of his algorithm is $O(N^{log7/log2}) = O(N^{Log_2 7})$ $= O(N^{\ln 7})$, which will be equal to $O(4^{Log_2 7}) = 14.84$, which is less than 64, i.e. to multiply two 4×4 matrices using Strassen's algorithm, the algorithm needs the time of 14.84 operations, rather than the time of 64 operations using the standard algorithm .

To simplify Strassen's algorithm, I will implement it on the following matrices multiplications:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

So, we run the below 7 quantities:

$P_1 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

$P_2 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$

$P_3 = (A_{11} - A_{21}) \times (B_{11} + B_{12})$

$P_4 = (A_{11} + A_{12}) \times B_{22}$

$P_5 = A_{11} \times (B_{12} - B_{22})$

$P_6 = A_{22} \times (B_{21} - B_{11})$

$P_7 = (A_{21} + A_{22}) \times B_{11}$

And produce the product matrix C, we run the below summations

$C_{11} = P_1 + P_2 - P_4 + P_6$

$C_{12} = P_4 + P_5$

$C_{21} = P_6 + P_7$

$C_{22} = P_2 - P_3 + P_5 - P_7$

The main drawback of Strassen's algorithm is the size product matrix should be product of 2, i.e. 2^1, 2^2, 2^3, 2^4, 2^5, …, so for matrices multiplication of size 5, where $2^2 < 5 < 2^3$, we need to pad the matrices by zeros, till we have 8×8 matrices and multiply them. Recent studies study the arbitrary size of the multiplied matrices by Strassen algorithm in more details [43] but without getting better performance. So for Strassen's algorithm for matrices multiplication of size 3×3, we have 7 multiplies and 18 adds. The complexity of the algorithm for matrices multiplication of size n×n can be computed as 7*T(n/2) + 18*(n/2)2 $= O(N^{Log_2 7}) = O(8^{Log_2 7}) = O(2^{3^{log_2 7}}) = O(2^{log_2 7})^3 = 7^3$; while if we use standard matrices multiplication, the complexity is $O(5^3) = 5^3 = 125$ which is less than $7^3 = 343$, which is 2.744 times the complexity of Strassen's algorithm. And the difference become so huge when we have big matrix size, let us say a matrices multiplication of size 127×127, and 127 is not multiplicand of 2, the nearest multiple of 2 is 128, so, 128×128 matrices multiplication, we do have complexity of $O(128^{Log_2 7}) = O(2^{7^{log_2 7}}) = O(2^{log_2 7})^7 = 7^7 =$ 823,543; while if we go for the standard matrices multiplication where the complexity is $O(N^3) = O(127^3) = 2{,}048{,}383$; which is 2.49 times the complexity of Strassen's algorithm, so Strassen's algorithm has positive effect. While if we consider matrices multiplication of size 70×70, again, we need to pad it with zeros till we reach 128×128 matrices multiplication, where the complexity of standard multiplication is $O(N^3) = O(70^3) =$ 343,000, so, the complexity of Strassen's algorithm is 2.4 times the complexity of standard multiplication, so Strassen's algorithm has negative effect.

It is clear that the evaluations of intermediate values P_1, P_2, P_3, P_4, P_5, P_6, and P_7 are independent and hence, can be computed in parallel.

Another drawback of Strassen's algorithm is the communication between the processors [44, 45, and 46]. Some other researches [47] focus on optimizing the communication between processors at the execution of Strassen's algorithm, where they could obtain some success at different ranges according to the size of the matrices and number of processors; but they could not eliminate the communication among the processors to zero.

On the other hand, many researches have tried to extract parallelism from Strassen's algorithm, and standard matrices multiplication algorithm [48], and many researches have tried to extract the algorithm on multi-core CPUs [49, 50, and 51] to exploit more performance. In 2007, Paolo D'Alberto, Alexandru Nicolau [52] have tried to exploit Strassen's full potential across different new processors' architectures, and they could achieve some success for some cases, but still, they have to work with homogenous processors. While in 2009, Paolo D'Alberto, Alexandru Nicolau have developed adaptive recursive Strassen-Winograd's matrix multiplication (MM) that uses automatically tuned linear algebra software [53], to achieve up to 22% execution-time reduction for a single core system and up to 19% for a two dual-core processor system.

2.4. Parallel Matrix Multiplication Algorithms

Parallel algorithms of matrices multiplication are parallelization of the standard matrices multiplication method.

2.4.1. Systolic Algorithm

One of the old parallel algorithms returns to 1970, but still active algorithm till moments [54]. It is limited to square matrix multiplication only. In this algorithm, matrices A, B are decomposed into submatrices of size $\sqrt{P} \times \sqrt{P}$ each, where P is number of processors. The basic idea of this algorithm is the data exchange and communication occurs between the nearest-neighbors.

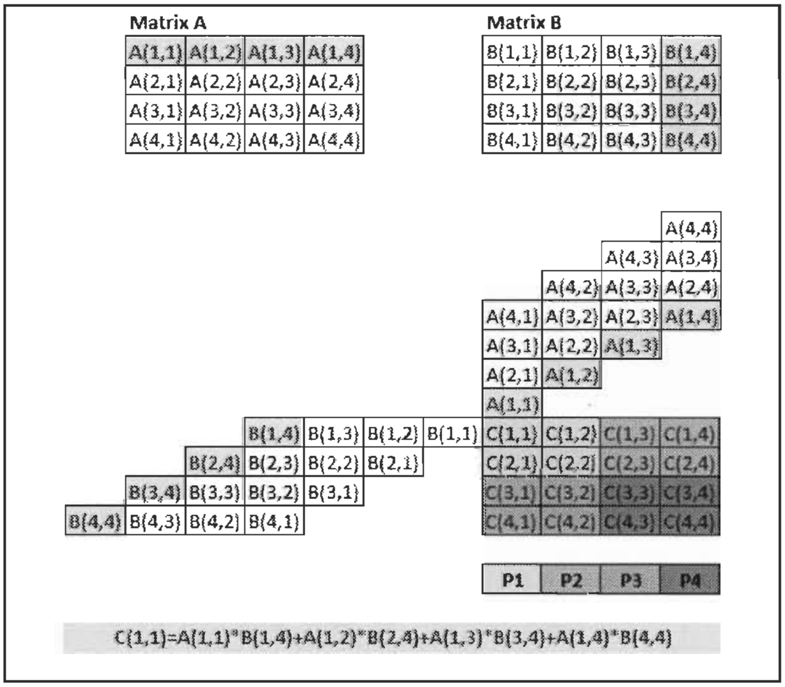

Matrix A
A{1,1} A{1,2} A{1,3} A{1,4}
A{2,1} A{2,2} A{2,3} A{2,4}
A{3,1} A{3,2} A{3,3} A{3,4}
A{4,1} A{4,2} A{4,3} A{4,4}
Matrix B
B{1,1} B{1,2} B{1,3} B{1,4}
B{2,1} B{2,2} B{2,3} B{2,4}
B{3,1} B{3,2} B{3,3} B{3,4}
B{4,1} B{4,2} B{4,3} B{4,4}
A{4,4}
A{4,3} A{3,4}
A{4,2} A{3,3} A{2,4}
A{4,1} A{3,2} A{2,3} A{1,4}
A{3,1} A{2,2} A{1,3}
A{2,1} A{1,2}
A{1,1}
B{1,4} B{1,3} B{1,2} B{1,1} C{1,1} C{1,2} C{1,3} C{1,4}
B{2,4} B{2,3} B{2,2} B{2,1} C{2,1} C{2,2} C{2,3} C{2,4}
B{3,4} B{3,3} B{3,2} B{3,1} C{3,1} C{3,2} C{3,3} C{3,4}
B{4,4} B{4,3} B{4,2} B{4,1} C{4,1} C{4,2} C{4,3} C{4,4}
P1 P2 P3 P4
C{1,1}=A{1,1}*B{1,4}+A{1,2}*B{2,4}+A{1,3}*B{3,4}+A{1,4}*B{4,4}

Figure 2-2 Systolic Algorithm

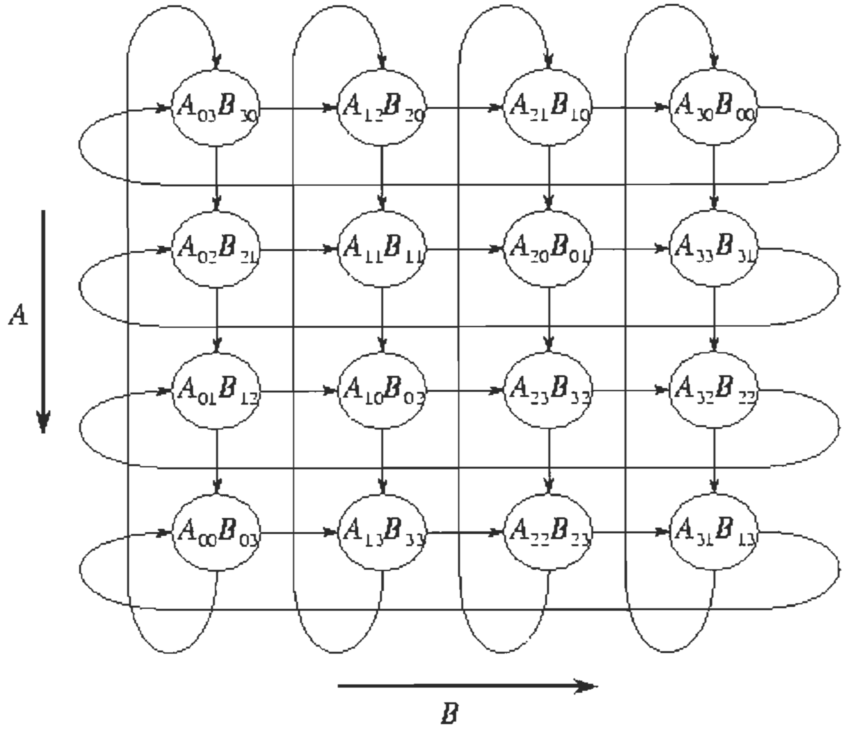


Figure 2-3 Layout of the A and B matrices in the systolic matrix-matrix multiplication algorithm for A4x4×B4x4 task mesh. The arrows show the direction of data movement during execution of the systolic algorithm. [55]

Table 2-1 The performance of the algorithm being studied by [33]

Task	Execution time
Transpose B matrix	$2n^2 t_f$
Send A, B matrices to the processors	$2m^2p\ t_c$
Multiply the elements of A and B	$m^2n\ t_f$
Switch processors' B sub-matrix	$n^2\ t_c$
Generate the resulting matrix	$n^2\ t_f + n^2\ t_c$
Total execution time	$\mathbf{t_f(m^2\ n + 3n^2\ ) + t_c(4n^2\ )}$

2.4.2. Cannon Algorithm

It is a memory efficient if the multiplied matrices are square. The blocks of matrix A to rotate vertically while matrix B blocks to rotate horizontally, this can be handled using circular shifts to generate the product matrix C. In fact, it replaces the traditional loop

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} \times B_{k,j}$$

With the loop

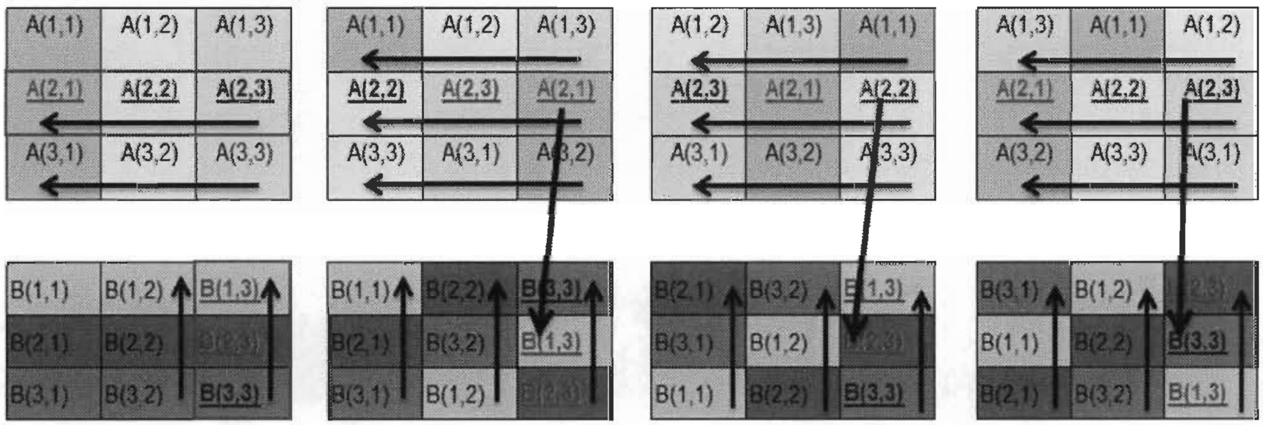
$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k)mod\sqrt{p}} \times B_{(i+j+k)mod\sqrt{p},j}$$

The Pseudo-code for the Cannon Matrix multiplication algorithm

```
% p number of processors
% s size of the matrix
For i = 0 to p - 1
A(i, j)=A(i,(i+j) mod p %left-circular-shift row i of A by i shifts
(skew of A)

For j = 0 to p - 1
B(i, j)=B((i+j) mod p, j) %up-circular-shift row j of B by j shifts (skew of B)

For (i = 0 to p-1) and (j = 0 to p-1)
C(i,j) = C(i,j) + $\sum_{k=1}^{p} A(i,k) * B(k,j)$
A(i, j)=A(I,(i-j) mod p              % left-circular-shift each row of A by 1
B(i, j)=B((i-j) mod p, j)       % up-circular-shift each column of B by 1
```



A, B initially   A, B after skewing   A, B after shift k = 1   A, B after shift k = 2

C(2,3) = A(2,1)*B(1,3) + A(2,2)*B(2,3) + A(2,3)*B(3,3)

Figure 2-4 Cannon's algorithm layout for n = 3

```
// Skew A & N
for i = 0 to s-1                                   // $s = sqrt(p)$
left-circular-shift row i of A by i    // cost = $s*(\alpha + n2/p/\beta)$
for i = 0 to s-1
up-circular-shift column i of B by i          // cost = $s*(\alpha + n2/p/\beta)$

// Multiply and shift
for k = 0 to s-1
local-multiply                                     // cost = $2*(n/s)3 = 2*n3/p3/2$
left-circular-shift each row of A by 1        // cost = $\alpha + n2/p/\beta$
up-circular-shift each column of B by 1       // cost = $\alpha + n2/p/\beta$
```

- Total Time = $2*n3/p + 4* s*\alpha + 4*\beta*n2/s$

- Parallel Efficiency = $2*n3 / (p * \text{Total Time})$

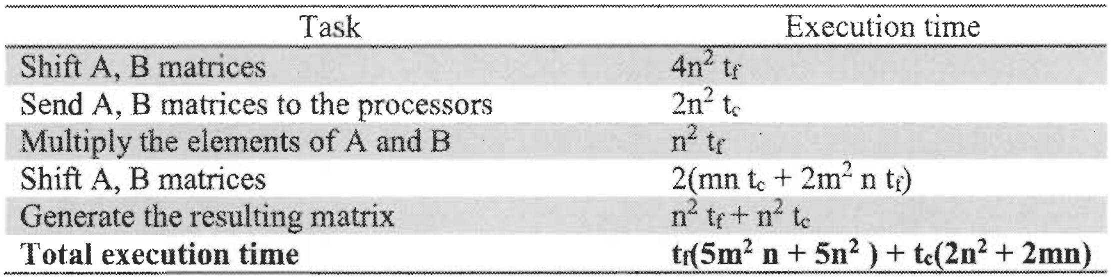
  $= 1/(1 + \alpha * 2*(s/n)3 + \beta * 2*(s/n))$

  $= 1/(1 + O(sqrt(p)/n))$

- Grows to 1 as $n/s = n/sqrt(p) = sqrt(\text{data per processor})$ grows

- Better than 1D layout, which had Efficiency = $1/(1 + O(p/n))$

Table 2-2 The performance of the algorithm being studied by [56]



Task	Execution time
Shift A, B matrices	$4n^2 t_f$
Send A, B matrices to the processors	$2n^2 t_c$
Multiply the elements of A and B	$n^2 t_f$
Shift A, B matrices	$2(mn\ t_c + 2m^2 n\ t_f)$
Generate the resulting matrix	$n^2 t_f + n^2 t_c$
Total execution time	$\mathbf{t_f(5m^2 n + 5n^2) + t_c(2n^2 + 2mn)}$

2.4.3. Fox and Otto's algorithm

Fox's algorithm for multiplication organize C =A x B into submatrix on a P processors. The algorithm runs P times, in each turn, it broadcasts corresponding submatrix of A on each row of the processes, run local computation and then shift array B for the next turn computation. The main disadvantages, it is applied only for square matrices.

This algorithm being written in general in HPJava, we still use Adlib.remap to broadcast submatrix, matmul is a subroutine for local matrix multiplication. Adlib.shift is used to shift array$B$, and Adlib.copy copies data back after shift, it can also be implemented as nested over and for loops.

```
  Group p=new Procs2(P,P);

  Range x=p.dim(0);
  Range y=p.dim(1);

  on(p) {
    //input a, b here;
      float [[#,#, , ]] a = new float [[x,y,B,B]];
      float [[#,#, , ]] b = new float [[x,y,B,B]];

      float [[#,#, , ]] c = new float [[x,y,B,B]];
      float [[#,#, , ]] temp = new float [[x,y,B,B]];

      for (int k=0; k<p; k++) {
             over(Location i=x|:) {
                   float [[,]] sub = new float [[B,B]];

                   //Broadcast submatrix in 'a' ...
                   Adlib.remap(sub, a[[i, (i+k)%P, z, z]]);

                   over(Location j=y|:) {
                   //Local matrix multiplication ...
                    matmul(c[[i, j, z, z]], sub, b[[i, j, z,
z]]);
                                      }
                                }

      //Cyclic shift 'b' in 'y' dimension ...
      Adlib.shift(tmp, b, 1, 0, 0); // dst, src, shift, dim,
mode;
      Adlib.copy(b, tmp);
                          }
  }
```

“Two efforts to implement Fox's algorithm on general 2-D grids have been made: Choi, Dongarra and Walker developed `PUMMA' [50] for block cyclic data decompositions, and Huss-Lederman, Jacobson, Tsao and Zhang developed `BiMMeR' for the virtual 2-D torus wrap data layout”[57].

2.5.Conclusion

In this chapter I have addressed the use of matrix multiplication as benchmark and the definition of matrix multiplication, serial algorithms and parallel algorithms.

Parallel algorithms for carrying out matrix multiplication in different architecture since 1969 had been studied and analyzed. Parallel algorithms for matrix multiplication have common mode, it is subdividing the matrices into small size matrices and distributing them among the processors to achieve faster running time.

We found out that although so many parallel matrix algorithms have been developed since Cannon Algorithm and Fox algorithm four decades ago, all these algorithms – described in Appendix B – still use the same methodology and framework of Cannon and Fox algorithms in term of data decomposition and communication among the processors. On addition, all these algorithms could not achieve a distinguished performance against both Cannon and Fox algorithms, which keep both algorithms as bench mark algorithms in parallel matrix multiplication.

Finally, Identification of BLAS, and the development till considering multi-core architecture had been shown in Appendix A.

Chapter 3 - ITPMMA algorithm for Parallel Matrix Multiplication (STMMA)

3.1.Introduction

Several parallel matrix multiplication algorithms had been described and analyzed in the previous chapter. All parallel matrix multiplication algorithms based on decomposing the multiplied matrices into smaller size blocks of data, the blocks will be mapped and distributed among the processors, so each processor run matrix multiplication on the assigned blocks; this will reduce the whole time of the matrix multiplication operation to less than the time needed to complete this operation by utilizing one processor. All existing parallel matrix multiplications algorithms suffer from four drawbacks:

1. To define the optimal size of the block of the decomposed matrices, so the whole operation can be produced by the minimum time. For example, multiplying matrices of size 64×64 over 4 processors, should the block size be 4×4 or 8×8 or 16×16.

2. The number of exchanged messages between the processors are highly time consuming; it is proportional to the number of the processors. On the other hand, the time of forwarding the messages relays partly on the network structure.

3. Data dependency between the processors, where some processors will stay idle waiting an intermediate calculation results from other processors. Data dependency increases as number of blocks/processors increases.

4. Some algorithms suffer from another drawback, which is the load balance, especially with non-square matrix multiplications.

In this chapter, new parallel matrix multiplication algorithm will be introduced, to overcome the above four drawbacks, which returns in vast difference in the performance in terms of processing time and load balance. For example, for a matrices multiplication of 5000×5000, it consumes 2812 seconds using cannon algorithm, while only 712 seconds needed using the new algorithm, which implies 4 times faster. I have implemented the algorithm initially using Microsoft C++ ver. 6, with MPI Library. I have execute it at Processors Intel(R) Core(TM) i5 CPU 760 @2.80GHz 2.79 GHz, the installed memory (RAM) was 4.00 GB, System type: 64-bit Operating System, Windows 7 Professional. Later I have implemented the algorithm using CLUMEQ supercomputer, where I have used up to 128 processors in parallel.

I will reference to the new algorithm by the name ITPMMA algorithm. The basic concepts of ITPMMA algorithm were published on "Sub Tasks Matrix Multiplication Algorithm (STMMA)", [59].

3.2.ITPMMA algorithm for Parallel Matrix Multiplication (STMMA)

Unlike previous algorithms, ITPMMA Algorithm for Parallel Matrix Multiplication does not define new data movement or circulation; instead, it generates independent serial tasks follows the serial matrix multiplications shown below:

```
for|| I=0 to s-1 {
 for J=0 to s-1 {
      for K=0 to s-1 {
       $CIJ = CIJ + AIK \times BKJ$ }}}
```

ITPMMA Algorithm decomposes the parallel matrix multiplication problem into several independent vector multiplication tasks, keeping the processing details of each independent task to the processors, which are multicore processors, so I am utilizing the modern architecture processors' capabilites. ITPMMA Algorithm has the advantage to utilize the up-to-date processor architecture features in terms of multicores and cache memories. In fact, ITPMMA Algorithm defines each independent task as set of instructions to produce one element of the result matrix. Each single set is fully independent of production of any other set, or independent task. Each single independent task is being processed by one single processor. Once each processor has completed the independent task been assigned to it, and produced an element of the result matrix, it processes the next independent task, to produce the next element, and so on. ITPMMA Algorithm implies:

1. Zero data dependences so better processors' utilization since no processor stays on hold, waits for other processors' output,
2. Zero data transfer among the processors of the cluster, so faster processing. Communications in ITPMMA Algorithm is limited to the time needed to send the independent tasks lists by the server node processor to different processors, and to the time needed to receive back alerts and results by the other processors to the server node processor.

In this context ITPMMA Algorithm has advantages of efficient use of the processors' time. Figure 3-1 simulates the serial matrix multiplication using one single core processor. Figure 3-2 simulates the previous parallel algorithms. It is clear that both processors P2 and P3 are on hold, till processor P1 over and switch to idle status. On addition, processors P4 will not start processing till both P2 and P3 over. The efficiency of using the processors' time is so poor. Each processor computes for 4 units of time and 3 units for communication, and been on hold or idle for 9 units of time. So the computing efficiency of each processor is 4/16 =25%. On the other hand, figure 3-3 simulates parallel multiplication using new algorithm which is called Independent Tasks Parallel Matrix Multiplication Algorithm (ITPMMA). Instead of decompose the data among the processors; we distribute the independent tasks among the processors. So, processor P1 should produce the first row of Matrix C shown in figure 2, while processor P2 should produce the second row of Matrix C, and processor P3 should produce the third row, finally processor P4 should produce the fourth row of Matrix C. For that, we need only 4 units of time to accomplish the tasks, in addition, no communication among the processors, also, no time for assembling the results, only to deliver it. One of the major advances of ITPMMA algorithm is efficiency of using the processors, no processor gets on hold or idle. The efficiency of each processor is 4/5 = 90%.

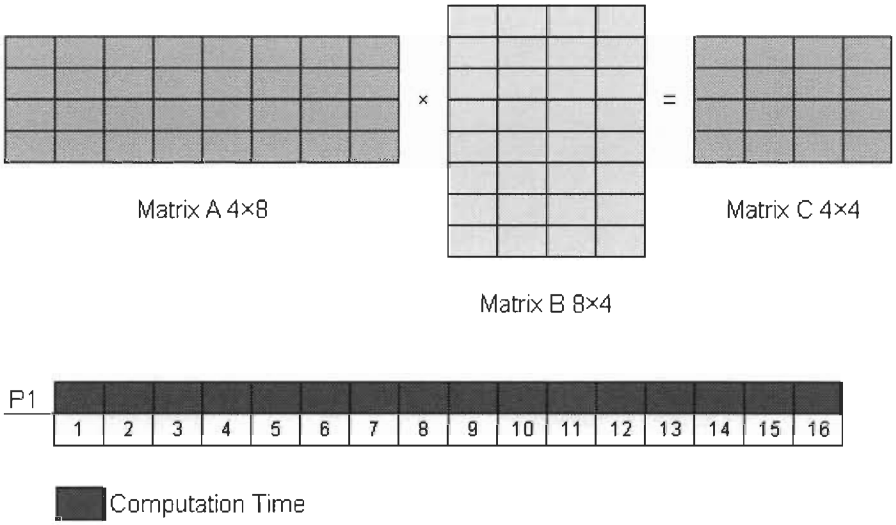


Figure 3-1, Serial Matrix Multiplication

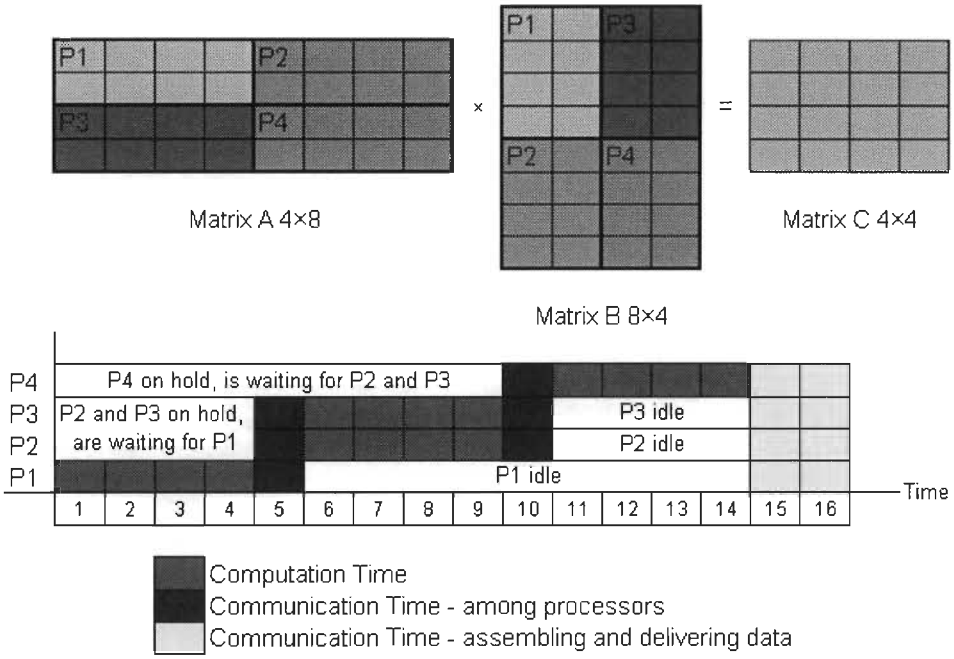

P1
P2
P3
P4
Matrix A 4×8
×
P1
P3
P2
P4
Matrix B 8×4
=
Matrix C 4×4
P4
P3
P2
P1
P4 on hold, is waiting for P2 and P3
P2 and P3 on hold, are waiting for P1
P3 idle
P2 idle
P1 idle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Time
Computation Time
Communication Time - among processors
Communication Time - assembling and delivering data

Figure 3-2 Simulation of pre ITPMMA Algorithms

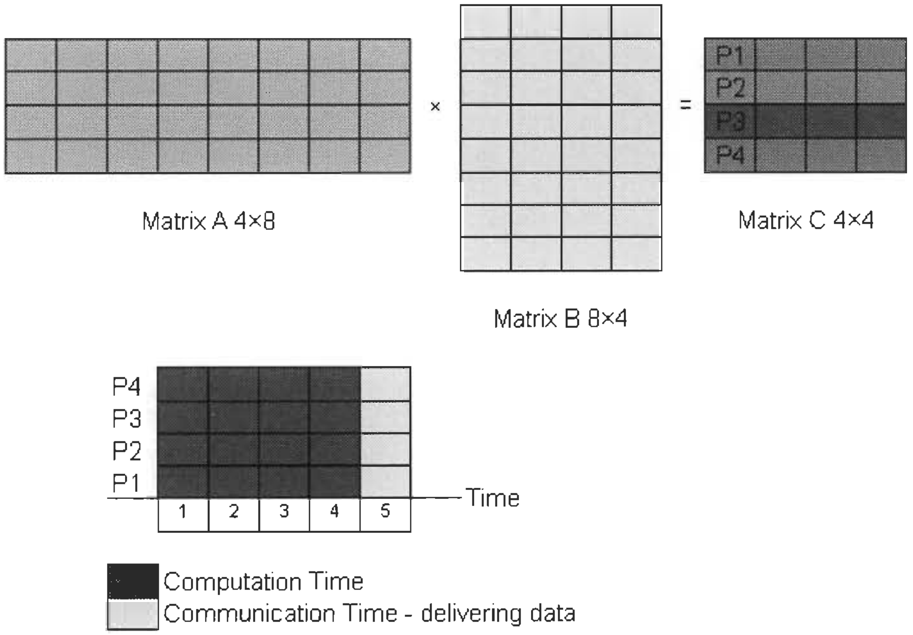


Figure 3-3 Simulation of ITPMMA Algorithm

3.2.1. ITPMMA flowchart

ITPMMA Algorithm for parallel matrix multiplications, on the contrary of other parallel matrix multiplication algorithms, it depends on reformatting the matrix multiplication process into many independent vector multiplication operations, each vector multiplication operation will be carried out by single processor, to avoid any data dependency and processor-to-processor data transferring time. Figure 3-3 shows ITPMMA Algorithm flowchart.

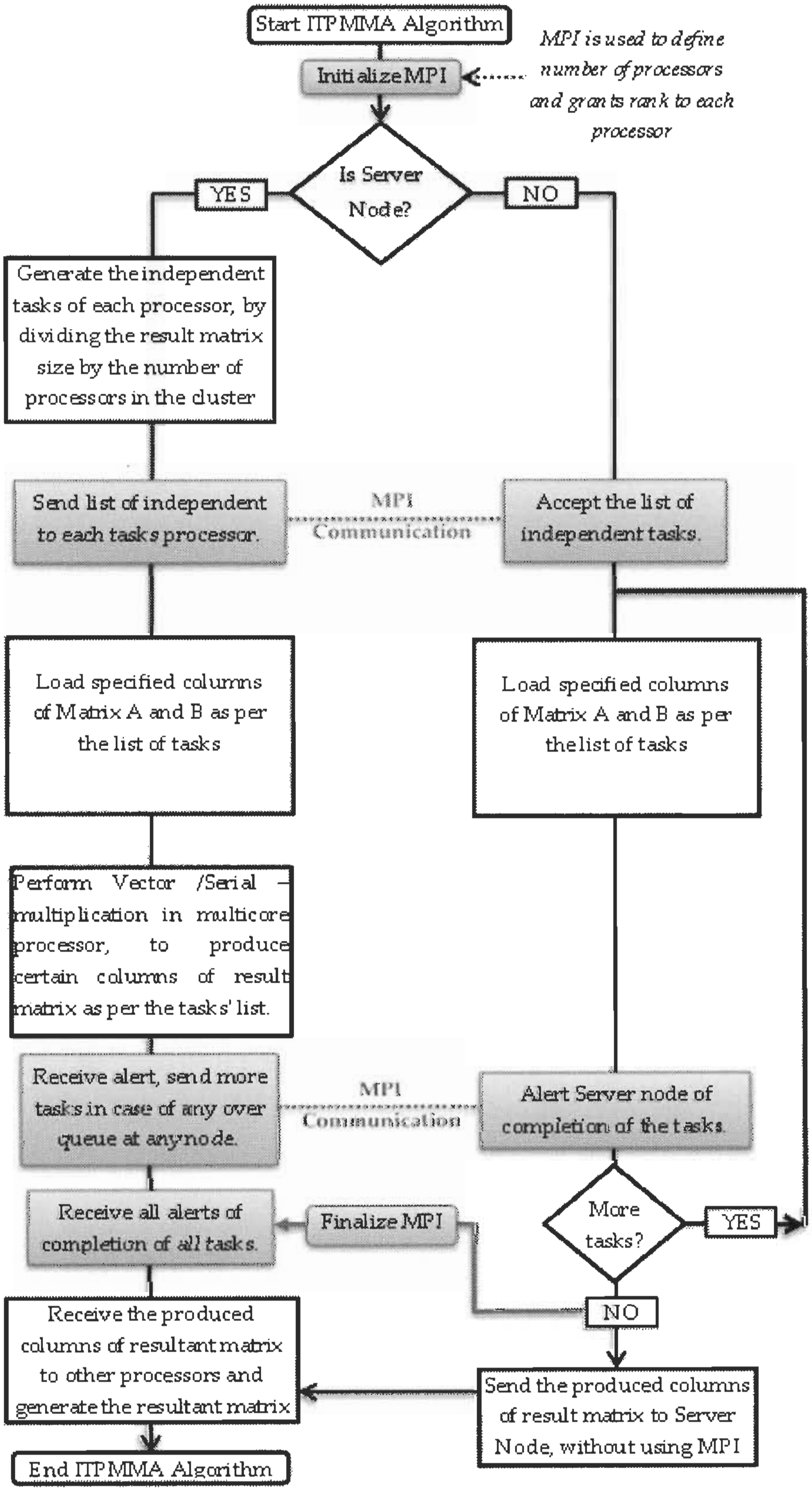

Start ITPMMA Algorithm
Initialize MPI
MPI is used to define number of processors and grants rank to each processor
Is Server Node?
YES
NO
Generate the independent tasks of each processor, by dividing the result matrix size by the number of processors in the cluster
Send list of independent to each tasks processor.
MPI Communication
Accept the list of independent tasks.
Load specified columns of Matrix A and B as per the list of tasks
Load specified columns of Matrix A and B as per the list of tasks
Perform Vector /Serial – multiplication in multicore processor, to produce certain columns of result matrix as per the tasks' list.
Receive alert, send more tasks in case of any over queue at any node.
MPI Communication
Alert Server node of completion of the tasks.
Receive all alerts of completion of all tasks.
Finalize MPI
More tasks?
YES
NO
Receive the produced columns of resultant matrix to other processors and generate the resultant matrix
Send the produced columns of result matrix to Server Node, without using MPI
End ITPMMA Algorithm

Figure 3-4 ITPMMA Algorithm flowchart

ITPMMA Algorithm uses MPI library to define the number of active processors on the cluster and rank them. In addition, MPI library is used to transfer the lists of tasks to the processors and to forward task completion alerts from different processers to the node processor. So the server node: received bird sow

1. Defines the processors available in the parallel cluster and to rank them.
2. Generate the the independent tasks, so produce an element of the result matrix is considered as independent task.
3. Divides the last of independent tasks by the number of active processors in the cluster. So for the output matrix of size 7000×7000 and number of processors is 128 active processors, (7000×7000)/128=382812.5, so 64 processors produce 382,812 independent tasks and 64 processors produce 382,813 independent tasks. So the time needed to process 7000×7000 matrix multiplication in parallel of 128 processors is equivalent process 620×620 in one processor.
4. Sends for each processor list of independent tasks to be carried out.
5. Receives from each processor alert of completion when all assigned tasks have been carried out.
6. Redistributes the processor of any faulted processor(s) to other active ones by sending extra tasks for the high computing power processors, once the initial sent list of tasks of these processors had been carried out. Redistribution includes shifting tasks from the over queued low computing power processors to the high computing ones; so this algorithm address the hetregenious enviroment.

MPI library will not be used to exchange any data at all, as all tasks are independent, so, no processor receives any data from another processor to be able to complete its work, nor any processor communicate with non-server node processor.

I will explain the algorithm using different four examples:

1. Square Matrix multiplication, and the size of the result matrix is multiple of the number of processors being used in parallel, like $A_{4x4} \times B_{4x4} = C_{4x4}$, for four processors in parallel; well, the size of the result matrix is 4×4, is multiple of the number of processor in use, which is 4.

2. Square Matrix multiplication, and the size of the result matrix is not multiple of the number of processors being used in parallel, $A_{12x12} \times B_{12x12} = C_{12x12}$, for eight processors in parallel; it is obvious that 12 is not multiple of 8.

3. Non Square Matrix multiplication, and the size of result matrix is multiple of the number of the processors being used in parallel, like $A_{12x12} \times B_{12x16} = C_{12x16}$, for four processors in parallel; it is obvious that both 12 and 16 are multiple of 4.

4. Non Square Matrix multiplication, and the size result matrix is not multiple of the number of the processors being used in parallel, like $A_{12x12} \times B_{12x18} = C_{12x18}$, for four processors in parallel; it is obvious that 18 is not multiple of 4.

The second and the fourth example will help me to show how ITPMMA algorithm will address the problem of load balance to be very obvious. On the other hand all previous algorithms for parallel matrix multiplication are limited to square matrix multiplication only.

3.3.ITPMMA applied to different matrices sizes

3.3.1. Square Matrix multiplication, size of the result matrix is multiple of the number of processors in parallel

In this example, $A_{4x4} \times B_{4x4}$ being executed on four processors P_0, P_1, P_2, and P_3, matrix A will be sent to all processors, while one single column of matrix B will be sent to each matrix, so each processor produces part of the matrices multiplication result as shown of Figure. 3-4.

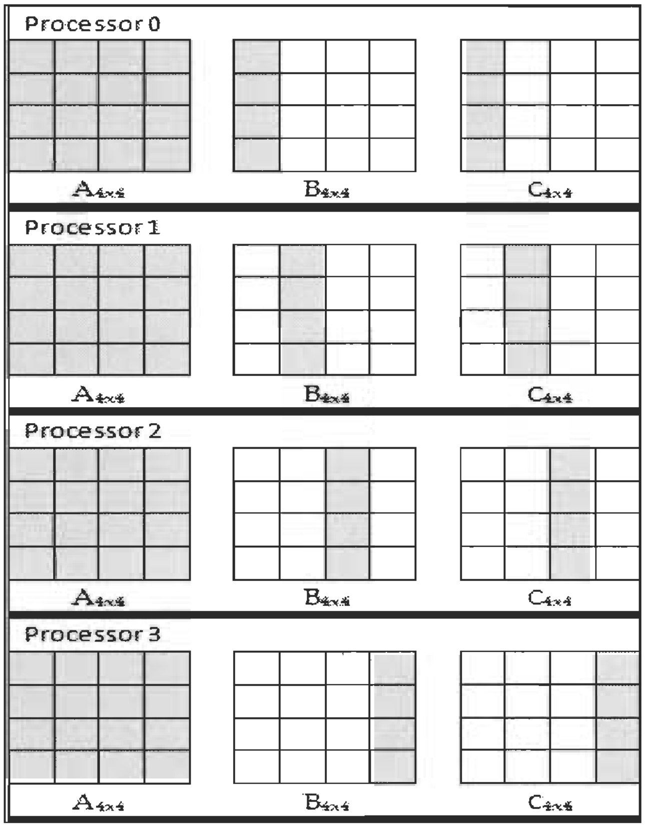
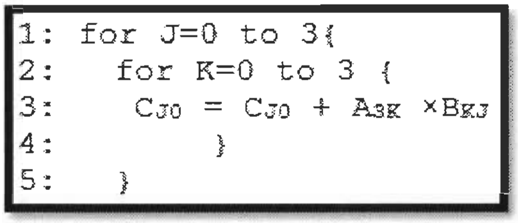


Figure 3-5 Task distribution of $A_{4x4} \times B_{4x4}$, where each processor will produce part of the result matrix C_{4x4}

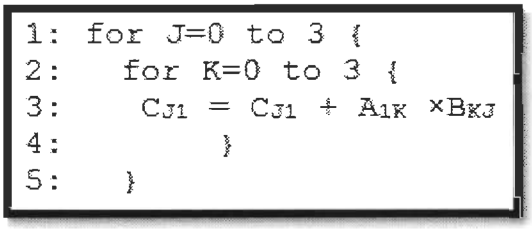
Each processor processes separate independent tasks, so no messages of intermediate results will be exchanged, instead each processor produces what it needs when needed, that is to reduce the time consumed for exchange intermediate results between the processors and to avoid data dependencie which put the processor on hold waiting results from other processors. So, P_0 will execute the code shown in Figure 3-5:



```
for J=0 to 3{
   for K=0 to 3 {
    $C_{J0} = C_{J0} + A_{3K} \times B_{KJ}$
         }
   }
```

Figure 3-6: Pseudo code executed by processor P0

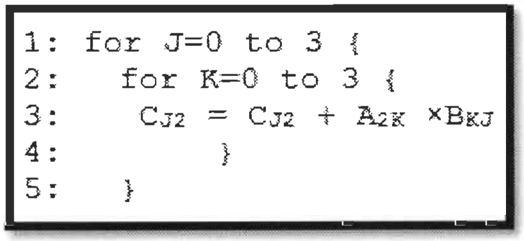
While P_1 will execute the code shown in Figure 3-6:



```
for J=0 to 3 {
   for K=0 to 3 {
    $C_{J1} = C_{J1} + A_{1K} \times B_{KJ}$
         }
   }
```

Figure 3-7: Pseudo code executed by processor P1

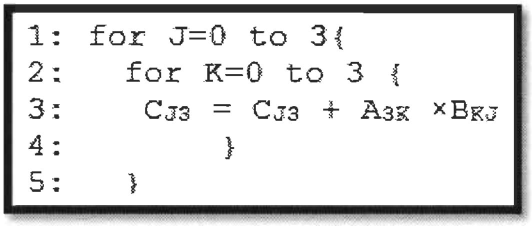
While P2 will execute the code shown in Figure 3-7:



```
for J=0 to 3 {
   for K=0 to 3 {
    $C_{J2} = C_{J2} + A_{2K} \times B_{KJ}$
         }
   }
```

Figure 3-8: Pseudo code executed by processor P2

While P3 will execute the code shown in Figure 3-8:



```
for J=0 to 3{
  for K=0 to 3 {
   $C_{J3} = C_{J3} + A_{3K} \times B_{KJ}$
       }
  }
```

Figure 3-9: Pseudo code executed by processor P3

As we can see, no processor waits its entries from another processor(s), and no processor sends some results to other processor, so we could reduce the data dependency and exchanged messages to zero, which has played backward role on the performance of the previous parallel matrix multiplications. To generalize the case mentioned above, for different matrices size, I will use the example of $A_{12x12} \times B_{12x12} = C_{12x12}$, for four parallel processors. So, each processor will produce three columns of the matrix C, processor P_0 will produce three columns, these are the first and fifth and ninth columns of the matrix $C_{12\times12}$, Figure 3-9 shows task distribution of $A_{12x12} \times B_{12x12} = C_{12\times12}$.

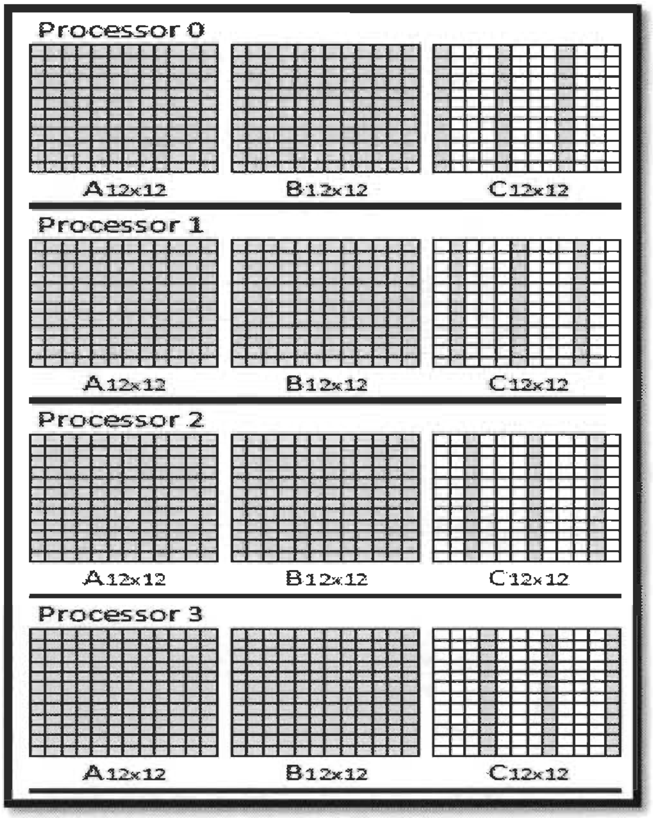
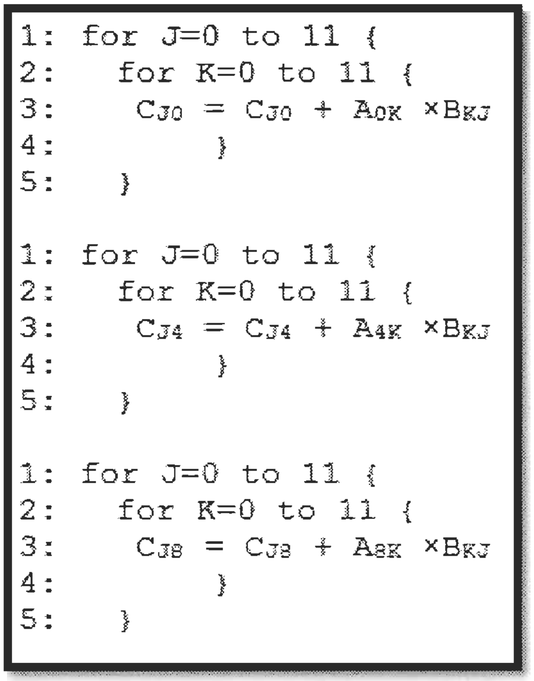


Figure 3-10 Task distribution of A_{12x12} × B_{12x12}, where each processor will produce part of the result matrix C_{12x12}

Processor 0 executes the code shown in Figure 3-10, to produce the first and the fifth and the ninth columns of the result matrix C_{12x12}.



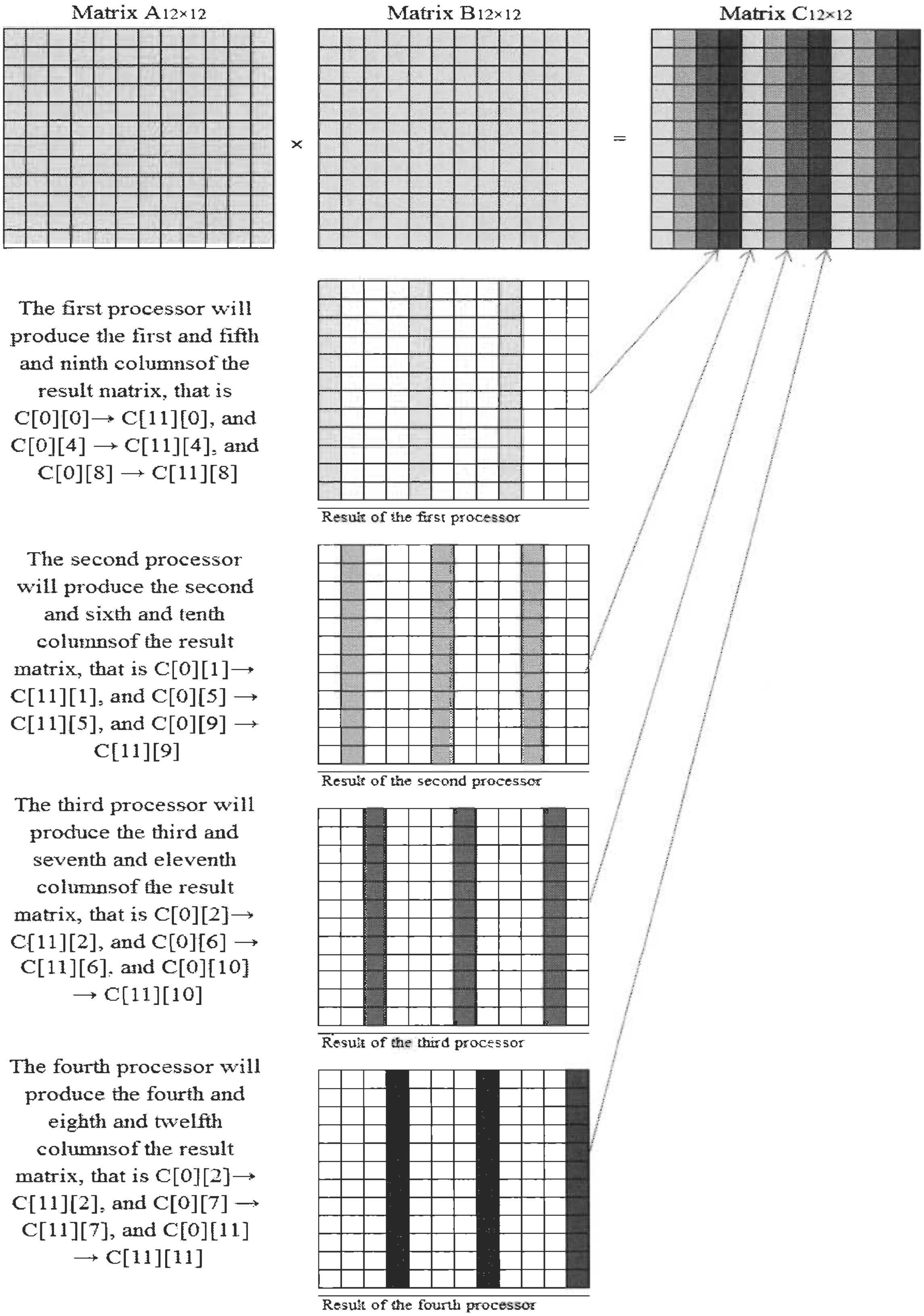
```
for J=0 to 11 {
  for K=0 to 11 {
   $C_{J0} = C_{J0} + A_{0K} \times B_{KJ}$
      }
  }

for J=0 to 11 {
  for K=0 to 11 {
   $C_{J4} = C_{J4} + A_{4K} \times B_{KJ}$
      }
  }

for J=0 to 11 {
  for K=0 to 11 {
   $C_{J8} = C_{J8} + A_{8K} \times B_{KJ}$
      }
  }
```

Figure 3-11: Pseudo code executed by processor P0, to produce the first and the fifth and the ninth columns of the result matrix C_{12x12}

While processor P_1 produces different three columns, these are the second and sixth and tenth columns of the matrix $C_{12\times12}$, and so on for the remaining processors. Figure 3-11 shows the details of ITPMMA algorithm for the multiplication of $A_{12x12} \times B_{12x12}$, using four processors in parallel.


Matrix $A_{12\times12}$
×
Matrix $B_{12\times12}$
=
Matrix $C_{12\times12}$
Result of the first processor
Result of the second processor
Result of the third processor
Result of the fourth processor

The first processor will produce the first and fifth and ninth columnsof the result matrix, that is C[0][0]→ C[11][0], and C[0][4] → C[11][4], and C[0][8] → C[11][8]

The second processor will produce the second and sixth and tenth columnsof the result matrix, that is C[0][1]→ C[11][1], and C[0][5] → C[11][5], and C[0][9] → C[11][9]

The third processor will produce the third and seventh and eleventh columnsof the result matrix, that is C[0][2]→ C[11][2], and C[0][6] → C[11][6], and C[0][10] → C[11][10]

The fourth processor will produce the fourth and eighth and twelfth columnsof the result matrix, that is C[0][2]→ C[11][2], and C[0][7] → C[11][7], and C[0][11] → C[11][11]

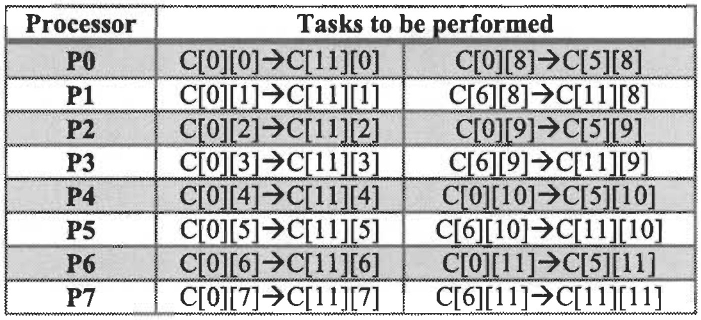
Figure 3-12: $A_{12x12} \times B_{12x12}$ using ITPMMA algorithm for parallel matrix multiplication

The tasks are fully independent tasks – so no processor needed to exchange data with other processors – so there was no time being wasted by exchanging messages, and no processor has stayed idle waiting its input from other processors, well, this is the essence of ITPMMA algorithm.

3.3.2. Square Matrix multiplication, size of the result matrix is not multiple of the number of processors in parallel

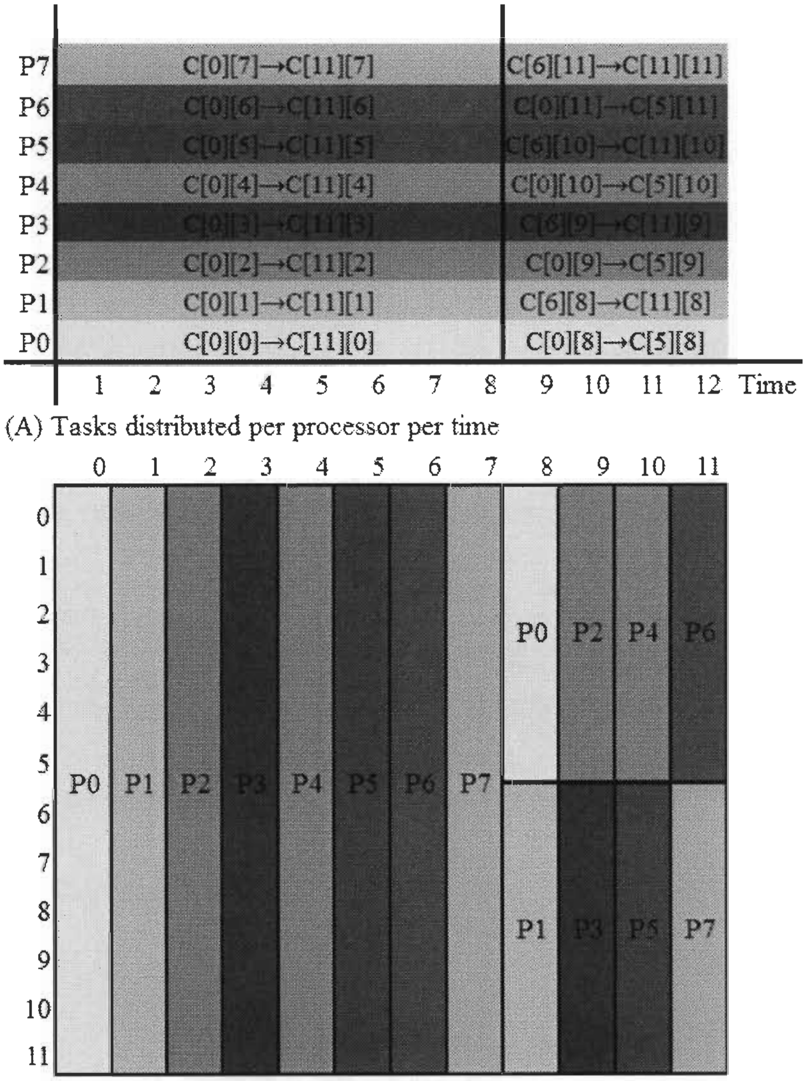
For this case, the size of the multiplied matrices is not multiple of the number of the processor. In this example, we are to multiply $A_{12x12} \times B_{12x12}$, while the number of processors is eight, where 12 is not multiple of 8. In this case, we have to overcome this issue, and schedule the independent tasks between the processors equally, so the load is balanced, and no processor will stay idle while other processors still overloaded. So, we can run the schedule shown in Table 3-1, which satisfy the criteria of ITPMMA algorithm.

Table 3-1 Tasks to be performed by each processor



Processor	Tasks to be performed	
P0	C[0][0]→C[11][0]	C[0][8]→C[5][8]
P1	C[0][1]→C[11][1]	C[6][8]→C[11][8]
P2	C[0][2]→C[11][2]	C[0][9]→C[5][9]
P3	C[0][3]→C[11][3]	C[6][9]→C[11][9]
P4	C[0][4]→C[11][4]	C[0][10]→C[5][10]
P5	C[0][5]→C[11][5]	C[6][10]→C[11][10]
P6	C[0][6]→C[11][6]	C[0][11]→C[5][11]
P7	C[0][7]→C[11][7]	C[6][11]→C[11][11]

Each processor produce 18 elements of the output matrix C, which satisfies the load balance between the processors, also, each task – and so each processor – is independent from any other tasks, and so, there is no exchanged messages. Figure 3-12 (A) shows the time chart for the eight processors for this example $A_{12x12} \times B_{12x12} = C_{12x12}$, it is obvious that all processors working, and no processor is idle, and no processor finishes its tasks before the others, which implies the highest efficiency of using the processors and the highest load balance been achieved. On the other hand, Figure 3-12 (B) shows the map of between the elements of the output matrix C_{12x12}, and the processor that will produce each. Finally, Figure 3-12 (C) shows the total number of tasks being produced by each processor.



(A) Tasks distributed per processor per time

(B) Elements of Matrix $C_{12\times12}$, mapped per processor where each element being produced

Total elements being produced by a processor	
P0	18
P1	18
P2	18
P3	18
P4	18
P5	18
P6	18
P7	18

(C) Toral tasks per each processor

Figure 3-13 Tasks to be performed by each processor of the eight processor to produce the matrix $C_{12\times12}$

3.3.3. Non - square Matrix multiplication, size of the result matrix is multiple of the number of processors in parallel

First of all, the previous parallel matrix multiplication algorithms avoided non-square matrices multiplications, as they cannot perform it proper, some algorithms pad zeroes to make the matrices square, and then execute the multiplication. The main point here is to define and schedule the tasks within the three constraints:

1. Task independency, implies no processor will get intermediate results from another processor, to process its tasks.
2. Load balance, implies all processors will execute same size of tasks in term of number and size of math operations.
3. Processor efficiency, which implies no processor, will stay idle while another processors still over queued with tasks.

I will use the example of multiplying $A_{12x12} \times B_{12x16} = C_{12x16}$ at four processors. The tasks distribution is shown in Figure 3-13. In this case, the algorithm defines producing each element of the result matrix as an independent task, so total number of the independent tasks is 12×16= 192, so each processor produces 192/4=48 independent tasks, which is equavelent to four columns.

The first processor will produce the first and fifth and ninth and the thirteenth columns of the result matrix, that is C[0][0]→ C[11][0], and C[0][4] → C[11][4], and C[0][8] → C[11][8] and C[0][12] → C[11][12].

The second processor will produce the second and sixth and tenth and the fourteenth columns of the result matrix, that is C[0][1]→ C[11][1], and C[0][5] → C[11][5], and C[0][9] → C[11][9] , and C[0][13] → C[11][13].

The third processor will produce the third and seventh and eleventh and fifteenth columns of the result matrix, that is C[0][2]→ C[11][2], and C[0][6] → C[11][6], and C[0][10] → C[11][10], and C[0][14] → C[11][14].

The fourth processor will produce the fourth and eighth and twelfth and sixteenth columns of the result matrix, that is C[0][2]→ C[11][2], and C[0][7] → C[11][7], and C[0][11] → C[11][11], and C[0][15] → C[11][15].

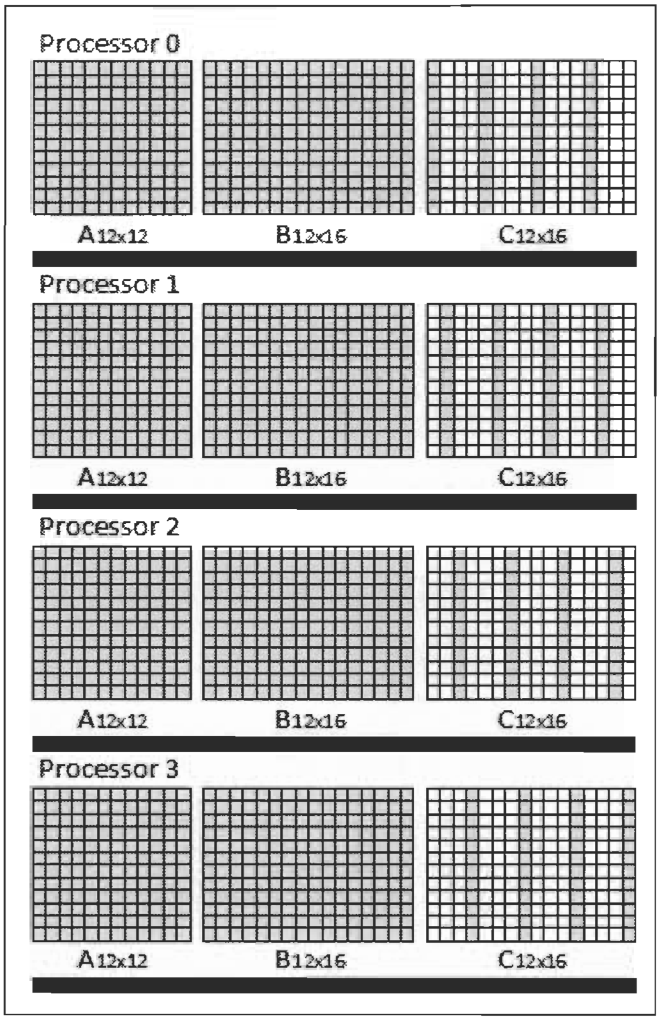


Figure 3-14 Task distribution of $A_{12x12} \times B_{12x16}$, where each processor will produce different part of the result matrix C_{12x16}

3.3.4. Non - square Matrix multiplication, size of the result matrix is not multiple of the number of processors in parallel

The other case that to test, is the matrix multiplication where the size of the multiplied matrices is not multiple of the number of the processors, so distribution of the independent and equal tasks will take different way. I will use the example of multiplying $A_{12x12} \times B_{12x18} = C_{12x18}$ at four processors. Figure 3-14 shows task distribution of the tasks to multiply $A_{12x12} \times B_{12x18}$ in parallel using 4 processors. So, we have 12×18=216 independent tasks, each processor should process 216/4=54 tasks. The output matrix has 18 columns each of 12 elements or independent tasks. So, 54/12=4.5, so each processor of the four processors produces 4 columns (each column of 12 elements) of the result matrix C_{12x18}, plus six elements, this makes the total number of elements is 4×12+6=54 elements. Each element will be produced by single one processor from A to Z, which implies the independency, so no intermediate values to be exchanged.

Last example in this subsection, for multiplying matrices A(3 × 3) * B(3 × 5) = C(3 × 5) and we have 3 processors. So we have 3 × 5 = 15 independent tasks for 3 processors, so 15 (independent tasks) / 3 (processors) = 5 independent tasks to be processed by each processor.

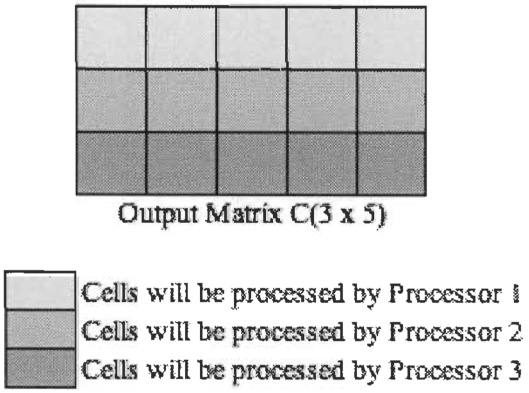

Output Matrix C(3 x 5)
Cells will be processed by Processor 1
Cells will be processed by Processor 2
Cells will be processed by Processor 3

Figure 3-15 Multiplions of matrices A(3 × 3) * B(3 × 5) = C(3 × 5) using 3 processors

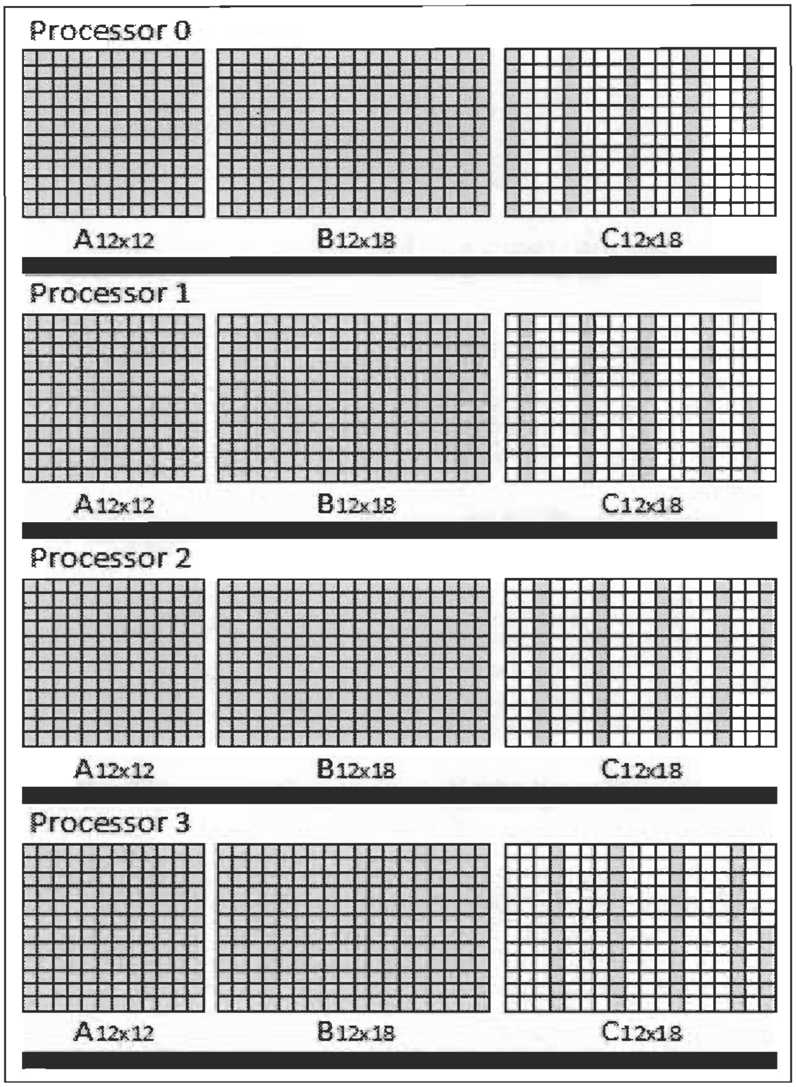


Figure 3-16 Task distribution of A_{12x12} × B_{12x18}, where each processor will produce part of the result matrix C_{12x18}.

3.4.ITPMMA properties

In ITPMMA Algorithm:

1. No processor stays idle to wait output of some processor else. Since all tasks are independent tasks, and equal in size, in term of number and size of the mathematical operations.

2. No communication time cost among the processors in term of data movement; communication is only between the server node processor and other processors to define the rank of each processor, and to send the tasks lists.

3. Each processor utilizes its full capabilities, like the multicores and cache memory, to complete the current task as fast as possible. This implies ITPMMA algorithm consider the up-to-date structure of the processor to complete the task in shortest time.

4. Load balanced as the tasks already balanced among the processors. In case a processor completed its tasks before other processors, the server node processor will be alerted, so, the server node processor may redirect some other`s over queued task(s) to it. The over queue phenomenon could happen when different processors architectures of different capabilities are involved in the parallel cluster.

3.4.1. ITPMMA Complexity

ITPMMA Algorithm execution time equals to (Serial matrix multiplication time divided by number of processors) – time for creating and distribution the lists of tasks.

$$T_p(calc) = \left(\frac{n^2}{p}\right).(2n-1).\tau \qquad \text{Eq. } 3-1$$

Multiplying two N × N matrices requires N multiplications and N − 1 additions operations for each element of the result matrix. Since there are N^2 elements in the matrix this yields a total of N^2 (2N − 1) floating-point operations, or about $2N^3$ for large N.

So for multiplying two matrices of 10×10 each, total number of operations is 102 (2×10 − 1) = 1900 floating-point operations (FLOP). For N = 1000, $2N^3$ =2 megaFLOP (million floating point operations). For N = 32768, Number of float point operations is 105,553,116,266,496 ≈ 100 teraFLOP (trillion floating point operations). Using ITPMMA algorithm, for 16 processors in the cluster, this number of operations must be divided by 16 which yield 6,597,069,766,656 ≈ 6 teraFLOP. So, the speedup is 100/6=15. The experiment in Table 3 shows the speed up of 13, with less of 2 times, which is result of time to upload the required data and the time which the server node needs to create and distribute the independent tasks.

On the other hand, the analytical solution of ITPMMA Algorithm shown in equation 3 – 1, implies the communication time is negligible, since the communication through the cluster is limited to sending the list of tasks to the processors, and neither the elements of the multiplied matrices nor the intermediate results to be transmitted between the processors. By comparing ITPMMA analytical solution with the analytical solutions of Fox algorithm and Cannon algorithm developed at the experiments have been carried out on 2005 at Lobachevsky State University of Niznhi Novgorod [60] where the total execution time of Fox algorithm is

$$T_p = q[\left(\frac{n^2}{p}\right).\left(\frac{2n}{q} - 1\right) + \left(\frac{n^2}{p}\right) \cdot \tau + (q\ log_2 q + (q - 1)(\alpha + \frac{w\left(\frac{n^2}{p}\right)}{\beta}) \qquad \text{Eq. 3 - 2}$$

And the total time of Cannon algorithm is

$$T_p = q[\left(\frac{n^2}{p}\right).\left(\frac{2n}{q} - 1\right) + \left(\frac{n^2}{p}\right) \cdot \tau + (2q + 2)(\alpha + \frac{w\left(\frac{n^2}{p}\right)}{\beta}) \quad \text{Eq. } 3 - 3$$

Equations 3-1 and 3-2 and 3-3 are compatible with the experiments we carried out in next section.

3.4.2. ITPMMA Load Balancing

Multi-core processors have very high specifications were not familiar before:

1. ~1 TFLOP of compute power per core
2. 61+ of cores, 100+ hardware threads
3. Highly heterogeneous architectures (cores + specialized cores + accelerators/coprocessors)
4. Deep memory hierarchies.

The pre ITPMMA Algorithms had been developed before the availability of these specifications, and so, these specifications have never been utilized before. In ITPMMA Algorithm, we decompose the parallel matrix multiplication into independent serial tasks to be executed in parallel, so each serial task utilizes new modern processor architecture. ITPMMA Algorithm does not interfere how each processor should process its task. The modern multicore processors have its own management algorithm to decompose the tasks into smaller tasks and distribute these tasks in parallel among the cores of the processor, also, it has own algorithm to use different levels of cache memory. The pre ITPMMA Algorithms allow each processor in the cluster to process simple equal multiplication or addition tasks using only one single core. In addition, these tasks are simple, so it does not utilize the multiple cores or cache memory at all. In addition, the modern processors implement the concept of multithreading, which results in better speed up levels, and it is used when the modern processor to process huge tasks.

ITPMMA Algorithm utilizes the new features of the modern processors in terms of their ability to process huge size and complicated tasks faster than old processors – single core and without cache – which implies that the concept that has been used long by all previous parallel algorithm which states that the smaller block size the much faster parallel processing speed is not valid anymore. Instead, the smaller block size, where there is less number of mathematic operations, is the less utilizing the modern processors' capabilities. Figure 3-2 simulates Block-Striped Decomposition parallel matrix multiplication algorithm, where processor P1 and P4 are idle, while processor P2 and P3 still working. Figure 3-3 simulates ITPMMA Algorithm, where most of the processors become idle at the same time.

3.4.3. ITPMMA Communication Cost

Communication cost between processors in ITPMMA Algorithm among the processors equals to zero. There is no communication to take place between processors. Processors receives list of tasks to be executed from the server node processor, once they executed individually, they alerts and server node processor using MPI libraries. The tasks all are independent, so no data dependency needs any communication among the processors.

3.4.4. ITPMMA algorithm (STMMA) Efficiency

We will consider the below equation to calculate the efficiency of ITPMMA (STMMA) algorithm:

$$C_{m\times n}=A_{m\times k} \times B_{k\times n} \qquad \text{Eq. 3-4}$$

Sequential time of matrix multiplication algorithm T_{SEQ} is N^3 (in case of square matrix multiplications where $m=n=k=N$). The parallel time of multiplying the same square matrices using ITPMMA (STMMA) algorithm T_{PRL} is N^3/p, while p is number of processors being used in the parallel processing.

The startup cost or latency is to be neglected in the network of sufficient bandwidth.

The speed up is the Sequential Time T_{SEQ} *to* parallel time T_{PRL}

$$\text{Speed up} = T_{SEQ} / T_{PRL}$$

While

$$\text{Efficiency } (\eta) = \text{Speedup} / p$$

3.4.4.1. Comparisons of ITPMMA with selected algorithms

In this part, I will compare analytically between ITPMMA algorithm and selected parallel matrix multiplication algorithms. The comparison of ITPMMA (STMMA) algorithm will consider the following algorithms:

1. Systolic algorithm.
2. Cannon's algorithm.
3. Fox's algorithm with square decomposition.
4. Fox's algorithm with scattered decomposition

While the comparison of ITPMMA (STMMA) algorithm with the algorithms PUMMA (MBD2), SUMMA and DIMMA will be shown in Appendix C.

The above selected algorithms being compared theoretically and experimentally on so many journal and conference papers, also, it appears on so many literature and books which target parallel matrix multiplication.

Through the theoretical analysis, the following symbols will be used[1]:

- f = number of arithmetic operations units
- tf = time per arithmetic operation $<<$ tc (time for communication)
- c = number of communication units
- $q = f / c$ average number of flops per communication access
- Minimum possible time = $f^* tf$ when no communication
- Efficiency(speedup) $SP=q^*(tf/tc)$
- $f * tf + c^* tc = f * tf * (1 + tc/tf * 1/q)$
- $m2 = n2/p$

I will summarize the ITPMMA (STMMA) algorithm's tasks and execution time in Table 3-4. As it is shown, there is no tasks like shift or transpose or broadcast or switch as it is on the other algorithms shown on the tables 3-5 till Table 3-11. ITPMMA (STMMA) algorithm implies send the matrices to the processors, each processor will generate the result matrix elements by perform multiplication and addition operations on the matrices being transferred to it, finally the results will be collected from different processors to have on single complete result matrix.

Table 3-2 ITPMMA (STMMA) algorithm's tasks and execution time [59]

Task	Execution time

[1] These definitions appeared in several literatures.

Send A, B matrices to the processors	$2mnp\ t_c$
Multiply A and B	$m^2n\ t_f$
Generate the resulting matrix	$2mn\ t_c$
Total execution time	$2mn\ t_c\ (1 + p) + m^2n\ t_f$

Table 3-3 Systolic Algorithm [61]

Task	Execution time	ITPMMA (STMMA)
Transpose B matrix	$2n^2\ t_f$	0
Send A, B matrices to the processors	$2m^2p\ t_c$	$2m^2p\ t_c$
Multiply the elements of A and B	$m^2n\ t_f$	$m^2n\ t_f$
Switch processors' B sub-matrix	$n^2\ t_c$	0
Generate the resulting matrix	$n^2\ t_f + n^2\ t_c$	$2mn\ t_c$
Total execution time	$t_f(m^2\ n + 3n^2) + t_c(4n^2)$	$2mn\ t_c\ (1 + p) + m^2n\ t_f$

Table 3-4 Cannon's Algorithm [62]

Task	Execution time	ITPMMA (STMMA)
Shift A, B matrices	$4n^2\ t_f$	0
Send A, B matrices to the processors	$2n^2\ t_c$	$2mnp\ t_c$
Multiply the elements of A and B	$n^2\ t_f$	$n^2\ t_f$
Shift A, B matrices	$2(mn\ t_c + 2m^2\ n\ t_f)$	0
Generate the resulting matrix	$n^2\ t_f + n^2\ t_c$	$2mn\ t_c$
Total execution time	$t_f(5m^2\ n + 5n^2\ ) + t_c(2n^2 + 2mn)$	$2mn\ t_c\ (\ 1 + p\ ) + m^2n\ t_f$

Table 3-5 Fox's Algorithm with square decomposition [63]

Task	Execution time	ITPMMA (STMMA)
Send B matrix	$n^2\ t_c$	$2mnp\ t_c$
Broadcast the diagonal elements of A	$mnp\ t_c$	0

Multiply A and B	$m^2n\ t_f$	$n^2\ t_f$
Shift A, B matrices	$mn\ t_c + 2m^2\ n\ t_f$	0
Generate the resulting matrix	$n^2\ t_f + n^2\ t_c$	$2mn\ t_c$
Total execution time	$t_f(3m^2\ n + n^2\ ) + t_c(2n^2 + mn(p+1))$	$2mn\ t_c\ (\ 1 + p\ ) + m^2n\ t_f$

Table 3-6 Fox's Algorithm with scattered decomposition, [64]

Task	**Execution time**	**ITPMMA (STMMA)**
Scatter A	$n^2\ t_c$	0
Broadcast the diagonal elements of B	$mnp\ t_c$	$2mnp\ t_c$
Multiply A and B	$m^2n\ t_f$	$n^2\ t_f$
Switch processors' A submatrix	$mn\ t_c$	0
Generate the resulting matrix	$n^2\ t_f + n^2\ t_c$	$2mn\ t_c$
Total execution time	$t_f(m^2\ n + n^2\ ) + t_c(2n^2 + 2m^2\ n + mnp)$	$2mn\ t_c\ (\ 1 + p\ ) + m^2n\ t_f$

By summarizing the above tables (Table 3-4 till Table 3-11) I will conclude the following execution time table for all algorithms, Table 3-12.

Table 3-7 Algorithms' execution time summary table

Algorithm	Execution Time
ITPMMA (STMMA) **Algorithm**	$2mn\ t_c (1 + p) + m^2n\ t_f$
Systolic Algorithm	$t_f(m^2 n + 3n^2) + t_c(4n^2)$
Cannon Algorithm	$t_f(5m^2 n + 5n^2) + t_c(2n^2 + 2mn)$
Fox's Algorithm with square decomposition	$t_f(3m^2 n + n^2) + t_c(2n^2 + mn(p+1)$
Fox's Algorithm with scattered decomposition	$t_f(m^2 n + n^2) + t_c(2n^2 + 2m^2 n + mnp)$
PUMMA (MBD2)	$t_f(m\ 2n + n^2) + t_c(2n^2 + m^2 root(p)(p+1))$
SUMMA	$t_f(m\ 2n + n^2) + t_c(n^2 + 2mnp)$
DIMMA	$t_f(m\ 2n + n2) + t_c(n^2 + 2mnp)$

It is so obvious that ITPMMA (STMMA) algorithm has the minimum execution cost.

3.5.Conclusion

In this chapter, I have introduced new parallel matrix multiplication algorithm, ITPMMA algorithm, which is faster than the previous parallel matrix multiplication in term of analytical analysis against Cannon Algorithm and Fox algorithm.

On the other hand, ITPMMA algorithm had been compared analytically against PUMMA, SUMMA, and DIMMA as shown in Appendix C.

Chapter 4 - ITPMMA Analysis and Results

4.1. Introduction

In this chapter, I will implement and execute ITPMMA algorithm into two different real parallel environments:

1. Simple parallel environment of four PCs, and then sixteen PCs connected via Ethernet card. The PCs exchange the data using MPI library. The PCs processors are Intel(R) Core(TM) i5 CPU 760 @2.80GHz 2.79 GHz, installed memory (RAM) is 4.00 GB, System type: 64-bit Operating System, Windows 7 Professional. This environment is considered as heterogeneous environment since all PCs been assembled by different vendors and no consideration of homogeneity has been considered in assembling the PCs.
2. CLUMEQ supercomputer homogenous environment to be utilized here to test the algorithm in big size environment in term of number of processors. CLUMEQ is a Supercomputer Consortium Laval UQAM McGill and Eastern ITPMMA based in McGill University founded in 2001. It has three clusters, Colosse, Krylov, and Guillimin. I have used Guillimin Cluster in this thesis. Guillimin is a compute cluster comprised of 1200 compute nodes and 34 infrastructure nodes. These nodes are each a pair of Intel Westmere-EP processors each with 6 cores and 24, 36 or 72 G Bytes of RAM memory per node. In total, Guillimin consists of 14400 cores and 46 T Bytes of memory. All nodes are connected via a high-performance QDR

InfiniBand network. Guillimin is also connected to the outside world via a 10 Gigabit Ethernet network. A parallel file system (GPFS) provides a usable capacity of 2 P Bytes, [65].

ITPMMA algorithm will be tested against both Cannon Algorithm and Fox Algorithm. The advantages of ITPMMA algorithm like reducing the exchanged messages among processors to zero will be explored. The load balance mechanism of ITPMMA algorithm will be explored against the other two algorithms, especially in the case of non-square matrix multiplication. The speedup and the efficiency of ITPMMA algorithm will be explored lively.

4.2.Simple parallel environment

I have executed both algorithms, ITPMMA and Cannon for three square matrices size, 100×100, and 500×500, and 5000×5000. And I have executed each experiment twice, one time for four processors, and the second time for sixteen processors. The reading of these experiments and the calculations are shown in Table 4-1.

Table 4-1 Comparison between the performance of ITPMMA (STMMA) and Cannon Algorithms for matrices of 100×100, 500×500, and 5000×5000

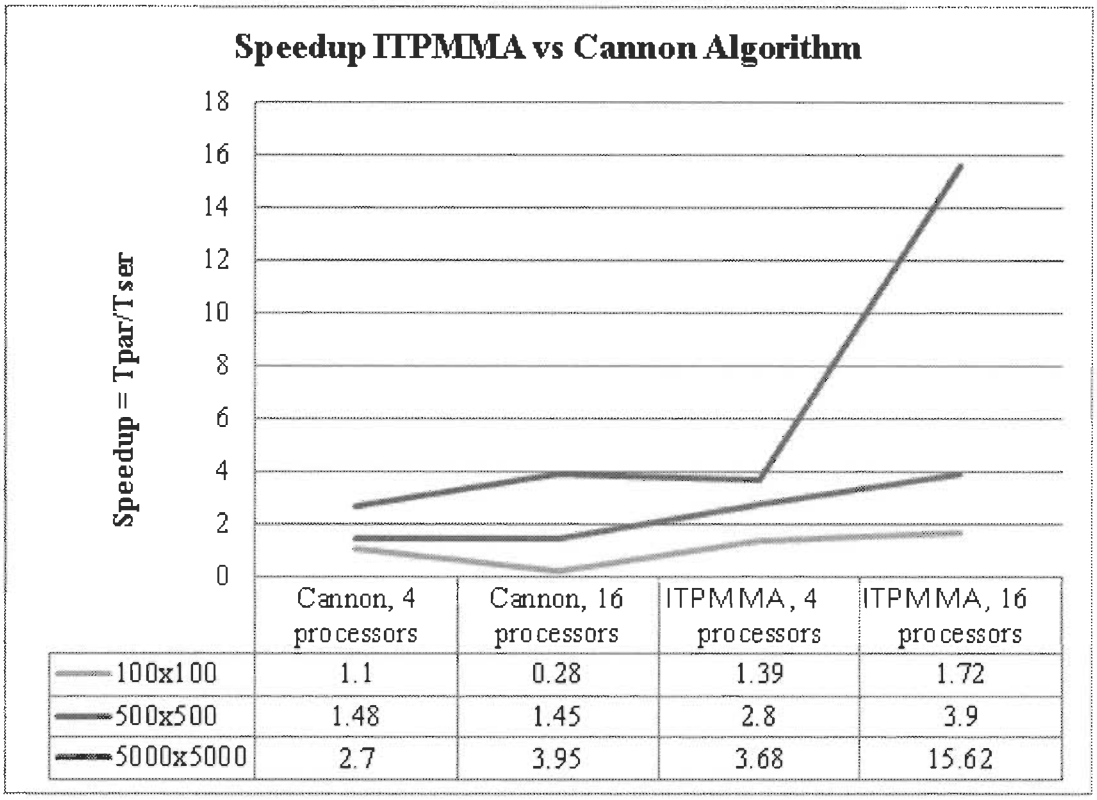
Test ID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Algorithm	**Cannon**		**STMMA**		**Cannon**		**STMMA**		**Cannon**		**STMMA**	
Matrix Size (n)	100	100	100	100	500	500	500	500	5000	5000	5000	5000
Decomposition (number of blocks)	4	16	-	-	4	16	-	-	4	16	-	-
Number of processor	4	16	4	16	4	16	4	16	4	16	4	16

Ts (1 processor) (ms)	43				874				11121			
Tp (ms)	39	**154**	31	25	594	601	312	225	4102	2812	3051	712
Speed up	1.1	**0.28**	1.39	1.72	1.48	1.45	2.8	3.9	2.7	**3.95**	3.68	15.62
Efficiency(1-2)	27.5	1.75	34.75	10.75	9.25	9.06	70	24.375	67.5	24.675	25	97.625

The table above shows the following facts:

1. Speedup for Cannon algorithm is:
    a. For 4 processors, speedups obtained are: 1.1, 1.48, and 2.7.
    b. For 16 processors, speedups obtained are 0.28, 1.45, and 3.95.
2. Speedup for ITPMMA algorithm is:
    a. For 4 processors, speedups obtained are: 1.39, 2.8, and 3.68.
    b. For 16 processors, speedups obtained are: 1.72, 3.9 and 15.62.

It is so clear, that ITPMMA algorithm exceed Cannon algorithm by several times ranging between 1.72/0.28 ≈ 6 times to 3.9/1.45 ≈ 2 times at test IDs 4 and 8 respectively. ITPMMA algorithm is 15.62/3.95 ≈ 4 times better than cannon algorithm, obtained at test ID 12, where ITPMMA algorithm completed the matrix multiplication of 5000 matrix size at 16 parallel processors at 712 seconds, where Cannon algorithm completed the same operations at 2812 seconds. Figure 4-1 shows a chart of the speedup of both ITPMMA and Cannon algorithm.



	Cannon, 4 processors	Cannon, 16 processors	ITPMMA, 4 processors	ITPMMA, 16 processors
100x100	1.1	0.28	1.39	1.72
500x500	1.48	1.45	2.8	3.9
5000x5000	2.7	3.95	3.68	15.62

Figure 4-1 Speed up of ITPMMA compared with Cannon

The chart above on figure 4-1 shows ITPMMA algorithm for 4 processors is better than Cannon algorithm for 16 processors in case matrix size of 100 and 500.

Another distinguished point is clear in the chart that Cannon algorithm for 4 processors – figure 4-1 – is better that Cannon algorithm itself for 16 processors for matrix size 100 in term of speed up. It is shown in test ID 2, where the speedup is less than one, which implies the serial test is faster than the parallel one. That happened at Cannon algorithm when multiplying two matrices at size 100, at 16 parallel processors, it takes 154 seconds while the serial multiplication takes only 43 seconds. This low speedup shows the high dependency of several calculations on other calculations. In Cannon algorithm, matrix of 100 size – figure 4-3 – divided between 16 processors, that means, each processor of the 16 processors did some task (addition and multiplication) on producing each element of the result matrix; that means the total 100 operations of multiplication and the total 100 operations of addition to produce one single element of the output matrix being distributed among the 16 processors, so fifteen processors will not be able to complete their tasks unless each receives output from the adjacent processor. Figure 4-2 and 4-3 show the speed up charts for 4 and 16 processors respectively.

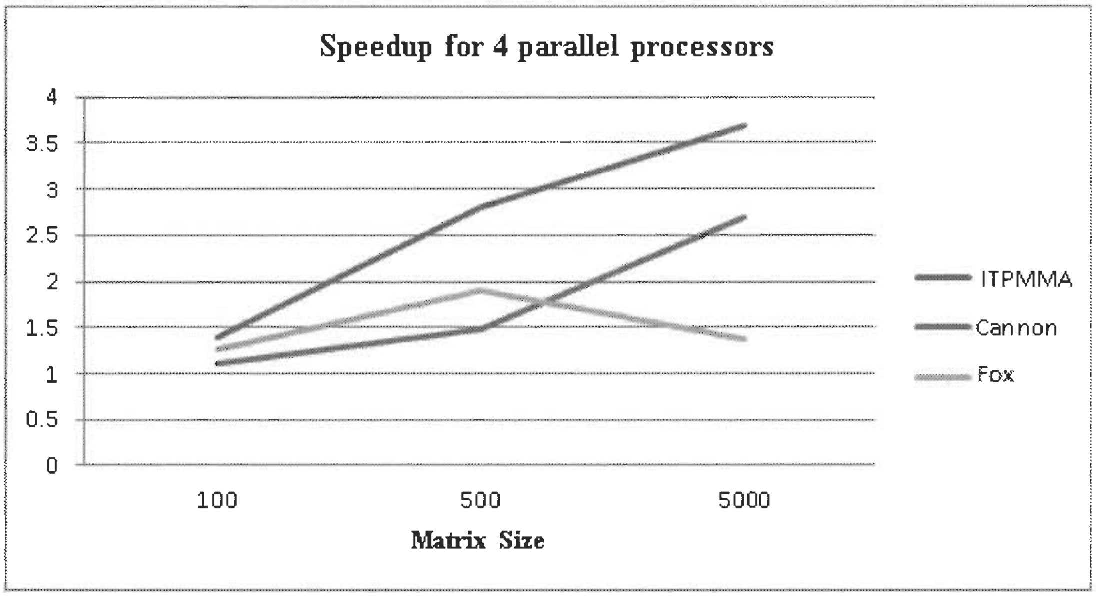


Figure 4-2: Speed up comparison for 4 processors between ITPMMA Cannon algorithms

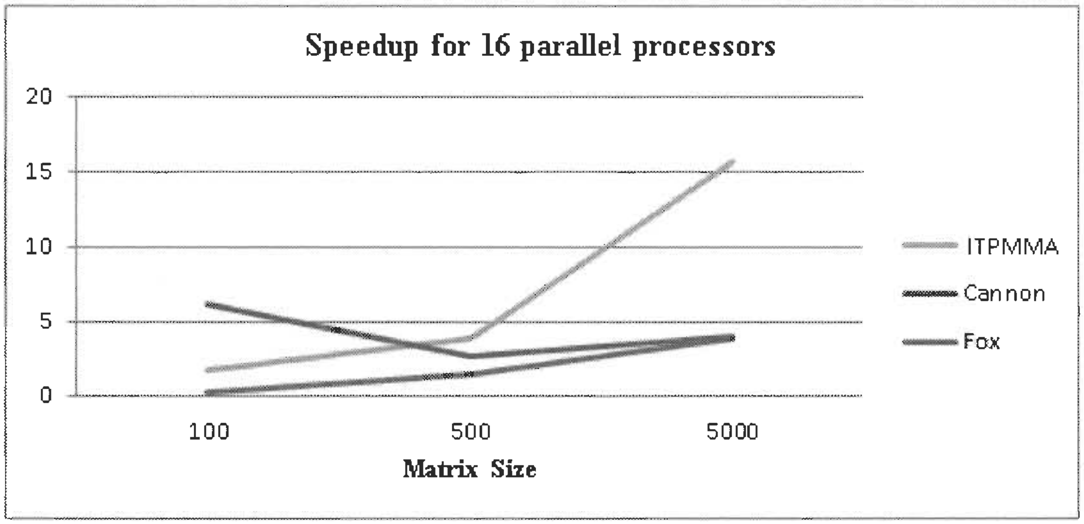


Figure 4-3: Speed up comparison for 16 processors between ITPMMA Cannon algorithms

Another point, the comparison between both algorithms in terms of speedup is varying, which I will return it to the PCS I have used and to the compatibilities between the hardware parts. This point will fade when using CLUMEQ supercomputer, when the compatibility between the hardware parts is high.

4.3. Guillimin Cluster at CLUMEQ supercomputer environment

In this subchapter, I will analyze four groups of experiments being carried out on Guillimin Cluster at CLUMEQ supercomputer. That is to address the speedup, efficiency, load balance and the performance of ITPMMA algorithm, against both Cannon Algorithm, and Fox Algorithm. These four groups are:

1. Several sizes of matrices, where the matrices size is multiple of the number of processors, to address the speedup aspects, Table 4-2.
2. Several sizes of matrices, where the matrices size is not multiple of the number of processors, to address the load balance aspects, Table 4-3 and Table 4-4.
3. Non-square matrices with different number of processors, to address the load balance aspects, and to address the advantages of ITPMMA algorithm over other algorithms in dealing with this case, non-square matrices multiplication, Table 4-5.
4. Fixed sizes of matrices with different number of processors, where the matrices size is multiple of the number of processors, to address the performance, where fewer resources (number of processors) are needed by ITPMMA algorithm less than resources needed by other algorithms to do the same task, Table 4-6.

4.3.1. Speedup calculations

Speed up of a parallel algorithm is the time consumed in execution the parallel algorithm divided by the time consumed in executing the same problem in serial mode. In this test, I have used different sizes matrices and multiplied them in serial first, then in parallel. Parallel execution repeated three times, the first time, I have used ITPMMA algorithm, while the second time I have used Cannon Algorithm, and the third time I have used Fox Algorithm. In this test, I have used the matrices size as multiple of the number of the processors to guarantee equaled distribution of the data blokes on the processors for both Cannon and Fox Algorithms, to guarantee full efficiency of both algorithms, while this is not the case for ITPMMA algorithm to guarantee equaled distribution of the data. These tests show the speed of ITPMMA algorithm over both Cannon and Fox Algorithms. Table 4-2, shows the readings of these tests.

For matrix multiplication of size 32768×32768 and double precision floating point type – each element of the matrix occupy 8 bytes, ITPMMA algorithm needs 1098 seconds to complete this operation over 16 processors. While, Cannon Algorithm needs 6010 seconds, Fox Algorithm needs 7150 seconds, over the same conditions and same number of processors. Full comparison of the time consumed for different matrices size for the different algorithms, and the speed up are shown on Table 4-2. In figure 4-4 speed up of ITPMMA algorithm against both Cannon and Fox Algorithm.

Table 4-2 Different sizes of matrices with 16 processors in parallel, where the matrices size is multiplicand of the number of processors.

Algorithm	Matrix Size							
	128	512	1024	2048	4096	8192	16384	32768
Processing Time								
Serial	40,00	222,00	535,00	998,00	1900,00	3810,00	7900,00	16300,00
ITPMMA	3,00	17,00	40,00	75,00	130,00	251,00	499,00	1098,00
Cannon	11,00	65,00	152,00	320,00	605,00	1400,00	2700,00	6010,00
Fox	14,00	88,00	212,00	430,00	820,00	1701,00	3980,00	7150,00
ITPMMA Task-Adjustment	0	5	112	181	356	455	565	601
Speed up								
ITPMMA	13,33	13,06	13,38	13,31	14,62	15,18	15,83	14,85
Cannon	3,64	3,42	3,52	3,12	3,14	2,72	2,93	2,71
Fox	2,86	2,52	2,52	2,32	2,32	2,24	1,98	2,28

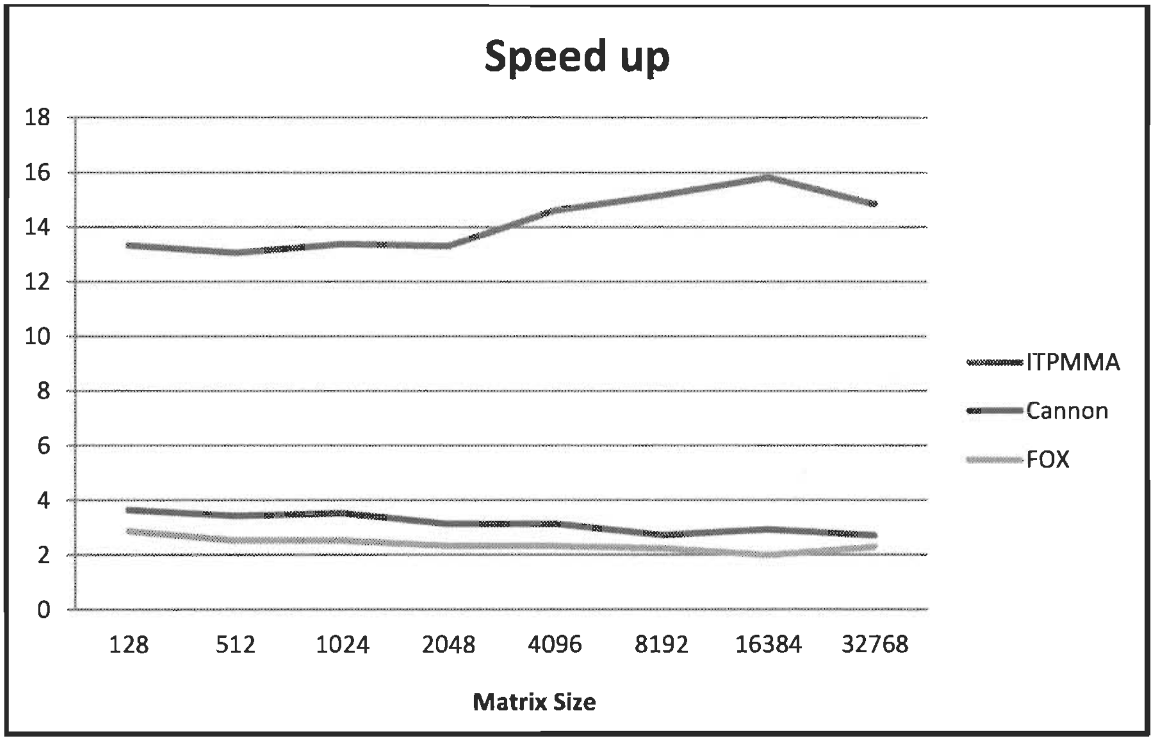


Figure 4-4 ITPMMA Algorithm Speed up against both Cannon Algorithm, and Fox Algorithm where the matrices size is multiplicand of the number of processors

From Table 4-2 we can notice that the processing time at small size matrices are close to each other for all algorithms, then it diverges as the size of the matrices increases. It is quite clear the differences between the results of ITPMMA algorithm against both Cannon

Algorithm, and Fox Algorithm. The resources and the data – the matrices – are identical in this test; it is only the way of processing. For ITPMMA algorithm, each element of the result matrix will be produced by one single processor, while this is not the case for the other two algorithms, where several processors will work in producing each element of the result matrix. To produce one element of the result matrix in case of $A_{32768,\ 32768} \times B_{32768,\ 32768}$, a raw of matrix A to be multiplied by a column of matrix B, element by element and then to sum up the results to produce the result matrix element, that is total of 32,768 multiplication operations and 32,768 addition operations, with total of 65,536 operations, utilizing its advanced capabilities of multicore and different level of cache memories. In the case of ITPMMA algorithm, one single processor will be processing these 65,536 operations, while the other two algorithms, several processors will share these operations as each processor will work on a small block size of matrix A and B, and then, each processor will send its local results to other processor – which could be on hold waiting the results of other processors because of the high data and functionally dependencies of Cannon and FOX algorithms – to complete the processing of several elements of the result matrix once. Sending the local result of each processor to other processor(s) it is time consuming, in addition to that, some processor will complete processing its current block, and then staying on hold till it receives the local result of other processor(s). So I conclude here that the difference between ITPMMA algorithm and Cannon and Fox algorithms are:

1. The time consumed in sending local results to other processors in both Cannon and Fox algorithms increases as number of processors increases and as number of blocks increases, while it is zero in ITPMMA algorithm.

2. The time, where some processors stay on hold waiting local results of other processor(s) to complete its task, or to start new tasks in both Cannon and Fox algorithms fluctuate from time to time and from processor to processor, while it is zero in ITPMMA algorithm.

Finally, the above two advantages of ITPMMA algorithm make it faster than Cannon and Fox algorithms.

4.3.2. Load balance calculations

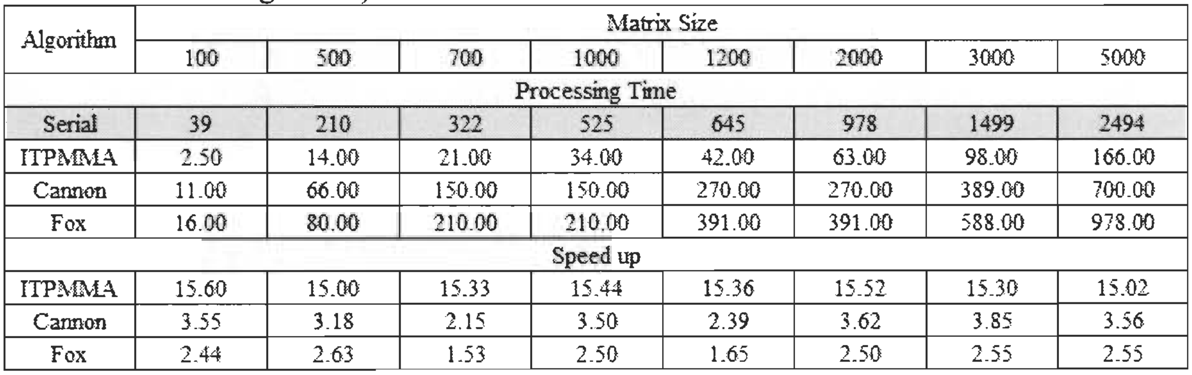
Load balance implies all processors will do the same size of work, for example, same number of same types of operations on same type of data. In heterogeneous parallel environment, the above definition is not fruitful as long as the processors of the environment have different capabilities in term of computing power, nodes, and cache levels and sizes; so we are interested in keeping all processors working, and not turned idle as long the whole operation of matrix multiplication is not completed yet. In this sub chapter, I will consider two groups of experiments:

1. Different sizes matrices being multiplied by different algorithms with same number of processors, where the matrices size is not multiple of the number of processors. In this case, the load of data is not multiple of the number of processors, this issues being solved for both Cannon and Fox Algorithms by padding zeroes to the multiplied matrices, so its size is multiple of the number of processors, so each matrix will be divided in several blocks where each block size is N/p, where N is the size of the matrix and p is the number of the processors in use. This implies multiplying more size matrices – with extra data to be multiplied, while these data is

not needed – in case of Cannon and Fox Algorithms, more than the size of the multiplied matrices in case of ITPMMA algorithm. This experiment will show the vast differences between the time needed to carry out the matrices multiplication of the three algorithms, to show the poorness of the load balance strategy used by both Cannon and Fox Algorithms, which implies padding zeros to have the optimized matrices sizes, which will consume same time as matrix have adjusted size.

2. Non-square matrices with different number of processors, to show much worse case for both Cannon and Fox Algorithms, where the multiplied matrices are not square too, which implies padding more zeroes to adjust the both number of columns and number of rows of the multiplied matrices to be multiple of the number of the processors in use. Again we will see vast difference between the results of ITPMMA algorithm and Cannon and Fox Algorithms. This experiment in addition the previous experiment, shows the absent of load balance concept and strategy at both Cannon and Fox Algorithms, while it is a default concept at ITPMMA algorithm.

Table 4-3 multiplying different sizes of matrices on 16 processors, where the matrices size is not multiplicand of the number of processors (balanced load will be clear with ITPMMA algorithm).



Algorithm	Matrix Size							
	100	500	700	1000	1200	2000	3000	5000
Processing Time								
Serial	39	210	322	525	645	978	1499	2494
ITPMMA	2.50	14.00	21.00	34.00	42.00	63.00	98.00	166.00
Cannon	11.00	66.00	150.00	150.00	270.00	270.00	389.00	700.00
Fox	16.00	80.00	210.00	210.00	391.00	391.00	588.00	978.00
Speed up								
ITPMMA	15.60	15.00	15.33	15.44	15.36	15.52	15.30	15.02
Cannon	3.55	3.18	2.15	3.50	2.39	3.62	3.85	3.56
Fox	2.44	2.63	1.53	2.50	1.65	2.50	2.55	2.55

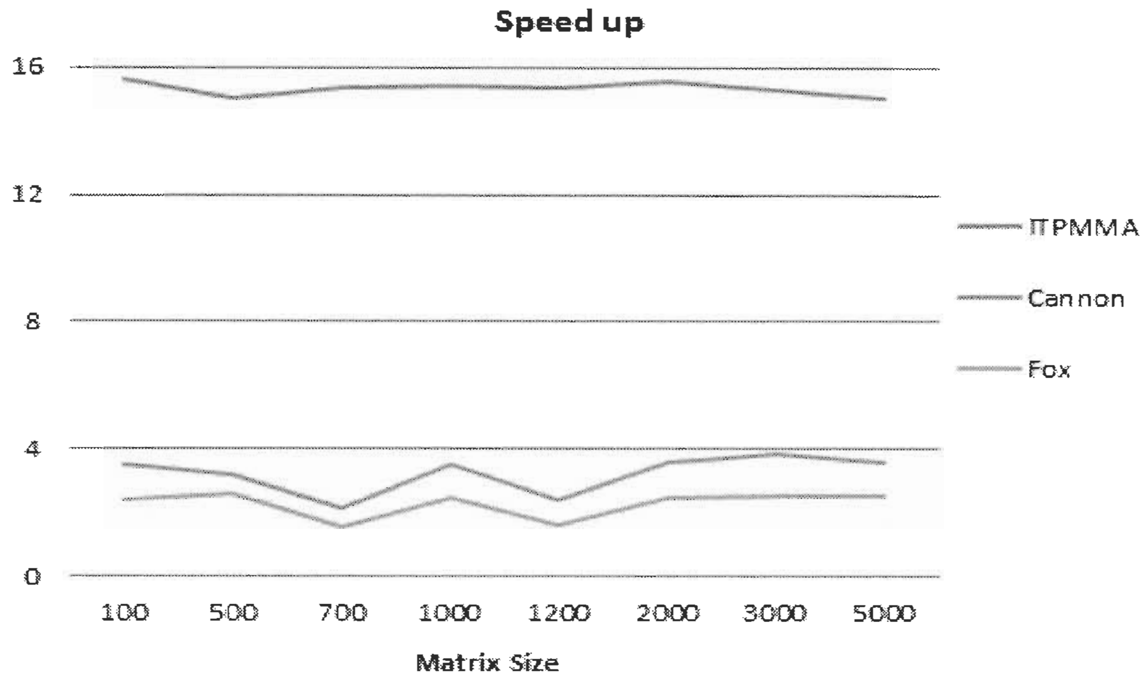


Figure 4-5 Speed up of ITPMMA algorithm against both Cannon Algorithm, and Fox Algorithm where the matrices size is not multiple of the number of processors

The differences between the above tests are so clear concerning time of execution, since some processors in case of Cannon and Fox Algorithms are busy with multiplying extra zeroes data being padded to matrices to fix its sizes, to let the processors accept the blocks have some actual data, while the remain of the block data is just zeroes to complete the block to the defined size. Different block sizes can have better effects for both Cannon and Fox Algorithms. For example for matrices size 100 and number of 16 processors in parallel, to calculate the block size 100/16 = 6.25, so both multiplied matrices to be padded by extra zeroes to be able to generate blocks of 7 size, so the matrix size will be 7 * 16 = 112, so all processors will keep busy all the time of the multiplication operation of matrix size of 112 instead of 100 for both Cannon and fox algorithms but not for ITPMMA algorithm.

Table 4-4 comparison between different size matrices multiplication, where the size of the matrices are multiple and non-multiple of the processors.

Processors = 16								
Algorithm	Matrix Size							
	128	512	1024	2048	4096	8192	16384	32768
ITPMMA	3	17	40	75	130	251	499	1089
Cannon	11	65	152	320	605	1400	2700	6010
Fox	14	88	212	430	820	1701	3980	7150
Algorithm	Matrix Size							
	100	500	700	1000	1200	2000	3000	5000
ITPMMA	2.5	14	21	34	42	63	98	166
Cannon	11	66	150	150	270	270	389	700
Fox	16	80	210	210	391	391	588	978

By having a look at Table 4-4, data being summarized from Table 4-2 and Table 4-3, it so obvious that Cannon algorithm needed 11.0 seconds to multiply matrices of size 128 at 16 processors, which is equal the time needed 11.0 seconds to multiply a smaller matrix size of 100, at the same number of processors. While the case is different when ITPMMA algorithm is used, where 3 seconds where enough to multiply 128 matrices sizes while 2.5 seconds were needed for multiplying 100 matrices sizes at the same number of processors. Similar cases could be noticed at the matrices of size 512 and 500, 1024 and 1000, 2048 and 2000, as shown in Table 4-4 and Figure 4-6.

On the other hand, multiplying matrices of sizes 700 and 1000 consumed 150 seconds by Cannon and 210 seconds by Fox algorithms, while multiplying matrices of sizes 700 consumed only 21 seconds using ITPMMA algorithm, and multiplying matrices of sizes 1000 consumed only 34 seconds, which implies the successful load balance strategy applied by ITPMMA algorithm, Table 4-3, Figure 4-5.

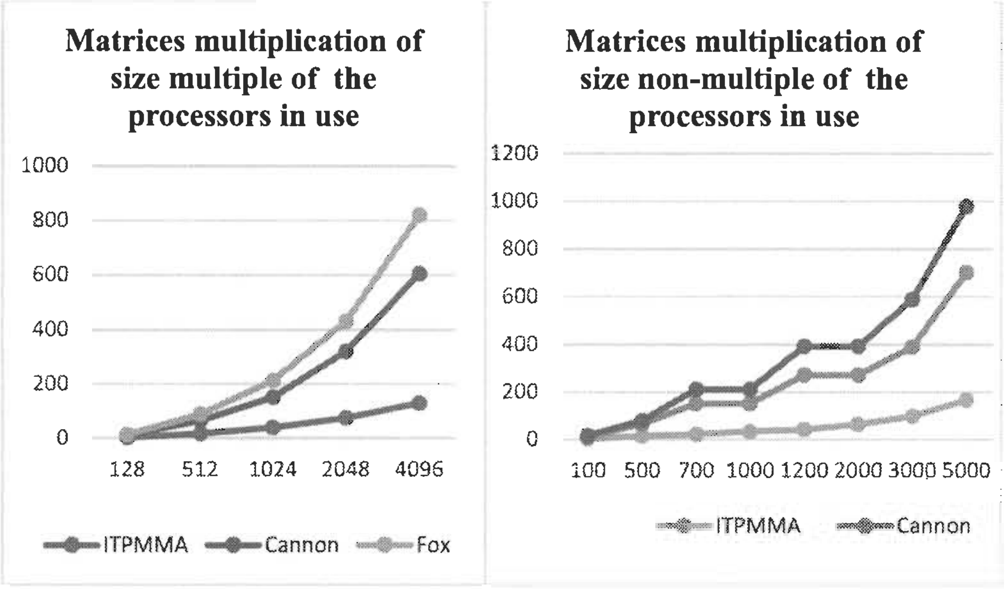


Figure 4-6 Matrices multiplication of size multiple and non-multiple of the processors in use

The third and fourth groups of experiments hold in this sub-chapter are shown in Table 4-5 and Figure 4-7. Here, I have considered more limitation of both Cannon and Fox algorithm for generating the blocks when the matrices in is non-square plus its sizes is not multiple of number of processors in use.

Table 4-5 Non-square matrices – 750 × 700 – with different number of processors

Algorithm	Number of Processors					
	2	4	16	32	64	128
ITPMMA	112	58	30	16	9	5
Cannon	589.00	297.00	152.00	120.00	85.00	65.00
Fox	756.00	400.00	212.00	135.00	92.00	75.00

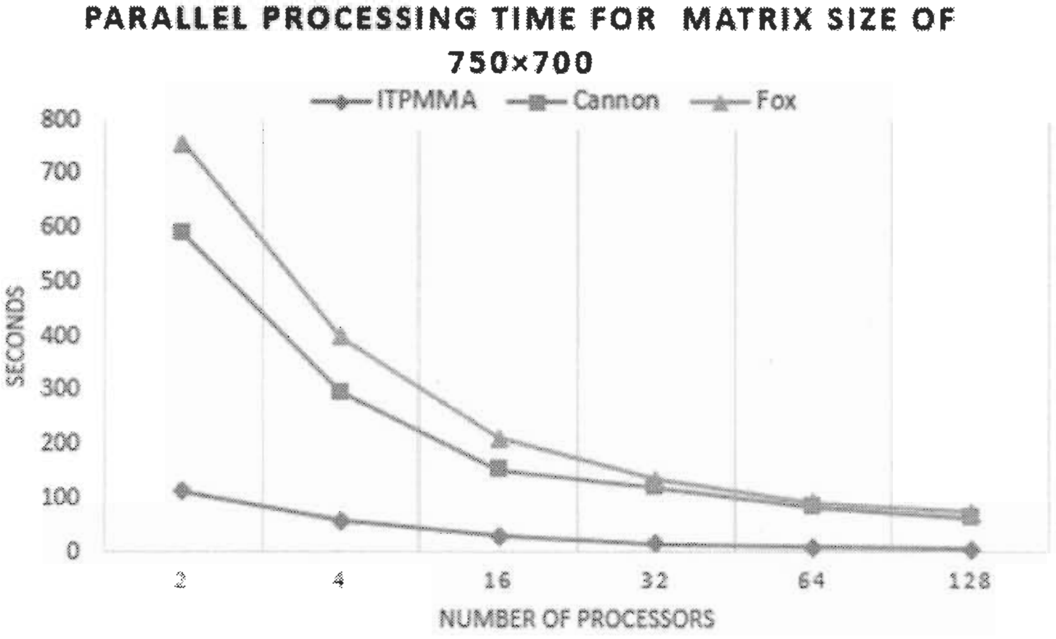


Figure 4-7 Parallel Processing Time ITPMMA algorithm against both Cannon Algorithm, and Fox Algorithm where the matrices size of 750×700 is not multiple of the number of processors

Last two tests show clearly that multiplying two matrices of size 750×700 using Cannon or Fox Algorithms needs same time as multiplying two matrices of size 1024×1024 using the same algorithm. while the result was completely different when using ITPMMA algorithm, where the time needed for multiplying two matrices of size 750×700 is less than the time needed to multiply two matrices of size 1024×1024 by about 4/34=11.7%, while the for both Cannon and Fox algorithms, the time consumed to multiply two matrices of size 1024×1024 on 16 processors is 152 and 212 seconds respectively, which is the same time consumed by both algorithms to multiply two matrices of size 750×700 on 16 processors. The test been conducted for several different numbers of processors, where it is so obvious in figure 4 -8, A, B, C.

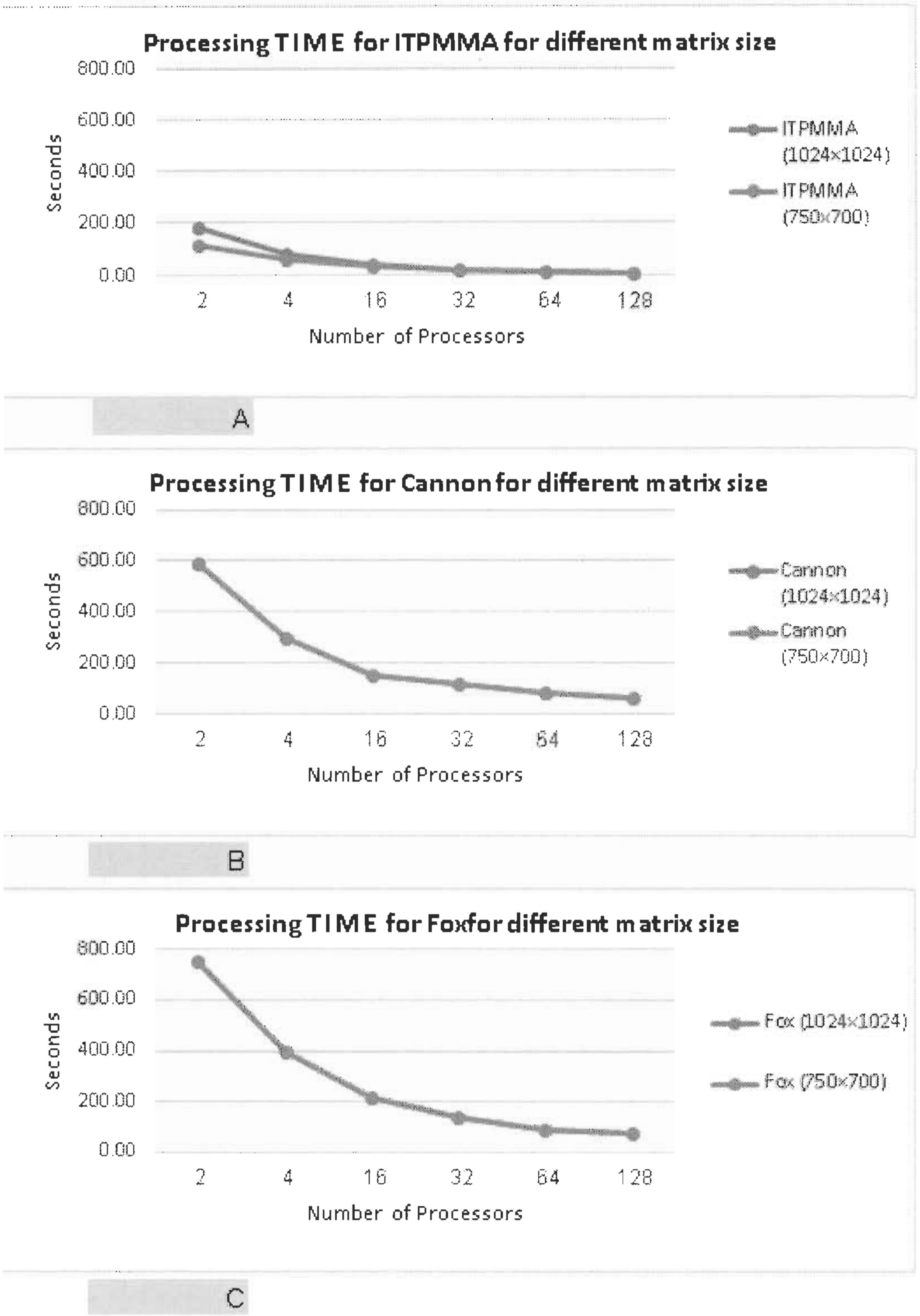


Figure 4-8 Parallel Processing Time for matrices multiplication of two different sizes 750×700, and 1024×1024. A: ITPMMA algorithm, B: Cannon Algorithm, C: Fox Algorithm

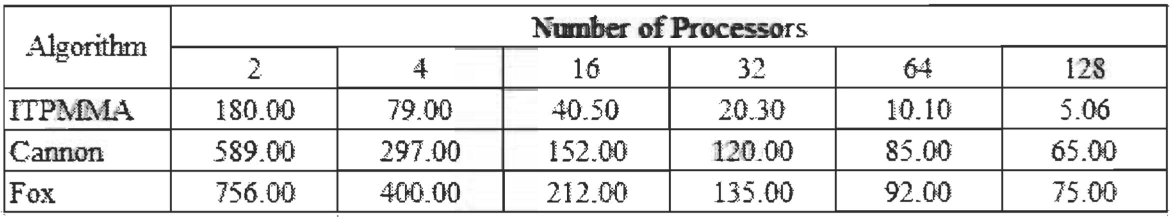
4.3.3. Efficiency Calculations

The efficient implementation of certain algorithm implies the maximum utilization of the resources, which are the processors in parallel processing. In the test below, I have applied same matrix data and size at several numbers of processors for the three algorithms, ITPMMA, Cannon and Fox Algorithms, so the mathematic operations will be same in number and size, since I am using the same matrices. So the different in processing time is result of time consumed in non-common tasks – in both Cannon and Fox algorithms – where it is absent in ITPMMA algorithm, like:

1. Podcasting the data blocks over the parallel environment using MPI library.
2. Communication among processors and sending the local results of each processor to the adjacent processors, so it carries out its tasks.
3. The time that some processors stay on hold as a result of data dependencies, which is fluctuate from case to case.
4. The time that some processors turned idle, as no more tasks to be executed.

After processing the matrix multiplication of matrices of size 1024 using the three algorithms Cannon and Fox and ITPMMA, on different number of processors, time of execution is summarized in Table 4-6, and Figure 4-9.

Table 4-6 Fixed size of matrices with different number of processors, where the matrices size is multiple of the number of processors



Algorithm	Number of Processors					
	2	4	16	32	64	128
ITPMMA	180.00	79.00	40.50	20.30	10.10	5.06
Cannon	589.00	297.00	152.00	120.00	85.00	65.00
Fox	756.00	400.00	212.00	135.00	92.00	75.00

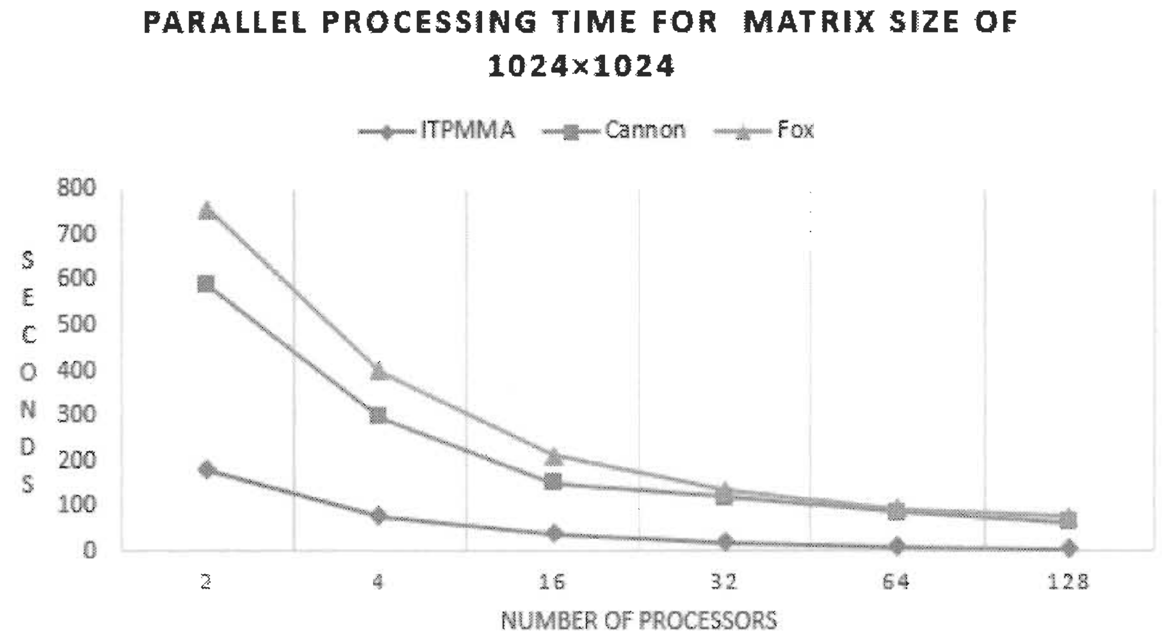


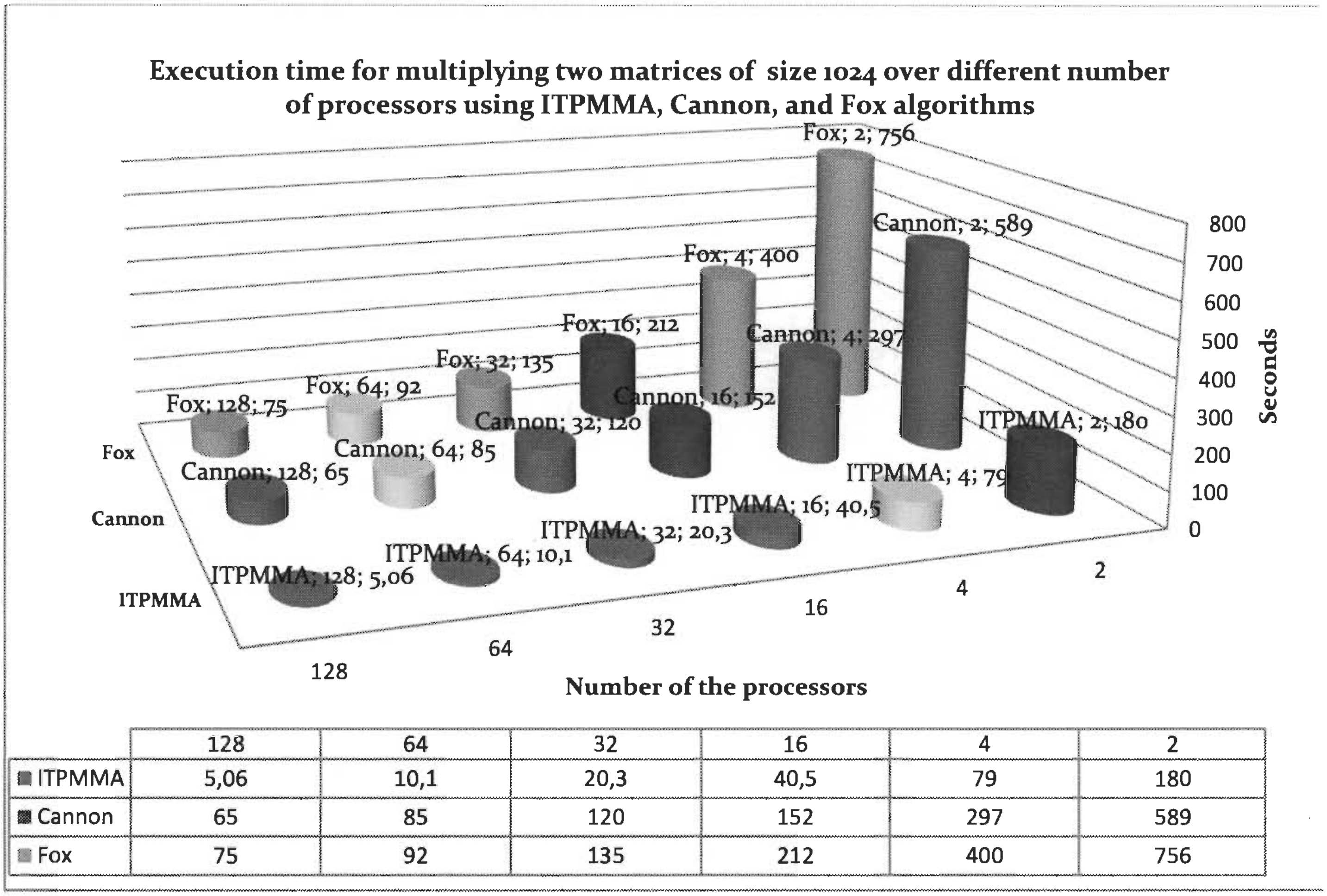
Figure 4-9 Parallel Processing Time for ITPMMA algorithm against both Cannon Algorithm, and Fox Algorithm where the matrices size is multiple of the number of processors for matrices of 1024×1024

In fact, calculating the exact time consumed on non-common tasks listed above was not possible because of the lack of functions provided by the operating system of Guillimin cluster at CLUMEQ super computer.

4.3.4. Performance calculations

In performance calculation, I will compare the performance of ITPMMA algorithm by Cannon and Fox algorithm using the time of execution only. Unfortunately, I do lack for the tools to calculate the size of memory – RAM and Cache - utilized for each execution. Also, I do not have tools at CLUMEQ to calculate the utilization of the processors time, wither as whole or individually. For that, I will consider the start and end time of the whole execution, supposed the same resources – same hardware – have been used in all executions. For example, for multiplying matrices of 1024 size using Cannon Algorithm on

64 processes needs 85 seconds, while same job - multiplying matrices of 1024 size - could be achieved using ITPMMA algorithm on 4 processors at 79 seconds. So I would say that ITPMMA algorithm performance is 64/4= 16 time over Cannon Algorithm, in this case, according to Table 4-6. Figure 4-10 uses bubbles to show the execution time for multiplying matrices of 1024 size using the three algorithms, ITPMMA, Cannon and Fox, at different number of processors that is 2, 4, 16, 32, 64, and 128 processors. It is obvious that the green blue at processor 64 is close to the red bubble at processor 4, which confirms that ITPMMA algorithm needs fewer resources than what Cannon algorithm needs to complete same tasks at same time.



	128	64	32	16	4	2
■ ITPMMA	5,06	10,1	20,3	40,5	79	180
■ Cannon	65	85	120	152	297	589
■ Fox	75	92	135	212	400	756

Figure 4-10 Execution time per processor for ITPMMA, Cannon, and Fox algorithms

The performance of ITPMMA algorithm shown in the last experiment implies its advantages of other two algorithms, Cannon and Fox.

4.4.Conclusion

In this chapter, I have applied several experiments on CLUMEQ supercomputer, to evaluate the speedup, efficiency and the performance of ITPMMA algorithm against very well-known algorithms in parallel matrix multiplication, Cannon and Fox algorithms.

The experiments showed obviously the vast differences between the three algorithms in terms of speedup, efficiency and the performance. In fact, this vast difference return to the fact, that both Cannon and Fox algorithms which being developed about four and half decades ago, both algorithms do not consider the development and the advances in computer architecture which added to the computer different level of cache memories, and different number of cores; also, the vast size of RAMs available nowadays for use, in addition to the new address bus mother boards, which has reached 64 bits. Another basic reason is the communication time between the processors which consume time more than the time for execution the multiplication.

Finally, I would like to state, upon the experiments being hold in this chapter, and upon the results being obtained, that Cannon Algorithm and Fox Algorithm, and other algorithms based on blocking the input data onto smaller blocks, these algorithms which have been developed based on poor computer architecture in term of RAM and single core and absent of cache memories, these algorithms become part of the past. While new era will be open for ITPMMA algorithm, and for algorithms base on defining the problem in term of smaller size problem, to avoid data exchanged between processors and to avoid dependency of some processors, let us call them x processors, on the output of other processors, let us call them y processors, which if y processors fail, the whole algorithm will fail.

Chapter 5 - The clustered 1 Dimension decomposition technique

Implementing all numerical problems as independent tasks is not within hands always. So, within the novel frame work for parallel processing we developed new data decomposition technique, called clustered 1-dimension decomposition technique, to overcome the limitation of different data decomposition techniques which have been utilized by different parallel algorithms, which were the primary idea behind the parallel processing, and to reduce communication time among processors. One dimension and two dimensions' data decomposition techniques dominate since the start of parallel algorithm on 1969. Data decomposition techniques imply performing two tasks:

a. Mapping array of processes into n-dimensional grid.
b. Distributing data over process grid

5.1.Previous work

Over the last few years, a lot of attention was paid to load balancing for linear algebra kernels. One-dimensional data decomposition is used in different applications and computational kernels [66, 67]. Some few customizations for linear algebra problems were proposed in [68, 69].

Kalinov and Lastovetsky [70, 71] proposed extension of two-dimensional data decomposition. Their data decomposition technique has the same disadvantage as the technique of Crandall and Quinn and Kaddoura et al. Barbosa at al [72].

Beaumont at al [69, 73] stressed attention on intercoupling of mapping of processes into 2D grid and data distribution. Crandall and Quinn [74] have developed three-dimensional decomposition technique, which suffers from heavy data dependences.

5.2. One and two dimensions' data decomposition techniques

One and two dimensions' data decomposition techniques dominate on parallel algorithms since 1969. The best way to present these techniques is to solve Laplace Equation using Gauss-Seidel method shown in equation 5-1.

$$f_{i,j}^{n+1} = \frac{1}{4}\left[f_{i+1,j}^{n} + f_{i-1,j}^{n} + f_{i,j+1}^{n} + f_{i,j-1}^{n}\right] \qquad \text{Eq. 5-1}$$

Can be solved using Gauss-Seidel algorithm using the following serial pseudocode

```
repeat until convergence
for i from 1 until n do
     C ← 0
     for j from 1 until n do
if   j ≠ i      then
          C←C+$A_{ij}$*$C_j$
end if
end (j-loop)
Ci←$\frac{1}{A_{ii}}(b_i - c)$
end (i-loop)
check if convergence is reached
end (repeat)
```

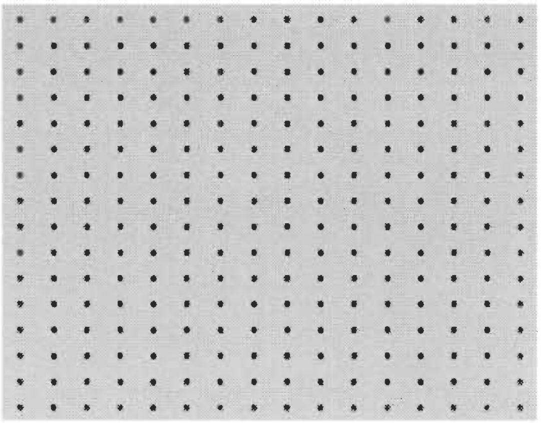


Figure 5-1 Laplace Equation problem presentation, reds are boundary points while blue are interior points

Figure 5-1 present Laplace Equation problem, where red color are the boundary points and it is given, while the blue points to be calculated using equation 5-1. For parallel solution, the matrix of Laplace equation be decomposed and distributed among the processors. For that we have two classic techniques:

a. One-dimension data decomposition, shown in Figure 5-2.

b. Two dimensions' data decomposition, shown in Figure 5-3.

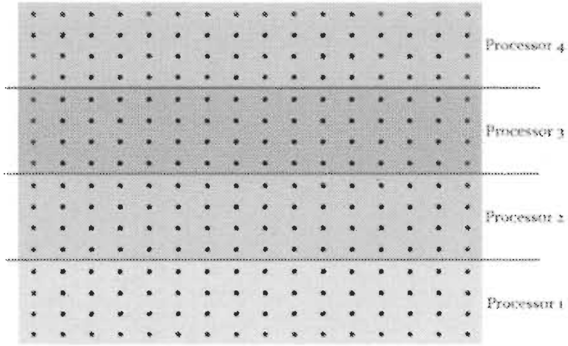


Figure 5-2 1D Decomposition

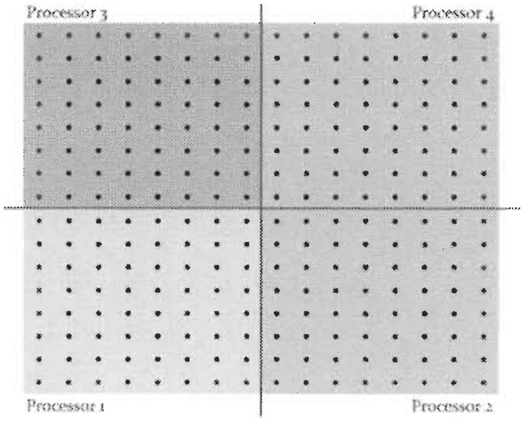


Figure 5-3 2D Decomposition

2D decomposition suggests better performance since processors 2 and 3 work at the same time, so, processor1 starts working, once it is over, processors 2 and 3 start immediately since all input data, including boundary points, are available. Figure 5-4 shows the Data Dependency in 2D decomposition algorithm.

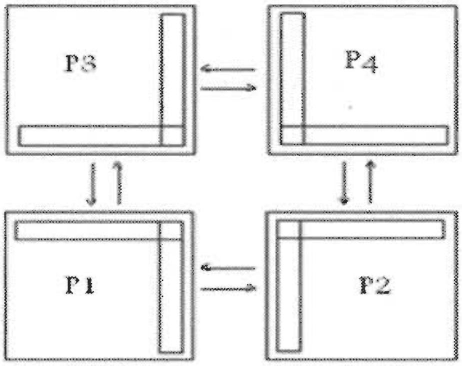


Figure 5-4 Data dependency of 2D Decomposition

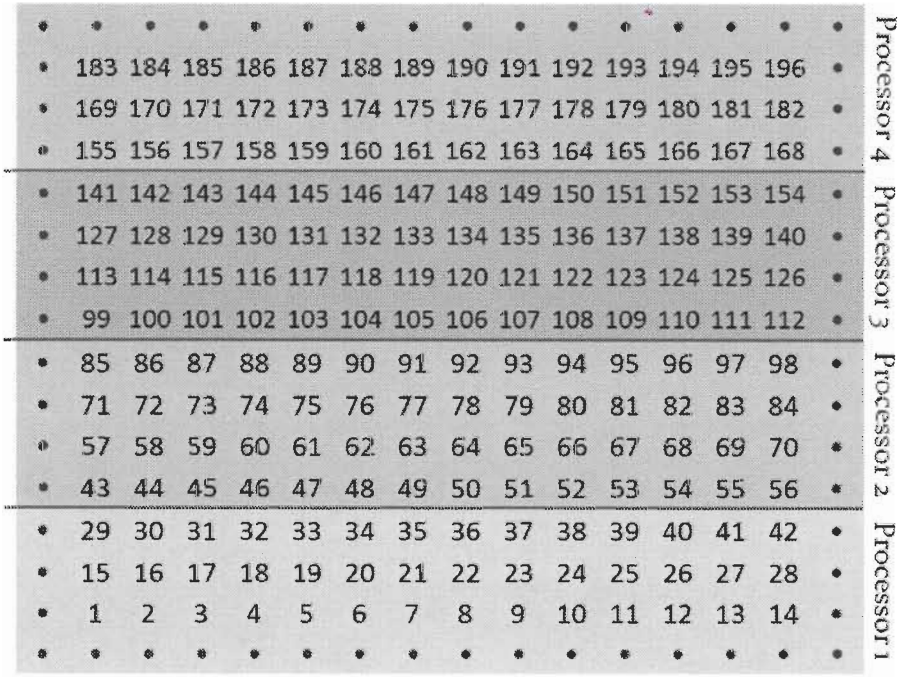
Table 5-1 shows the results of several experiments being conducted on 2 dimensions' algorithm. Still, while processor 1 is working, other three processors are idle, and while processors 2 and 3 are working, processors and 4 are idle, and while processor 1 is working, other three processors are idle, which implies poor resource utilization.

Table 5-1 Parallel Laplace`s Equation Solution using Gauss-Seidel Iterative method on 2D

Teat ID	1	2	3	4	5	6
Matrix Size (n)	**48**	**96**	**192**	**48**	**96**	**192**
Number of processors (nproc=nblock*nblock)	4	4	4	16	16	16
Decomposition (Number of the blocks)	4	4	4	16	16	16
Node edge (nodeedge)	24	48	96	12	24	48
ts (Serial processing time)	1	2	8	1	2	8
tp (Parallel processing time)	0.866	3.192	5.239	30.845	166.303	254.562
Speed up	1.155	0.626	1.527	0.032	0.012	0.031
Efficiency	0.288	0.156	0.381	0.002	7.50E-04	0.002

5.3.Clustered one-dimension data decomposition technique

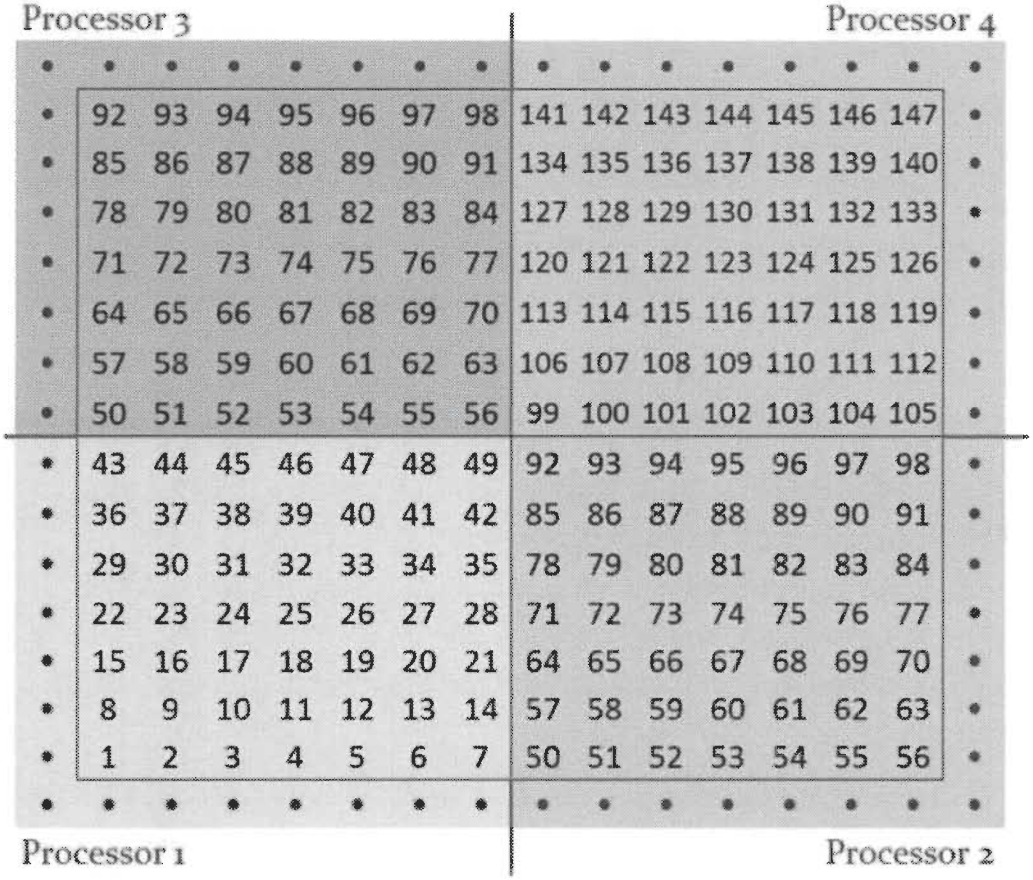
The new decomposition technique guarantees better resources utilization, so idle time of processors will be reduced. In 1D decomposition computing power is %25 since one processor works at a time, and the speed up equals to zero, as shown in Figure 5-5.



•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	Processor 4
•	183	184	185	186	187	188	189	190	191	192	193	194	195	196	•	
•	169	170	171	172	173	174	175	176	177	178	179	180	181	182	•	
•	155	156	157	158	159	160	161	162	163	164	165	166	167	168	•	
•	141	142	143	144	145	146	147	148	149	150	151	152	153	154	•	Processor 3
•	127	128	129	130	131	132	133	134	135	136	137	138	139	140	•	
•	113	114	115	116	117	118	119	120	121	122	123	124	125	126	•	
•	99	100	101	102	103	104	105	106	107	108	109	110	111	112	•	
•	85	86	87	88	89	90	91	92	93	94	95	96	97	98	•	Processor 2
•	71	72	73	74	75	76	77	78	79	80	81	82	83	84	•	
•	57	58	59	60	61	62	63	64	65	66	67	68	69	70	•	
•	43	44	45	46	47	48	49	50	51	52	53	54	55	56	•	
•	29	30	31	32	33	34	35	36	37	38	39	40	41	42	•	Processor 1
•	15	16	17	18	19	20	21	22	23	24	25	26	27	28	•	
•	1	2	3	4	5	6	7	8	9	10	11	12	13	14	•	
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	

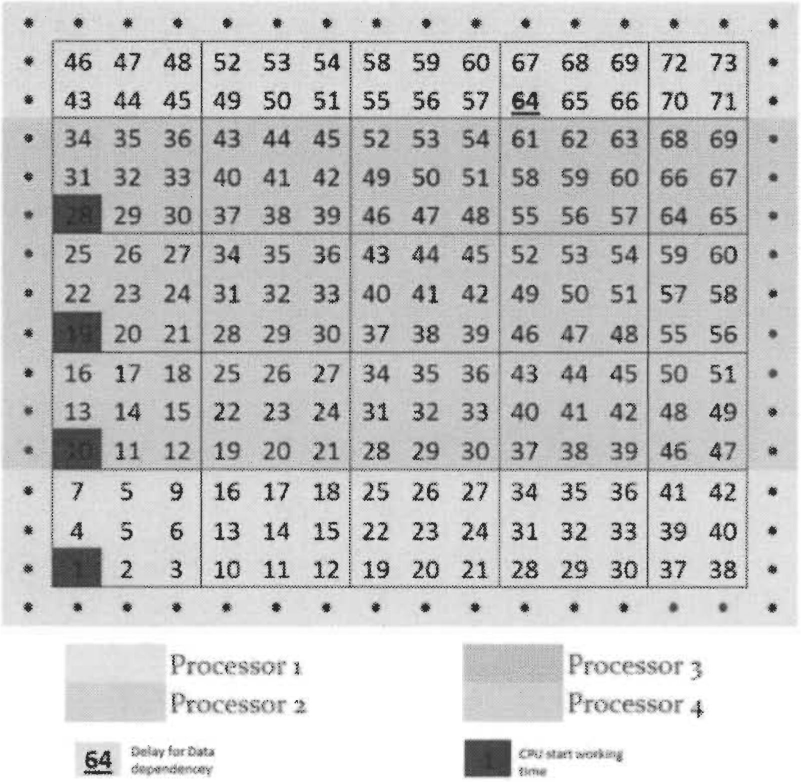
Figure 5-5 1D decomposition technique, total time units is 196 units

While computing power is better in 2D decomposition, since processors 2 and 3 work at same time, the speed up is 1.33, so the job will be completed within %75 of the serial processing time, as shown in figure 5-6. But we should expect 4 speed up since we use 4 processors. In the suggested algorithm, we reached speed up of about 3.5 times. Figure 5-7 shows the data decomposition of the new algorithm, and figure 5-8 shows the processors utilization of the new data decomposition technique.



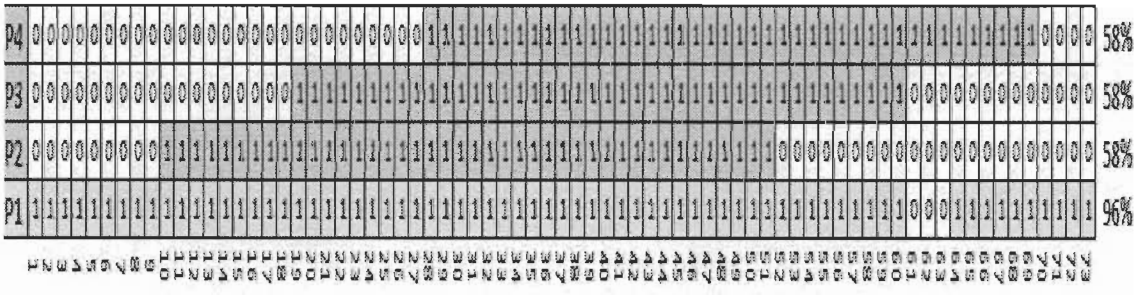
92	93	94	95	96	97	98	141	142	143	144	145	146	147
85	86	87	88	89	90	91	134	135	136	137	138	139	140
78	79	80	81	82	83	84	127	128	129	130	131	132	133
71	72	73	74	75	76	77	120	121	122	123	124	125	126
64	65	66	67	68	69	70	113	114	115	116	117	118	119
57	58	59	60	61	62	63	106	107	108	109	110	111	112
50	51	52	53	54	55	56	99	100	101	102	103	104	105
43	44	45	46	47	48	49	92	93	94	95	96	97	98
36	37	38	39	40	41	42	85	86	87	88	89	90	91
29	30	31	32	33	34	35	78	79	80	81	82	83	84
22	23	24	25	26	27	28	71	72	73	74	75	76	77
15	16	17	18	19	20	21	64	65	66	67	68	69	70
8	9	10	11	12	13	14	57	58	59	60	61	62	63
1	2	3	4	5	6	7	50	51	52	53	54	55	56

Figure 5-6 2D decomposition technique, total time units is 147 units



•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
•	46	47	48	52	53	54	58	59	60	67	68	69	72	73	•
•	43	44	45	49	50	51	55	56	57	64	65	66	70	71	•
•	34	35	36	43	44	45	52	53	54	61	62	63	68	69	•
•	31	32	33	40	41	42	49	50	51	58	59	60	66	67	•
•	28	29	30	37	38	39	46	47	48	55	56	57	64	65	•
•	25	26	27	34	35	36	43	44	45	52	53	54	59	60	•
•	22	23	24	31	32	33	40	41	42	49	50	51	57	58	•
•	19	20	21	28	29	30	37	38	39	46	47	48	55	56	•
•	16	17	18	25	26	27	34	35	36	43	44	45	50	51	•
•	13	14	15	22	23	24	31	32	33	40	41	42	48	49	•
•	20	11	12	19	20	21	28	29	30	37	38	39	46	47	•
•	7	5	9	16	17	18	25	26	27	34	35	36	41	42	•
•	4	5	6	13	14	15	22	23	24	31	32	33	39	40	•
•	1	2	3	10	11	12	19	20	21	28	29	30	37	38	•
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•

Figure 5-7 Clustered 1D decomposition technique, total time units is 73 units



P4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	58%
P3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	58%
P2	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	58%
P1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	96%
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	

Figure 5-8 Clustered 1D decomposition processors utilization

5.4.Analytical Analysis

For 1D decomposition, each process holds $n \times n/p$ sub-grid, while for 2D decomposition each process holds $n/\sqrt{p} \times n/\sqrt{p}$ sub-grid. For clustered 1D decomposition each process holds $n \times c$, repeatedly, where c is the cluster size. In fig 4, c equals to 3.

Sequential time $T_s = (N - 2)^2 \times t_{calc}$, where t_{calc} equals the time to calculate the value of one element in the grid.

5.5.Experimental Results

Several experiments have been carried out on the clustered 1-dimension data decomposing the same tests being carried out into 2 dimensions data decomposition technique. The results are shown in the table below:

Table 5-2 Parallel Laplace`s Equation Solution using Gauss-Seidel Iterative method on Clustered1D

Teat ID	7	8	9	10	11	12
Matrix Size (n)	**48**	**96**	**192**	**48**	**96**	**192**
Number of processors (nproc=nblock*nblock)	**4**	**4**	**4**	**16**	**16**	**16**
Decomposition (Number of the blocks)	**48/12=4**	**96/12=8**	**192/12=16**	**48/12=4**	**96/12=8**	**192/12=16**
Node edge (nodeedge)	96	384	1536	24	96	384
ts (Serial processing time)	1	2	8	1	2	8
tp (Parallel processing time)	0.611	1.076	3.61	0.204	0.448	1.569
Speed up	1.637	1.858	2.216	4.902	4.464	5.099
Efficiency	0.409	0.465	0.554	0.306	0.279	0.319

Figure 5-9 shows the speedup and efficiency of parallel gauss-seidel iterative solution of Laplace`s equation in 2 dimensions, where the speedup and efficiency drop when matrix size increased to 96 for the half, and raised up when matrix size increased to 192. Figure 5-10 shows the same parameters of speedup and efficiency of parallel gauss-seidel iterative solution of Laplace`s equation in clustered 1-dimension data decomposition, where we can see the advantages of the in clustered 1 dimension data decomposition technique. Figure 5-11 and 5-12 compare the speed up of the parallel gauss-seidel iterative solution of Laplace`s equation over both data decomposition techniques, on 4 processors and 16 processors respectively.

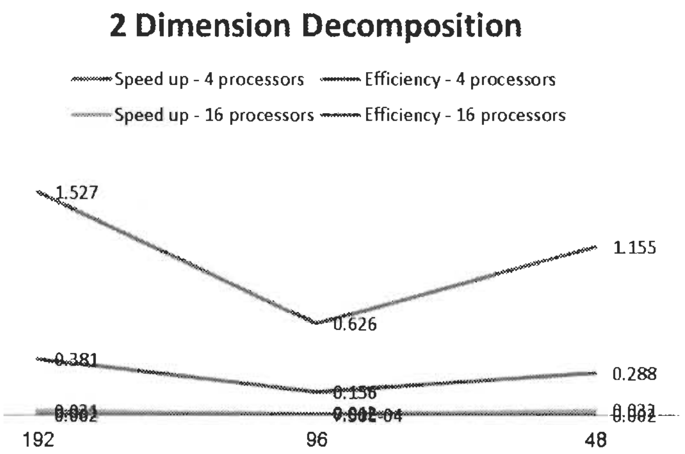


Figure 5-9 Speed up and efficiency of parallel gauss-seidel iterative solution of Laplace`s equation in 2 dimentions

Better speedup has been achieved. Speed up of 42%, 296%, and 180% at 4 processors for matrix size of 48, 96, 192 respectively, as shown in figure 11. While at 16 processors, better speedup has been achieved, it was about 5 using clustered 1-dimension data decomposition, while there was no any speed up using 2 dimensions data decomposition, where the serial algorithm is faster than the parallel algorithm, as shown in figure 12.

It is noticed that the speed up at clustered 1 dimension data decomposition using 16 processors at 96 matrix size is less than in case of matrix size is 48 or 192, which is a direct result of more overhead communication between the processors, where it has been reduced when the matrix size is more huge in size, which implies the scalability of the parallel gauss-seidel iterative solution of Laplace`s equation in clustered 1 dimension data decomposition.

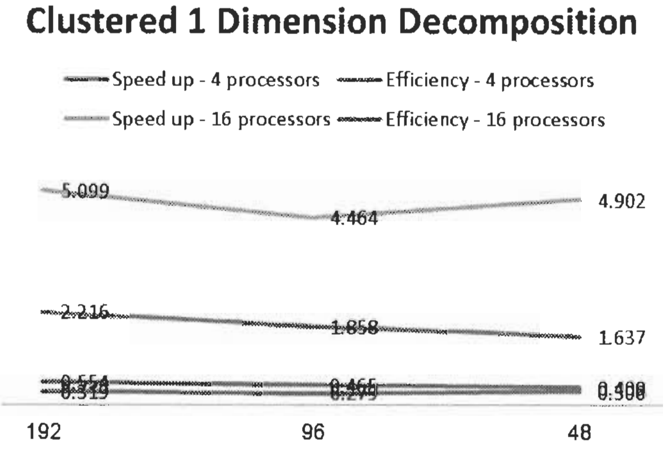


Figure 5-10 Speed up and efficiency of parallel gauss-seidel iterative solution of Laplace`s equation in clustered 1 dimensional decomposition

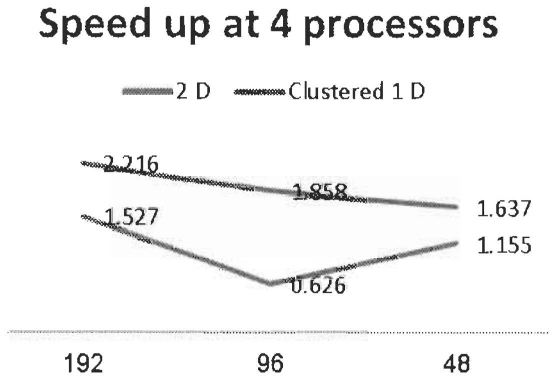


Figure 5-11 Speed up of parallel gauss-seidel iterative solution of Laplace`s equation in both data decomposition techniques at 4 processors

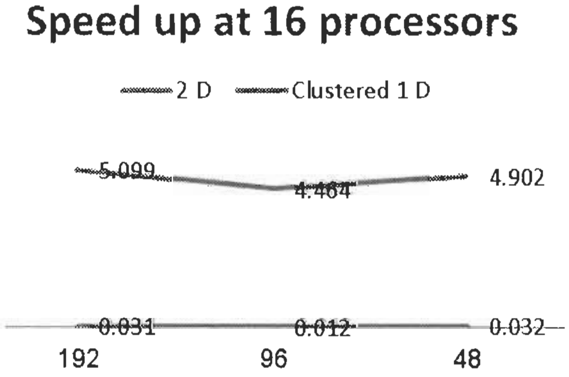
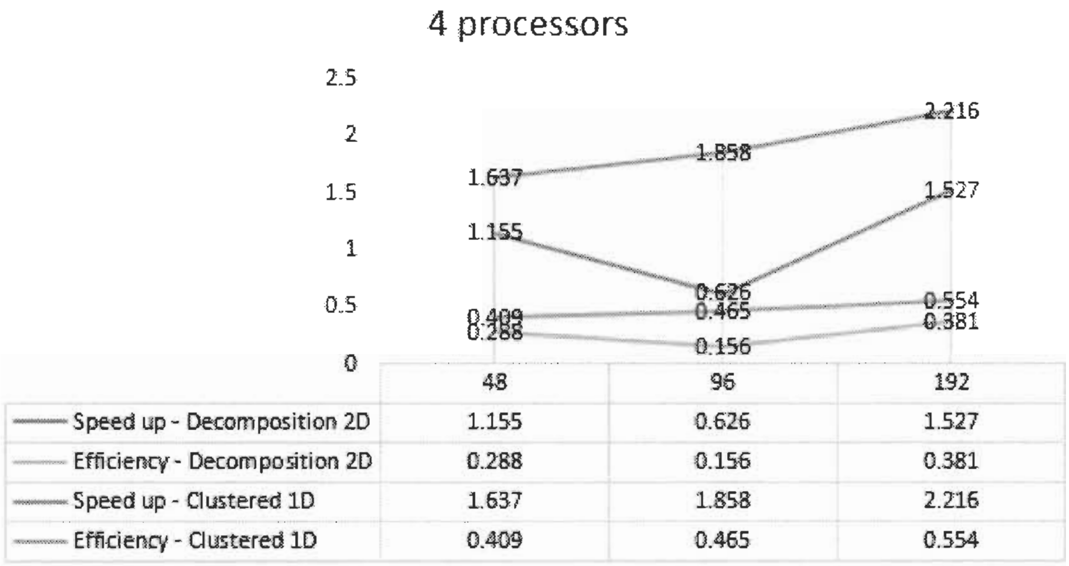


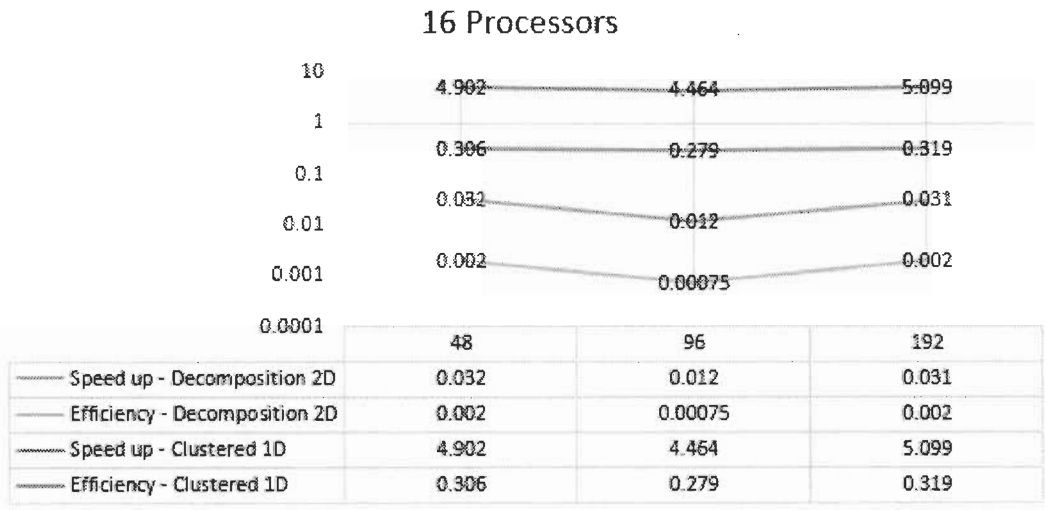
Figure 5-12 Speed up of parallel gauss-seidel iterative solution of Laplace`s equation in both data decomposition techniques at 16 processors

Figure 5-13 and 5-14 show comparison of the speed up and efficiency of parallel gauss-seidel iterative solution of Laplace`s equation in clustered 1D vs Decomposition 2D, for four and sixteen processers respectively.



	48	96	192
—— Speed up - Decomposition 2D	1.155	0.626	1.527
—— Efficiency - Decomposition 2D	0.288	0.156	0.381
—— Speed up - Clustered 1D	1.637	1.858	2.216
—— Efficiency - Clustered 1D	0.409	0.465	0.554

Figure 5-13 Speed up and efficiency of parallel gauss-seidel iterative solution of Laplace`s equation in clustered 1D vs Decomposition 2D, for four processers



	48	96	192
—— Speed up - Decomposition 2D	0.032	0.012	0.031
—— Efficiency - Decomposition 2D	0.002	0.00075	0.002
—— Speed up - Clustered 1D	4.902	4.464	5.099
—— Efficiency - Clustered 1D	0.306	0.279	0.319

Figure 5-14 Speed up and efficiency of parallel gauss-seidel iterative solution of Laplace`s equation in clustered 1D vs Decomposition 2D, for sixteen processers

Finally, I have carried out the tests for very large matrices sizes on 128 processors, where we have used matrices of size 1024, 4096, 16384, 65536, 262,144 and 1,048,576. The tests been carried out using both algorithms Decomposition 2D and clustered 1D, as shown in Figure 17 and 18 respectively, while figure 19 shows the comparison between them, where the speed up vary from 11 times to 197 times.

Table 5-3 Speed up and efficiency of parallel gauss-seidel iterative solution of Laplace`s equation in Decomposition 2D, for 128 processers, for very large matrices sizes

Test ID	13	14	15	16	17	18
Algorithm	Decomposition 2D					
Matrix Size	1024	4096	16384	65536	262144	1048576
Number of processors	128	128	128	128	128	128
Decomposition (number of blocks)	1024/8=128	512	2048	8192	32768	131072
Node edge						
Ts (serial processing time)	55	229	1647	NA	NA	NA
Tp (Parallel processing time)	8	101	135	25847	NA	NA
Speed up	6.77	2.28	12.2	NA	NA	NA
Efficiency	0,054	0,018	0,095	NA	NA	NA

Table 5-4 Speed up and efficiency of parallel gauss-seidel iterative solution of Laplace`s equation in Clustered 1D, for 128 processers, for very large matrices sizes

Test ID	19	20	21	22	23	24
Algorithm	Clustered 1D					
Matrix Size	1024	4096	16384	65536	262144	1048576
Number of processors	128	128	128	128	128	128
Decomposition (number of blocks)	1024/8=128	512	2048	8192	32768	131072
Node edge	16	64	256	1024	4096	16384
Ts (serial processing time)	55	229	1647	NA	NA	NA
Tp (Parallel processing time)	0.75	23	66	131	399	958

Speed up	73	10	25	NA	NA	NA
Efficiency	0.57	0.07	0.19	NA	NA	NA

Table 5-5 Speed up comparison for clustered 1D vs Decomposition 2D for parallel gauss-seidel iterative solution of Laplace`s equation in, for 128 processers, at very large matrices size

Matrix Size	1024	4096	16384	65536	262144	1048576
Tp (Decomposition 2D parallel processing time	8	101	135	25847	NA	NA
Tp (Clustered 1D parallel processing time	0.75	23	66	131	NA	NA
Speed up of clustered 1D over Decomposition 2D	11	4	2	197	NA	NA

It is obvious that clustered 1D is scalable compared with Decomposition 2D technique.

5.6.Conclusion

The advantages of clustered 1 dimensional decomposition technique over 2D decomposition are quite obvious. It reduces the communication among the processors, on the other hand, it utilizes the processors time better, which results in higher efficiency than before. The speed up of clustered 1 dimensional decomposition ranges between 1.5 and 5 while efficiency ranges 30% and 55%. For 2D decomposition, the speed up ranges between 0.012 and 1.527 while efficiency ranges 0.075 % and 38.1%.

On the other hand, the scalability of clustered 1D decomposition technique for carrying the calculation at very large matrices size exceeded 1 million elements, is very obvious; 2D decomposition algorithm technique could not carry out the calculations and there were no results.

Chapter 6 - Conclusion and Recommendation

The factors to be considered when evaluate parallel algorithms are:

2. Accuracy, where the result of the serial and parallel are same.
3. Efficiency, which implies the full successful utilization of the resources.
4. Stability for different type and size of resources.
5. Portability, which is hardware independent.
6. Maintainability, where the algorithm can solve different sizes of the problem at different number of resources.

To gain high parallel efficiency, three figures must be minimized:

1. Communication costs.
2. Load imbalance level.
3. Dependency level.

ITPMMA could minimize these three figures by redefining the problem in terms of several independent problems, as much as we can, keeping the three figure at lowest, so we grantee full successful utilization of the resources.

The term defining the problem in terms of several independent problems is requiring full understand of the hardware resources, which include the amount of cache memory available , size of RAM in use, motherboard address system, 32 or 64 bits, number of cores in the processor. Also, the operating system methodology for managing the users' tasks is important to manage distributing cost of the data to the different processors.

In this thesis, several parallel matrix multiplication algorithms have been studied in details, and an advanced algorithm has been developed to address the drawbacks of the existing algorithms and address the advances in the modern processors architecture, which use multi cores per processor, plus the advances in the availability of several levels of cache memory, on one single chip.

In summary, I have addressed four essential issues in the design of parallel independent subtask algorithms:

1. Reform the problem in terms of independent tasks, to reduce the communication time to zero.
2. Load balancing.
3. Efficiency of the processors.
4. Compatibility with both homogenous and heterogeneous environment of processors, so the algorithm does not require certain amount of cache memory for example, or any certain hardware requirement.

On the other hand, for numerical problems that we could not redefine it in term of independent tasks, new data decomposition techniques has been developed to minimize the three figures of:

1. Communication costs.
2. Load imbalance level.

3. Dependency level.

6.1.Future Research Directions

Large-scale computing clusters of parallel heterogeneous nodes equipped with multi-core processing units are getting increasingly popular in the scientific community as well as in commercial community since it provides mainframe computing power at low price. To advance the developing of parallel programming algorithms, a lot of work to be done in this filed, and this could be summarized as following:

1. Common intermediate results to be executed at certain server and to be broadcasted. The communication between processors are eliminated in ITPMMA algorithm, while the intermediate results, which was material of exchange messages, will be generated locally at each processor needs any, as the time consumed in generating these intermediate result is less, and cannot be compared by the time of communication, especially in light of the high in-cache memory available with modern processors. But still, there is enough room to analyze this issue and study it, so the intermediate results to be utilized and to be transferred to the processors that it need it, while these processors are busy executing another operation. I have utilized intermediate results in solving Laplace equation using Jacobi iteration – which is shown in chapter 5.
2. To consider the new architecture of the dual cores, and quad cores and multi cores. When looking at Strassen's matrix multiplication [52], Strassen achieved success in his algorithm by replacing computationally expensive MMs with matrix additions (MAs). For architectures with simple memory hierarchies, having fewer operations directly translates into an efficient utilization of the CPU and, thus, faster execution. However,

for modern architectures with complex memory hierarchies, the operations introduced by the MAs have a limited in-cache data reuse and thus poor memory-hierarchy utilization, thereby overshadowing the (improved) CPU utilization, and making Strassen's algorithm (largely) useless on its own. Well, we do need to consider the new modern architectures with complex memory hierarchies, as there is new approach recently started recently targeting new processors architecture as in [44, 46, 75]

3. Memory allocation for huge matrices. Another issue to be addressed in matrices multiplication and any other algorithms deal with massive amount of data. This issue is being addressed in [76]. In fact, a lot of work to be done in this field, when we take the architecture of the modern processors, which has a lot of in-cache memory.

4. Data flow and data decomposition techniques to be advanced, by following backward view strategy that is to look at the result of the numerical problem and to divide it to smaller tasks.

5. Finally, this topic, parallel processing is not limited for matrix multiplication, it is being used heavily in so many classic field as mentioned in the introduction; in fact, it extended recently on the world of database, where billions of records to be manipulated, and it is so compatible with independent tasks.

References

[1] S. Gill, "Parallel Programming," *The Computer Journal*, vol. 1, no. 1, pp. 2–10, 1958.

[2] C. Geoffrey, R. Williams, and P. Messina, Parallel Computing Works. San Francisco, CA: Morgan Kaufmann Inc., 1994.

[3] K. Pingali, "Parallel programming languages," tech. rep., Cornell University, 1998.

[4] P. Cockshott and K. Renfrew, SIMD programming for Windows and Linux. Springer, 2004.

[5] A. Scheinine, "Introduction to Parallel Programming Concepts," tech. rep., Louisiana State University, 2009.

[6] K. Asanovic and et la., "A view of the parallel computing landscape," Commun. ACM, 52(10), pp. 56–67, 2009.

[7] P. Mckenney, M. Gupta, M. Michael, P. Howard, J. Triplett, and Walpole, "Is Parallel Programming Hard, and if so, why?," tech. rep., Portland State University, 2009.

[8] G. Tournavitis, Z. Wang, B. Franke, and B. M., "Towards a Holistic Approach to Auto-Parallelization," in ACM, PLDI09, October 2009.

[9] R. Leupers, "Code selection for media processors with simd instructions," Proceedings of the conference on Design, automation and test i Europe, (New York, NY), pp. 4–8, ACM Press, 2000.

[10] F. Luebke and G. Humphreys, "How gpus work," IEEE Computing, pp. 126–130, 2007.

[11] M. Hassaballah, S. Omran, and Y. Mahdy, "A review of SIMD multimedia extensions and their usage in scientific and engineering applications," Comput. J. 51, pp. 630–649, 2008.

[12] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate gpu vs.cpu performance without the answer," International Symposium on Performance Analysis of Systems and Software (ISPASS)), 2011.

[13] A. Krall and S. Lelait, "Compilation techniques for multimedia processors," International Journal of Parallel Programming, vol. 28, no. 4, pp. 347–361, 2000.

[14] N. Srereman and G. Govindarjan, “A vectorising compiler for multiimedia extensions,” vol. 28, pp. 363–400, 2000.

[15] A. Richards, “The Codeplay Sieve C++ Parallel Programming System,” 2006.

[16] NVIDIA Tesla K-Series, Data sheet, Feb 16, 2016, link: http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf

[17] A. Aho, J. Hopcroft, J. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Massachusetts, 1974.

[18] G. H. Golub, C. F. Van Loan, Matrix Computation, Second Edition, The John Hopkins University Press, Baltimore, Maryland, 1989.

[19] D. G. Luenberger, Linear and Nonlinear Programming, Second Ed., Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, USA, 1984.

[20] J. Rice, Matrix Computations and Mathematical Software, McGraw-Hill, New York, 1981.

[21] J. Rice, Numerical Methods, Software, and Analysis, 2nd. Ed., Academic Press, San Diego, 1992.

[22] Mathur, K. and S.L. Johnsson, 0000. Multiplication of Matrices of Arbitrary Shape on a Data Parallel Computer, Parallel Computing, 20: 919-952.

[23] L. E. Cannon, A cellular computer to implement the Kalman Filter Algorithm, Technical report, Ph.D. Thesis, Montana State University, 14 July 1969.

[24] Fox, G., S. Otto and A. Hey, 1987. Matrix algorithms on a hypercube I: matrix multiplication,' Parallel Computing, 3: 17-31.

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

[26] Van De Geijin, R. and J. Watts, 1995. SUMMA: Scalable Universal Matrix Multiplication Algorithm LAPACK Working Note 99, Technical Report CS-95- 286, University of Tennessee.

[27] Jaeyoung Choi, A New Parallel Matrix Multiplication Algorithm on Distributed – Memeory Concurrent Computers. High Performance Computing and Grid in Asia Pacific Region, International Conference on (1997) ISBN: 0-8186-7901-8, pp: 224

[28] ScaLAPACK Users' Guide, Authors: L. S. Blackford , J. Choi , A. Cleary , E. D'Azevedo , J. Demmel , I. Dhillon , J. Dongarra , S. Hammarling , G. Henry , A. Petitet , K. Stanley , D. Walker and R. C. Whaley, Published: 1997, ISBN: 978-0-89871-400-5, eISBN: 978-0-89871-964-2, Book Code: SE04, Series: Software, Environments and Tools,

Pages: xxvi + 319, DOI: http://dx.doi.org/10.1137/1.9780898719642

[29] Ravi Reddy, Alexey Lastovetsky, "HeteroMPI+ScaLAPACK: Towards a ScaLAPACK (Dense Linear Solvers) on Heterogeneous Networks of Computers", Chapter: High Performance Computing - HiPC 2006, Volume 4297 of the series Lecture Notes in Computer Science pp 242-253

[30] Ardavan Pedram, Masoud Daneshtalab and Sied Mehdi Fakhraie, 2006. An Efficient Parallel Architecture for Matrix Computations, 1-4244-0772- 9/06 ©2006 IEEE.

[31] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin and Katherine Yelick, 2008. Avoiding Communication in Sparse Matrix Computations, 978-1-4244-1694-3/08 ©2008 IEEE.

[32] Zhaoquan Cai and Wenhong Wei, 2008. General Parallel Matrix Multiplication on the OTIS Network, 978-0-7695-3122-9/08 © 2008 IEEE.

[33] Ioannis Sotiropoulos and Ioannis Papaefstathiou, 2009. A fast parallel matrix multiplication reconfigurable unit utilized in face recognitions system, 978-1-4244-3892-1/09/ ©2009 IEEE.

[34] Nathalie Revol and Philippe Théveny, 2012, “Parallel Implementation of Interval Matrix Multiplication” Reliable Computing 19, 1 (2013) 91-106.

[35] Jian-Hua Zheng, Liang-Jie Zhang, Rong Zhu, Ke Ning1, and Dong Liu, Parallel Matrix Multiplication Algorithm Based on Vector Linear Combination Using MapReduce, 2013 IEEE Ninth World Congress on Services, 978-0-7695-5024-4/13 $26.00 © 2013 IEEE, DOI 10.1109/SERVICES.2013.67

[36] Ashley DeFlumere, Alexey Lastovetsky, Searching for the Optimal Data Partitioning Shape for Parallel Matrix Matrix Multiplication on 3 Heterogenous Processors, 2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops, 978-1-4799-4116-2/14 $31.00 © 2014 IEEE, DOI 10.1109/IPDPSW.2014.8

[37] Khalid Hasanov, Jean-Noël Quintin, Alexey Lastovetsky, "Hierarchical approach to optimization of parallel matrix multiplication on large-scale platforms", J Supercomput, DOI 10.1007/s11227-014-1133-x, © Springer Science+Business Media New York 2014

[38] Tania Malik, Vladimir Rychkov, Alexey Lastovetsky, Jean-Noel Quintin, Topology-aware Optimization of Communications for Parallel Matrix Multiplication on Hierarchical Heterogeneous HPC Platforms, 2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops, 978-1-4799-4116-2/14 $31.00 © 2014 IEEE, DOI 10.1109/IPDPSW.2014.10

[39] http://www.intel.com/content/www/us/en/benchmarks/workstation/xeon-e5-2600-v2/xeon-e5-2600-v2-spec-linpack-stream.html] Accessed at 24 Dec 2013.

[40] DON COI'PERSMITtt and SIIMUEI, WINOGRAD. J. Symbolic Computation (1990) 9, 251-280.

[41] Andrew James Stothers, On the Complexity of Matrix Multiplication, Doctor of Philosophy, University of Edinburgh, 2010.

[42] Virginia Vassilevska Williams, Breaking the Coppersmith-Winograd barrier, UC Berkeley and Stanford University, 2011.

[43] I. HEDTKE, "Strassen's Matrix Multiplication Algorithm for matrices of arbitrary order" NUMERISCHE MATHEMATIK May 2011.

[44] M. A. Ismail, S. H. Mirza, Talat Altaf, "Concurrent Matrix Multiplication on Multi-Core Processors" International Journal of Computer Science and Security (IJCSS), Volume (5): Issue (2) pp.: 208-220 2011

[45] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Graph expansion and communication costs of fast matrix multiplication". In SPAA '11: Proceedings of the 23rd annual symposium on parallelism in algorithms and architectures (New York, NY, USA, 2011), ACM, pp. 1-12.

[46] S. H. Fuller, and L. Millett, I. Eds. " The Future of Computing Performance: Game Over or Next Level?", The National Academies Press, Washington, D.C., 2011. 200 pages, http://www.nap.edu

[47] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, O. Schwartz, "Communication-optimal parallel algorithm for strassen's matrix multiplication", SPAA '12 Proceedings of the 24th ACM symposium on Parallelism in algorithms and architectures, Pages 193-204, ISBN: 978-1-4503-1213-4 DOI: 10.1145/2312005.2312044, 2012.

[48] S. Huss-Lederman, E. M. Jacobson, A. Tsao, T. Turnbull, J. R. Johnson, "Implementation of Strassen's algorithm for matrix multiplication" Supercomputing '96 Proceedings of the 1996 ACM/IEEE conference on Supercomputing, Article No. 32, ISBN: 0-89791-854-1 DOI:10.1145/369028.369096, ©1996

[49] V. Saravanan, M. Radhakrishnan, A.S.Basavesh, and D.P. Kothari, " A Comparative Study on Performance Benefits of Multi-core CPUs using OpenMP" IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 1, No 2, January 2012, ISSN (Online): 1694-0814 www.IJCSI.org

[50] M. A. Ismail, S.H. Mirza & T. Altaf "Concurrent Matrix Multiplication on Multi-Core Processors" International Journal of Computer Science and Security (IJCSS), Volume (5): Issue (2): 2011

[51] C. B. Kayhan, M. ˙Imre, "A parallel implementation of Strassen's matrix multiplication algorithm for wormhole-routed all-port 2D torus networks" The Journal of Supercomputing archive, Volume 62 Issue 1, Pages 486-509, DOI: 10.1007/s11227-011-0730-1, October 2012.

[52] Paolo D'Alberto, Alexandru Nicolau "Adaptive Strassen's Matrix Multiplication" ICS '07 Proceedings of the 21st annual international conference on Supercomputing Pages 284 – 292, ISBN: 978-1-59593-768-1 DOI: 10.1145/1274971.1275010 ©2007.

[53] P. D'Alberto, A. Nicolau, Adaptive Winograd's matrix multiplications, ACM Transactions on Mathematical Software (TOMS), Volume 36 Issue 1, Article No. 3, DOI: 10.1145/1486525.1486528, March 2009.

[54] E. I. Milovanovic′, B. M. Randjelovic′, I. Zˇ . Milovanovic′, M. K. Stojcˇev, "Matrix Multiplication on Linear Bidirectional Systolic Arrays" SER. A: APPL. MATH. INFORM. AND MECH. vol. 2, 1 (2010), 11-20.

[55] http://www.mcs.anl.gov/~itf/dbpp/text/node45.html

[56] Z. A.A. Alqadi, M. Aqel, and I. M. M. El Emary, "Performance Analysis and Evaluation of Parallel Matrix Multiplication Algorithms" World Applied Sciences Journal 5 (2): 211-214, 2008.

[57] 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.

[58] MPI: A Message-Passing Interface Standard, Version 2.2, Message Passing Interface Forum, September 4, 2009 www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[59] Hussam Hussein Abuazab, Adel Omar Dahmane, and Habib Hamam, "Sub Tasks Matrix Multiplication Algorithm (STMMA)", World Applied Sciences Journal 18 (10): 1455-1462, 2012, DOI: 10.5829/idosi.wasj.2012.18.10.1818

[60] J. R. Herrero, J. J. Navarro"A Study on Load Imbalance in Parallel Hypermatrix Multiplication using OpenMP", Parallel Processing and Applied Mathematics Lecture Notes in Computer Science Volume 3911, 2006, pp 124-131

[61] Ziad Alqadi and Amjad Abu-Jazzar, 2005. Analysis of program methods used for optimizing matrix multiplication, J. Eng., Vol. 15, NO. 1: 73-78.

[62] Agarwal, R.C., F. Gustavson and M. Zubair, 1994. A high-performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication, IBM J. Res. Develop., Volume 38, Number 6.

[63] Agarwal, R.C., S.M. Balle, F.G. Gustavson, M. Joshi and P. Palkar, 1995. A 3-Dimensional Approach to Parallel Matrix Multiplication, IBM J.Res. Develop., Volume 39, Number 5, pp: 1-8, Sept.

[64] Alpatov, P., G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, Robert van de Geijn and J. Wu, Plapack: Parallel Linear Algebra Package-Design Overview, Proceedings of SC 97, to appear.

[65] http://www.clumeq.ca/index.php/en/support/144-quelles-sont-les-specifications-de-guillimin Computing Conference.

[66] B.Chetverushkin, N.Churbanova, A.Lastovetsky, and M.Trapeznikova, "Parallel Simulation of Oil Extraction on Heterogeneous Networks of Computers", Proceedings of the 1998 Conference on Simulation Methods and Applications (CSMA'98), the Society for Computer Simulation, November 1-3, 1998, Orlando, Florida, USA, pp.53-59.

[67] J.Barbosa, J.Tavares and A.J.Padilha, “A group block distribution strategy for a heterogeneous machine”, Proceedings of the IASTED International Conference NPDPA 2002, 2002, pp.378-383

[68] Sean P. Peisert and Scott B. Baden, “A Programming Model for Automated Decomposition on Heterogeneous Clusters of Multiprocessors”, UCSD CSE Technical Report, 2001.

[69] Olivier Beaumont, Vincent Boudet, Antoine Petitet, Fabrice Rastello, and Yves Robert. A Proposal for a Heterogeneous Cluster ScaLAPACK (Dense Linear Solvers). IEEE Trans. Computers, 2001, Vol.50, No.10, pp.1052-1070

[70] A.Kalinov “Heterogeneous Two-Dimension Block-Cyclic Data Distribution for Solving Linear Algebra Problems on Heterogeneous Networks”, Programming and Computer Software, 2, 1999, pp. 3-11

[71] A.Kalinov, and A.Lastovetsky, “Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers”, Journal of Parallel and Distributed Computing, Vol.61, 2001, pp.520-535

[72] J.Barbosa, J.Tavares and A.J.Padilha, “Linear Algebra Algorithms in a Heterogeneous Cluster of Personal Computers”, Proceedings of 9th Heterogeneous Computing Workshop (HCW 2000), IEEE Computer Society, Cancun, Mexico, May 1, 2000, pp.147-159

[73] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert, “Load balancing strategies for dense linear algebra kernels on heterogeneous two-dimensional grids”, In 14th International Parallel and Distributed Processing Symposium (IPDPS'2000), pages 783-792, 2000

[74] Crandall, P. E., and Quinn, M. J., "Three-Dimensional Grid Partitioning for Network Parallel Processing," Proceedings of the ACM 1994 Computer Science Conference (March 1994), pp. 210-217.

[75] M. Ali Ismail, T. Altaf, S.H. Mirza “Parallel Matrix Multiplication on Multi-Core Processors using SPC3 PM”, International Conference on Latest Computational Technologies (ICLCT'2012) March 17-18, 2012 Bangkok

[76] M. Krishnan, J. Nieplocha, “Memory efficient parallel matrix multiplication operation for irregular problems”, CF '06 Proceedings of the 3rd conference on Computing frontiers, Pages 229-240, ISBN:1-59593-302-6 DOI: 10.1145/1128022.1128054

[77] LAWSON, C., HANSON, R. KINCAID, D., AND KROGH, F. Basic linear algebra subprograms for Fortran usage. ACM Trans. Math. Softw. 5 (1979), 308-323.

[78] LAWSON, C., HANSON, R., KINCAID, D., AND KROGH, F. Algorithm 539: Basic linear algebra subprograms for Fortran usage. ACM Trans. Math. Softw. 5 (1979), 324-325.

[79] DONGARRA, J. J., DUCROZ, J., HAMMARLING, S., AND HANSON, R. An extended set of Fortran basic linear algebra subprograms. ACM Trans. Math. Softw. 24, 1 (Mar. 1988), l-17.

[80] DONGARRA, J. J., DUCROZ, J., HAMMARLING, S., AND HANSON, R. An extended set of Fortran basic linear algebra subprograms: Model implementation and test programs. ACM Trans. Math. Softw. 14, 1 (Mar. 1988), 18-32.

[81] Tinetti F., Barbieri A., "Parallel Computing on Clusters: Performance Evaluation Tool of communications", Proceedings VIII Argentine Congress being targeted are holders of Computing (CACIC), Fac. of Exact and Natural Sciences, Los Angeles times Buenos Aires, Buenos Aires, Argentina, 15 to 18 Oct 2002, P 123.

[82] Alfredo Buttari, Julien Langoub, Jakub Kurzaka, Jack Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures", Parallel Computing 35 (2009) 38–53.

[83] Matrix multiplication on the multi-core Anemone processor, Numerical libraries for Epiphany architecture, Paralant Ltd. (www.paralant.com), April 2011.

[84] 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.

[85] J. Choi, J. J. Dongarra, R. Pozo, D. C. Sorensen, and D. W. Walker. CRPC Research into Linear Algebra Software for High Performance Computers. International Journal of Supercomputing Applications, 8(2):99-118, Summer 1994.

[86] J. Choi, J. J. Fongarra, David W. Walker, "PUMMA Parallel Universal Matrix multiplication Algorithms on Distributed Memory Concurrent Computers"

[87] R.C. Agarwal, F. Gustavson and M. Zubair, 1994. "A high-performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication", IBM J. Res. Develop., Volume 38, Number 6.

[88] R. van de Geijn and J. Watts. SUMMA Scalable Universal Matrix Multiplication Algorithm. LAPACK Working Note 99, Technical Report CS-95-286, University of Tennessee, 1995.

[89] R.Van de Geijn, J. Watts (April 1997) SUMMA: Scalable universal matrix multiplication algorithm. Concurrency: Pract Exp 9(4):255-274

[90] M. D. SCHATZ, J. POULSON, and R. A. VAN DE GEIJN, "Scalable Universal Matrix Multiplication Algorithms: 2D and 3D Variations on a Theme"

[91] J. Choi, "A New Parallel Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers" Center for Research on Parallel Computation, Rice University, 6100 South Main Street, CRPC - MS 41, Housten, TX 77005. Sep 1997

[92] M. Krishnan and J. Nieplocha, "SRUMMA: A Matrix Multiplication Algorithm Suitable for Clusters and Scalable Shared Memory Systems", ipdps, vol. 1, pp.70b, ISBN: 0-7695-2132-0, 18th International Parallel and Distributed Processing Symposium (IPDPS'04), April 2004.

[93] M. Krishnan and J. Nieplocha, "Optimizing Parallel Multiplication Operation for Rectangular and Transposed Matrices", icpads, pp.257, ISBN: 0-7695-2152-5, 10th International Conference on Parallel and Distributed Systems (ICPADS'04), July 2004.

[94] H. Shan and J. P. Singh, "A Comparison of Three Programming Models for Adaptive Applications on the Origin2000," in proceedings of Supercomputing, 2000.

[95] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, E. Aprà, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit", International Journal of High Performance Computing Applications archive, Volume 20 Issue 2, Pages 203 – 231, May 2006.

[96] M. Hermanns, S. Krishnamoorthy, F. Wolf, "A Scalable Replay-based Infrastructure for the Performance Analysis of One-sided Communication", In Proc. of the 1st Intl. Workshop on High-performance Infrastructure for Scalable Tools (WHIST), Tucson, AZ, USA, June 2011.

[97] D. Coppersmith and S. Winograd, "Matrix Multiplication via Arithmetic Progressions" J. Symbolic Computation (1990) 9, 251-280

[98] H. Cohn, C. Umans. Group-theoretic approach to Fast Matrix Multiplication. Proceedings of the 44th Annual Symposium on Foundations of Computer Science, 2003.

[99] H. Cohn, R. Kleinberg, B. Szegedy, C. Umans. Group-theoretic Algorithms for Matrix Multiplication. Proceedings of the 46th Annual Symposium on Foundations of Computer Science, 2005.

[100] James and Liebeck. Representations and Characters of Groups. Cambridge University Press, 1993.

[101] I. Martin Isaacs. Character Theory of Finite Groups. AMS Bookstore, 2006.

[102] P. Bürgisser, M. Clausen, and M.A. Shokrollahi, Algebraic complexity theory, Springer Verlag, 1997

[103] L. Cohn Henry, B. Sxegedy, C. M. Umans, Patent No.: US 7,792,894 B1. Data of Patent: Sep. 7, 2010

[104] H. Cohn and C. Umans. A Group-theoretic Approach to Fast Matrix Multiplication. Proceedings of the 44th Annual Symposium on Foundations of Computer Science, 11–14 October 2003, Cambridge, MA, IEEE Computer Society, pp. 438–449, arXiv:math.GR/0307321.

[105] H. Cohn, R. Kleinberg , B. Szegedy , C. Umans. "Group-theoretic algorithms for matrix multiplication" 46th Annual IEEE Symposium on 23–25 Oct 2005.

[106] Duc Kien Nguyen, Ivan Lavall′ EE, Marc Bui and H.A. Quoc Trung, 2005. A General Scalable Implementation of Fast Matrix Multiplication Algorithms on Distributed Memory Computers,0-7695-2294-7/05 © 2005 IEEE

[107] Ardavan Pedram, Masoud Daneshtalab and Sied Mehdi Fakhraie, 2006. An Efficient Parallel Architecture for Matrix Computations, 1-4244-0772- 9/06 ©2006 IEEE.

[108] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin and Katherine Yelick, 2008. Avoiding Communication in Sparse Matrix Computations, 978-1-4244-1694-3/08 ©2008 IEEE.

[109] Zhaoquan Cai and Wenhong Wei, 2008. General Parallel Matrix Multiplication on the OTIS Network, 978-0-7695-3122-9/08 © 2008 IEEE.

[110] Ioannis Sotiropoulos and Ioannis Papaefstathiou, 2009. A fast parallel matrix multiplication reconfigurable unit utilized in face recognitions system, 978-1-4244-3892-1/09/ ©2009 IEEE.

[111] A. J. Stothers, "On the Complexity of Matrix Multiplication", Doctor of Philosophy University of Edinburgh 2010.

[112] V. Vassilevska Williams, "Breaking the Coppersmith-Winograd barrier", UC Berkeley and Stanford University, Unpublished manuscript (2011)

[113] Nathalie Revol and Philippe Théveny, 2012, "Parallel Implementation of Interval Matrix Multiplication" Reliable Computing 19, 1 (2013) 91-106.

[114] Jian-Hua Zheng, Liang-Jie Zhang, Rong Zhu, Ke Ning1, and Dong Liu, Parallel Matrix Multiplication Algorithm Based on Vector Linear Combination Using MapReduce, 2013 IEEE Ninth World Congress on Services, 978-0-7695-5024-4/13 $26.00 © 2013 IEEE, DOI 10.1109/SERVICES.2013.67

[115] H. Cohn, C. Umans. "Fast matrix multiplication using coherent configurations", Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (New Orleans, Louisiana, USA, January 6-8, 2013), 2013, pages 1074-1087.

[116] Ashley DeFlumere, Alexey Lastovetsky, Searching for the Optimal Data Partitioning Shape for Parallel Matrix Matrix Multiplication on 3 Heterogenous Processors, 2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops, 978-1-4799-4116-2/14 $31.00 © 2014 IEEE, DOI 10.1109/IPDPSW.2014.8

[117] Khalid Hasanov, Jean-Noël Quintin, Alexey Lastovetsky, "Hierarchical approach to optimization of parallel matrix multiplication on large-scale platforms", J Supercomput, DOI 10.1007/s11227-014-1133-x, © Springer Science+Business Media New York 2014

[118] Tania Malik, Vladimir Rychkov, Alexey Lastovetsky, Jean-Noel Quintin, Topology-aware Optimization of Communications for Parallel Matrix Multiplication on Hierarchical Heterogeneous HPC Platforms, 2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops, 978-1-4799-4116-2/14 $31.00 © 2014 IEEE, DOI 10.1109/IPDPSW.2014.10

[119] Alpatov, P., G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, Robert van de Geijn and J. Wu, 1997. Plapack: Parallel Linear Algebra Package, Proceedings of the SIAM Parallel Processing Conference.

[120] Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney and D. Sorensen, 1990 Lapack: A Portable Linear Algebra Library for High Performance Computers, Proceedings of Supercomputing '90, IEEE Press, pp: 1-10.

[121] Barnett, M., S. Gupta, D. Payne, L. Shuler, R. van de Geijn and J. Watts, 1994. Interprocessor Collective Communication Library (InterCom), Scalable High Performance

Appendix A- Basic Linear Algebra Subroutine (BLAS)

In 1973, Hanson, Krogh, and Lawson adopted a set of basic routines for problems in linear algebra, which known later as basic linear algebra subprograms (BLAS) fully described in [77, 78].

It contains subprograms for basic operations on vectors and matrices, to achieve high performance for calculations involving linear algebra. So, instead of processing one multiplication or one addition operation, the compiler will send block of data (vector or matrix) to the processor to be executed. Linear Algebra Package (LAPACK) is a higher-level package built on the same ideas. There are three levels of BLAS subroutines:

1. Level 1 (or L1 BLAS): for operations between vectors, such as $y = ax + y$, where the complexity $O(n\ )$ operations.
2. Level 2 (or L2 BLAS): for operations with matrices and vectors, such as in the equation $y = aAx + by$, where the complexity $O(n^2)$ operations. [79, 80].
3. Level 3 (or L3 BLAS): for operations with matrices, such as $C = aAB + bC$, where the complexity $O(n^3)$ operations.

The performance obtainable by each subroutine of level 3 BLAS is similar to the one that can be obtained with matrix multiplication. [81].

Unfortunately, this approach of software construction is often not well suited to computers with a hierarchy of memory (such as global memory, cache memory, and vector registers) and true parallel-processing computers, like multi cores (dual and quad cores) computers;

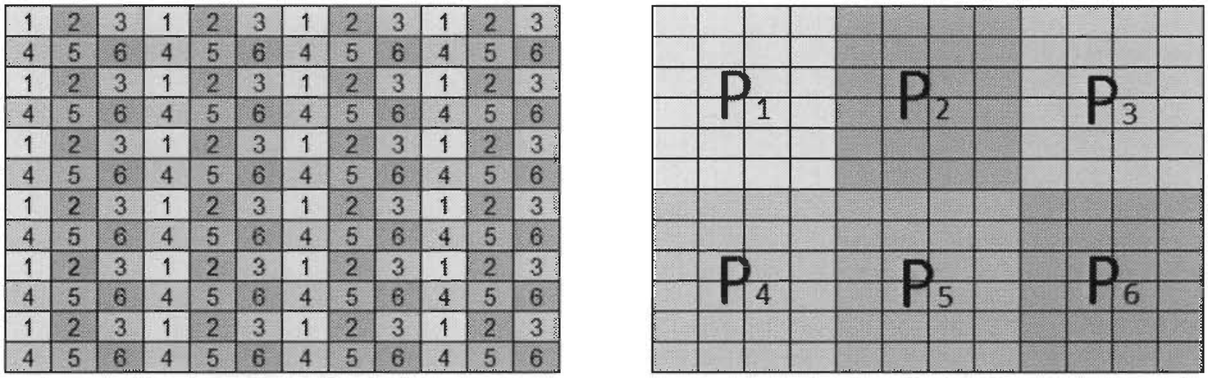
Linear algebra algorithms have to be reformulated or new algorithms have to be developed in order to take advantage of the architectural features on these new processors, [82].

New functions to be included afterwards in level 3 BLAS, to consider the multi-core processors, like the function xGEMM (matrix multiplication) multi-core implementation being presented on BittWare's Anemone floating-point FPGA co-processor that has 16-core Epiphany cores on an eMesh, [83].

Appendix B- Parallel Matrix Multiplication Algorithms

B.1. PUMMA (Parallel Universal Matrix Multiplication)

PUMMA[84], SUMMA, and DIMMA are numerical algorithms for dense matrices on distributed- memory concurrent computers are based on a block cyclic data distribution [85]. The three algorithms have the same matrix-point of-view and processor point-of-view as shown in the figure below:



1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6
1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6
1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6
1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6
1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6
1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6

The result matrix

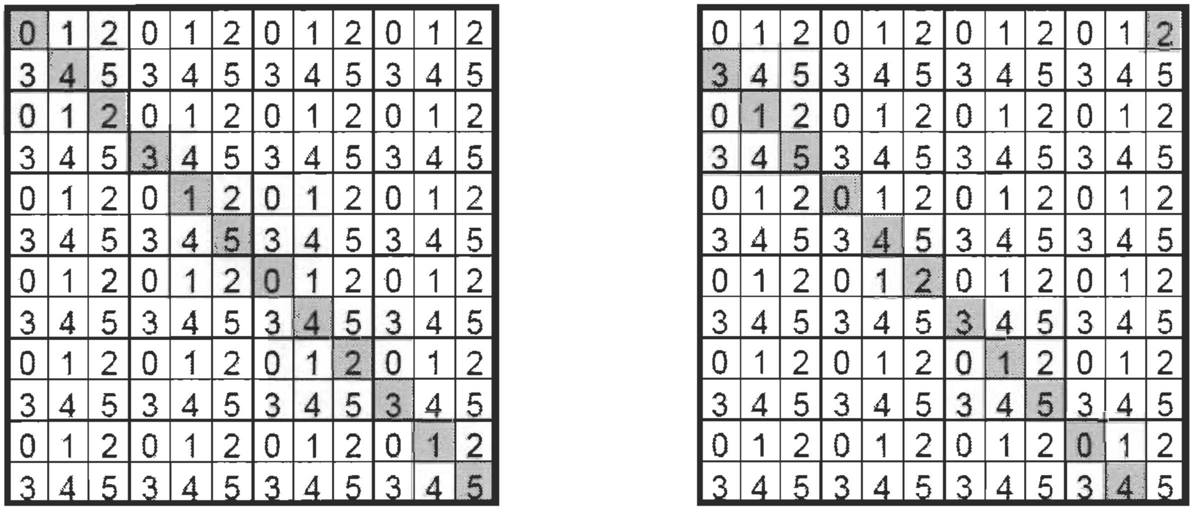
The distribution of the load among the processors

Figure B-0-1 matrix-point of-view and processor point-of-view for PUMMA, SUMMA, and DIMMA

The figure above shows that processor P_1 will produce the yellow cells of the result matrix, while processor P_2 will produce blue cells of the result matrix, and processor P_3 will produce orange cells of the result matrix, and so on. So, each processor has several blocks of the matrices A and B, need to be passed to the processors, PUMMA [86] suggested three variant distributions of the algorithm:

1. Single Diagonal Broadcast: Only one wrapped diagonal is column cast in each stage. In implementing the algorithm, the size of the sub matrices multiplied in each

processor should be maximized to optimize the performance of the sequential xGEMM routine, as shown in the figure below, where the broadcasted column is shown in gray, in the two adjacent broadcasting steps shown.



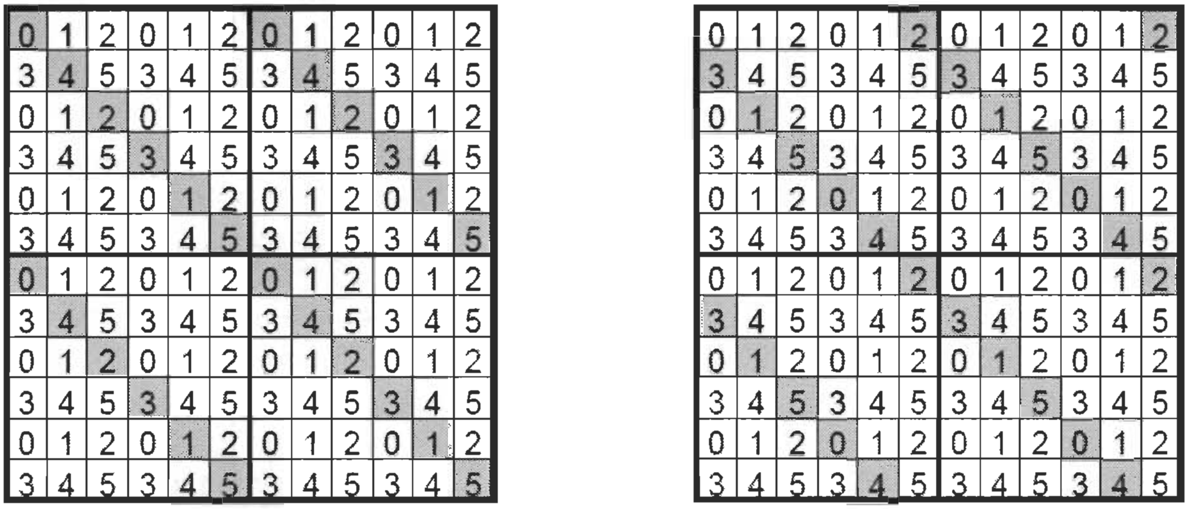
0	1	2	0	1	2	0	1	2	0	1	2
3	**4**	5	3	4	5	3	4	5	3	4	5
0	1	**2**	0	1	2	0	1	2	0	1	2
3	4	5	**3**	4	5	3	4	5	3	4	5
0	1	2	0	**1**	2	0	1	2	0	1	2
3	4	5	3	4	**5**	3	4	5	3	4	5
0	1	2	0	1	2	**0**	1	2	0	1	2
3	4	5	3	4	5	3	**4**	5	3	4	5
0	1	2	0	1	2	0	1	**2**	0	1	2
3	4	5	3	4	5	3	4	5	**3**	4	5
0	1	2	0	1	2	0	1	2	0	**1**	2
3	4	5	3	4	5	3	4	5	3	4	**5**

0	1	2	0	1	2	0	1	2	0	1	**2**
3	4	5	3	4	5	3	4	5	3	4	5
0	**1**	2	0	1	2	0	1	2	0	1	2
3	4	**5**	3	4	5	3	4	5	3	4	5
0	1	2	**0**	1	2	0	1	2	0	1	2
3	4	5	3	**4**	5	3	4	5	3	4	5
0	1	2	0	1	**2**	0	1	2	0	1	2
3	4	5	3	4	5	**3**	4	5	3	4	5
0	1	2	0	1	2	0	**1**	2	0	1	2
3	4	5	3	4	5	3	4	**5**	3	4	5
0	1	2	0	1	2	0	1	2	**0**	1	2
3	4	5	3	4	5	3	4	5	3	**4**	5

Figure B-0-2 PUMMA Single Data Broadcasting

2. Multiple Diagonal Broadcast 1 (MDB1).

PUMMA algorithm utilize the concept of Lowest Common Multiplicand (LCM) to find an optimal size of the block where each processor can process, optimal concerning the size of the multiplied matrices and number of processors, so the algorithm belong to certain processor concerning certain block, need not to be changed for another block, i.e. each processor will have program to process certain cells of the input matrices, while the input matrices will be broadcast in what is called Multiple Diagonal Broadcast 1 as shown in the figure below, this will result in better performance as it reduces the communication latency.



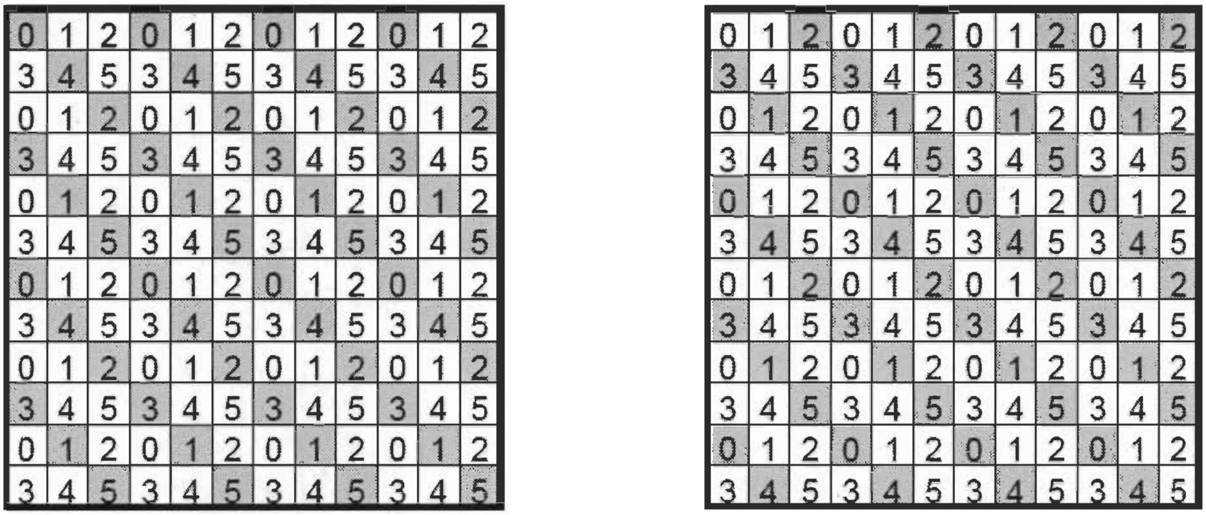
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5

0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5

Figure B-0-3 PUMMA Multiple Data Broadcasting 1

3. Multiple Diagonal Broadcast 2 (MDB2).

In this algorithm, the granularity if the algorithm is increased [86] and this can be achieved by broadcasting more diagonals to feed more processors, avoiding keep some of them idle waiting data of next operation. So and according LCM computation, the figure below shows MDB2 broadcasting diagonals.



0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5

0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5

Figure B-0-4 PUMMA Multiple Data Broadcasting 2

However, PUMMA makes it difficult to overlap computation with communication since it always deals with the largest possible blocks for both computation and communication, and it requires large memory space to store them temporarily, which makes it impractical in real applications. On the other hand, PUMMA, SUMMA, and DIMMA like other algorithms, are not utilizing the parallel structure of the new processors like dual core processors.

B.2. SUMMA (Scalable Universal Matrix Multiplication)

Agrawal, Gustavson and Zubair [87] proposed another matrix multiplication algorithm by efficiently overlapping computation with communication on the Intel iPSC/860 and Delta system. Van de Geijn andWatts [88, 89] independently developed the same algorithm on the Intel paragon and called it SUMMA.

In SUMMA, matrices A and B are divided into several columns and rows of blocks, respectively, whose block sizes are kb. Processors multiply the first column of blocks of A with the first row of blocks of B. Then processors multiply the next column of blocks of A and the next row of blocks of B successively.

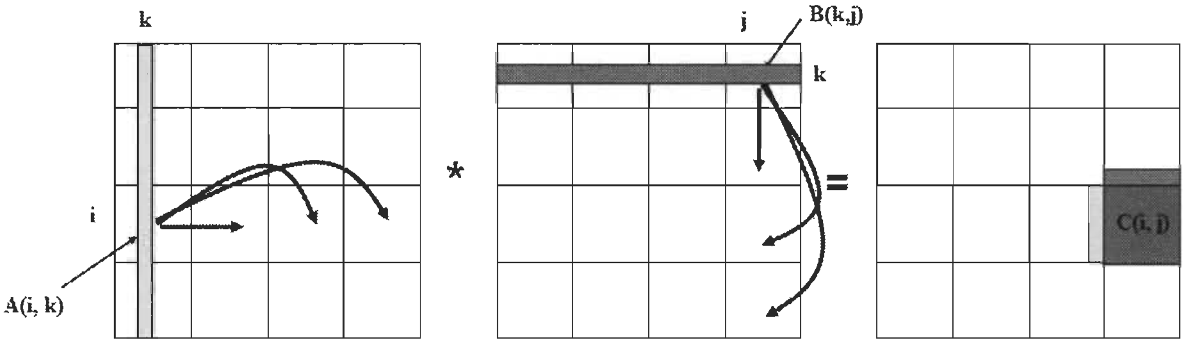


Figure B-0-5 SUMMA algorithm

While this equation will take place

$$C(i,j) = c(i,j) + \sum_{k} A(i,k) * B(k,j)$$

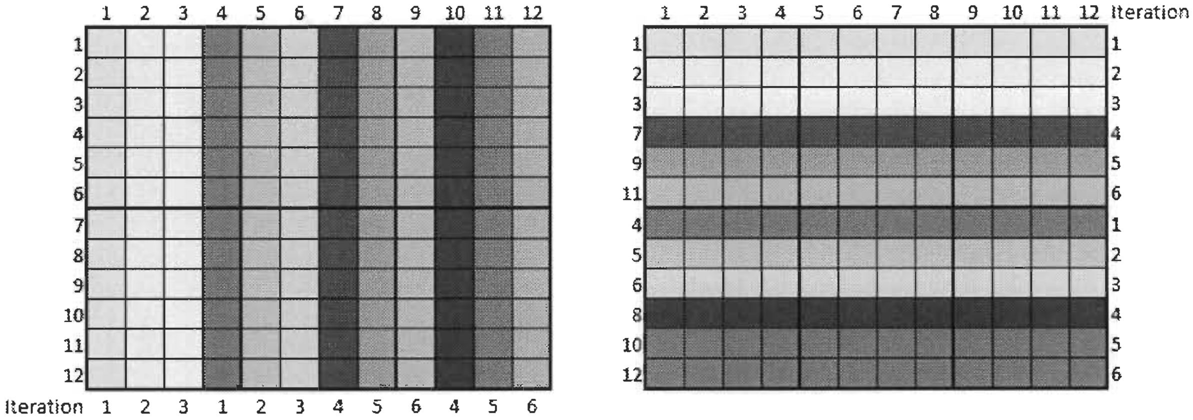


Figure B-0-6 SUMMA Algorithm iterations

As the snapshot of the figure above shows, the first column of processors P1 and P4 begins broadcasting the first column of blocks of A (A(:; 1)) along each row of processors .

The same time, the first row of processors, P1, P2, and P3 broadcasts the first row of blocks of B (B(0; :)) along each column of processors. After the local multiplication, the second column of processors, P2 and P5, broadcasts A(:; 2) rowwise, and the second row of processors, P4, P5, and P6, broadcasts B(2; :) columnwise. This procedure continues until the last column of blocks of A and the last row of blocks of B.

In 2009 Martin D. Schatz, Jack Poulson, and Robert A. Van De Geijn [90] have extended SUMMA to SUMMA 3D, where each processor will multiply vector by vector and rather than multiplying cell by cell, this results in minimizing the communication time significantly. This development synchronize with the new architecture of multi-code processors, where the vectors multiplication will be parallelized between the cores of the processors, Yet, I will have extra development in my ITPMMA algorithm rather than to consider the new architecture of multi-code processors to reduce the communication time, I have independent sub tasks, where it is not the case in SUMMA 3D, also, I will not be able to apply vector multiplication in different problem like solving Laplace equation using Jacobi iteration.

B.3. DIMMA (Distribution Independent Matrix Multiplication)

DIMMA algorithm is based on two new ideas [91]:

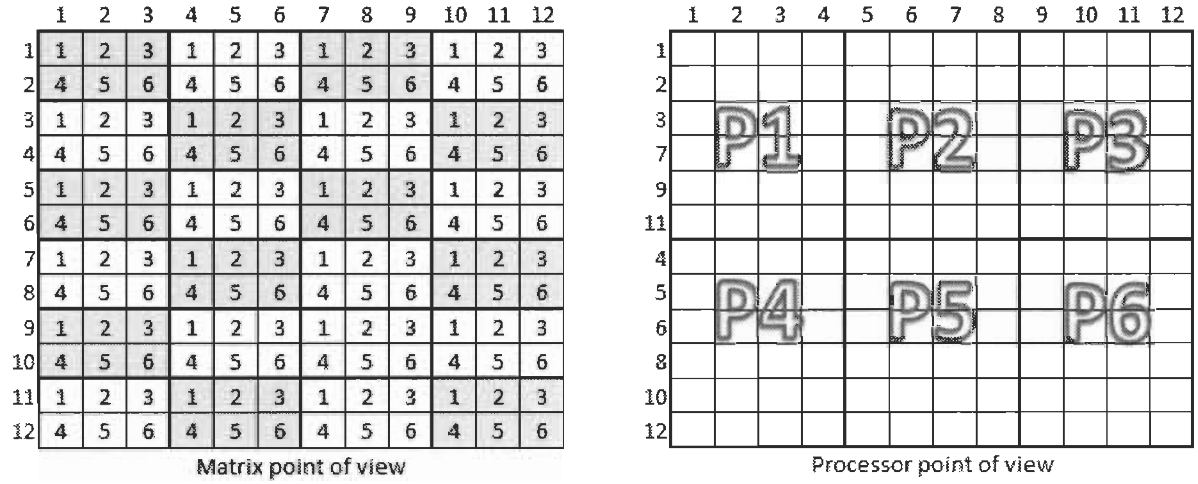
1. It uses a modified pipelined communication scheme to overlap computation and communication effectively.
2. Exploits the LCM block concept –where the size of the block will be the lowest common multiplier of the matrices size – to obtain the maximum performance of the sequential BLAS - Basic Linear Algebra Subprograms - routine in each

processor even when the block size is very small as well as very large. Basic Linear Algebra Subprograms (BLAS) is API which performs matrix multiplication. This includes:

1. SGEMM for single precision,
2. DGEMM for double-precision,
3. CGEMM for complex single precision, and
4. ZGEMM for complex double precision.

GEMM is often tuned by high-performance computing vendors to run as fast as possible because it is the building block for so many other routines.

The figure below shows how this –DIMMA – algorithm works. The numbered squares represent blocks of elements, and the number indicates the location in the processor grid - all blocks labeled with the same number are stored in the same processor. The slanted numbers, on the left and on the top of the matrix, represent indices of a row of blocks and of a column of blocks, respectively. Figure 1(b) reflects the distribution from a processor point-of-view, where each processor has 6×4 blocks.



	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	1	2	3	1	2	3	1	2	3
2	4	5	6	4	5	6	4	5	6	4	5	6
3	1	2	3	1	2	3	1	2	3	1	2	3
4	4	5	6	4	5	6	4	5	6	4	5	6
5	1	2	3	1	2	3	1	2	3	1	2	3
6	4	5	6	4	5	6	4	5	6	4	5	6
7	1	2	3	1	2	3	1	2	3	1	2	3
8	4	5	6	4	5	6	4	5	6	4	5	6
9	1	2	3	1	2	3	1	2	3	1	2	3
10	4	5	6	4	5	6	4	5	6	4	5	6
11	1	2	3	1	2	3	1	2	3	1	2	3
12	4	5	6	4	5	6	4	5	6	4	5	6

Matrix point of view

Processor point of view

Figure B-0-7 DIMMA Algorithm

With this modified communication scheme, DIMMA is implemented as follows. After the first procedure, that is, broadcasting and multiplying A(:; 0) and B(0; :), the first column of processors, P0 and P3, broadcasts A(:; 6) along each row of processors, and the first row of processors, P0, P1, and P2 sends B(6; :) along each column of processors, as shown in Figure 6. The value 6 appears since the LCM of P = 2 and Q = 3 is 6.

For the third and fourth procedures, the first column of processors, P0 and P3, broadcasts row wise A(:; 3) and A(:; 9), and the second row of processors, P3, P4, and P5, broadcasts column wise B(3; :) and B(9; :), respectively. After the first column of processors, P0 and P3, broadcasts all of their columns of blocks of A along each row of processors, the second column of processors, P1 and P4, broadcasts their columns of A.

The parallel matrix multiplication requires O (N3) ops and O (N2) communications, i.e., it is computation intensive. For a large matrix, [91] the performance difference between

SUMMA and DIMMA may be marginal and negligible. But for small matrix of N = 1000 on a 16×16 processor grid, the performance difference is approximately 10%.

DIMMA algorithm depend in dividing the matrices into small blocks, small enough to be utilized by the`upper levels of the memory hierarchy like registers, cache, which is faster than to data in lower levels memory like RAM or any shared memory else.

The absent point here is, the advance in the development of operating systems take this point in its duties, while keep transferring the data between the processors is more time consuming, which will result in massive success my algorithm ITPMMA.

B.4. SRUMMA a matrix multiplication algorithm suitable for clusters and scalable shared memory systems (2004).

SRUMMA algorithm [92] create list of tasks where a task computes each of the *Aik×Bkj* products, corresponding to the block matrix multiplication equation

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

Reorder the *task list* according to the communication domains for processors at which the *Aik×Bkj* are stored. The "diagonal shift" algorithm is used to sort the task list so that the communication pattern reduces the communication contention on clusters. The processor start producing the products, then each processor will gather the results from the other processors to perform the summation procedure and to produce the final matrix result.

The main advantage of this algorithm will appear with shared environment where the shared memory will be available like Cray X1 and the SGI Altix, where the communication time between processors will be reduced to almost zero.

In clustered parallel environment, SURMMA algorithm has no significant advance. Also, it is not considering the new multi core CPUs, where it is parallelized.

After three months, Manojkumar Krishnan and Jarek Nieplocha advanced their SURMMA algorithm [93] to improve its performance over transpose and rectangular matrices; it differs from the other parallel matrix multiplication algorithms by the explicit use of shared memory and remote memory access (RMA) communication rather than message passing. The new advances in SURMMA algorithm was succeful for some cases shown in the paper [93], but still suffers from Shared memory model characteristic, which is much easier to use but it ignores data locality/placement. Given the hierarchical nature of the memory subsystems in modern computers this characteristic can have a negative impact on performance and scalability. Careful code restructuring to increase data reuse and replacing fine grain load/stores with block access to shared data can address the problem and yield performance for shared memory that is competitive with message-passing [94, 95, 96].

B.5. Coppersmith and Winograd (CW) Algorithm

This algorithm have been developed in 1987 [97] by Don Coppersmith and Shmuel Winograd, then being advanced later on, mainly by them [98, 99, 100]. The algorithm running time has improved to O ($n^{2.38}$). The basic idea behind this algorithm is to apply Schönhage Theorem [101, 102], which implies embedding the matrices as elements of group algebra

$$(\sum_g g.a_g).(\sum_g g.b_g) = \sum_g g.c_g \text{ where } c_g = \sum_{i,j:i.j=g} a_i.b_j$$

The algorithm outlined by Jenya Krishtein as following:

- Embed matrices A,B into the elements $\overline{A}, \overline{B}$ of the group algebra C[G]
- Multiplication of $\overline{A}, \overline{B}$ in the group algebra is carried out in the Fourier domain after performing the Discrete Fourier Transform (DFT) of $\overline{A}, \overline{B}$
- The product $\overline{\mathrm{A}}\,\overline{\mathrm{B}}$ is found by performing the inverse DFT
- Entries of the matrix AB can be read off from the group algebra product $\overline{\mathrm{A}}\,\overline{\mathrm{B}}$.

This algorithm being refined in 2010 by Henry L. Cohn, Balázs Sxegedy, Christopher M. Umans [103] as shown in the following fig. [3-12][3-13]

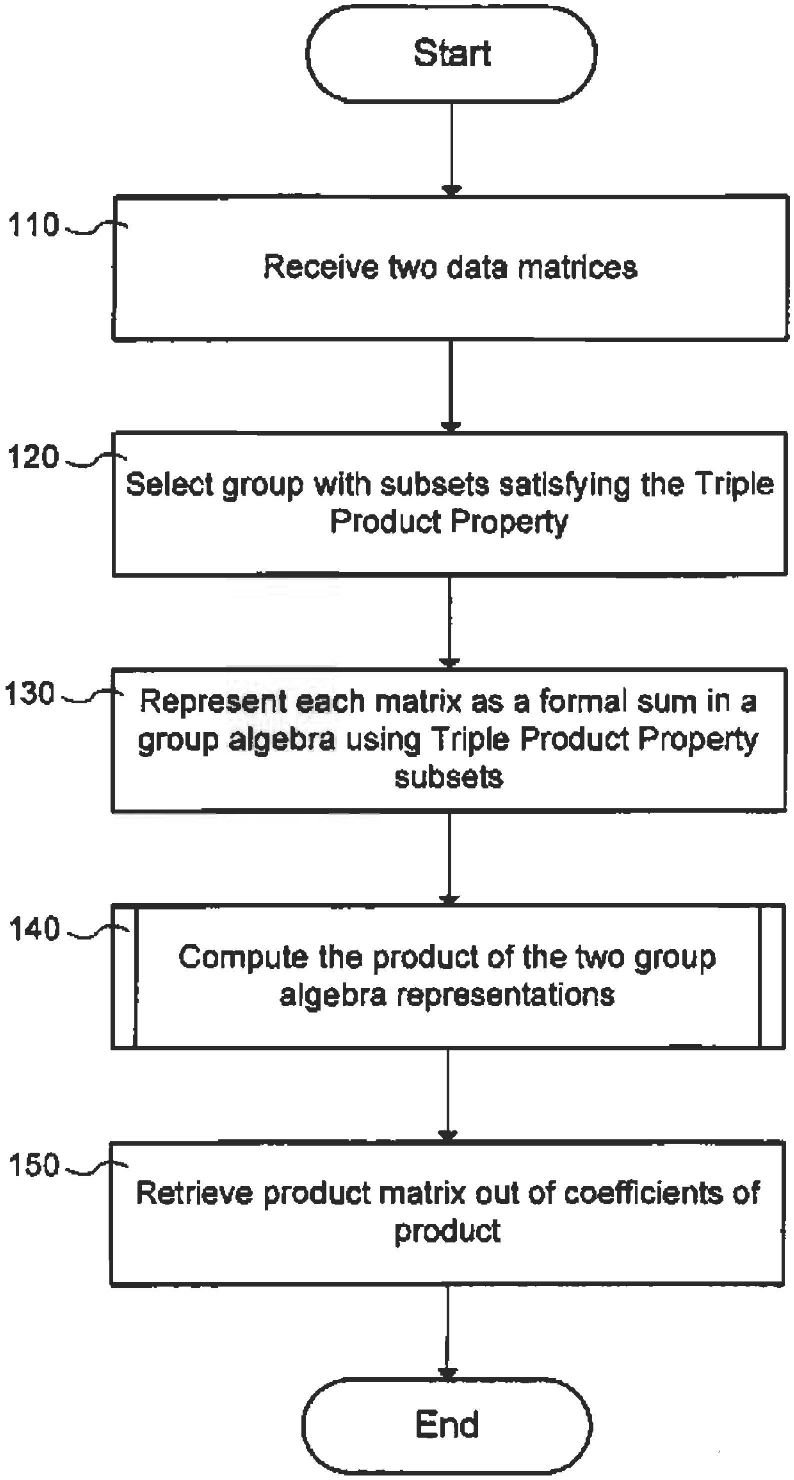

Start
110
Receive two data matrices
120
Select group with subsets satisfying the Triple Product Property
130
Represent each matrix as a formal sum in a group algebra using Triple Product Property subsets
140
Compute the product of the two group algebra representations
150
Retrieve product matrix out of coefficients of product
End

Figure B-0-8 Coppersmith and Winograd 2010 (1/2) [13]

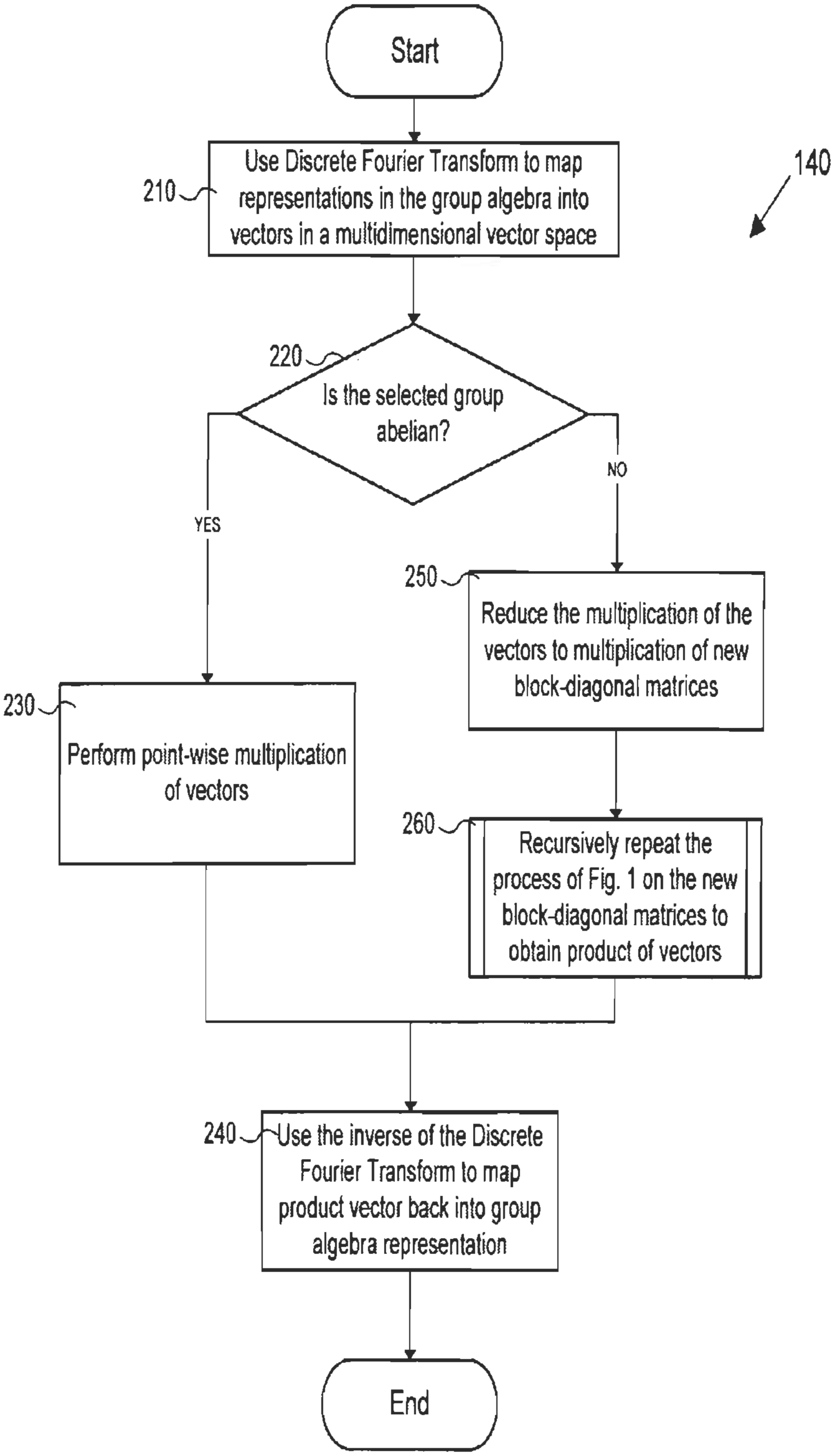

Start
210
Use Discrete Fourier Transform to map representations in the group algebra into vectors in a multidimensional vector space
140
220
Is the selected group abelian?
YES
NO
250
Reduce the multiplication of the vectors to multiplication of new block-diagonal matrices
230
Perform point-wise multiplication of vectors
260
Recursively repeat the process of Fig. 1 on the new block-diagonal matrices to obtain product of vectors
240
Use the inverse of the Discrete Fourier Transform to map product vector back into group algebra representation
End

Figure B-0-9 Coppersmith and Winograd 2010 (2/2) [102]

Well, this algorithm was an open in developing new approach having different terms rather than the terms of Systolic Algorithm, which result in less running time of O ($n^{2.38}$), rather than running time of Systolic Algorithm which is O ($n^{2.807}$).

Here, in this thesis, I am introducing new approach by ITPMMA algorithm, as I am not using the terms of any previous approach; instead I am defining new approach that has several independent tasks to eliminate the consuming time for waiting intermediate results and consuming time for transferring the intermediate results between the processors. Also, the both previous approaches suffering from non-optimized load balance.

The remaining sections of this chapter will give the reader ideas about the algorithms being advanced (CW) Algorithm.

B.6. Group-theoretic Algorithms for Matrix Multiplication

In 2003, Cohn and Umans [104] introduced a new, group-theoretic framework for designing and analyzing matrix multiplication algorithms. In 2005, together with Kleinberg and Szegedy [105], they obtained several novel matrix multiplication algorithms using the new framework; however they were not able to exceed running time of O ($n^{2.376}$).

B.7. NGUYEN et al Algorithms for Matrix Multiplication

In 2005, NGUYEN et al. [106] combined the use of Fast Multipole Method (FMM) algorithms and the parallel matrix multiplication algorithms, which gave remarkable results. Nevertheless, the algorithm still suffers data dependency and high communication cost among the processors. Moreover, the algorithm does not address heterogeneous environments.

B.8. Pedram et al Algorithms for Matrix Multiplication

On 2006, Pedram et al. [107], have developed high-performance parallel hardware engine for matrix power, matrix multiplication, and matrix inversion, based on distributed memory. They have used Block-Striped Decomposition algorithm directly to implement the algorithm hardware wise. There was obvious drawback in term of speed up of efficacy of using the processors; instead the algorithm reduces memory bandwidth by taking advantage of reuse data, which increases the data dependencies.

B.9. James Demmel Algorithms for Matrix Multiplication

On 2008 James Demmel developed a new algorithm to minimize the gap between computation and communication speed, which continues to widen [108]. The performance of sparse iterative solvers was the aim of this algorithm, where it produced speedup of over three times of serial algorithm. In fact, the increasing gap between computation and communication speed, is one of the main points to be addressed by reducing the communication between processors as much as possible. The algorithm is still suffering data dependency and communication; especially for large matrices sizes.

B.10. Cai and Wei Algorithms for Matrix Multiplication

On 2008 Cai and Wei [109] developed new matrix mapping scheme to multiply two vectors, a vector and a matrix, and two matrices which can only be applied to optical transpose interconnection system (OTIS-Mesh), not to general OTIS architecture, to reduce communication time. They have achieved some improvements compared to Cannon algorithm, but it was expensive.

B.11. Sotiropoulos and Papaefstathiou Algorithms for Matrix Multiplication

On 2009, Sotiropoulos and Papaefstathiou implement Block-Striped Decomposition algorithm using FPGA device [110]. There is no achievement in terms of reducing data dependencies and communication cost.

B.12. Andrew Stothers 2010

In his PhD thesis [111], Andrew Stothers worked in the complexity of matrix multiplication algorithms generated by CW and the advanced algorithms of CW, and he conclude it is likely that any gains obtained in reducing running time will be very small.

B.13. Breaking the Coppersmith-Winograd barrier 2011

Virginia Vassilevska Williams [112] has developed new tools for analyzing matrix multiplication constructions similar to the Coppersmith- Winograd construction, and obtain a new improved bound on O $(n^{2.3727})$.

B.14. Nathalie Revol and Philippe Théveny 2012

On 2012, Nathalie Revol and Philippe Théveny developed new algorithm, called "Parallel Implementation of Interval Matrix Multiplication" to address the implementation of the product of two dense matrices on multicore architectures [113]. The algorithm produced accurate results but the performance was poor.

B.15. Jian-Hua Zheng 2013

On 2013 Jian-Hua Zheng [114] proposed new technique based in data reuse. It suffers from a lot of data dependency and high communication cost.

B.16. Fast matrix multiplication using coherent configurations 2013

Henry Cohn and Christopher Umans [115] found that running time of matrix multiplication could be reduced by embedding large matrix multiplication instances into small commutative coherent configurations.

B.17. Square-Corner algorithm 2014

Another research has been conducted on 2014 [116], a new decomposition technique called Square-Corner instead of and Block Rectangle partition shapes to reduce the communication time is proposed. The research was limited to only three heterogeneous processors. For some cases, they have reported less communication time and therefore showed a performance improvement.

B.18. Khalid Hasanov algorithm 2014

Also, on 2014 Khalid Hasanov [117] introduced hierarchy communication scheme to reduce the communication cost to SUMMA algorithm. Although achieved some better performance, pre ITPMMA algorithm drawbacks like data dependency and communication cost are still there. Moreover, this algorithm is for homogenous environment.

B.19. Tania Malik et al algorithm 2014

On 2014, Tania Malik et al. [118] proposed new network topology to decrease communication time among the processors. The algorithm suffers from more data dependencies between the processors.

Appendix C- ITPMMA execution time vs different Parallel Algorithms

Table C-0-1 PUMMA (MBD2), [119]

Task	Execution time	ITPMMA (STMMA)
Scatter A	$m^2p\ t_c$	0
Broadcast the diagonal elements of B	npt_c	0
Multiply A and B	$m^2n\ t_f$	$n^2\ t_f$
Switch processors' A submatrix	m^2 root(p) t_c	0
Generate the resulting matrix	$n^2\ t_f + n2\ t_c$	$2mn\ t_c$
Total execution time	$t_f(m\ 2n + n^2) + t_c(2n^2 + m^2$root(p)(p+1))	$2mn\ t_c\ (1 + p) + m^2n\ t_f$

Table C-0-2 SUMMA, [120]

Task	Execution time	ITPMMA (STMMA)
Broadcast A and B	$2mnp\ t_c$	$2mnp\ t_c$
Multiply A and B	$m^2n\ t_f$	$m^2n\ t_f$

Generate the resulting matrix	n^2 t_f + n^2 t_c	2mn t_c
Total execution time	t_f(m 2n + n^2) +t_c(n^2 + 2mnp)	2mn t_c (1 + p) + m^2n t_f

Table C-0-3 DIMMA, [121]

Task	Execution time	ITPMMA (STMMA)
Broadcast A and B	2mnp t_c	2mnp t_c
Multiply A and B	m^2n t_f	m^2n t_f
Generate the resulting matrix	n^2 t_f + n^2 t_c	2mn t_c
Total execution time	t_f(m 2n + n2) +t_c(n^2 + 2mnp)	2mn t_c (1 + p) + m^2n t_f

Appendix D- ITPMMA Algorithm pseudocode

ITPMMA pseudocode for four processor and the multiplied and multiplicand matrices are square of size 12.

```
int numprocs, myrank;
int ML = 12; //matrix length

MPI_Init(&argc, &argv)
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, & myrank);
MPI_Status status;

if ( myrank == 0) {
// Load Matrix A from a file
For (int i=0; i< ML, i++)
For (int k= 0, k< ML k++)
Load A[i][k];

/* Load the required columns of matrix B, that is columns B[][0], B[][4], B[][8]
*/
For (int i=0; i< ML, i++)
For (int k= myrank, k< ML k=k+ numprocs)
{
BB[i] = B[i][k];
/* multiply the loaded matrices*/
for | | J=0 to ML {
 for K=0 to ML {
  $C_{0J} = C_{0J} + A_{0K} \times BB_J$
           }}
}
// Send the result to the
// server node of processor 0 using batch command, so we do
// not need to use the MPI_Send and MPI_Receive as
// these data is matured and not to be processed any more.
}
if ( myrank == 1) {
// Load Matrix A from a file
For (int i=0; i< ML, i++)
For (int k= 0, k< ML k++)
Load A[i][k];

// Load the required columns of matrix B, that is columns //B[][0], B[][4], B[][8]
For (int i=0; i< ML, i++)
For (int k= myrank, k< ML k=k+ numprocs)
{
BB[i] = B[i][k];
/ multiply the loaded matrices*/
for | | J=0 to 3 {
 for K=0 to 3 {
  $C_{1J} = C_{1J} + A_{1K} \times BB_J$
```

```
          }}
}
// Send the result to the
// server node of processor 0 using batch command, so we do
// not need to use the MPI_Send and MPI_Receive as
// these data is matured and not to be processed any more.
}

if ( myrank == 2) {
// Load Matrix A from a file
For (int i=0; i< ML, i++)
For (int k= 0, k< ML k++)
Load A[i][k];
// Load the required columns of matrix B, that is columns //B[][0], B[][4], B[][8]
For (int i=0; i< ML, i++)
For (int k= myrank, k< ML k=k+ numprocs)
{
BB[i] = B[i][k];
/* multiply the loaded matrices*/
for | | J=0 to 3 {
 for K=0 to 3 {
  C2J = C2J + A2K ×BBJ}}
}
// Send the result to the
// server node of processor 0 using batch command, so we do
// not need to use the MPI_Send and MPI_Receive as
// these data is matured and not to be processed any more.
}

if ( myrank == 3) {
// Load Matrix A from a file
For (int i=0; i< ML, i++)
For (int k= 0, k< ML k++)
Load A[i][k];

// Load the required columns of matrix B, that is columns //B[][0], B[][4], B[][8]
For (int i=0; i< ML, i++)
For (int k= myrank, k< ML k=k+ numprocs)
{
BB[i] = B[i][k];
/* multiply the loaded matrices*/
for | | J=0 to 3{
 for K=0 to 3 {
  C3J = C3J + A3K ×BBJ}}
}
// Send the result to the
// server node of processor 0 using batch command, so we do
// not need to use the MPI_Send and MPI_Receive as
// these data is matured and not to be processed any more.
```

Annexe E - Résumé détaillé

Dans cette thèse, un nouveau cadre pour le traitement parallèle est introduit. L'objectif principal est de considérer l'architecture moderne des processeurs et de réduire le temps de communication entre les processeurs de l'environnement parallèle.

Plusieurs algorithmes parallèles ont été développés dans les quatre dernières décennies, en se basant sur une décomposition des données et un traitement parallèle. Ces algorithmes souffrent de deux groupes d'inconvénients:

1. Tous les algorithmes n'utilisent pas les avancées dans l'architecture des processeurs modernes, comme des noyaux multiples et le niveau de mémoire cache différent. Lorsque chaque processeur est capable de traiter plusieurs multiplications simples et sommations. Les processeurs (à cette époque) ne disposaient pas d'un nombre élevé de processeurs et une grande quantité de mémoire cache.

2. Tous les algorithmes ne présentent pas de solution optimale en termes de :

    a. La définition de la taille optimale de bloc de matrices à décomposer.

    b. Le temps de communication des messages échangés entre les processeurs, qui est proportionnelle au nombre de processeurs et le nombre de blocs des matrices décomposées, qui dépasse parfois le temps de calcul.

    c. La dépendance des données entre les processeurs, par conséquent plusieurs processeurs demeurent en attente jusqu'à ce qu'il obtenir les résultats des processeurs précédents.

    d. Mauvais équilibre de charge, en particulier avec des matrices non carrées.

Le nouveau cadre pour le traitement parallèle qui est proposé dans cette thèse permettra de surmonter les inconvénients ci-dessus. Pour cela, le nouveau cadre est basé sur le développement de deux techniques:

1. Une technique pour reconstruire le problème à exécuter en parallèle dans la nouvelle structure qui se compose d'un ensemble de tâches indépendantes. Donc, chaque processeur exécute certaines de ces tâches de manière indépendante. La nouvelle structure réduit les dépendances entre les processeurs, de sorte que le temps de communication est réduit.
2. Une nouvelle technique de décomposition de données. Certains problèmes numériques comme l'équation de Laplace ne pourraient pas être reconstruites en ensemble de tâches indépendantes. Pour cela, afin de réduire le temps de communication entre les processeurs, nous utilisons une nouvelle technique de décomposition de données, appelée la technique « clustered one dimension decomposition »

La première technique sera mise en œuvre sur les multiplications de matrices parallèles. En fait, la multiplication des matrices parallèles été utilisée comme problème référence pour tous les algorithmes parallèles. Il est l'un des problèmes numériques les plus fondamentaux dans les sciences et l'ingénierie; en commençant par les transactions quotidiennes de base de données dans les index, les prévisions météorologiques, l'océanographie, l'astrophysique, la mécanique des fluides, le génie nucléaire, le génie chimique, la robotique et l'intelligence artificielle, la détection de pétrole et de minéraux, la détection géologique, le recherche médicale et l'armée, la communication et télécommunication, analyse de l'ADN matériel, les simulations du tremblement de terre, l'extraction de données et le traitement de l'image.

Dans cette thèse, un nouvel algorithme parallèle de multiplication de matrice a été développé en utilisant un nouveau cadre qui implique de générer des tâches indépendantes entre les processeurs, de réduire le temps de communication entre les processeurs à zéro et d'utiliser l'architecture moderne des processeurs pour des résultats d'efficacité à 97% contre 25% précédemment.

3. D'autre part, la seconde technique, « clustered one dimensions decomposition » a été mise en œuvre et appliquée à la résolution de l'équation de Laplace en utilisant la méthode de Gauss-Seidel. Ainsi, la décomposition des données lors de la résolution de l'équation de Laplace, en utilisant la nouvelle technique de décomposition en réduisant le coût de la communication entre les processeurs; se traduit par un taux d'efficacité de 55% contre 30% pour la technique de décomposition de données à deux dimensions.

Les algorithmes de traitement parallèle précédents

Tous les algorithmes parallèles basés sur la décomposition d'un produit de matrices en blocs de données de taille plu petite, les blocs seront bien assortis et répartis entre les processeurs, de sorte que chaque processeur exécute une partie du calcul. Ceci réduit le temps de calcul entier. Dans le cas de tous les algorithmes parallèles de produit de matrices, nous avons utilisé la multiplication de matrices comme un problème de référence, car il avait été utilisé historiquement à cet objectif. D'autres facteurs influencent également la performance de ce genre d'algorithmes:

1. La taille optimale de blocs de matrices décomposées, de sorte qu'on minimise le temps d'exécution.
2. Les messages échangés entre les processeurs sont très gourmands en terme de temps d'exécution, et il dépend grandement de la structure du réseau. Le temps de

communication est proportionnel au nombre de blocs et le nombre de processeurs, ce qui dépasse en général le temps requis d'exécution.

3. La dépendance de données entre les processeurs où certains processeurs demeurent en attente le temps que les calculs intermédiaires des autres processeurs se fassent.

4. Certains algorithmes souffrent d'un autre inconvénient, qui est l'équilibrage de charge, en particulier avec les multiplications de matrices non carrées.

Figure E-1 ci-dessous simule la multiplication de matrices en série à l'aide d'un seul processeur. La Figure E-2 montre le comportement des algorithmes parallèles précédents. Il est clair que les deux processeurs P2 et P3 sont en attente, jusqu'à ce processeur P1 passe à l'état de repos. P4 ne pourra commencer le traitement avant que P2 et P3 ne complètent leurs tâches. L'efficacité du temps des processeurs est ainsi affectée grandement. L'efficacité de calcul de chaque processeur est 16/4 = 25%.

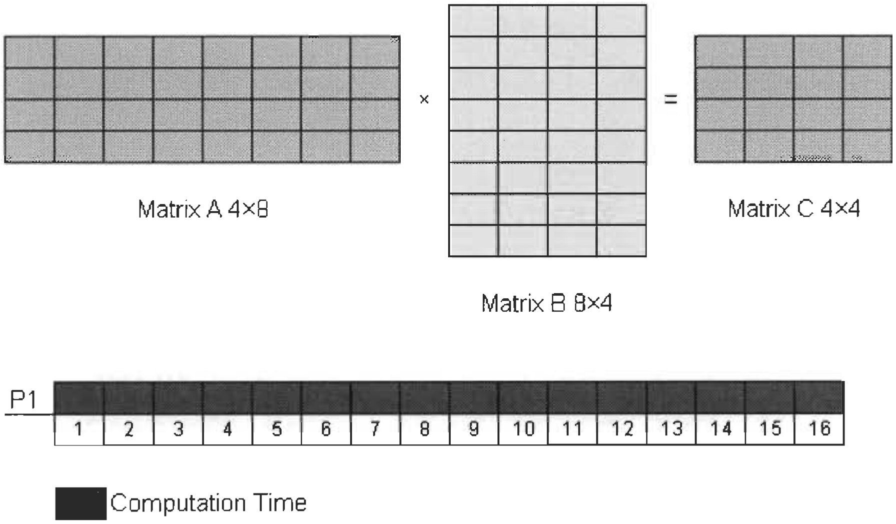


Figure E - 1 Multiplication de matrice en série

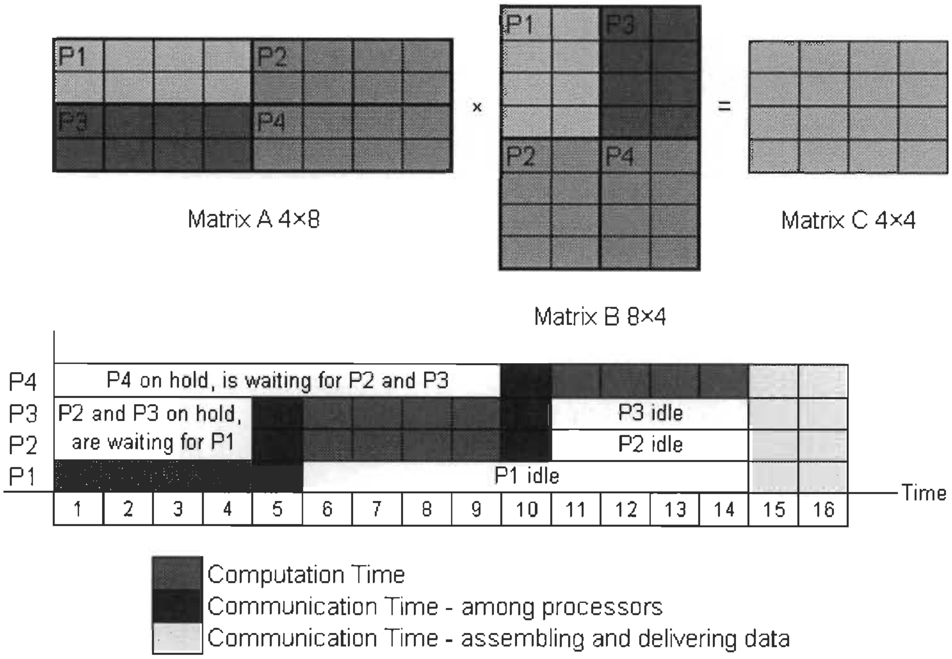

P1
P2
P3
P4
Matrix A 4×8
×
P1
P3
P2
P4
Matrix B 8×4
=
Matrix C 4×4
P4
P3
P2
P1
P4 on hold, is waiting for P2 and P3
P2 and P3 on hold, are waiting for P1
P3 idle
P2 idle
P1 idle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Time
Computation Time
Communication Time - among processors
Communication Time - assembling and delivering data

Figure E - 2 Schématisation des Algorithmes pré ITPMMA

Nouveau cadre pour le traitement parallèle

Dans cette thèse, un nouvel algorithme parallèle est proposé pour la multiplication parallèle des matrices basée sur la reconstruction du problème de multiplication en un ensemble de tâches indépendantes, de sorte que chaque processeur ne repose pas sur un autre processeur pour traiter certaines tâches, car chaque tâche est indépendante. Figure E-3 schématise la multiplication parallèle en utilisant le nouvel algorithme qui est appelé « Independent Tasks Parallel Matrix Multiplication Algorithm » (ITPMMA). Au lieu de décomposer les données entre les processeurs; nous distribuons les tâches entre les processeurs. Alors, le processeur P1 devrait produire la première ligne de la matrice C (voir Figure E-2), tandis que le processeur P2 devrait produire la deuxième ligne de la matrice C, le processeur P3 devrait produire la troisième, enfin processeur P4 devrait produire la quatrième ligne de la matrice

C. Pour cela, nous avons besoin de seulement 4 unités de temps pour accomplir les tâches. Il est à noter qu'il n'y a pas de communication entre les processeurs et pas de temps pour assembler les résultats; seulement pour le livrer. L'une des avancées majeures de l'algorithme ITPMMA est l'efficacité de l'utilisation des processeurs, aucun processeur n'est en attente ou inactif. L'efficacité de chaque processeur est 4/5 = 90%.

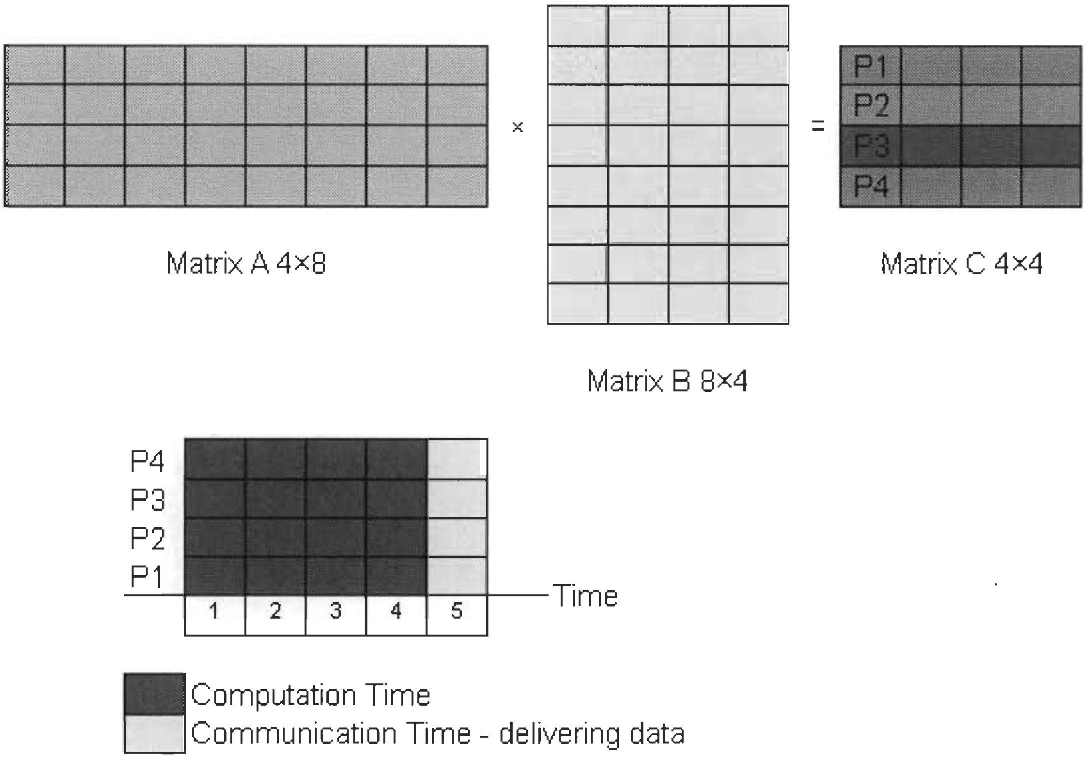


Figure E - 3 Schématisation de l'Algorithme ITPMMA

ITPMMA

ITPMMA est un algorithme pour la multiplication parallèle de matrices. Contrairement aux autres algorithmes de multiplication de matrice parallèles présents dans la littérature, l'ITPMMA propose un reformatage du processus de multiplication de matrice en plusieurs opérations indépendantes de multiplications vectorielles. Chaque opération de multiplication vectorielle est affectée à un seul processeur, afin d'éviter toute dépendance de données et temps pour transfert de données processeur à processeur. La figure E-4 montre l'organigramme de l'ITPMMA.

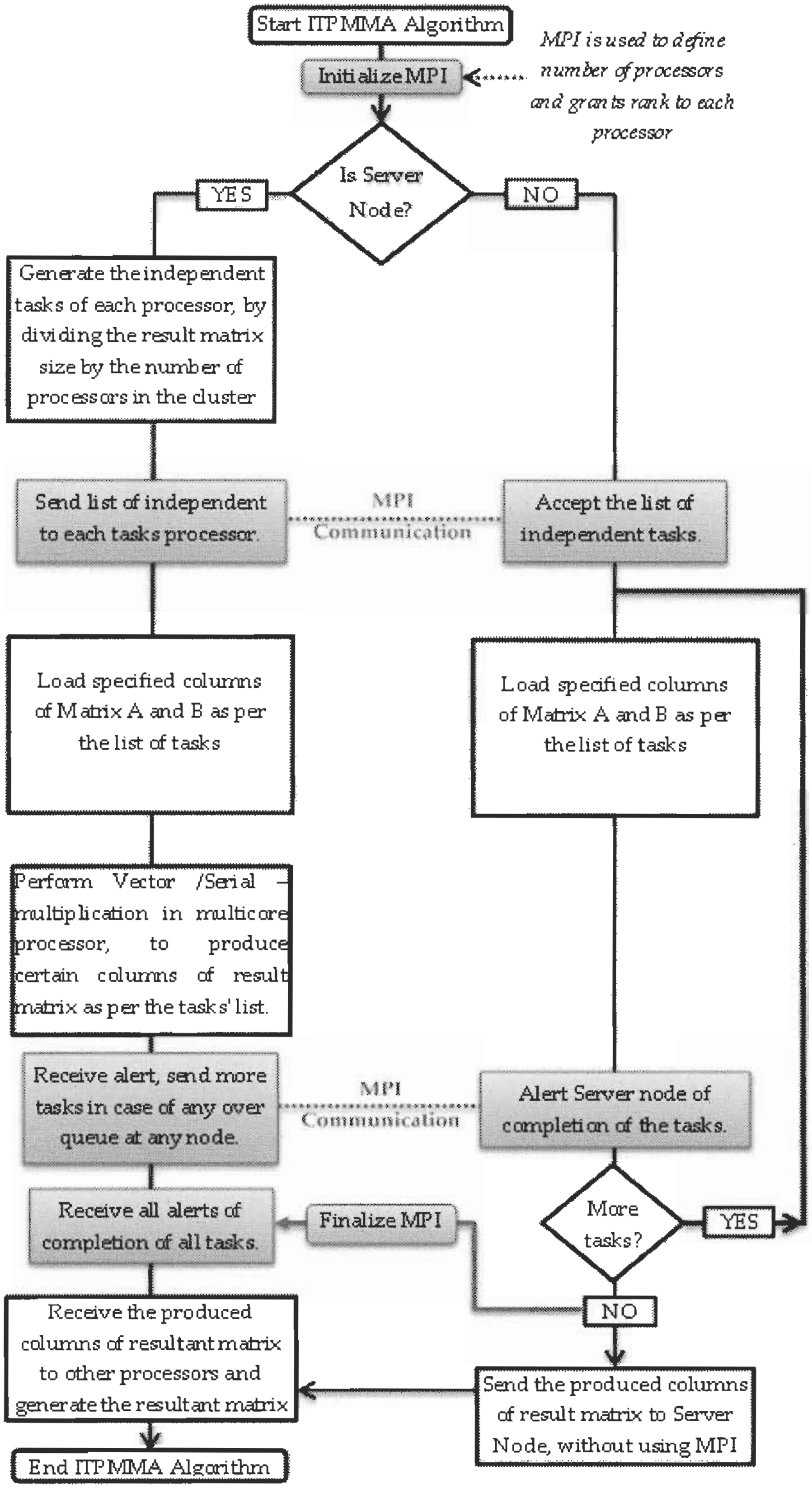

Start ITPMMA Algorithm
Initialize MPI
MPI is used to define number of processors and grants rank to each processor
Is Server Node?
YES
NO
Generate the independent tasks of each processor, by dividing the result matrix size by the number of processors in the cluster
Send list of independent to each tasks processor.
MPI Communication
Accept the list of independent tasks.
Load specified columns of Matrix A and B as per the list of tasks
Load specified columns of Matrix A and B as per the list of tasks
Perform Vector /Serial – multiplication in multicore processor, to produce certain columns of result matrix as per the tasks' list.
Receive alert, send more tasks in case of any over queue at any node.
MPI Communication
Alert Server node of completion of the tasks.
More tasks?
YES
Receive all alerts of completion of all tasks.
Finalize MPI
NO
Receive the produced columns of resultant matrix to other processors and generate the resultant matrix
Send the produced columns of result matrix to Server Node, without using MPI
End ITPMMA Algorithm

Figure E - 4 organigrammes de l' Algorithme ITPMMA

L'algorithme ITPMMA utilise la bibliothèque MPI pour définir le nombre de processeurs actifs sur le cluster et les classer. En outre, la bibliothèque MPI est utilisée pour transférer les listes de tâches à des processeurs et de transmettre des alertes d'accomplissement des tâches de différents transformateurs au processeur de nœud. Ainsi, le nœud de serveur:

1. Définit les processeurs disponibles dans le cluster parallèle et les classes.
2. Envoie pour chaque liste de processeurs des tâches à effectuer.
3. Reçoit une alerte d'achèvement de chaque processeur lorsque toutes les tâches assignées sont effectuées.
4. Envoie des tâches supplémentaires pour les processeurs possédant une grande puissance de calcul, une fois la liste initiale des tâches transmises à ces processeurs ait été effectuée. Ces tâches seront transférées à partir de processeurs à faible puissance de calcul.

La bibliothèque MPI ne sera pas utilisée pour échanger les données, car toutes les tâches sont indépendantes. Aucun processeur ne reçoit des données d'un autre processeur pour compléter ses travaux. Aucun processeur ne communique non plus avec le processeur de nœud non-serveur.

Quatre cas différents peuvent se manifester :

1. Multiplication de matrices carrées avec une taille de la matrice résultante qui est un multiple du nombre de processeurs utilisés en parallèle. Par exemple, A4x4 × B4x4 = C4x4, en utilisant quatre processeurs en parallèle.

2. Multiplication de matrices carrées avec une taille de la matrice résultante qui n'est pas un multiple du nombre de processeurs utilisés en parallèle. Par exemple, A12x12 × B12x12 = C12x12, et pour huit processeurs en parallèle.

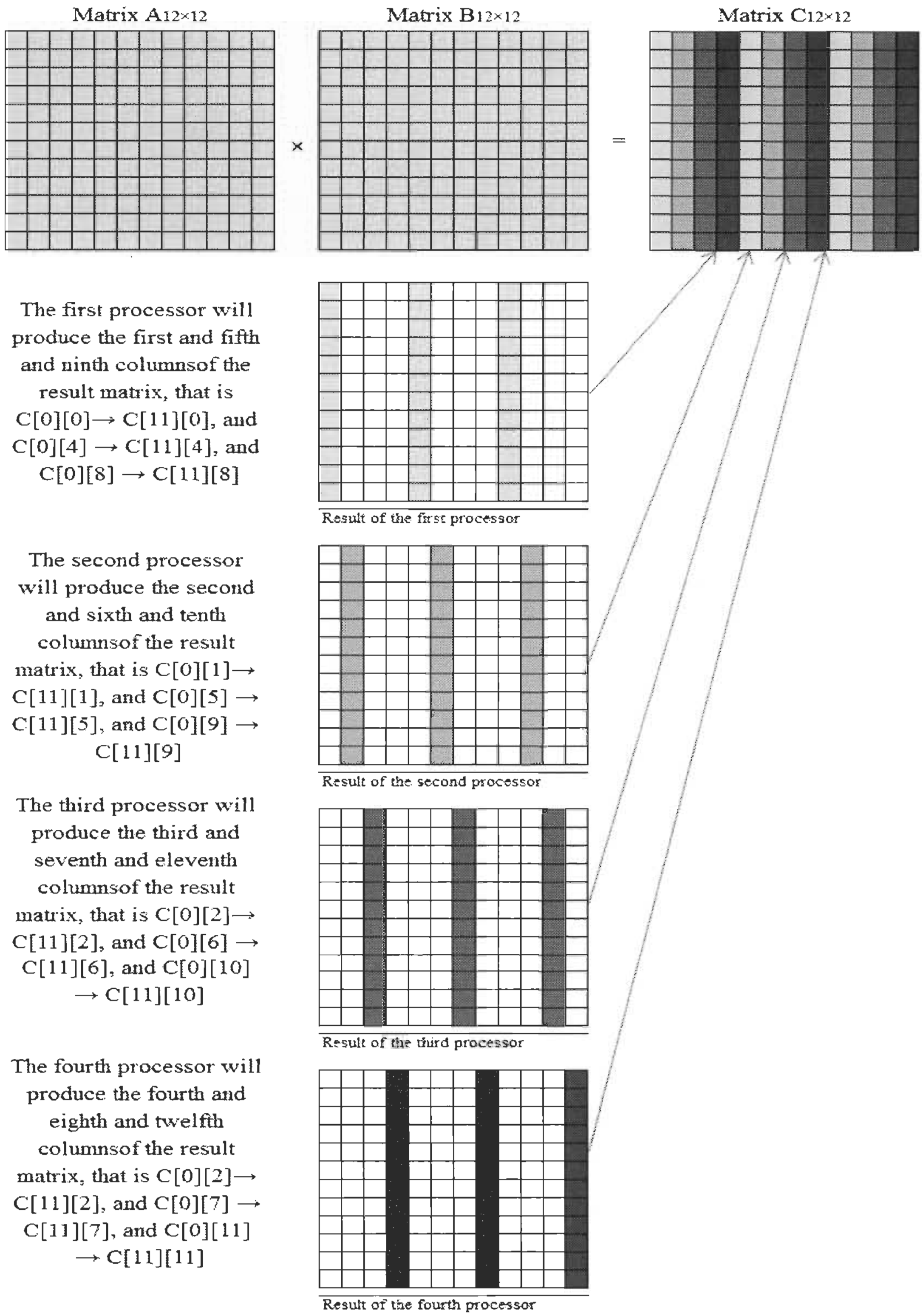
3. Multiplication de matrices non carrées, et la taille de la matrice résultante est un multiple du nombre de processeurs utilisés en parallèle. Par exemple, A12x12 × B12x16 = C12x16, en utilisant quatre processeurs en parallèle.

4. Multiplication de matrices non carrées avec une taille de la matrice résultante qui n'est pas un multiple du nombre de processeurs utilisés en parallèle. Par exemple, A12x12 × B12x18 = C12x18, pour quatre processeurs en parallèle.

Les deuxième et quatrième exemples aideront à montrer comment l'algorithme ITPMMA abordera le problème de l'équilibre de charge. D'autre part, tous les algorithmes précédents pour la multiplication parallèle de matrices sont optimisés dans le cas de multiplication de matrices carrées.

Multiplication de matrice carrée, la taille de la matrice résultante est multiple du nombre de processeurs en parallèle

Figure E - 5 montre la mise en œuvre de l'ITPMMA dans le cas du produit A12x12 × B12x12 = C12x12, en utilisant quatre processeurs parallèles. Chaque processeur produira trois colonnes de la matrice C. Le processeur P0 produira trois colonnes : la première, la cinquième et la neuvième colonne de la matrice C12 × 12. Le processeur P1 produira la deuxième, la sixième et la dixième colonne. P2 produira la troisième, la septième et la onzième colonne. Enfin, P3 produira la quatrième, la huitième et la douzième colonne.


Matrix A12×12
×
Matrix B12×12
=
Matrix C12×12
Result of the first processor
Result of the second processor
Result of the third processor
Result of the fourth processor

The first processor will produce the first and fifth and ninth columnsof the result matrix, that is C[0][0]→ C[11][0], and C[0][4] → C[11][4], and C[0][8] → C[11][8]

The second processor will produce the second and sixth and tenth columnsof the result matrix, that is C[0][1]→ C[11][1], and C[0][5] → C[11][5], and C[0][9] → C[11][9]

The third processor will produce the third and seventh and eleventh columnsof the result matrix, that is C[0][2]→ C[11][2], and C[0][6] → C[11][6], and C[0][10] → C[11][10]

The fourth processor will produce the fourth and eighth and twelfth columnsof the result matrix, that is C[0][2]→ C[11][2], and C[0][7] → C[11][7], and C[0][11] → C[11][11]

Figure E - 5 × A12x12 B12x12 utilisant l'algorithme ITPMMA pour une multiplication matricielle en parallèle

D'autres exemples sont présentés dans les chapitres 3 et 4.

Propriétés de l'ITPMMA

1. Aucun processeur ne reste inactif à attendre la sortie de d'autres processeurs sachant que toutes les tâches sont indépendantes.
2. Aucun coût pour le temps de communication entre les processeurs en terme de transfert de données. La communication est seulement entre le processeur de nœud du serveur et d'autres processeurs pour l'envoi de la liste des tâches.
3. Chaque processeur utilise sa pleine capacité, comme les noyaux multiples et la mémoire cache, pour compléter la tâche actuelle, aussi vite que possible. Cela implique que l'algorithme ITPMMA prend en charge la structure des processeurs afin d'accomplir la tâche dans le temps le plus court.
4. La charge est équilibrée sachant que les tâches sont déjà équilibrées entre les processeurs. Dans le cas où un processeur a terminé ses tâches avant d'autres processeurs, le processeur serveur sera alerté et pourra rediriger des tâches (s). Le phénomène de sur file d'attente pourrait se produire lorsque des processeurs d'architectures différents avec différentes capacités sont impliqués dans le cluster parallèle.

Accélération des calculs à l'aide de l'algorithme ITPMMA

En effectuant plusieurs expérimentations sur CLUMEQ supercomputer, nous avons obtenu les résultats suivants. Tableau E - 1 et Figure E-6 montrent des résultats de multiplication de matrices de tailles différentes sur 16 processeurs, où la taille des matrices est un multiple du nombre de processeurs.

Tableau E - 1 Différentes tailles de matrices avec 16 processeurs en parallèle, où la taille des matrices est un multiple du nombre de processeurs

Algorithm	Matrix Size							
	128	512	1024	2048	4096	8192	16384	32768
	Processing Time							
Serial	40,00	222,00	535,00	998,00	1900,00	3810,00	7900,00	16300,00
ITPMMA	3,00	17,00	40,00	75,00	130,00	251,00	499,00	1098,00
Cannon	11,00	65,00	152,00	320,00	605,00	1400,00	2700,00	6010,00
Fox	14,00	88,00	212,00	430,00	820,00	1701,00	3980,00	7150,00
ITPMMA Task-Adjustment	0	5	112	181	356	455	565	601
	Speed up							
ITPMMA	13,33	13,06	13,38	13,31	14,62	15,18	15,83	14,85
Cannon	3,64	3,42	3,52	3,12	3,14	2,72	2,93	2,71
Fox	2,86	2,52	2,52	2,32	2,32	2,24	1,98	2,28

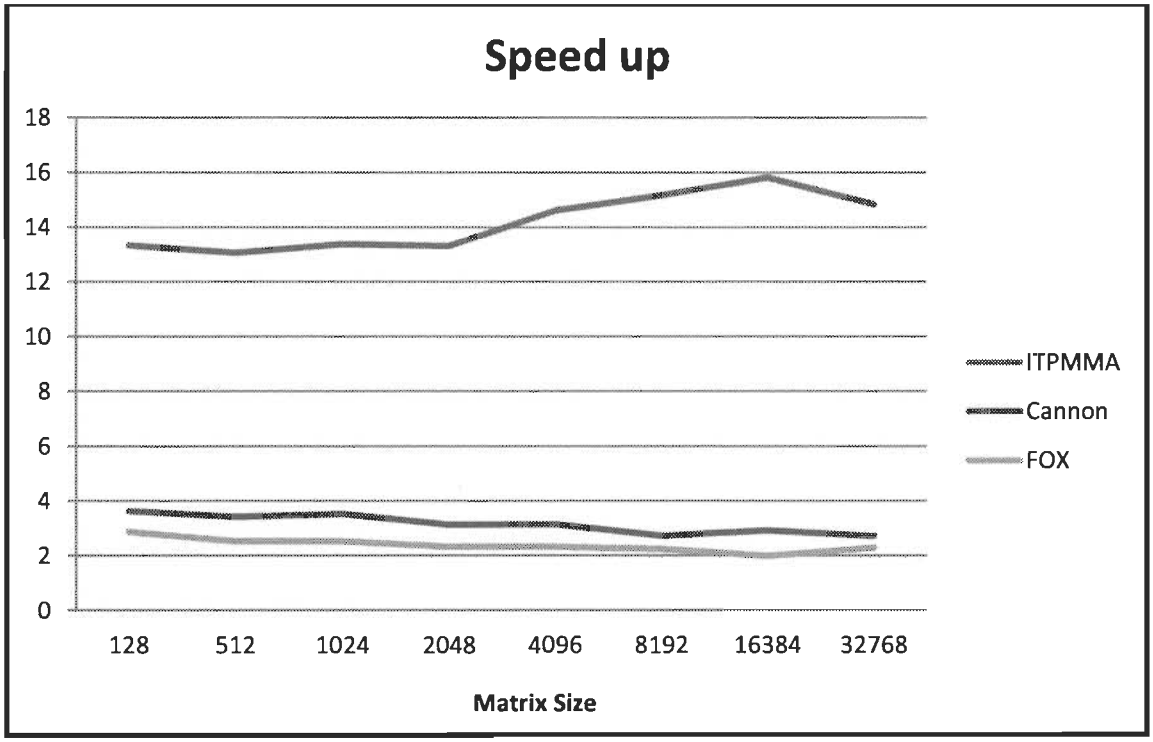


Figure E - 6 Comparaison des performances dans le cas où la taille des matrices est un multiple du nombre de processeurs.

Tableau E - 2 et Figure E-7 montrent les résultats de multiplication de matrices de tailles différentes sur 16 processeurs, où la taille des matrices n'est pas un multiple du nombre de processeurs (équilibrage de charge sera pris en charge par l'algorithme ITPMMA).

Tableau E – 2 multipliant les matrices de tailles différentes sur 16 processeurs, où la taille des matrices n'est pas un multiple du nombre de processeurs

Algorithm	Matrix Size							
	100	500	700	1000	1200	2000	3000	5000
Processing Time								
Serial	39	210	322	525	645	978	1499	2494
ITPMMA	2.50	14.00	21.00	34.00	42.00	63.00	98.00	166.00
Cannon	11.00	66.00	150.00	150.00	270.00	270.00	389.00	700.00
Fox	16.00	80.00	210.00	210.00	391.00	391.00	588.00	978.00
Speed up								
ITPMMA	15.60	15.00	15.33	15.44	15.36	15.52	15.30	15.02
Cannon	3.55	3.18	2.15	3.50	2.39	3.62	3.85	3.56
Fox	2.44	2.63	1.53	2.50	1.65	2.50	2.55	2.55

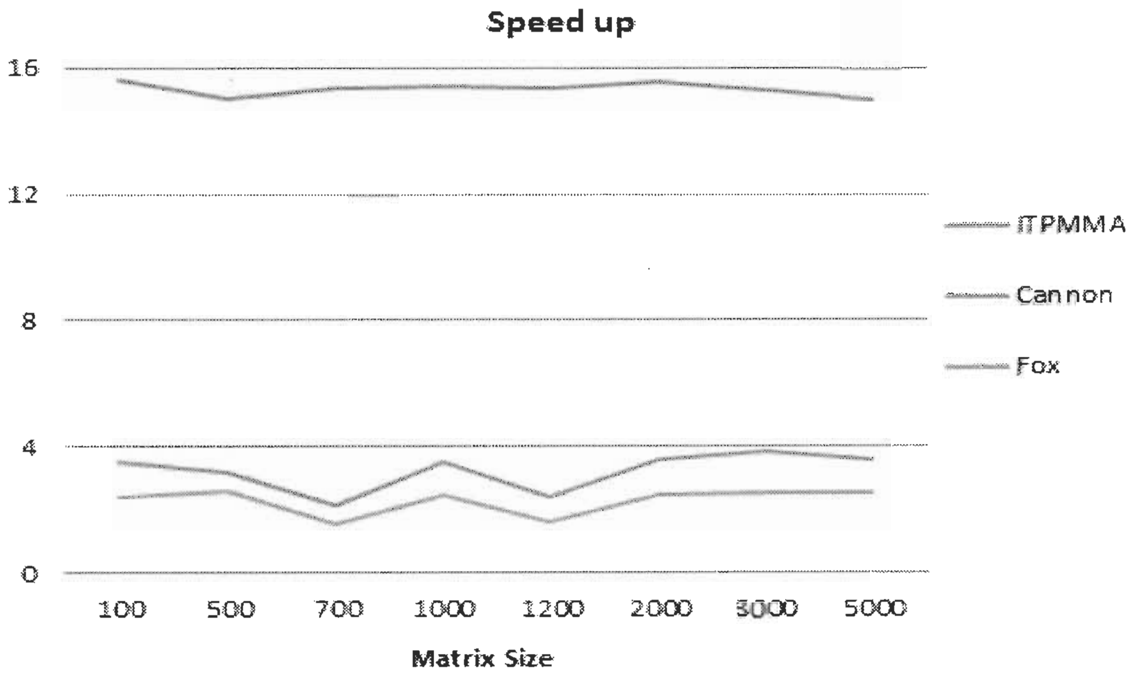


Figure E - 7 Comparaison des performances dans le cas où la taille des matrices n'est pas un multiple du nombre de processeurs.

Les troisième et quatrième groupes d'expérimentation sont présentés dans le Tableau E - 3 et la figure E-8. Nous avons considéré davantage de restriction aux algorithmes de Cannon et Fox pour générer des blocs lorsque les matrices sont non carrées et en plus, la taille des matrices n'est pas un multiple du nombre de de processeurs utilisés.

Tableau E – Produit de matrices non carrées - 750 × 700 - avec un nombre différent de processeurs

Algorithm	Number of Processors					
	2	4	16	32	64	128
ITPMMA	112	58	30	16	9	5
Cannon	589.00	297.00	152.00	120.00	85.00	65.00
Fox	756.00	400.00	212.00	135.00	92.00	75.00

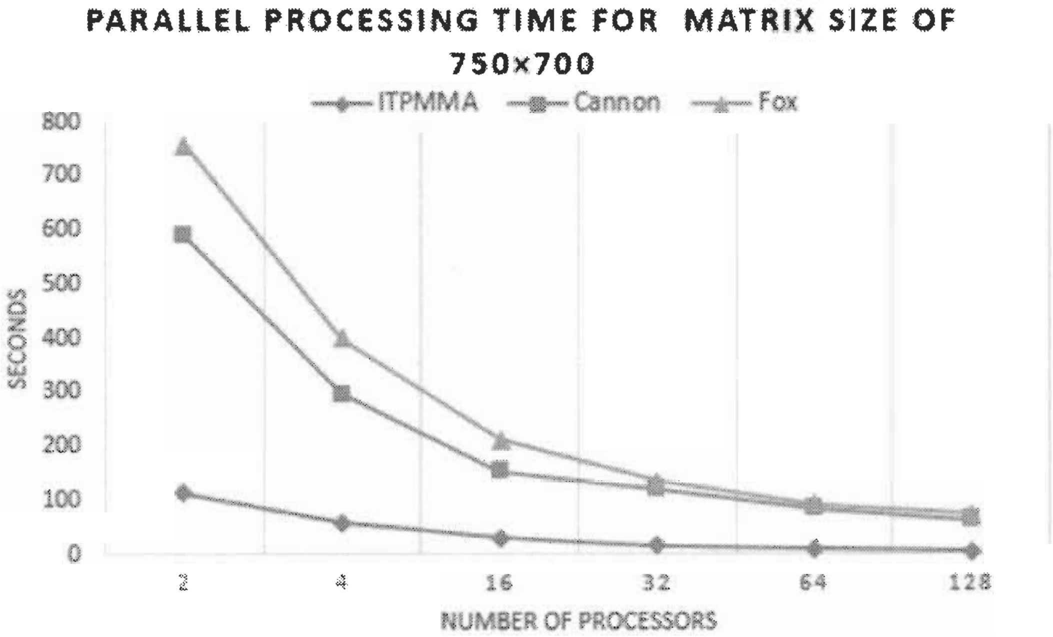


Figure E - 8 Délai de traitement parallèle ITPMMA algorithme contre les deux l'algorithme Cannon, et l'Algorithme Fox où la taille des matrices de 750 × 700 n'est pas multiple du nombre de processeurs

Les deux derniers tests montrent clairement que la multiplication de deux matrices de taille 750 × 700 en utilisant l'algorithme proposé ITPMMA présente de meilleurs résultats que les algorithmes Cannon et Fox. La Figure E-9 montre également l'efficacité de la proposition lorsque la taille des matrices est augmentée; et ceci pour différents nombres de processeurs..

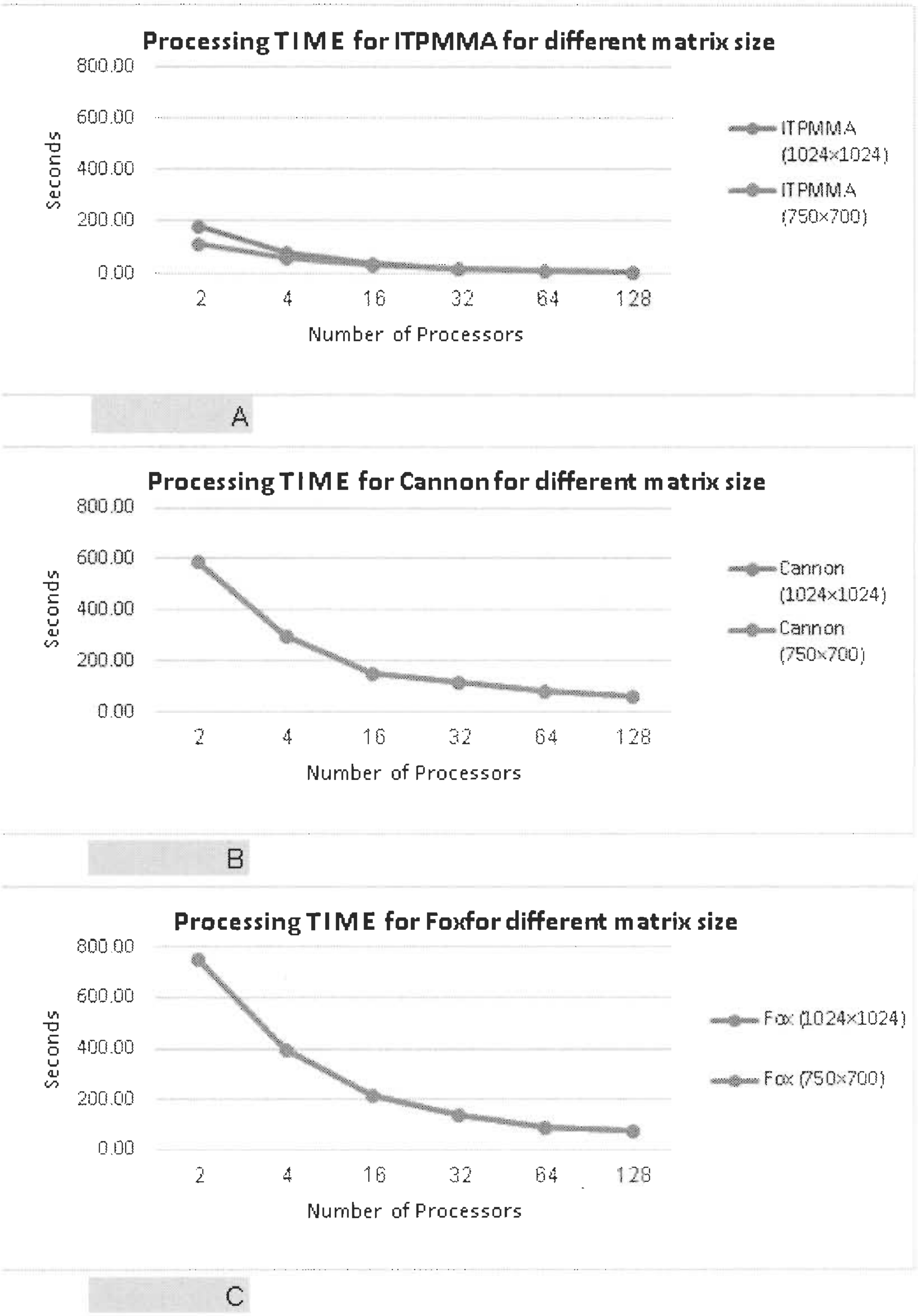


Figure E - 9 Délai de traitement parallèle pour les multiplications de matrices de deux tailles différentes de 750 × 700 et 1024 × 1024. A: L'algorithme ITPMMA, B: L'Algorithme Cannon, C: L'Algorithme Fox.

Pour les calculs d'efficacité, le Tableau E-4, la Figure E-10 et Figure E-11 présentent les résultats dans le cas d'une multiplication de matrices carrées avec un nombre de processeurs multiple de la taille des matrices.

Tableau E - 4 matrices de taille fixe avec un nombre différent de processeurs, où la taille des matrices est multiple du nombre de processeurs

Algorithm	Number of Processors					
	2	4	16	32	64	128
ITPMMA	180.00	79.00	40.50	20.30	10.10	5.06
Cannon	589.00	297.00	152.00	120.00	85.00	65.00
Fox	756.00	400.00	212.00	135.00	92.00	75.00

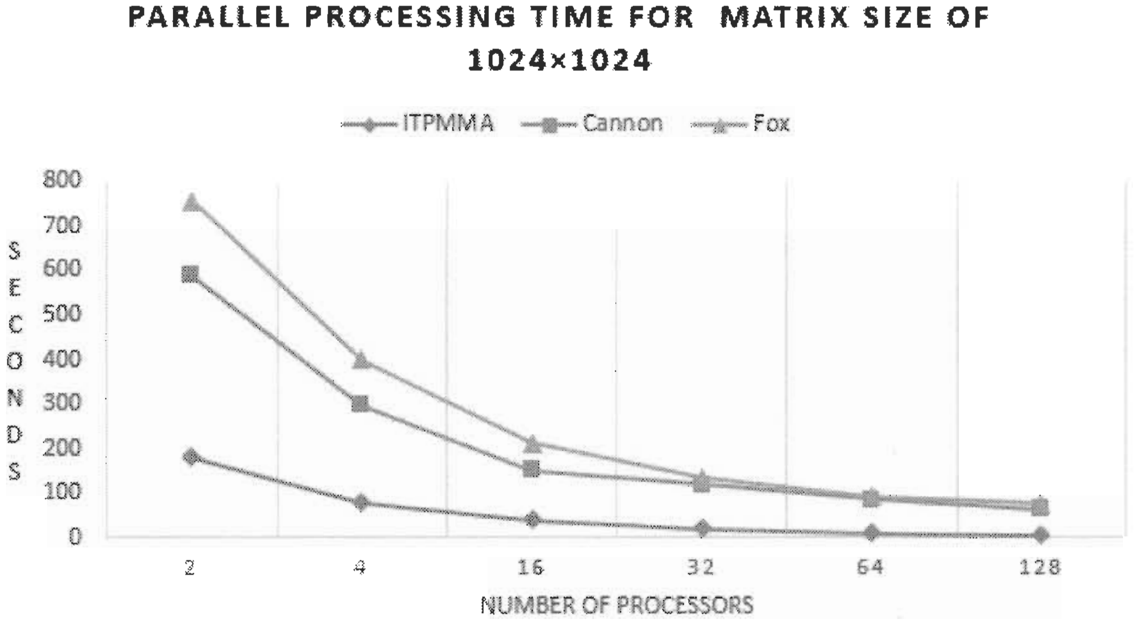


Figure E - 10, Temps pour traitement parallèle de l'algorithme ITPMMA contre l'algorithme Cannon et l'Algorithme de Fox où la taille des matrices est multiple du nombre de processeurs pour les matrices de 1024 × 1024.

Par exemple, pour multiplier les matrices de la taille 1024 en utilisant l'algorithme Cannon sur 64 processus, 85 secondes sont nécessaires. Alors que l'algorithme proposé permet d'exécuter le même produit matriciel en seulement 79 secondes et en utilisant que 4 processeurs. Dans ce cas, l'algorithme ITPMMA présente une efficacité 64/4 = 16 fois plus grande.

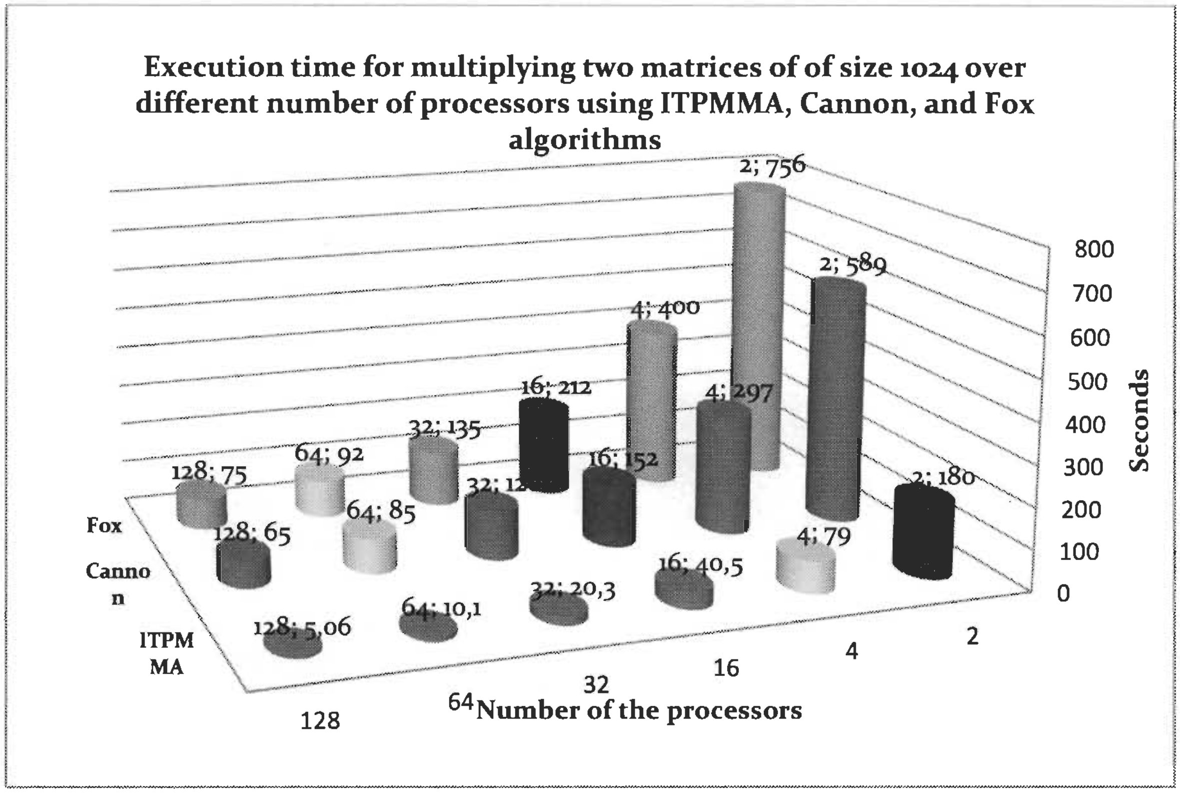


Figure E - 11 Le temps d'exécution par processeur pour ITPMMA, Cannon, et Fox.

La technique de décomposition 1D en grappe

Une mise en œuvre de tous les problèmes numériques en tâches indépendantes n'est pas toujours possible. Nous avons développé une nouvelle technique de décomposition de données, appelé «cluster 1-dimension decomposition technique» pour surmonter la limitation de différentes techniques de décomposition de données qui ont été utilisées par différents algorithmes parallèles. Les techniques de décomposition à une dimension (1D) et à 2 dimensions (2D) dominent depuis le début des propositions d''algorithmes parallèles. Les techniques de décomposition de données impliquent deux tâches:

a) Mise en correspondance de tableau processus en grille de n-dimensions.

b) Distribution des données sur la grille de processeurs.

Équation de Laplace en utilisant la méthode itérative Gauss-Seidel en grappe 1D

La résolution de l'équation de Laplace en utilisant la méthode de Gauss-Seidel sera utilisée comme méthode de référence dans le domaine de la décomposition de données. Figure E-12 montre la décomposition de données en 2D. L'équation est résolue en utilisant la méthode de Gauss-Seidel en deux environnements parallèles, 4 processeurs et 16 processeurs. Le Tableau E - 5 montre les résultats du calcul de l'équation de Laplace en utilisant la méthode de Gauss-Seidel utilisant la matrice de données de 48, 96 et 192, et en utilisant 4 et 16 processeurs en parallèle. On constate principalement la dépendance des données.

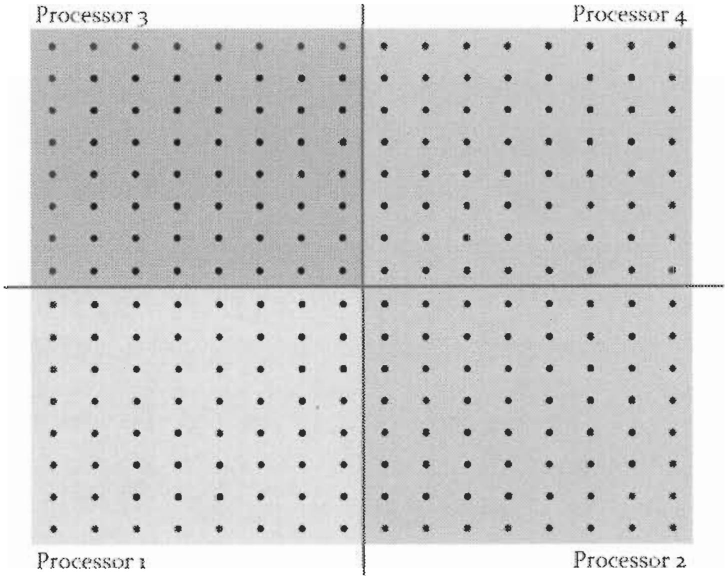
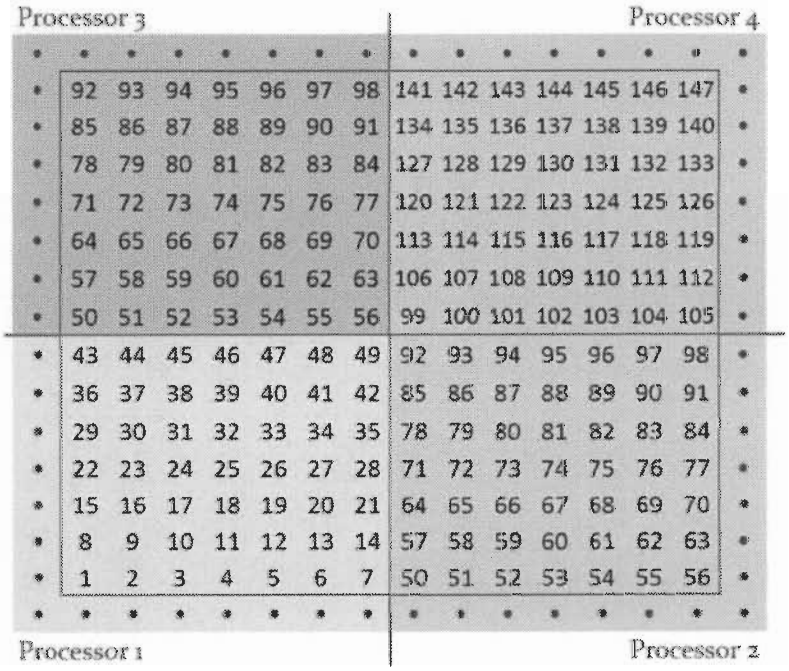


Figure E - 12, décomposition 2D.

Tableau E - 5 La Solution parallèle de l'équation Laplace en utilisant la méthode itérative de Gauss-Seidel sur 2D

Teat ID	1	2	3	4	5	6
Matrix Size (n)	48	96	192	48	96	192
Number of processors (nproc=nblock*nblock)	4	4	4	16	16	16
Decomposition (Number of the blocks)	4	4	4	16	16	16
Node edge (nodeedge)	24	48	96	12	24	48
ts (Serial processing time)	1	2	8	1	2	8
tp (Parallel processing time)	0.866	3.192	5.239	30.845	166.303	254.562
Speed up	1.155	0.626	1.527	0.032	0.012	0.031
Efficiency	0.288	0.156	0.381	0.002	7.50E-04	0.002

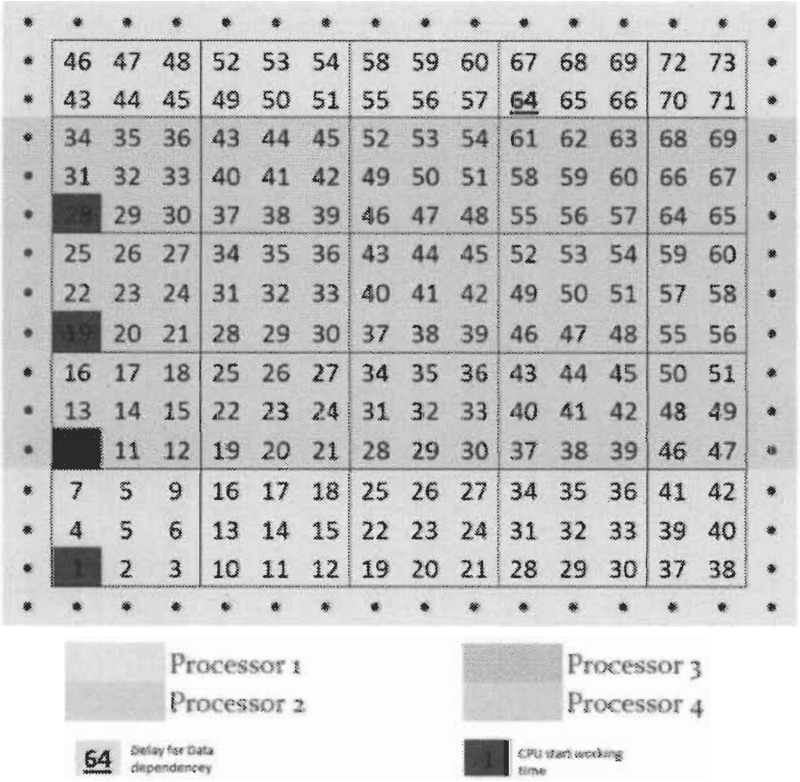


•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
•	92	93	94	95	96	97	98	141	142	143	144	145	146	147	•
•	85	86	87	88	89	90	91	134	135	136	137	138	139	140	•
•	78	79	80	81	82	83	84	127	128	129	130	131	132	133	•
•	71	72	73	74	75	76	77	120	121	122	123	124	125	126	•
•	64	65	66	67	68	69	70	113	114	115	116	117	118	119	•
•	57	58	59	60	61	62	63	106	107	108	109	110	111	112	•
•	50	51	52	53	54	55	56	99	100	101	102	103	104	105	•
•	43	44	45	46	47	48	49	92	93	94	95	96	97	98	•
•	36	37	38	39	40	41	42	85	86	87	88	89	90	91	•
•	29	30	31	32	33	34	35	78	79	80	81	82	83	84	•
•	22	23	24	25	26	27	28	71	72	73	74	75	76	77	•
•	15	16	17	18	19	20	21	64	65	66	67	68	69	70	•
•	8	9	10	11	12	13	14	57	58	59	60	61	62	63	•
•	1	2	3	4	5	6	7	50	51	52	53	54	55	56	•
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•

Figure E - 13 La technique de décomposition 2D, les unités de temps total est 147 unités.

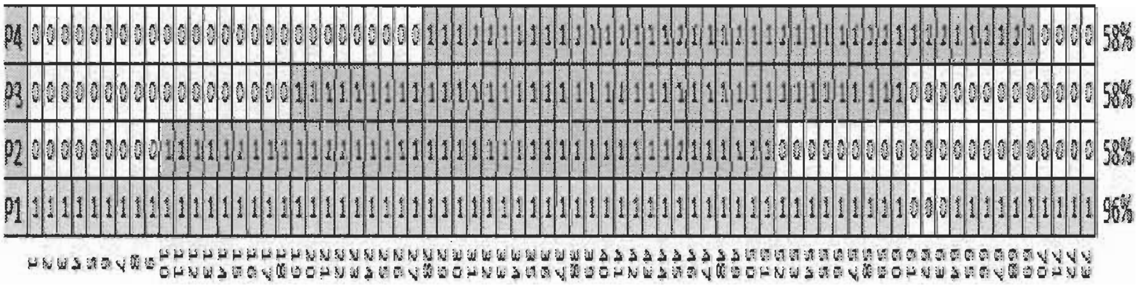
Figure E - 14, schématise l'algorithme proposé. Les unités du temps de traitement pour chaque processeur utilisant «cluster 1D technique de décomposition». Le nombre d'unités de temps est de 73 au lieu de 147. Figure E - 15 montre l'utilisation de chaque processeur. La vitesse est jusqu'à 196/73 = 2,31, donc l'efficacité = 2,31 / 4 = 57,9. Le

Tableau E - 6, montre les résultats du calcul de l'équation de Laplace en utilisant la méthode de Gauss-Seidel sur 1D cluster, en utilisant trois dimensions de la matrice de données de 48, et 96 et 192, en 4 et 16 processeurs en parallèle. Les avantages de la technique de décomposition cluster 1-dimensions pa rapport à la décomposition 2D sont évidents.



•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
•	46	47	48	52	53	54	58	59	60	67	68	69	72	73	•
•	43	44	45	49	50	51	55	56	57	64	65	66	70	71	•
•	34	35	36	43	44	45	52	53	54	61	62	63	68	69	•
•	31	32	33	40	41	42	49	50	51	58	59	60	66	67	•
•	28	29	30	37	38	39	46	47	48	55	56	57	64	65	•
•	25	26	27	34	35	36	43	44	45	52	53	54	59	60	•
•	22	23	24	31	32	33	40	41	42	49	50	51	57	58	•
•	19	20	21	28	29	30	37	38	39	46	47	48	55	56	•
•	16	17	18	25	26	27	34	35	36	43	44	45	50	51	•
•	13	14	15	22	23	24	31	32	33	40	41	42	48	49	•
•		11	12	19	20	21	28	29	30	37	38	39	46	47	•
•	7	5	9	16	17	18	25	26	27	34	35	36	41	42	•
•	4	5	6	13	14	15	22	23	24	31	32	33	39	40	•
•	1	2	3	10	11	12	19	20	21	28	29	30	37	38	•
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•

Figure E - 14,« cluster 1D technique de décomposition» les unités de temps total est 73 unités



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	
P4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	58%
P3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	58%
P2	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	58%
P1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	96%

Figure E - 15 Cluster 1D décomposition - utilisation de processeurs.

Tableau E - 6, La Solution parallèle de l'équation Laplace utilisant méthode itérative de Gauss-Seidel sur 1D cluster

Teat ID	7	8	9	10	11	12
Matrix Size (n)	**48**	**96**	**192**	**48**	**96**	**192**
Number of processors (nproc=nblock*nblock)	**4**	**4**	**4**	**16**	**16**	**16**
Decomposition (Number of the blocks)	**48/12=4**	**96/12=8**	**192/12=16**	**48/12=4**	**96/12=8**	**192/12=16**
Node edge (nodeedge)	96	384	1536	24	96	384
ts (Serial processing time)	1	2	8	1	2	8
tp (Parallel processing time)	0.611	1.076	3.61	0.204	0.448	1.569
Speed up	1.637	1.858	2.216	4.902	4.464	5.099
Efficiency	0.409	0.465	0.554	0.306	0.279	0.319

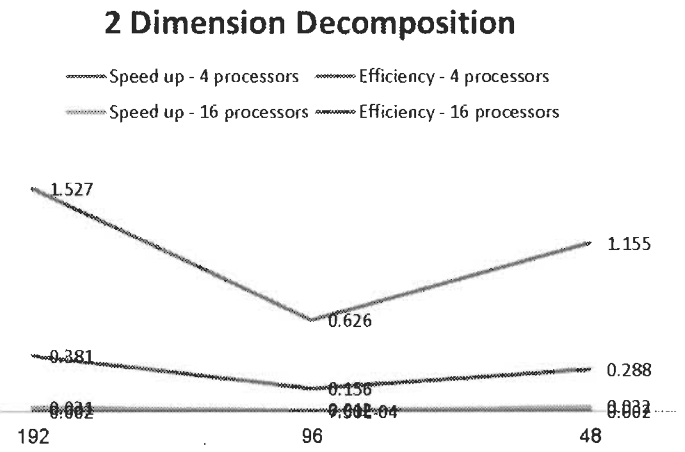


Figure E - 16 L'accélération et l'efficacité de la solution itérative Gauss-Seidel parallèle de l'équation Laplace en 2 dimensions

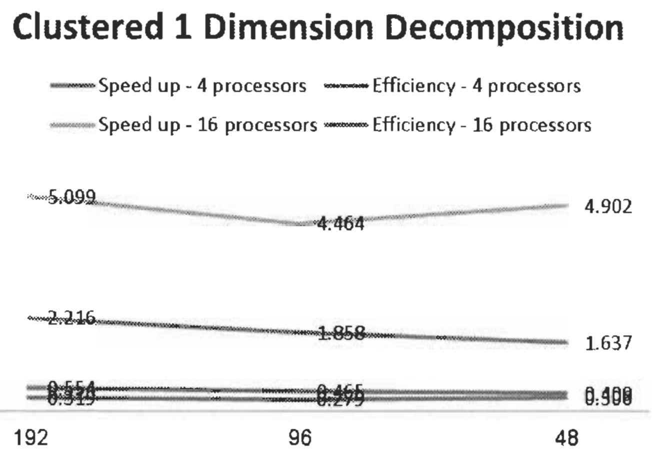


Figure E - 17 L'accélération et l'efficacité de la solution itérative Gauss-Seidel parallèle de l'équation Laplace en 2 dimensions

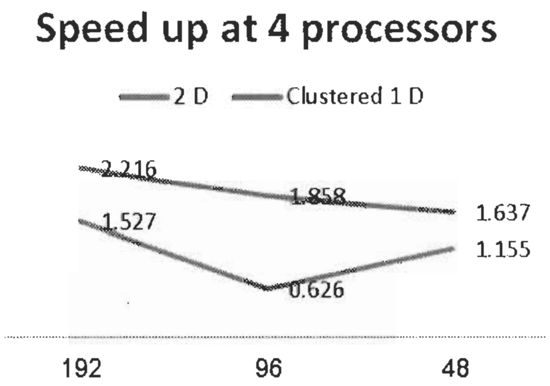


Figure E - 18 Accélération de la solution parallèle Gauss-Seidel itérative de l'équation Laplace avec les deux techniques de décomposition de données à 4 processeurs.

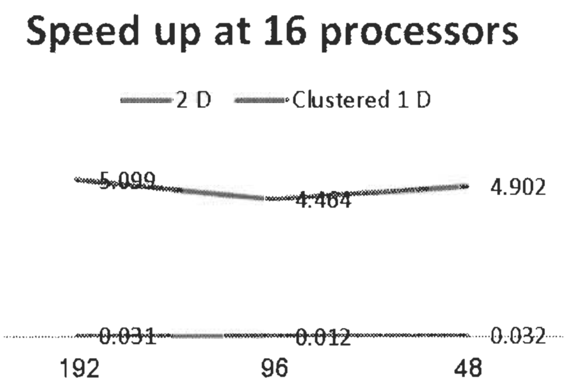


Figure E - 19 Accélération de la solution parallèle Gauss-Seidel itérative de l'équation Laplace avec les deux techniques de décomposition de données à 16 processeurs