NEW SEQUENTIAL AND PARALLEL DIVISION FREE METHODS FOR DETERMINANT OF MATRICES

SHARMILA KARIM

DOCTOR OF PHILOSOPHY

UNIVERSITI UTARA MALAYSIA

2013

Permission to Use

In presenting this thesis in fulfilment of requirements for a postgraduate degree from Universiti Utara Malaysia, I agree that the University Library may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by my supervisor(s) or, in their absence, by the Dean of the Awang Had Salleh Graduate School. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall given to me and to Universiti Utara Malaysia for any scholarly use which may be made of any material from my thesis.

Requests for permission to copy or to make other use of materials in this thesis, in whole or in part, should be addressed to:

Dean of Awang Had Salleh Graduate School
UUM College of Arts and Sciences
Universiti Utara Malaysia
06010 Sintok
Kedah Darul Aman

Abstrak

Penentu berperanan penting dalam kebanyakan aplikasi aljabar linear. Pencarian penentu menggunakan kaedah pembahagian bukan bebas akan menghadapi masalah sekiranya pemasukan matriks diwakili dalam ungkapan nisbah atau polinomial dan juga apabila kesilapan titik apungan wujud. Bagi mengatasi masalah ini, kaedah pembahagian bebas digunakan. Dua kaedah pembahagian bebas yang biasa digunakan dalam pencarian penentu adalah pendaraban silang dan pengembangan kofaktor. Walau bagaimanapun, pendaraban silang yang menggunakan Petua Sarrus hanya berhasil untuk matriks berperingkat kurang atau sama dengan tiga, sedangkan apabila berhadapan dengan matriks yang bersaiz besar, pengembangan kofaktor memerlukan pengiraan yang terlalu panjang dan rumit. Oleh itu, kajian ini berusaha membangunkan kaedah berjujukan dan kaedah selari yang baharu untuk mencari penentu bagi matriks. Kajian ini juga berhasrat untuk mengitlakkan Petua Sarrus bagi sebarang peringkat matriks segi empat sama berpandukan pilih atur yang diperolehi menggunakan set penjana. Dua strategi diperkenalkan bagi menjana set penjana yang berlainan iaitu operasi kitaran dan operasi saling tukar dua unsur. Beberapa hasil teori dan sifat matematik dalam penjanaan pilih atur dan penentuan penentu turut dibina bagi menyokong kajian ini. Keputusan berangka menunjukkan masa pengiraan kaedah baharu yang dicadangkan adalah lebih baik jika dibandingkan dengan kaedah sedia ada. Masa pengiraan kaedah berjujukan baharu yang dibangunkan tertakluk kepada penjanaan set penjana. Oleh demikian, dua strategi selari dibangunkan untuk menyelaraskan algoritma ini bagi mengurangkan masa pengiraan. Keputusan berangka turut menunjukkan bahawa kaedah selari berupaya mengira penentu lebih cepat berbanding kaedah berjujukan, khususnya apabila tugas diagihkan dengan sama rata. Kesimpulannya, kaedah baharu yang telah dibangunkan boleh diguna sebagai alternatif yang berdaya saing dalam pencarian penentu bagi matriks. .

Kata kunci: Penentu, Pilih atur, Set penjana, Kaedah tanpa pembahagi, Kaedah jujukan dan selari

Abstract

A determinant plays an important role in many applications of linear algebra. Finding determinants using non division free methods will encounter problems if entries of matrices are represented in rational or polynomial expressions, and also when floating point errors arise. To overcome this problem, division free methods are used instead. The two commonly used division free methods for finding determinant are cross multiplication and cofactor expansion. However, cross multiplication which uses the Sarrus Rule only works for matrices of order less or equal to three, whereas cofactor expansion requires lengthy and tedious computation when dealing with large matrices. This research, therefore, attempts to develop new sequential and parallel methods for finding determinants of matrices. The research also aims to generalise the Sarrus Rule for any order of square matrices based on permutations which are derived using starter sets. Two strategies were introduced to generate distinct starter sets namely the circular and the exchanging of two elements operations. Some theoretical works and mathematical properties for generating permutation and determining determinants were also constructed to support the research. Numerical results indicated that the new proposed methods performed better than the existing methods in term of computation times. The computation times in the newly developed sequential methods were dominated by generating starter sets. Therefore, two parallel strategies were developed to parallelise this algorithm so as to reduce the computation times. Numerical results showed that the parallel methods were able to compute determinants faster than the sequential counterparts, particularly when the tasks were equally allocated. In conclusion, the newly developed methods can be used as viable alternatives for finding determinants of matrices.

Keywords: Determinant, Permutation, Starter sets, Division free method, Sequential and parallel methods

Acknowledgement

First for all, I would like to express my deep sense of gratitude to Allah S.W.T for His Merciful and Kindness in giving me a chance to complete this study. I am highly indebted to Prof. Dr. Zurni bin Omar, Assoc. Prof. Dr. Haslinda binti Ibrahim, and Assoc. Prof. Dr. Khairil Iskandar bin Othman for their invaluable help and guidance during the course of research. I highly appreciated them for constantly encouraging me by giving their critics on my work. I am grateful to them for having given me the support and confidence.

I am also indebted to the Government of Malaysia (Ministry of Higher Education), and Universiti Utara Malaysia for sponsoring this research.

I would like to thank my family. I could not have made it to this level without their love and prayers. Gratitude to my parents, Haji Karim bin Karon and Hajjah Besah binti Nordin, and my siblings. They always believed in me and have always supported me.

Last but not least, I would like to thanks all my friends including Hasimah binti Sapiri, Azizah binti Mohd Rohni, Idariyana binti Mohd Seran, Fatimah Bibi binti Hamzah, Teh Raihana Nazirah binti Roslan, Hashibah binti Hamid, Hartima binti Abu Talib and INSPEM, UPM especially to Miss Hazwanie Abdullah for helping me in conducting HPC machines and INSPEM's staff for giving me a convenient office to work in.

Declaration Associated With This Thesis

Karim, S., Ibrahim, H., Omar, Z., Othman. K. I.& Sulaiman, M. (2009). Generating permutation based on starter sets, Eletronics *Proceeding of 5th Asian Mathematical Conference*,1, 90-96. available url: http://www.mat.usm.my/AMC 20200920 Proceedings/Pure/P134.PDF

Karim, S., Ibrahim, H., Omar, Z., Othman. K. I., & Sulaiman, M.(2009). New Recursive Method for Generating Permutation. *Prosiding Simposium Kebangsaan Sains Matematik ke-17*, 829-835.

Karim, S., Omar, Z. & Ibrahim, H. (2011). New Scheme in Determining nth Order Diagrams for Cross Multiplication Method via Combinatorial Approach. *Proceeding of World Academy of Science Engineering and Technology*, Issue 78, 641-644.

Karim, S., Ibrahim, H., Omar, Z., & Othman, K.I. (2011). Parallel Strategy for Starter Sets to List All Permutation Based on Cycling Restriction. *Proceeding of The Third International Conference Engineering and Technology*, 251-256.

Karim, S., Ibrahim, H.,& Omar, Z. (2011). New division free algorithm for finding the determinant, Far East Journal of Mathematical Sciences, 49, 43-55.

Karim, S., Omar, Z. & Ibrahim, H. (2011). Integrated strategy for generating permutation. International Journal of Contemporary Mathematical Sciences, Vol. 6, no. 24, 1167 - 1174.

Table of Contents

List of Figures

List of Tables

List of Appendices

Glossary of Terms

Abbreviation	Details
ATM	Across The Method
ATT	Across The Time
CO	Circular Operation
CP	Circular Permutation
D	Dimension
DFM	Division Free Method
GB	Giga Bytes
HPC	High Performance Computing
ID	IDentity
ISS	Initial Starter Sets
ISSG1	Initial Starter Sets Generator of first strategy
ISSG2	Initial Starter Sets Generator of second strategy
MPI	Message Passing Interface
nDFM	non Division Free Method
PDATM1	Permutation Determinant for Across The Method of circular strategy
PDATM2	Permutation Determinant for Across The Method of exchange strategy
PERATM1	Permutation for Across The Method of circular strategy
PERATM2	Permutation for Across The Method of exchange strategy
PERMUT1	Permutation of circular strategy
PERMUT2	Permutation of exchange strategy
PERMUTIT3	Permutation of iterative circular strategy
PERMUTDET1	Permutation determinant of circular strategy
PERMUTDET2	Permutation determinant of exchange strategy
PERMUTDETIT3	Permutation determinant of iterative circular strategy
RAM	Random Access Memory
RoCP	Reverse of Circular Permutation
SIMD	Single Instruction Multiple Data
SPU	Synergistic Processing Unit

CHAPTER ONE

INTRODUCTION TO DETERMINANT METHODS

1.1 Background of the Study

Matrices and determinants are the backbone of linear algebra (Bernstein, 2008). A determinant provides useful geometrical and algebraical information of a square matrix. Algebraically, a matrix has an inverse if and only if the determinant is not zero. This happens when the vectors are linearly independent. Meanwhile geometrically, the row entries of $n \times n$ matrix define the edges of a parallelepiped in n-dimensional space, of which the area and volume are the absolute value of the determinant of a square matrix for spaces R^2 and R^3 respectively.

The determinant has been the subject of study for over 200 years. The name determinant was introduced by Carl Friedrich Gauss (1777-1855) while discussing quadratic forms. The term determinant was used because it determined the properties of the quadratic form (O'Connor & Robertson, 1996). The theory of determinant was expanded gradually during the 18th century through the theory of equations in the work of Leibniz, Maclaurin, Cramer and Laplace (Rice & Torrence, 2006). Then it became an increasingly significant subject in the mathematical area by the 19th century.

The applications of determinant can be found in various areas for example in mathematical physics in which any solvable equation having a solution can be expressed as a determinant (Vein & Dale, 1999). The determinant is required in inverse kinematics singularity analysis of parallel manipulator which this manipulator is described as 6×6 transformation matrix (Luyang et al., 2006). Meanwhile from the statistical perspective, the determinant is used in normalizing the constant of the probability density function of the multivariate normal distribution, and is also involved in experimental design (Harville, 1997). In addition, the determinant is a beneficial tool in eigenvalue problems in which

that problem can be reduced to the problem of finding roots of a determinant polynomial (Abdi, 2007)

The definition of the determinant comes from the following theorem.

Theorem 1.1.1. *A function on the $n \times n$ matrices is called a* determinant function *if and only if it satisfies the following properties*

(i) The determinant is linear in each row if the other rows of the matrix are held fixed.

(ii) Let A be $n \times n$ matrix with two rows identical. Then $det(A) = 0$.

(iii) If I is the $n \times n$ identity matrix, then $det(I) = 1$.

One can then define the determinant as a unique function with the above properties (Schneider & Barker, 1989). In proving the above theorem, Leibniz formula can be obtained as follows:

$$det(A) = \sum_{(\sigma(1),\sigma(2),\ldots,\sigma(n))\in S_n} sign(\sigma) \cdot a_{1\sigma(1)} \cdot a_{2\sigma(2)} \ldots \cdot a_{n\sigma(n)}. \tag{1.1}$$

The summation is the set of all permutations σ of n elements. The sign of a permutation is defined in terms of the number of inversions as given below:

$$sign(\sigma) = (-1)^{\text{number of inversion } \sigma}. \tag{1.2}$$

The number of terms $a_{1\sigma(1)} \cdot a_{2\sigma(2)} \ldots \cdot a_{n\sigma(n)}$ in the sum equals $n!$. As n increases, the number of terms grows rapidly. Furthermore the arrangement of the terms are related to the generation permutation method. This formula (1.1) has been cited as a definition of determinant by a numerous researchers i.e. Kaltofen (1992), Iqbal (1995), Mahajan and Vinay (1997), Sengupta (1997), Rote (2001), Shin (2002), and Thongchiew (2007).

From this definition, a great deal of work has been done in seeking to find the efficient

ways of calculating determinants. These methods can be classified into two categories: direct methods and non-direct methods (Rezaee & Rezaifar, 2007).

In the direct method, a specific mathematical formula is used to obtain the determinant of matrix without converting the original matrix into other matrix forms(Rezaee & Rezaifar, 2007). Examples of the direct method are cross multiplication, cofactor expansion, condensation method and permutation. On the other hand, the non-direct methods involved matrix decomposition where the matrix is factorized into some different form (Simon & Blume, 1994). An example of the non-direct methods is the Gauss elimination (Rezaee & Rezaifar, 2007).

The most commonly used techniques for finding determinant are cross multiplication, cofactor expansion, and Gauss elimination. All these techniques were discussed in great details in many textbooks such as Anton (2000), Anton and Busby (2002), Brestscher (2009), Hasiung and Mao (1998), Johnson et al. (2002), Perry (1988), Wilde (1988), and Scneider et al. (1982). In this study, a survey was done on some existing determinant methods. The drawbacks of these methods are also highlighted.

1.1.1 A survey of Determinant Methods

Let $A = [a_{ij}]$ represents an arbitrary $n \times n$ matrix. The determinant of A is denoted by $|A|$ or det(A). The arbitrary determinant is det(A)= $|a_{ij}|_n = |C_1\ C_2\ C_3\ ...\ C_n|$ when represented in column indices.

(i) Cross multiplication

Pierre Frédéric Sarrus (1853) introduced the cross multiplication method which was

called the Sarrus Rule with a single third order diagram as follows:

$$\left|\begin{array}{ccc|cc} a_{11} & a_{12} & a_{13} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{23} & a_{21} & a_{22} \\ a_{31} & a_{32} & a_{33} & a_{31} & a_{32} \end{array}\right|$$

In order to construct this diagram, one needs to append the first two columns to the right of the matrix. Then multiply the elements on the main diagonal and its parallel line diagonal, and add them. The same procedure is applied to the secondary diagonal and its parallel diagonal, and adds up them. The determinant of this matrix is equal to summation of (main + parallel) diagonal products subtract to the summation of (secondary + parallel) diagonal products, i.e.

$$[a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32}] - [a_{13}a_{22}a_{31} + a_{11}a_{23}a_{32} + a_{12}a_{21}a_{33}]$$

The cross multiplication technique is easy to use especially when the size of the square matrix is small ($n \leq 3$)(Khattar, 2010). Osborn (1960) proved that this method cannot be extended to the fourth-order determinant by following a single diagram of third-order determinant principle. This is due to the fact that $n!$ different diagonal products are needed in order to find the determinant. He also did a survey over thirty textbooks chosen at random, in which each included a discussion of the determinant using the Sarrus Rule. Only one book stated that the Sarrus Rule is invalid for $n > 3$ and while most of the books mentioned that the scheme cannot be generalised to any n. However this does not imply that generalization is impossible.

In spite of Osborn work, Bankier (1961) and Pavlovic (1961) tried to extend Sarrus Rule and they derived the construction of the nth order diagram scheme based on $\frac{(n-1)!}{2}$ permutation column until $n = 5$. The permutation column is represented by the array of matrix column indices. Unfortunately, repetition of diagonal products existed for the fifth order diagram which was constructed from the $[1, 2, 5, 4, 3]$ and the $[1, 3, 4, 5, 2]$ permutation column as follows:

$$[1,2,5,4,3]$$

$$|A_{1,2,5,4,3}| = \begin{vmatrix} a_{11} & a_{12} & a_{15} & a_{14} & a_{13} & a_{11} & a_{12} & a_{15} & a_{14} \\ a_{21} & a_{22} & a_{25} & a_{24} & a_{23} & a_{21} & a_{22} & a_{25} & a_{24} \\ a_{31} & a_{32} & a_{35} & a_{34} & a_{33} & a_{31} & a_{32} & a_{35} & a_{34} \\ a_{41} & a_{42} & a_{45} & a_{44} & a_{43} & a_{41} & a_{42} & a_{45} & a_{44} \\ a_{51} & a_{52} & a_{55} & a_{54} & a_{53} & a_{51} & a_{52} & a_{55} & a_{54} \end{vmatrix}$$

$$[1,3,4,5,2]$$

$$|A_{1,3,4,5,2}| = \begin{vmatrix} a_{11} & a_{13} & a_{14} & a_{15} & a_{12} & a_{11} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{23} & a_{24} & a_{25} & a_{22} & a_{21} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{33} & a_{34} & a_{35} & a_{32} & a_{31} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{43} & a_{44} & a_{45} & a_{42} & a_{41} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{53} & a_{54} & a_{55} & a_{52} & a_{51} & a_{53} & a_{54} & a_{55} \end{vmatrix}$$

This clearly shows that the generation of specific $\frac{(n-1)!}{2}$ permutation columns for the formulation of nth order diagram is yet to be discovered.

Recently, Hajrijaz (2009) introduced three methods to determine the determinant of the third order matrix. For each method, six diagonals would be formed.

Consider matrix A

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

One of his methods said that elements a_{13} and a_{33} would be placed before the first row and third row respectively. Then elements a_{11} and a_{31} would be placed after the first row and third row respectively. It can be seen through the following example.

Example 1.1.2.

$$\begin{array}{c|ccc|c} a_{13} & a_{11} & a_{12} & a_{13} & a_{11} \\ & a_{21} & a_{22} & a_{23} & \\ a_{33} & a_{31} & a_{32} & a_{33} & a_{31} \end{array}$$

The process of elements product and its signs for six diagonals are similar to the Sarrus Rule for 3×3 matrix. The determinant of this matrix is equal to determinant using the Sarrus Rule. Unfortunately this new method only works for 3×3 matrix. The other two methods also showed the similar result.

(ii) Cofactor expansion

There is another way to compute the determinant of an $n \times n$ matrix A using the determinant of some of its submatrices as follows

$$det(A) = \sum^{n} (-1)^{i+j} a_{ij} |M_{ij}| \tag{1.3}$$

where $|M_{ij}|$ is the determinant of the $(n-1) \times (n-1)$ matrix obtained from A by omitting the ith row and jth column of A. $|M_{ij}|$ is called the minor of entry a_{ij} and $(-1)^{i+j}|M_{ij}|$ is called the cofactor of a_{ij}. The cofactor expansion method is performed by rewriting the determinant of an n by n matrix as the sum of products. The products are entries on a specific row (or column) and their cofactors, using the above equation. Each rewriting is called an expansion. This expansion is also called by *minor expansion* and *Laplace expansion* in which this method was introduced by Laplace in year 1772 (Muir, 1933). However many researchers had used term minor expansion in spite of cofactor expansion i.e. Horowitz and Sahni (1975), Gentleman and Johnson (1976), Griss (1976), Smit (1979), Sasaki and Kanada (1981), and Umeda and Sasaki (2006). The shortcoming of this method is that we need to reduce the size of a matrix to 2×2 or 3×3 in order to obtain the determinant. Furthermore this method required the $n!$ products and the process of finding the determinant are inductive where the cofactors themselves are determinants (Krattenthaler, 1999;

Solyts, 2002; Shin, 2002; Jones, 2005; Goldfinger, 2008). In addition Gentleman and Johnson (1974) said that any direct implementation of this recursive method is inefficient and lead to repeated calculation of minors.

(iii) Condensation method

Dodgson (1866) proposed a new method for determining the determinant called the condensation method, as follows:

Let an $n \times n$ matrix $A = \{a_{ij}\}$ with pivot $a_{11} \neq 0$ by forming an $(n-1) \times (n-1)$ matrix $B = \{b_{ij}\}$ such that

$$b_{ij} = a_{1,1}a_{i+1,j+1} - a_{1,j+1}a_{i+1,1}.$$

Then

$$det(A) = \frac{det(B)}{(a_{11})^{n-1}}.$$

The weak point of this method is that we need to reduce the size of a matrix to 2×2 or 3×3 in order to obtain the determinant. This method has a fatal defect where the determinant of any interior matrix cannot be zero. Employ some remedies like row/column exchanges can be effective in discarding the defect, but they may not always work (Abeles, 2008). That was the disadvantage of non division free method. Vieira (2010) proposed a reduction method for finding the determinant which was quite similar to Dodgson's work.

(iv) Teimoori, Bayat, Amiri and Sarijloo (2005) method

The general formula of this method for finding the determinant is as follows:

$$|A| = \frac{1}{a_{21}a_{31}\cdots a_{(n-1)1}} \begin{vmatrix} \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} & \cdots & \begin{vmatrix} a_{11} & a_{1n} \\ a_{21} & a_{2n} \end{vmatrix} \\ \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} & \cdots & \begin{vmatrix} a_{21} & a_{2n} \\ a_{31} & a_{3n} \end{vmatrix} \\ \vdots & \ddots & \vdots \\ \begin{vmatrix} a_{(n-1)1} & a_{(n-1)2} \\ a_{n1} & a_{n2} \end{vmatrix} & \cdots & \begin{vmatrix} a_{(n-1)1} & a_{(n-1)n} \\ a_{n1} & a_{nn} \end{vmatrix} \end{vmatrix} \tag{1.4}$$

where the elements of $(n-1)\times(n-1)$ matrix in (1.4) is a determinant of 2×2 matrices. Then in order to find the determinant of that $(n-1)\times(n-1)$ matrix, the formula will be used again until the matrix is reduced to 2×2 or 3×3 matrix. Thus the process of calculating the determinant is highly dependence on the successfully determinant of the reduced matrix. Furthermore this method is not a division free and it does not work when the denominator is zero.

(v) Rezaee and Rezaifar (2007) method

The general formula of this method for finding the determinant is given by:

$$|A| = \frac{1}{|A_{11,nn}|} \begin{vmatrix} |A_{11}| & |A_{1n}| \\ |A_{n1}| & |A_{nn}| \end{vmatrix} \tag{1.5}$$

where A_{ij} is obtained by deleting the ith row and jth column. There are four $(n-1)\times(n-1)$ matrices and one $(n-2)\times(n-2)$ matrix where the entries itself is a determinant. In order to determine the determinant of these four $(n-1)\times(n-1)$ matrices i.e. $|A_{11}|, |A_{1n}|$, $|A_{n1}|$ and $|A_{nn}|$, the general formula (1.5) will be used again. For example take A_{11} as $(n-1)\times(n-1)$ matrix B. Then the determinant

of A_{11} is as follows:

$$|A_{11}| = \frac{1}{|B_{11,(n-1)(n-1)}|} \begin{vmatrix} |B_{11}| & |B_{1n-1}| \\ |B_{(n-1)1}| & |B_{(n-1)(n-1)}| \end{vmatrix} \tag{1.6}$$

Thus the process has high dependence on the determinant for $(k \times k)$ matrices where $2 < k < n$ and problems will arise when the denominator is zero.

(vi) Gauss elimination method

Gauss elimination method is a standard procedure to calculate the determinant. The given matrix is converted into an upper triangular matrix using elementary row operations. The value of the determinant is the product of its main diagonal elements of the upper triangular matrix. Gauss elimination is numerically unstable when a pivot element $(a_{kk})^{k-1}$ is zero or close to zero. In cases like these, pivoting is required. Pivoting works by interchanging the rows of $A(k)$ to obtain better pivot elements (Rezaee & Rezaifar, 2007). This division problem was also emphasized by Mahajan and Vinay (1997), Soltys (2002), and Shin (2002). Furthermore this method has problems in handling symbolic elements of square matrices (Sasaki & Murao, 1982; Shin, 2002).

In most cases, it is not easy to calculate the determinant of matrices of order $n \geq 5$ by hand-computing. Computing the determinant is also a time consuming process for larger matrices (Rice & Torrence, 2006). Regarding that, the development of sequential algorithms for finding determinant and comparing them over execution time has been carried out extensively.

A number of studies had modified Gauss elimination algorithm to determine the determinant of matrices with multivariate polynomial and fractional entries by introducing fraction free Gauss elimination method (Sasaki & Murao, 1982; Umeda & Sasaki, 2006). However in fraction free Gauss elimination method, fraction term still exist as follows

(Li, 2009a; Lee & Saunder, 1995):

$$a_{i,j}^{(k)} = \begin{bmatrix} a_{1,1}^{0} & \cdots & a_{1,k}^{0} & a_{1,j}^{0} \\ \vdots & & \vdots & \vdots \\ a_{k,1}^{0} & \cdots & a_{k,k}^{0} & a_{k,j}^{0} \\ a_{i1}^{0} & \cdots & a_{i,k}^{0} & a_{i,j}^{0} \end{bmatrix}, i > k, j > k$$

$$a_{0,0}^{(-1)} = 1, a_{i,j}^{(0)} = a_{i,j}$$

$$a_{i,j}^{(k)} = \frac{a_{k,k}^{(k-1)} a_{i,j}^{(k-1)} - a_{i,k}^{(k-1)} a_{k,j}^{(k-1)}}{a_{k-1,k-1}^{(k-2)}}$$

Although the fraction free Gauss elimination method is more efficient compared to the Gauss elimination method, Umeda and Sasaki (2006) found that the former method cannot conveniently be applied for finding the determinant of matrix with rational function elements where minor expansion performed better in terms of computation time compared to the fraction free Gauss elimination method. In other words, the multiplication and division of polynomial cannot be done in constant time where Gaussian elimination or its variants are not superior to cofactor expansion.

Rezaee and Rezaifar (2007) also compared their method with the cofactor expansion method over time computation and discovered their method was faster. They pointed out that their algorithm has weakness caused by 2×2 matrices repetition. For example, case $n = 4$, 2×2 matrices were repeated five times which had affected computation time.

Besides these, little effort has been made in finding the determinant using the permutation method. Thongchiew (2007) developed the permutation algorithm based on a partial reversion method and applied it to determine the determinant. However, he did not attempt to compare his method to other existing permutation generation methods.

Shin (2002) made comparison Gauss elimination method with Cofactor expansion method

and permutation approach for determining the symbolic determinant. He concluded that the permutation approach was a better method compared to the Gauss elimination method since the latter method encountered fraction problem. In addition he also discussed his new permutation generation method but unfortunately the permutation pattern was similar to the ones developed by Thongchiew (2007) and Zaks (1984). Shin (2002) also highlighted determinant method which did not have division had advantage over division method. In addition, the use of division method in finding determinant seems like a liability for hand computing (Bressoud & Propp, 1999). Thus we categorize the direct and non direct methods for finding the determinant to the division free method (DFM) and non division free method (nDFM). DFM are cross multiplication and cofactor expansion, whereas nDFM are Gauss elimination, Cholesky decomposition method, and Condensation Method.

1.1.2 Problem Statement

The division free methods have advantages over the non division free methods due to the following reasons:

(i) floating point errors can be avoided in division free methods (Mahajan & Vinay, 1997).

(ii) division free methods cater the problems when the entries of matrices are polynomial, rational or other complicated expressions such as multivariate polynomial (Rote, 2001).

The advantages in (i) is related to the exactness problem. The exactness problem arises in geometric algorithm where the determinant is used. Geometric prediction determines the control flow and hence has to be evaluated exactly. This means that determinant computation routine has to produce exact results (Goldberg, 1991). In addition rounding errors are inevitable in numerical computation (Li, 2009b).

However in division free category, cofactor expansion was not efficient in time computation where the order of complexity is $O(n((n-1)!)^3)$(Shin, 2002; Goldfinger, 2008). Thus the cross multiplication is an alternative method. However it works only for the order of matrix $n \leq 3$ and yet to be generalized. Bankier (1961) and Pavlovic (1961) attempted to extend cross multiplication method via permutation approach but they failed to generate a specific rule for determining $\frac{(n-1)!}{2}$ of the nth order diagram which tally to the third order diagram with a single permutation column $[1, 2, 3]$ as follows:

$$\left|\begin{array}{ccc|cc} a_{11} & a_{12} & a_{13} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{23} & a_{21} & a_{22} \\ a_{31} & a_{32} & a_{33} & a_{31} & a_{32} \end{array}\right|$$

Thus, there is a need to construct new algorithms to generalize the cross multiplication method to any size of matrix using permutation approach. Since permutation generation is time consuming (Sedgewick, 2002) which fall under Non-Polynomial (NP) time, the implementation of sequential algorithms to parallel computation is the option for reducing the computation time (Akl & Bruda, 2001).

To our present knowledge, no research has been conducted in developing sequential and parallel division free methods for finding the determinant of matrices of any order using the generalised cross multiplication method (Sarrus Rule).

1.2 Research Question

This study aims to develop new sequential and parallel division free methods for finding the determinant of matrices using generalised cross multiplication method. Therefore, the following are the research questions to be solved:

(i) what is an appropriate sequential method for generating permutation which correspond to cross multiplication method.

(ii) how to develop new permutation method which can be used for finding determinant to generalise the cross multiplication method.

(iii) what is a suitable method for parallelise all the new sequential algorithm for generating permutation and finding the determinant.

(iv) how to measure the performance of all new parallel algorithms.

1.3 Objective of the Study

The main objective of this research is to develop new sequential and parallel division free methods for finding the determinant of matrices using the generalised cross multiplication method. This can be accomplished by

(i) constructing new sequential algorithms for generating permutation.

(ii) developing new division free sequential methods for finding the determinant using generalised cross multiplication.

(iii) parallelizing the sequential methods for permutation and finding the determinant.

(iv) analyzing the performance of parallel algorithms in terms of speedup and efficiency.

1.4 Methodology

This study was carried out in six phases as follows:

Phase One: Information Gathering

In this phase, our review will focus on the methods for finding the determinant The strengths and weaknesses of each method will be studied and highlighted.

Phase Two: Method and Algorithm Development

(i) Method development

We used different approach of permutation to find the determinant for matrices of order $n \leq 6$ to see the patterns of construction. Meanwhile we did

investigation and backtracking on cross multiplication method, and extracted its permutation pattern from its main diagonal and secondary diagonal. From these patterns, we produced some important theorems as a basis for finding determinant of any size $n \times n$ matrices.

(ii) Sequential, and Parallel algorithm

After the cross multiplication method is extended and modified based on permutation construction, a new sequential algorithm was developed. For parallel algorithm, according to Kokosi$\acute{n}$ski (1990) who has listed two different approaches for designing parallel generation of permutation as follows:

(i) apply the sequential algorithm to the model of parallel computation.

(ii) design the parallel algorithms for a model of parallel computation with any number processors.

Thus, we developed both approaches to design the parallel algorithm. Regarding to development of parallel algorithm in two approaches, two different method were constructed. Fundamentally, every parallel algorithms involves a collection of tasks that can be execute concurrently.

Phase Three: Implementation Mechanism

Both sequential, and parallel algorithms will be implemented into C codes. For parallel programming environment, we considered message passing model.

Phase Four: Analysis of Sequential algorithm

For this phase, we compared the computation time and time complexity of the developed method with the existing method in sequential. We divided the analysis into two stages as follows:

Stage 1: Comparing among the permutation method and its application for the determining the determinant.

Thongchiew (2007) method was selected for the comparison since only Thongchiew

(2007) developed partial reversion permutation generation method and applied it for determining the determinant. We also developed Langdon (1967) method program and apply his algorithm to find the determinant.

Stage 2: Comparing to the others direct determinant method

In this stage, Cofactor Expansion method was selected to compare with our method because it is a division free method based on classical definition of determinant.

Phase Five: Analysis of Parallel algorithm

For this phase, to measure the performance of the parallel algorithms, the *speedup* and *efficiency* were used.

Phase Six: Documentation

The writing was carried out along the study duration.

1.5 Significance of the Study

The development of the new sequential division free methods for finding the determinant contributes to the body of knowledge in linear algebra. Moreover division free computation for determinant plays significant role in estimating the parallel complexity of basic linear algebra problems, such as matrix inversion (Kaltofen, 1992). Besides the time consuming computation which occur in this sequential algorithms, developing new parallel methods can be seen as an alternative solution for faster computation. In addition, even finding the determinant is one of the oldest topic in linear algebra, there is still a need for developing new methods that are well-suited to modern high performance computers. Thus it would become a valuable contribution to the computational mathematics area.

The construction of new permutation generations also contribute to the combinatorial design problems such as linear assignment problems (Rolfe, 2008), Latin square enumerations (Fike, 1975) and Travelling Salesman Problems (Aziz et al., 2009). We hope that

our parallel methods for permutation generation especially in starter sets generation can be further used to solve some of these problems.

1.6 Scope of the Study

This study focused on generating permutation method and it's application for finding the determinant of real square matrices, not on symbolic matrices. This study include theoretical development for generalization, and implementation the algorithm in C Language.

1.7 Assumption and Limitation of the Study

The numerical tests in this research were performed on the Sunfire V1280 parallel computer which installed at the Institute for Mathematical Research (INSPEM), Universiti Putra Malaysia (UPM) as this facilities is not available at Universiti Utara Malaysia (UUM). The new algorithms are tested on real value matrix only and run on the matrix of order up to 14.

1.8 Thesis Outline

Chapter One of this thesis describes the background, problem statement, objective and significance of the study.

Next, Chapter Two provides some fundamental concepts for permutation, determinant and matrices, and parallel computing. This chapter also reviews of relevant literature related to the present study are done which include permutation generation method, sequential and parallel algorithm of permutation generation method, and finding determinant via permutation approach.

Chapter Three focuses on generation of starter sets and permutation by employing certain rule and also presents some theoretical works. Then the numerical results of the

algorithm are given in terms of computation times and order of complexity.

The construction of a division free method for finding the determinant using permutations will be presented in Chapter Four which is aim for the Sarrus Rule generalisation. In this chapter, the sequential algorithms development for finding determinant by application of permutation generation are discussed.

Chapter Five is the extension from work in Chapter Four and Chapter Five where the parallel method are extended from sequential method as Across The Time (ATT) method. Meanwhile for Across The Method (ATM) method, new parallel methods of permutation generation and its application for finding the determinant are also developed. Their performances are measured in terms of speedup and efficiency.

Chapter Six summarizes the study and some suggestions on future work are also recommended.

CHAPTER TWO

PERMUTATION, DETERMINANT, PARALLEL COMPUTING AND ITS RELATED STUDIES

2.1 Introduction

In this chapter, we outline the basic concepts of permutation, determinant, and parallel computing that will be needed in Section 2.2, 2.3 and 2.4 respectively. This chapter also discussed some related studies on permutation and determinant techniques in Section 2.5.

2.2 Permutation

Definition 2.2.1. *A linear ordering of the elements of the set* $[n] = [1, 2, 3, \cdots, n-1, n]$ *is called a permutation.*

Definition 2.2.2. *Let* $a = a_1, a_2, \cdots, a_{n-1}, a_n$ *be a permutation.* (a_i, a_j) *is an inversion of* a *if* $i < j$ *but* $a_i > a_j$*.*

Example 2.2.3. *Permutation* $[3, 1, 2, 5, 4]$ *has three inversions, namely* $(3, 1), (3, 2)$*, and* $(5, 4)$

Definition 2.2.4. *A permutation is called odd (even) if it has an odd (even) number of inversion.*

Definition 2.2.5. *The identity permutation* $[1, 2, 3, \cdots, n-1, n]$*, denoted by* ϵ *is the permutation that leaves all integers fixed.*

Definition 2.2.6. *A transposition is a permutation that interchanges two integers* k *and* l*,* $k \neq l$*, but leaves all other integers fixed.*

Definition 2.2.7. *The number of circular arranging of* n *distinct object is* $(n-1)!$*.*

Theorem 2.2.8. *Of the* $n!$ *permutations of the elements* $a_1, a_2, \cdots, a_{n-1}, a_n$ *there are as many that have an even number of inversions as there are that have an odd number; that is there are* $\frac{n!}{2}$ *in each of the two classes (Muir, 1933).*

Lemma 2.2.9. *The maximum number of inversions is* $\frac{n(n-1)}{2}$ *(Hsiung & Mao, 1998).*

2.3 Determinant of Matrix

Let row i (the ith row) and column j (the jth column) of the determinant $A = |a_{ij}|_n$ be denoted by the symbols R_i and C_j respectively:

$$R_i = [a_{i1}a_{i2}...a_{in}]$$
$$C_j = [a_{1j}a_{2j}...a_{nj}]^T$$

where T denoted the transpose. Now we may write $A = |C_1C_2C_3...C_n|$.
The column vector notation is more economical in space and will be used exclusively in later chapters.

Definition 2.3.1. *A pattern in an* $n \times n$ *matrix of* A *is a way to choose* n *entries of the matrix so that there is one chosen entry in each row and in each column of A. (Bretscher, 2009).*

Definition 2.3.2. *Main diagonal of square matrix which generated based on any column array is a diagonal from the left-hand top corner to the right-hand bottom corner of square matrix (Muir, 1933)*

Definition 2.3.3. *Secondary diagonal of square matrix which generated based on any column array is the diagonal from the right-hand top corner to the left-hand bottom corner of square matrix (Muir, 1933).*

Definition 2.3.4. *Any parallel lines to these (main or secondary diagonal) is a minor diagonal (Hanus, 1886).*

Definition 2.3.5. *The products of all elements in main diagonal, and in secondary diagonals are called terms (Muir, 1933).*

A common way to introduce the determinant in a first course of linear algebra as the following (Reffgen, 2003):

Definition 2.3.6. *The determinant* $det : Mn(R) \to R$ *is the unique mapping which satisfies the following three conditions:*

(i) The determinant of the unit matrix equals one.

(ii) The determinant depends linearly on each column.

(iii) The determinant changes sign if two columns in the matrix change place.

In other words, $det(A)$ *is a multilinear, alternating function in the columns of* $A \in Mn(R)$.

If the matrix $A \in Mn(R)$ is given by

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

then the three conditions in Definition 2.3.6 and straightforward calculations give

$$det(A) = \sum_{(\sigma(1),\sigma(2),...,\sigma(n))\in S_n} sign(\sigma) \cdot a_{1\sigma(1)} \cdot a_{2\sigma(2)} ... \cdot a_{n\sigma(n)} \tag{2.1}$$

where the sum is taken over all permutations of the numbers $1, 2, ..., n$, showing the uniqueness of the determinant, and the existence follows if one, for example, shows that formula (2.1) satisfies the three conditions of the definition of the determinant.
An alternative for equation (2.1) is

$$det(A) = \sum_{(\phi(1),\phi(2),...,\phi(n))\in S_n} sign(\phi) \cdot a_{\phi(1)1} \cdot a_{\phi(2)2} ... \cdot a_{\phi(n)n} \tag{2.2}$$

The summation is taken over the set of all permutations ϕ of n elements. The sign of a permutation is defined in terms of the number of inversions.

$$sign(\phi) = (-1)^{\text{number of inversion } \phi} \tag{2.3}$$

Theorem 2.3.7. *The number of the terms in the determinant of order n is $n!$ (Scott, 2009).*

Let us recall the following well-known elementary properties of the determinant:

(i) The determinant of a matrix vanishes if and only if the matrix is singular.

(ii) The determinant of a matrix A remains unchanged if we add a multiple of one column to another column.

(iii) The determinant of a matrix A equals the determinant of its transpose matrix.

(iv) The determinant is multiplicative, i.e. $det(AB) = det(A)det(B)$.

2.4 Parallel Computing

The world's computing requirements are constantly growing, resulting in major challenges in the form of problems requiring heavy computation. Thus a significant reduction in the time is required to solve the problem (Akl & Bruda, 2001). Therefore a method to increase computational speed for a given application has to allocate the tasks among multiple processors. A parallel computer is a specially designed computer system containing multiple processors or several independent computers that are interconnected in specific ways. Some definitions of the term which are related to this study are given as follows.

Definition 2.4.1. *An algorithm is a finite set of instruction for solving a problem (Gupta et al., 2008).*

Definition 2.4.2. *A sequential algorithm is an algorithm designed for a single-processor machine (Horowitz et al., 2008).*

Definition 2.4.3. *A parallel algorithm is an algorithm designed for a multi-processor machine (Horowitz et al., 2008).*

Definition 2.4.4. *A program is the expression of an algorithm in a programming language (Horowitz et al., 2008).*

Definition 2.4.5. *The time complexity of an algorithm is the amount of computer time its needs to run to completion (Horowitz et al., 2008).*

Definition 2.4.6. *A recursive program is just a program that call themselves in order to obtain a solution to a problem (Standish, 1997).*

Definition 2.4.7. *A task is a program or a part of a program in execution (Leopald, 2001).*

Definition 2.4.8. *Process is used synonymously with tasks (Leopald, 2001).*

Definition 2.4.9. *Parallelism is a process of performing tasks concurrently (Lewis & El-Rewini, 1992).*

Definition 2.4.10. *Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (Almasi & Gottlieb, 1989).*

Definition 2.4.11. *Distributed computing is any computing that involves multiple computers remote from each other that each has a role in a computation problem or information processing. In distributed computing, each processor has its own private memory (distributed memory). Information is exchanged by passing messages between the processors.*

The difference between parallel computing and distributed computing is the former splits an application up into tasks that are executed at the same time, whereas the latter splits an application up into tasks that are executed at different locations, using different resources (Leopald, 2001)

Definition 2.4.12. *Parallel programming is the technique of creating a single computer program in such a way that it can be executed by more than one processor simultaneously (Brawer, 1989).*

Definition 2.4.13. *Parallel processing is the solution of a single problem by dividing it into a number of subs-problem, each of which may be solved by a separate processor (Chalmers & Tidmus, 1996).*

Definition 2.4.14. *Parallel system is the combination of an algorithm and parallel architecture on which it is implemented (Grama et al., 2003).*

Definition 2.4.15. *Distributed system is a collection of autonomous computers that are interconnected with each other and cooperate, thereby sharing resources (Leopald, 2001).*

Definition 2.4.16. *Embarrassingly parallel computation is one that can be immediately divided into completely independent parts that can be executed simultaneously (Wilkinson & Allen, 2005).*

2.4.1 Parallel Computing Platform

Parallel computer requires a suitable computing platform namely Shared Memory Multiprocessor System, Distributed Shared Memory and Message-Passing Multicomputer.

(i) Shared Memory Multiprocessor System

This architecture accomplishes interprocessor coordination through a global memory shared by all processors. Two key elements of a conventional computer system are the processor and the memory.

(ii) Distributed Shared Memory

This system where each processor has access to the whole memory using a single memory addresses space. A processor can access data from location, which is not in its local memory by using message passing from the processor to the location or from the location to the processor.

(iii) Message-Passing Multicomputer

A multiprocessor system can be created by connecting complete computers, which each computer consists of a processor and local memory, through an interconnection network. Interactions between processors must be accomplished using messages, the Message Passing Interface (MPI) is used where MPI is a message passing library standard developed by the Message Passing Interface Forum. Message Passing Interface is a process uses the library calls to exchange messages with another

process. This message passing allows processes running on multiple processors to cooperate in problems solving (Hussain & Ahmed, 2005). MPI and Parallel Virtual Machine (PVM) are considered as the most popular programming techniques for parallel computers including massively parallel computers, and PC/workstation clusters (Fadlallah et al., 2000).

According to Gropp et al. (1997), the fundamental of MPI are as follows:

(i) MPI is a library, not language. It specifies the names, calling sequences, and results of subroutines to be called from Fortran program the functions to be called from C programs, and the classes and the method that make up the MPI C++ library.

(ii) MPI is a specification, not a particular implementation.

(iii) MPI addresses the message passing model.

MPI contains approximately 125 functions. However MPI is reasonably easy to learn as a complete message-passing program can be written with only six basic functions. The following outline can be used to structure most MPI programs (Hussain & Ahmed, 2005; Lin & Snyder, 2009):

(i) All MPI programs must include a header file (in C, mpi.h; in FORTRAN, mpif.h).

(ii) All MPI programs must call MPI_Init() as the first MPI call, to initialize data structures by each process before any other MPI routines are invoked.

(iii) Most MPI programs call MPI_COMM_Size() to determine the size of the current virtual machine, that is, how many processes are running.

(iv) Most MPI programs call MPI_COMM_Rank() to determine their rank, which is a number between 0 and $p-1$, p is the number of processes in the communicator.

(v) Conditional process and general message passing can take place, for example the calls MPI_Send() and MPI_Recv(). MPI_Send() routine sends data to other process, and MPI_Recv() routine accepts the data. The messages are required to be specified in this communication.

(vi) All MPI programs must call MPI_Finalize as the last call to an MPI library routine to clean up MPI data structures.

The structure of general MPI program as follows (refer Figure 2.1):

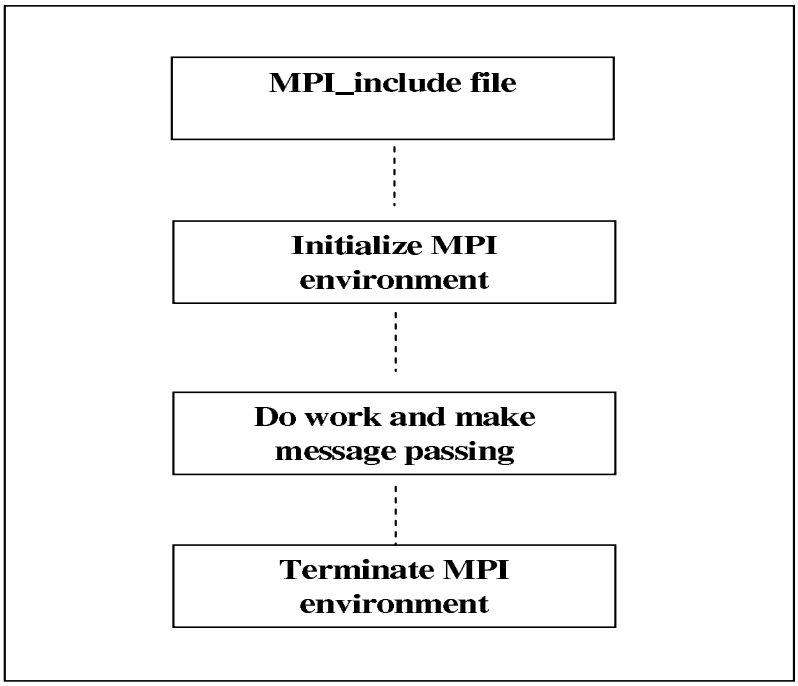

Figure 2.1: General MPI Program Structure

The paradigm of the parallel programming used with MPI is the master-slave parallel paradigm where the master is responsible for dividing task among other processors called the slaves. All the slaves execute the task given concurrently. Both the master code and the slave code are in the main program function. Thus the following MPI code segment illustrates how this could be achieved where *master()* and *slave()* are procedures to be executed by the master process and slave process respectively.

```
main(int argc, char *argv[])
{
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  /* find the process rank */
if(myrank == 0)
master();
else
slave();
MPI_Finalize();
}
```

2.4.2 Parallel Computer Sunfire 1280 Architecture

The parallel computer that has been used is the Sunfire V1280 High Performance Computer. The Sunfire V1280 is a distributed shared memory multiprocessor system which can accommodate up to twelve UltraSPARC III Cu processors populated on three CPU/memory boards. Each board includes four processors, all cache, and main memory. The detail of hardware configuration on Sunfire V1280 is shown in Table 2.1:

Table 2.1: Hardware Configuration of Sunfire 1280

Number of processors	4, 8, or 12 1.2GHz Ultra SPARC III Cu Processors
Processor	
Architecture	Superscalar SPARC V9, UltraSPARC III Cu architecture
Number of processors	4, 8, or 12 1.2GHz Ultra SPARC III Cu Processors
Processor	
Architecture	Superscalar SPARC V9, UltraSPARC III Cu architecture Cache per processor Level 1: Parity-protected 32 KB instruction and 64 KB data on chip (single-bit errors are corrected) Level 2: 8 MB external cache
Capacity	Up to 96 GB memory High throughput and low response times may be achieved by keeping data in memory. Up to 6 PCI slots 2x36 GB Ultra3-SCSI internal disks
1 integrated Ultra3-SCSI port	
System	
Main Memory	8 GB to 96 GB
I/O	6 short PCI slots (64 bit; one at 66 MHz, 5 at 33 MHz)
System Controller	Integrated Ultra3 SCSI supports up to 15 SCSI devices
Hard Disk	Two 73 GB disks Ultra3 SCSI internal disks
Network connectivity	Two integrated Gigabit Ethernet ports (66 MHz)
Removable media	DVD-ROM internal drive
Operating System	Solaris 8, Solaris 9 and Solaris 10
Languages	C, C++, Pascal, Fortran, Java
Networking	ONC/NFS, TCP/IP, SunLink OSI, X.25, DCE, NetWare

2.4.3 Performance of Parallel Algorithms

The performance of a sequential algorithm can be measured in terms of its computation time and memory space (Horowitz et al., 2008; Grama et al., 2003). Time complexity for sequential algorithm can be extended to parallel algorithms. However, parallel implementation may require expense communication between the parallel parts, which contributed more to time computation (Wilkinson & Allen, 2005).

2.4.3.1 Speedup

Speedup, $S(p)$ is defined as the ratio of the time taken to solve a problem in single processor to the time required to solve the same problem on multiprocessors. On the other hand, speedup factor is a measure of relative performance (Wilkinson & Allen, 2005).

$$Speedup = \frac{T_s}{T_p} \tag{2.4}$$

where T_s is the sequential time and T_p is the parallel time running on p processors.
Two possibilities exist for determining the time taken of a single processor (T_s) (Chalmers & Tidmus, 1996):

(i) the time obtained when executing an optimized sequential algorithm on a single processor, or

(ii) the time obtained when executing the parallel algorithm on one processor.

2.4.3.2 Efficiency

Efficiency is defined as the ratio of speedup to the number of processors (Grama et al., 2003) where efficiency means the utilization of processors being used on the computation.

$$Efficiency = \frac{Speedup}{P} \tag{2.5}$$

In an ideal parallel system, speedup is equal to p and efficiency is equal to one. Since there are various sources of performance loss, typically efficiency is less than 1 and diminishes as the number of processors is increased (Lin & Snyder, 2009).

2.4.3.3 Scalability

An important aspect of performance analysis is the study of how algorithm performance varies with parameters such as problem size, processor count, and message startup cost.

Evaluating the scalability of a parallel algorithm, mean how effective it can use when the number of processors increased. On the other hand, scalability is used to indicate that a parallel algorithm can accommodate increased data items with low and bounded increase in computational steps. It also can be used to indicate hardware design that allows the system to be increased in size and to obtain increased performance (Wilkinson & Allen, 2005). The parallel time complexity can generally be represented as

$$T(n,p) = O(\frac{T(n)}{p} + T_{comm}(n,p)) \tag{2.6}$$

where n is the problem size, p number of processors available, $T(n)$ is the time complexity of the best sequential algorithm, and $T_{comm}(n,p)$ is the overall communication overhead of a parallel time complexity (Li, 2010).

2.4.3.4 Cost

The cost of a parallel algorithm is the product of the number of processors used and its running time (Stojmenovic, 2006). In other words, cost equals the number of steps executed collectively by all processors in solving a problem in the worst case. A parallel algorithm is said to be cost optimal if its cost matches a lower bound on the number of operations required to solve the problem sequentially.

2.4.3.5 Big O notation

Big O notation is used in computer science, and mathematics to describe performance or complexity of an algorithm. Specifically big O notation is a convenient way to express the worst-case scenario for a given algorithm (Levitin, 2007). It is defined such as follows

Definition 2.4.17. *A function $g(N)$ is said to be $O(f(N))$ if there exist constants c_0 and N_0 such that $g(N)$ is less than $c_0 f(N)$ for all $N > N_0$.*

2.4.4 Performance Influence Factors

Execution time is influenced by many factors. Among these factors are

(i) hardware technology: factors on hardware like circuit interconnections and cooling, degree of integration play a role in determining speed.

(ii) architecture: selecting a system or optimizing a design can influence the performance of a parallel processing. The arithmetic unit, control unit, and memory impact the performance of each processor in the system. data movement and synchronization among processors contribute to the performance of the system.

(iii) operating system: the operating system shares resources among multiple users of the system and sharing resources among multiple processes belonging to one user's parallel program. So interprocess data movement, process control, synchronization and input/output is managed by the operating system.

(iv) language: the efficiency of the programming implementation can influence the system performance. The language that influences the system is determined by the compiler and the run-time system.

(v) algorithm: the criteria of the algorithm that directly influence the performance includes the depth of the dependence graph, its size or number of operations, the maximum, minimum and average parallelism.

2.5 Related Studies on Permutation and Determinant

This section discusses some permutation generation techniques dan division free method for determinant. Then sequential and parallel algorithms for permutation generation and determinant are reviewed respectively.

2.5.1 Permutation Generation Techniques

Permutation is a set of element arranged in a definite order. The generation of all $n!$ of n elements is a fundamental problem in combinatorics and has a long history. Numerous works on permutation had been done since early 1650s and permutation still being studied due to its importance. Permutation generation method is an essential method for solving combinatorial optimization problems such as Travelling Salesman Problems (TSP), Flow-shop Scheduling Problems (FSP), and Quadratic Assignment Problem (QAP) (Peng et al., 1999). In addition, permutation can be used as a tool for local memory sequence generation for data parallel programs (Huang et al., 2001).

Various methods for generating permutation had been developed such as lexicographic order (Ord-Smith, 1970), transposition (Wells, 1961; Heap, 1963; Lispki & Warsaw, 1979), cycling (Iyer, 1995), shift cursor and level (Gao & Wang, 2003), partial reversion (Zacks, 1984; Shin, 2002; Thongchiew, 2007), Viktorov (2007), Barisenko et al.(2008) and Ibrahim et al. (2010).

According to Sedgewick (1977), generating permutation under cycling restriction was initiated by Langdon (1967). This technique is simpler when compared to other technique under non exchanged based. Furthermore, Sedgewick (1977) claimed that cycling operation is powerful due to its simplicity. The idea of Langdon (1967) technique is cycling interchange n elements until two elements inductively. Every performing of any i successive cycling interchange will produce i cycling permutation where $1 < i \leq n$. The next cycle of interchange will give the original state of permutation. The following figure illustrates the list of permutation based on Langdon technique for four elements.

2	1	4	3
1	4	3	2
4	3	2	1
3	2	1	4
3	1	4	2
1	4	2	3
4	2	3	1
2	3	1	4
3	2	4	1
2	4	1	3
4	1	3	2
1	3	2	4

1	2	4	3
2	4	3	1
4	3	1	2
3	1	2	4
1	3	4	2
3	4	2	1
4	2	1	3
2	1	3	4
2	3	4	1
3	4	1	2
4	1	2	3
1	2	3	4

Figure 2.2: List of 4! Permutations Based on Langdon Technique

Meanwhile, Iyer (1995) introduced a technique which employed the cycling processes and copying column of matrices. However Iyer's technique was only valid for $n \leq 4$ because the repetition of permutations occurs for the case $n \geq 5$. The example for $n = 5$ is shown in Figure 2.3.

Example 2.5.1. *Case* $n = 5$

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

Figure 2.3: All Pivots for $n = 5$

Through employing cycling rotations and copying column matrices, both **bold** permutation pivot, i.e. $[1, 3, 2, 4, 5]$ and $[1, 5, 4, 2, 3]$ will produce similar permutation, as displays in Figure 2.4:

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

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

Figure 2.4: Repetition Permutations over Pivot $[1, 3, 2, 4, 5]$ *and* $[1, 5, 4, 2, 3]$

This problem occurred because Iyer (1995) employed the cycling $n - 1$ elements until 3 elements inductively, and copying column matrices. Other than Langdon (1967) and Iyer (1995), Ibrahim et al.(2010) introduced a new permutation technique based on distinct starter sets by employing cycling and reversing operations. Their crucial tasks were generating starter sets and eliminating the equivalence starter sets. Let consider $n = 4$ and $S = [1, 2, 3, 4]$. Without loss of generality, the element 1 is fixed in order to find the starter sets. Thus there are $(n - 1)! = 3!$ starter sets including their equivalence starter sets. For simplicity, we listed down the starter sets (**Column A**) and it equivalence starter sets (**Column B**):

1	2	3	4
1	2	4	3
1	3	4	2

Column A

1	4	3	2
1	3	4	2
1	2	4	3

Column B

Figure 2.5: Starter Sets and its Equivalence

Thus by employing the cycling operation and reversing the order of permutation, $2n$ distinct permutations are produced. However the equivalence starter sets will generate the same permutations and need to be discarded. The advantages of this technique are simple and easy to use. However, eliminating the equivalence starter sets becomes tedious when $n > 4$.

In this study, we will attempt to derive and enhance new recursive strategies to generate $\frac{(n-1)!}{2}$ starter sets without generating the equivalence starter sets. This derivation can be done by investigating on Iyer (1995) technique and then employing Ibrahim et al. (2010) method to list all $n!$ distinct permutations. This is an initial step of developing a new permutation method to generalise Sarrus Rule via combinatorial approach.

2.5.2 Sequential Algorithm For Generating Permutation

Algorithms can be designed as a recursive procedure (top-down), or iterative procedure (bottom-up) (Sedgewick, 1977; Stojmenovic, 2006). Recursive procedure is one that invokes itself repeatedly which the definition of procedure being defined is applied within its own definition (Hanly & Koffman, 2004; Reek, 1998). The example of recursive procedure for factorial number $F(n)$ is as follows:

Example 2.5.2. *Procedure F(n)*
if n = 0 return 1
else return F(n-1)n

Meanwhile, iterative procedure uses repetitive construction such as loops and sometimes additional data structures like stacks to solve the given problems (Reek, 1998). The example of iterative procedure for factorial number, $F(n)$ is as follows:

Example 2.5.3. *Procedure F(n)*
prod = 1
for ($k = 1$*;* $k \leq n$*;* $++k$*)*
$prod = prod * k$
return prod

Fike (1975) developed and compared his recursive and iterative programs without printing statements to generate all permutations based on the exchange of two consecutive

elements in terms of computation time. As a result, his recursive program is faster than the iterative program. Meanwhile in terms of time complexity, Sedgewick (1977) carried out the complexity analysis on Heap, Ives and Langdon algorithm (see Table 2.2).

Table 2.2: Time Complexity of Heap, Ives, and Langdon Algorithm

Algorithm	Time Complexity
Heap	$(19 + (\frac{1}{e}) + 10(e-2))n! + 6n + O(1)$
Ives	$9n! + 2(n-1)! + 18(n-2)! + O(n-4)!$
Langdon	$n!(2n + 10 + \frac{9}{n}) + (O(n-2)!)$

These three methods were selected as the best recursive, iterative, and cycling algorithm respectively. Overall, the order of complexity of these algorithms are $O(n!)$ for Heap and Ives algorithms, and $O(nn!)$ for Langdon algorithm. In spite of Sedgewick's survey in 1977, Knuth (2002) highlighted some demerits of the existing algorithms for permutation generations. Basically two demerits existed such as the re-visiting of the same permutations, and manipulation of the additional second array/table to generate all permutations. For example, the Lexicographic order and Langdon algorithm fall under the former demerit, whereas Ehrlich swaps and plain change algorithm fall under the latter demerit. These kinds of demerits added to additional spaces and time execution.

After 1977, some algorithms were developed in recursive ways such as Lipski and Warsaw (1979), Zaks (1984) and Iyer (1995), and iterative: Thongchiew (2007) and Viktorov (2007). The iterative algorithm seems to have advantage of giving easy control over generating the next permutation from the current one (Stojmenovic, 2006). However in many instances, the use of recursion enables us to specify a simple solution to a problem that would be very difficult to solve (Hanly & Koffman, 2004). Reek (1998) addressed that recursion is a powerful technique where a lot of problems were explained recursively only because they are clearer than non recursive explanations. Furthermore recursion is an example of divide-and-conquer problem solving strategy where the strategy proposed the splitting of the input into subproblems (Horowitz et al., 2008).

We will develop our algorithm in the recursive, and iterative procedures. Then these algorithms will be compared to Langdon algorithm (iterative), lexicographic order algorithm (recursive), and Thongchiew algorithm (iterative) over time computation. We will analyze order of complexity of the new developed algorithm.

2.5.3 Parallel Algorithm For Generating Permutation

Permutation generation is a time consuming operation for sequential algorithm. This disadvantage can be overcome by using parallel computers with several processors running simultaneously. Many literatures had given much attention in parallel algorithm for permutation generations in lexicographic order i.e. Tsay and Lee (1994), Akl et al. (1994) and Djamegni and Tchuente (1997). In spite of lexicographic order, there were also numerous works had been done for parallel algorithm development in minimal change order by Akl and Stojmenovic (1992), ranking and unranking by Kokosiński (1990) and Lin (1991), shuffling by Anderson (1990) and lower exceeding sequences by Alonso and Schott (1996).

Parallel algorithms for generating permutations of certain types were reviewed by Stojmenovic (2006) and Akl et al. (1994). They surveyed some developed parallel algorithms based on six properties (P) for shared memory computing, as follows:

P1 : The permutation are listed in lexicographic order, i.e. if $A = (a_1, a_2, ..., a_n)$ and $B = (b_1, b_2, ..., b_n)$ are permutations of $\{p_1, p_2, ..., p_n\}$, then A precedes B lexicographically if and only if, for some $j \geq 1, a_i = b_i$ when $i < j$, and $a_j < b_j$.

P2 : The algorithm is cost optimal.

P3 : Time required by the algorithm between two consecutive objects is a constant.

P4 : The parallel computation should be as simple as possible.

P5 : Each processor needs as little memory as possible.

P6 : The algorithm should produce all permutations of n elements for a given n.

Akl et al. (1994) and Djamegni and Tchuente (1997) had successfully developed a cost optimal systolic algorithm, and pipeline algorithm respectively in which their parallel algorithm complexity are the same as sequential algorithm for permutation generation complexity $O(n(n!))$ where n is a length of permutation. Meanwhile Tsay and Lee (1994) algorithm for lexicographic order is not optimal due to its two-dimensional triangular array in each processor where their complexity is $O(n^2(n!))$. In spite of cost optimal, Akl et al. (1994) algorithm is also adaptive as it can be run by any p number of processors by considering the following three cases:

(i) $p < n$: each processor do the job of $\frac{n}{p}$ processors in the original algorithm (with $\frac{n}{p}$ rounded appropriately if not an integer, then the last processors does slightly less work).

(ii) $p > n$, and $r = \frac{n}{p}$ is an integer: array is divided into r group of n processors, such that each group produced an interval of consecutive permutations of n elements.

(iii) $p > n$, and $r = \frac{n}{p}$ is not an integer : this case is handled by combining (i) and (ii).

In contrast, designing other pattern of permutation generation method in parallel, property 1 is not needed to be satisfied. For example Kokosiński (1990), Lin (1991) and Alonso and Schott (1996) algorithm are not based on lexicographic order. Furthermore, Stojmenovic (2006) and Akl et al. (1994) claimed that in designing parallel algorithm for listing cases of the restricted or generalized permutation such as permutation with repetitions, cyclic permutations, rosary permutations, alternate permutations, and linear extension still remain as open problems since 1994.

Meanwhile property 2 excluded the communication time among the processors which involved data passing/transferring especially for the parallel computers with message passing interface. Property 3 is limited to permutation generations based on exchange of two

elements, not for other restriction such as cyclic restriction involves more than two elements in a cyclic task.

Kokosiński (1990) listed two different approaches for designing parallel generation of permutation as follows:

(i) application of the sequential algorithm to models of parallel computation.

(ii) designing the parallel algorithms for models of parallel computations with any number of processors.

Kokosiński (1990) algorithm falls into the first approach where the sequential algorithm for permutation generation based on interactive cost decomposition of the symmetric group, is parallelized. Regarding to this, the ranking and unranking function techniques are used for the proper distribution generation tasks in the multiprocessor system. Two algorithms were developed namely the algorithm for ranking which requires linear time and the algorithm for unranking which has $O(n^2)$ time complexity. In his program, the N processors were assumed in the Single Instruction Multiple Data (SIMD) system and then the subset of permutation generations were allocated among processors: the first $N-1$ processors will generate $\frac{n!}{N}$ permutations and the Nth processor will generate only $(n! - \frac{(N-1)n!}{N})$ permutations while the last processor generated the remainder. No explanation was given about the value of N relates to n when N is not evenly divided $n!$. For example, $n = 12$ and $N = 13$. In this case, even $N < n!$ but it was not evenly divisible. Using the unranking function, the algorithm is not cost optimal i.e. $O(n^2(n!))$ but all $n!$ distinct permutations are listed.

Another cost optimal parallel permutation generation algorithm for linear arrays but not in lexicographic order pattern was given by Lin (1991). For parallel implementation, the number of processors is assumed to be equal to the number of elements in permutations.

Anderson (1990) had developed practical algorithms for generating random permutations

and it's framework of parallel algorithm was for small shared memory machines in order to get efficient implementation theoretically. The probability of permutation generated is $\frac{1}{n!}$ on parallel algorithm which means that there are no redundancies. This theory has been used by Alonso and Schott (1996) while Cong and Bader (2000) also worked on parallel algorithm for random permutation generation. Furthermore the process of swapping elements among processor was also under probability process. Alonso and Schott (1996) algorithm implementation procedure i.e. the jth processor selected randomly number in the interval $[1, i]$ and then all permutations were obtained with probability $\frac{1}{n!}$ which is equal to the probability of generating lower-exceeding sequences. In addition, their algorithm implemented a merging sort for all processors network. Meanwhile Cong and Bader (2006) highlighted that random permutation generations are useful in designing randomized algorithms which were low cost algorithms providing good cache performance for shared memory processors.

Overall, all previous works were implemented in different diagrams such as linear array processor and vector computer where every processor was responsible for producing one element of every permutation generated and the data were shared. Furthermore none of them had been applying their algorithms for finding the determinant. In contrast our algorithm will be implemented in distributed shared memory computer with Message Passing Interface (MPI). A distributed shared memory system containing p processors $p_0, p_1, p_2, \ldots, p_{p-1}$ connected by an interconnection network. Each processors has its own local memory and there is no global shared memory (Li, 2009). Every processor is responsible for generating n elements of permutation. The production of one element from each processor and passing to other processors will be time consuming because with MPI environment, data is not shared but copied. Message Passing is a powerful and a very general method of expressing parallelism (Pacheco, 1997). Quinn (2004) highlighted debugging message-passing programs were simpler than debugging shared-variable programs. Furthermore we will use Message Passing Interface for distributed shared computing to

develop parallel algorithm for permutation and applied that algorithm for finding the determinant of a square matrix.

2.5.4 Finding the Determinant Using n! Permutations

Application of permutations can be found in the combinatorial design problem such as linear assignment problem (Rolfe, 2008), Latin square enumeration (Fike, 1976) and Traveling Salesman Problem (Aziz et al., 2009). Besides the combinatorial design problem, the permutation also had been used to determine the determinant of a square matrix (Bankier, 1961; Pavlovic, 1961; Thongchiew, 2007).

Pavlovic (1961) listed three general procedures of Sarrus Rule for finding the determinant D of square matrix with order $n \geq 3$ where:

$$D = |a_{ij}| \ (i, j = 1, 2, 3, ..., n)$$

The algorithm was as follows:

(i) Find the $\frac{(n-1)!}{2}$ permutations to be obtained by permutating the columns from second to the nth (with condition: do not take two permutations whose indices proceed in the reverse order), the first column remaining in the same place of the diagram.

(ii) Rewrite the first $n-1$ columns to the right to all of $\frac{(n-1)!}{2}$ determinants obtained with the shown permutations of columns and carry out the multiplication of elements along the traced arrows.

(iii) Find the even or odd of every multiplication of elements in (ii).

From these three procedures, we found that in procedure (i), the specific method for finding $\frac{(n-1)!}{2}$ permutation is undefined. It can be seen from an example given in their paper for $n = 5$, there exist two permutations where the indices proceed in reverse order.

It violated the principle stated in procedure (i). Meanwhile, Bankier (1961) discussed the same procedure for $n \leq 5$.

Beside Pavlovic and Bankier's work, Thongchiew (2007) developed the sequential algorithm for finding determinant using permutation method based on Leibniz (1658) definition. He derived partial cyclic method for generating permutation and applied it in determining the determinant. Unfortunately, his algorithm was not analyzed and compared to other existing method. Inspiring by Pavlovic (1960), Bankier (1961), and Thongchiew (2007) work, we extend the Sarrus Rule method for any size of square matrices via permutation approach.

2.5.5 Division Free Parallel Algorithm for Finding Determinant

Among the dvision free method for finding the determinant, only cofactor expansion method was parallelized and done by Sasaki and Kanada (1981), and Goldfinger (2008). Sasaki and Kanada (1981) designed parallel algorithm for minor expansion. However, their parallel algorithms never have been tested and compared.

Meanwhile Goldfinger (2008) designed a parallel algorithm for cofactor expansion method in cell processor by distributing each $a_{ij}M_{ij}$ to different Synergistic Processing Unit (SPU). Cell processor is a creative architecture that allow for parallel computing and Single Instruction Multiple Data (SIMD) operations. The order complexity of entire algorithm is $O((n!)^2)$ which was improved than order complexity of sequential cofactor expansion algorithm $O(n((n-1)!)^3)$.

Overall, a parallel algorithm for finding the determinant based on permutation has yet to be constructed.

2.6 Summary

From the literature, we found that no research had been conducted in developing sequential and parallel algorithm for generating permutation and its application in determining the determinant of square matrix based on cross multiplication method known as the Sarrus Rule.

CHAPTER THREE

DEVELOPING NEW SEQUENTIAL ALGORITHMS FOR LISTING ALL PERMUTATIONS USING STARTER SETS

3.1 Introduction

In this chapter, we present two distinct techniques for generating starter sets and listing $n!$ distinct permutations. These approaches are motivated by Ibrahim et al.(2010) work where they introduced a new permutation technique based on distinct starter sets by employing circular and reversing operation. The crucial tasks of Ibrahim et al.(2010) method were distinct starter sets generation and the equivalence starter sets elimination. Although this technique was simple and easy to use, eliminating the equivalence starter sets unfortunately was a complicated process as the number of elements increased. We will overcome this drawback by proposing two new operation strategies for generating distinct starter sets and therefore the equivalence starter sets elimination process is avoidable. At the end, all $n!$ distinct permutations will be listed down.

This chapter begins with the introduction. Then some preliminary definitions are defined in Section 3.2. Meanwhile in Section 3.3, the algorithm developments are presented where two operation strategies for generating starter sets are derived. Finally the theoretical and numerical results of the algorithms are discussed in Section 3.4 and 3.5 respectively.

3.2 Preliminary Definitions

The following definitions will be used throughout this study.

Definition 3.2.1. *A starter set, S is a basis to enumerate other permutations.*

Definition 3.2.2. *An equivalence starter set is a set that can produce the same permutation from other starter set.*

Each starter set has an equivalence starter set where it can be produced by reversing the $(n-1)$ elements of the starter set itself.

Definition 3.2.3. *The circular operation (CO) over k elements is the process where the k elements of permutation are rotated.*

The algorithm of CO is as follows:

---

for $i = 1$ to k **do**
  $old = num[i]$
  $num[i] = num[i+1]$
  $num[i+1] = old$
end for

Definition 3.2.4. *The exchange operation is a interchanges process over two integers k and l, $k \neq l$, but leaves all other integers fixed.*

Definition 3.2.5. *The reverse set is a set that is produced by reversing the order of permutation set.*

Definition 3.2.6. *A Latin square of order n is an $n \times n$ array in which n distinct symbols are arranged where each element occurs once in each row and column.*

Definition 3.2.7. *The circular permutation of order n (CP) is a Latin square of order n which is obtained by employing the circular permutation operation over all elements.*

Example 3.2.8.

Let $n = 4$ and without loss of generality, take $A = [1, 3, 2, 4]$ as a starter with fixed element 1 . By rotating all elements to the left recursively, yield the next three arrays.

$$\begin{array}{cccc} 1 & 3 & 2 & 4 \\ 3 & 2 & 4 & 1 \\ 2 & 4 & 1 & 3 \\ 4 & 1 & 3 & 2 \end{array}$$

If we cycle the last row, we will get the original starter set and the cycle is completed. Hence, it takes four steps for the original starter set to get back to itself. Therefore we say the above 4×4 array of permutation as circular permutation of order 4.

Definition 3.2.9. *The reverse of circular permutation (RoCP) is also a Latin square of order n which is obtained by reversing arrangement element in each row of circular permutation.*

Example 3.2.10.

From Example 3.2.8, the following 4×4 array of permutation is the reverse of circular permutation.

$$\begin{array}{cccc} 4 & 2 & 3 & 1 \\ 1 & 4 & 2 & 3 \\ 3 & 1 & 4 & 2 \\ 2 & 3 & 1 & 4 \end{array}$$

The next example is given to demonstrate the equivalence starter sets generate the same permutations as generated by the starter sets.

Example 3.2.11.

Consider $n = 3$, and we fix element 1. There are two starters : $[1, 2, 3]$ and $[1, 3, 2]$. The circular process is applied on the both starters. The CP of each starter is listed as follows:

$$\left|\begin{array}{ccc||ccc} 1 & 2 & 3 & 1 & 3 & 2 \\ 2 & 3 & 1 & 3 & 2 & 1 \\ 3 & 1 & 2 & 2 & 1 & 3 \end{array}\right|$$

Then we apply reversing process to either CP of the starter set i.e. $[1, 2, 3]$ and its RoCP as follows:

$$\begin{vmatrix} 3 & 2 & 1 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \end{vmatrix}$$

The RoCP of the starter $[1, 2, 3]$ generate the same permutation as $[1, 3, 2]$. Therefore we called $[1, 3, 2]$ as equivalence starter. Thus we need to discard the equivalence starter to avoid repetition.

3.3 Algorithm Development for Permutation

The development of a new algorithm is divided into two stages. The first stage is starter sets generation while the second stage is permutation generation by exploiting the results in the first stage. The methods to generate $\frac{(n-1)!}{2}$ starter sets are based on circular (non exchange) and exchange operations. For listing all $n!$ distinct permutations, the CP and RoCP operations will be employed. Thus it would be worthwhile to see the advantages of these two operations in term of generating starter sets and order of complexity of the algorithms.

3.3.1 Circular Operation Strategy

Let S be the set of n elements such that $S = [1, 2, 3, ..., n]$. The circular operation will be used on both starter sets and permutation generations. Generally the operation starts from the last three elements until n elements are selected for cycling inductively. The process of generating starter sets is discussed in detail for case $n = 4$ and 5 in Section 3.3.1.1 and then exploited them for listing n! permutations in Section 3.3.1.2. Then, the generalisation of the circular operation is derived in Section 3.3.1.3.

3.3.1.1 Starter Sets Generation Under Circular Operation

A step by step procedure for generating starter sets is given for case $n = 4$ and 5.

Let $n = 4$.

Step 1: Let $S = [1, 2, 3, 4]$ be an initial permutation and without loss of generality, the first element is fixed.

Step 2: Identify the last three elements of initial permutation from Step 1. Employing CO to the last three elements on initial permutation from Step 1 will produce other three distinct starter sets is as follows:

1	2	3	4
1	3	4	2
1	4	2	3

Figure 3.1: List of Starter Sets for $n = 4$

Next, the circular strategy to obtain starter sets for $n = 5$ is demonstrated.

Step 1: Let $S = [1, 2, 3, 4, 5]$ be an initial permutation and without loss of generality, the first element is fixed.

Step 2: Identify the last three elements of initial permutation from Step 1. Employing CO to the last three elements on initial permutation from Step 1 will produce other three distinct starter sets is as follows:

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

Figure 3.2: Starter Sets by Performing CO over Last Three Elements

Step 3: Identify the last four elements of starter sets from Step 2. Employing CO to the last four elements on each starter set from Step 2 will produce 12 distinct starter sets is as follows:

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

Figure 3.3: List of Starter Sets for $n = 5$

Next section discusses the generation of n! permutation using starter sets.

3.3.1.2 Permutation Generation under Circular and Reversing Operation

The $n!$ distinct permutations are listed down where column A represents the CP and column B represents the RoCP. Each starter set is exploited by employing circular permutation and reversing operations over n elements. Figure 3.4 and 3.5 represent the $n!$ permutations on starter sets which was generated by the circular operation.

Case $n = 4$

1	**2**	**3**	**4**	4	3	2	1
2	3	4	1	1	4	3	2
3	4	1	2	2	1	4	3
4	1	2	3	3	2	1	4
1	**3**	**4**	**2**	2	4	3	1
3	4	2	1	1	2	4	3
4	2	1	3	3	1	2	4
2	1	3	4	4	3	1	2
1	**4**	**2**	**3**	3	2	4	1
4	2	3	1	1	3	2	4
2	3	1	4	4	1	3	2
3	1	4	2	2	4	1	3
Column A				**Column B**			

Figure 3.4: List of 4! Permutations

Case $n = 5$.

Column A					**Column B**				
1	**2**	**3**	**4**	**5**	5	4	3	2	1
2	3	4	5	1	1	5	4	3	2
3	4	5	1	2	2	1	5	4	3
4	5	1	2	3	3	2	1	5	4
5	1	2	3	4	4	3	2	1	5
1	**3**	**4**	**5**	**2**	2	5	4	3	1
3	4	5	2	1	1	2	5	4	3
4	5	2	1	3	3	1	2	5	4
5	2	1	3	4	4	3	1	2	5
2	1	3	4	5	5	4	3	1	2
1	**4**	**5**	**2**	**3**	3	2	5	4	1
4	5	2	3	1	1	3	2	5	4
5	2	3	1	4	4	1	3	2	5
2	3	1	4	5	5	4	1	3	2
3	1	4	5	2	2	5	4	1	3
1	**5**	**2**	**3**	**4**	4	3	2	5	1
5	2	3	4	1	1	4	3	2	5
2	3	4	1	5	5	1	4	3	2
3	4	1	5	2	2	5	1	4	3
4	1	5	2	3	3	2	5	1	3
1	**2**	**4**	**5**	**3**	3	5	4	2	1
2	4	5	3	1	1	3	5	4	2
4	5	3	1	2	2	1	3	5	4
5	3	1	2	4	4	2	1	3	5
3	1	2	4	5	5	4	2	1	3
1	**4**	**5**	**3**	**2**	2	3	5	4	1
4	5	3	2	1	1	2	3	5	4
5	3	2	1	4	4	1	2	3	5
3	2	1	4	5	5	4	1	2	3
2	1	4	5	3	3	5	4	1	2

Figure 3.5: List of $5!$ *Permutations*

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

Column A **Column B**

(Continue Figure 3.5)

As can be observed from Figures 3.4 and 3.5, there was no redundancy permutation occurs when the starter sets are exploited for generating all permutations by employing CP and RoCP operations.

Remark 3.3.1. *The **bold** permutation in **Column A** represents the starter sets.*

3.3.1.3 Circular Algorithm

Consider the algorithm of circular operation as PERMUT1 which is a recursion algorithm with only one recursive call. Let integer n be an initial inputs, and $temp = n - 1$ is an initial rank of the procedure PERMUT1. At each stage of recursion, a rank for algorithm PERMUT1 decreases from $n - 1$ to 2 where each recursion call updates the entries in the storage. The general algorithm for permutation generation by employing CP and RoCP operation on starter sets which is generated from the circular operation is as follows:

Let S be the set of n elements i.e. $S = [1, 2, 3, 4, \ldots, k, k + 1, \ldots, n - 1, n]$.

Algorithm 3.1 PERMUT1

```
PERMUT1(temp)
if temp = 2 then
    for i = 1 to n do
        performing CP and RoCP operation over all element
    end for
    return
end if
temp = temp − 1
for i = n to temp do
    performing circular operation (CO) to the last temp element
    call PERMUT1(temp)
end for
```

Step by step process of PERMUT1 algorithm.

Step 1: Let $[1, 2, 3, 4, \ldots, k, k + 1, \ldots, n - 2, n - 1, n]$ be an initial permutation and without

loss of generality, the first element is fixed.

Step 2: Identify the last three elements of each starter set in Step 1. By employing

circular operation (CO) to last three elements on each starter sets in Step 1,

the three distinct starter sets are obtained.

Step 3: Identify the last four elements of each starter set in Step 2. By employing circular

operation (CO) to last four elements on each starter set in Step 2, the 12

distinct starter sets are obtained.

$\vdots$

Step $n-2$: Identify the last $(n-1)$ elements of each starter set in Step $(n-3)$. By employing circular operation (CO) to the last $(n-1)$ elements on each starter set in Step $(n-2)$, the $\frac{(n-1)!}{2}$ distinct starter sets are obtained.

Step $n-1$: Perform CP and RoCP operations simultaneously to all n elements of $\frac{(n-1)!}{2}$ distinct starter sets and $n!$ distinct permutations are obtained.

Step n: Display all $n!$ permutations.

The second strategy for generating starter sets and listing all permutation is discussed in the following section.

3.3.2 Exchange Operation Strategy

Let S be the set of n elements such that $S = [1, 2, 3, ..., n]$. This strategy is different from the circular strategy. in term of starter sets generation. For initial start, an element on $(n-2)$th is selected to exchange until second element is selected for exchange to the right inductively. The process of generating starter sets is discussed in detail for case $n = 4$ and 5 in Section 3.3.2.1 and the generation of $n!$ permutation using starter sets is discussed in Section 3.3.2.2. Finally, the generalization of the exchange operation is given in Section 3.3.2.3.

3.3.2.1 Starter Sets Generation Under Exchange Operation

The step by step of starter sets derivation is demonstrated for $n = 4$ and 5 as follows.

Step 1: Let $[1, 2, 3, 4]$ be an initial permutation and without loss of generality, the first element is fixed.

Step 2: Identify the element in the $(n-2)$th position i.e. element '2'. Exchange this element until it reaches the nth (last) position. We produce three distinct starter sets as follows

1	2	3	4
1	3	2	4
1	3	4	2

Figure 3.6: List of Starter Sets for $n = 4$

Next, we shall illustrate the second strategy for $n = 5$.

Step 1: Set $[1, 2, 3, 4, 5]$ be an initial permutation and without loss of generality, the first element is fixed.

Step 2 :Identify the element in the $(n-2)$th position i.e. element '3'. Exchange this element until it reaches the nth (last)position. We produce three distinct starter sets as follows:

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

Figure 3.7: Starter Sets from the Exchange of the $(n-2)$*th Element*

Step 3 : Identify the element in the $(n-3)$th position i.e. '2' in each starter sets from Step 2. Exchange this element until it reaches the nth (last)position. We produce other 12 distinct starter sets as shown in Figure 3.8 below.

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

Figure 3.8: All Starter Set for $n = 5$

Next section describes the generation of n! permutation using starter sets.

3.3.2.2 Permutation Generation Under Circular and Reversing Operations

The starter sets are then exploited for listing all permutations using circular and reversing operations. Example for $n = 4$ and 5 are demonstrated.

Case $n = 4$.

1	**2**	**3**	**4**	4	3	2	1
2	3	4	1	1	4	3	2
3	4	1	2	2	1	4	3
4	1	2	3	3	2	1	4
1	**3**	**2**	**4**	4	2	3	1
3	2	4	1	1	4	2	3
2	4	1	3	3	1	4	2
4	1	3	2	2	3	1	4
1	**3**	**4**	**2**	2	4	3	1
3	4	2	1	1	2	4	3
4	2	1	3	3	1	2	4
2	1	3	4	4	3	1	2
Column A				**Column B**			

Figure 3.9: List of 4! Permutations

Case $n = 5$

Column A					**Column B**				
1	**2**	**3**	**4**	**5**	5	4	3	2	1
2	3	4	5	1	1	5	4	3	2
3	4	5	1	2	2	1	5	4	3
4	5	1	2	3	3	2	1	5	4
5	1	2	3	4	4	3	2	1	5
1	**3**	**2**	**4**	**5**	5	4	2	3	1
3	2	4	5	1	1	5	4	2	3
2	4	5	1	3	3	1	5	4	2
4	5	1	3	2	2	3	1	5	4
5	1	3	2	4	4	2	3	1	5
1	**3**	**4**	**2**	**5**	5	2	4	3	1
3	4	2	5	1	1	5	2	4	3
4	2	5	1	3	3	1	5	2	4
2	5	1	3	4	4	3	1	5	2
5	1	3	4	2	2	4	3	1	5
1	**3**	**4**	**5**	**2**	2	5	4	3	1
3	4	5	2	1	1	2	5	4	3
4	5	2	1	3	3	1	2	5	4
5	2	1	3	4	4	3	1	2	5
2	1	3	4	5	5	4	3	1	2
1	**2**	**4**	**3**	**5**	5	3	4	2	1
2	4	3	5	1	1	5	3	4	2
4	3	5	1	2	2	1	5	3	4
3	5	1	2	4	4	2	1	5	3
5	1	2	4	3	3	4	2	1	5
1	**4**	**2**	**3**	**5**	5	3	2	4	1
4	2	3	5	1	1	5	3	2	4
2	3	5	1	4	4	1	5	3	2
3	5	1	4	2	2	4	1	5	3
5	1	4	2	3	3	2	4	1	5

Figure 3.10: List of $5!$ *Permutations*

Column A					Column B				
1	**4**	**3**	**2**	**5**	5	2	3	4	1
4	3	2	5	1	1	5	2	3	4
3	2	5	1	4	4	1	5	2	3
2	5	1	4	3	3	4	1	5	2
5	1	4	3	2	2	3	4	1	5
1	**4**	**3**	**5**	**2**	2	5	3	4	1
4	3	5	2	1	1	2	5	3	4
3	5	2	1	4	4	1	2	5	3
5	2	1	4	3	3	4	1	2	5
2	1	4	3	5	5	3	4	1	2
1	**2**	**4**	**5**	**3**	3	5	4	2	1
2	4	5	3	1	1	3	5	4	2
4	5	3	1	2	2	1	3	5	4
5	3	1	2	4	4	2	1	3	5
3	1	2	4	5	5	4	2	1	3
1	**4**	**2**	**5**	**3**	3	5	2	4	1
4	2	5	3	1	1	3	5	2	4
2	5	3	1	4	4	1	3	5	2
5	3	1	4	2	2	4	1	3	5
3	1	4	2	5	5	2	4	1	3
1	**4**	**5**	**2**	**3**	3	2	5	4	1
4	5	2	3	1	1	3	2	5	4
5	2	3	1	4	4	1	3	2	5
2	3	1	4	5	5	4	1	3	2
3	1	4	5	2	2	5	4	1	3
1	**4**	**5**	**3**	**2**	2	3	5	4	1
4	5	3	2	1	1	2	3	5	4
5	3	2	1	4	4	1	2	3	5
3	2	1	4	5	5	4	1	2	3
2	1	4	5	3	3	5	4	1	2

(Continue Figure 3.10)

As can be observed from Figures 3.9 and 3.10, there was no redundancy permutation occurs when the starter sets are exploited for generating all permutations by employing CP and RoCP operations.

Remark 3.3.2. *The **bold** permutation in **Column A** represents the starter sets.*

3.3.2.3 Exchange Algorithm

Let define the algorithm for exchange operation strategy as a PERMUT2 which was also a recursion procedure. Let integer n be an initial input, and $temp = n-1$ is an initial rank of the procedure PERMUT2. At each stage of recursion, a rank for procedure PERMUT2 is reduced from $n-1$ to 2 where each recursion call updates the entries in the storage. The general algorithm for permutation generation by employing CP and RoCP operation on starter sets which is generated from the exchange operation is presented as follows:

Algorithm 3.2 PERMUT2

```
PERMUT2(temp)
if temp = 2 then
    for i = 1 to n do
        performing CP and RoCP operations for all element
    end for
    return
end if
temp = temp − 1
for i = temp to n do
    performing exchanges operation to the element at temp-th position
    call PERMUT2(temp)
end for
```

The general algorithm for permutation generation by employing CP and RoCP operations on starter sets which is generated from the exchange operation as follows:

Step 1: Set $[1, 2, 3, 4, \ldots, k, k+1, \ldots, n-2, n-1, n]$ be an initial permutation and without

loss of generality, the first element is fixed.

Step 2: Identify the element in the $(n-2)$th position of the initial permutation in Step 1.

Exchange this element until it reaches the nth (last) position. Hereby three

distinct starter sets are obtained.

Step 3: Identify the element in the $(n-3)$th position of each starter set in Step 2.

Exchange this element until it reaches the nth (last) position. Hereby 12 distinct

starter sets are obtained.

$\vdots$

Step $n-2$: Identify the element in the second position of each starter set in Step $(n-3)$.

Exchange this element until it reaches the nth (last) position. At this step, the $\frac{(n-1)!}{2}$ distinct starter sets are obtained.

Step $n-1$: Perform CP and RoCP operations simultaneously to all n elements of $\frac{(n-1)!}{2}$

distinct starter sets and $n!$ distinct permutations are obtained.

Step n: Display all $n!$ permutations.

The theoretical results for permutation generation are presented in the following section.

3.4 Theoretical Results

The following lemmas and theorem are produced from the recursive circular and exchange operation for starter set generation and employing CP and RoCP for listing all permutations.

Lemma 3.4.1. *The number of distinct permutations produced by each distinct starters set by performing the circular permutation and reversing of circular permutation operation over all elements is* $2n$.

Proof. Suppose a starter set $A = [1, 2, ..., n-1, n]$ with n distinct elements. By Definition

3.2.3, n elements are employed using CO, the following permutations are obtained.

$$\begin{array}{cccccc}
1 & 2 & 3 & ... & n-1 & n \\
2 & 3 & 4 & ... & n & 1 \\
3 & 4 & 5 & ... & 1 & 2 \\
4 & 5 & 6 & \dots & 2 & 3 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
n & 1 & \dots & \dots & n-2 & n-1
\end{array}$$

Thus n distinct circular permutations (CP) are produced. Then from Definition 3.2.8, the reversing of circular permutation operation of this CP, the next n distinct circular permutations are obtained as follows

$$\begin{array}{cccccc}
n & n-1 & ... & 3 & 2 & 1 \\
1 & n & ... & 4 & 3 & 2 \\
2 & 1 & ... & 5 & 4 & 3 \\
3 & 2 & \dots & 6 & 5 & 4 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
n-1 & n-2 & \dots & 2 & 1 & n
\end{array}$$

Since each CP and RoCP has order n, then the total of $2n$ distinct permutations is produced. □

Lemma 3.4.2. *The number of generated distinct starter sets under circular operation for* $n \geq 3$ *is* $\frac{(n-1)!}{2}$.

Proof. Suppose $[1, 2, 3, ..., n-3, n-2, n-1, n]$ be an initial starter for any $n \geq 3$. From Definition 3.2.3, by employing CO to the last three elements, three distinct starter sets are

produced as follows:

$$\begin{array}{cccccccc} 1 & 2 & 3 & ... & n-3 & n-2 & n-1 & n \quad \text{(starter 1)} \\ 1 & 2 & 3 & ... & n-3 & n-1 & n & n-2 \quad \text{(starter 2)} \\ 1 & 2 & 3 & ... & n-3 & n & n-2 & n-1 \quad \text{(starter 3)} \end{array}$$

Then for each previous starter set, the last four elements will be selected and by employing CO on these elements of previous starter sets. By definition 3.2.3 four distinct starters are produced as follows:

$$\begin{array}{lcccccccc} \text{From starter 1:} & \mathbf{1} & \mathbf{2} & \mathbf{3} & ... & \mathbf{n-3} & \mathbf{n-2} & \mathbf{n-1} & \mathbf{n} \\ & 1 & 2 & 3 & ... & n-2 & n-1 & n & n-3 \\ & 1 & 2 & 3 & ... & n-1 & n & n-3 & n-2 \\ & 1 & 2 & 3 & ... & n & n-3 & n-2 & n-1 \\ \text{From starter 2:} & \mathbf{1} & \mathbf{2} & \mathbf{3} & ... & \mathbf{n-3} & \mathbf{n-1} & \mathbf{n} & \mathbf{n-2} \\ & 1 & 2 & 3 & ... & n-1 & n & n-2 & n-3 \\ & 1 & 2 & 3 & ... & n & n-2 & n-3 & n-1 \\ & 1 & 2 & 3 & ... & n-2 & n-3 & n-1 & n \\ \text{From starter 3:} & \mathbf{1} & \mathbf{2} & \mathbf{3} & ... & \mathbf{n-3} & \mathbf{n} & \mathbf{n-2} & \mathbf{n-1} \\ & 1 & 2 & 3 & ... & n & n-2 & n-1 & n-3 \\ & 1 & 2 & 3 & ... & n-2 & n-1 & n-3 & n \\ & 1 & 2 & 3 & ... & n-1 & n-3 & n & n-2 \end{array}$$

Thus, at this stage the total starter sets is $3 \times 4 = 12$. The processes will be repeated recursively until the last $(n-1)$ elements are circulated.

$$
\begin{array}{rcl}
3 \text{ last elements} & \Rightarrow & 3 \text{ starter sets} \\
4 \text{ last elements} & \Rightarrow & 4 \text{ starter sets} \\
5 \text{ last elements} & \Rightarrow & 5 \text{ starter sets} \\
6 \text{ last elements} & \Rightarrow & 6 \text{ starter sets} \\
\vdots & & \vdots \\
\vdots & & \vdots \\
(n-2) \text{ last elements} & \Rightarrow & (n-2) \text{ starter sets} \\
(n-1) \text{ last elements} & \Rightarrow & (n-1) \text{ starter sets}
\end{array}
$$

By product rule, we produce

$$
\begin{aligned}
& (3 \times 4 \times ... \times n-1) \\
=& \frac{1 \times 2}{2} \times (3 \times 4 \times ... \times n-1)) \\
=& \frac{(n-1)!}{2} \text{distinct starter sets}
\end{aligned}
$$

□

Lemma 3.4.3. *The number of generated distinct starter sets under exchange operation for* $n \geq 3$ *is* $\frac{(n-1)!}{2}$.

Proof. Suppose $[1, 2, 3, ..., n-3, n-2, n-1, n]$ be an initial starter for any $n \geq 3$. The first element will be selected from $(n-2)$th position i.e. element $n-2$. Then by moving that element to the right until it reaches nth position, three distinct starter sets are obtained as follows:

$$
\begin{array}{cccccccccc}
1 & 2 & 3 & ... & n-3 & \mathbf{n-2} & n-1 & n & \text{(starter 1)} \\
1 & 2 & 3 & ... & n-3 & n-1 & \mathbf{n-2} & n & \text{(starter 2)} \\
1 & 2 & 3 & ... & n-3 & n-1 & n & \mathbf{n-2} & \text{(starter 3)}
\end{array}
$$

Then for each previous starter set, element in $(n-3)$th will be selected i.e. element $n-3$. Then by moving that element to the right until it reaches the nth position from each

previous starter set, four distinct starters are produced as follows:

$$
\begin{array}{rcccccccc}
\text{From starter 1:} & \mathbf{1} & \mathbf{2} & \mathbf{3} & ... & \mathbf{n-3} & \mathbf{n-2} & \mathbf{n-1} & \mathbf{n} \\
& 1 & 2 & 3 & ... & n-2 & \mathbf{n-3} & n-1 & n \\
& 1 & 2 & 3 & ... & n-2 & n-1 & \mathbf{n-3} & n \\
& 1 & 2 & 3 & ... & n-2 & n-1 & n & \mathbf{n-3} \\
\\
\text{From starter 2:} & \mathbf{1} & \mathbf{2} & \mathbf{3} & ... & \mathbf{n-3} & \mathbf{n-1} & \mathbf{n-2} & \mathbf{n} \\
& 1 & 2 & 3 & ... & n-1 & \mathbf{n-3} & n-2 & n \\
& 1 & 2 & 3 & ... & n-1 & n-2 & \mathbf{n-3} & n \\
& 1 & 2 & 3 & ... & n-1 & n-2 & n & \mathbf{n-3} \\
\text{From starter 3:} & \mathbf{1} & \mathbf{2} & \mathbf{3} & ... & \mathbf{n-3} & \mathbf{n-1} & \mathbf{n} & \mathbf{n-2} \\
& 1 & 2 & 3 & ... & n-1 & \mathbf{n-3} & n & n-2 \\
& 1 & 2 & 3 & ... & n-1 & n & \mathbf{n-3} & n-2 \\
& 1 & 2 & 3 & ... & n-1 & n & n-2 & \mathbf{n-3}
\end{array}
$$

After the element $(n-3)$th is selected, the total starter sets is $3 \times 4 = 12$. The processes will be repeated recursively until the element in second position is chosen.

$$
\begin{array}{ccc}
(n-2)th \text{ position} & \Rightarrow & 3 \text{ starter sets} \\
(n-3)th \text{ position} & \Rightarrow & 4 \text{ starter sets} \\
(n-4)th \text{ position} & \Rightarrow & 5 \text{ starter sets} \\
(n-5)th \text{ position} & \Rightarrow & 6 \text{ starter sets} \\
\vdots & & \vdots \\
(n-i+1)th \text{ position} & \Rightarrow & i \text{ starter sets} \\
(n-i)th \text{ position} & \Rightarrow & i+1 \text{ starter sets} \\
(n-i-1)th \text{ position} & \Rightarrow & i+2 \text{ starter sets} \\
\vdots & & \vdots \\
3rd \text{ position} & \Rightarrow & n-2 \text{ starter sets} \\
2nd \text{ position} & \Rightarrow & n-1 \text{ starter sets}
\end{array}
$$

Using product rule, we have

$$\begin{aligned}&(3 \times 4 \times ... \times n-1)\\ =&\frac{1 \times 2}{2} \times (3 \times 4 \times ... \times n-1))\\ =&\frac{(n-1)!}{2}\text{distinct starter sets}\end{aligned}$$

□

Remark 3.4.4. *The formula of* $\frac{(n-1)!}{2}$ *was only defined for* $n \geq 3$*. For case* $n = 2$ *is impossible because it has only one distinct starter set while* $\frac{(2-1)!}{2} = \frac{1}{2}$.

Theorem 3.4.5. *Employing Circular Permutation and Reverse of Circular Permutation on* $\frac{(n-1)!}{2}$ *distinct circular starter sets produces* $n!$ *distinct permutations.*

Proof. From Lemma 3.4.2 there are $\frac{(n-1)!}{2}$ distinct starter sets produced under circular operation. Then from Lemma 3.4.1, $2n$ distinct permutations are obtained by employing circular and reversing operation on the starter sets. Thus $\frac{(n-1)!}{2} \times 2n = n!$ permutations are generated. □

Theorem 3.4.6. *Employing Circular Permutation and Reverse of Circular Permutation on* $\frac{(n-1)!}{2}$ *distinct exchange starter sets produces* $n!$ *distinct permutations.*

Proof. From Lemma 3.4.3, there are $\frac{(n-1)!}{2}$ distinct starter sets produced under exchange operation. Then from Lemma 3.4.1, $2n$ distinct permutations are obtained by employing circular and reversing operation on the starter sets. Thus $\frac{(n-1)!}{2} \times 2n = n!$ permutations are generated. □

Numerical results will be presented in the following section.

3.5 Numerical Results

For permutation generation algorithm performance, our new algorithms were compared to other non exchanges based permutation generation program namely Lexicographic order

(Ord-Smith,1970), Langdon (1967), and Thongchiew (2007) in term of time computation. All sequential algorithms are implemented in C language and tested on the HP Computer with Intel Xeon E5504 2.0 GHz processor and 4.00 GB Random Access Memory (RAM). The result is an execution time without printing statements.

For the time computation among the recursive algorithm, our two recursive algorithms were compared to Lexicographic order algorithm. The results are displayed in Table 3.1.

Table 3.1: The Computation Time of Recursive Algorithm (in seconds)

n	PERMUT1	PERMUT2	Lexicographic
8	0.000746	0.000753	0.003411
9	0.007398	0.007478	0.031300
10	0.110915	0.079344	0.311021
11	1.038668	0.935565	3.402234
12	12.560315	11.967977	40.772082
13	172.104967	167.362079	530.842444
14	2561.104967	2480.992924	7454.880165
15	41885.652796	38993.829780	117521.839515

As shown in Table 3.1, new recursive algorithm (PERMUT1 and PERMUT2) took less time when compared to Lexicographic order. The results among iterative algorithms are displayed in Table 3.2. In spite of recursive algorithm, we also implemented PERMUT1 algorithm in iterative manner since Langdon (1967) and Thongciew (2007) algorithm were developed in iterative procedure. We name our iterative algorithm as PERMUTIT3.

Table 3.2: The Computation Time of Iterative Algorithm (in seconds)

n	PERMUTIT3	Langdon	Thongchiew
8	0.000788	0.001575	0.007705
9	0.007757	0.015513	0.069762
10	0.104555	0.170119	0.696822
11	1.033967	2.011029	7.675245
12	13.282527	26.356982	92.155745
13	183.498911	365.671963	1198.827993
14	2711.660657	5427.021299	22448.205515
15	42586.687827	85173.826685	246139.962499

From Table 3.2, we observed that PERMUTIT3 is faster than Langdon (1967) and Thongchiew (2007). Langdon (1967) is two times slower than PERMUTIT3 for $n > 9$. Meanwhile Thongchiew (2007) is the slowest among these three algorithms.

The following table shows the results of run times among new algorithms for permutation generation.

Table 3.3: The Computation Time Among New Algorithms (in seconds)

n	PERMUT1 (1st strategy rec.)	PERMUT2 (2nd strategy rec.)	PERMUTIT3 (1st strategy iter.)
8	0.000746	0.000753	0.000092
9	0.007398	0.007478	0.007757
10	0.110915	0.079344	0.104555
11	1.038668	0.935565	1.033967
12	12.560315	11.96799	13.282527
13	172.104967	167.362079	183.498911
14	2561.104967	2480.992924	2711.660657
15	41885796.652	38993.829780	42586.687827

Table 3.3 indicates that PERMUT1 and PERMUT2 algorithms performed better than PERMUTIT3 algorithm except at $n = 8$. In other words, the recursive algorithms are the best in terms of time computation. The time of execution was incremented consistently when n became larger. For $n = 8$ until 15, our programs (PERMUT1, PERMUT2 and PERMUTIT3) ran better in time compared to Lexicographic order, Langdon (1967) and Thongchiew (2007). The factor that contributed to lesser execution time of our algorithm was due to fact that CP and RoCP operations were performed simultaneously for listing all permutations. On the other hand, Langdon (1967) and Thongchiew (2007) algorithms generated all permutations and required more steps. This factor might have affected the computational time.

Remark 3.5.1. *All the results from the programs are given in* ***Appendix E****. The example output is given for* $n = 5$.

Next section, the order of complexity of our algorithms is discussed.

3.5.1 Computational Complexity of Permutation Algorithm

The time $T(n)$ calculation of a program is the sum of the compile time and the run (execution) time. Finding the exact formula of run time is an impossible task, since the time needed for an addition, subtraction, multiplication, etc., often depended on the numbers being added, subtracted, multiplied, etc (Horowitz et al., 2008). So we identified the most dominant operation of the algorithm which contributed to the total running time.

Since the new algorithm used recursion to complete tasks which depended on the previous rank of the task, the order of complexity of this algorithm will be calculated on the recursion function such as PERMUT1, and PERMUT2. Meanwhile for PERMUTIT3, we assumed that the order of complexity similar to PERMUT1.

3.5.1.1 Pseudocode of Circular Operation Strategy under Recursion (PERMUT1)

This pseudocode PERMUT1 is a recursive algorithm for generating permutation under circular operation. The input data is n which represents a number of elements and the output is a list of $n!$ distinct permutations.

Pseudocode PERMUT1($temp$)

1: **if** $temp = 2$ **then**
2: **for** $i = 1$ to n **do**
3: $old = a[1]$
4: **for** $k = 1$ to $n - 1$ **do**
5: $a[k] = a[k + 1]$
6: **end for**
7: $a[n] = old$
8: **end for**
9: **return**
10: **end if**

11: $temp = temp - 1$

12: **for** $i = n$ to $temp$ **do**

13: $old = a[temp]$

14: **for** $k = temp$ to $n - 1$ **do**

15: $a[k] = a[k + 1]$

16: **end for**

17: $a[n] = old$

18: PERMUT1($temp$)

19: **end for**

The critical section in this algorithm is steps 12 - 19 where starter sets are generated. There is a nested loop whose contains a recursive call on less rank ($temp$ gets smaller). The initial $temp$ = $n - 1$. At step 11, the value of $temp$ is decreasing. The process will stop when $temp = 2$. On the other hand, the recursion call will stop when $temp = 2$ or the recursion will not be called when $n = 3$.

The order of complexity for steps 12 -19 is calculate as follows:

For any value of $temp$ for loop at steps 14-16, the operation needs is

$$(n - 1 - temp)$$

For the outer loop at steps 12 -19, the total operation is

$$(2 + [n - 1 - temp])$$

Since we set up the initial $temp = n - 1$, at steps 11, the new $temp = n - 2$.

$$\begin{aligned} &(2 + [n - 1 - (n - 2)]) \\ =&(2 + [1]) \end{aligned}$$

$temp = n - 3$

$$\begin{aligned}&(2 + [n - 1 - (n - 3)]) \times (2 + [1])\\ =&(2 + [2]) \times (2 + [1])\end{aligned}$$

$temp = n - 4$

$$\begin{aligned}&(2 + [n - 1 - (n - 4)]) \times (2 + [2]) \times (2 + [1])\\ =&(2 + [3]) \times (2 + [2]) \times (2 + [1])\end{aligned}$$

Until when $temp = 2$, number of operation is

$$\begin{aligned}&(2 + [n - 1 - (2)]) \times \cdots \times (2 + [2]) \times (2 + [1])\\ =&(n - 1) \times \cdots \times (5) \times ((4) \times (3)\\ \cong&((n - 1)!)\end{aligned}$$

The order of complexity for steps 11-19 are $O(((n - 1)!))$. From steps 1-9, double loops exist which has the complexity $O(n^2)$. Then in order to generate all permutations, all starter sets need to be exploited by performing that double loops cycling process. So by multiplying n^2 to $O(((n - 1)!)$, it is equal to $O((n(n)!))$. The order of complexity of the algorithm is $O((n(n)!))$.

3.5.1.2 Pseudocode of Exchange Operation Strategy under Recursion(PERMUT2)

The pseudocode of PERMUT2 is a recursive algorithm for generating permutation under circular operation. Meanwhile for generating starter sets, exchange operation is used. The input data is n which represents a number of elements and the output is a list of all distinct $n!$ permutations.

Pseudocode PERMUT2(k)

1: **if** $k = 2$ **then**
2: **for** $i = 1$ to n **do**
3: $old = a[1]$
4: **for** $k = 1$ to $n - 1$ **do**
5: $a[k] = a[k + 1]$
6: **end for**
7: $a[n] = old$
8: **end for**
9: **return**
10: **end if**
11: $temp = k - 1$
12: **for** $i = temp$ to n **do**
13: **if** $(i \neq n)$ **then**
14: $old = a[i]$
15: $a[i] = a[i + 1]$
16: $a[i + 1] = old$
17: **else**
18: $old = a[n]$
19: **for** $i = n$ to $temp - 1$ **do**
20: $a[i] = a[i - 1]$
21: **end for**
22: $a[temp] = old$
23: **end if**
24: PERMUT2($temp$)
25: **end for**

The critical section in this algorithm is steps 12 - 25 where it represents the steps for starter sets generation. There is a nested loop and in that loop, there is recursive call on

less temp (temp gets smaller). The initial $temp = k$ = $n - 1$. At step 12, the value of $temp$ decreases. The process of recursion starts at $k = n - 1$ and will stop when $k = 2$. On the other hand, the starter sets generation process will stop at $k = 2$.

At the steps $13 - 23$ which lies under loop at step 12, a number of computation of the step from 13 until 16 is a constant time, $O(1)$ complexity. Meanwhile for steps 18 -22 has $O(temp)$ complexity.

Thus the number of computation for steps 13 -17, is $\prod(temp - 1))$ for temp = $n - 1$ until 2. Meanwhile for next steps $18 - 22$, for each value of $temp$ from 3 until $n - 1$, it will be run once. Thus its complexity is $O((n - 2)^2)$.

Thus the order of complexity for starter sets generation is $O((n-2)!)+O((n-1)^2)$. After the starter sets are produced and stops at $temp = 2$, the program continues for generating all permutation which lies at steps 2 - 9.

Then by multiplying n^2 to the order of complexity of the starter set generation , the order of complexity of the algorithm is $O(n^2(n - 2)!) + O(n^4 - n^2) \cong O(n(n)!)$

3.5.1.3 Pseudocode of Circular Operation Strategy under Iteration (PERMUTIT3)

This pseudocode PERMUTIT3 is an iterative algorithm for generating permutation under circular strategy. The input data is n which represents as a number of elements and the output is a list of all $n!$ distinct permutations.

Pseudocode PERMUTIT3($temp, n$)

1: $k = temp$
2: **while** $k > 2$ **do**
3: print(n)
4: $k = temp$

5: **while** $k > 2$ **do**

6: $old = a[1]$

7: **for** $i = 1$ to $k - 1$ **do**

8: $a[i] = a[i + 1]$

9: **end for**

10: $a[k] = old$

11: **if** $k = 2$ or $a[k]! = k$ **then**

12: break

13: $k = k - 1$

14: **end if**

15: **end while**

16: **end while**

This algorithm starts with $temp = 3$. The order of complexity of this algorithm is O($(n(n)!)$ since PERMUTIT3 is an iterative algorithm for PERMUT1.

Generally in calculating order of complexity, the constant value is discarded. The comparison in term of order of complexity between existing permutation algorithms and new developed algorithms is given in Table 3.4.

Table 3.4: Comparison of Algorithm Order of Complexity

Algorithm	Order Complexity
PERMUT1	$O(nn!)$
PERMUT2	$O(nn!)$
Langdon (Sedgewick,1977)	$O(nn!)$

From Table 3.4, the result in term of order complexity verifies that new developed algorithms are comparable to Langdon algorithm.

3.6 Summary

The central idea of our work for listing permutation is a starter sets generation. The two new different strategies to generate the starter sets are presented based on circular operation, and exchange operation. Then both strategies are exploited to generate all $n!$ permutations using CP and RoCP operation. Furthermore, the proposed algorithms are verified by some theoretical works. The major difference of our strategies from other conventional permutation methods is that we employ the starter sets to list all permutations.

Thus, the first objective of this study i.e. constructing new sequential algorithms for permutation generation was achieved. The contributions of this chapter are as follows:

(i) New strategies for generating starter sets without generating the equivalence starter sets have been proposed and also supported by some new theoretical works.

(ii) New recursive algorithms and an iterative algorithm for permutation generation have been developed.

These new algorithms are proven better in computation time compared to Langdon, Thongchiew and Lexicographic algorithms for non-exchanges based category. Meanwhile in term of order complexity, new permutation algorithms are good as the Langdon algorithm.

CHAPTER FOUR

DEVELOPING NEW SEQUENTIAL DIVISION FREE METHOD FOR DETERMINANT

4.1 Introduction

Division free methods have two advantages. They can cater the entries of matrices in rational (Rote, 2001; Shin, 2002) and error of floating points can be avoided (Mahajan & Vinay, 1997). The examples of division free methods are cofactor expansion and cross multiplication method. The later method is also known as the Sarrus Rule only works for matrices of order $n \leq 3$. In this chapter, we derive a new division free method for computing the determinant by using new permutation algorithms which have been constructed in Chapter Three.

This chapter begins with some preliminary definitions in Section 4.2. Then it is followed by derivation of algorithm for computing the determinant of a square matrix in Section 4.3. Next, in Section 4.4, the general algorithm is derived. Meanwhile Section 4.5 presents some new theoretical works related to derivation of a new method. Finally, the performances of the new algorithm are analyzed in Section 4.6.

4.2 Preliminary Definitions

Let $A = [a_{ij}]$ represents an arbitrary $n \times n$ matrix. The determinant of A is denoted by $|A|$ or det(A). The arbitrary determinant, det(A)= $|a_{ij}|_n = |C_1\ C_2\ C_3\ ...\ C_n|$ is represented in column indices. The following definitions are given to define the main diagonal product and secondary diagonal product of a square matrix.

Definition 4.2.1. *The Main Diagonal Product (MDP) is a product of all n entries in the main diagonal of the square matrix.*

Definition 4.2.2. *The Secondary Diagonal Product (SDP) is a product of all n entries in the secondary diagonal of the square matrix.*

Example 4.2.3.

Let A be a matrix of 4×4 as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Then MDP and SDP of A are $a_{11}a_{22}a_{33}a_{44}$ and $a_{14}a_{23}a_{32}a_{41}$ respectively.

Definition 4.2.4. *The even (odd) starter sets is an even (respectively odd) permutation if it has an even (respectively odd) number of inversion.*

Definition 4.2.5. *The starter sets matrix of order n, A_i is the matrices which is generated from n column indices of starter sets where $1 \leq i \leq \dfrac{(n-1)!}{2}$.*

Definition 4.2.6. *The nth order diagram is a diagram generated from starter sets matrix of order n by appending the first $(n-1)$ columns to the right of origin starter set matrix.*

Definition 4.2.7. *The product diagonal of $A_{i,k}$ or $|A_{i,k}|$ is a sum of Main Diagonal product (MDP) and Secondary Diagonal Product (SDP) where $1 \leq i \leq \dfrac{(n-1)!}{2}$ and $0 \leq k \leq n-1$.*

Now we investigate the relationship between circular permutation and the Sarrus Rule for case $n = 3$.

Example 4.2.8. *Given the Sarrus rule of third order diagram as follows:*

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{23} & a_{21} & a_{22} \\ a_{31} & a_{32} & a_{33} & a_{31} & a_{32} \end{vmatrix}$$

Let us refer to the column indices of elements in the main diagonal $[a_{11}, a_{22}, a_{33}]$. We extract the column indices and we have $[1, 2, 3]$. Parallel elements to the main diagonal is $[a_{12}, a_{23}, a_{31}]$ which has the column indices $[2, 3, 1]$. Finally the diagonal column elements $[a_{13}, a_{21}, a_{32}]$ corresponds to the column indices $[3, 1, 2]$. We repeat the same process for secondary diagonal and its parallel diagonals. The resulted permutations which are extracted for both diagonals and its parallel diagonals are as below:

$[1, 2, 3] \longrightarrow [2, 3, 1] \longrightarrow [3, 1, 2]$ (main diagonal and its parallel diagonal)

$[3, 2, 1] \longrightarrow [1, 3, 2] \longrightarrow [2, 1, 3]$ (secondary diagonal and its parallel diagonal)

The circular pattern appears in the main diagonal and its parallel diagonal column indices. A similar pattern also appears in the secondary diagonal and its parallel diagonal column indices. This result can be rearranged as shown in Table 4.1:

Table 4.1: A pair of Main Diagonal and Secondary Diagonal Column Indices

Main diagonal and its parallel diagonal	Secondary diagonal and its parallel diagonal
$[1, 2, 3]$	$[3, 2, 1]$
$[2, 3, 1]$	$[1, 3, 2]$
$[3, 1, 2]$	$[2, 1, 3]$

From Table 4.1, it is shown that the secondary diagonal column indices is the reverse of the main diagonal column indices. It is also applicable to other parallel main diagonal to parallel secondary diagonal.

From the Sarrus Rule, the determinant is given by

$$[a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32}] - [a_{13}a_{22}a_{31} + a_{11}a_{23}a_{32} + a_{12}a_{21}a_{33}]. \qquad (4.1)$$

The rearrangement of the result in Equation 4.1 with respect to the circular permutation

of element column indices in pair of diagonals from Table 4.1 will give

$$[a_{11}a_{22}a_{33} - a_{13}a_{22}a_{31}] + [a_{12}a_{23}a_{31} - a_{11}a_{23}a_{32}] + [a_{13}a_{21}a_{32} - a_{12}a_{21}a_{33}]. \quad (4.2)$$

The third order diagram of the Sarrus Rule as shown below

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{23} & a_{21} & a_{22} \\ a_{31} & a_{32} & a_{33} & a_{31} & a_{32} \end{vmatrix}$$

can be further decomposed as

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} + \begin{vmatrix} a_{12} & a_{13} & a_{11} \\ a_{22} & a_{23} & a_{21} \\ a_{32} & a_{33} & a_{31} \end{vmatrix} + \begin{vmatrix} a_{13} & a_{11} & a_{12} \\ a_{23} & a_{21} & a_{22} \\ a_{33} & a_{31} & a_{32} \end{vmatrix}.$$

The MDP and SDP in the above matrices is equivalence to Equation 4.2. Extending this concept for any $n \times n$ matrix will give

$$MDP = sign(\sigma) \prod_{j=1}^{n} a_{j\sigma(j)}$$

$$SDP = sign(\sigma) \prod_{j=1}^{n} a_{j\sigma(n-j+1)}.$$

Now, we generalise that notation for circular process. For each k cycle, $k \in \{0, 1, 2, 3, ..., n-1\}$, and $i = (1, 2, ..., \frac{(n-1)!}{2})$, then Product Diagonal (PD) is a summation of MDP and its SDP for each k, i which equals to

$$PD(A_{i,k}) = (MDP+SDP)(A_{i,k}) = sign(\sigma)[\prod_{i=1}^{n} a_{(i)(\sigma(i)_k)}]+sign(\sigma)[\prod_{i=1}^{n} a_{(i)(\sigma(n+1-i)_k)}] \quad (4.3)$$

When $k = 0$; $PD(A_{j,0})$ is the PD of starter matrix of A_i.

Then using Equation 4.3, we can derive a division free method for finding determinant in the next section.

4.3 A Division Free Method Development for Finding Determinant

In this section, we construct a new approach for finding determinant by applying the permutations in Chapter Three for finding determinant. Our approach is best introduced by an example. Let $1 \leq i \leq \frac{(4-1)!}{2} = 3$ and consider matrix A of size 4×4 as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Step 1: Find the starter sets using circular operation.

$[1,2,3,4], [1,3,4,2], [1,4,2,3]$ as constructed in Chapter Three.

Step 2: Construct matrices based on each starter set from Step 1:

Starter set :[1,2,3,4]

$$A_1 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Starter set :[1,3,4,2]

$$A_2 = \begin{bmatrix} a_{11} & a_{13} & a_{14} & a_{12} \\ a_{21} & a_{23} & a_{24} & a_{22} \\ a_{31} & a_{33} & a_{34} & a_{32} \\ a_{41} & a_{43} & a_{44} & a_{42} \end{bmatrix}$$

Starter set :[1,4,2,3]

$$A_3 = \begin{bmatrix} a_{11} & a_{14} & a_{12} & a_{13} \\ a_{21} & a_{24} & a_{22} & a_{23} \\ a_{31} & a_{34} & a_{32} & a_{33} \\ a_{41} & a_{44} & a_{42} & a_{43} \end{bmatrix}$$

Step 3: Calculate the sum of sign diagonal product of each A_i.

Let i = 1 and $k = 0$

$$A_1 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

if $n \equiv 0$ or 1 mod 4, then the sign of its secondary diagonal is equal to its main diagonal sign.

$PD(A_{1,0})$= $(-1)^0 a_{11}a_{22}a_{33}a_{44} + a_{14}a_{23}a_{32}a_{41}$

Total PD = $PD(A_{1,0})$

Step 4: Employ circular operation on A_1 where all the columns of A_1 cycled. Then the matrix become $A_{1,1}$ and do Step 3.

$$[2, 3, 4, 1]$$

$$A_{1,1} = \begin{bmatrix} a_{12} & a_{13} & a_{14} & a_{11} \\ a_{22} & a_{23} & a_{24} & a_{21} \\ a_{32} & a_{33} & a_{34} & a_{31} \\ a_{42} & a_{43} & a_{44} & a_{41} \end{bmatrix}$$

$PD(A_{1,1})$= $(-1)^3 a_{12}a_{23}a_{34}a_{41} - a_{11}a_{24}a_{33}a_{42}$

Total PD = $PD(A_{1,0}) + PD(A_{1,1})$

Step 5: Repeat Step 4 and stop after k= 3.

$$[3, 4, 1, 2]$$

$$A_{1,2} = \begin{bmatrix} a_{13} & a_{14} & a_{11} & a_{12} \\ a_{23} & a_{24} & a_{21} & a_{22} \\ a_{33} & a_{34} & a_{31} & a_{32} \\ a_{43} & a_{44} & a_{41} & a_{42} \end{bmatrix}$$

$PD(A_{1,2})$= $(-1)^4 a_{13}a_{24}a_{31}a_{42} + a_{12}a_{21}a_{34}a_{43}$

Total PD = $PD(A_{1,0}) + PD(A_{1,1}) + PD(A_{1,2})$

$$[4, 1, 2, 3]$$

$$A_{1,3} = \begin{bmatrix} a_{14} & a_{11} & a_{12} & a_{13} \\ a_{24} & a_{21} & a_{22} & a_{23} \\ a_{34} & a_{31} & a_{32} & a_{33} \\ a_{44} & a_{41} & a_{42} & a_{43} \end{bmatrix}$$

$PD(A_{1,3})$= $(-1)^3 a_{14}a_{21}a_{32}a_{43} - a_{13}a_{22}a_{31}a_{44}$

Total PD = $PD(A_{1,0}) + PD(A_{1,1}) + PD(A_{1,2}) + PD(A_{1,3})$

Step 6: Go to Step 2 for next $i = 2$ and repeat Step 3, 4 and 5. Do until $i = 3$

$$A_2 = \begin{bmatrix} a_{11} & a_{13} & a_{14} & a_{12} \\ a_{21} & a_{23} & a_{24} & a_{22} \\ a_{31} & a_{33} & a_{34} & a_{32} \\ a_{41} & a_{43} & a_{44} & a_{42} \end{bmatrix}$$

$PD(A_{2,0})$= $(-1)^2 a_{11}a_{23}a_{34}a_{42} + a_{12}a_{24}a_{33}a_{41}$

Total PD = $[\sum_{k=0}^{3} PD(A_{1,k})] + PD(A_{2,0})$

$$[3, 4, 2, 1]$$

$$A_{2,1} = \begin{bmatrix} a_{13} & a_{14} & a_{12} & a_{11} \\ a_{23} & a_{24} & a_{22} & a_{21} \\ a_{33} & a_{34} & a_{32} & a_{31} \\ a_{43} & a_{44} & a_{42} & a_{41} \end{bmatrix}$$

$PD(A_{21})$= $(-1)^5 a_{13}a_{24}a_{32}a_{41} - a_{11}a_{22}a_{34}a_{43}$

Total PD = $[\sum_{k=0}^{3} PD(A_{1,k})] + PD(A_{2,0}) + PD(A_{2,1})$

$$[4, 2, 1, 3]$$

$$A_{2,2} = \begin{bmatrix} a_{14} & a_{12} & a_{11} & a_{13} \\ a_{24} & a_{22} & a_{21} & a_{23} \\ a_{34} & a_{32} & a_{31} & a_{33} \\ a_{44} & a_{42} & a_{41} & a_{43} \end{bmatrix}$$

$PD(A_{2,2})$= $(-1)^4 a_{14}a_{22}a_{31}a_{43} + a_{13}a_{21}a_{32}a_{44}$

Total PD = $[\sum_{k=0}^{3} PD(A_{1,k})] + PD(A_{2,0}) + PD(A_{2,1}) + PD(A_{2,2})$

$$[2, 1, 3, 4]$$

$$A_{2,3} = \begin{bmatrix} a_{12} & a_{11} & a_{13} & a_{14} \\ a_{22} & a_{21} & a_{23} & a_{24} \\ a_{32} & a_{31} & a_{33} & a_{34} \\ a_{42} & a_{41} & a_{43} & a_{44} \end{bmatrix}$$

$PD(A_{2,3})$= $(-1)^1 a_{12}a_{21}a_{33}a_{44} - a_{14}a_{23}a_{31}a_{42}$

Total PD = $[\sum_{k=0}^{3} PD(A_{1,k})] + PD(A_{2,0}) + PD(A_{2,1}) + PD(A_{2,2}) + PD(A_{2,3})$

$$A_3 = \begin{bmatrix} a_{11} & a_{14} & a_{12} & a_{13} \\ a_{21} & a_{24} & a_{22} & a_{23} \\ a_{31} & a_{34} & a_{32} & a_{33} \\ a_{41} & a_{44} & a_{42} & a_{43} \end{bmatrix}$$

$PD(A_{3,0})$= $(-1)^2 a_{11}a_{24}a_{32}a_{43} + a_{13}a_{22}a_{34}a_{41}$

Total PD = $[\sum_{k=0}^{3}(PD(A_{1,k}) + PD(A_{2,k}))] + PD(A_{3,0})$

$$[4, 2, 3, 1]$$

$$A_{3,1} = \begin{bmatrix} a_{14} & a_{12} & a_{13} & a_{11} \\ a_{24} & a_{22} & a_{23} & a_{21} \\ a_{34} & a_{32} & a_{33} & a_{31} \\ a_{44} & a_{42} & a_{43} & a_{41} \end{bmatrix}$$

$PD(A_{3,1})$= $(-1)^5 a_{14}a_{22}a_{33}a_{41} - a_{11}a_{23}a_{32}a_{44}$

Total PD = $[\sum_{k=0}^{3}(PD(A_{1,k}) + PD(A_{2,k}))] + PD(A_{3,0}) + PD(A_{3,1})$

$$[2, 3, 1, 4]$$

$$A_{3,2} = \begin{bmatrix} a_{12} & a_{13} & a_{11} & a_{14} \\ a_{22} & a_{23} & a_{21} & a_{24} \\ a_{32} & a_{33} & a_{31} & a_{34} \\ a_{42} & a_{43} & a_{41} & a_{44} \end{bmatrix}$$

$PD(A_{3,2})$= $(-1)^2 a_{12}a_{23}a_{31}a_{44} + a_{14}a_{21}a_{33}a_{42}$

Total PD = $[\sum_{k=0}^{3}(PD(A_{1,k})+PD(A_{2,k}))]+PD(A_{3,0})+PD(A_{3,1})+PD(A_{3,2})$

$$[3, 1, 4, 2]$$

$$A_{3,3} = \begin{bmatrix} a_{13} & a_{11} & a_{14} & a_{12} \\ a_{23} & a_{21} & a_{24} & a_{22} \\ a_{33} & a_{31} & a_{34} & a_{32} \\ a_{43} & a_{41} & a_{44} & a_{42} \end{bmatrix}$$

$PD(A_{3,3})$= $(-1)^3 a_{13}a_{21}a_{34}a_{42} - a_{12}a_{24}a_{31}a_{43}$

Total PD = $[\sum_{k=0}^{3}(PD(A_{1,k})+PD(A_{2,k}))]+PD(A_{3,0})+PD(A_{3,1})+PD(A_{3,2})+$

$PD(A_{3,3})$

Step 7: Calculate the $det(A)$

$$det(A) = \sum_{k=0}^{3}(PD(A_{1,k}) + PD(A_{2,k}) + PD(A_{3,k})).$$

It can be simplified as follows:

$$det(A) = \sum_{i=1}^{3}\sum_{k=0}^{3}(PD(A_{i,k})).$$

In terms of any n cases where n is the order of a square matrix:

$$det(A) = \sum_{i=1}^{\frac{(n-1)!}{2}} \sum_{k=0}^{n-1} (PD(A_{i,k})). \tag{4.4}$$

Representing $(PD(A_{i,k}))$ is as defined in Equation 4.3, we can rewrite Equation 4.4

$$det(A) = \sum_{i=1}^{\frac{(n-1)!}{2}} (\sum_{k=0}^{n-1} sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(j)_k)}] + sign(\sigma)[\prod_{j=1}^{n} a_{(i)(\sigma(n+1-j)_k)}]). \tag{4.5}$$

Equation 4.5 represents as a division free formula for the generalised Sarrus Rule (cross multiplication method). The total of diagonal products from cycle $k = 0$ until $n - 1$ for each starter matrix is $2n$. As the result the total of all diagonal products is $\frac{(n-1)!}{2} \times 2n = n!$.

The sequential steps of the proposed method can be explained in mathematical expression where $A_{i,k}$, $1 \leq i \leq \frac{(n-1)!}{2}$ and $0 \leq k \leq n-1$ were constructed can be condensed to $\frac{(n-1)!}{2}$ of nth order diagrams.

Let consider $n = 4$. By using the result in the Step 1 (page 77), $[1, 2, 3, 4]$, $[1, 3, 4, 2]$,$[1, 4, 3, 2]$ are defined as the starter sets, and also represented the column indices for $n = 4$. Based on each starter set, the starter set matrix is generated. Then fourth order diagram is developed by appending the first three columns to the right of the starter matrix. The resulted three types of fourth order diagram as below:

Starter set: $[1, 2, 3, 4]$

$$|A_1| = \begin{vmatrix} a_{11} & a_{12} & a_{13} & a_{14} & |a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} & a_{24} & |a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} & a_{34} & |a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} & a_{44} & |a_{41} & a_{42} & a_{43} \end{vmatrix}$$

which condensed from $|A_{1,0}|+ |A_{1,1}| +|A_{1,2}|+|A_{1,3}|$.

Starter set :$[1, 3, 4, 2]$

$$|A_2| = \begin{vmatrix} a_{11} & a_{13} & a_{14} & a_{12} & |a_{11} & a_{13} & a_{14} \\ a_{21} & a_{23} & a_{24} & a_{22} & |a_{21} & a_{23} & a_{24} \\ a_{31} & a_{33} & a_{34} & a_{32} & |a_{31} & a_{33} & a_{34} \\ a_{41} & a_{43} & a_{44} & a_{42} & |a_{41} & a_{43} & a_{44} \end{vmatrix}$$

which condensed from $|A_{2,0}|+ |A_{2,1}| +|A_{2,2}|+|A_{2,3}|$.

Starter set : $[1, 4, 2, 3]$

$$|A_3| = \begin{vmatrix} a_{11} & a_{14} & a_{12} & a_{13} & |a_{11} & a_{14} & a_{12} \\ a_{21} & a_{24} & a_{22} & a_{23} & |a_{21} & a_{24} & a_{22} \\ a_{31} & a_{34} & a_{32} & a_{33} & |a_{31} & a_{34} & a_{32} \\ a_{41} & a_{44} & a_{42} & a_{43} & |a_{41} & a_{44} & a_{42} \end{vmatrix}$$

which condensed from $|A_{3,0}|+ |A_{3,1}| +|A_{3,2}|+|A_{3,3}|$.

Furthermore in spite of appending the first three columns to the right of the starter matrix, we can also append the last three columns to the left of the starter matrix as follows:

Starter set: $[1, 2, 3, 4]$

$$|A_1| = \begin{vmatrix} a_{12} & a_{13} & a_{14} & |a_{11} & a_{12} & a_{13} & a_{14} \\ a_{22} & a_{23} & a_{24} & |a_{21} & a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} & |a_{31} & a_{32} & a_{33} & a_{34} \\ a_{42} & a_{43} & a_{44} & |a_{41} & a_{42} & a_{43} & a_{44} \end{vmatrix}$$

which condensed from $|A_{1,1}|+ |A_{1,2}| +|A_{1,3}|+|A_{1,0}|$.

Starter set :$[1, 3, 4, 2]$

$$|A_2| = \begin{vmatrix} a_{13} & a_{14} & a_{12} & |a_{11} & a_{13} & a_{14} & a_{12} \\ a_{23} & a_{24} & a_{22} & |a_{21} & a_{23} & a_{24} & a_{22} \\ a_{33} & a_{34} & a_{32} & |a_{31} & a_{33} & a_{34} & a_{32} \\ a_{43} & a_{44} & a_{42} & |a_{41} & a_{43} & a_{44} & a_{42} \end{vmatrix}$$

which condensed from $|A_{2,1}|$+ $|A_{2,2}|$ +$|A_{2,3}|$+$|A_{2,0}|$.

Starter set : $[1, 4, 2, 3]$

$$|A_3| = \begin{vmatrix} a_{14} & a_{12} & a_{13} & |a_{11} & a_{14} & a_{12} & a_{13} \\ a_{24} & a_{22} & a_{23} & |a_{21} & a_{24} & a_{22} & a_{23} \\ a_{34} & a_{32} & a_{33} & |a_{31} & a_{34} & a_{32} & a_{33} \\ a_{44} & a_{42} & a_{43} & |a_{41} & a_{44} & a_{42} & a_{43} \end{vmatrix}$$

which condensed from $|A_{3,1}|$+ $|A_{3,2}|$ +$|A_{3,3}|$+$|A_{3,0}|$.

The process of determining the sign of each main diagonal and its secondary diagonal for each nth order diagram which corresponds to Sarrus Rule can be demonstrated by the following example:

Example 4.3.1.

Given a starter set $= [1, 3, 4, 2]$ and 4th order diagram is constructed from starter sets $[1, 3, 4, 2]$ is as follows:

$$|A_2| = \begin{vmatrix} a_{11} & a_{13} & a_{14} & a_{12} & |a_{11} & a_{13} & a_{14} \\ a_{21} & a_{23} & a_{24} & a_{22} & |a_{21} & a_{23} & a_{24} \\ a_{31} & a_{33} & a_{34} & a_{32} & |a_{31} & a_{33} & a_{34} \\ a_{41} & a_{43} & a_{44} & a_{42} & |a_{41} & a_{43} & a_{44} \end{vmatrix}$$

The sign of the term of the main diagonal, $a_{11}a_{23}a_{34}a_{42}$ is $(+1)$ with total inversion =2. A set of its parallel term with its column indices:

$$a_{13}a_{24}a_{32}a_{41} \text{ with column indices } [3,4,2,1]$$
$$a_{14}a_{22}a_{31}a_{43} \text{ with column indices } [4,2,1,3]$$
$$a_{12}a_{21}a_{33}a_{44} \text{ with column indices } [2,1,3,4]$$

In spite of using total inversion for finding the sign of the product, there is an alternative way to find the sign of the product. The following properties are derived based on even/odd number of n.

(i) If n is even, then the sign of the set parallel diagonal product is alternate between positive and negative which depends on the sign of the main diagonal product.

(ii) If n is odd, then the sign of the set parallel diagonal product is the same with the sign of the main diagonal product.

Hence, by followed the property (i) where $n = 4$ is even, the sign of terms as below:

$$(+1)a_{11}a_{23}a_{34}a_{42}$$
$$(-1)a_{13}a_{24}a_{32}a_{41}$$
$$(+1)a_{14}a_{22}a_{31}a_{43}$$
$$(-1)a_{12}a_{21}a_{33}a_{44}$$

This condition is also valid for the secondary diagonal product and its parallel diagonal product.

However in order to find the sign of the secondary diagonal, the property is derived as follows:

$$\text{sign of } SDP = \begin{cases} \text{sign of } MDP & \text{if } n \equiv 0 \text{ or } 1 \pmod 4 \\ (-1) \cdot (\text{sign of } MDP) & \text{otherwise} \end{cases}$$

For this case $4 \equiv 0 \bmod 4$, then its SDP sign is the same to its MDP sign as follows:

$$(+1)a_{12}a_{24}a_{33}a_{41}$$

$$(-1)a_{11}a_{22}a_{34}a_{43}$$

$$(+1)a_{13}a_{21}a_{32}a_{44}$$

$$(-1)a_{14}a_{23}a_{31}a_{42}$$

$$|A_2| = \begin{vmatrix} +1 & -1 & +1 & -1(+1) & (-1) & (+1) & (-1) \\ a_{11} & a_{13} & a_{14} & a_{12} & |a_{11} & a_{13} & a_{14} \\ a_{21} & a_{23} & a_{24} & a_{22} & |a_{21} & a_{23} & a_{24} \\ a_{31} & a_{33} & a_{34} & a_{32} & |a_{31} & a_{33} & a_{34} \\ a_{41} & a_{43} & a_{44} & a_{42} & |a_{41} & a_{43} & a_{44} \end{vmatrix}$$

Remark 4.3.2. *The symbol () is used for secondary diagonal and its parallel diagonal.*

The following example for n is odd where $n = 5$ is given .

Example 4.3.3.

Given a starter set:$[1, 3, 4, 2, 5]$. The 5th order diagram is constructed from starter sets $[1, 3, 4, 2, 5]$ is as follows:

$$|A_2| = \begin{vmatrix} a_{11} & a_{13} & a_{14} & a_{12} & a_{15} & |a_{11} & a_{13} & a_{14} & a_{12} \\ a_{21} & a_{23} & a_{24} & a_{22} & a_{25} & |a_{21} & a_{23} & a_{24} & a_{22} \\ a_{31} & a_{33} & a_{34} & a_{32} & a_{35} & |a_{31} & a_{33} & a_{34} & a_{32} \\ a_{41} & a_{43} & a_{44} & a_{42} & a_{45} & |a_{41} & a_{43} & a_{44} & a_{42} \\ a_{51} & a_{53} & a_{54} & a_{52} & a_{55} & |a_{51} & a_{53} & a_{54} & a_{52} \end{vmatrix}$$

The sign of the term of the main diagonal, $a_{11}a_{23}a_{34}a_{42}a_{55}$ is (+1) with total inversion =2. Now we shall see a set of its parallel term with its sign:

$$(+1)a_{11}a_{23}a_{34}a_{42}a_{55}$$

$$(+1)a_{13}a_{24}a_{32}a_{45}a_{51}$$

$$(+1)a_{14}a_{22}a_{35}a_{41}a_{53}$$

$$(+1)a_{12}a_{25}a_{31}a_{43}a_{54}$$

$$(+1)a_{15}a_{21}a_{33}a_{44}a_{52}$$

This condition is also valid for the secondary diagonal product and its parallel diagonal product.

If $n \equiv 0$ or $1 \bmod 4$, its secondary diagonal sign is the same with its main diagonal sign. So the sign of the secondary diagonal is (+1).

$$(+1)a_{15}a_{22}a_{34}a_{43}a_{51}$$

$$(+1)a_{11}a_{25}a_{32}a_{44}a_{53}$$

$$(+1)a_{13}a_{21}a_{35}a_{42}a_{54}$$

$$(+1)a_{14}a_{23}a_{31}a_{45}a_{52}$$

$$(+1)a_{12}a_{24}a_{33}a_{41}a_{55}$$

$$|A_2| = \begin{vmatrix} +1 & +1 & +1 & +1 & +1(+1) & (+1) & (+1) & (+1) & (+1) \\ a_{11} & a_{13} & a_{14} & a_{12} & a_{15} & |a_{11} & a_{13} & a_{14} & a_{12} \\ a_{21} & a_{23} & a_{24} & a_{22} & a_{25} & |a_{21} & a_{23} & a_{24} & a_{22} \\ a_{31} & a_{33} & a_{34} & a_{32} & a_{35} & |a_{31} & a_{33} & a_{34} & a_{32} \\ a_{41} & a_{43} & a_{44} & a_{42} & a_{45} & |a_{41} & a_{43} & a_{44} & a_{42} \\ a_{51} & a_{53} & a_{54} & a_{52} & a_{55} & |a_{51} & a_{53} & a_{54} & a_{52} \end{vmatrix}$$

The next example demonstrate for determining the determinant which follow the Sarrus Rule for order of matrix $n = 4$.

Example 4.3.4.

Let matrix A with size 4×4 defined as follows:

$$A = \begin{bmatrix} 1 & 4 & 6 & 8 \\ 7 & 10 & -5 & 5 \\ 8 & 1 & 5 & 11 \\ 4 & 3 & 7 & 9 \end{bmatrix}$$

The three fourth order diagrams are developed from the three generated starter matrices with its starter sets: S_1, S_2, S_3. All starter sets are even permutations where all main diagonals of all the fourth order diagrams have positive sign.

$$S_1 = [1, 2, 3, 4]$$

$$|A_1| = \begin{vmatrix} +1 & -1 & +1 & -1(+1) & (-1) & (+1) & (-1) \\ 1 & 4 & 6 & 8 & 1 & 4 & 6 \\ 7 & 10 & -5 & 5 & 7 & 10 & -5 \\ 8 & 1 & 5 & 11 & 8 & 1 & 5 \\ 4 & 3 & 7 & 9 & 4 & 3 & 7 \end{vmatrix}$$

$$\begin{aligned} |A_1| &= ([1 \times 10 \times 5 \times 9] - [4 \times (-5) \times 11 \times 4] + [6 \times 5 \times 8 \times 3] - [8 \times 7 \times 1 \times 7]) \\ &+ ([8 \times (-5) \times 1 \times 4] - [1 \times 5 \times 5 \times 3] + [4 \times 7 \times 11 \times 7] - [6 \times 10 \times 8 \times 9]) \\ &= (450 + 880 + 720 - 392) + (-160 - 75 + 2156 - 4320) \\ &= -741 \end{aligned}$$

$$S_2 = [1, 3, 4, 2]$$

$$|A_2| = \begin{vmatrix} +1 & -1 & +1 & -1(+1) & (-1) & (+1) & (-1) \\ 1 & 6 & 8 & 4 & 1 & 6 & 8 \\ 7 & -5 & 5 & 10 & 7 & -5 & 5 \\ 8 & 5 & 11 & 1 & 8 & 5 & 11 \\ 4 & 7 & 9 & 3 & 4 & 7 & 9 \end{vmatrix}$$

$$\begin{aligned} |A_2| &= ([1 \times (-5) \times 11 \times 3] - [6 \times 5 \times 1 \times 4] + [8 \times 10 \times 8 \times 7] - [4 \times 7 \times 5 \times 9]) \\ &+ ([4 \times 5 \times 5 \times 4] - [1 \times 10 \times 11 \times 7] + [6 \times 7 \times 1 \times 9] - [8 \times (-5) \times 8 \times 3]) \\ &= (-165 - 120 + 4480 - 1260) + (400 - 770 + 378 + 960) \\ &= 3903 \end{aligned}$$

$$S_3 = [1, 4, 2, 3]$$

$$|A_3| = \begin{vmatrix} +1 & -1 & +1 & -1(+1) & (-1) & (+1) & (-1) \\ 1 & 8 & 4 & 6 & 1 & 8 & 4 \\ 7 & 5 & 10 & -5 & 7 & 5 & 10 \\ 8 & 11 & 1 & 5 & 8 & 11 & 1 \\ 4 & 9 & 3 & 7 & 4 & 9 & 3 \end{vmatrix}$$

$$\begin{aligned} |A_3| &= ([1 \times 5 \times 1 \times 7] - [8 \times 10 \times 5 \times 4] + [4 \times (-5) \times 8 \times 9] - [6 \times 7 \times 11 \times 3]) \\ &\quad + ([6 \times 10 \times 11 \times 4] - [1 \times (-5) \times 1 \times 9] + [8 \times 7 \times 5 \times 3] - [4 \times 5 \times 8 \times 7]) \\ &= (35 - 1600 - 1440 - 1386) + (2640 + 45 + 840 - 1120) \\ &= -1986 \end{aligned}$$

Thus, the determinant of matrix A:

$$\begin{aligned} det(A) &= |A_1| + |A_2| + |A_3| \\ &= 1176 \end{aligned}$$

The general algorithm for finding the determinant using permutation will be discussed in the next section.

4.4 General Algorithm for Finding Determinant Using Permutation

The division free algorithm for determining the determinant of $n \times n$ matrix by using our permutation algorithm is described as follows:

Step 1: Generate the starter sets and denote them by $i = 1, 2, .., \frac{(n-1)!}{2}$.

Step 2: Generate matrix based on each starter sets where starting with starter set $i = 1$ and $k = 0$.

Step 3: Find product of element in the main diagonal and secondary diagonal. Sum up

both of them. Simultaneously calculate the sign of each diagonal where if $n \equiv 0$ or 1 mod 4, then secondary diagonal sign is the same with its main diagonal sign and if $n \equiv 2$ or 3 mod 4 its secondary diagonal sign is (-1) is multiply with its main diagonal sign .

Step 4: Employ circular operation on matrix in Step 2 and do Step 3.

Step 5: Repeat Step 4 until $k = n - 1$.

Step 6: Go to Step 2 for next $i = 2$ and repeat Step 3, 4 and 5. Stop after $i = \frac{(n-1)!}{2}$.

Step 7: Total up of PD = $det(A)$.

The following pseudocode of the algorithm is also given.

Algorithm 4.1 PERMUTDET1

PERMUTDET1(temp)
if $temp = 2$ **then**
  for $i = 1$ to n **do**
    old = num[i]
    for $k = i$ to $n - 1$ **do**
      num[k] = num[k+1]
    end for
    num[n] = old
    calculate $PD_i = SDP_i + MDP_i$
    find the sign of $PD[i]$
    $\sum_{i=1}^{n} PD_i$
  end for
  $det(A) = \sum_{j=1}^{\frac{(n-1)!}{2}} [\sum_{i=1}^{n} PD_i]_j$
  return
end if
$temp = temp - 1$
for $i = n$ to $temp$ **do**
  old = num[i]
  for $k = i$ to $n - 1$ **do**
    num[k] = num[k+1]
  end for
  num[n] = old
  call PERMUTDET1(temp)
end for

For the exchange operation, the process is similar to PERMUTDET1 algorithm. Refer to the following algorithm.

Algorithm 4.2 PERMUTDET2

```
PERMUTDET2(temp)
if temp = 2 then
    for i = 1 to n do
        old = num[i]
        for k = 1 to n − 1 do
            num[k] = num[k+1]
        end for
        num[n] = old
        calculate PD_i = SDP_i + MDP_i
        find the sign of PDi
        Σ_{i=1}^{n} PD_i
    end for
    det(A) = Σ_{j=1}^{(n−1)!/2} [Σ_{i=1}^{n} PD_i]_j
    return
end if
temp = temp − 1
for i = temp to n do
    if i! = n then
        old = num[i]
        num[i]=num[i+1]
        num[i+1] = old
    else
        old = num[i]
        for k = 1 to n − 1 do
            num[k] = num[k+1]
        end for
        num[n] = old
    end if
    call PERMUTDET2(temp)
end for
```

The procedure of the generalised Sarrus Rule is summarised as follows:

(i) Find $\frac{(n-1)!}{2}$ of $n \times n$ matrices which are to be generated from $\frac{(n-1)!}{2}$ starter sets.

(ii) Rewrite the first $n-1$ columns to the right to all of $\frac{(n-1)!}{2}$ matrices obtained which are called as $\frac{(n-1)!}{2}$ of nth order diagram, $|A_i|$ (Equation 4.6) where where $1 \leq i \leq \frac{(n-1)!}{2}$ and carry out the multiplication of elements along the traced arrows and total up for each nth order diagrams (Equation 4.6).

Given any starter set: $[1, 3, 2, ..., n-1, n]$.

$$|A_i| = \begin{vmatrix} a_{11} & a_{13} & \cdots & a_{1(n-1)} & a_{1n} & |a_{11} & a_{13} & \cdots & a_{1(n-1)} \\ a_{21} & a_{23} & \cdots & a_{2(n-1)} & a_{2)n} & |a_{21} & a_{23} & \cdots & a_{2(n-1)} \\ a_{31} & a_{33} & \cdots & a_{3(n-1)} & a_{3n} & |a_{31} & a_{33} & \cdots & a_{3(n-1)} \\ a_{41} & a_{43} & \cdots & a_{4(n-1)} & a_{4n} & |a_{41} & a_{43} & \cdots & a_{4(n-1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & |\vdots & \vdots & \ddots & \vdots \\ a_{n-1)1} & a_{n-1)3} & \cdots & a_{(n-1)(n-1)} & a_{(n-1)n} & |a_{n-1)1} & a_{n-1)3} & \cdots & a_{(n-1)(n-1)} \\ a_{n1} & a_{n3} & \cdots & a_{n(n-1)}) & a_{nn} & |a_{n1} & a_{n3} & \cdots & a_{n(n-1)} \end{vmatrix} \tag{4.6}$$

(iii) Find the sign of the main diagonal of each $\frac{(n-1)!}{2}$ of $n \times n$ matrices and the sign of other products, two properties were set up, firstly (a) for sign of the parallel diagonal product , and secondly (b) for the secondary diagonal :

(a(i)) If n is even, then the sign of the set parallel diagonal product is alternate between positive and negative which depended on the sign of the main diagonal product.

(a(ii))If n is odd, the sign of the set parallel diagonal product is the same to the sign of the main diagonal product.

(b)

$$\text{sign of } SDP = \begin{cases} \text{sign of } MDP & \text{if } n \equiv 0 \text{ or } 1 \pmod 4 \\ (-1) \cdot (\text{sign of } MDP) & \text{otherwise} \end{cases}$$

The new theoretical work will be discussed in next section.

4.5 Theoretical Results

An even and odd permutation is important with respect to its diagonal product while determining the determinant of a square matrix.

Lemma 4.5.1. *The total number of inversion for the permutation is $k-1$ if the last k number of elements of identity permutation are performs by circular operation where $0 < k < n$.*

Proof. Let $S = [1, 2, 3, ..., n-2, n-1, n]$ be the identity permutation of n distinct elements.

Perform circular operation over the last two elements on identity permutation gives

$[1, 2, 3, ..., n-2, n, n-1]$ and the total inversion is one i.e. $2-1$ where one element: $n > n-1$.

Next, perform the circular operation over the last three elements on succeeding permutation, gives

$[1, 2, 3, ..., n-1, n, n-2]$ and the total inversions = 2 where two elements: $n-1 > n-2$ and $n > n-2$.

Suppose the total inversion of permutation is k if the circular process is performed over the last $k+1$ elements on identity permutation

Permutation $[1, 2, 3, ...k, k+2, k+3, ..., n-3, n-2, n-1, n, k+1]$ with total inversion k need to be proven.

Then the total inversion is $(k+1)-1 = k$.

The proof is shown and it is also true when we select the first k elements. □

Lemma 4.5.2. *The total number of nth order diagram is* $\frac{(n-1)!}{2}$.

Proof. By Lemma 3.4.2 and 3.2.3, the total number of starter sets is $\frac{(n-1)!}{2}$. Then by definition 4.2.5, nth order diagram is a diagram generated from starter sets matrix of order n by appending the first $(n-1)$ columns to the right of origin generated starter set matrix. Therefore the total number of nth order diagram is $\frac{(n-1)!}{2}$. □

Theorem 4.5.3. *There are n number of n order of matrix are decomposed from nth order diagram.*

Proof. By Definition 4.2.6, the size of nth order diagram is $n \times (2n-1)$ where the total column is $(2n-1)$. The process in determining of first n order matrix is begin by selecting the first n columns of nth order diagram. Then, the next second n order matrix is derived by selecting second column until $(n+1)$th column. Next, third n order matrix is derived by selecting third column until $(n+2)$th column. This process continues until nth n order matrix is derived by selecting the last nth column until $(2n-1)$th column. Thus, there are n number of n order of matrices derived from nth order diagram. □

Theorem 4.5.4. *The value of nth order diagrams, A_i as follows*

$$det(A_i) = \sum_{k=0}^{n-1} sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(j)_k)}] + sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(n+1-j)_k)}].$$

Proof. From Theorem 4.5.3, there are n number of n order of matrices $A_{i,k}$, $0 \leq k \leq n-1$ derived from nth order diagram. By Definition 4.2.1 and 4.2.2, the total diagonal product for each $A_{i,k}$ is given by

$$PD(A_{i,k}) = (MDP+SDP)(A_{i,k}) = sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(j)_k)}]+sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(n+1-j)_k)}].$$

Then

$$det(A_i) = \sum_{k=0}^{n-1} PD(A_{i,k})) = (\sum_{k=0}^{n-1} sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(j)_k)}] + sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(n+1-j)_k)}].$$

□

Theorem 4.5.5. *The number of even and odd starter sets under recursive circular operation for* $n \geq 5$ *is* $\frac{(n-1)!}{4}$.

Proof. From Theorem 2.2.8, we have $\frac{n!}{2}$ permutation of two classes(even and odd permutation). From Lemma 3.4.2, the total number of starter set under circular operation is $\frac{(n-1)!}{2}$ for $n \geq 3$. Then the number of even and odd starter sets is $\frac{(n-1)!}{4}$. It is only valid for $n \geq 5$. For case $n = 3$ and 4, it is not valid because $\frac{(3-1)!}{4}$ and $\frac{(4-1)!}{4}$ are not evenly divided.

□

Theorem 4.5.6. *The number of the even and odd starter set under recursive exchange operation for* $n \geq 5$ *is* $\frac{(n-1)!}{4}$.

Proof. From Theorem 2.2.8, we have $\frac{n!}{2}$ permutations of the two classes (even and odd permutation). From Lemma 3.4.3, the total number of starter set under exchange operation is $\frac{(n-1)!}{2}$ for $n \geq 3$. Then the number of even and odd starter sets is $\frac{(n-1)!}{4}$. It only true for $n \geq 5$. For case $n = 3$ and 4, it is not valid because $\frac{(3-1)!}{4}$ and $\frac{(4-1)!}{4}$ are not evenly divided. □

Theorem 4.5.7.

$$\textit{sign of } SDP = \begin{cases} \textit{sign of } MDP & \textit{if } n \equiv 0 \textit{ or } 1 \textit{ (mod } 4\textit{)} \\ (-1) \cdot (\textit{sign of } MDP) & \textit{otherwise} \end{cases}$$

Proof. For $n = 2$ is a trivial case because it only has two permutations i.e. $[1, 2]$ is an even permutation and $[2, 1]$ is an odd permutation as reverse of $[1, 2]$.

Without loss of generality, it is enough to prove that the theorem based on identity permutation as main diagonal and its reverse as the secondary diagonal which is based on Lemma 2.2.9 where the maximum number of inversion is $\frac{n(n-1)}{2}$ with respect to the number of inversion for reverse permutation of identity permutation i.e. $\sum_{i=0}^{n-1} i =$

$\frac{n(n-1)}{2}$. As we know, the identity permutation is an even permutation with a number of inversion equals to zero.

Case : $n \equiv 0$ or 1 mod 4.

(i) $n \equiv 0$ mod 4 where n is a multiply of 4.

Let $n = 4k$ where $k \in Z^+$. Then the number of inversion for the secondary diagonal becomes

$$\frac{n(n-1)}{2} = \frac{4k(4k-1)}{2} = 2k(4k-1)$$

which is an even number. Thus its secondary diagonal has the same sign with its main diagonal.

(ii) $n \equiv 1$ mod 4.

Let $n = 4k + 1$ where $k \in Z^+$. Then the number of inversion for the secondary diagonal becomes

$$\frac{n(n-1)}{2} = \frac{(4k+1)(4k+1-1)}{2} = 2k(4k+1)$$

which is an even number and thus same sign with its main diagonal.

Case: $n \equiv 2$ or 3 mod 4.

(i) $n \equiv 2$ mod 4.

Let $n = 4k + 2$ where $k \in Z^+$. Then the number of inversion for the secondary diagonal becomes

$$\frac{n(n-1)}{2} = \frac{(4k+2)(4k+2-1)}{2} = (2k+1)(4k+1)$$

which is an odd number. Thus its secondary diagonal has (-1) multiply to the sign of its main diagonal.

(ii) $n \equiv 3 \bmod 4$.

Let $n = 4k + 3$ where $k \in Z^+$. Then the number of inversion for the secondary diagonal becomes

$$\frac{n(n-1)}{2} = \frac{(4k+3)(4k+3-1)}{2} = (4k+3)(2k+1)$$

which is also an odd number. Therefore its secondary diagonal has (-1) multiply to sign of its main diagonal.

□

Theorem 4.5.8. *The determinant of any square matrix A via Sarrus Rule is*

$$det(A) = \sum_{i=1}^{\frac{(n-1)!}{2}} \sum_{k=0}^{n-1} sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(j)_k)}] + sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(n+1-j)_k)}].$$

Proof. From Lemma 4.5.2, there are $\frac{(n-1)!}{2}$ number of distinct nth order diagram which generated from $\frac{(n-1)!}{2}$ starter sets matrices, A_i where $i \leq i \leq \frac{(n-1)!}{2}$. For every A_i,

$$det(A_i) = \sum_{k=0}^{n-1} sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(j)_k)}] + sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(n+1-j)_k)}].$$

where follow to Definition 4.2.1 and 4.2.2. The sign of each terms follow Equation 2.3. Then total all n order diagrams determinant, $det(A_i)$:

$$det(A) = \sum_{i=1}^{\frac{(n-1)!}{2}} det(A_i) = \sum_{i=1}^{\frac{(n-1)!}{2}} \sum_{k=0}^{n-1} sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(j)_k)}] + sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(n+1-j)_k)}].$$

□

Numerical results is presented in following section.

4.6 Numerical Results of Division Free Algorithms

Since our algorithm employs permutation methods (PERMUT1, PERMUT2, PERMUTIT3) for finding the determinant which corresponds to cross multiplication method rule, it is sufficient to compare it with the existing division free algorithms to determine the performance of the new algorithms. The existing division free algorithms are the cofactor expansion, Langdon (1967), and Thongchiew (2007). The new algorithms are PERMUTDET1 (recursive circular operation), PERMUTDET2 (recursive exchange operation) and PERMUTDETIT3 (iterative circular operation).

The results are given and represented in computation time (in seconds). All programs were tested on the HP Computer with Intel Xeon E5504 2.0 GHz processor and 4.00 GB Random Access Memory (RAM). We also tested the same matrices with other mathematical software i.e. Mathematical Laboratory (Matlab) to check the result.

Table 4.2: The Computation Time of New Sequential Determinant Algorithm (in seconds)

n	PERMUTDET1	PERMUTDET2	PERMUTDETIT3
7	0.001469	0.001397	0.001509
8	0.012187	0.012070	0.012419
9	0.125879	0.124636	0.128921
10	1.458077	1.456752	1.482345
11	18.2335528	18.177466	18.562527
12	248.345662	247.785887	251.013982
13	3637.202.73	3634.131510	3750.651724
14	57583796.117	57447.850780	58015.462781

As shown in Table 4.2, the computation times of the three new algorithms indicate that PERMUTDETIT3 is the slowest compared to recursive algorithms, PERMUTDET1 and PERMUTDET2. For both recursive programs, the computation times algorithm of the exchange operation (PERMUTDET2) is faster than the circular operation (PERMUTDET1).

Table 4.3: The Computation Time Among Sequential Determinant Algorithm based on Permutation (in seconds)

n	PERMUTDET1 (circular operation recursive)	PERMUTDET2 (exchange operation recursive)	PERMUTDETIT3 (circular operation iteration)	Thongchiew (iteration)	Langdon (circular iteration)
7	0.001479	0.001397	0.001509	0.002717	0.002757
8	0.012070	0.012070	0.012419	0.022498	0.022875
9	0.125564	0.124636	0.128921	0.228657	0.244429
10	1.455614	1.456752	1.482345	2.600765	2.870651
11	18.308862	18.177466	18.562527	32.450887	36.571486
12	248.345662	247.785887	251.013982	442.093281	505.857066
13	3637.202.73	3634.131510	3750.651724	6498.272820	7533.040560
14	57583.117796	57447.850780	58015.462781	110582.376448	118163.653816

Table 4.3 shows the results of comparison over time computation among permutation methods which have been applied for determining the determinant. As can be seen, the execution time grows dramatically when the size of the matrix increases, and our new algorithms generate lesser time than permutation program Langdon (1967) and Thongchiew (2007). The results indicate that the new algorithms are better in term of computation times than these three division free algorithms for finding the determinant.

Two advantages of this new algorithm compared to Langdon (1967) and Thongchiew (2007) algorithm have been identified. First, the latter algorithms generated all $n!$ permutation whereas our algorithm generated only $\frac{n!}{2}$ permutations. The next $\frac{n!}{2}$ permutations are generated by reversing the order of permutation of the first $\frac{n!}{2}$ permutations. Second, we only calculate the sign of the main diagonal column indices and the sign of the secondary diagonal column indices depends on to the sign of main diagonal column indices, whereas for Langdon (1967) and Thongchiew (2007) algorithms, the sign of each $n!$ product terms is computed. These two advantages contribute to lesser computation times for the new algorithm.

The result shown in Table 4.4 is the comparison of the three new algorithms to cofactor expansion.

Table 4.4: The Comparison of Computation Time of New Algorithms to Cofactor Expansion (in seconds)

n	PERMUTDET1	PERMUTDET2	PERMUTDETIT3	Cofactor expansion
7	0.001469	0.001397	0.001509	0.003914
8	0.012070	0.012070	0.012419	0.033226
9	0.125564	0.124636	0.128921	0.381172
10	1.455614	1.456752	1.482345	3.872222
11	18.308862	18.177466	18.562527	48.079466
12	248.345662	247.785887	251.013982	653.706605
13	3637.202.73	3634.131510	3750.651724	9816.224
14	57583.117796	57447.850780	58015.462781	172279.667287

As shown in Table 4.4 , the cofactor expansion algorithm performed slower than the new algorithms and also permutation algorithms i.e. Langdon (1967) and Thongchiew (2007).

In order to verify the correctness of the new algorithms, the determinant results are given for $n = 7, 8, 9, 10$ in Table 4.5. The detail of the result for new programs are given in **Appendix E**. The example output is given for $n = 5$.

Table 4.5: The Determinant Result from New Algorithms

n	Matrix of A	Det(A) of PERMUTDET1, PERMUTDET2 and PERMUTDETIT3
7	$\begin{bmatrix} 6 & 1 & 6 & 5 & 3 & 2 & 5 \\ 0 & 5 & 6 & 0 & 5 & 6 & 6 \\ 0 & 1 & 6 & 0 & 4 & 4 & 2 \\ 2 & 3 & 6 & 5 & 6 & 5 & 5 \\ 6 & 1 & 1 & 5 & 1 & 2 & 0 \\ 4 & 5 & 0 & 4 & 3 & 3 & 4 \\ 4 & 2 & 5 & 2 & 0 & 4 & 1 \end{bmatrix}$	5088.000000
8	$\begin{bmatrix} 1 & 3 & 6 & 4 & 1 & 4 & 6 & 6 \\ 2 & 0 & 1 & 1 & 1 & 3 & 1 & 3 \\ 3 & 6 & 3 & 4 & 7 & 4 & 6 & 1 \\ 4 & 6 & 5 & 4 & 6 & 7 & 7 & 6 \\ 3 & 2 & 5 & 0 & 3 & 3 & 3 & 6 \\ 7 & 3 & 2 & 5 & 1 & 0 & 5 & 7 \\ 5 & 4 & 3 & 4 & 6 & 5 & 5 & 3 \\ 3 & 5 & 1 & 2 & 4 & 3 & 6 & 2 \end{bmatrix}$	9166.000000
9	$\begin{bmatrix} 5 & 8 & 7 & 4 & 8 & 1 & 3 & 0 & 7 \\ 2 & 8 & 2 & 7 & 6 & 7 & 5 & 7 & 8 \\ 3 & 0 & 0 & 6 & 5 & 0 & 4 & 7 & 6 \\ 5 & 8 & 5 & 2 & 0 & 2 & 0 & 6 & 4 \\ 8 & 1 & 7 & 3 & 2 & 6 & 2 & 3 & 6 \\ 2 & 3 & 7 & 2 & 1 & 5 & 5 & 1 & 6 \\ 3 & 7 & 2 & 3 & 7 & 4 & 4 & 2 & 5 \\ 6 & 0 & 1 & 4 & 5 & 4 & 4 & 5 & 0 \\ 5 & 6 & 7 & 5 & 3 & 6 & 3 & 2 & 0 \end{bmatrix}$	-2699924.000000
10	$\begin{bmatrix} 1 & 7 & 4 & 0 & 9 & 4 & 8 & 8 & 2 & 4 \\ 5 & 5 & 1 & 7 & 1 & 1 & 5 & 2 & 7 & 6 \\ 1 & 4 & 2 & 3 & 2 & 2 & 1 & 6 & 8 & 5 \\ 7 & 6 & 1 & 8 & 9 & 2 & 7 & 9 & 5 & 4 \\ 3 & 1 & 2 & 3 & 3 & 4 & 1 & 1 & 3 & 8 \\ 7 & 4 & 2 & 7 & 7 & 9 & 3 & 1 & 9 & 8 \\ 6 & 5 & 0 & 2 & 8 & 6 & 0 & 2 & 4 & 8 \\ 6 & 5 & 0 & 9 & 0 & 0 & 6 & 1 & 3 & 8 \\ 9 & 3 & 4 & 4 & 6 & 0 & 6 & 6 & 1 & 8 \\ 4 & 9 & 6 & 3 & 7 & 8 & 8 & 2 & 9 & 1 \end{bmatrix}$	-24623624.000000

For calculation the order of complexity of the new algorithms (PERMUTDET1, PERMUTDET2 and PERMUTDETIT3), we multiply n to the order of complexity permutation algorithm in Chapter Three due to multiplication among n elements in permutation array. On the other hand, the formula for division free method given in Equation 4.5 as follows:

$$det(A) = \sum_{i=1}^{\frac{(n-1)!}{2}} \sum_{k=0}^{n-1} sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(j)_k)}] + sign(\sigma)[\prod_{j=1}^{n} a_{(j)(\sigma(n+1-j)_k)}].$$

Based on that, the number of steps in calculation of finding determinant for matrix of order n is $\frac{(n-1)!}{2} \times 2n = n!$ where $2n$ is number of step for Product Diagonal (PD) operation for each $\frac{(n-1)!}{2}$ generated starter sets matrix.

The pseudocode of PERMUTDET1 is given as follows:

```
1:  PERMUTDET1(temp)
2:  if temp = 2 then
3:      for i = 1 to n do
4:          old = num[i]
5:          for k = i to n − 1 do
6:              num[k] = num[k+1]
7:          end for
8:          num[n] = old
9:          calculate PD_i = SDP_i + MDP_i
10:         find the sign of PD[i]
11:         ∑_{i=1}^{n} PD_i
12:     end for
13:     det(A) = ∑_{j=1}^{(n-1)!/2} [∑_{i=1}^{n} PD_i]_j
14:     return
15: end if
```

16: $temp = temp - 1$

17: **for** $i = n$ to $temp$ **do**

18: old = num[i]

19: **for** $k = i$ to $n - 1$ **do**

20: num[k] = num[k+1]

21: **end for**

22: num[n] = old

23: call PERMUTDET1(temp)

24: **end for**

From algorithm of PERMUT1, we extend it for finding determinant known as PERMUTDET1 algorithm, by adding multiplication operation as given in steps 9-13. At step 9, the number of step for multiplication is n. This operation is employed on every permutation array. Recall that the order of complexity for PERMUT1 is $O(nn!)$. Then by multiplying n to this order of complexity of PERMUT1, the order of complexity of PERMUTDET1 is given by

$$O(n^2 n!). \tag{4.7}$$

We conclude the order of complexity for PERMUDET2 and PERMUTDETIT3 are similar to PERMUTDET1 since their order of complexity of PERMUT2 and PERMUTIT3 are similar to PERMUT1.

Meanwhile the order of complexity of cofactor expansion is $O(n((n-1)!)^3)$(Shin, 2002; Goldfinger, 2008). Table 4.6 displays the order of complexity of all algorithms.

Table 4.6: The Comparison Order of Complexity of the New Algorithms to Cofactor Expansion and Permutation

Algorithm	Order of Complexity
Recursive circular strategy algorithm (PERMUTDET1)	$O(n^2(n!))$
Recursive exchange strategy algorithm (PERMUTDET2)	$O(n^2(n!))$
Iterative circular strategy algorithm (PERMUTDETIT3)	$O(n^2(n!))$
Cofactor expansion	$O(n((n-1)!)^3)$
Langdon	$O(n^2(n!))$

From Table 4.6, our new algorithms are comparable to Langdon algorithm. Meanwhile, the order of complexity of the cofactor expansion for the parallel algorithm is $O((n!)^2)$ (Goldfinger, 2008) which is higher if to our new sequential algorithms.

From the mathematical aspect, Langdon (1967) and Thongchiew (2007) methods are designed without any corresponding to matrix structure namely set of diagonals. In other word, firstly all $n!$ permutations are needed to be listed which represented as column indices and later the product of each element is calculated with respect to the permutation column indices. Langdon (1967) and Thongchiew (2007) used the following mathematical formulation given by Leibniz (1678).

$$det(A) = \sum_{\sigma \in S_n} sign(\sigma) \prod_{i}^{n} a_{i\ \sigma(i)} \tag{4.8}$$

The sign of a permutation is defined in terms of the total number of inversions as follows:

$$sign(\sigma) = (-1)^{\text{total number of inversions in } \sigma} \tag{4.9}$$

On the other hand, the new methods are derived based on a square matrix structure corresponds to its main diagonal and secondary diagonal. Specifically the column indices of the main diagonal is the reverse of column indices of the secondary diagonal or vice versa which has been discussed in Section 4.2. Furthermore, the new division free method is a generalisation of cross multiplication (Sarrus Rule). Without listing the permutations, we are still able to find the product indirectly by finding the $\frac{(n-1)!}{2}$ of the nth order diagrams where the nth order diagrams constructed by appending the first $n-1$ columns to the right of matrix.

For hand computation, our new methods are easy used to calculate until $n \leq 5$ but for other permutation methods, it became tedious because we need list all the permutation. However for our methods, special strategies have been derived to generate starter sets for

$\frac{(n-1)!}{2}$ of the nth order diagrams. For example when $n = 5$, only 12 starter sets are needed to be found to list all 120 permutations in the new method. Meanwhile, for other permutation methods, 120 permutations are needed to be listed before performing the calculation of all elements in each permutation array.

Two operation strategies are developed for listing $\frac{(n-1)!}{2}$ of the nth order diagrams based on circular and exchange. Between these two operations, the exchange strategy is slightly simpler compared to the circular strategy because only two elements are involved in the former strategy. In circular strategy, more than two elements are involved.

Let demonstrate the hand computation for $n = 4$ to show the difference clearly.

Example 4.6.1.

For exchanging operation, only two columns are exchanged for generating next n order matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \Longrightarrow \begin{bmatrix} a_{11} & a_{13} & a_{12} & a_{14} \\ a_{21} & a_{23} & a_{22} & a_{24} \\ a_{31} & a_{33} & a_{32} & a_{34} \\ a_{41} & a_{43} & a_{42} & a_{44} \end{bmatrix} \Longrightarrow \begin{bmatrix} a_{11} & a_{13} & a_{14} & a_{12} \\ a_{21} & a_{23} & a_{24} & a_{22} \\ a_{31} & a_{33} & a_{34} & a_{32} \\ a_{41} & a_{43} & a_{44} & a_{42} \end{bmatrix}$$

Meanwhile for circular operation, three columns are cycled.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \Longrightarrow \begin{bmatrix} a_{11} & a_{13} & a_{14} & a_{12} \\ a_{21} & a_{23} & a_{24} & a_{22} \\ a_{31} & a_{33} & a_{34} & a_{32} \\ a_{41} & a_{43} & a_{44} & a_{42} \end{bmatrix} \Longrightarrow \begin{bmatrix} a_{11} & a_{14} & a_{12} & a_{13} \\ a_{21} & a_{24} & a_{22} & a_{23} \\ a_{31} & a_{34} & a_{32} & a_{33} \\ a_{41} & a_{44} & a_{42} & a_{43} \end{bmatrix}$$

In addition, exchange operation also showed simplicity in finding the sign of parallel diagonal of each starter set matrix compared to circular operation. When two columns are exchanged, the sign of a parallel diagonal for the next starter sets matrix will be alternated.

On the other hand, in circular operation, we have to calculate the inversion and then the sign of parallel diagonal for next starter sets matrix is determined.

4.7 Summary

The circular permutation pattern on diagonal column indices is discovered in cross multiplication method for case $n = 3$. This pattern is then extended for general cases by producing $\frac{(n-1)!}{2}$ starter set matrices. Then, the first $(n-1)$ columns are appended to the right of all starter set matrices for generating $\frac{(n-1)!}{2}$ of nth order diagrams. Next, by employing the product rule over sign $2n$ diagonal entries which are the main rule in cross multiplication method (Sarrus Rule), the determinant of the square matrix is calculated. As a result, the sequential division free method for finding determinant are derived and thus the second objective of this study is achieved. The contribution of this chapter is a new division free method and its algorithms are developed for square matrices. Moreover some new theoretical works are also developed. The numerical results show that the new algorithms are faster than other existing division free algorithms in term of computation time. The order of complexity of our new sequential algorithms is $O(n^2 n!)$ which better than cofactor expansion.

CHAPTER FIVE

DEVELOPING NEW PARALLEL METHODS FOR DETERMINANT

5.1 Introduction

In this chapter, two techniques of parallelization are considered to develop the parallel algorithm. The two techniques are:

(i) across the time.

(ii) across the method.

In order to develop parallel algorithms for finding the determinant via permutation approach for across the time approach, the sequential algorithm is applied to model of parallel computation (Kokosiński, 1990). Meanwhile for across the method approach, we have to redesign our sequential algorithms so that the algorithms are parallel in nature. After permutation generation algorithms are successfully constructed for both approaches, it will be applied for finding the determinant. In writing programs for parallel computer, the challenging task is to identify any parallelisable parts and avoided any data dependencies (Mohd Saman & Evans, 1995).

Parallelisation the permutation generation algorithm is the most crucial task for developing the parallel algorithm for finding the determinant. In our case, our permutation generation algorithms are dependent on the starter sets generation. Thus, the starter sets generation task will be parallelised. In spite of our work, no study has been conducted in partitioning permutation based on number of permutations. From previous study, researchers had partitioned only the elements of permutation for shared memory processing (Tsay & Lee, 1994; Akl et al., 1994; Djamegni & Tchuente, 1997; Cong & Bader, 2006).

The remaining of this chapter is organized as follows: some preliminary definitions are

given in Section 5.2. Then in Section 5.3, we extend sequential algorithms of generating permutation and finding determinant for parallel computation. Section 5.4 gives the explanation of designing the parallel algorithm for permutation generation where two methods are developed based on exchanging two elements. Then the applications of the parallel algorithms for finding the determinant are also discussed in that section. In Section 5.5, some theoretical works are presented for across the method algorithm for permutation generation. Section 5.6 discusses the performance of Across The Time (ATT) and Across The Method (ATM) algorithms for permutation generation. Finally in last section, the performance of ATT and ATM algorithms for finding the determinant are analyzed based on speedup and efficiency in Section 5.7.

5.2 Preliminary Definitions

The following definitions will be used throughout this study.

Definition 5.2.1. *The kth rank starter sets is the starter sets produced by performing circular operation(exchange operation) over k elements (element in kth position) respectively.*

Definition 5.2.2. *The subDeterminant of A (subDet(A)) is summation of n diagonal products (PD) of starter sets matrix, A.*

Definition 5.2.3. *Initial starter sets for across the time parallelization is kth rank starter sets.*

Definition 5.2.4. *Initial starter sets for across the method parallelization is a starter set generated from the identity permutation.*

5.3 Parallelization Across the Time (ATT)

In this method, the existing sequential algorithms do not need to be modified and redesigned. We have to identify the independency of the data or process and reassign them

to parallel compiler where the tasks among processors are identical. The task division is statically determined and it is appropriate to adopt static allocation. Therefore, task partitioning in starter sets generation is determined. In addition, for data allocation among the processors, a cyclic allocation strategy is used as follows:

Suppose j is the number of starter sets and $p-1$ is the number of processors (slave). Compute $j \equiv i \bmod p-1$. If $i = 0$, then j is a multiple of $p-1$ and every $p-1$ processor (slave) should has $\frac{j}{p-1}$ starter sets. If $i > 0$, then the first i processors should get $\frac{j-i}{p-1}+1$ and the remaining $p-1-i$ processors should get $\frac{j-i}{p-1}$ starter sets.

5.3.1 Parallel Algorithm for Permutation

The sequential algorithms for generating permutation are divided into two parts as follows:

Part One: Starter sets generation

Part Two: $n!$ permutations generation by exploiting the results in Part One.

All $n!$ permutation generations are dependent on the starter sets generation. As described in Chapter Three, the total number of starter sets needed to list all permutations is $\frac{(n-1)!}{2}$. Two algorithms that have been developed for generating permutations used the following strategies:

(i) Starter sets and permutation generation based on circular operation (PERMUT1).

(ii) Starter sets generation based on exchanging two elements operation while permutation generation based on circular operation (PERMUT2).

Since the task for starter sets generation is partitioned where master p_0 is assigned to generate starter sets with certain k value of procedure PERMUT1 and PERMUT2, a new formula is derived. Let consider $k \in Z^+$ where $2 \leq k < n$, and $p-1$ is the number of processors excluding the master. Each k value for PERMUT1 and PERMUT2 algorithms represented the t number of starter sets.

Let t represents the number of starter sets. Begin with $k = n - 1$. In order to determine the value t with respect to k value, the following formula is employed:

$$t = \frac{(n + 1 - k)!}{2} \tag{5.1}$$

where $2 \leq k \leq n - 1$.

Then from Equation 5.1, we list the number of starter sets as follows:

Table 5.1: The Number of Starter Sets Corresponding to k

k	$n-1$	$n-2$	$n-3$	$n-4$	...	2
t	1	3	12	60	...	$\frac{(n-1)!}{2}$

As a solution, for any $\frac{t}{p-1} > 0$, master will be assigned to run PERMUT1 and PERMUT2 algorithms from $k = n - 1$ until 2.

Example 5.3.1.

Let $n = 7$, and consider the number of processors $p = 6$. Therefore the number of slave is 5. Referring to Table 5.1, the master will run algorithms PERMUT1 and PERMUT2 from $k = 6$ until $k = 4$ or $k = 3$ or $k = 2$ alternatively. It is a statically determined. In other words, we can easily change the k value of algorithms with respect to the number of processors in master part (p_0).

Now the details tasks for master (p_0), and slaves (p_j) where $1 \leq j \leq p - 1$ is described in the following procedure:

Step 1: Task for master.

(i) reads the value of n, number of elements

(ii) initializes the initial permutation in ordered list

(iii) runs PERMUT1 from $k = n - 1$ until $k = n - 3$ or $k = n - 4$

(iv) stores the result in (iii) as kth rank starter sets in two dimension array

(v) distributes the kth rank starter sets to all slaves

This procedure can be translated into the following algorithm:

Algorithm 5.1 PERMUT1 for Master

PERMUT1(k) is perform by P_0
if $k = n - 3$ **then**
processor p_0 stores the data in two dimension matrix array and broadcast to other processors
return
end if
$k = k - 1$
for $i = n$ to k **do**
performing CO to the last k element
call PERMUT1(k)
end for

Step 2: Tasks for each slaves p_j where $j = 1$ to $p - 1$

(i) receives the kth rank starter sets and store it in its own memory

(ii) runs PERMUT1 from $k - 1$ until 2.

(iii) performs circular permutation and reversing of circular permutation

operation for all elements for each $(n - 1)$th rank starter sets.

(iv) stores all permutations in two dimension array

(v) sends all results to master

Below is the algorithm for each slave:

Algorithm 5.2 PERMUT1 for Each Slave

```
do in parallel
for j = 1 to p − 1 do
    PERMUT1(k) is perform by p_j
    if temp= 2 then
        for i = 1 to n do
            performing CP and RoCP for all element
        end for
        return
    end if
    k = k − 1
    for i = n to k do
        performing CO to the last k element
        call PERMUT1(k)
    end for
end for
```

Step 3 : Tasks for master

(i) receives the data from all slaves

(ii) prints the data

The tasks for master and slaves for PERMUT2 algorithm are allocated in same manner.

Algorithm 5.3 PERMUT2 for Master

```
PERMUT2(k) is perform by p_0
if k = n − 3 then
    processor p_0 stored the data in 2D matrix array and broadcast to other processors
    return
end if
k = k − 1
for i = k to n do
    performing exchanging process to the element at k-th position
    call PERMUT2(k)
end for
```

Algorithm 5.4 PERMUT2 algorithm for Each Slave

```
do in parallel
for j = 1 to p − 1 do
    PERMUT2(k) is perform by p_j
    if temp= 2 then
        for i = 1 to n do
            performing CP and RoCP for all element
        end for
        return
    end if
    k = k − 1
    for i = k to n do
        performing exchanging process to the element at k-th position
        call PERMUT2(k)
    end for
end for
```

Each slave generates starter sets from $k-1$ until 2 and performed CP and RoCP simultaneously to list all $n!$ permutations . In the next section, the parallel algorithm for permutation generation is applied for finding the determinant.

5.3.2 Parallel Algorithm for Finding Determinant

Since the permutation generation algorithms have been successfully parallelized, it would be easy to employ it for finding the determinant. The tasks of finding the determinant are assigned to the slaves. Each slave is responsible to find the product of elements in the main diagonal and its secondary diagonal based on the permutations.

In ATT algorithm, the master only generates starter sets at a particular rank and broadcast them to all slaves. Then each slave generates $\frac{(n-1)!}{2(p-1)}$ permutations and calculates the diagonal product. After that, each slave sends the result to master.

The tasks for master (p_0) and each slave (p_j) where $1 \leq j \leq p-1$ in ATT algorithm are described in details as follows:

Step 1 : Tasks for master

(i) reads the value of n, number of elements, and all elements of matrix A

(ii) initializes the initial permutation in ordered list

(iii) runs PERMUT1 from $k = n - 1$ until $k = n - 3$ or $k = n - 4$

(iv) stores the result in (iii) as kth rank starter sets in two dimension array

(v) distributes the k rank starter sets, value of n and matrix A to all other slaves

The algorithm for master tasks is similar to Algorithm 5.1.

Step 2 : Tasks for each slave p_j where $j = 1$ to $p - 1$

(i) receives the kth rank starter sets, value of n and matrix A and store it in its own memory

(ii) runs PERMUTDET1 from $k - 1$ until 2.

(iii) performs CO on each $(n - 1)$th starter sets

(iv) simultaneously calculates the product of elements in the main diagonal (PMD) and its secondary diagonal(PSD)

(v) sums up all values as $subDet(A)$ and sends to master

This procedure for each slave can be presented in the following algorithm:

Algorithm 5.5 PERMUTDET1 for Each Slave

do in parallel
for $j = 1$ to $p - 1$ **do**
  PERMUTDET1(k) is perform by p_j
  if $k = 2$ **then**
    for $l = 1$ to n **do**
      performing CO on all starter sets
      calculate $PD_l = SDP_l + MDP_l$
      total up $\sum_{k=1}^{n} PD_l$
    end for
    $subDet(A)_j + = \sum_{i=1}^{\frac{(n-1)!}{2(p-1)}} [\sum_{l=1}^{n} PD_l]_i$
    return
  end if
  $k = k - 1$
  for $i = n$ to k **do**
    performing CO to the last k element
    call PERMUTDET1(k)
  end for
end for

Step 3 : Tasks for master

(i) receives the value $subDet(A)_j$ from all slaves.

(ii) totals up $det(A) = \sum_{j=1}^{p-1} subDet(A)_j$.

The parallel process in determining the determinant is demonstrated by given an example for $n = 4$.

Example 5.3.2. *Let consider number of processors* $p = 3$ *and* $n = 4$.

Step 1 : Tasks for master

There are three starter sets generated by master i.e. $[1, 2, 3, 4]$, $[1, 3, 4, 2]$ and $[1, 4, 2, 3]$.

Then master broadcasts these three starter sets to all slaves p_1 and p_2.

Step 2 : Tasks for each slave

Do in parallel

P_1 generates the fourth order diagram based on starter sets $[1, 2, 3, 4]$ and $[1, 4, 2, 3]$ and calculates the product of eight diagonals for that diagram simultaneously.

Starter set: $[1, 2, 3, 4]$

$$|A_1| = \begin{vmatrix} a_{11} & a_{12} & a_{13} & a_{14} & |a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} & a_{24} & |a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} & a_{34} & |a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} & a_{44} & |a_{41} & a_{42} & a_{43} \end{vmatrix}$$

$$subDet(A_1) = \sum_{k=1}^{4} PD(A_{1,k})$$

Starter set : $[1, 4, 2, 3]$

$$|A_2| = \begin{vmatrix} a_{11} & a_{14} & a_{12} & a_{13} & |a_{11} & a_{14} & a_{12} \\ a_{21} & a_{24} & a_{22} & a_{23} & |a_{21} & a_{24} & a_{22} \\ a_{31} & a_{34} & a_{32} & a_{33} & |a_{31} & a_{34} & a_{32} \\ a_{41} & a_{44} & a_{42} & a_{43} & |a_{41} & a_{44} & a_{42} \end{vmatrix}$$

$$subDet(A_2) = \sum_{k=1}^{4} PD(A_{2,k})$$

Then total $subDet(A)$ = $subDet(A_1) + subDet(A_2)$. Slave p_1 sends that value to master.

For p_2 generates fourth order diagram based on starter sets $[1, 3, 4, 2]$ and calculates the product of eight diagonals for that diagram simultaneously.

Starter set :$[1, 3, 4, 2]$

$$|A_3| = \begin{vmatrix} a_{11} & a_{13} & a_{14} & a_{12} & |a_{11} & a_{13} & a_{14} \\ a_{21} & a_{23} & a_{24} & a_{22} & |a_{21} & a_{23} & a_{24} \\ a_{31} & a_{33} & a_{34} & a_{32} & |a_{31} & a_{33} & a_{34} \\ a_{41} & a_{43} & a_{44} & a_{42} & |a_{41} & a_{43} & a_{44} \end{vmatrix}$$

$$subDet(A_3) = \sum_{k=1}^{n} PD(A_{3,k}).$$

Then total $subDet(A)$ = $subDet(A_3)$. Slave p_2 sends that value to master.

Step 3 : Tasks for master

Master receives the result from all slaves and sums all the value as follows:

$$det(A) = \sum_{i=1}^{3}(subDet(A_i)).$$

For any n, we consider two cases for $p-1$ number of slave is as follows:

(i) $\frac{(n-1)!}{2} \equiv 0\ mod(p-1)$, then every slave calculate

$$subDet(A)_j = \sum_{i=1}^{\frac{(n-1)!}{2(p-1)}} \sum_{k=1}^{n} PD(A_{i,k}) \tag{5.2}$$

where $1 \leq j \leq p-1$.

(ii) $\frac{(n-1)!}{2} \equiv r\ mod(p-1)$ where $1 \leq r < p-1-r$, the first r processors calculate

$$subDet(A)_j = \sum_{i=1}^{\frac{(n-1)!-r}{2(p-1)}+1} \sum_{k=1}^{n} PD(A_{i,k}) \tag{5.3}$$

where $1 \leq j \leq r$,

The remaining $p-1-r$ slaves calculate

$$subDet(A)_j = \sum_{i=1}^{\frac{(n-1)!-r}{2(p-1)}} \sum_{k=1}^{n} PD(A_{i,k}) \tag{5.4}$$

where $p-1-r \leq j \leq p-1$

$$PD(A_{i,k}) = \sum_{k=1}^{n} (MDP + SDP)A_{i,k} \tag{5.5}$$

The detail of $(MDP + SDP)A_{i,k}$ can be referred in Equation 4.3 in Chapter Four.

The next section discusses the parallelisation across the method for generating permutation and finding the determinant.

5.4 Parallelization Across The Method (ATM)

According to Burrage (1995), across the method is a more naturalistic approach compared to across the time method. The alteration and restyling of sequential algorithms is required in this approach. Thus new algorithms are parallel in nature and fit to parallel computers well. However this approach is quite challenging because not only new algorithms have to be developed, the parallel executions also need to be considered.

In order to develop a new parallel algorithm for finding the determinant, once again we have to design the parallel algorithm of generating permutation.

5.4.1 Derivation of Parallel Algorithm for Permutation Generation

The alteration of circular operation (PERMUT1) from the initial permutation without starter sets allocation fails because the patterns of the generated permutation are not consistent with the original circular permutation for all $n > 3$. Thus as alternative, an exchange of two consecutive elements technique will be used.

The new parallel algorithm for generating permutation are divided to three parts as follows:

(i) Initial starter sets generation from identity permutation.

(ii) Starter sets generation from initial starter sets.

(iii) Listing all $n!$ permutation using result in (ii).

For cases $n = 2$, and 3 are trivial. We assume that there are $p - 1$ processors (slaves) for any n where $0 < p - 1 \leq n - 1$. Each $n > 3$, $n - 1$ initial starter sets are generated without depending on each other. These $n - 1$ initial starter sets are generated from identity permutation i.e. $[1, 2, 3, ..., k - 1, k, k + 1, ..., n - 1, n]$. Two parallel algorithms are developed as across the method namely PERATM1 and PERATM2.

5.4.1.1 Initial Starter Sets Generation

The process of identifying the initial starter sets for each processor is complicates because the dependency of data/process among processors is needed to be avoided. We illustrate the example for case $n = 4, 5$, and 6 in PERATM1.

(i) First strategy for initial starter sets (ISSG1)

Assume there are $n - 1$ slaves. Without loss of generality, the first element is fix. Identify element in the second position i.e. element '2' exchange with element in jth position from identity permutation where $1 < j < n$. For case $j = n$, the element '2' exchange two times where firstly exchange with element in $(n - 1)$th position then exchange with element in nth position. The process of generating initial starter sets depends on identity permutation. So every slave must generate the identity permutation and this task is done independently. The examples given below are for case $n = 4, 5$, and 6 in PERATM1.

Case $n = 4$

$$1 \quad 3 \quad 4 \quad \mathbf{2}$$

$$1 \quad \mathbf{2} \quad 3 \quad 4$$

$$1 \quad 3 \quad \mathbf{2} \quad 4$$

Case $n = 5$

$$1 \quad 4 \quad 3 \quad 5 \quad \mathbf{2}$$

$$1 \quad \mathbf{2} \quad 3 \quad 4 \quad 5$$

$$1 \quad 3 \quad \mathbf{2} \quad 4 \quad 5$$

$$1 \quad 4 \quad 3 \quad \mathbf{2} \quad 5$$

Case $n = 6$

$$1 \quad 5 \quad 3 \quad 4 \quad 6 \quad \mathbf{2}$$

$$1 \quad \mathbf{2} \quad 3 \quad 4 \quad 5 \quad 6$$

$$1 \quad 3 \quad \mathbf{2} \quad 4 \quad 5 \quad 6$$

$$1 \quad 4 \quad 3 \quad \mathbf{2} \quad 5 \quad 6$$

$$1 \quad 5 \quad 3 \quad 4 \quad \mathbf{2} \quad 6$$

See the following pseudocode ISSG1$(i, 2)$. The term ID in pseudocode mean any slave identity value.

Algorithm 5.6 ISSG1 $(i, 2)$

$ISSG1(i, 2)$ is perform by p_i
do in parallel
for $i = ID$ to $n - 1$; $i+ = p - 1$ **do**
  if $i = 1$ **then**
    $old = a[2]$
    $a[2] = a[n - 1]$
    $a[n - 1] = a[n]$
    $a[n] = old$
  else
    $old = a[i]$
    $a[i] = a[2]$
    $a[2] = old$
  end if
end for

(ii) Second strategy for initial starter sets (ISSG2)

Assume there are $n-1$ slaves. Without loss of generality, the last element is fix. Identify element in $(n-1)$th position i.e. element '$n-1$' exchange with element in jth position from identity permutation where $1 < j \leq n-1$. For case $j = 1$, element '$n-1$' exchange two times where firstly exchange with element in second position then exchange with element in first position. The process of generating initial starter sets depends on identity permutation. So every slave must generate the identity permutation and this task is done independently. The examples given below are for case $n = 4, 5,$ and 6 in PERATM2.

Case $n = 4$

$$\begin{array}{cccc} \mathbf{3} & 1 & 2 & 4 \\ 1 & \mathbf{3} & 2 & 4 \\ 1 & 2 & \mathbf{3} & 4 \end{array}$$

Case $n = 5$

$$\begin{array}{ccccc} \mathbf{4} & 1 & 3 & 2 & 5 \\ 1 & \mathbf{4} & 3 & 2 & 5 \\ 1 & 2 & \mathbf{4} & 3 & 5 \\ 1 & 2 & 3 & \mathbf{4} & 5 \end{array}$$

Case $n = 6$

$$\begin{array}{cccccc} \mathbf{5} & 1 & 3 & 4 & 2 & 6 \\ 1 & \mathbf{5} & 3 & 4 & 2 & 6 \\ 1 & 2 & \mathbf{5} & 4 & 3 & 6 \\ 1 & 2 & 3 & \mathbf{5} & 4 & 6 \\ 1 & 2 & 3 & 4 & \mathbf{5} & 6 \end{array}$$

See the following pseudocode ISSG2$(i, n-1)$. The term ID in pseudocode mean any slave identity value.

Algorithm 5.7 ISSG2 $(i, n-1)$

$ISSG2(i, n-1)$ is perform by p_i
do in parallel
for $i = ID$ to $n-1$; $i+ = p-1$ **do**
  if $i = 1$ **then**
    $old = a[2]$
    $a[2] = a[n-1]$
    $a[n-1] = old$
    $t = old$
    $old = a[i]$
    $a[i] = a[2]$
    $a[2] = old$
  else
    $old = a[i]$
    $a[i] = a[n-1]$
    $a[n-1] = old$
  end if
end for

For the case where the number of slaves $p < n-1$ and $n-1 \equiv r$ mod $(p-1)$ where $0 < r < p-1$, the cyclic allocation is employs in order to generate $(n-1)$ initial starter sets where the first r slaves will generate $\frac{n-1-r}{p-1} + 1$ initial starter sets. Meanwhile the remaining slaves $p-1-r$ slaves will generate $\frac{n-1-r}{p-1}$ initial starter sets. The starter sets generation from initial starter sets in each slave will be discussed in next section.

5.4.1.2 Starter Sets Generation from Initial Starter Sets

Since distinct initial starter sets are generated by slaves, it is easy to assign the slave to generate all distinct starter sets. Each slave will generate starter sets recursively where each initial starter set is exploited to produce $\frac{(n-2)!}{2}$ starter sets. The examples for $n = 4$ and 5 are demonstrated for PERATM1 algorithm and PERATM2 algorithm. The general algorithm for starter sets generation will then be derived. Assume that there is $n-1$ number of slaves.

(i) Starter sets generation where initial starter sets are produced from ISSG1($i, 2$).

Case $n = 4$ and $k = 4$.

Processor 1 generated $[1, 3, 4, 2]$

Processor 2 generated $[1, 2, 3, 4]$

Processor 3 generated $[1, 3, 2, 4]$

Case $n = 5$

Processor 1

Step 1 : Initial starter set $[1, 4, 3, 5, 2]$ is produced and $k = 4$.

Step 2 : New $k = 4 - 1 = 3$. Identify the element in the third position i.e. element '3'. Exchange this element until it reaches the 5th (last) position.

$$1 \quad 4 \quad 3 \quad 5 \quad 2$$

$$1 \quad 4 \quad 5 \quad 3 \quad 2$$

$$1 \quad 4 \quad 5 \quad 2 \quad 3$$

Processor 2

Step 1 : Initial starter set $[1, 2, 3, 4, 5]$ is produced and $k = 4$

Step 2 : New $k = 4 - 1$. Identify the element in the third position i.e. element '3'. Exchange this element until it reaches the 5th (last) position.

$$1 \quad 2 \quad 3 \quad 4 \quad 5$$

$$1 \quad 2 \quad 4 \quad 3 \quad 5$$

$$1 \quad 2 \quad 4 \quad 5 \quad 3$$

Processor 3

Step 1 : Initial starter set $[1, 3, 2, 4, 5]$ is produced and $k = 4$.

Step 2 : New $k = 4 - 1$. Identify the element in the third position i.e. element '2'. Exchange this element until it reaches the 5th (last) position.

$$1 \quad 3 \quad 2 \quad 4 \quad 5$$

$$1 \quad 3 \quad 4 \quad 2 \quad 5$$

$$1 \quad 3 \quad 4 \quad 5 \quad 2$$

Processor 4

Step 1 : Initial starter set $[1, 4, 3, 2, 5]$ is produced and $k = 4$.

Step 2 : New $k = 4 - 1$. Identify the element in the third position i.e. element '3'. Exchange this element until it reaches the 5th (last) position.

$$1 \quad 4 \quad 3 \quad 2 \quad 5$$

$$1 \quad 4 \quad 2 \quad 3 \quad 5$$

$$1 \quad 4 \quad 2 \quad 5 \quad 3$$

The general algorithm of starter sets generation from initial starter sets for each processor, p_i where $1 \leq i \leq p - 1$ as follows:

Step 1: Initial starter sets is produced after performing procedure ISSG1$(i, 2)$ and set $k = n - 1$.

Step 2: New $k = k - 1$. Identify the element in the kth position. Exchange this element until it reaches the nth (last) position.

Step 3: Test whether $k = 3$. If true, the process is stopped otherwise go to Step 2 for identifying the element in new kth position for each tth rank starter where $t = n + 1 - k$.

(ii) Starter sets generation where initial starter sets are produced from ISSG2($i, n-1$).

Case $n = 4$ and $k = 2$

Processor 1 generated $[3, 1, 2, 4]$

Processor 2 generated $[1, 2, 3, 4]$

Processor 3 generated $[1, 3, 2, 4]$

Case $n = 5$,

Processor 1

Step 1 : Initial starter set $[1, 2, 3, 4, 5]$ is produced and $k = 2$.

Step 2 : New $k = 2 + 1$. Identify the element in the third position i.e. element '3'. Exchange this element until it reaches the first position.

$$1 \quad 2 \quad 3 \quad 4 \quad 5$$

$$1 \quad 3 \quad 2 \quad 4 \quad 5$$

$$3 \quad 1 \quad 2 \quad 4 \quad 5$$

Processor 2

Step 1 : Initial starter set $[2, 1, 3, 4, 5]$ is produced and $k = 2$.

Step 2 : New $k = 2 + 1$. Identify the element in the third position i.e. element '3'. Exchange this element until it reaches the first position.

$$2 \quad 1 \quad 3 \quad 4 \quad 5$$

$$2 \quad 3 \quad 1 \quad 4 \quad 5$$

$$3 \quad 2 \quad 1 \quad 4 \quad 5$$

Processor 3

Step 1 : Initial starter set $[1, 2, 4, 3, 5]$ is produced and $k = 2$

Step 2 : New $k = 2 + 1$. Identify the element in the third position i.e.

element '4'. Exchange this element until it reaches the first position.

$$1 \quad 2 \quad 4 \quad 3 \quad 5$$
$$1 \quad 4 \quad 2 \quad 3 \quad 5$$
$$4 \quad 1 \quad 2 \quad 3 \quad 5$$

Processor 4

Step 1 : Initial starter set $[1, 4, 3, 2, 5]$ is produced and $k = 2$

Step 2 : New $k = 2 + 1$. Identify the element in the third position i.e. element '3'. Exchange this element until it reaches the first position.

$$1 \quad 4 \quad 3 \quad 2 \quad 5$$
$$1 \quad 3 \quad 4 \quad 2 \quad 5$$
$$3 \quad 1 \quad 4 \quad 2 \quad 5$$

The general algorithm of starter sets generation from initial starter sets for each processor, p_i where $1 \leq i \leq p - 1$ as follows:

Step 1: Initial starter sets is produced after performing procedure ISSG2$(i, n - 1)$ and set $k = 2$.

Step 2: New $k = k + 1$. Identify the element in the kth position. Exchange this element until it reaches the first position.

Step 3: Test whether $k = n - 2$. If true, the process is stopped otherwise go to Step 2 for identifying the element in new kth position for each kth rank starter sets.

5.4.1.3 Permutation Generation

Circular permutation and reversing of circular permutation operation are employed on the starter sets for listing all $n!$ permutations. Illustration is given only for the starter sets

which were developed in ISSG1($i, 2$).

Case $n = 4$

Processor 1: $[1, 3, 4, 2]$

1	3	4	2
3	4	2	1
4	2	1	3
2	1	3	4

2	4	3	1
1	2	4	3
3	1	2	4
4	3	1	2

Processor 2: $[1, 2, 3, 4]$

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

4	3	2	1
1	4	3	2
2	1	4	3
3	2	1	4

Processor 3: $[1, 3, 2, 4]$

1	3	2	4
3	2	4	1
2	4	1	3
4	1	3	2

4	2	3	1
1	4	2	3
3	1	4	2
2	3	1	4

Case $n = 5$

Processor 1: $[1, 4, 3, 5, 2]$

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

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

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

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

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

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

Processor 2: $[1, 2, 3, 4, 5]$

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

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

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

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

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

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

Processor 3: $[1, 3, 2, 4, 5]$

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

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

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

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

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

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

Processor 4: $[1, 4, 3, 2, 5]$

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

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

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

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

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

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

The difference between PERATM1 and PERATM2 algorithms can be observed through starter sets generation. In Table 5.2, $a[i]$ is represented as an element in ith position. The detail is as follows :

Table 5.2: The Difference of PERATM1 and PERATM2 in Starter Set Generation

	PERATM1	PERATM2
Fixed Element	$a[1]$	$a[n]$
Initial element exchange	$a[n-2]$	$a[3]$
Last element exchange	$a[3]$	$a[n-2]$
Direction of exchange	Exchange to the right	Exchange to the left

Both of them use similar CP and RoCP operations to generate all $n!$ permutations.

5.4.2 Parallel Algorithm for Permutation Generation

The model of our parallel computational graph with master-slave approach is as follows where there is no communication between slaves:

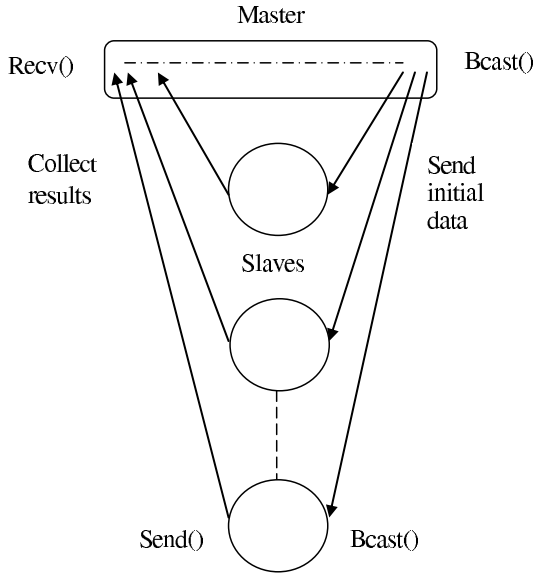

Figure 5.1: Parallel Computational Graph with Master-Slave Approach

The general task for master (p_0) and slaves (p_i) where $1 \leq i \leq p-1$ for PERATM1 algorithm is as follows:

Step 1 : Task for master

(i) reads and stores the value of n, number of elements.

(ii) broadcasts the value of n to all slaves.

Step 2 : Task for each slave p_i where $1 \leq i \leq n-1$

(i) receives the value of n and store it in its own memory.

(ii) performs initialization based on ISSG1$(i, 2)$ algorithm (refer page 123) and stores initial starter sets as second rank starter sets.

(iii) determines all $(n-2)$th rank starter sets.

(iv) performs circular and reversing of circular operations for all elements for each $(n-2)$th rank starter set.

(v) sends all results to master.

Step 3 : Task for master

(i) receive results from all slaves.

(ii) prints the results.

The general task for master (p_0) and slaves (p_i) where $1 \leq i \leq p-1$ for PERATM2 algorithm is as follows:

Step 1 : Master

(i) reads and stores the value of n, number of elements.

(ii) broadcasts the value of n to all slaves

Step 2 : Task for each slave p_i where $1 \leq i \leq n-1$

(i) receives the value of n and store it in own memory.

(ii) performs initialization based on ISSG2$(i, n-1)$ algorithm (refer page 125) and stores initial starter sets as second rank starter sets.

(iii) determines all $(n-2)$th rank starter sets.

(iv) performs circular and reversing of circular operations for all elements for

each $(n-2)$th rank starter sets.

(v) sends all results to master.

Step 3 : Task for master

(i) receives results from all slaves.

(ii) prints the results.

5.4.3 Parallel Algorithm for Finding the Determinant

Master only broadcasts the value of n and $n \times n$ matrix to all slaves and receives results from slaves. Then all slaves generate $\frac{(n-1)!}{2(p-1)}$ permutations and calculate the diagonal products. After that, slaves send the results to master.

The general task for master (p_0) and slaves (p_i) where $1 \leq i \leq p-1$ for PDATM1 is as follows:

Step 1 : Task for master

(i) reads and stores the value of n, number of elements and all elements of matrix A.

(ii) broadcasts the value of n and matrix A to all slaves.

Step 2 : Task for every slave p_i where $1 \leq i \leq n-1$

(i) receives the value of n and store it in its own memory.

(ii) performs initialization based on ISSG1$(i, 2)$ algorithm (refer page 123) and stores initial starter sets as second rank starter sets.

(iii) determine all $(n-2)$ rank starter sets.

(iv) performs CO to each $(n-2)$ rank starter set which was allocated using cyclic allocation.

(v) calculates the product of elements in the main diagonal(MDP)

and its secondary diagonal(PSD).

(vi) sums up all values as $subDet(A)$ and sends to master.

This procedure for each slave can be presented in the following algorithm:

Algorithm 5.8 PDATM1 for Each Slave

PDATM1(k)
if $k = 3$ **then**
  for $i = 1$ to n **do**
    performing CO to all starter sets
    calculate $PD_i = MDP_i + SDP_i$
    $\sum_{i=1}^{n} PD_i$
  end for
  $subDet(A)_i = \sum_{j=1}^{\frac{(n-2)!}{2(p-1)}} [\sum_{i=1}^{n} PD_i]_j$
  return
end if
$k = k - 1$
for $i = k$ to n **do**
  performing exchanging process to the element at kth position
  call PDATM1(k)
end for

Step 3 : Master

(i) receives the values $subDet(A)_i$ from all slaves.

(ii) total up $det(A) = \sum_{i=1}^{p-1} subDet(A)_i$.

Then for the PDATM2, the different tasks arise for each slave is as follows:

Algorithm 5.9 PDATM2 for Each Slave

PDATM2(k)
if $k = n - 2$ **then**
  for $i = 1$ to n **do**
    performing CO to all starter sets
    calculate $PD_i = MDP_i + SDP_i$
    $\sum_{i=1}^{n} PD_i$
  end for
  $subDet(A)_i = \sum_{j=1}^{\frac{(n-2)!}{2(p-1)}} [\sum_{i=1}^{n} PD_i]_j$
  return
end if
$k = k + 1$
for $i = k$ to 1 **do**
  performing exchanging process to the element at kth position to the first position
  call PDATM2(k)
end for

The parallel process in finding the determinant using across the method algorithm for $n = 4$ is given as follows:

Example 5.4.1. *Different from ATT algorithm, master only broadcasts data $n = 4$ and square matrix to all slaves. Let consider the number of slaves equal to three. Slave p_1 generates initial starter set $[1, 3, 4, 2]$,p_2 generates initial starter set $[1, 2, 3, 4]$ and p_3 generates initial starter set $[1, 3, 2, 4]$. Each slave performs its own task independently and simultaneously sends the results to master as follows:*

Slave p_1

Starter set :$[1, 3, 4, 2]$

$$|A_1| = \begin{vmatrix} a_{11} & a_{13} & a_{14} & a_{12} & |a_{11} & a_{13} & a_{14} \\ a_{21} & a_{23} & a_{24} & a_{22} & |a_{21} & a_{23} & a_{24} \\ a_{31} & a_{33} & a_{34} & a_{32} & |a_{31} & a_{33} & a_{34} \\ a_{41} & a_{43} & a_{44} & a_{42} & |a_{41} & a_{43} & a_{44} \end{vmatrix}$$

$$subDet(A_1) = \sum_{k=1}^{4} PD(A_{1,k})$$

Slave p_2

Starter set: $[1, 2, 3, 4]$

$$|A_2| = \begin{vmatrix} a_{11} & a_{12} & a_{13} & a_{14} & |a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} & a_{24} & |a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} & a_{34} & |a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} & a_{44} & |a_{41} & a_{42} & a_{43} \end{vmatrix}$$

$$subDet(A_2) = \sum_{k=1}^{4} PD(A_{2,k})$$

Slave p_3

Starter set :$[1, 3, 2, 4]$

$$|A_3| = \begin{vmatrix} a_{11} & a_{13} & a_{12} & a_{14} & |a_{11} & a_{13} & a_{12} \\ a_{21} & a_{23} & a_{22} & a_{24} & |a_{21} & a_{23} & a_{22} \\ a_{31} & a_{33} & a_{32} & a_{34} & |a_{31} & a_{33} & a_{32} \\ a_{41} & a_{43} & a_{42} & a_{44} & |a_{41} & a_{43} & a_{42} \end{vmatrix}$$

$$subDet(A_3) = \sum_{k=1}^{4} PD(A_{3,k})$$

Master receives the result from all slaves and sums all the value as follows:

$$det(A) = \sum_{i=1}^{3} (subDet(A_i))$$

For any n, we consider two cases for $p-1$ number of slaves:

(i) If $n - 1 \equiv 0 \pmod{p-1}$, then every slave calculates

$$subDet(A)_i = \sum_{j=1}^{\frac{(n-2)!}{2(p-1)}} \sum_{k=1}^{n} PD(A_{j,k}) \tag{5.6}$$

where $1 \leq i \leq p-1$.

(ii) If $n-1 \equiv r\ (mod p-1)$ where $1 \leq r < p-1$, the first r slaves calculate

$$subDet(A)_i = \sum_{j=1}^{\frac{(n-2)!-r}{2(p-1)}+1} \sum_{k=1}^{n} PD(A_{j,k}) \tag{5.7}$$

where $1 \leq i \leq r$. The remaining $p-1-r$ slaves calculate

$$subDet(A)_i = \sum_{j=1}^{\frac{(n-2)!-r}{2(p-1)}} \sum_{k=1}^{n} PD(A_{j,k}) \tag{5.8}$$

where $p-1-r \leq i \leq p-1$.

The theoretical results for permutation generation are presented in the following section.

5.5 Theoretical Results for Across The Method Algorithm

The following lemmas and theorem are produced from ATM algorithm for permutation generation.

Lemma 5.5.1. *The number of initial starter sets produced under procedure ISSG1*$(i,2)$ *for* $n \geq 4$ *and* $1 \leq i < n$ *is* $n-1$.

Proof. In procedure ISSG1$(i,2)$, i represents the processor p_i and number 2 is denoted the element in the second position is selected to be exchanged. For $i \neq 1$, the exchange process is between elements $a[i]$ and $a[2]$ from identity permutation $[1,2,3,\ldots,k,k+1,\ldots,n-3,n-2,n-1,n]$. Thus from $i=2$ until $i=n-1$, there are $(n-2)$ initial starters sets produce:

$$\begin{aligned}
i = 2 &\rightarrow [1,2,3,\ldots,k,k+1,\ldots,n-3,n-2,n-1,n] \\
i = 3 &\rightarrow [1,3,2,\ldots,k,k+1,\ldots,n-3,n-2,n-1,n] \\
\vdots &\rightarrow \qquad\qquad \vdots \\
i = n-2 &\rightarrow [1,n-2,3,\ldots,k,k+1,\ldots,n-3,2,n-1,n] \\
i = n-1 &\rightarrow [1,n-1,3,\ldots,k,k+1,\ldots,n-3,n-2,2,n]
\end{aligned}$$

For $i = 1$, the exchange process is done twice with firstly exchanging between $a[2]$ with $a[n-1]$. Then new $a[n-1]$ exchanges with $a[n]$. Thus another initial starter sets is produced as follows:

$$\rightarrow \quad [1, n-1, 3, \ldots, k, k+1, \ldots, n-3, n-2, n, 2]$$

Thus, the total number of initial starter sets is $(n-1)$. □

Lemma 5.5.2. *The number of initial starter sets produced under procedure ISSG2*$(i, n-1)$ *for* $n \geq 4$ *and* $1 \leq i < n$ *is* $n-1$.

Proof. In procedure ISSG2$(i, n-1)$, i represents the processor p_i and number $(n-1)$ represents element in $(n-1)$th position is selected to be exchanged. For $i \neq 1$, the exchange process is between elements $a[i]$ and $a[n-1]$ from identity permutation $[1, 2, 3, \ldots, k, k+1, \ldots, n-3, n-2, n-1, n]$. Thus from $i = 2$ until $i = n-1$, there are $(n-2)$ initial starters sets produced as follows:

$$\begin{aligned}
i = 2 \quad &\rightarrow \quad [1, n-1, 3, \ldots, k, k+1, \ldots, n-3, n-2, 2, n] \\
i = 3 \quad &\rightarrow \quad [1, 2, n-1, \ldots, k, k+1, \ldots, n-3, n-2, 3, n] \\
\vdots \quad &\rightarrow \quad \vdots \\
i = n-2 \quad &\rightarrow \quad [1, 2, 3, \ldots, k, k+1, \ldots, n-3, n-1, n-2, n] \\
i = n-1 \quad &\rightarrow \quad [1, 2, 3, \ldots, k, k+1, \ldots, n-3, n-2, n-1, n]
\end{aligned}$$

For $i = 1$, the exchange process of exchanging is done twice where with firstly between $a[2]$ with $a[n-1]$. Then new $a[2]$ with $a[1]$. Thus another initial starter sets is produced as follows:

$$\rightarrow \quad [n-1, 1, 3, \ldots, k, k+1, \ldots, n-3, n-2, 2, n]$$

Thus, the total number of initial starter sets is $(n-1)$. □

Theorem 5.5.3. *The number of starter sets from each initial starter sets by exchanging one element to the right for* $n \geq 4$ *is* $\dfrac{(n-2)!}{2}$.

Proof. From Lemma 5.5.1, there are $(n-1)$ starter sets for $n \leq 4$ by exchanging one element to the right. In order to show that each initial starter set produce $\frac{(n-2)!}{2}$ starter sets, it is enough to prove by selecting one initial starter set. Let consider identity permutation, $[1, 2, 3, \ldots, k, k+1, \ldots, n-3, n-2, n-1, n]$ as one of the initial starter sets. The first element will be selected from $(n-2)$th position i.e. element $n-2$. Then by moving that element to the right until it reaches nth position, three distinct starter sets are produced :

$$\begin{array}{cccccccc} 1 & 2 & 3 & \ldots & n-3 & \mathbf{n-2} & n-1 & n \quad (\text{starter } 1) \\ 1 & 2 & 3 & \ldots & n-3 & n-1 & \mathbf{n-2} & n \quad (\text{starter } 2) \\ 1 & 2 & 3 & \ldots & n-3 & n-1 & n & \mathbf{n-2} \quad (\text{starter } 3) \end{array}$$

Then for each previous starter set, the element in $(n-3)$th will be selected i.e. element $n-3$. Then by moving that element to the right until it reaches nth position from each previous starter set, four distinct starter sets are produced:

$$\begin{array}{lccccccc} \text{From starter 1:} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \ldots & \mathbf{n-3} & \mathbf{n-2} & \mathbf{n-1} & \mathbf{n} \\ & 1 & 2 & 3 & \ldots & n-2 & \mathbf{n-3} & n-1 & n \\ & 1 & 2 & 3 & \ldots & n-2 & n-1 & \mathbf{n-3} & n \\ & 1 & 2 & 3 & \ldots & n-2 & n-1 & n & \mathbf{n-3} \\ \\ \text{From starter 2:} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \ldots & \mathbf{n-3} & \mathbf{n-1} & \mathbf{n-2} & \mathbf{n} \\ & 1 & 2 & 3 & \ldots & n-1 & \mathbf{n-3} & n-2 & n \\ & 1 & 2 & 3 & \ldots & n-1 & n-2 & \mathbf{n-3} & n \\ & 1 & 2 & 3 & \ldots & n-1 & n-2 & n & \mathbf{n-3} \\ \text{From starter 3:} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \ldots & \mathbf{n-3} & \mathbf{n-1} & \mathbf{n} & \mathbf{n-2} \\ & 1 & 2 & 3 & \ldots & n-1 & \mathbf{n-3} & n & n-2 \\ & 1 & 2 & 3 & \ldots & n-1 & n & \mathbf{n-3} & n-2 \\ & 1 & 2 & 3 & \ldots & n-1 & n & n-2 & \mathbf{n-3} \end{array}$$

Thus the total starter sets are $3 \times 4 = 12$. The processes will continue recursively until

element in third position is selected.

$$\begin{array}{rcl}
(n-2)th \text{ position} & \Rightarrow & 3 \text{ starter sets} \\
(n-3)th \text{ position} & \Rightarrow & 4 \text{ starter sets} \\
(n-4)th \text{ position} & \Rightarrow & 5 \text{ starter sets} \\
(n-5)th \text{ position} & \Rightarrow & 6 \text{ starter sets} \\
\vdots & & \vdots \\
(n-i+1)th \text{ position} & \Rightarrow & i \text{ starter sets} \\
(n-i)th \text{ position} & \Rightarrow & i+1 \text{ starter sets} \\
(n-i-1)th \text{ position} & \Rightarrow & i+2 \text{ starter sets} \\
\vdots & & \vdots \\
third \text{ position} & \Rightarrow & n-2 \text{ starter sets}
\end{array}$$

By product rule,

$$\begin{aligned}
& 3 \times 4 \times ... \times n-2 \\
=& \frac{1 \times 2}{2} \times (3 \times 4 \times ... \times n-2)) \\
=& \frac{(n-2)!}{2} \text{distinct starter sets}
\end{aligned}$$

□

This theorem is also define for exchanging one element to the left which is from Lemma 5.5.2, the initial starter sets are produced under procedure ISSG2$(i, n-1)$. This theorem is not valid for $n = 3$ is since it has only three distinct starter sets while $\frac{(3-2)!}{2} = \frac{1}{2}$ where is not evenly divisible.

Theorem 5.5.4. *The total number of starter sets from all* $n-1$ *initial starter sets by exchanging one element to the right for* $n \geq 4$ *is* $\frac{(n-1)!}{2}$.

Proof. From Theorem 5.5.4, the total starter sets for one initial starter sets is $\frac{(n-2)!}{2}$.

Thus the total starter sets for $(n-1)$ initial starter sets are:

$$
\begin{aligned}
&= \frac{(n-2)!}{2} \times (n-1) \\
&= \frac{(n-1)!}{2}
\end{aligned}
$$

□

Numerical results will be presented in the following section.

5.6 Performance of Parallel Algorithm for Permutation Generation

The performance of both across the time and across the method strategies for permutation generation and its application for finding the determinant are analysed in this section.

5.6.1 Numerical Results of Across The Time Permutation Algorithm

PERMUT1 and PERMUT2 are tested for two cases. In the first case, the master broadcasts an initial starter sets matrix of size $12 \times n$ to all slaves while in the second case, master broadcasts the initial starter sets matrix of size $60 \times n$ to slaves. The selection number of 12 and 60 initial starter sets (ISS) is based on the starter sets number $\frac{(n-1)!}{2}$. In all tables, p and n represent the number of processors and the number of element respectively. The total number of processor available is 10. The computation time given in seconds.

Table 5.3: The Computation Time of PERMUT1 with 12 Initial Starter Sets (in seconds)

	n			
p	10	11	12	13
1	0.288820	2.983579	37.634830	455.784241
2	0.288928	3.311018	37.460641	511.226828
3	0.144903	1.655674	20.775988	282.508526
4	0.096881	1.105749	13.876782	188.605803
5	0.074026	0.829904	10.390958	141.345779
6	0.072624	0.828566	10.387230	141.301121
7	0.050432	0.554404	6.928929	94.442333
8	0.048530	0.553717	6.933739	94.348762
9	0.052930	0.560323	7.163270	94.902682
10	0.066153	0.557403	7.120658	95.101400

From Table 5.3, the computation times decrease. However, the computation times deteriorate for $8 \leq p \leq 10$. This situation occurs because process of generating permutation is optimal at $p = 7$. The total number row that will be evenly allocated to the slaves since $\frac{p-1}{6} = 2$. However when the number of slaves increases, for example $p = 9$, the two rows of initial starter sets matrix is allocated to first four slaves, while the remaining four slaves get one row. This causes unload balancing. In order to reduce unload balancing, the number of initial starter sets is change to 60 which gives the following results.

Table 5.4: The Computation Time of PERMUT1 with 60 Initial Starter Sets (in seconds)

	n			
p	10	11	12	13
1	0.288820	2.983579	37.634830	455.783526
2	0.289141	2.999253	41.616693	510.207046
3	0.144663	1.659321	20.775683	282.541250
4	0.096680	1.104178	13.848872	188.374949
5	0.074534	0.830221	10.392067	141.277263
6	0.060285	0.665733	8.310123	113.187266
7	0.048844	0.552607	6.941704	94.357537
8	0.047319	0.499046	6.235494	84.880762
9	0.041098	0.448966	5.654323	77.233163
10	0.043334	0.389413	5.047950	67.763967

From Table 5.4, the computation times reduce when the number of the initial starter sets matrix row increases from 12 to 60. The computation timed continue to decrease as the number of processors increases.

The speedup and efficiency of PERMUT1 from $10 \leq n \leq 13$ for 12 and 60 initial starter sets are presented in Tables 5.5 and 5.6 respectively.

Table 5.5: The Speedup of PERMUT1 with 12 and 60 Initial Starter Sets

	n							
	10		11		12		13	
p	12	60	12	60	12	60	12	60
1	1	1	1	1	1	1	1	1
2	0.9996	0.9989	0.9011	0.9948	1.0046	0.9043	0.8915	0.8933
3	1.9932	1.9965	1.8020	1.7981	1.8115	1.8115	1.6133	1.6132
4	2.9812	2.9874	2.6982	2.7021	2.7121	2.7175	2.4166	2.4196
5	3.9016	3.8750	3.5951	3.5937	3.6219	3.6215	3.2246	3.2262
6	3.9769	4.7909	3.6009	4.4816	3.6232	4.5288	3.2256	4.0268
7	5.7269	5.9131	5.3816	5.3991	5.4315	5.4216	4.8261	4.8304
8	5.9514	6.1037	5.3883	5.9780	5.4278	6.0356	4.8308	5.3697
9	5.4566	7.0276	5.3247	6.6454	5.2539	6.6559	4.8026	5.9014
10	4.3659	6.6649	4.3993	7.6617	5.0037	7.4555	4.5815	6.7260

Table 5.6: The Efficiency of PERMUT1 with 12 and 60 Initial Starter Sets

	n							
	10		11		12		13	
p	12	60	12	60	12	60	12	60
1	1	1	1	1	1	1	1	1
2	0.4998	0.4994	0.4974	0.4506	0.5023	0.4522	0.4458	0.4467
3	0.6644	0.6655	0.6007	0.5994	0.6038	0.6038	0.5378	0.5377
4	0.7453	0.7469	0.6746	0.6755	0.6780	0.6793	0.6042	0.6049
5	0.7803	0.775	0.7190	0.7187	0.7244	0.7243	0.6449	0.6452
6	0.6628	0.7985	0.6015	0.7469	0.6039	0.7548	0.5376	0.6711
7	0.8181	0.8447	0.7688	0.7713	0.7759	0.7745	0.6894	0.6900
8	0.7439	0.7630	0.6735	0.7472	0.6785	0.7545	0.6038	0.6712
9	0.6063	0.7808	0.5916	0.7339	0.5838	0.7395	0.5336	0.6557
10	0.4366	0.6665	0.4399	0.7662	0.5004	0.7456	0.4581	0.6726

Tables 5.5 and 5.6 show that the speedup of the algorithm for 60 initial starter sets is better than 12 initial starter sets especially for p = 8, 9, and 10. This is due to the allocation number of the rows for $60 \times n$ initial starter sets matrix among processors is more evenly distributed if compared to the allocation rows for $12 \times n$ initial starter sets matrix. Furthermore, the efficiency of the parallel algorithm also increases when the

number of the initial starter sets matrix rows is 60. Refer Figures 5.2 and 5.3 for speedup, and Figures 5.4 and 5.5 for efficiency.

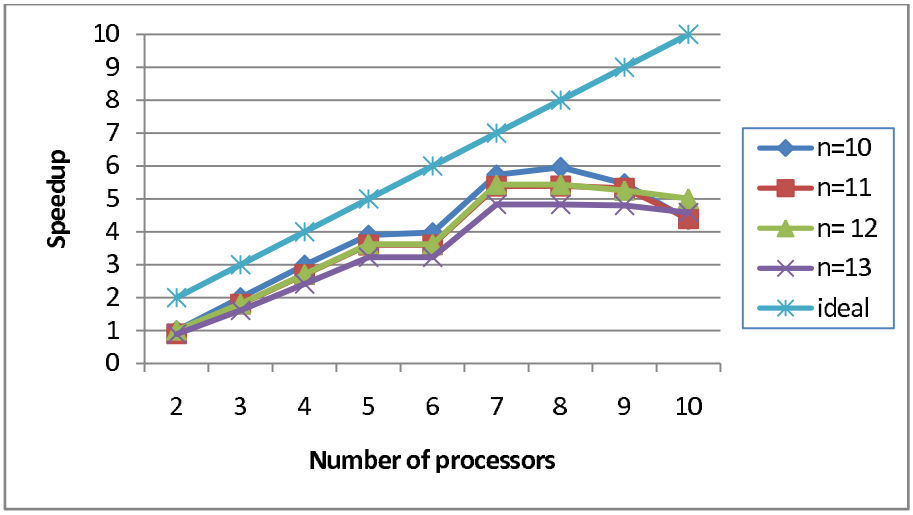

Figure 5.2: Speedup versus Number of Processors for PERMUT1 with 12 Initial Starter Sets

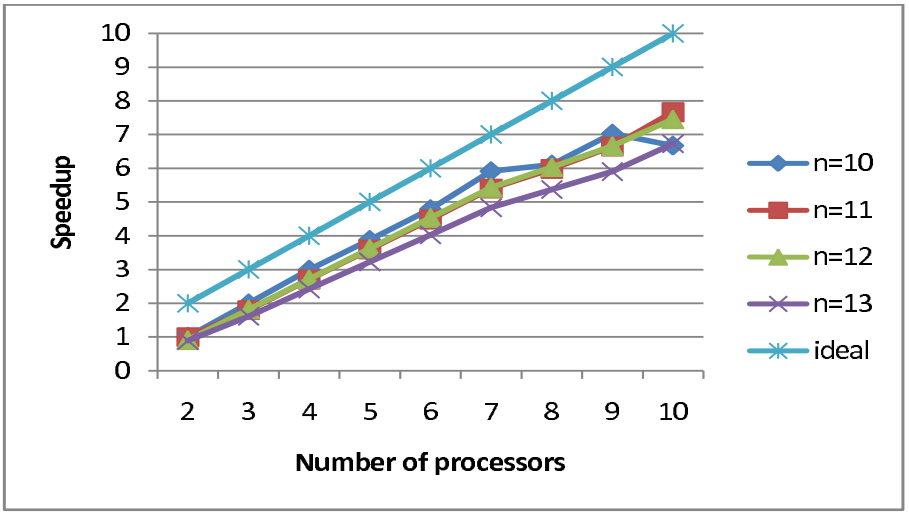

Figure 5.3: Speedup versus Number of Processors for PERMUT1 with 60 Initial Starter Sets

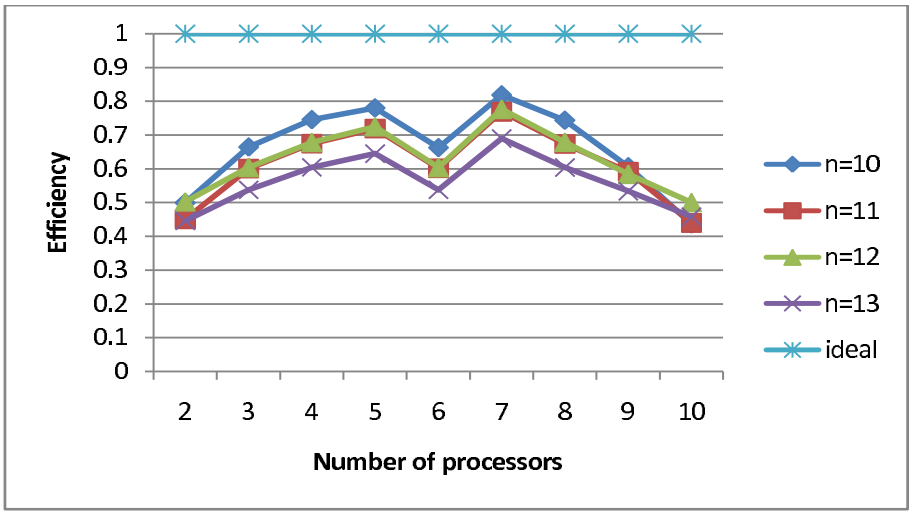

Figure 5.4: Efficiency versus Number of Processors for PERMUT1 with 12 Initial Starter Sets

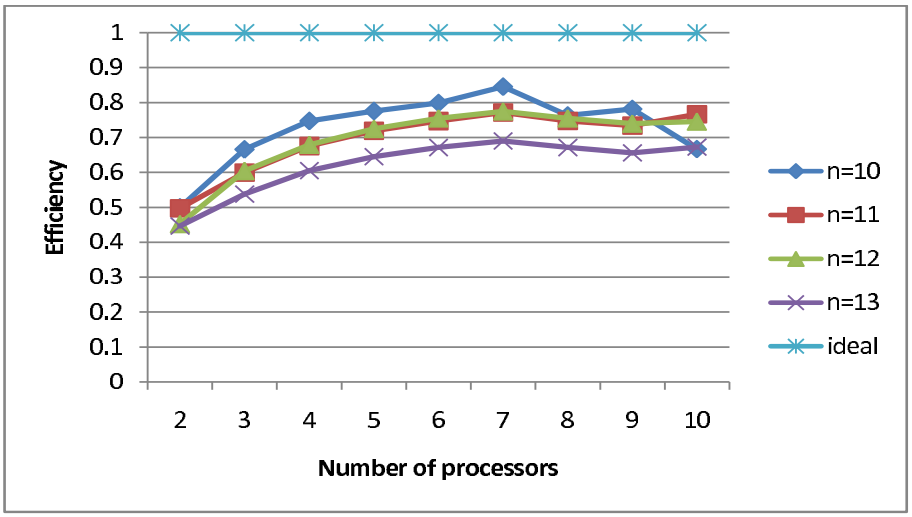

Figure 5.5: Efficiency versus Number of Processors for PERMUT1 with 60 Initial Starter Sets

As can be observed from Figures 5.2 and 5.3, the speedup of PERMUT1 with initial 60 starter sets is better than PERMUT1 with 12 initial starter sets. Meanwhile from Figures 5.4, and 5.5, the PERMUT1 with 60 initial starter sets are more efficient along the number of processors compared to its own algorithm with 12 initial starter sets. The cost overhead due to broadcasting n and a starter set matrix from master to slaves can be seen when the size of matrix increases as n increases. The cost overhead also contributed by the process of sending data from slaves to master.

The next results display the computation time for generating permutation using parallel algorithm PERMUT2. The algorithm is also tested for using 12 and 60 initial starter sets.

Table 5.7: The Computation Time of PERMUT2 with 12 Initial Starter Sets (in seconds)

	n			
p	10	11	12	13
1	0.215220	2.241888	31.688313	440.066193
2	0.222532	2.605106	32.858107	451.884762
3	0.112166	1.296642	16.451070	226.298348
4	0.074612	0.861711	13.217642	163.709182
5	0.056348	0.651444	8.231780	113.114765
6	0.056547	0.647946	8.218523	112.976828
7	0.037718	0.437311	5.489789	82.685768
8	0.039397	0.433838	5.483988	82.783656
9	0.059407	0.437702	5.623919	82.995641
10	0.047827	0.448921	5.502316	77.190598

Table 5.8: The Computation Time of PERMUT2 with 60 Initial Starter Sets (in seconds)

	n			
p	10	11	12	13
1	0.215220	2.241888	31.688313	440.066193
2	0.223814	2.342861	32.876404	451.884762
3	0.111480	1.293892	16.470241	227.191016
4	0.075979	0.864197	10.980008	151.086499
5	0.056474	0.647320	8.233840	113.228971
6	0.045577	0.523354	6.586692	90.491564
7	0.038220	0.434187	5.490308	75.482394
8	0.035936	0.391076	4.942402	67.973839
9	0.030943	0.348380	4.401244	60.435723
10	0.027486	0.310298	4.077362	53.545169

PERMUT2 algorithm shows similar performance as PERMUT1. The performance of PERMUT2 for 60 initial starter sets is better than 12 initial starter sets in terms of speedup and efficiency. However PERMUT2 algorithm runs faster than PERMUT1 algorithm over p processors by comparing Tables 5.7 dan 5.8 with Tables 5.3 and 5.4 respectively. The speedup and efficiency of PERMUT2 for $10 \leq n \leq 13$ using 12 and 60 initial starter sets can be found in Tables 5.9 and 5.10.

Table 5.9: The Speedup of PERMUT2 with 12 and 60 Initial Starter Sets

	n							
	10		11		12		13	
p	12	60	12	60	12	60	12	60
1	1	1	1	1	1	1	1	1
2	0.9552	0.9616	0.8606	0.9569	0.9644	0.9639	0.9738	0.9738
3	1.9188	1.9306	1.7289	1.7327	1.9262	1.9239	1.9446	1.9369
4	2.8845	2.8326	2.6017	2.5942	2.3974	2.8860	2.6881	2.9127
5	3.8194	3.8381	3.4414	3.4633	3.8495	3.8485	3.8904	3.8865
6	3.8060	4.7221	3.4600	4.2837	3.8557	4.8110	3.8952	4.8631
7	5.7060	5.6311	5.1265	5.1634	5.7722	5.7717	5.3222	5.8301
8	5.4628	5.9890	5.1676	5.7326	5.7783	6.4115	5.3159	6.4741
9	3.6228	6.9554	5.1219	6.4352	5.6346	7.1998	5.3148	7.2816
10	4.5000	7.8302	4.9939	7.2250	5.7591	7.7718	5.7010	8.2186

Table 5.10: The Efficiency of PERMUT2 with 12 and 60 Initial Starter Sets

	n							
	10		11		12		13	
p	12	60	12	60	12	60	12	60
1	1	1	1	1	1	1	1	1
2	0.4776	0.4808	0.4303	0.4784	0.4822	0.4820	0.4869	0.4869
3	0.6396	0.6435	0.5763	0.5776	0.6421	0.6413	0.6482	0.6456
4	0.7211	0.7081	0.6504	0.6486	0.5993	0.7215	0.6720	0.7282
5	0.7639	0.7676	0.6883	0.7697	0.7699	0.7697	0.7781	0.7773
6	0.6343	0.7870	0.5767	0.7139	0.6426	0.8018	0.6492	0.6793
7	0.8151	0.8044	0.7323	0.7376	0.8246	0.8245	0.7603	0.8329
8	0.6829	0.7486	0.6460	0.7165	0.7223	0.8014	0.6645	0.8093
9	0.4025	0.7728	0.5691	0.7150	0.6261	0.8000	0.5905	0.8091
10	0.4500	0.7830	0.4994	0.6916	0.5759	0.7772	0.5701	0.8286

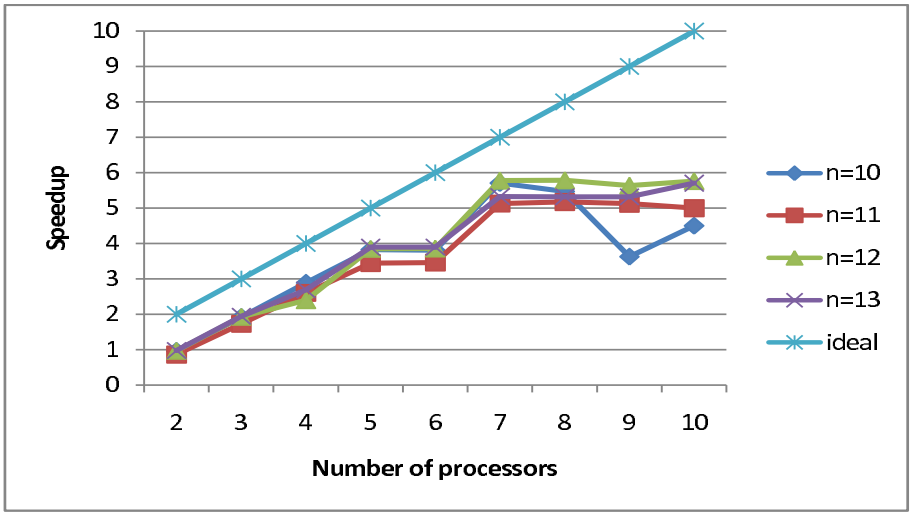

Figure 5.6: Speedup versus Number of Processors for PERMUT2 with 12 Initial Starter Sets

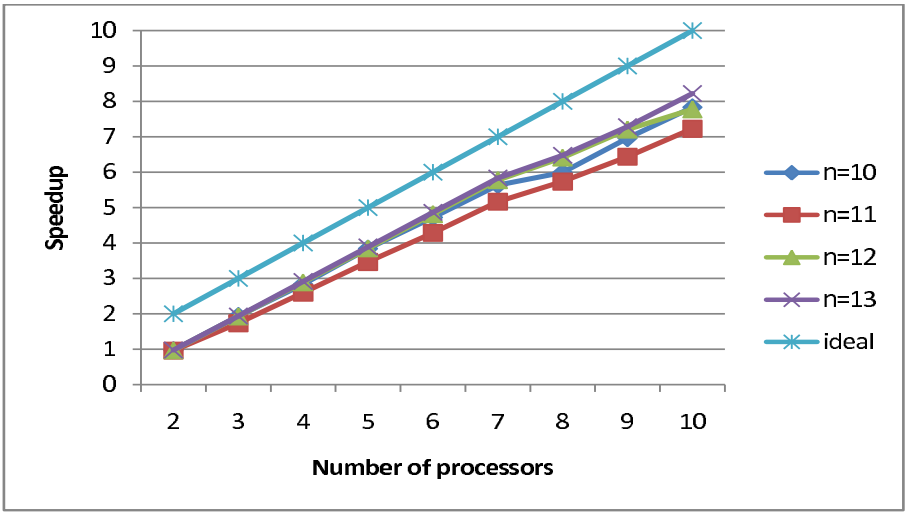

Figure 5.7: Speedup versus Number of Processors for PERMUT2 with 60 Initial Starter Sets

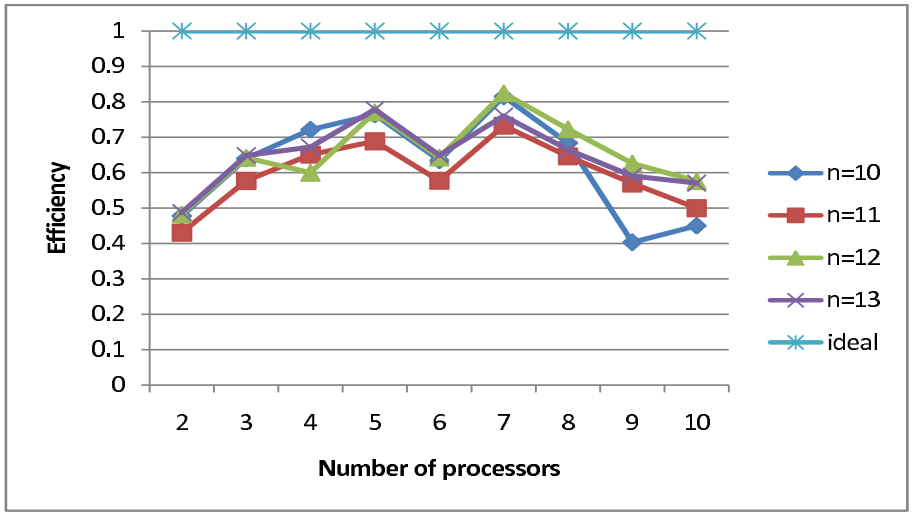

Figure 5.8: Efficiency versus Number of Processors for PERMUT2 with 12 Initial Starter Sets

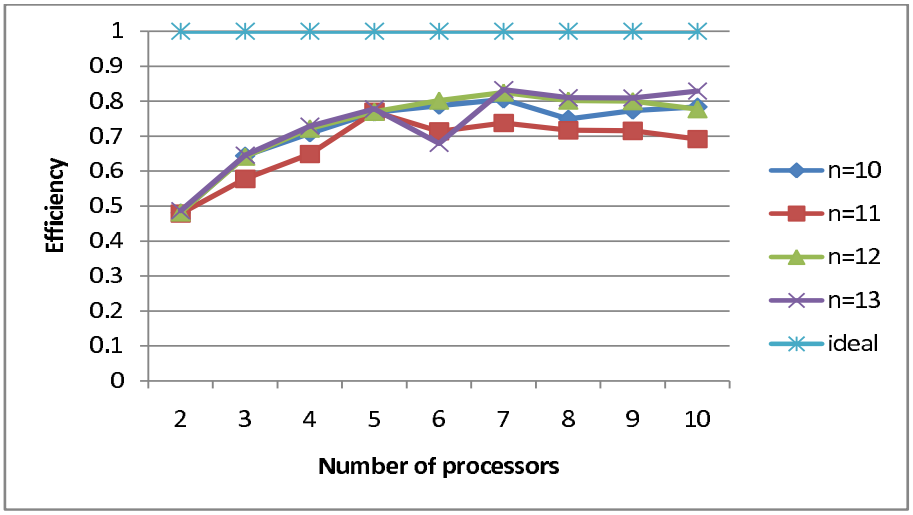

Figure 5.9: Efficiency versus Number of Processors for PERMUT2 with 60 Initial Starter Sets

Both algorithms (PERMUT1 and PERMUT2) for permutation generation show almost 80 percent efficiency when the size of the matrix of initial starter sets is change from $12 \times n$ to $60 \times n$. However the time execution for $p = 2$ is slower compared to $p = 1$ due to parallel overhead.

The results are given only for case $10 \leq n \leq 13$ which is our algorithms limitation due to permutation generation is a heavy computation. However results show the algorithm is working and applicable to run with any number of processors before it is applied for finding the determinant. The order of complexity for sequential permutation generation algorithm is $O(nn!)$ which a heavy computation or time consuming. From the numerical results, the time execution is reduced by parallelising the sequential algorithms (PERMUT1 and PERMUT2).

The parallel time complexity can generally be represented as

$$T(n,p) = O(\frac{T(n)}{p} + T_{comm}(n,p)) \approx O(\frac{T(n)}{p}) \tag{5.9}$$

where n is the problem size, p number of processors available, $T(n)$ is the time complexity of the best sequential algorithm, and $T_{comm}(n,p)$ is the overall communication overhead of a parallel time complexity (Li, 2009). Given the pseudocode PERMUT1(k) for slaves as follows:

Pseudocode PERMUT1(k)

1: **do in parallel**
2: **for** $j = 1$ to $p - 1$ **do**
3: PERMUT1(k) is performed by p_j
4: **if** $k = 2$ **then**
5: $old = a[1]$
6: **for** $i = 1$ to $n - 1$ **do**

7: $a[i] = a[i+1]$

8: **end for**

9: $a[n] = old$

10: **return**

11: **end if**

12: $k = k - 1$

13: **for** $i = n$ to k **do**

14: $old = a[k]$

15: **for** $j = k$ to $n - 1$ **do**

16: $a[i] = a[i+1]$

17: **end for**

18: $a[n] = old$

19: call PERMUT1(k)

20: **end for**

21: **end for**

For every slave p_j, the critical section is lines 13 - 20 where starter sets are generated. There is nested loop and in that loop, there is recursive call on less (k gets smaller). The initial $k = n-3$ for 12 ISS and $k = n-4$ for 60 ISS (refer Table 5.1). At line 12, the value of k is decreasing. The process will stop when $k = 2$. On the other hand, the recursion call will stop when $k = 2$ or the recursion will not be called when $n = 3$.

The order of complexity for lines 13 -20 is calculate as follows:

For any value of k for loop at lines 14-16,

$$(n - 1 - k)$$

For the outer loop at lines 13 -20,

$$(2 + [n - 1 - k])$$

Let set up the initial $k = n - 3$ for 12 ISS, at line 12, the new value k is $n - 4$.

$$\begin{aligned}&(2 + [n - 1 - (n - 4)])\\ =&(2 + [3])\end{aligned}$$

$k = n - 5$

$$\begin{aligned}&(2 + [n - 1 - (n - 5)]) \times (2 + [1])\\ =&(2 + [4]) \times (2 + [1])\end{aligned}$$

$temp = n - 6$

$$\begin{aligned}&(2 + [n - 1 - (n - 6)]) \times (2 + [4]) \times (2 + [3])\\ =&(2 + [5]) \times (2 + [4]) \times (2 + [3])\end{aligned}$$

Until when $k = 2$,

$$\begin{aligned}&(2 + [n - 1 - (2)]) \times \cdots \times (2 + [2]) \times (2 + [1])\\ =&(n - 1) \times \cdots \times (7) \times ((6) \times (5)\\ \cong&(\frac{(n-1)!}{12})\end{aligned}$$

Thus the order of complexity at lines 13-20 are $O(\frac{(n-1)!}{12})$.
From lines 4-11, double loops which has the complexity $O(n^2)$ exists. In order to generate all permutations, all starter sets need to be exploited by performing that double loops cycling process. So by multiplying n^2 to $O(\frac{(n-1)!}{12})$, it is equal to $O(\frac{nn!}{12})$.

Meanwhile for PERMUT1 with 60 ISS, the order of complexity for every slave is $O(\frac{nn!}{60})$.
Then for any initial k where $n - 1 \leq k \leq 2$, the order of complexity of our PERMUT1(k)

for every slave is

$$O(\frac{nn!}{(n-1-k)!}). \tag{5.10}$$

Technically calculation of the time complexity of overall communication overhead of a parallel algorithm is difficult. Thus, we only calculate the order of complexity of main task in our parallel algorithm for PERMUT1. For PERMUT2, its order of complexity is similar to PERMUT1 because similarities of order complexity and the task allocation to master and slaves.

The next section discusses about numerical results of parallel permutation algorithm for across the method approach.

5.6.2 Numerical Results of Across The Method Permutation Algorithm

In this section, the performance of the two parallel algorithms for generating permutation namely PERATM1 and PERATM2 is presented. The total initial starter sets are $n-1$ for these algorithms.

The time computation for PERATM1 and PERATM2 is given in Tables 5.11 and 5.12 respectively. From Tables 5.11 and 5.12, the computation time reduces until $p = 7$ for $10 \leq n \leq 13$. In general, the results also indicate that the time computation for $p > 7$ are about the same except for the case $n = 10$. At $n = 10$ and $p = 10$, the workload among the slaves are equally allocated. Therefore time computation is decreased.

Table 5.11: The Computation Time of PERATM1 (in seconds)

	n			
p	10	11	12	13
1	0.213560	2.495277	31.790697	395.085497
2	0.264305	3.369504	38.062418	573.582670
3	0.163369	1.683488	20.760161	287.268842
4	0.098515	1.240107	15.374426	191.218912
5	0.088280	1.010999	11.533463	143.593139
6	0.067034	0.677695	11.545756	143.412291
7	0.065794	0.675261	7.700118	95.817563
8	0.102532	0.674890	7.704120	95.726524
9	0.069749	0.882673	7.850826	95.983782
10	0.050604	0.814327	7.853307	95.692595

Table 5.12: The Computation Time of PERATM2 (in seconds)

	n			
p	10	11	12	13
1	0.193152	2.498813	31.789985	439.895013
2	0.244638	2.545330	35.878914	492.610659
3	0.136010	1.420710	19.548346	246.379205
4	0.083146	1.135502	13.033181	164.612632
5	0.074166	0.851266	9.778800	123.195581
6	0.056164	0.570117	8.835864	123.168945
7	0.055377	0.569366	6.531600	82.489105
8	0.069764	0.570049	6.546532	82.172228
9	0.054901	0.580490	6.542568	82.163520
10	0.032537	0.571731	6.529212	82.140822

Table 5.13: The Speedup and Efficiency of PERATM1

	n							
	10		11		12		13	
p	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency
1	1	1	1	1	1	1	1	1
2	0.8080	0.4040	0.7405	0.3702	0.8352	0.4176	0.6888	0.3444
3	1.3072	0.4357	1.4822	0.4941	1.5313	0.5104	1.3753	0.4584
4	2.1678	0.5419	2.0121	0.5030	2.0678	0.5170	2.0661	0.5165
5	2.4191	0.4838	2.4681	0.4936	2.7564	0.5513	2.7514	0.5503
6	3.1858	0.5305	3.6820	0.6137	2.7535	0.4589	2.7549	0.4592
7	3.2459	0.4637	3.6953	0.5279	4.1286	0.5898	4.1233	0.5890
8	2.0828	0.2604	3.6973	0.4622	4.1265	0.5158	4.1272	0.5159
9	3.0618	0.3402	2.8269	0.3141	4.0493	0.4499	4.1162	0.4574
10	4.2202	0.4220	3.0642	0.3064	4.0481	0.4048	4.1287	0.4129

Table 5.14: The Speedup and Efficiency of PERATM2

	n							
	10		11		12		13	
p	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency
1	1	1	1	1	1	1	1	1
2	0.7895	0.3948	0.9817	0.4909	0.8860	0.4430	0.8930	0.4465
3	1.4201	0.4734	1.7588	0.5863	1.6262	0.5421	1.7854	0.5951
4	2.3230	0.5808	2.2006	0.5502	2.4392	0.6098	2.6723	0.6681
5	2.6043	0.5209	2.9354	0.5871	3.2509	0.6502	3.5707	0.7141
6	3.4391	0.5732	4.3830	0.7305	3.5978	0.5996	3.5715	0.5953
7	3.4879	0.4983	4.3888	0.6270	4.8671	0.6953	5.3328	0.7618
8	2.7686	0.3461	4.3835	0.5479	4.8560	0.6070	5.3533	0.6692
9	3.5182	0.3909	4.3047	0.4783	4.8589	0.5399	5.3539	0.5949
10	5.9364	0.5936	4.3783	0.4378	4.8689	0.4869	5.3554	0.5355

The graphs of speedup for PERATM1 and PERATM2 are shown in Figures 5.10 and 5.11 respectively.

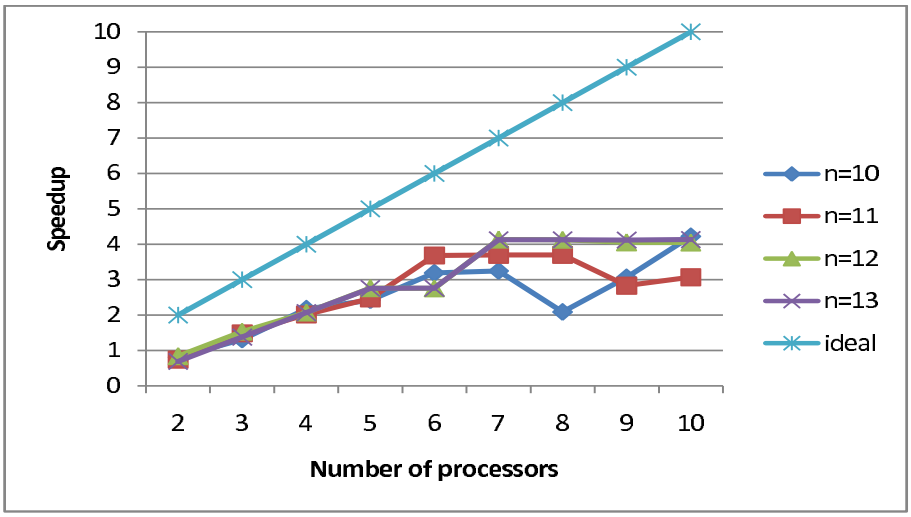

Figure 5.10: Speedup versus Number of Processors for PERATM1

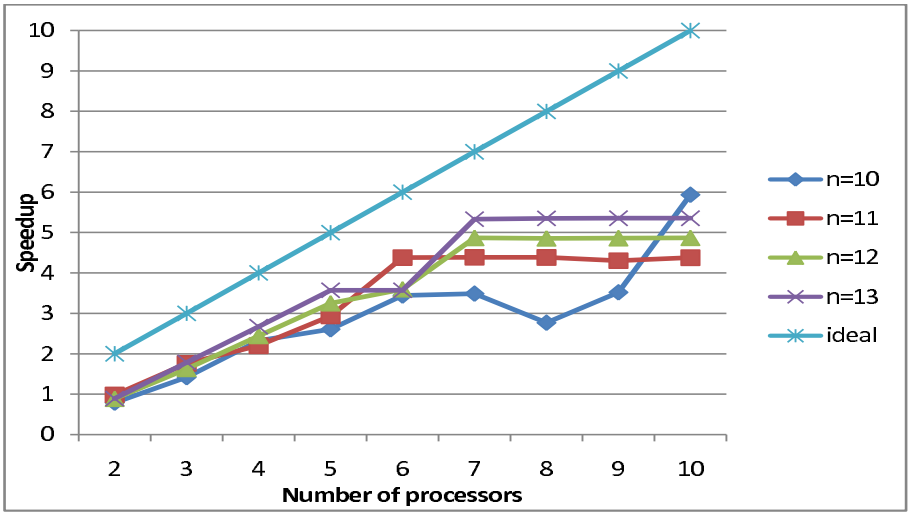

Figure 5.11: Speedup versus Number of Processors for PERATM2

The graphs of efficiency for PERATM1 and PERATM2 are display in Figures 5.12 and 5.13 respectively.

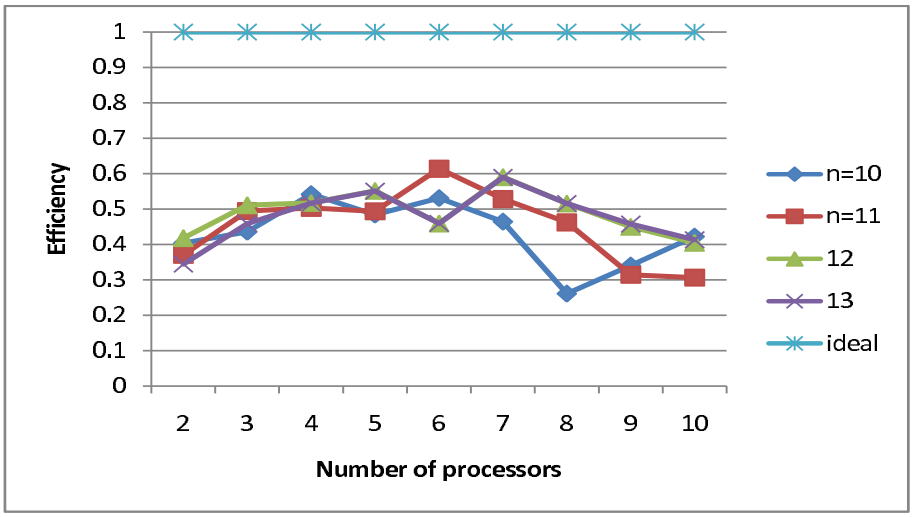

Figure 5.12: Efficiency versus Number of Processors for PERATM1

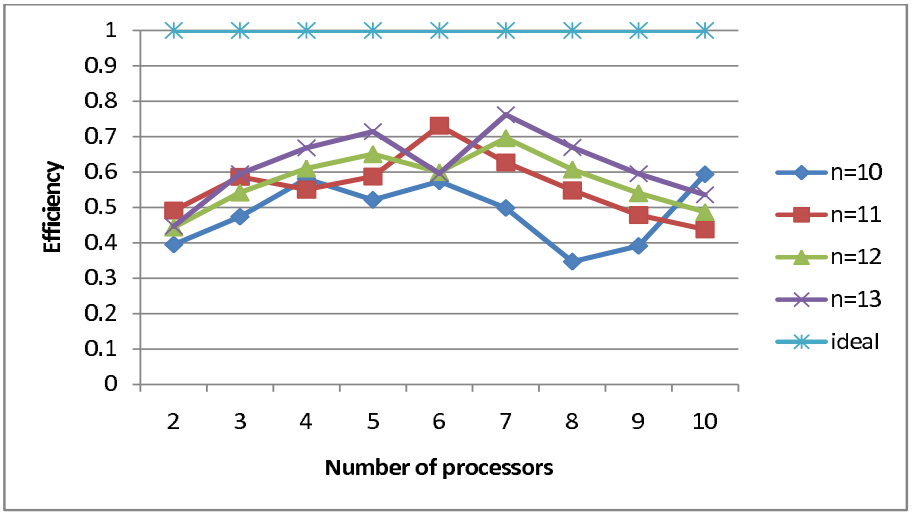

Figure 5.13: Efficiency versus Number of Processors for PERATM2

Since PERATM2 is also similar to PERATM1 where their initial starter sets is $n-1$, their performance is quite similar where the speedup of the algorithm is almost consistent from $p = 7$ until 10. From Figure 5.12 and 5.13, the processors are optimal utilized when $n-1$ is a multiple of number of slaves $(p-1)$. For example $n = 13$ with $p = 7$ where number of slave is six. The efficiency is 0.6137 and 0.7618 for PERATM1 and PERATM2 respectively by referring to Table 5.13 and 5.14 which are the highest value compare to others. These values are far from ideal efficiency which is equal to one due to imbalance

workload among slaves. In terms of adaptive, these algorithms can used only for $p \leq n$ where the number of slaves is $p-1$. In other words, these algorithms are not appropriate for $p > n$ due to the maximum value of initial starter sets is $(n-1)$. This value evenly divisible by $p-1$.

Next we analyse the order of complexity for PERATM1. Given the pseudocode of PERATM1 for every slave as follows:

Pseudocode PERATM1(k)

1: **if** $k = 3$ **then**
2: **for** $i = 1$ to n **do**
3: $old = a[1]$
4: **for** $j = 1$ to $n-1$ **do**
5: $a[j] = a[j+1]$
6: **end for**
7: $a[n] = old$
8: **end for**
9: **return**
10: **end if**
11: $temp = k - 1$
12: **for** $i = temp$ to n **do**
13: **if** $(k \neq n)$ **then**
14: $old = a[k]$
15: $a[k] = a[k+1]$
16: $a[k+1] = old$
17: **else**
18: $old = a[n]$
19: **for** $k = n$ to $temp - 1$ **do**

20: $old = a[k]$

21: $a[k] = a[k-1]$

22: **end for**

23: $a[temp] = old$

24: **end if**

25: PERATM1($temp$)

26: **end for**

The critical section is lines 12 - 25 where it represents the steps for starter sets generation. There is a nested loop and in that loop, there is recursive call on less (temp gets smaller).

The initial $temp = k = n - 2$. At line 12, the value of $temp$ decreases. The process of recursion starts at $k = n - 1$ and will stop when $k = 3$. On the other hand, the starter sets generation process will stop at $k = 2$.

The complexity of the process from 13 until 16 is a constant time, $O(1)$ complexity meanwhile the process from 18 -22 has $O(temp)$ complexity.

Let value $temp = n - 2$, the order of complexity is

$$n - 2$$

$temp = n - 3$,the order of complexity is

$$(n-3) \times (n-2)$$

until $temp = 3$

$$\begin{aligned} & 3 \times \ldots \times (n-3) \times (n-2) \\ =& \frac{(n-2)!}{2} \end{aligned}$$

The complexity of process from 13 -16, is $O((n-2)!)$ from temp = $n-2$ until 3. Meanwhile for second process $18-22$, for each value of $temp$ from 3 until $n-2$, it will be run once. Thus its complexity is $O((n-2)^2)$

Thus the order of complexity for starter sets generation is $O((n-2)!) + O((n-2)^2)$. After the starter sets are produced and stopped at $temp = 3$, the program continues for generating all permutation which lies on lines 2 - 9.

Then by multiplying n^2 to the order of complexity of the starter set generation times complexity, the results is

$$O(n^2(n-2)!) + O(n^4 - 4n^2) \cong O(\frac{nn!}{(n-1)}). \tag{5.11}$$

The Equation 5.12 is an order complexity for every slave with a single initial starter sets. Since there are $(n-1)$ initial starter sets, total order of complexity is

$$O(\frac{nn!}{(n-1)}(n-1)) = O(nn!). \tag{5.12}$$

For order of complexity for PERATM2, its order complexity similar to PERATM1.

5.7 Performance of Parallel Algorithm for Determinant Method

In this section, the performance of parallel algorithm for determinant method using generalised Sarrus Rule will be analyzed based on speedup and efficiency.

5.7.1 Numerical Results of Parallel Across The Time Determinant Algorithm

PERMUTDET1 and PERMUTDET2 programs were tested for 12 and 60 initial starter sets. The master generated and broadcasted 12 and 60 initial starter sets matrix to all slaves. The tables below display the computation times, speedup and efficiency for 12 and 60 initial starter sets where p and n represent the number of processors and the order of square matrix respectively.

(i) Numerical results of PERMUTDET1

Table 5.15: The Computation Time of PERMUTDET1 with 12 Initial Starter Sets (in seconds)

	n						
p	8	9	10	11	12	13	14
1	0.026133	0.277507	3.208710	39.756847	543.402635	7070.729986	109169.746383
2	0.028524	0.290990	3.304407	41.063474	551.681789	7931.108916	109680.683212
3	0.014386	0.145181	1.644350	20.391221	273.142838	4038.878652	61278.975939
4	0.009815	0.096612	1.096464	13.640050	182.242177	2692.438573	40899.633004
5	0.009088	0.074317	0.822519	10.217560	136.846571	1983.723957	30692.248135
6	0.008112	0.072702	0.822392	10.199061	136.580548	2019.623608	30696.165309
7	0.007737	0.069624	0.550738	6.819972	91.365417	1322.518147	20437.828378
8	0.007606	0.070480	0.550410	6.813234	91.317413	1343.806241	20452.453250
9	0.009377	0.055667	0.548802	6.823104	91.273815	1326.725195	20472.878685
10	0.009489	0.073566	0.612842	7.001980	94.416001	1323.463794	20757.794087

Table 5.16: The Computation Time of PERMUTDET1 with 60 Initial Starter Sets (in seconds)

	n						
p	8	9	10	11	12	13	14
1	0.028004	0.257683	3.279323	40.648766	543.402635	7070.441568	109169.746383
2	0.028704	0.260941	3.305567	41.054788	551.704318	7178.042383	111318.187305
3	0.014595	0.145435	1.645490	20.433106	274.029770	3974.403738	61362.405426
4	0.011912	0.098655	1.097224	13.636194	182.157679	2645.278626	40916.681567
5	0.009914	0.074915	0.823250	10.231400	136.596410	1983.584290	30685.907836
6	0.008602	0.060737	0.660942	8.198199	109.403499	1569.446268	24550.541218
7	0.007947	0.051493	0.551944	6.821716	91.135985	1325.199676	20467.813066
8	0.005037	0.046218	0.497147	6.157250	82.153446	1185.673391	18405.949398
9	0.005157	0.041877	0.459811	5.679931	73.462465	995.581149	16440.119595
10	0.008534	0.037326	0.386627	4.807537	64.135519	921.464709	15488.655460

From Table 5.15, for all n, the execution time for $p \geq 8$ were not reduced further due to uneven load balancing. Thus is because 12 starter sets are not divisible by the number of processor. Instead of that, execution time at $p = 5$ and 6 also shows similarity. To overcome unbalanced load task, the number of starter sets was changed to 60 (5×12) which followed permutation numbers.

As displayed in Table 5.16, there is a significant reduction in computational time when the total number of initial starter sets was changed from 12 to 60. The execution time is decreasing when number of processors increases. This is due to the allocation number of the rows for $60 \times n$ initial starter sets matrix among processors is evenly distributed if compares to the allocation rows for $12 \times n$ initial starter sets matrix.

The speedup and efficiency result for PERMUTDET1 can be observed from Tables 5.17 and 5.18 respectively.

Table 5.17: The Speedup of PERMUTDET1 with 12 and 60 Initial Starter Sets

	n													
	8		9		10		11		12		13		14	
p	12	60	12	60	12	60	12	60	12	60	12	60	12	60
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0.9162	0.9756	0.9537	0.9875	0.9710	0.9921	0.9681	0.9903	0.9850	0.9849	0.8915	0.9850	0.9953	0.9847
3	1.8165	1.9187	1.9114	1.7718	1.9513	1.9929	1.9497	1.9894	1.9894	1.9830	1.7507	1.7790	1.7815	1.7791
4	2.6625	2.3509	2.8723	2.6112	2.9264	2.9887	2.9147	2.9809	2.9817	2.9507	2.6261	2.6729	2.6692	2.6681
5	2.8755	2.8247	3.7341	3.4397	3.9011	3.9834	3.8910	3.9729	3.9709	3.9782	3.5644	3.5645	3.5569	3.5577
6	3.2215	3.2555	3.8170	4.2426	3.9017	4.9616	3.8978	4.9583	3.9786	4.9670	3.5010	4.5051	3.5564	4.4508
7	3.3776	3.5238	3.9858	5.0042	5.8262	5.9414	5.8295	5.9587	5.9476	5.9625	5.3464	5.3354	5.3416	5.3386
8	3.4358	5.5605	3.9374	5.5754	5.8297	6.5963	5.8352	6.6018	5.9507	6.6145	5.2617	5.9632	5.3377	5.9312
9	2.7869	5.4303	4.9851	6.1533	5.8467	7.1319	5.8268	7.1566	5.9535	7.3970	5.3295	7.1018	5.3324	6.5606
10	2.7540	3.2814	3.7722	6.9036	5.2358	8.4819	5.6778	8.4552	5.7554	8.4727	5.3426	7.6730	5.2592	7.0484

Table 5.18: The Efficiency of PERMUTDET1 with 12 and 60 Initial Starter Sets

	n													
	8		9		10		11		12		13		14	
p	12	60	12	60	12	60	12	60	12	60	12	60	12	60
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0.4851	0.4878	0.4769	0.4937	0.4855	0.4951	0.4841	0.5035	0.4925	0.4925	0.4458	0.4925	0.4977	0.4924
3	0.6055	0.6397	0.6371	0.5906	0.6504	0.6643	0.6499	0.7247	0.6631	0.6610	0.5836	0.5930	0.5938	0.5930
4	0.6656	0.5877	0.7181	0.6528	0.7316	0.7472	0.7316	0.7541	0.7454	0.7377	0.6565	0.6708	0.6673	0.5667
5	0.5751	0.5649	0.7468	0.6879	0.7802	0.7967	0.7782	0.8040	0.7942	0.7956	0.7129	0.7129	0.7114	0.7115
6	0.5369	0.5426	0.6362	0.7071	0.6503	0.8269	0.6630	0.8362	0.6631	0.8278	0.5835	0.7509	0.5927	0.7418
7	0.4825	0.5034	0.5694	0.7049	0.8323	0.8488	0.8328	0.8613	0.8497	0.8518	0.7638	0.7622	0.7631	0.7627
8	0.4298	0.6951	0.4922	0.6969	0.7287	0.8245	0.7294	0.8350	0.7438	0.8245	0.6258	0.7454	0.6672	0.7414
9	0.3096	0.6034	0.5539	0.6837	0.6496	0.7924	0.6474	0.8046	0.6615	0.8219	0.5922	0.7891	0.5925	0.7289
10	0.2754	0.3281	0.3772	0.6904	0.5236	0.8482	0.5678	0.8555	0.5755	0.8473	0.5343	0.7673	0.5259	0.7048

The graphs of the speedup for PERMUTDET1 with 12 and 60 initial starter sets can be found in Figures 5.14 and 5.15 respectively.

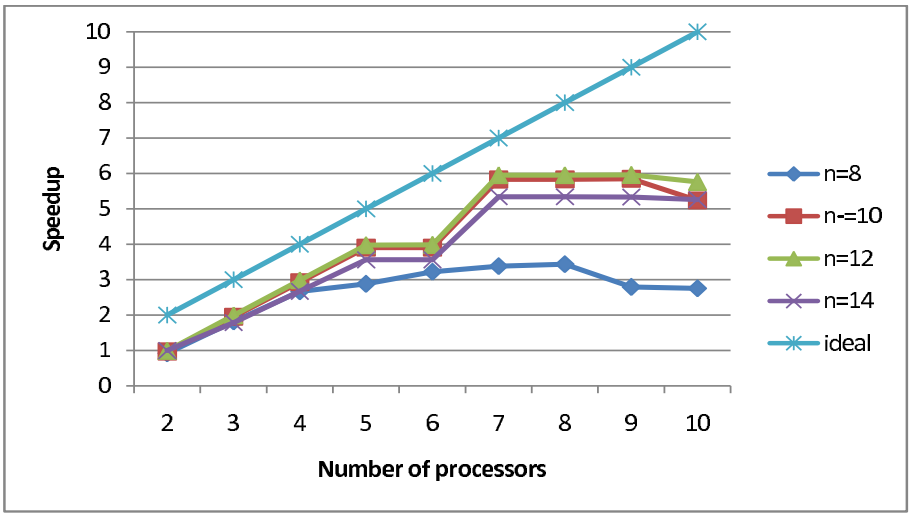

Figure 5.14: Speedup versus Number of Processors for PERMUTDET1 with 12 Initial Starter Sets

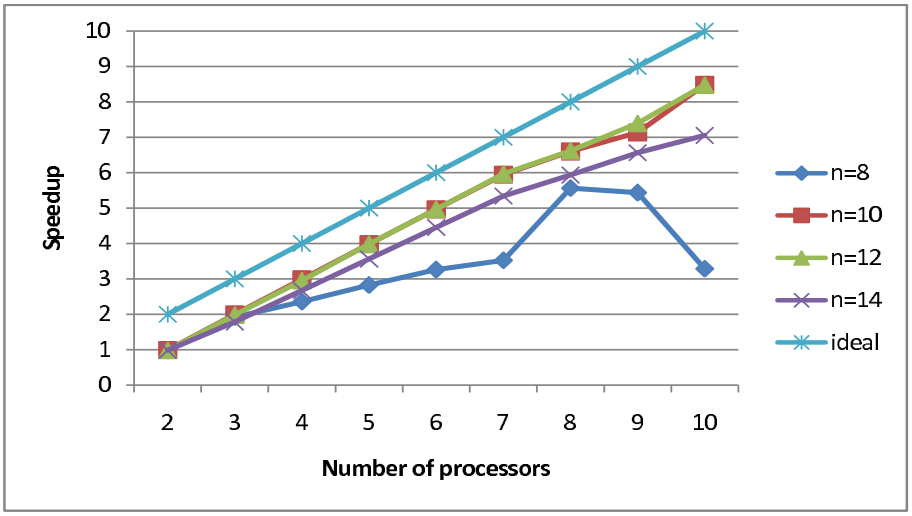

Figure 5.15: Speedup versus Number of Processors for PERMUTDET1 with 60 Initial Starter Sets

From Figure 5.14, the speedup increases until processor 7. Then the speedup degraded slowly with the increasing number of processors. This degradation might be due to the imbalance load among the slaves and also communication overhead. This imbalance load among the slaves has been improved in PERMUTDET1 with 60 initial starter sets as shown in Figure 5.15.

Meanwhile graphs of the efficiency for PERMUTDET1 with 12 and 60 initial starter sets are display in Figures 5.16, and 5.17.

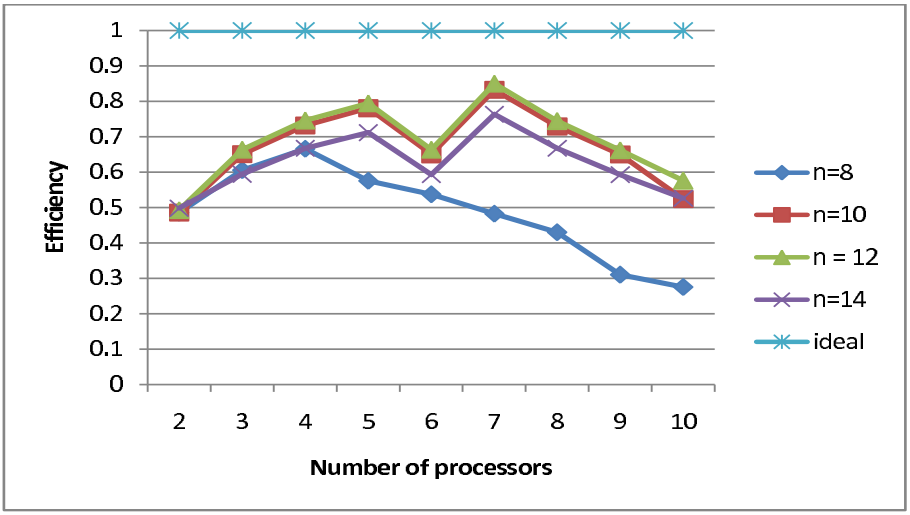

Figure 5.16: Efficiency versus Number of Processors for PERMUTDET1 with 12 Initial Starter Sets

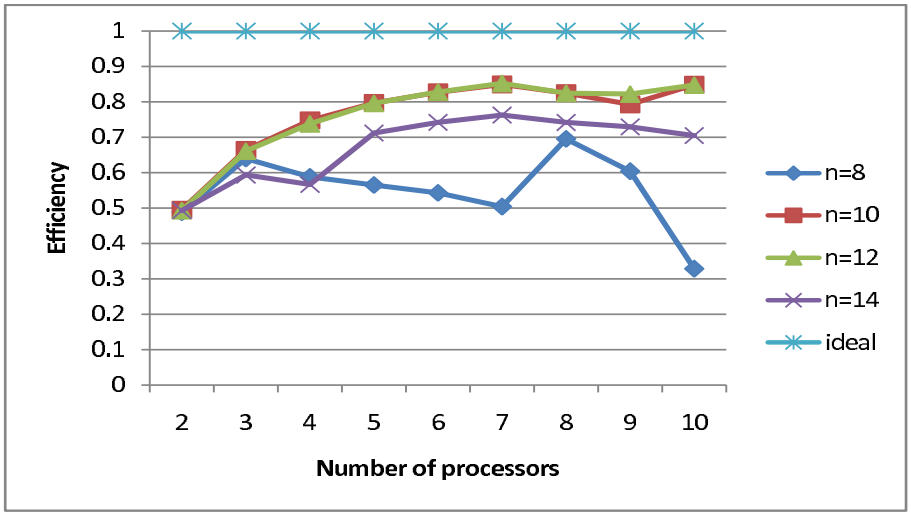

Figure 5.17: Efficiency versus Number of Processors for PERMUTDET1 with 60 Initial Starter Sets

In terms of efficiency, the results in Figures 5.16 and 5.17 indicate that seven processors is fully utilised for all sizes of matrices where an algorithm efficiency close to 0.8.

The numerical results for PERMUTDET2 algorithm are presented in the following section.

(ii) Numerical results for PERMUTDET2

The result in terms of speedup and efficiency for PERMUDET2 is almost similar to PERMUTDET1. The tables below display the computation times, speedup and efficiency for 12 and 60 initial starter sets.

Table 5.19: The Computation Time of PERMUTDET2 with 12 Initial Starter Sets (in seconds)

	n						
p	8	9	10	11	12	13	14
1	0.026554	0.245947	3.119146	38.842880	526.485917	7001.386234	107810.910401
2	0.028530	0.285689	3.293924	40.088831	535.855841	7089.665860	110146.005701
3	0.014240	0.143357	1.619442	20.011530	267.906912	3985.641360	61277.247936
4	0.011195	0.095699	1.080343	13.346751	178.662455	2659.505056	40826.217429
5	0.008968	0.071969	0.810334	10.014028	134.055068	1946.939937	30626.856467
6	0.008490	0.072265	0.810109	10.010797	133.965306	1946.603467	30593.787876
7	0.007463	0.050163	0.541241	6.681097	90.629363	1298.639396	20429.059508
8	0.007427	0.051667	0.541996	6.679235	89.394837	1298.167086	20705.704107
9	0.005608	0.050383	0.550207	6.711501	90.024354	1299.940680	20463.603330
10	0.005595	0.057094	0.561136	6.697337	89.559362	1319.955882	20455.766965

Table 5.20: The Computation Time of PERMUTDET2 with 60 Initial Starter Sets (in seconds)

	n						
p	8	9	10	11	12	13	14
1	0.026570	0.273442	3.119162	38.843362	526.485917	7646.452197	107810.910401
2	0.028466	0.288698	3.254940	40.250653	543.015538	7824.546646	111409.435212
3	0.016534	0.144576	1.629141	20.131551	271.179954	3909.183096	61285.423033
4	0.012004	0.098327	1.087568	13.433204	181.107622	2608.503037	40835.349947
5	0.009751	0.074675	0.816500	10.084417	135.839972	1954.937527	30636.984716
6	0.008687	0.060640	0.654106	8.060286	108.619628	1565.535091	24536.489139
7	0.007734	0.051229	0.546053	6.722696	90.588628	1305.490613	20427.359781
8	0.006820	0.045778	0.492247	6.054021	81.942486	1174.772299	18390.205384
9	0.008006	0.049840	0.443480	5.584887	73.513499	1045.769742	16663.769329
10	0.012394	0.039140	0.384178	4.885317	64.246603	924.671144	15136.050890

As shown in Tables 5.19 and 5.20, there is an improvement of performance in term of computation time of the program. The time computation is decreasing for 60 initial starter sets compare to the algorithm for 12 initial starter sets especially on the processors where the number of starter sets is not evenly divisible by $(p-1)$.

The speedup and efficiency result for PERMUTDET2 can be observed from Tables 5.21 and 5.22 respectively.

Table 5.21: The Speedup of PERMUTDET2 with 12 and 60 Initial Starter Sets

	n													
	8		9		10		11		12		13		14	
p	12	60	12	60	12	60	12	60	12	60	12	60	12	60
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0.9307	0.9334	0.8609	0.9471	0.9469	0.9583	0.9686	0.9650	0.9825	0.9695	0.9875	0.9772	0.9788	0.9677
3	1.8647	1.6069	1.7156	1.8913	1.9260	1.9146	1.9310	1.9295	1.9652	1.9415	1.7557	1.9560	1.7594	1.7591
4	2.3719	2.2134	2.5358	2.7809	2.8872	2.8868	2.9103	2.8918	2.9468	2.9070	2.6326	2.9313	2.6407	2.6401
5	2.9610	2.7248	3.4174	3.6618	3.8492	3.8675	3.8788	3.8518	3.9274	3.8758	3.5961	3.9113	3.5201	3.5190
6	3.1277	3.0586	3.4034	4.5092	3.8503	4.7686	3.8801	4.8191	3.9300	4.8471	3.5967	4.8842	3.5239	4.3939
7	3.5581	3.4355	4.9029	5.3376	5.7629	5.7122	5.8138	5.7779	5.8092	5.8118	5.3913	5.8571	5.2773	5.2778
8	3.5753	3.8959	4.8815	5.5550	5.7549	6.3366	5.8155	6.4161	5.8894	6.4251	5.3633	6.5089	5.2070	5.8624
9	4.7350	3.3187	4.2201	6.1658	5.6690	7.0334	5.7875	6.9551	5.8482	7.1617	5.3859	7.3118	5.2684	6.4698
10	4.7460	2.1438	4.3077	7.1176	5.5586	8.1190	5.7997	7.9510	5.8786	8.1948	5.3043	8.2694	5.2704	7.1229

Table 5.22: The Efficiency of PERMUTDET2 with 12 and 60 Initial Starter Sets

	n													
	8		9		10		11		12		13		14	
p	12	60	12	60	12	60	12	60	12	60	12	60	12	60
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0.4654	0.4667	0.4304	0.4736	0.4735	0.4792	0.4843	0.4825	0.4912	0.4848	0.4938	0.4886	0.4894	0.4838
3	0.6216	0.5356	0.5719	0.6304	0.6420	0.6382	0.6437	0.6432	0.6551	0.6472	0.5852	0.6520	0.5865	0.5864
4	0.5929	0.5534	0.6339	0.6952	0.7218	0.7217	0.7276	0.7229	0.7367	0.7268	0.6582	0.7328	0.6602	0.6600
5	0.5922	0.5450	0.6835	0.7324	0.7698	0.7735	0.7758	0.7704	0.7855	0.7752	0.7192	0.7823	0.7040	0.7038
6	0.5213	0.5098	0.5672	0.7515	0.6417	0.7948	0.6467	0.8032	0.6550	0.8079	0.5995	0.8140	0.5873	0.7323
7	0.5083	0.4908	0.7004	0.7625	0.8233	0.8160	0.8305	0.8254	0.8299	0.8302	0.7702	0.8367	0.7539	0.7540
8	0.4469	0.4869	0.6102	0.6944	0.7194	0.7921	0.7269	0.8020	0.7362	0.8031	0.6704	0.8136	0.6509	0.7328
9	0.5261	0.3687	0.4689	0.6851	0.6300	0.7815	0.6431	0.7728	0.6498	0.7957	0.5984	0.8124	0.5854	0.7189
10	0.4746	0.2144	0.4308	0.7112	0.5559	0.8119	0.5800	0.7951	0.5879	0.8195	0.5304	0.8269	0.5270	0.7123

From Tables 5.21 and 5.22, it can be observed that PERMUTDET2 with 60 initial starter sets shows some improvements in term of speedup and efficiency compare to PERMUTDET2 with 12 initial starter sets.

The graphs of the speedup for PERMUTDET2 with 12 and 60 initial starter sets can be found in Figures 5.18, and 5.19 respectively. Meanwhile the graphs of efficiency for PERMUTDET2 with 12 and 60 initial starter sets are displayed in Figures 5.20 and 5.21.

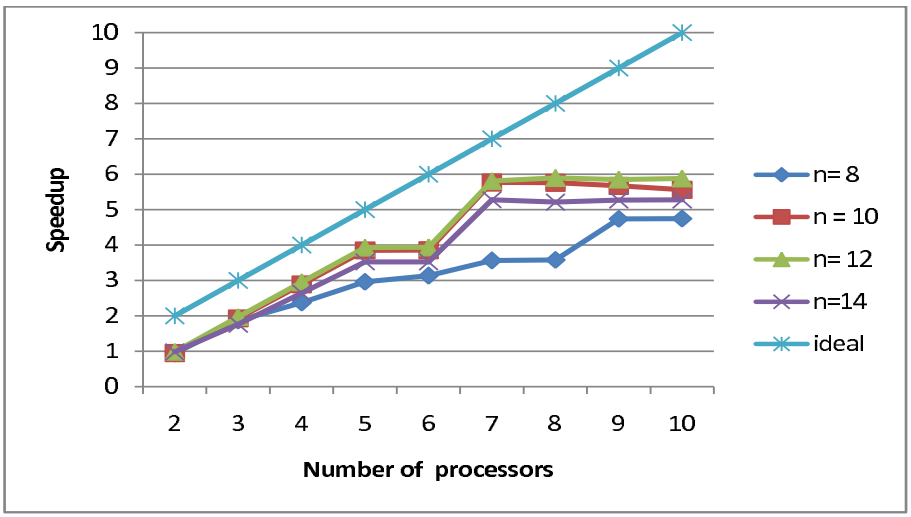

Figure 5.18: Speedup versus Number of Processors for PERMUTDET2 with 12 Initial Starter Sets

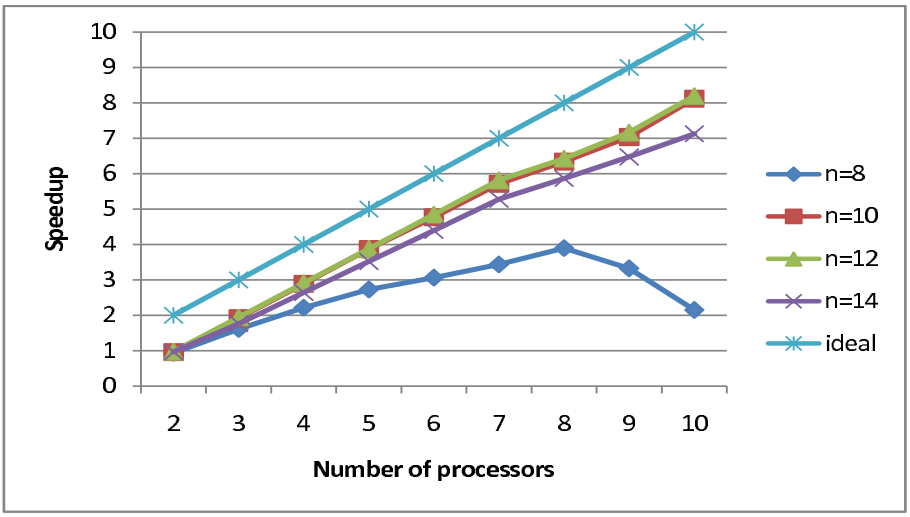

Figure 5.19: Speedup versus Number of Processors for PERMUTDET2 with 60 Initial Starter Sets

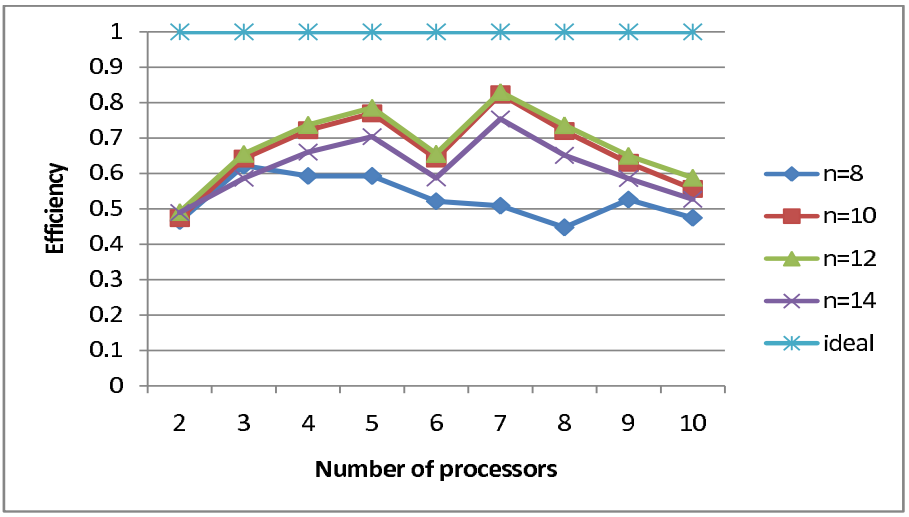

Figure 5.20: Efficiency versus Number of Processors for PERMUTDET2 with 12 Initial Starter Sets

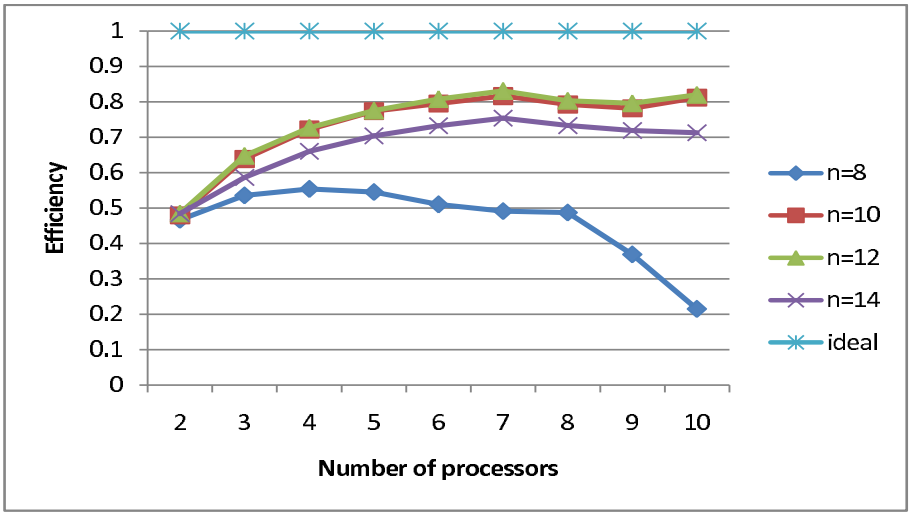

Figure 5.21: Efficiency versus Number of Processors for PERMUTDET2 with 60 Initial Starter Sets

Overall, the results shows that the performances of the programs in term of speedup and efficiency are improved when the 60 initial starter sets were employed instead of 12. The total number of processors which is fully utilized is seven due to load balancing. For the case of 12 initial starter sets program, the degradation performance is due to the imbalance workload among the slaves. Fortunately, this drawback can be overcome by changing the number of initial starter sets. Thus, that is an advantage of applying our new parallel permutation algorithm for finding the determinant where the number of starter sets easily can be changed statically.

In spite of that, the performances in term of speedup and efficiency is drops for PERMUTDET2 with 60 initial starter sets compare to 12 initial starter sets for the case $n = 8$. Cost overhead is one of the influence factor since for sequential algorithm, the time computation is small for $n = 8$ (refer Table 4.2 in page 100). From our points of view, overheads appearing in this parallel computation tests that may affect the speedup and efficiency are the communication time for sending and receiving message, imbalance workload, idle and selection of day or night time for test parallel algorithm. The temperature at night is lower than during the day which also give affect machine cooling. Meanwhile imbalance workload may cause extra computation for slaves.

The order of complexity for parallel determinant algorithm by referring to the Equation 5.11 for any k, where $n-1 \leq k \leq 2$ is

$$O(\frac{n^2 n!}{(n-1-k)!}). \tag{5.13}$$

We only enhance the parallel permutation algorithm for determining determinant by performing multiplication among element in every permutation array. The time complexity of under multiplication operation is $O(n)$. Thus the order of complexity of parallel permutation algorithm for slave is multiplies with n as given in Equation 5.14.

5.7.2 Numerical Analysis of Across The Method Determinant Algorithm

The two algorithms namely PDATM1 and PDATM2 are designed so that the algorithms are parallel in nature and suit parallel computers well. These PDATM1 and PDATM2 programs adapt only for $p-1 \leq n-1$ where $p-1$ is a number of slaves, whereas $n-1$ is a number of initial starter sets . Therefore the algorithms are not appropriate for $p > n$ because the maximum value of initial starter sets is $n-1$. The time computation for PDATM1 and PDATM2 is given in Tables 5.23 and 5.24 respectively. Then Tables 5.25 and 5.26 show the speedup and efficiency for PDATM1 and PDATM2.

From Tables 5.23 and 5.24, the computation time reduces until $p = 7$ for $7 \leq n \leq 13$. The results also indicate that the time computation for $p > 7$ are about the same except for the case $n = 10$. At $n = 10$ and $p = 10$, the workload among the slaves are equally allocated. Therefore time computation is decreased. Meanwhile for $n < 7$, the degradation shown in time computation.

Table 5.23: The Computation Time of PDATM1 (in seconds)

	n								
p	5	6	7	8	9	10	11	12	13
1	0.000240	0.000580	0.003282	0.026053	0.264444	2.999546	39.254678	498.113951	7198.568207
2	0.000367	0.000698	0.003433	0.027867	0.283830	3.235128	40.146991	538.796167	7805.359469
3	0.000402	0.000554	0.003335	0.016655	0.144472	1.804817	20.146898	293.971907	3881.664948
4	0.001992	0.002265	0.003186	0.012921	0.108775	1.076775	16.062247	196.053785	2588.536675
5	0.002327	0.002344	0.003174	0.010072	0.073325	1.078895	12.050565	147.257013	1942.145987
6	-	0.002564	0.003156	0.009710	0.073014	0.721488	8.051058	147.036512	1941.636469
7	-	-	0.003169	0.008563	0.071413	0.717673	8.036877	98.248541	1295.637155
8	-	-	-	0.004944	0.073334	0.716227	8.046728	97.646182	1302.795633
9	-	-	-	-	0.040546	0.648765	8.001682	97.563896	1298.973674
10	-	-	-	-	-	0.370078	7.994326	99.065405	1299.147333

Table 5.24: The Computation Time of PDATM2 (in seconds)

	n								
p	5	6	7	8	9	10	11	12	13
1	0.000240	0.000580	0.003282	0.026046	0.264423	2.999285	39.254678	498.113951	7198.568207
2	0.000364	0.000674	0.003347	0.028025	0.279364	3.195193	39.693149	532.166287	7714.406439
3	0.000322	0.000573	0.003444	0.015699	0.141198	1.774798	19.848664	290.195376	3493.174552
4	0.002101	0.000488	0.003168	0.011957	0.104811	1.067870	15.867711	193.562736	2572.697770
5	0.002380	0.002327	0.002948	0.009871	0.071935	1.065492	11.912885	145.392356	1930.522394
6	-	0.002702	0.003006	0.009706	0.071829	0.712793	7.956491	145.200988	1930.712630
7	-	-	0.003243	0.008020	0.071610	0.713128	7.951471	96.948137	1290.831371
8	-	-	-	0.008752	0.071611	0.720544	7.949094	97.209251	1287.881610
9	-	-	-	-	0.220509	0.813252	8.072372	96.914022	1306.757691
10	-	-	-	-	-	0.367720	7.166406	100.361621	1296.423246

Table 5.25: The Speedup and Efficiency of PDATM1

	n											
	8		9		10		11		12		13	
p	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency
1	1	1	1	1	1	1	1	1	1	1	1	1
2	1.0398	0.5199	1.0360	0.5180	1.0316	0.5158	0.9778	0.4889	0.9245	0.4622	0.9222	0.4611
3	1.7397	0.5799	2.0353	0.6784	1.8492	0.6164	1.9484	0.6495	1.6944	0.5648	1.8545	0.6182
4	2.2425	0.5606	2.7032	0.6758	3.0994	0.7748	2.4439	0.6110	2.5407	0.6352	2.7809	0.6952
5	2.8768	0.5754	4.010	0.8020	3.0934	0.6187	3.2575	0.6515	3.3828	0.6766	3.7065	0.7413
6	2.9840	0.4973	4.0272	0.6712	4.6257	0.7709	4.8757	0.8126	3.3877	0.5646	3.7075	0.6179
7	3.3837	0.4769	4.1175	0.5882	4.6503	0.6643	4.8843	0.6978	5.0699	0.7243	5.5560	0.7937
8	5.8606	0.7325	4.0096	0.5012	4.6597	0.5825	4.8783	0.6098	5.1012	0.6377	5.5255	0.6907
9	-	-	7.2521	0.8058	5.1443	0.5716	4.9058	0.5451	5.1055	0.5673	5.5417	0.6157
10	-	-	-	-	9.0181	0.9018	4.9103	0.4910	5.0281	0.5028	5.5410	0.5541

Table 5.26: The Speedup and Efficiency of PDATM2

	n											
	8		9		10		11		12		13	
p	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency
1	1	1	1	1	1	1	1	1	1	1	1	1
2	1.0339	0.5170	1.0525	0.5262	1.0445	0.5222	0.9889	0.4944	0.9360	0.4680	0.9331	0.4665
3	1.8457	0.6152	2.0825	0.6942	1.8804	0.6268	1.9777	0.6592	1.7165	0.5717	2.0607	0.6869
4	2.4233	0.6058	2.8054	0.7013	3.1253	0.7813	2.4739	0.6184	2.5734	0.6433	2.7981	0.6995
5	2.9354	0.5871	4.0876	0.8175	3.1323	0.6264	3.2951	0.6590	3.4260	0.6852	3.7288	0.7458
6	2.9853	0.4975	4.0936	0.6823	4.6821	0.7804	4.9337	0.8223	3.5305	0.5884	3.7284	0.6214
7	3.6128	0.5161	4.1061	0.5132	4.6799	0.6685	4.9368	0.7053	5.1379	0.7336	5.5767	0.7966
8	3.3107	0.4184	4.1061	0.4562	4.6318	0.5789	4.9382	0.6173	5.1241	0.6405	5.5895	0.6987
9	-	-	1.3334	0.1482	4.1038	0.4559	4.8628	0.5403	5.1397	0.5711	5.5087	0.6121
10	-	-	-	-	9.0759	0.9057	5.4776	0.5477	4.9632	0.4963	5.5526	0.5527

Figures 5.22 and 5.23 show the graphs of speedup versus the number of processors used for PDATM1 and PDATM2.

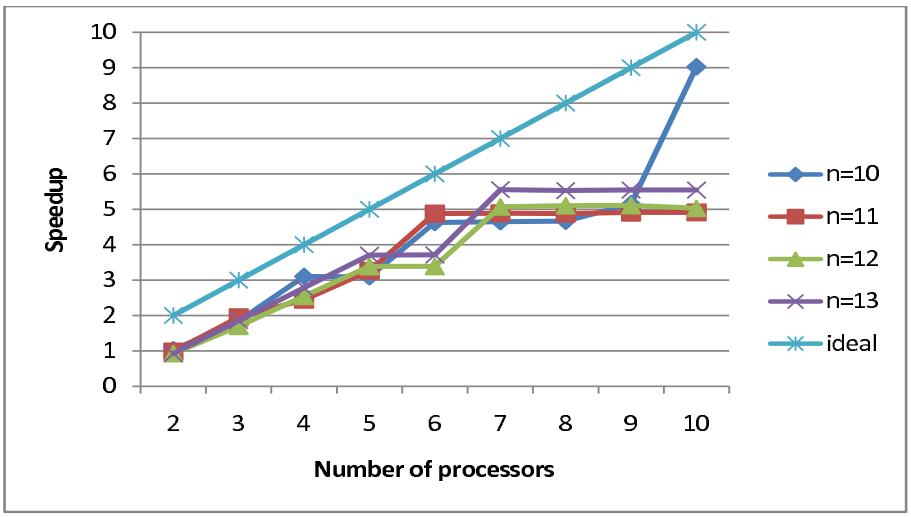

Figure 5.22: Speedup versus Number of Processors for PDATM1

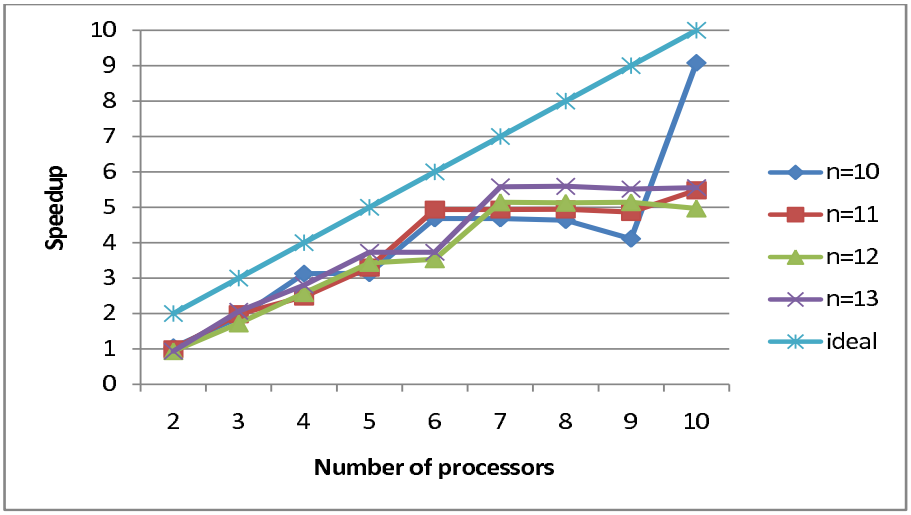

Figure 5.23: Speedup versus Number of Processors for PDATM2

Figures 5.24 and 5.25 show the graphs of efficiency versus the number of processors used for PDATM1 and PDATM2.

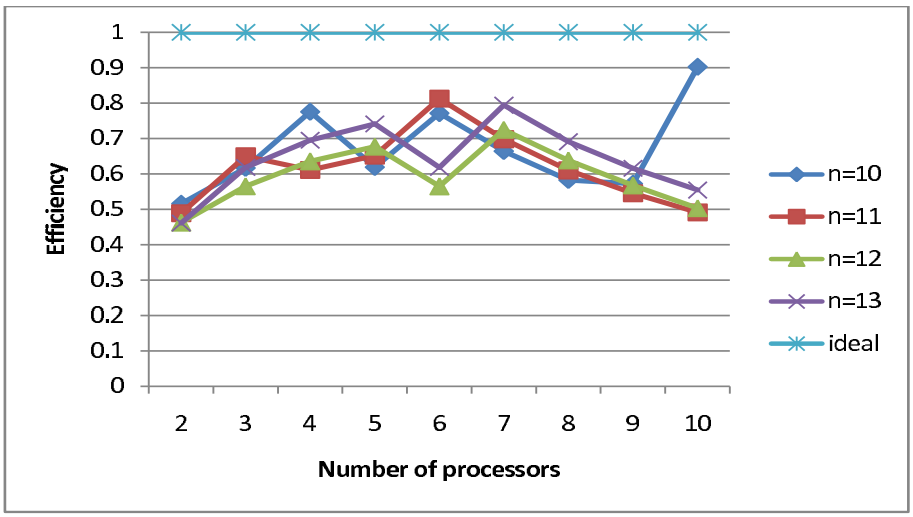

Figure 5.24: Efficiency versus Number of Processors for PDATM1

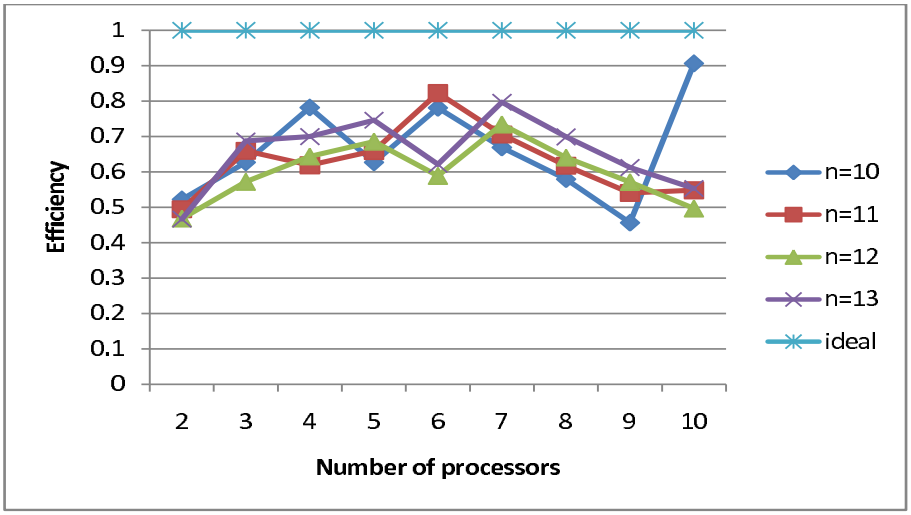

Figure 5.25: Efficiency versus Number of Processors for PDATM2

Overall, the speedup and efficiency of both programs PDATM1 and PDATM2 for finding the determinant shows the fluctuation trend when the number of processors increases due to imbalance workload except for $n = 10$. When $n = 10$, both graphs shows a remarkable performance where the efficiency is almost 90 percent due to the number of processors equal to 10. At this moment, the task among the slave are equally balance.

The order of complexity for parallel determinant algorithm by referring to Equation 5.12 for each slave is

$$O(\frac{n^2 n!}{(n-1)}). \tag{5.14}$$

We only enhance the parallel permutation algorithm for determining determinant by performing multiplication among element in every permutation array. The time complexity under multiplication operation is $O(n)$. Thus the order of complexity of parallel permutation algorithm for slave is multiply with n. Equation 5.15 represents the order of complexity for every slave with a single initial starter sets. Since there are $(n-1)$ initial starter sets, total order of complexity is

$$O(\frac{n^2 n!}{(n-1)}(n-1)) = O(n^2 n!). \tag{5.15}$$

Remark 5.7.1. *The algorithms are designed only for $p \leq n$. Thus the empty box from all tables mean that the algorithm is not suitable for $p > n$. That is the limitation of the algorithm.*

5.7.3 Comparison Between Execution Time of Sequential, Across the Time and Across the Method Program for Finding the Determinant

The sequential algorithms for finding the determinant are also tested in parallel computer using one processor. The computation times of the parallel algorithm given are performed for $p = 2$. The execution times for parallel algorithms are longer than the sequential algorithm due to parallel overhead. Please refer to Tables 5.27, 5.28 and 5.29.

Table 5.27: The Computation Time for PERMUTDET1 under Circular Strategy (in seconds)

n	PERMUTDET1	PERMUTDET1 12 ISS	PERMUTDET1 60 ISS
7	0.003033	0.003294	0.003468
8	0.026133	0.028524	0.028704
9	0.277507	0.290990	0.260941
10	3.208710	3.304407	3.305567
11	39.756847	41.063474	41.054788
12	543.402635	551.681789	551.704318
13	7070.729986	7931.108916	7178.042383
14	109169.746383	109680.683212	111318.187305

Table 5.28: The Computation Time for PERMUTDET2 under Exchange Strategy (in seconds)

n	PERMUTDET2	PERMUTDET2 12 ISS	PERMUTDET2 60 ISS
7	0.002961	0.003311	0.004443
8	0.026554	0.028530	0.028466
9	0.245947	0.285689	0.288698
10	3.119146	3.293924	3.254940
11	38.842880	40.088831	40.250653
12	526.485917	535.855841	543.015538
13	7001.386234	7089.665860	7824.546646
14	107810.910401	110146.005701	111409.435212

Table 5.29: The Computation Time for the Sequential Algorithms and Across The Method Algorithms (in seconds)

n	PERMUTDET1	PERMUTDET2	PDATM1	PDATM2
7	0.003033	0.002961	0.003433	0.003347
8	0.026133	0.026554	0.027867	0.028025
9	0.277507	0.245947	0.283830	0.279364
10	3.208710	3.119146	3.235128	3.195193
11	39.756847	38.842880	40.146991	39.693149
12	543.402635	526.485917	538.796167	532.166287
13	7070.729986	7001.386234	7805.359469	7714.406439

As shown in Table 5.29, the sequential program based on exchanging two elements (PERMUTDET2) performed better than the sequential program based on circular (PERMUTDET1). It exactly followed the same results in Section 4.6 in Chapter Five. From this result, we can conclude that new sequential algorithms (PERMUTDET1 and PERMUTDET2) perform better than Langdon, Thongchiew and cofactor expansion algorithms.

5.8 Summary

We have presented several parallel computation technique for generating permutation and determining determinant using generalised Sarrus Rule. In order to parallelize the determinant algorithm, the permutation generation algorithm is needed to be parallelized because our determinant algorithm is dominated by permutation generation. Permutation algorithms is paralellsied for across the time and across the method strategies. Each strategy is then analysed in term of computation time, speedup, efficiency and order of complexity. The main contributions of this chapter are as follows:

(i) Development of across the time parallel algorithms for permutation algorithm (PERMUT1 and PERMUT2).

(ii) Derivation of the two strategies namely PERATM1 and PERATM2 for across the method parallel algorithm to generate the permutations and some new theoretical works also developed.

(iii) Development of parallel methods for determining determinant by using parallel permutation generation algorithms in (i) and (ii).

CHAPTER SIX

CONCLUSION AND RECOMMENDATIONS

This thesis has contributed in extending the Sarrus Rule in determining the determinant using permutation. The list of contributions is as follows:

(i) New permutation generation methods based on starter sets without producing the equivalence starter sets.

Two strategies for generating starter sets are developed based on circular and exchanged two elements operations. From this strategies, three sequential algorithms for generating permutation are developed namely PERMUT1 (recursive circular algorithm), PERMUT2 (recursive exchange algorithm) and PERMUTIT3 (iterative circular algorithm).

(ii) New sequential division free method for finding determinant has been developed.

The generalization of the Sarrus Rule for finding determinant of square matrices has been made. Extending the strategies in (i), three division free sequential algorithms for finding determinant of matrices of any order namely PERMUTDET1,PERMUTDET2 and PERMUTDETIT3 have been proposed.

(iii) New parallel methods for finding determinant of square matrices using permutation

The sequential algorithms for permutation generation and finding determinant have been parallelised using across the time and across the method techniques.

(iv) In developing the above methods and algorithms, seven lemmas and ten theorems have derived.

6.1 Summary

Chapter One discusses the determinant methods which can be categorised as division free and non division free methods. The advantages and disadvantages of the methods have been studied. One of advantages of the division free methods is that they can tackle problems when the entries of matrices are represented in rational or polynomial expressions. Moreover, the division free methods can also avoid floating point error. Among division free methods, cross multiplication method or the Sarrus Rule is frequently used. However this method was designed to find the determinant of square matrices with order $n \leq 3$. Thus, this study attempts to generalise the Sarrus Rule for any order of square matrices.

In Chapter Two, some fundamental concepts for permutation, determinant and matrices, and parallel computing have been mentioned. Chapter Two also reviews the existing permutation generation methods, sequential and parallel algorithms of permutation generation methods, and division free methods for finding determinant using permutation. From the literature, we found that no research had been done to find the determinant of square matrices of any order based on cross multiplication method using permutation.

Chapter Three proposes two strategies to generate starter sets namely circular and exchange operations. These strategies will guarantee that the generated starter sets are distinct and their equivalence starter sets will not be produced. To list all $n!$ permutations, the circular and reversing operation are employed on those starter sets. The numerical results have shown that the performance of the new developed algorithms for listing all permutations is better than the existing algorithms in term of time computation and order of complexity.

The construction of a new division free method for finding the determinant was proposed in Chapter Four. The distinct $n!$ permutations obtained in Chapter Three are presented in column indices form. The permutations generated from circular permutation operation on

n elements produce main diagonal products whereas permutations generated from reverse of circular permutation operation produce secondary diagonal products. The determinant is obtained by summing up all the signed main diagonal products and signed secondary diagonal products. The numerical results have shown that the new algorithms performed better than other division free algorithms in term of time computation and order of complexity.

In Chapter Five, two parallel strategies have been introduced to parallelise the sequential method developed in Chapter Four in order to reduce the computation times. In the first strategy, the master generates initial starter sets and the broadcasts them to the slaves. Each slave then continues to generate starter sets based on the assigned initial starter sets and eventually produces the corresponding permutations. The collection of permutations produce by all slaves is the complete permutations. Each slave calculates sub-determinant and send the result to the master to compute the determinant. On the other hand, in the second strategy, the master only broadcasts the value of n to each slave. All starter sets and permutations generation are performed by each slave. Similar to the first strategy, each slave calculates sub-determinant and send it to the master to sum up the total value. The numerical results showed that the parallel methods generate permutation and compute the determinants faster than the sequential counterparts particularly when the tasks were equally allocated.

6.2 Future Work

The new sequential algorithms for generating starter sets are constructed by fixing an element in the first position in either exchange or circular operations. It would be interesting to derive new strategy by fixing an element in any position.

In this research, the allocation of initial starter sets to slaves is predetermined by using a static scheduling in parallel algorithm. To improve the computation time, it would

worthwhile to distribute the initial starter sets to each slave using dynamic scheduling.

The number of initial starter sets used in the developed parallel algorithm across the method is $n-1$. Further research should consider using different number of initial starter sets to increase the efficiency of slaves.

REFERENCES

Abeles, F.F. (2008). Dodgson condensation: The historical and mathematical development of an experimental method.*Linear Algebra and its Application*, 429, 429-438.

Abdi, H. (2007). The Eigen-decomposition: eigenvalues and eigenvectors.In: Neil Salkind (Ed.) (2007), *Encyclopedia of Measurement and Statistics*. Thousand Oaks (CA): Sage.

Akl, S. G., & Bruda, S. D. (2001). Improving a solution's quality through parallel processing. *Journal of Supercomputing*, 19, 221-233.

Akl, S. G., Meijer, H., & Stojmenovic, I. (1994). An optimal systolic algorithm for generating permutation in lexicographic order. *J. of Parallel and Distributed Computing*, 20(1), 84-91.

Akl, S. G., & Stojmenovic, I. (1992). A simple optimal systolic algorithm for generating permutations. In Albert Zomaya, *Parallel Processing Letter Parallel Computing: Paradigms and Application*(pp.639-670). London: International Thomson Computer Press.

Almasi, G. S., & Gottlieb, A. (1989). *Highly parallel computing*. Redwood City: Benjamin-Cummings publishers.

Alonso, L., & Schott, R. (1996). A parallel algorithm for the generation of a permutation and applications. *Theoretical Computer Science*, 159, 15-28.

Anderson, R. J. (1990). Parallel algorithms for generating random permutations on a shared memory machine, *ACM*, 95-102.

Anton, H. (2000). *Elementary linear algebra*(8th ed.). New York: John Wiley.

Anton, H., & Busby, R.C. (2002). *Contemporary linear algebra*. New York: John Wiley.

Aziz, A. I., Haron, N., Mehat, M., Jung, L.T., Mustapa, A. N., & Akhir, P. E. A. (2009). Solving traveling problem on cluster compute nodes. *International Journal of Computers*, 3(2), 260-269.

Bankier,J. D. (1961). The diagrammatic expansion of Determinants. *The American Mathematical Monthly*, 68(8), 788-790.

Bernstein, D. S. (2008). *Matrix mathematics*(2nd ed.). New Jersey: Princeton University Press.

Barisenko, A. A., Kalashnikov, V. V., Kulik, I. A., & Goryachev,O. E. (2008). Generation of permutations based upon factorial numbers, *IEEE*, 57-61.

Brawer, S. (1989). *Introduction to parallel programming*. Academic Press, Inc.

Bressoud, D. M., & Propp, J. (1999). How the alternating sign matrix conjecture was solved.*Notices of AMS*,46,637-646.

Brestscher, O. (2009). *Linear algebra with applications*(4th ed.). New Jersey:Prentice Hall International.

Burrage, K. (1995). *Parallel and sequential method for ordinary differential Equation*. New York: Oxford University Press Inc.

Chalmers, A. & Tidmus, J. (1996). *Practical parallel processing, an introduction to problem solving in parallel*. London : International Thomson Computer Press.

Cong, G., & Bader, D. A. (2006). An empirical analysis of parallel random permutations algorithm on SMPs. Technical Report, Georgia Institute of Technology.

Dodgson, C. L. (1866). Condensation of determinants, being a new and brief method for computing their arithmetic Values. *Proc. Roy. Soc.* Ser. A 15, 150-155.

Djamegni, C. T., & Tchuenté, M. (1997). A cost-optimal pipeline algorithm for permutation generation in lexicographic order. *J. of Parallel and Distributed Computing*, 44(2), 153-159.

Fadlallah, G., Lavoie, M., & Dessaint, L-A. (2000). Parallel computing environments

and methods. *IEEE*, 2-7.

Fike, C. T. (1975). A permutation generation method.*The Computer Journal*, 18(1), 21-22.

Gao, J., & Wang, D. (2003). Permutation generation: Two new permutation algorithms. Retrieved from http://arxiv.org/abs/cs/0306025. on September 2008.

Gentleman, W. M., & Johnson, S. C. (1974). The evaluation of determinants by expansion by minors and the general problem of substitution. *Mathematics of Computation*, 28(126), 543-548.

Gentleman, W. M., & Johnson, S. C. (1976). Analysis of algorithms, a case study: Determinants of matrices with polynomial entries. *ACM Transactions on Mathematical Software*, 2(3), 232-241.

Goldberg, D. (1991). What every computer scientist should know about floating point arithmetic,*ACM Computing Surveys*, 21(1),5-48.

Goldfinger, Y. (2008). Determinant by cofactor expansion using Cell processor. Retrieved on December, 2008 from http://mc2.umbc.edu/docs/goldfinger.pdf on 16/11/2008.

Grama, A., Gupta, A., Karypis, G. & Kumar, V. (2003). *Introduction to parallel computing*(2nd ed.). Essex: Addison-Wesley.

Griss, M. L. (1976). An efficient sparse minor expansion algorithm. *Proceedings of ACM annual conference*, Houston, Texas, United States, 429-434

Gropp, W., Lusk, E., & Skjellum, A. (1997). *Using MPI portable parallel programming with the message-passing interface*(2nd ed.). London: MIT Press.

Gupta, P., Agarwal, V., & Varshney, M. (2008). *Design and analysis of algorithms*. New Delhi: Prentice-Hall of India

Hajrizaj, D. (2009). New method to compute the determinant of a 3×3 matrix. *International J. Algebra*, 3, 211-219.

Hanly, J. R., & Koffman, E. B. (2004). *Problem solving and program design in C*

($4th$ ed.). Boston: Pearson Addison-Wesley.

Hanus, P. H. (1886). *An elementary treatise on the theory of determinants*. Boston: Gin and Company. Retrieved from http://archive.org/stream/elementarytreati00hanuuoft #page/4/mode/2up

Harville, D. A. (1997). Matrix algebra from a statistician's perspective. pp.179-208. New York : Springer-Verlag.

Hasiung, C. Y., & Mao, G. Y. (1998). *Linear algebra*. London: World Scientific Publishing.

Heap, B. R. (1963). Permutations by interchanges. *Computer J.* 6, 293-294.

Horowitz, E., Sahni, S., & Rajasekaran, S. (2008). *Computer algorithms/C++* (2nd ed.). New Jersey: Silicon Press.

Horowitz, E., & Sahni, S. (1975). On Computing the Exact Determinant of Matrices with Polynomial Entries.*Journal of the ACM*, 38-50.

Huang,T. C., Shiu, L.C., & Huang, J. H. (2001). Efficient local memory sequence generation for data parallel programs using permutation.*Journal of Systems Achitecture*, 47, 505-515.

Hussain, S. J., & Ahmed, G. (2005). A comparative study and analysis of PVM and MPI for parallel and distributed systems. *IEEE*, 183-187.

Ibrahim, H., Omar, Z., & Rohni, A. M. (2010). New algorithm for listing all permutations. *Modern Applied Science* , 4(2), 89-94.

Iqbal, K. (1995). An algorithm for computation of determinants of polynomial matrices. Proceeding of the American Control Conferences, 2536-2537.

Iyer, M. (1995). Permutation generation using matrices. *Dr.Dobbs Journal*. Retrieved from http://www.ddj.com/184409671. on December 2008.

Johnson, L. W., Riess, R. D. & Arnold, J. T. (2002). *Introduction to linear algebra* (5th ed.). New York: Addison-Wesley Publishing Co.

Jones, M. A. (20055). Determinant Algorithm generation with numlists. *Dr.Dobbs Journal*. Retrieved from http://www.ddj.com/184402006. on 14 Februrary 2010.

Kaltofen, E. (1992). On computing Determinant of matrices without divisions. *ACM*, 324-349.

Khattar, D. (2010). *The Pearson guide to complete mathematics for AIEEE.* (3th ed.). India :Pearson Education.

Knuth, D. E. (2002). *The art of computer programming*, vol.4, New York: Addison-Wesley:

Kokosiński, Z. (1990). On generation of permutations through decomposition of symmetric groups into cosets. *BIT*, 30, 583-591.

Krattenthaler, C. (1999). Advanced determinant calculus. *Sminaire Lotharingien Combin. 42*,B42q, 1-67. Retrieved from http://www.mat.univie.ac.at/ slc/wpapers/s42kratt.pdf on September 3008.

Langdon, G. G. (1967). An Algorithm for generating permutations. *Communication of ACM*, 10(5), 298-299.

Lee, H. R., & Saunder, B. D. (1995). Fraction free gaussian elimination for sparse matrices, *Journal of Symbolic Computation*, 19(5), 393-402.

Leopald, C. (2001). *Parallel and distributed computing, a survey of models, paradigm, and approaches*. New York : John Wiley & Sons.

Levitin, A. (2007). *Introduction to the design & analysis of algorithms*(2nd ed.). New York: Addison-Wesley.

Lewis, T. G., & El-Rewini, H. (1992). *Introduction to parallel computing*. New Jersey: Prentice-Hall.

Li, K. (2010). Fast and highly scalable parallel computations for fundamental matrix problem on distributed memory systems. *The Journal Supercomputing*,271-297.

Li, Y. (2009a). An explicit construction of Gauss-Jordan elimination matrix. Retrived from http://arxiv.org/abs/0907.5038. on 17.Mac.2011.

Li, Y. (2009b). An effective hybrid algorithm for computing symbolic determinants. *Applied Mathematic and Computation*, 2495-2501.

Lin, C., & Snyder, L. (2009). *Principles of parallel programming*. Boston: Pearson

Addison-Wesley.

Lin, C. (1991). Parallel permutation generation on linear array. *International Journal of Computer Mathematic*, 38, 113-121.

Lipski, W., & Warsaw, J. (1979). More on permutation generation method. *Computing*, 357-365.

Luyang, L., Hongtao, W., & Jianying, Z. (2006). Architeture singularity analysis for a class of parallel manipulators. *Proceeding of International Technology and Innovation Conference 2006*, 2003-2006.

Mahajan, M. & Vinay, V. (1997). Determinant: Combinatorics, algorithms and complexity. *Chicago Journal of Theoretical Computer Science*, 730-738.

Mohd Saman, M. Y., & Evans, D. J. (1995). Top-down for finding optimal grain size of parallel tasks.*Pertanika J. Scince& Technology*,3(2),241-259.

Muir, T. (1933). *A treatise in the theory of determinants*. Ney York: Dover Publications.

Ord-Smith, R.J. (1970). Generating of Permutation Sequences: Part 1. *The Computer Journal*, 13, 152-155.

Osborn, R. (1960). Concerning fourth-order determinants. *The American Mathematical Monthly*, 67(7), 682-683.

O'Connor, J. J., & Robertson, E. F. (1996). Matrices and determinants. Retrieved from (http://www-history.mcs.st-andrews.ac.uk/HistTopics/ on September 2008.

Pacheco, P. S. (1997). *Parallel programming with MPI*. San Francisco: Morgan Kaufmann Publisher.

Pavlovic, S. V. (1961). On the generalisation of the Sarrus's Rule. *Mathematika I Fizika*, No.54,pp.19-23. Retrieved from http://pefmath2.etf.bg.ac.yu/files/40/54.pdf. on December, 2008.

Peng, T., Jian, M., & Dong-Ma, Z. (1999). Application of the simulated annealing algorithm to the combinatorial optimisation problem with permutation property:

An investigation of generation mechanism. *European Journal of Operational Research* , 81-94.

Perry, W. L. (1988). *Elementary linear algebra*. New York: McGraw Hill Inc.

Quinn, M. J. (2004). *Parallel programming in C with MPI and OpenMP*. New York : Mc Graw Hill.

Reek, K. A. (1998). *Pointers on C*. Massachusetts: Addison-Wesley.

Reffgen, A. (2003). The determinant in finite and infinite-dimensional vector spaces(Unpublished master dissertation). Matematiska institutionen, Lund University, Sweden.

Rezaee, H., & Rezaifar,O. (2007). A new approach for finding the determinant of matrices. *Applied Mathematics and Computation*,188, 1445-1454.

Rice, A., & Torrence, E. (2006). Lewis Carrol's condensation methods for evaluating determinants. Retrieved from http://www.MAA.ORG/MATHHORIZONS. 12-15.

Rolfe, T. J. (2008). The assignment problem: Exploring paralleism(1). *Bulletin of ACM SIG on Computer Science Education*.

Rote, G. (2001). Division-free algorithms for the determinant and the Praffian: Algebraic and combinatorial approaches. *Computational Discrete Mathematics*, 119-135.

Sasaki, T., & Kanada, Y. (1981). Parallelism in algebraic computation and parallel algorithms for symbolic linear systems. *Proceeding of the AMC on Symbolic and Algebraic Computation*, 160-167.

Sasaki, T., & Murao, H.(1982). Efficient Gaussian elimination method for symbolic determinants and linear systems. *ACM Trans. Math. Software* , 8/3, 277-289.

Scott, R. F. (2009). *A Treatise of the theory of determinants*. BiblioBazaar.

Schneider, H., & Barker, G. P. (1989). *Matrices and linear algebra*. New York: Dover publications inc.

Scneider, M. D., Steeg, M. & Young, H. F. (1982). *Linear algebra*. New York: Macmillan Publishing Co.

Sedgewick, R. (1977). Permutation generation methods. *Computing Surveys*, 9(2), 137-164.

Sedgewick, R. (2002). Permutation generation methods. Talks Dagstuhl Workshop on Data Structures, Wadern, Germany, February, 2002. Retrieved from http://www.cs.princeton.edu/ rs/talks/perms.pdf. on December, 2008.

Sengupta, R. (1997). Cancellation is exponentially powerful for computing the determinant. *Information Processing Letters*,62, 177–181.

Shin, D. W. (2002). The permutation algorithm for non-sparse matrix determinant in symbolic computation. *Proceeding on the 15th CISL winter workshop*, Japan. Retrieved from http://icat.snu.ac.kr:3333/ww/pdf/ww200210/pdf. on November 2008.

Simon, C. P., & Blume, L. E. (1994). Mathematics for economists. Chapter 7. W. W. Norton & Company: New York.

Smit, J. (1979). New recursive minor expansion algorithmd, a presentation om a comparable context. Lecture Notes In Computer Science, *Proceedings of the International Symposiumon on Symbolic and Algebraic Computation*, 72, 74-87.

Soltys, M. (2002). Berkowitz's algorithm and clow sequences. *The electronic Journal of Linear Algebra*, Vol.9, 42-54.

Standish, T. A. (1997). *Data structures, algorithms & software principles in C*. New York : Addison-Wesley Publishing.

Stojmenovic, I. (2006). Listing combinatoric objects in parallel. *The International Journal og Parallel, Emergent and Distributed Systems*, 21(2), 127-146.

Teimoori, H., Bayat, M., Amiri, A. & Sarijloo, E. (2005). A new parallel algorithm for evaluating the determinant of a matrix of order n[PowerPoint slides]. Retrieved from http://www.math.tu-berlin.de/EuroComb05/Talks/Poster. on December 2008,

Thongchiew, K. (2007). A computerize algorithm for generating permutation and it's application in determining a determinant. *Proc.of World Academy of Science,*

Engineering and Technology, 21 , 178-183.

Tsay, J. C., & Lee, W. P. (1994). An systolic design for generating permutations in lexicographic order. *Parallel Computing*, 20(3), 353-361.

Umeda, Y., & Sasaki, T. (2006). Computing determinants of rational functions. *AGM SIGSAM Bulletin*, 40(1), 2-8.

Vein, R., & Dale, P.(1999). *Determinants and their applications in mathematical physics*. New York: Springer Verlag.

Vieira, R. S. (2010). A new Algorithm for determinant evaluation-the reduction method. Retrieved from http://arxiv.org/ PS_cache/ arxiv/ pdf/ 1012/ 1012.1790v2.pdf. on 8 March 2011.

Viktorov, O. V. (2007). Permutation generation algorithm. *Asian Journal of Information Technology*, 6(9), 956-957.

Wells, M. B. (1961). Generations of permutations by transposition. *Math Comp.*, 15,192 -195.

Wilde, C. (1988). *Linear algebra*. Massachusetts:Addison-Wesley Publishing Co.

Wilkinson, B., & Allen, M. (2005). *Parallel programming. techniques and application using networked workstations and parallel computers*(2nd ed.). New Jersey : Pearson Prentice-Hall.

Zaks, S. (1984). A new algorithm for generation of permutations. *BIT*, 24, 196-204.

Appendix A

Sequential Permutation Program

A. **PERMUT1 program**

```
#include <stdio.h>
#include <sys/timeb.h>
#include <time.h>
int num[20];
int n;
void PERMUT1(int *num,int k,int n);
void rightRotate(int *num,int k);
void print(int *num);

void main ()
{ int i;
struct timeb time_before,time_after;
long total_time_taken;
printf(" enter the size of the element =");
scanf("%d", &n);
ftime(&time_before);
for ( i=1; i<=n; i++){
num[i]= i ;}
 PERMUT1(num,n-1,n);
ftime(&time_after);
/*Calculate the time difference in milliseconds */
total_time_taken = (time_after.time  - time_before.time) *
1000 +(time_after.millitm- time_before.millitm);
printf ("\n Total time of program execution (
in miliseconds)= %4d \n", total_time_taken);
}
void rightRotate(int *num,int i)
{
 int old,k;
 for(k = i; k < n; ++k){
 old = num[k];
 num[k] = num[k+1];
 num[k+1] = old;}
}
void PERMUT1 (int *num, int k, int n)
{
    int   temp,  i, old;
if (k == 1)
{ for(i = 1; i<=n; ++i){
 rightRotate(num,1);
 print(num);
}
       return ;
     }
temp = k-1;
    for (i = n; i >= temp; i--)
        { rightRotate (num,temp);
        PERMUT1(num,temp,n);
```

```
        }
}
void print(int *num)
{ int j;
for(j= 1; j<=n; ++j)
printf("%d", num[j]);
printf("\n");
for(j= 1; j<=n; ++j)
printf("%d", num[n-j+1]);
printf("\n");
}
```

B. **PERMUT2 program**

```
#include <stdio.h>
#include <time.h>
#define MAX 50
int num[MAX];
int n;
#define maxCols 50
int a[maxCols];
int n;
void print(int n);
void move( int k, int j,int n);
void rightRotate(int n);
void PERMUT2(int temp, int n);
int main (int argc, char* argv[])
{
  int n,i;
  struct timeb time_before,time_after;
long total_time_taken;
printf(" enter the size of the element =");
scanf("%d", &n);
ftime(&time_before);
for( i=1;i<= n;i++){
  a[i] =i;}
PERMUT2(n-1,n);
ftime(&time_after);
/*Calculate the time difference in milliseconds */
total_time_taken = (time_after.time  - time_before.time) *
1000 +(time_after.millitm- time_before.millitm);
printf ("\n Total time of program execution (
in miliseconds)= %4d \n", total_time_taken);
}
void print(int n)
{ int j;
for(j= 1; j<=n; ++j)
printf("%d", a[j]);
printf("\n");
for(j= 1; j<=n; ++j)
printf("%d", a[n-j+1]);
printf("\n");
}
void move(int k, int j, int n)
{
 int t,old;
 if (k!= n){
t = a[k];
```

```
a[k] = a[k+1];
a[k+1] = t;}
 else {
         old = a[n];
for(k = n; k> j; k--){
a[k] = a[k-1];}
a[j] = old;
 }
}
void rightRotate(int n)
{
 int old,k;
old = a[1];
for(k = 1; k<n; ++k){
a[k] = a[k+1];}
        a[n] = old;
}
void PERMUT2(int k, int n)
{   int i,temp;
if( k == 2){
         for(i = 1; i<=n; ++i){
rightRotate(n);
print(n);}
return;
}
temp = k-1;
for(i = temp; i<= n ; i++){
move(i,temp,n);
PERMUT2(temp,n);
}
}
```

C. **PERMUTIT3 program**

```
#include <stdio.h>
#include <sys/timeb.h>
#include <time.h>
#define MAX 20
int num[MAX];
int n;
void starter(int temp, int n);
void print(int n);
int main (int argc , char* argv[])
{    int i;
struct timeb time_before,time_after;
double total_time_taken;
printf(" enter the size of the element=");
scanf("%d",&n);
ftime(&time_before);
for ( i=1; i<=n; i++){
num[i]= i ;}
starter(3,n);
ftime(&time_after);
/*Calculate the time difference in milliseconds */
total_time_taken = (time_after.time  - time_before.time) *
1000 +(time_after.millitm- time_before.millitm);
printf ("\n Total time of program execution
 (in miliseconds)= %f \n", total_time_taken);
```

```
}
 void PERMUTIT3(int h, int n)
 {    int k,i, temp;
     k=h;
    while(k>2){
print(n);
      k=h;
      while(k>2){
     temp = num[1];
        for(i=1; i<k; i++)
{num[i]=num[i+1];}
        num[k] = temp;
        if(k==2|| num[k] != k) break;
        k--;
       }
    }
  }

void print(int n)
{ int j;
for(j= 1; j<=n; ++j)
printf("%d", num[j]);
    printf("\n");
for(j= 1; j<=n; ++j)
printf("%d", num[n-j+1]);
    printf("\n");
}
```

Appendix B

Sequential Determinant Program

A. **PERMUTDET1 program**

```
#include<stdio.h>
#include <stdlib.h>
#include <sys/timeb.h>
#include <time.h>
#define MAX 20
int  a[MAX][MAX];
double sumBothDiag[MAX];
int num[20];
int n;
void initiate(int *num);
void PERMUTDET1(int *num,int k, int n);
void rightRotate(int *num,int n);
double deter(int *num,double  *sumBothDiag);
void productDiagonal(int *num, double *sumBothDiag);
int sign(int *num);
void print ();
int main (int argc, char* argv[])
{ int i,j;
struct timeb time_before,time_after;
double total_time_taken;
printf(" enter the size of the element=");
    scanf("%d" ,&n);
for(i=1; i<= n; ++i)
            for(j=1; j<= n; ++j)
               a[i][j] = rand() % 10;
       for(i=1; i<= n; ++i)
        {
             for(j=1; j<= n; ++j)
                printf("%3d", a[i][j]);
            printf("\n");
        }
if ( n <= 2 )
{
if (n ==2)
printf("( DETERMINANT OF THE MATRIX = %f",
a[1][1] * a[2][2] - a[1][2] * a[2][1]);
else    printf( "DETERMINANT OF THE MATRIX = %f", a[1][1]);
}
else {
ftime(&time_before);
initiate(num);
 PERMUTDET1(num,n-1,n);
ftime(&time_after);
/*Calculate the time difference in milliseconds */
total_time_taken = (time_after.time  - time_before.time) *
1000 +(time_after.millitm- time_before.millitm);
printf ("\n Total time of program execution
(in miliseconds)= %f \n", total_time_taken);}
```

```
}
void initiate(int *num)
{ int i;
 for (i=1; i<=n; i++)
 {  num[i]= i ;}
}
void rightRotate(int *num,int i)
{
 int old,k;
 old = num[i];
 for(k = i; k<n; ++k){
 num[k] = num[k+1];
 }
 num[n] = old;
}
double deter(int *num,double *sumBothDiag)
{
int i;
for(i = 1; i<=n; ++i){
 rightRotate(num,1);
productDiagonal(num,sumBothDiag);
}
return 0;
}
void PERMUTDET1(int *num,int k, int n)
{
    int     i,temp;
if (k == 2)
{ deter(num,sumBothDiag);

       return ;
     }
temp = k-1;
    for (i = n; i >= temp; i--)
        {
rightRotate (num,temp);
PERMUTDET1(num,temp, n);
        }
}
int sign(int *num)
{
int g,h,l;
for(g=1,l=1;g<n; g++)
for (h=g+1;h<=n; h++)
{
if(num[h]<num[g])
l*=-1;}
return (l);
}
void productDiagonal(int *num, double *sumBothDiag)
{
   int j;
   int p =1;
  double s, prodMainDiag[MAX],prodSecDiag[MAX];
   int k =1;
  sumBothDiag[0] = 0;
  if(n % 4 == 0 || n%4 == 1){
p *=sign(num)  ;
```

```
for(j= 1,s=1; j<=n; j++){
s *= a[j][num[j]];}
prodMainDiag[k] =s*p;
for(j= 1,s=1; j<=n; j++){
s *= a[j][num[n-j+1]];}
prodSecDiag[k] =s*p;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);}
  else {
p *=sign(num)  ;
for(j= 1,s=1; j<=n; j++){
s *= a[j][num[j]];}
prodMainDiag[k] =s*p;
for(j= 1,s=1; j<=n; j++){;
s *= a[j][num[n-j+1]];}
prodSecDiag[k] =(-1)*s*p;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);}
}
void print ()
{
printf("determinant of A= %f", sumBothDiag[1]);
printf("\n");
}
```

B. **PERMUTDET2 program**

```
#include<stdio.h>
#include <stdlib.h>
#include <sys/timeb.h>
#include <time.h>
int  a[50][50];
double sumBothDiag[50];
int num[50];
int n;
void initiate(int *num);
void PERMUTDET2(int *num,int k, int n);
void move(int *num, int k, int j, int n);
void move2(int *num, int k, int n);
double deter(int *num, double  *sumBothDiag);
void productDiagonal(int *num, double *sumBothDiag);
int sign(int *num);
void print ();

int main (int argc, char* argv[])
{ int i,j;
struct timeb time_before,time_after;
double total_time_taken;
printf(" enter the size of the element=");
    scanf("%d" ,&n);
for(i=1; i<= n; ++i)
            for(j=1; j<= n; ++j)
               a[i][j] = rand() % 10;
       for(i=1; i<= n; ++i)
        {
             for(j=1; j<= n; ++j)
                printf("%3d", a[i][j]);
            printf("\n");
        }
if ( n <= 2 )
```

```
{
if (n ==2)
printf("( DETERMINANT OF THE MATRIX = %f",
a[1][1] * a[2][2] - a[1][2] * a[2][1]);
else    printf( "DETERMINANT OF THE MATRIX = %f", a[1][1]);
}
else {
ftime(&time_before);
initiate(num);
 PERMUTDET2(num,n-1,n);
ftime(&time_after);
/*Calculate the time difference in milliseconds */
total_time_taken = (time_after.time  - time_before.time) *
1000 +(time_after.millitm- time_before.millitm);
printf ("\n Total time of program execution
(in miliseconds)= %f \n", total_time_taken);}
}
void initiate(int *num)
{ int i;
 for (i=1; i<=n; i++)
 {  num[i]= i ;}
}
void move(int *num, int k, int j, int n)
{
 int t,old;
       // printf("k = %d\n",k);
// printf("j= %d\n",j);
 if (k!= n){
t = num[k];
num[k] = num[k+1];
num[k+1] = t;}
 else {
      old = num[n];
 for(k = n; k>j; k--){
 num[k] = num[k-1];}
 num[j] =old;
 }
}
void rightRotate(int n)
{
 int old,k;
old = num[1];
for(k = 1; k<n; ++k){
num[k] = num[k+1];}
         num[n] = old;}
void PERMUTDET2(int *num,int k, int n)
{
    int     i, temp;
if (k == 2)
{ for(i = 1; i<=n; ++i){
rightRotate(n);
productDiagonal(num,sumBothDiag);
}
       return ;
     }
temp = k-1;
for(i = temp; i<= n; i++){
move(num,i,temp,n);
```

```
PERMUTDET2(num,temp,n);
}
}
int sign(int *num)
{
int g,h,l;
for(g=1,l=1;g<n; g++)
for (h=g+1;h<=n; h++)
{
if(num[h]<num[g])
l*=-1;}
return (l);
}
void productDiagonal(int *num, double *sumBothDiag)
{  int j;
   int p =1;
   double s;
   double prodMainDiag[50],prodSecDiag[50];
   int k =1;
  sumBothDiag[0] = 0;
  if(n % 4 == 0 || n%4 == 1){
p *=sign(num)  ;
for(j= 1,s=1; j<=n; j++){
s *= a[j][num[j]];}
prodMainDiag[k] =s*p;
for(j= 1,s=1; j<=n; j++){
s *= a[j][num[n-j+1]];}
prodSecDiag[k] =s*p;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);}
  else {
p *=sign(num)  ;
for(j= 1,s=1; j<=n; j++){
s *= a[j][num[j]];}
prodMainDiag[k] =s*p;
for(j= 1,s=1; j<=n; j++){;
s *= a[j][num[n-j+1]];}
prodSecDiag[k] =(-1)*s*p;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);}
}
void print ()
{printf("determinant of A= %f", sumBothDiag[1]);
printf("\n");}
```

C. **PERMUTITDET3 program**

```
#include<stdio.h>
#include <stdlib.h>
#include <sys/timeb.h>
#include <time.h>
#define MAX 50
int num[MAX];
int n;
int  a[50][50];
double sumBothDiag[50];
void PERMUTITDET3(int n);
void print();
double deter(int *num,double *sumBothDiag);
void productDiagonal(int *num, double *sumBothDiag);
```

```
int sign(int *num);
int main (int argc , char* argv[])
{    int i,j;
struct timeb time_before,time_after;
double total_time_taken;
printf(" enter the size of the element=");
scanf("%d",&n);
for(i=1; i<= n; ++i)
            for(j=1; j<= n; ++j)
               a[i][j] = rand() % 10;
       for(i=1; i<= n; ++i)
        {
             for(j=1; j<= n; ++j)
                printf("%3d", a[i][j]);
            printf("\n");
        }
ftime(&time_before);
for ( i=1; i<=n; i++){
num[i]= i ;}
PERMUTITDET3(n);
print ();
ftime(&time_after);
/*Calculate the time difference in milliseconds */
total_time_taken = (time_after.time  - time_before.time) *
1000 +(time_after.millitm- time_before.millitm);
printf ("\n Total time of program execution
(in miliseconds)= %f \n", total_time_taken);}
void rightRotate(int n)
{int old,k;
 for(k = 1; k<n; ++k){
 old = num[k];
 num[k] = num[k+1];
 num[k+1] = old;}
}
void PERMUTITDET3(int n)
 {    int k,i, temp;
     k=n;
    while(k>2){
productDiagonal(num,sumBothDiag);
      k=n;
      while(k>2){
        temp = num[1];
        for(i=1; i<k; i++)
{num[i]=num[i+1];}
        num[k] = temp;
        if(k==2|| num[k] != k) break;
        k--;
      }
    }
  }
int sign(int *num)
{
int g,h,l;
for(g=1,l=1;g<n; g++)
for (h=g+1;h<=n; h++)
{ if(num[h]<num[g])
l*=-1;}
return (l);}
```

```
void productDiagonal(int *num, double *sumBothDiag)
{   int j;
   int p =1;
   double s;
   double prodMainDiag[50],prodSecDiag[50];
   int k =1;
  sumBothDiag[0] = 0;
  if(n % 4 == 0 || n%4 == 1){
p *=sign(num)  ;
for(j= 1,s=1; j<=n; j++){
s *= a[j][num[j]];}
prodMainDiag[k] =s*p;
for(j= 1,s=1; j<=n; j++){
s *= a[j][num[n-j+1]];}
prodSecDiag[k] =s*p;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);}
  else {
p *=sign(num)  ;
for(j= 1,s=1; j<=n; j++){
s *= a[j][num[j]];}
prodMainDiag[k] =s*p;
for(j= 1,s=1; j<=n; j++){;
s *= a[j][num[n-j+1]];}
prodSecDiag[k] =(-1)*s*p;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);}
}
void print ()
{
printf("determinant of A= %f", sumBothDiag[1]);
printf("\n");
}
```

Appendix C

Parallel Permutation Program

A. **PERMUT1 program**

```
#include<stdio.h>
#include <mpi.h>
#include <stdlib.h>
#define maxRows 30
#define maxCols 10
int num[maxCols],AA[maxRows][maxCols];
int AAnew[maxRows][maxCols];
int cols;
int p;
int tag = 0,R=0, T=0; NB;
int pid;
void matrixPermute(int *num,int k,int cols);
void matrixPermute2 (int *num, int k, int cols, int d);
void rightRotate(int *num,int i);
void print(int *num);
void print2(int *num);
int factorial(int cols );

int main (int argc, char* argv[])
{   int i,j;
int myID;
int  d , N, rg, rg1, ed, rg2;
double starttime, endtime, elapsed;

MPI_Status status;

/* MPI initialization */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &myID);

if (myID == 0) // master process
{
printf(" enter the size of the element/cols =");
     scanf ("%d", &cols);
starttime = MPI_Wtime();
for (i=1; i<= cols; i++){
num[i]= i ;}

matrixPermute(num,cols-1,cols);
printf("numbers of rows = %d\n", R);
NB = (int)(R) /(p-1) ; /* number of block */
  printf("numbers of block = %d\n", NB);
/* send AA  to other processes*/

//printf("**master sending AA to all**\n");
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&R, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
            MPI_Bcast(&AA, (R+1)*maxCols, MPI_INT, 0, MPI_COMM_WORLD);
            N = factorial(cols)/(2*R);

//////Get result from slaves//////

for (d=1; d<= p-1; d++)  {
MPI_Recv(&T, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
// MPI_Recv(&AAnew, (maxRows)*maxCols, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            pid=status.MPI_SOURCE;

            printf("T = %d*\n", T);
         rg = (int )R /(p-1);
rg1 = R %(p-1);
rg2 = N* rg;
            ed = 2*rg2;

/* for (i=1; i<=rg2; i++){
for (j=1; j<= cols; j++)

       printf("%d", AAnew[i][j]);

for (j=1; j<= cols; j++)
               printf("%d", AAnew[i][cols-j +1]);
   printf("\n");
}


if(rg1 != 0){

while (pid <= rg1){

for (i = rg2+1; i<= ed ;i++){

for (j=1; j<= cols; j++)

       printf("%d", AAnew[i][j]);

for (j=1; j<= cols; j++)
               printf("%d", AAnew[i][cols-j +1]);
   printf("\n");
}

}
}*/







}
free(num);
free(AAnew);
free(AA);
} // end master  process

//////////////////////SLAVE PROCESS ////////////////////////////////////////////////////////////////////////////////
```

```
else {
/*receive A using broadcast*/
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&R, 1, MPI_INT, 0, MPI_COMM_WORLD);

MPI_Bcast(&AA, (R+1)*maxCols, MPI_INT, 0,  MPI_COMM_WORLD);


// calculate range of rows to be processed
for(d = myID; d<= R; d+= p-1){
for (j = 1; j <= cols; j++){
num[j] = AA[d][j];}
matrixPermute2(num,cols-3,cols,d);
} // end d

MPI_Send(&T, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD);
// MPI_Send(&AAnew, (maxRows)*maxCols, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD);

free(num);
free(AAnew);
free(AA);
} // end slaves process


if (myID == 0)
   {
    endtime = MPI_Wtime();
    elapsed = endtime-starttime;
    printf("\n\nParallel Time %f seconds\n",elapsed);
   }

MPI_Finalize();
return 0;

}

void rightRotate(int *num,int i)
{
 int old,k;
  old = num[i];
for(k = i; k<cols; ++k){
num[k] = num[k+1];}
        num[cols] = old;
}

void matrixPermute (int *num, int k, int cols)
{
    int     i,temp;
if (k == cols-3)
{ print(num);
       return ;
     }
temp = k-1;
    for (i = cols; i >= temp; i--)
        {
rightRotate (num,temp);
matrixPermute (num,temp,cols);
```

```
        }
}
void print(int *num)
{ int j;
R= R+1;
for(j= 1; j<= cols; ++j){
 //printf("%d",num[j]);
AA[R][j]=num[j];
}
}
void matrixPermute2 (int *num, int k, int cols, int d)
{
    int     i,temp;
if (k == 2) {
for(i = cols; i>= 1; i--){
rightRotate (num,1);
print2(num);
}
       return ;
     }
temp = k-1;
    for (i = cols; i >= temp; i--)
        {
rightRotate (num,temp);
matrixPermute2 (num,temp,cols,d);
        }
}
void print2(int *num)
{ int j;
T= T+1;
for(j= 1; j<= cols; ++j){
// AAnew[T][j]=num[j];
}
}

int factorial(int cols )
{ int i=1,current=1;
while (current<= cols)
  { i *= current;
    current++;
   } return (i);
}
```

B. **PERATM2 program**

```
 #include<stdio.h>
#include <mpi.h>
#include <stdlib.h>
#define maxRows 3000
#define maxCols 50
int num[maxCols],AA[maxRows][maxCols];
int AAnew[maxRows][maxCols];
int cols;
int p;
int tag = 0,R=0,T=0, NB;
int pid;
void print(int *num, int cols);
```

```
void print2(int *num, int cols);
void move(int *num, int k, int j, int cols);
void rightRotate(int *num,int cols);
void per(int *num,int k, int cols);
void per2(int *num,int k, int cols, int d);
int main (int argc, char* argv[])
{
   int i,j;
int myID;
int  d;
double starttime, endtime, elapsed;

MPI_Status status;

/* MPI initialization */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &myID);

if (myID == 0) // master process
{
printf(" enter the size of the element/cols =");
scanf("%d", &cols);
starttime = MPI_Wtime();
for( i=1;i<= cols;i++){
  num[i] =i;}
per(num,cols-1,cols);
printf("numbers of rows = %d\n", R);
NB = (int)(R) /(p-1) ; /* number of block */
  printf("numbers of block = %d\n", NB);
/* send AA  to other processes*/

//printf("**master sending AA to all**\n");
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&R, 1, MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Bcast(&AA, (R+1)*maxCols, MPI_INT, 0, MPI_COMM_WORLD);


//////Get result from slaves//////

for (d=1; d<= p-1; d++)  {
// printf("master receiving  AA from %d *\n", d);
MPI_Recv(&T, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
// MPI_Recv(&AAnew, maxRows*maxCols, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            pid=status.MPI_SOURCE;
            printf("T = %d *\n",T);
/* for (i=1; i<= T; i++){
for (j=1; j<= cols; j++)
       printf("%d", AAnew[i][j]);
for (j=1; j<= cols; j++)
               printf("%d", AAnew[i][cols-j +1]);
   printf("\n");
}*/


}
free(num);
free(AAnew);
```

```
free(AA);

} // end master

//////////////////////SLAVE PROCESS ////////////////////////////////////////////////////////////////////////////////

else {
/*receive A using broadcast*/
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&R, 1, MPI_INT, 0, MPI_COMM_WORLD);
// NB = (int)(R) /(p-1) ; /* number of block */

MPI_Bcast(&AA, (R+1)*maxCols, MPI_INT, 0,  MPI_COMM_WORLD);


// calculate range of rows to be processed

for(d = myID; d<= R; d+= p-1){
for (j = 1; j <= cols; j++){
num[j] = AA[d][j];}
per2(num,cols-3, cols, d);
} // end d

        MPI_Send(&T, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD);
// MPI_Send(&AAnew, maxRows*maxCols, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD);

free(num);
free(AAnew);
free(AA);
} // end slaves process
if (myID == 0)
   {
    endtime = MPI_Wtime();
    elapsed = endtime-starttime;
    printf("\n\nParallel Time %f seconds\n",elapsed);
   }

MPI_Finalize();
return 0;

}

void move(int *num, int k, int j, int cols)
{
 int t,old;
if(k != cols){
t = num[k];
num[k] = num[k+1];
num[k+1] = t;}
else
{    old = num[cols];
for(k = cols; k>j; k--){
num[k] = num[k-1];}
num[j] = old;
}
}
void rightRotate(int *num,int cols)
{
```

```
 int old,k;
 old = num[1];
for(k = 1; k<cols; ++k){
num[k] = num[k+1];}
        num[cols] = old;
}

void per(int *num,int k, int cols)
{   int i,temp;
if( k == cols-3){
print(num,cols);

return;
}
temp = k-1;
for(i = temp; i<= cols; i++){
move(num,i,temp, cols);
per(num,temp,cols);
}
}

void print(int *num, int cols)
{ int j;
R= R+1;
for(j= 1; j<= cols; ++j){
// printf("%d",num[j]);
AA[R][j]=num[j];
}
}

void print2(int *num, int cols)
{ int j;
T= T+1;
// for(j= 1; j<= cols; ++j){
// AAnew[T][j]=num[j];
// printf("\n newAA[%d][%d]= %d", T,j,AAnew[T][j]);
// }

}

void per2(int *num,int k, int cols, int d)
{   int i, temp,old;
if( k == 2){
for(i = 1; i<=cols; i++){
old = num[1];
for(k = 1; k<cols; ++k){
num[k] = num[k+1];}
        num[cols] = old;
print2(num,cols);
   }
return;
}
temp = k-1;
for(i = temp; i<= cols; i++){
move(num,i,temp, cols);
per2(num,temp,cols,d);
}
}
```

C. **PERATM1 program**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#define maxRows 16
#define maxCols 16
int num[maxCols];
int AA[maxRows][maxCols];
int cols;
int myID,p, R= 0;
int tag = 0;
void initiate(int *num);
void print(int *num,int k, int d);
void move(int *num, int k, int j,int cols);
void move2(int *num, int k, int i, int cols);
void per(int *num,int temp, int cols, int d);
void rightRotate(int *num,int cols);

int main(int argc,char* argv[])
{ int myID,pmyID;
    int N,i,d,j;
double starttime, endtime, elapsed;
MPI_Status status;
/* MPI initialization */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &myID);
   N = (cols-1)/ p-1;
if (myID == 0) // master process
{
printf(" enter the size of the element =");
scanf("%d", &cols);
starttime = MPI_Wtime();
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
for (d= 1; d<= p-1; d++)  {
// printf("master receiving  AA from %d *\n", d);
MPI_Recv(&R, 1, MPI_INT, d, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
// MPI_Recv(&AA, (R+1)*maxCols, MPI_INT, d, MPI_ANY_TAG,
 MPI_COMM_WORLD, &status);
//MPI_ANY_SOURCE ditukar oleh d
printf(" R = %d *\n ", R);
// pmyID=status.MPI_SOURCE;
/* for (i=1; i<= R; i++){
for (j=1; j<= cols; j++)
       printf("%d", AA[i][j]);
for (j=1; j<= cols; j++)
               printf("%d", AA[i][cols-j +1]);
   printf("\n");
}*/

}
free(AA);
}
else{
```

```
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
for(d= myID; d< cols; d += p-1){
for (i=1; i<=cols; i++)
{  num[i]= i ;}
move2(num,d,2, cols);
per(num,cols-1,cols,d);
}
MPI_Send(&R, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD);
// MPI_Send(&AA, (R+1)*maxCols, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD);
free(AA);
}
if (myID == 0)
   {
    endtime = MPI_Wtime();
    elapsed = endtime-starttime;
    printf("\n\nParallel Time %f seconds\n",elapsed);
   }
MPI_Finalize();
return 0;
}
void move(int *num, int k, int j, int cols)
{
 int t,old;
if(k != cols){
t = num[k];
num[k] = num[k+1];
num[k+1] = t;}
else
{    old = num[cols];
for(k = cols; k>j; --k){
old = num[k];
num[k] = num[k-1];}
num[j] = old;
}
}
void move2(int *num, int k, int i, int cols)
{
 int t,old;

 if (k == 1){
 t= num[i];
 num[i] = num[cols-1];
 num[cols-1] = t;
 old = t;
 old =num[cols-1];
 num[cols-1] = num[cols];
 num[cols] = old;
 }
 else {
t = num[k];
num[k] = num[i];
num[i] = t;}
}
void per(int *num, int t, int cols, int d)
{   int i,k,j,old,temp;
if( t == 3){
for(k = 0; k< cols; k++){
old = num[1];
```

```
for(j = 1; j<cols; ++j){
num[j] = num[j+1];}
        num[cols] = old;
print(num,k,d);
   }
return;
}
    temp = t-1;
for(i = temp; i<= cols; i++){
move(num,i,temp, cols);
per(num,temp,cols, d);
}
}
 void print(int *num, int k, int d)
{ int j;
R=R+1;
// for(j= 1; j<= cols; ++j){
// AA[R][j]=num[j];
// printf("\n AA[%d][%d] = %d",d+R,j,AA[d+R][j]);
// }
}
```

D. **PERATM2 program**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/timeb.h>
#include <mpi.h>
#define maxCols 11
#define maxRows 1000
int num[maxCols];
int AA[maxRows][maxCols];
int cols;
int myID,p, R= 0, source;
int tag = 0;
void initiate(int *num);
void print(int *num, int k, int d);
void move(int *num, int k, int j, int cols);
void move2(int *num, int k, int cols);
void per(int *num,int temp, int cols, int d);
void rightRotate(int *num,int cols);
int factorial(int cols );
int main(int argc,char* argv[])
{
int myID,pid;
    int N,i,d,j, rg, rg1, ed, rg2,rg3;
    struct timeb time_before, time_after;
    double starttime, endtime, elapsed;
MPI_Status status;
/* MPI initialization */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &myID);
  // N = (cols-1)/ p-1;
if (myID == 0) // master process
{
printf(" enter the size of the element =");
```

```
scanf("%d", &cols);
ftime(&time_before); // = MPI_Wtime();
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
for (d= 1; d<= p-1; d++)  {
// source = d;
printf("master receiving  AA from %d *\n", d);
// MPI_Recv(&R, 1, MPI_INT, source, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
MPI_Recv(&AA, maxRows*maxCols, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
// printf(" R = %d *\n ", R);
pid=status.MPI_SOURCE;
         rg = N/(cols-1);
rg1 = (int)(cols-1)/(p-1);
rg2 = rg*rg1;
rg3 = (cols-1) % (p-1);
            ed = rg2*(rg3+1);
for (i=1; i<=rg2; i++){
for (j=1; j<= cols; j++)
// printf("\n AAnew[%d][%d] = %d",i,j,AAnew[i][j]);
      printf("%d", AA[i][j]);
// printf("\n");
for (j=1; j<= cols; j++)
              printf("%d", AA[i][cols-j +1]);
  printf("\n");
}

}

free(AA);
}
else{
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
for(d= myID; d< cols; d += p-1){
// initiate(num);
for (i=1; i<=cols; i++)
{  num[i]= i ;}
move2(num,d,cols-1);
per(num,2,cols,d);
}
// MPI_Send(&R, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD);
MPI_Send(&AA, maxRows*maxCols, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD);
}
if (myID == 0)
   {
   ftime(&time_after); // = MPI_Wtime();
    elapsed = endtime-starttime;
    printf("\n\nParallel Time %f seconds\n",elapsed);
   }
MPI_Finalize();
return 0;
}
void move(int *num, int k, int j, int cols)
{
 int t,old;
 if (k!= cols){
t = num[k];
num[k] = num[k+1];
num[k+1] = t;}
```

```
 else {
      old = num[cols];
 for(k = cols; k>2; k--){
 num[k] = num[k-1];}
num[2] = old;
 }
}
void move2(int *num, int k, int j)
{
 int t,old;
 if (k == 1){
 t= num[k+1];
 num[k+1] = num[cols-1];
 num[cols-1] = t;
 old = t;
 old =num[cols-1];
 num[cols-1] = num[cols];
 num[cols] = old;
 }
 else {
t = num[k];
num[k] = num[j];
num[j] = t;}
}
void per(int *num,int temp, int cols, int d)
{   int i,k, old,j;
if( temp == cols-2){
for(k = 1; k<= cols; k++){
old = num[1];
for(j = 1; j<cols; ++j){
num[j] = num[j+1];}
        num[cols] = old;
print(num,k,d);
   }
return;
}
for(i = temp+1; i<= cols ; i++){
move(num,i,temp+1,cols);
per(num,temp+1,cols,d);
}
}
void print(int *num, int k, int d)
{ int j;
R=R+1;
for(j= 1; j<= cols; ++j){
AA[R][j]=num[j];
// printf("\n AA[%d][%d] = %d",d+R,j,AA[d+R][j]);
}
}
```

Appendix D

Parallel Determinant Program

A. **PERMUTDET1 program**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#define maxRows 40
#define maxCols 15
int num[maxCols], AA[maxRows][maxCols];
double det[maxRows];
double newDet[maxRows];
int cols;
int p;
int tag = 0, R=0, NB;
int pid;
int  a[maxCols][maxCols];
double sumBothDiag[maxRows];
void print(int *num);
void rightRotate(int *num,int i);
void PERMUTDET(int *num, int k, int cols);
void PERMUTDET1(int *num, int k, int cols, int d);
void productDiagonal(int *num,int k,
double *sumBothDiag,int d);
int sign();

int main (int argc, char* argv[])
{ int i,j;
int myID;
int d;
double sum;
double starttime, endtime, elapsed;
MPI_Status status;
/* MPI initialization */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &myID);
if (myID == 0) // master process
{
printf(" enter the size of the element/cols =");
     scanf ("%d", &cols);
 /* for(i= 1; i<=cols; i++)
for(j=1; j<= cols; j++){
printf("a[%d][%d]=",i,j);
scanf("%f",&a[i][j]);
}*/
for(i=1; i<= cols; ++i)
            for(j=1; j<= cols; ++j)
               a[i][j] = rand() % 10;
       for(i=1; i<= cols; ++i)
        {
```

```
            for(j=1; j<= cols; ++j)
               printf("%3d", a[i][j]);
            printf("\n");
        }
starttime = MPI_Wtime();
if ( cols <= 2 )
{
if (cols ==2)
printf("( DETERMINANT OF THE MATRIX =
%f",a[1][1] * a[2][2] - a[1][2] * a[2][1]);
else
printf( "DETERMINANT OF THE MATRIX = %f", a[1][1]);
}
else {

for (i=1; i<= cols; i++){
num[i]= i ;}
PERMUTDET(num,cols-1,cols);
printf("numbers of rows = %d\n", R);
NB = (int)(R) /(p-1) ; /* number of block */
printf("numbers of block = %d\n", NB);
/* send AA  to other processes*/

//printf("**master sending AA to all**\n");
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&a, maxCols*maxCols, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&R, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&AA, (R+1)*maxCols, MPI_INT, 0, MPI_COMM_WORLD);
//////Get result from slaves//////
sum=0;
for (d=1; d<= p-1; d++)  {
MPI_Recv(&det, maxRows, MPI_DOUBLE, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            pid=status.MPI_SOURCE;
for (i=1; i<=R; i++){
sum += det[i];}
}
printf ("\ndeterminant value of the  matrix = %f",sum);
}
free(num);
    free(det);
free(AA);
free(a);
} // end master process
//////////////////////SLAVE PROCESS //////////////////
else {
/*receive A using broadcast*/
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&a, maxCols*maxCols, MPI_INT, 0, MPI_COMM_WORLD);

MPI_Bcast(&R, 1, MPI_INT, 0, MPI_COMM_WORLD);
// NB = (int)(R) /(p-1) ; /* number of block */

MPI_Bcast(&AA, (R+1)*maxCols, MPI_INT, 0,  MPI_COMM_WORLD);
// calculate range of rows to be processed

for(d = myID; d<= R; d+= p-1){   // start d
for (j = 1; j <= cols; j++){
```

```
num[j] = AA[d][j]; }
PERMUTDET1(num,cols-3,cols,d);
} // end d
MPI_Send(&det, maxRows, MPI_DOUBLE, 0, MPI_ANY_TAG,
MPI_COMM_WORLD);
free(num);
free(AA);
free(a);
free(det);
free(newDet);
free(sumBothDiag);
} // end slaves process
if (myID == 0)
   {
endtime = MPI_Wtime();
elapsed = endtime-starttime;
 printf("\n\nParallel Time %f seconds\n",elapsed);
   }
MPI_Finalize();
return 0;
}
void rightRotate(int *num,int i)
{
 int old,k;
 old = num[i];
 for( k = i; k< cols; ++k){
 num[k] = num[k+1];}
num[cols] = old;
}
void PERMUTDET(int *num, int k, int cols)
{
    int     i,temp;
if (k == cols-3)
{ print(num);
       return ;
     }
temp = k-1;
    for (i = cols; i >= temp; i--)
        {
rightRotate (num,temp);
matrixPermute (num,temp,cols);
        }
}
void PERMUTDET1 (int *num, int k, int cols, int d)
{   int i, temp;
if( k == 2){
for(i = cols; i>=1; i--){
rightRotate(num,1);
productDiagonal(num, i,sumBothDiag, d);
  }
for(i = cols; i>=1; i--){
            det[d] += sumBothDiag[i];
}
return;
}
temp = k-1;
    for (i = cols; i >= temp; i--)
        {
```

```
rightRotate (num,temp);
PERMUTDET1 (num,temp,cols,d);
        }
}
int sign()
{
int g,h,l;
for(g=1,l=1;g<cols; g++)
for (h=g+1;h<=cols; h++)
{
if(num[h]<num[g])
l*=-1;}
return (l);
}
void productDiagonal(int *num, int k,
double *sumBothDiag, int d)
{
 int j;
int t =1;
double s;
double prodMainDiag[maxRows],prodSecDiag[maxRows];
  sumBothDiag[k] = 0;
  if(cols % 4 == 0 || cols %4 == 1){
t *=sign(num)  ;
//printf("\nt= %d",t);
for(j= 1,s=1; j<=cols; j++){
s *= a[j][num[j]];}
prodMainDiag[k] =s*t;
//printf("\nprodMainDiag[%d] = %f",d, prodMainDiag[d]);
for(j= 1,s=1; j<=cols; j++){
s *= a[j][num[cols-j+1]];}
prodSecDiag[k] =s*t;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);
  }
  else {
t *=sign(num)  ;
for(j= 1,s=1; j<=cols; j++){
s *= a[j][num[j]];}
prodMainDiag[k] =s*t;
for(j= 1,s=1; j<=cols; j++){;
s *= a[j][num[cols-j+1]];}
prodSecDiag[k] =(-1)*s*t;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);
  }
}
void print(int *num)
{ int j;
R= R+1;
for(j= 1; j<= cols; ++j){
// printf("%d",num[j]);
AA[R][j]=num[j];
}
}
```

B. **PERMUTDET2 Program**

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <time.h>
#include <mpi.h>
#define maxRows 20
#define maxCols 15
int num[maxCols],AA[maxRows][maxCols];
double det[maxRows];
double newDet[maxRows];
int cols;
int p;
int tag = 0,R=0, NB;
int pid;
int  a[maxCols][maxCols];
double sumBothDiag[maxRows];
void print(int *num,int cols);
void move(int *num, int k, int j, int cols);
void rightRotate(int *num,int cols);
void per(int *num,int k, int cols);
void per2(int *num,int k, int cols, int d);
void productDiagonal(int *num,int k,
double *sumBothDiag, int d);
int sign();

int main (int argc, char* argv[])
{ int i,j;
int myID;
int d;
double sum;
double starttime, endtime, elapsed;

MPI_Status status;
/* MPI initialization */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &myID);
if (myID == 0) // master process
{
printf(" enter the size of the element/cols =");
     scanf ("%d", &cols);
 /* for(i= 1; i<=cols; i++)
for(j=1; j<= cols; j++){
printf("a[%d][%d]=",i,j);
scanf("%f",&a[i][j]);
}*/
for(i=1; i<= cols; ++i)
            for(j=1; j<= cols; ++j)
               a[i][j] = rand() % 10;
       for(i=1; i<= cols; ++i)
        {
             for(j=1; j<= cols; ++j)
                printf("%3d", a[i][j]);
            printf("\n");
        }
starttime = MPI_Wtime();
if ( cols <= 2 )
{
if (cols ==2)
printf("( DETERMINANT OF THE MATRIX = %f",
a[1][1] * a[2][2] - a[1][2] * a[2][1]);
```

```
else
printf( "DETERMINANT OF THE MATRIX = %f", a[1][1]);
}
else {

for (i=1; i<= cols; i++){
num[i]= i ;}
per(num,cols-1,cols);
// printf("numbers of rows = %d\n", R);
NB = (int)(R) /(p-1) ; /* number of block */
// printf("numbers of block = %d\n", NB);
/* send AA  to other processes*/
//printf("**master sending AA to all**\n");
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&a, maxCols*maxCols, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&R, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&AA, (R+1)*maxCols, MPI_INT, 0, MPI_COMM_WORLD);

//////Get result from slaves//////
sum=0;
for (d=1; d<= p-1; d++)  {
MPI_Recv(&det, maxRows, MPI_DOUBLE, MPI_ANY_SOURCE,
 MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            pid=status.MPI_SOURCE;
for(i = 1; i<= R; i++)
sum += det[i];
}
printf ("\ndeterminant value of the  matrix =
 %f",sum);
}
free(num);
    free(det);
free(AA);
free(a);
} // end master process
/////////////////////////SLAVE PROCESS ////////
else {
/*receive A using broadcast*/
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&a, maxCols*maxCols, MPI_INT, 0, MPI_COMM_WORLD);

MPI_Bcast(&R, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&AA, (R+1)*maxCols, MPI_INT, 0,  MPI_COMM_WORLD);

// calculate range of rows to be processed

for(d = myID; d<= R; d+= p-1){   // start d
for (j = 1; j <= cols; j++){
num[j] = AA[d][j]; }
per2(num,cols-3,cols,d);
} // end d
MPI_Send(&det, maxRows, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD);
free(num);
free(AA);
free(a);
free(det);
free(newDet);
free(sumBothDiag);
```

```
} // end slaves process
if (myID == 0)
   {
    endtime = MPI_Wtime();
    elapsed = endtime-starttime;
    printf("\n\nParallel Time %f seconds\n",elapsed);
   }
MPI_Finalize();
return 0;
}
void move(int *num, int k, int j, int cols)
{
 int t,old;
if(k != cols){
t = num[k];
num[k] = num[k+1];
num[k+1] = t;}
else
{   old = num[cols];
for(k = cols; k>j; k--){
num[k] = num[k-1];}
num[j] = old;
}}
void rightRotate(int *num,int cols)
{
 int old,k;
 old = num[1];
for(k = 1; k<cols; ++k){
num[k] = num[k+1];}
        num[cols] = old;
}
void per(int *num,int k, int cols)
{   int i,temp;
if( k == cols-3){
print(num,cols);
return;
}
temp = k-1;
for(i = temp; i<= cols; i++){
move(num,i,temp, cols);
per(num,temp,cols);
}
}
void per2(int *num,int k, int cols, int d)
{   int i,temp;
if( k == 2){
for(i = 1; i<=cols; i++){
// move2(num,i,cols);
rightRotate(num, cols);
productDiagonal(num, i,sumBothDiag, d);
  }
for(i = 1; i<=cols; i++){
            det[d] += sumBothDiag[i];
}
return;
}
temp = k-1;
for(i = temp; i<= cols; i++){
```

```
move(num,i,temp, cols);
per2(num,temp,cols,d);
}
}
int sign()
{
int g,h,l;
for(g=1,l=1;g<cols; g++)
for (h=g+1;h<=cols; h++)
{
if(num[h]<num[g])
l*=-1;}
return (l);
}
void productDiagonal(int *num, int k,
 double *sumBothDiag, int d)
{
   int j;
   int t =1;
   double s;
   double prodMainDiag[maxRows],prodSecDiag[maxRows];
  sumBothDiag[k] = 0;
  if(cols % 4 == 0 || cols %4 == 1){
t *=sign()  ;
//printf("\nt= %d",t);
for(j= 1,s=1; j<=cols; j++){
s *= a[j][num[j]];}
prodMainDiag[k] =s*t;
for(j= 1,s=1; j<=cols; j++){
s *= a[j][num[cols-j+1]];}
prodSecDiag[k] =s*t;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);
  }
  else {
t *=sign()  ;
for(j= 1,s=1; j<=cols; j++){
s *= a[j][num[j]];}
prodMainDiag[k] =s*t;
for(j= 1,s=1; j<=cols; j++){;
s *= a[j][num[cols-j+1]];}
prodSecDiag[k] =(-1)*s*t;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);
  }
}
void print(int *num,int cols)
{ int j;
R= R+1;
for(j= 1; j<= cols; ++j){
// printf("%d",num[j]);
AA[R][j]=num[j];
}
}
```

C. **PDATM1 program**

```
#include<stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#include <mpi.h>
#define maxRows 50
#define maxCols 50
int num[maxCols];
int cols;
int p;
int tag = 0,R=0, NB;
int pid;
int  a[maxCols][maxCols];
double sumBothDiag[maxCols];
double det[maxCols];
void print(int d);
void per(int temp, int cols, int d);
void rightRotate(int cols);
double productDiagonal( int k,
double *sumBothDiag, int d);
void move(int k, int j,int cols);
void move2(int k, int i, int cols);
int sign();
void initiate();

int main (int argc, char* argv[])
{ int i,j;
int myID,d;
    double sum1, sum2, sum;
double starttime, endtime, elapsed;

MPI_Status status;
/* MPI initialization */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &myID);
if (myID == 0) // master process
{
printf(" enter the size of the element/cols =");
     scanf ("%d", &cols);
 /* for(i= 1; i<=cols; i++)
for(j=1; j<= cols; j++){
printf("a[%d][%d]=",i,j);
scanf("%f",&a[i][j]);
}*/
 for(i=1; i<= cols; ++i)
            for(j=1; j<= cols; ++j)
               a[i][j] = rand() % 10;
       for(i=1; i<= cols; ++i)
        {
             for(j=1; j<= cols; ++j)
                printf("%3d", a[i][j]);
            printf("\n");
        }
starttime = MPI_Wtime();
if ( cols <= 2 )
{
if (cols ==2)
printf("( DETERMINANT OF THE MATRIX = %f",
a[1][1] * a[2][2] - a[1][2] * a[2][1]);
else
printf( "DETERMINANT OF THE MATRIX = %f", a[1][1]);
```

```
}
else {
//printf("**master sending a to all**\n");
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&a, maxCols*maxCols, MPI_INT, 0, MPI_COMM_WORLD);
//////Get result from slaves//////
 sum = 0;
for (d=1; d<= p-1; d++)  {
MPI_Recv(&det, maxCols, MPI_DOUBLE, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
pid= status.MPI_SOURCE;
for(i = 1; i<= cols-1; i++){
sum1 += det[i];
}
}
printf("\ndeterminant value of the matrix = %f", sum);
}
free(a);
free(det);
} // end master process
//////////////////////SLAVE PROCESS /////
else {
/*receive A using broadcast*/
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&a, maxCols*maxCols, MPI_INT, 0, MPI_COMM_WORLD);
for(d= myID; d< cols; d += p-1){
initiate();
move2(d,2, cols);
per(cols-1,cols,d);
// printf("\n newDet[%d] = %f",d, newDet[d]);

}
MPI_Send(&det, maxCols, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD);

free(a);
free(det);
free(sumBothDiag);
} // end slaves process
if (myID == 0)
   {
    endtime = MPI_Wtime();
    elapsed = endtime-starttime;
    printf("\n\nParallel Time %f seconds\n",elapsed);
   }
MPI_Finalize();
return 0;
}
void initiate()
{ int i;
 for (i=1; i<=cols; i++)
 {  num[i]= i ;}
}
void rightRotate(int cols)
{
 int old,k;
  old = num[1];
for(k = 1; k<cols; ++k){
num[k] = num[k+1];}
```

```
        num[cols] = old;
}
void move2(int k, int i, int cols)
{
 int t,old;
 if (k == 1){
 t= num[i];
 num[i] = num[cols-1];
 num[cols-1] = t;
 old = t;
 old =num[cols-1];
 num[cols-1] = num[cols];
 num[cols] = old;
 }
 else {
t = num[k];
num[k] = num[i];
num[i] = t;}
}
void move(int k, int j, int cols)
{
 int t,old;
if(k != cols){
t = num[k];
num[k] = num[k+1];
num[k+1] = t;}
else
{   old = num[cols];
for(k = cols; k>j; --k){
num[k] = num[k-1];}
num[j] = old;
}
}
void per( int t, int cols, int d)
{   int i,k,temp;
if( t == 3){
for( k = 0; k< cols; k++){
rightRotate(cols);
productDiagonal( k, sumBothDiag, d);
}
for(k=0; k< cols;k++){
            det[d] += sumBothDiag[k];
}
 return;
}
 temp = t-1;
for(i = temp; i<= cols; i++){
move(i,temp, cols);
per(temp,cols, d);
}
  }
int sign()
{
int g,h,l;
for(g=1,l=1;g<cols; g++)
for (h=g+1;h<=cols; h++)
{
if(num[h]<num[g])
```

```
l*=-1;}
return (l);
}
double productDiagonal( int k, double *sumBothDiag, int d)
{
   int j;
   int t=1;
   double s;
   double prodMainDiag[maxCols],prodSecDiag[maxCols];
 sumBothDiag[k] = 0;
  if(cols %4 == 0 || cols %4 == 1){
t *=sign();
//printf("\n t= %d",t);
for(j= 1,s=1; j<=cols; j++){
s *= a[j][num[j]];}
prodMainDiag[k] =s*t;
for(j= 1,s=1; j<=cols; j++){
s *= a[j][num[cols-j+1]];}
prodSecDiag[k] =s*t;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);
  }
  else {
t *=sign();
for(j= 1,s=1; j<=cols; j++){
s *= a[j][num[j]];}
prodMainDiag[k] =s*t;
for(j= 1,s=1; j<=cols; j++){;
s *= a[j][num[cols-j+1]];}
prodSecDiag[k] =(-1)*s*t;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);
  }
return 0;
}
```

D. **PDATM2 program**

```
#include<stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#define maxRows 300
#define maxCols 50
int num[maxCols];
int cols;
int p;
int tag = 0,R=0;
int pid;
int  a[maxCols][maxCols];
double sumBothDiag[maxCols];
double det[maxCols];
double newDet[maxCols];
void print(int d);
void per(int temp, int cols, int d);
void rightRotate(int cols);
double productDiagonal( int k, double *sumBothDiag, int d);
void move2( int k, int cols);
void move( int k, int cols);
int sign();
```

```
void initiate( );

int main (int argc, char* argv[])
{ int i,j;
int myID;
int   d;
    double sum,sum1,sum2;
double starttime, endtime, elapsed;
MPI_Status status;
/* MPI initialization */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &myID);
if (myID == 0) // master process
{
printf(" enter the size of the element/cols =");
     scanf ("%d", &cols);
 /* for(i= 1; i<=cols; i++)
for(j=1; j<= cols; j++){
printf("a[%d][%d]=",i,j);
scanf("%f",&a[i][j]);
}*/
for(i=1; i<= cols; ++i)
            for(j=1; j<= cols; ++j)
               a[i][j] = rand() % 10;
       for(i=1; i<= cols; ++i)
        {
             for(j=1; j<= cols; ++j)
                printf("%3d", a[i][j]);
            printf("\n");
        }
starttime = MPI_Wtime();
if ( cols <= 3 )
{
if (cols ==3){
initiate();
per(2,cols,0);}
else if (cols==2)
printf("( DETERMINANT OF THE MATRIX = %f",
a[1][1] * a[2][2] - a[1][2] * a[2][1]);
else
printf( "DETERMINANT OF THE MATRIX = %f", a[1][1]);
}
else {
//printf("**master sending a to all**\n");
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&a, maxCols*maxCols, MPI_INT, 0, MPI_COMM_WORLD);

//////Get result from slaves//////
sum = 0;
for (d=1; d<= p-1; d++)  {

MPI_Recv(&det, maxCols, MPI_DOUBLE, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
pid= status.MPI_SOURCE;
for(i = 1; i<= cols-1; i++){
sum1 += det[i];
}
```

```
}
printf("\ndeterminant value of the matrix = %f", sum);
}
free(newDet);
free(a);

} // end master process
//////////////////////SLAVE PROCESS ///////////////
else {
/*receive A using broadcast*/
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&a, maxCols*maxCols, MPI_INT, 0, MPI_COMM_WORLD);

for(d= myID; d< cols; d += p-1){
initiate();
move2(d,cols-1);
per(2,cols,d);
}
MPI_Send(&det, maxCols, MPI_DOUBLE, 0,
MPI_ANY_TAG, MPI_COMM_WORLD);
free(a);
free(det);
free(newDet);
free(sumBothDiag);
} // end slaves process
if (myID == 0)
   {
    endtime = MPI_Wtime();
    elapsed = endtime-starttime;
    printf("\n\nParallel Time %f seconds\n",elapsed);
   }
MPI_Finalize();
return 0;
}
void initiate()
{ int i;
 for (i=1; i<=cols; i++)
 {  num[i]= i ;}
}
void rightRotate(int cols)
{
 int old,k;
 old = num[1];
for(k = 1; k<cols; ++k){
num[k] = num[k+1];}
        num[cols] = old;
}
void move2( int k, int cols)
{
 int t,old;
 if (k == 1){
 t= num[k+1];
 num[k+1] = num[cols];
 num[cols] = t;
 old = t;
 old =num[k];
 num[k] = num[2];
 num[2] = old;
```

```
 }
 else {
t = num[k];
num[k] = num[cols];
num[cols] = t;}
}
void move( int k, int cols)
{
 int t,old;
if(k !=1){
t = num[k];
num[k] = num[k-1];
num[k-1] = t;}
else
{
 old = num[1];
for(k = 1; k<cols; ++k){
num[k] = num[k+1];}
        num[cols] = old;
}
}
void per( int temp, int cols, int d)
{   int i,k;
if( temp == cols-2){
for( k = 0; k< cols; k++){
rightRotate(cols);
productDiagonal( k, sumBothDiag, d);
}
for(k=0; k< cols;k++){
            det[d] += sumBothDiag[k];
}
 return;
}
for(i = temp+1; i>= 1; i--){
move(i,temp+1);
per(temp+1,cols, d);
}
  }
int sign()
{
int g,h,l;
for(g=1,l=1;g<cols; g++)
for (h=g+1;h<=cols; h++)
{
if(num[h]<num[g])
l*=-1;}
return (l);
}
double productDiagonal( int k, double *sumBothDiag, int d)
{
   int j;
   int t=1;
   double s;
   double prodMainDiag[maxCols],prodSecDiag[maxCols];
 sumBothDiag[k] = 0;
  if(cols %4 == 0 || cols %4 == 1){
t *=sign();
for(j= 1,s=1; j<=cols; j++){
```

```
s *= a[j][num[j]];}
prodMainDiag[k] =s*t;
for(j= 1,s=1; j<=cols; j++){
s *= a[j][num[cols-j+1]];}
prodSecDiag[k] =s*t;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);
  }
  else {
t *=sign();
for(j= 1,s=1; j<=cols; j++){
s *= a[j][num[j]];}
prodMainDiag[k] =s*t;
for(j= 1,s=1; j<=cols; j++){;
s *= a[j][num[cols-j+1]];}
prodSecDiag[k] =(-1)*s*t;
sumBothDiag[k] += (prodSecDiag[k] + prodMainDiag[k]);
  }
return 0;
}
```

Appendix E

Permutation and Determinant Program Output

A. **PERMUT1 program output**

```
 enter the size of the element =5
45321 12354
53214 41235
32145 54123
21453 35412
14532 23541
53241 14235
32415 51423
24153 35142
41532 23514
15324 42351
32451 15423
24513 31542
45132 23154
51324 42315
13245 54231
24531 13542
45312 21354
53124 42135
31245 54213
12453 35421
53421 12435
34215 51243
42153 35124
21534 43512
15342 24351
34251 15243
42513 31524
25134 43152
51342 24315
13425 52431
42531 13524
25314 41352
53142 24135
31425 52413
14253 35241
25341 14352
53412 21435
34125 52143
41253 35214
12534 43521
34521 12543
45213 31254
52134 43125
21345 54312
13452 25431
45231 13254
52314 41325
```

```
23145 54132
31452 25413
14523 32541
52341 14325
23415 51432
34152 25143
41523 32514
15234 43251
23451 15432
34512 21543
45123 32154
51234 43215
12345 54321
```

B. **PERMUT2 program output**

```
  enter the size of the element =5
42351 15324
23514 41532
35142 24153
51423 32415
14235 53241
43251 15234
32514 41523
25143 34152
51432 23415
14325 52341
43521 12534
35214 41253
52143 34125
21435 53412
14352 25341
24351 15342
43512 21534
35124 42153
51243 34215
12435 53421
42531 13524
25314 41352
53142 24135
31425 52413
14253 35241
45231 13254
52314 41325
23145 54132
31452 25413
14523 32541
45321 12354
53214 41235
32145 54123
21453 35412
14532 23541
24531 13542
45312 21354
53124 42135
31245 54213
12453 35421
32451 15423
```

```
24513 31542
45132 23154
51324 42315
13245 54231
34251 15243
42513 31524
25134 43152
51342 24315
13425 52431
34521 12543
45213 31254
52134 43125
21345 54312
13452 25431
23451 15432
34512 21543
45123 32154
51234 43215
12345 54321
Press any key to continue
```

C. **PERMUTIT3 program output**

```
  enter the size of the element=5
12345 54321
23451 15432
34512 21543
45123 32154
51234 43215
23415 51432
34152 25143
41523 32514
15234 43251
52341 14325
34125 52143
41253 35214
12534 43521
25341 14352
53412 21435
41235 53214
12354 45321
23541 14532
35412 21453
54123 32145
23145 54132
31452 25413
14523 32541
45231 13254
52314 41325
31425 52413
14253 35241
42531 13524
25314 41352
53142 24135
14235 53241
42351 15324
23514 41532
35142 24153
```

```
51423 32415
42315 51324
23154 45132
31542 24513
15423 32451
54231 13245
31245 54213
12453 35421
24531 13542
45312 21354
53124 42135
12435 53421
24351 15342
43512 21534
35124 42153
51243 34215
24315 51342
43152 25134
31524 42513
15243 34251
52431 13425
43125 52134
31254 45213
12543 34521
25431 13452
54312 21345
Press any key to continue
```

D. **PERMUTDET1 program Output**

```
 enter the size of the element=5
 1  7  4  0  9
 4  8  8  2  4
 5  5  1  7  1
 1  5  2  7  6
 1  4  2  3  2

prodMainDiag = -120.000000
prodSecDiag = -560.000000
prodMainDiag = 0.000000
prodSecDiag = -120.000000
prodMainDiag = -84.000000
prodSecDiag = 0.000000
prodMainDiag = -10080.000000
prodSecDiag = -280.000000
prodMainDiag = -1120.000000
prodSecDiag = -1008.000000
prodMainDiag = 0.000000
prodSecDiag = 60.000000
prodMainDiag = 168.000000
prodSecDiag = 0.000000
prodMainDiag = 2240.000000
prodSecDiag = 840.000000
prodMainDiag = 2520.000000
prodSecDiag = 448.000000
prodMainDiag = 40.000000
prodSecDiag = 2520.000000
```

```
prodMainDiag = -5040.000000
prodSecDiag = 0.000000
prodMainDiag = -448.000000
prodSecDiag = -28.000000
prodMainDiag = -280.000000
prodSecDiag = -1344.000000
prodMainDiag = -180.000000
prodSecDiag = -560.000000
prodMainDiag = 0.000000
prodSecDiag = -540.000000
prodMainDiag = 2688.000000
prodSecDiag = 392.000000
prodMainDiag = 20.000000
prodSecDiag = 224.000000
prodMainDiag = 0.000000
prodSecDiag = 720.000000
prodMainDiag = 216.000000
prodSecDiag = 0.000000
prodMainDiag = 3920.000000
prodSecDiag = 720.000000
prodMainDiag = -84.000000
prodSecDiag = -112.000000
prodMainDiag = 0.000000
prodSecDiag = -168.000000
prodMainDiag = -1200.000000
prodSecDiag = 0.000000
prodMainDiag = -2520.000000
prodSecDiag = -80.000000
prodMainDiag = -224.000000
prodSecDiag = -2520.000000
prodMainDiag = 0.000000
prodSecDiag = 840.000000
prodMainDiag = 240.000000
prodSecDiag = 0.000000
prodMainDiag = 280.000000
prodSecDiag = 240.000000
prodMainDiag = 1008.000000
prodSecDiag = 560.000000
prodMainDiag = 28.000000
prodSecDiag = 1008.000000
prodMainDiag = -1440.000000
prodSecDiag = 0.000000
prodMainDiag = -112.000000
prodSecDiag = -40.000000
prodMainDiag = -392.000000
prodSecDiag = -1344.000000
prodMainDiag = -72.000000
prodSecDiag = -392.000000
prodMainDiag = 0.000000
prodSecDiag = -2160.000000
prodMainDiag = 672.000000
prodSecDiag = 560.000000
prodMainDiag = 28.000000
prodSecDiag = 224.000000
prodMainDiag = 0.000000
prodSecDiag = 504.000000
prodMainDiag = 5400.000000
prodSecDiag = 0.000000
```

```
 prodMainDiag = 1120.000000
 prodSecDiag = 180.000000
 prodMainDiag = -960.000000
 prodSecDiag = -1960.000000
 prodMainDiag = 0.000000
 prodSecDiag = -96.000000
 prodMainDiag = -120.000000
 prodSecDiag = 0.000000
 prodMainDiag = -504.000000
 prodSecDiag = -20.000000
 prodMainDiag = -784.000000
 prodSecDiag = -2016.000000
 prodMainDiag = 0.000000
 prodSecDiag = 240.000000
 prodMainDiag = 48.000000
 prodSecDiag = 0.000000
 prodMainDiag = 196.000000
 prodSecDiag = 48.000000
 prodMainDiag = 2016.000000
 prodSecDiag = 784.000000
 prodMainDiag = 400.000000
 prodSecDiag = 5040.000000
 prodMainDiag = -144.000000
 prodSecDiag = 0.000000
 prodMainDiag = -392.000000
 prodSecDiag = -16.000000
 prodMainDiag = -448.000000
 prodSecDiag = -2352.000000
 prodMainDiag = -900.000000
 prodSecDiag = -2240.000000
 prodMainDiag = 0.000000
 prodSecDiag = -1080.000000
 prodMainDiag = 2352.000000
 prodSecDiag = 224.000000
 prodMainDiag = 32.000000
 prodSecDiag = 392.000000
 prodMainDiag = 0.000000
 prodSecDiag = 2880.000000
 prodMainDiag = 1080.000000
 prodSecDiag = 0.000000
 prodMainDiag = 112.000000
 prodSecDiag = 90.000000
 determinant of A= -2122.000000
Press any key to continue
```

D. **PERMUTDET2 program Output**

```
  enter the size of the element=5
  1  7  4  0  9
  4  8  8  2  4
  5  5  1  7  1
  1  5  2  7  6
  1  4  2  3  2
prodMainDiag= 240.000000
prodSecDiag= 280.000000
prodMainDiag= 0.000000
prodSecDiag= 240.000000
```

```
prodMainDiag= 840.000000
prodSecDiag= 0.000000
prodMainDiag= 1008.000000
prodSecDiag= 28.000000
prodMainDiag= 560.000000
prodSecDiag= 1008.000000
prodMainDiag= 0.000000
prodSecDiag= -120.000000
prodMainDiag= -96.000000
prodSecDiag= 0.000000
prodMainDiag= -1960.000000
prodSecDiag= -960.000000
prodMainDiag= -2016.000000
prodSecDiag= -784.000000
prodMainDiag= -20.000000
prodSecDiag= -504.000000
prodMainDiag= 720.000000
prodSecDiag= 0.000000
prodMainDiag= 224.000000
prodSecDiag= 20.000000
prodMainDiag= 392.000000
prodSecDiag= 2688.000000
prodMainDiag= 720.000000
prodSecDiag= 3920.000000
prodMainDiag= 0.000000
prodSecDiag= 216.000000
prodMainDiag= -1344.000000
prodSecDiag= -280.000000
prodMainDiag= -28.000000
prodSecDiag= -448.000000
prodMainDiag= 0.000000
prodSecDiag= -5040.000000
prodMainDiag= -540.000000
prodSecDiag= 0.000000
prodMainDiag= -560.000000
prodSecDiag= -180.000000
prodMainDiag= -120.000000
prodSecDiag= -560.000000
prodMainDiag= 0.000000
prodSecDiag= -120.000000
prodMainDiag= -84.000000
prodSecDiag= 0.000000
prodMainDiag= -10080.000000
prodSecDiag= -280.000000
prodMainDiag= -1120.000000
prodSecDiag= -1008.000000
prodMainDiag= 0.000000
prodSecDiag= 240.000000
prodMainDiag= 48.000000
prodSecDiag= 0.000000
prodMainDiag= 196.000000
prodSecDiag= 48.000000
prodMainDiag= 2016.000000
prodSecDiag= 784.000000
prodMainDiag= 400.000000
prodSecDiag= 5040.000000
prodMainDiag= -1440.000000
prodSecDiag= 0.000000
```

```
prodMainDiag= -112.000000
prodSecDiag= -40.000000
prodMainDiag= -392.000000
prodSecDiag= -1344.000000
prodMainDiag= -72.000000
prodSecDiag= -392.000000
prodMainDiag= 0.000000
prodSecDiag= -2160.000000
prodMainDiag= 672.000000
prodSecDiag= 560.000000
prodMainDiag= 28.000000
prodSecDiag= 224.000000
prodMainDiag= 0.000000
prodSecDiag= 504.000000
prodMainDiag= 5400.000000
prodSecDiag= 0.000000
prodMainDiag= 1120.000000
prodSecDiag= 180.000000
prodMainDiag= -84.000000
prodSecDiag= -112.000000
prodMainDiag= 0.000000
prodSecDiag= -168.000000
prodMainDiag= -1200.000000
prodSecDiag= 0.000000
prodMainDiag= -2520.000000
prodSecDiag= -80.000000
prodMainDiag= -224.000000
prodSecDiag= -2520.000000
prodMainDiag= 0.000000
prodSecDiag= 60.000000
prodMainDiag= 168.000000
prodSecDiag= 0.000000
prodMainDiag= 2240.000000
prodSecDiag= 840.000000
prodMainDiag= 2520.000000
prodSecDiag= 448.000000
prodMainDiag= 40.000000
prodSecDiag= 2520.000000
prodMainDiag= -144.000000
prodSecDiag= 0.000000
prodMainDiag= -392.000000
prodSecDiag= -16.000000
prodMainDiag= -448.000000
prodSecDiag= -2352.000000
prodMainDiag= -900.000000
prodSecDiag= -2240.000000
prodMainDiag= 0.000000
prodSecDiag= -1080.000000
prodMainDiag= 2352.000000
prodSecDiag= 224.000000
prodMainDiag= 32.000000
prodSecDiag= 392.000000
prodMainDiag= 0.000000
prodSecDiag= 2880.000000
prodMainDiag= 1080.000000
prodSecDiag= 0.000000
prodMainDiag= 112.000000
prodSecDiag= 90.000000
```

```
determinant of A= -2122.000000
Press any key to continue
```

F. **PERMUTDETIT3 program Output**

```
 enter the size of the element=5
  1  7  4  0  9
  4  8  8  2  4
  5  5  1  7  1
  1  5  2  7  6
  1  4  2  3  2
prodMainDiag= 112.000000
prodSecDiag= 90.000000
prodMainDiag= 2352.000000
prodSecDiag= 224.000000
prodMainDiag= 32.000000
prodSecDiag= 392.000000
prodMainDiag= 0.000000
prodSecDiag= 2880.000000
prodMainDiag= 1080.000000
prodSecDiag= 0.000000
prodMainDiag= -784.000000
prodSecDiag= -2016.000000
prodMainDiag= -960.000000
prodSecDiag= -1960.000000
prodMainDiag= 0.000000
prodSecDiag= -96.000000
prodMainDiag= -120.000000
prodSecDiag= 0.000000
prodMainDiag= -504.000000
prodSecDiag= -20.000000
prodMainDiag= 400.000000
prodSecDiag= 5040.000000
prodMainDiag= 0.000000
prodSecDiag= 240.000000
prodMainDiag= 48.000000
prodSecDiag= 0.000000
prodMainDiag= 196.000000
prodSecDiag= 48.000000
prodMainDiag= 2016.000000
prodSecDiag= 784.000000
prodMainDiag= 0.000000
prodSecDiag= -1080.000000
prodMainDiag= -144.000000
prodSecDiag= 0.000000
prodMainDiag= -392.000000
prodSecDiag= -16.000000
prodMainDiag= -448.000000
prodSecDiag= -2352.000000
prodMainDiag= -900.000000
prodSecDiag= -2240.000000
prodMainDiag= 3920.000000
prodSecDiag= 720.000000
prodMainDiag= 2688.000000
prodSecDiag= 392.000000
prodMainDiag= 20.000000
```

```
prodSecDiag= 224.000000
prodMainDiag= 0.000000
prodSecDiag= 720.000000
prodMainDiag= 216.000000
prodSecDiag= 0.000000
prodMainDiag= -1120.000000
prodSecDiag= -1008.000000
prodMainDiag= -120.000000
prodSecDiag= -560.000000
prodMainDiag= 0.000000
prodSecDiag= -120.000000
prodMainDiag= -84.000000
prodSecDiag= 0.000000
prodMainDiag= -10080.000000
prodSecDiag= -280.000000
prodMainDiag= 40.000000
prodSecDiag= 2520.000000
prodMainDiag= 0.000000
prodSecDiag= 60.000000
prodMainDiag= 168.000000
prodSecDiag= 0.000000
prodMainDiag= 2240.000000
prodSecDiag= 840.000000
prodMainDiag= 2520.000000
prodSecDiag= 448.000000
prodMainDiag= 0.000000
prodSecDiag= -540.000000
prodMainDiag= -5040.000000
prodSecDiag= 0.000000
prodMainDiag= -448.000000
prodSecDiag= -28.000000
prodMainDiag= -280.000000
prodSecDiag= -1344.000000
prodMainDiag= -180.000000
prodSecDiag= -560.000000
prodMainDiag= 1120.000000
prodSecDiag= 180.000000
prodMainDiag= 672.000000
prodSecDiag= 560.000000
prodMainDiag= 28.000000
prodSecDiag= 224.000000
prodMainDiag= 0.000000
prodSecDiag= 504.000000
prodMainDiag= 5400.000000
prodSecDiag= 0.000000
prodMainDiag= -224.000000
prodSecDiag= -2520.000000
prodMainDiag= -84.000000
prodSecDiag= -112.000000
prodMainDiag= 0.000000
prodSecDiag= -168.000000
prodMainDiag= -1200.000000
prodSecDiag= 0.000000
prodMainDiag= -2520.000000
prodSecDiag= -80.000000
prodMainDiag= 28.000000
prodSecDiag= 1008.000000
prodMainDiag= 0.000000
```

```
prodSecDiag= 840.000000
prodMainDiag= 240.000000
prodSecDiag= 0.000000
prodMainDiag= 280.000000
prodSecDiag= 240.000000
prodMainDiag= 1008.000000
prodSecDiag= 560.000000
prodMainDiag= 0.000000
prodSecDiag= -2160.000000
prodMainDiag= -1440.000000
prodSecDiag= 0.000000
prodMainDiag= -112.000000
prodSecDiag= -40.000000
prodMainDiag= -392.000000
prodSecDiag= -1344.000000
prodMainDiag= -72.000000
prodSecDiag= -392.000000
determinant of A= -2122.000000

Press any key to continue
```