PARALLEL GAUSS ELIMINATION WITH PARTIAL PIVOTING USING PTHREADS

Submitted
By

SONALI MUKHERJEE

EXAMINATION ROLL NUMBER : **M4SWE10-11**

REGISTRATION NUMBER: **104330 of 2008-2010**

A Thesis submitted to

The Faculty of Engineering & Technology of Jadavpur University

In Partial Fulfillment of the Requirements for the Degree of

MASTER OF ENGINEERING

In

SOFTWARE ENGINEERING

Under the Supervision

Of

Mr. UTPAL KUMAR RAY

DEPARTMENT OF INFORMATION TECHNOLOGY

JADAVPUR UNIVERSITY
2010

JADAVPUR UNIVERSITY

FACULTY OF ENGINEERING & TECHNOLOGY

CERTIFICATE OF APPROVAL

The thesis at instance is hereby approved as a creditable study of an Engineering subject carried out and presented in a manner satisfactory to warrant its acceptance as a prerequisite to the degree for which it has been submitted. It is understood that by this approval the undersigned does not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein, but approve this thesis for the purpose for which it is submitted.

__________________________ __________________________

Signature of the Examiner ***Signature of the Supervisor***

ACKNOWLEDEMENT

With great pleasure I would like to express my profound gratitude and indebtedness to ***Mr. Utpal Kr. Ray****, Department of Information Technology, Jadavpur University, Kolkata, for his continuous guidance, valuable advice and constant encouragement throughout the project work. His valuable and constructive suggestions at many difficult situations are immensely acknowledged. I am in short of words to express his contribution to this thesis through criticism, suggestions and discussions.*

I would like to take this opportunity to thank ***Prof. Samiran Chattopadhyay****, Head of the Department, Department of Information Technology, Jadavpur University, Kolkata, for his co-operation.*

My wholehearted thanks to staff of the Department of Information Technology, Jadavpur University, Kolkata, and my friends for their help and vital suggestions.

Regards

____

Sonali Mukherjee

Dedicated to my

Beloved

Parents & Daughter

ABSTRACT

In mathematics, the theory of linear systems is a branch of linear algebra, a subject which is fundamental to modern mathematics. Computational algorithms for finding the solutions are an important part of numerical linear algebra, and such methods play a prominent role in engineering, physics, chemistry, computer science, statistics and economics. Many scientific & engineering field like structural analysis (civil engineering), heat transport (mechanical engineering), analysis of power grids (electrical engineering), production planning (economics), regression analysis (statistics) problems can take the form of a system of linear equations. There are many algorithms are used to solve the system of linear equations. Since linear systems are derived from realistic problems are often quite large. It is required to study and analysis how to solve them efficiently on parallel computers.

One of the most efficient direct method to solve the linear system is Gauss Elimination which is alone is sufficient for many applications. But naïve Gauss Elimination has some pitfall which can be overcome by Gauss Elimination with Partial Pivoting algorithm. This algorithm is useful to solve the liner system with millions number of unknown variables. So it's a good point to start the parallelization of this useful algorithm.

Within our thesis we developed two parallel approaches of this algorithm row oriented and column oriented approaches by pthread which is applicable for multiprocessor machines. With the experimental result we saw that Row oriented approach of this algorithm gives much better results in comparison to column oriented approach of this algorithm. The speedup and efficiency of number of variables with respect to number of processors becomes more and more ideal when the number of variables is increasing in case of row oriented parallelization of GEPP, which is required for parallelization. Row oriented approach is suitable when number of variables is much more grater than the number of processors. The column oriented approach is suitable just for the opposite when number of variables is lesser than the number of processors.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

1.2 FOCUS

1.3 ORGANIZATION

Chapter 1

Introduction

The thesis deals with Row Oriented and Column Oriented parallelization of Gauss Elimination with Partial Pivoting mainly for dense matrix using Pthread. The thesis contains a comparison between the two parallel approaches of the algorithm implemented by pthread. There are many researches have been done for parallelization of Gauss Elimination mostly using MPI but the majority are for sparse matrices and without partial pivoting.

1.1 MOTIVATION

Parallelization of Numerical algorithms are one of the most important research area of parallel processing since it is particularly useful in projects that require complex computations. With the help of parallel processing, highly complicated scientific problems that are otherwise extremely difficult to solve can be solved effectively.

Gauss's Elimination Method is one of the most efficient direct methods for solving unknown variables of linear systems which could be simply expressed in terms of matrices, it alone is sufficient for many applications. Large data set or may be large file having large data handling is a major concern in today's computing including the scalability and speedup. By using parallel techniques the large computation can be split and distributed to other processors to speedup the computation. As we all know that if the total calculation job is broken down into smaller jobs and distributed among several processors then it will be much more efficient compared to one processor is responsible for the whole job. It should speed up the algorithm.

1.2 FOCUS

Our focus in this thesis will be on methods suitable for multiprocessor computer. At first we have to develop a sequential coding on Gauss Elimination with Partial

Pivoting algorithm. Thereafter we will narrow our coverage in parallel implementation by pthread, for that we concentrate on two parallel approaches of this algorithm which are Row Oriented and Column Oriented. After implementation of the two parallel approaches of the algorithm an detailed comparison have been done to validate that how the sequential, row oriented and column oriented gauss elimination with partial pivoting algorithm responses for different number of variables and processors.

1.3 ORGANIZATION

The organization of this thesis is as follows:

- Chapter 2 presents Introduction of Gauss Elimination
- Chapter 3 presents the Sequential Gauss Elimination with Partial Pivoting
- Chapter 4 presents the various Parallel Gauss Elimination Techniques
- Chapter 5 gives overview of PThread (POSIX Thread)
- Chapter 6 presents Design & Implementation of Row oriented and Column oriented Parallel Gauss Elimination with Partial Pivoting
- Chapter 7 Results, Conclusion & Further Work
- Finally the Appendices gives the complete information about PThread in terms of installation, list of PThread calls and commands and compiling and executing "Parallel Gauss Elimination with Partial Pivoting" program
- The programs source code is presented in CD (Compact Disk) attached to the project report at the end.

CHAPTER 2

INTRODUCTION OF GAUSS ELIMINATION METHOD

2.1 OVERVIEW OF LINEAR SYSTEM

2.1.1 LINEAR EQUATION

2.1.2 LINEAR SYSTEM

2.1.3 DIFFERENT NUMERICAL METHODS USED TO SOLVE LINEAR SYSTEM

2.2 WORKING PRINCIPLE OF GAUSS ELIMINATION METHOD

2.2.1 EXPLANATION WITH AN EXAMPLE

2.3 WHY GAUSS ELIMINATION WITH PARTIAL PIVOTING IS NEEDED?

2.3.1 PITFALLS OF GAUSS ELIMINATION METHOD

2.3.2 TECHNIQUES FOR IMPROVING THE GAUSS ELIMINATION METHOD

2.3.3 HOW DOES GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING DIFFERS FROM GAUSS ELIMINATION

Chapter 2

Introduction of Gauss Elimination Method

In this chapter we will discuss about linear systems, how Gauss Elimination method works and why partial pivoting is needed.

2.1 OVERVIEW OF LINER SYTEM

Theory of Linear Systems is the branch of Linear Algebra and algorithms for solving the system is the part of Numerical Linear Algebra which plays a prominent role in Engineering, Physics, Chemistry, Computer Science, Statistics and Economics. Non-Linear Equations can often be approximated by a linear system, e.g. Partial Differential Equations. These equations could be simply expressed in terms of matrices. With n equations, the highest possible number of unknown variables that can be solved for is n.

2.1.1 LINEAR EQUATION

A linear equation in the n variables $x_1,x_2,\ldots\ldots,x_n$ is an equation that can be expressed as

$$a_1x_1 + a_2x_2 +\ldots\ldots\ldots\ldots\ldots.+ a_nx_n = b \quad \ldots\ldots\ldots.. (1)$$

where $a_1,a_2,\ldots\ldots\ldots.,a_n$ and b are constant.

2.1.2 LINEAR SYSTEM

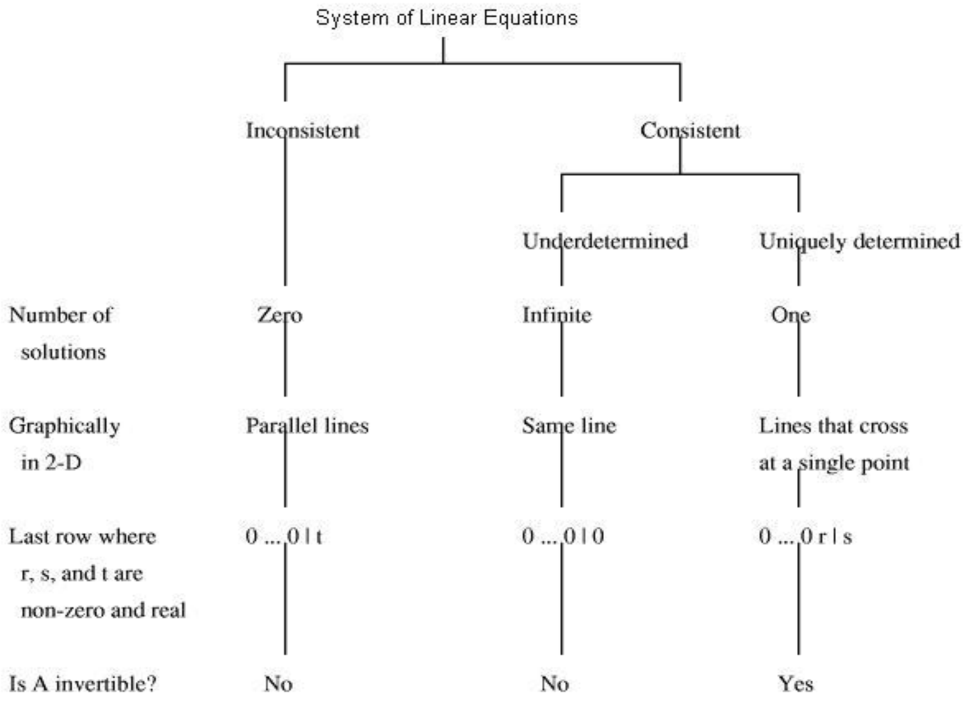
A finite set of linear equations in the n unknown variables $x_1,x_2,\ldots\ldots,x_n$ of the form

$$\left.\begin{array}{c} a_{11}x_1 + a_{12}x_2 + \ldots\ldots\ldots\ldots\ldots\ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots\ldots\ldots\ldots\ldots\ldots + a_{2n}x_n = b_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots\ldots\ldots\ldots\ldots\ldots + a_{nn}x_n = b_n \end{array}\right\} \quad \ldots\ldots\ldots (2)$$

is called a system of linear equations or linear system where the coefficients a_{ij} $(i,j = 1,2,\ldots..,n)$ and the constants b_i $(i = 1,2,\ldots\ldots..,n)$ are given numbers. The matrix notation of the linear system is

$$Ax = b \quad \ldots\ldots\ldots\ldots\ldots \quad (3)$$

where $A = [a_{ij}]$ $(i,j = 1,2,\ldots..,n)$ is an n X n matrix called the coefficient matrix and $b = [b_i]^T$ $(i = 1,2,\ldots\ldots..,n)$ is a column n-vector which are prescribed and $x = [x_i]^T$ $(i=1,2,\ldots\ldots\ldots.,n)$ is the n-column vector of unknown variables to be determined.



Linear System at a Glance

2.1.3 DIFFERENT NUMERICAL METHODS USED TO SOLVE LINEAR SYSTEM

There are direct methods and iterative methods for solving all types of linear systems.

DIRECT METHOD

The direct or exact method theoretically gives an exact solution in a (predictable) finite number of steps of arithmetical operations. Final system is so simple that its solution may be readily computed. In this method it is worthwhile counting the total number of arithmetical operations necessary in the process which gives an inverse measure of efficiency of the method. At every step a new equivalent system of equations is obtained from the previous one by performing some arithmetical operations, so that the coefficients and constants of the system of equations computed at every step are affected with round-off and other errors which are unavoidable. All these errors invariably make the solution erroneous to some extent. Cramer's Rule, Gauss's Elimination, Gauss's Jordan Elimination, LU Decomposition, Cholesky Decomposition are examples of numerical direct methods to solve linear system.

ITERATIVE METHOD

The iterative methods are not expected to terminate in a pre assigned number of steps. Starting from an initial guess, it forms a sequence of successive approximations to the solution that converges to the solution. A convergence criterion is specified in order to decide when a sufficiently accurate solution has been found. If the process is convergent, it needs to find an estimate of the error committed in stopping with the k^{th} iterate. This estimation computed at every step which will help to decide when to stop the process for a required level of accuracy. The round off and other computational errors at the intermediate steps will not affect the final solution, for any erroneous iterative serves as a new initial guess for the next step of iteration. Its frequently used to solve the sparse linear systems generated when working with partial differential equations, using discrete methods. Forward/Backward Substitutions, Jacobi Algorithm, Gauss's-Seidel Algorithm, Jacobi Overrelaxation, Conjugate Gradient Method etc are examples of iterative methods.

2.2 WORKING PRINCIPLE OF GAUSS ELIMINATION METHOD

Gauss's elimination is one of the most reliable and stable direct methods for the solution of linear equations. It also provides a method for the inversion of matrices.

The approach is designed to solve a general set of n equations and n unknowns

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n &= b_2 \\ . \qquad\qquad & . \\ . \qquad\qquad & . \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n &= b_n \end{aligned}$$

Gaussian elimination consists of two steps

1. **FORWARD ELIMINATION OF UNKNOWNS**: In this step, the unknown is eliminated in each equation starting with the first equation. Forward elimination reduces Ax = b to an upper triangular or echelon form Tx = c or results in a degenerate equation with no solution, indicating the system has no solution. This is accomplished through the use of elementary row operations.

   Any row in a linear system can be replaced by the sum of that row and a nonzero multiple of any row of the system. This technique is used to eliminate nonzero elements from below the main diagonal in tridiagonal system.

2. **BACK SUBSTITUTION**: In this step, starting from the last equation, each of the unknowns is found. Back Substitution on Tx = c to find the solution of the system.

2.2.1 EXPLANATION WITH AN EXAMPLE

Here is a dense linear system that first we want to get into upper triangular form

$$\begin{aligned} 4x_1 + 6x_2 + 2x_3 - 2x_4 &= 8 \\ 2x_1 \qquad + 5x_3 - 2x_4 &= 4 \\ -4x_1 - 3x_2 - 5x_3 + 4x_4 &= 1 \\ 8x_1 + 18x_2 - 2x_3 + 3x_4 &= 40 \end{aligned}$$

Here A is 4X4 coefficient matrix, and det(A) $\neq 0$, therefore the system has an unique solution.

Step 1: Since $a_{11} \neq 0$**,** dividing the 1st equation by a_{11} and using the resulting equation to eliminate the variable x_1 from remaining three equations. We obtain the following system

$$\begin{aligned} x_1 + 1.5x_2 + 0.5x_3 - 0.5x_4 &= 2 \\ -3x_2 + x_3 - x_4 &= 0 \\ 3x_2 - 3x_3 + 2x_4 &= 9 \\ 6x_2 - 6x_3 + 7x_4 &= 24 \end{aligned}$$

where

$a_{1j}^{(1)} = a_{1j}^{(0)} / a_{11}^{(0)}$

$b_1^{(1)} = b_1^{(0)} / a_{11}^{(0)}$

$a_{ij}^{(1)} = a_{ij}^{(0)} - a_{i1}^{(0)} * a_{1j}^{(1)}$

$b_i^{(1)} = b_i^{(0)} - a_{i1}^{(0)} * b_1^{(1)}$, (i,j = 2,3,4)

Step 2: $a_{22}^{(1)} \neq 0$, dividing the 2nd equation by $a_{22}^{(1)}$ and using the resulting equation to eliminate the variable x_2 from remaining two equations. We obtain the following system

$$\begin{aligned} x_1 + 1.5x_2 + 0.5x_3 - 0.5x_4 &= 2 \\ -3x_2 + x_3 - x_4 &= 0 \\ x_3 + x_4 &= 9 \\ 2x_3 + 5x_4 &= 24 \end{aligned}$$

where

$a_{2j}^{(2)} = a_{2j}^{(1)} / a_{22}^{(1)}$

$b_2^{(2)} = b_2^{(1)} / a_{22}^{(1)}$

$a_{ij}^{(2)} = a_{ij}^{(1)} - a_{i2}^{(1)} * a_{2j}^{(2)}$

$b_i^{(2)} = b_i^{(1)} - a_{i2}^{(1)} * b_2^{(2)}$, (i,j =3,4) again $a_{33}^{(2)} \neq 0$, dividing the 3rd equation by $a_{33}^{(2)}$ and using the resulting equation to eliminate the variable x_3 from remaining one equation. We obtain the following system

$$\begin{aligned} x_1 + 1.5x_2 + 0.5x_3 - 0.5x_4 &= 2 \\ -3x_2 + x_3 - x_4 &= 0 \\ x_3 + x_4 &= 9 \\ 3x_4 &= 6 \end{aligned}$$

where

$a_{3j}^{(3)} = a_{3j}^{(2)} / a_{33}^{(2)}$

$b_3^{(3)} = b_3^{(2)} / a_{33}^{(2)}$

$a_{ij}^{(3)} = a_{ij}^{(2)} - a_{i3}^{(2)} * a_{3j}^{(3)}$

$b_i^{(3)} = b_i^{(2)} - a_{i3}^{(2)} * b_3^{(3)}$, (i,j =4)

This completes the transformation of the dense linear system into upper triangular system. At this point using the Back Substitution method the solution of the unknown variables can be determined. Which are as follows

$$x_1 = -3,\ x_2 = 1.66,\ x_3 = 7,\ x_4 = 2$$

2.3 WHY GAUSS ELIMINATION WITH PARTIAL PIVOTING IS NEEDED?

Gaussian elimination is numerically stable for diagonally dominant or positive-definite matrices. For general matrices Gaussian elimination with partial pivoting is stable.

2.3.1 PITFALLS OF GAUSS ELIMINATION METHOD

There are two pitfalls of Gauss elimination method.

1. ROUND-OFF ERROR : Gauss elimination method is prone to round-off errors. This is true when there are large numbers of equations as errors propagate. Also, if there is subtraction of numbers from each other, it may create large errors.

2. DIVISION BY ZERO : It is possible that division by zero may occur during forward elimination steps. If at any step any of the pivotal coefficient ($a_1^{(1)}$, $a_{22}^{(2)}$, $a_{33}^{(3)}$,….., $a_{n-1n-1}^{(n-1)}$) becomes zero, then normalization would require division by zero.

2.3.2 TECHNIQUES FOR IMPROVING THE GAUSS ELIMINATION METHOD

One method of decreasing the round-off error would be to use more significant digits, that is, to use double or quad precision for representing the numbers. To avoid division by zero as well as reduce (not eliminate) round-off error, Gaussian elimination with partial pivoting is the method of choice.

2.3.3 HOW DOES GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING DIFFER FROM GAUSS ELIMINATION

The two methods are the same, except in the beginning of each step of forward elimination; a row switching is done based on the following criterion. If there are n equations, then there are (n-1) forward elimination steps. At the beginning of the k^th^ step of forward elimination, one finds the maximum of ($|a_{k,k}|,|a_{k+1,k}|$, …………,$|a_{nk}|$). Then if the maximum of these values is $|a_{pk}|$ in the p^th^ row, $k \leq p \leq n$, then swap rows p and k. The other steps of forward elimination are the same as the Gauss elimination method. The back substitution steps stay exactly the same as the Gauss elimination method. Gauss's elimination does not exhibit good numerical stability on digital computers.

CHAPTER3

SEQUENTIAL GAUSS ELIMINATION WITH PARTIAL PIVOTING

3.1 FLOWCHART

3.2 IMPLEMENTATION OF SEQUENTIAL GEPP

Chapter 3

Sequential Gauss Elimination with Partial Pivoting

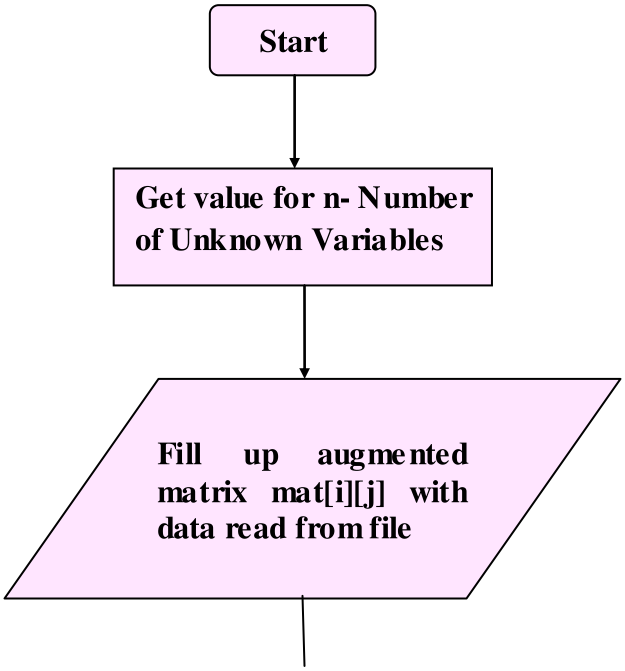
In this chapter we will discuss about the flowchart of the Gauss elimination with partial pivoting algorithm and implementation details of that algorithm. Also contains the detail of compilation and execution of the executable code.

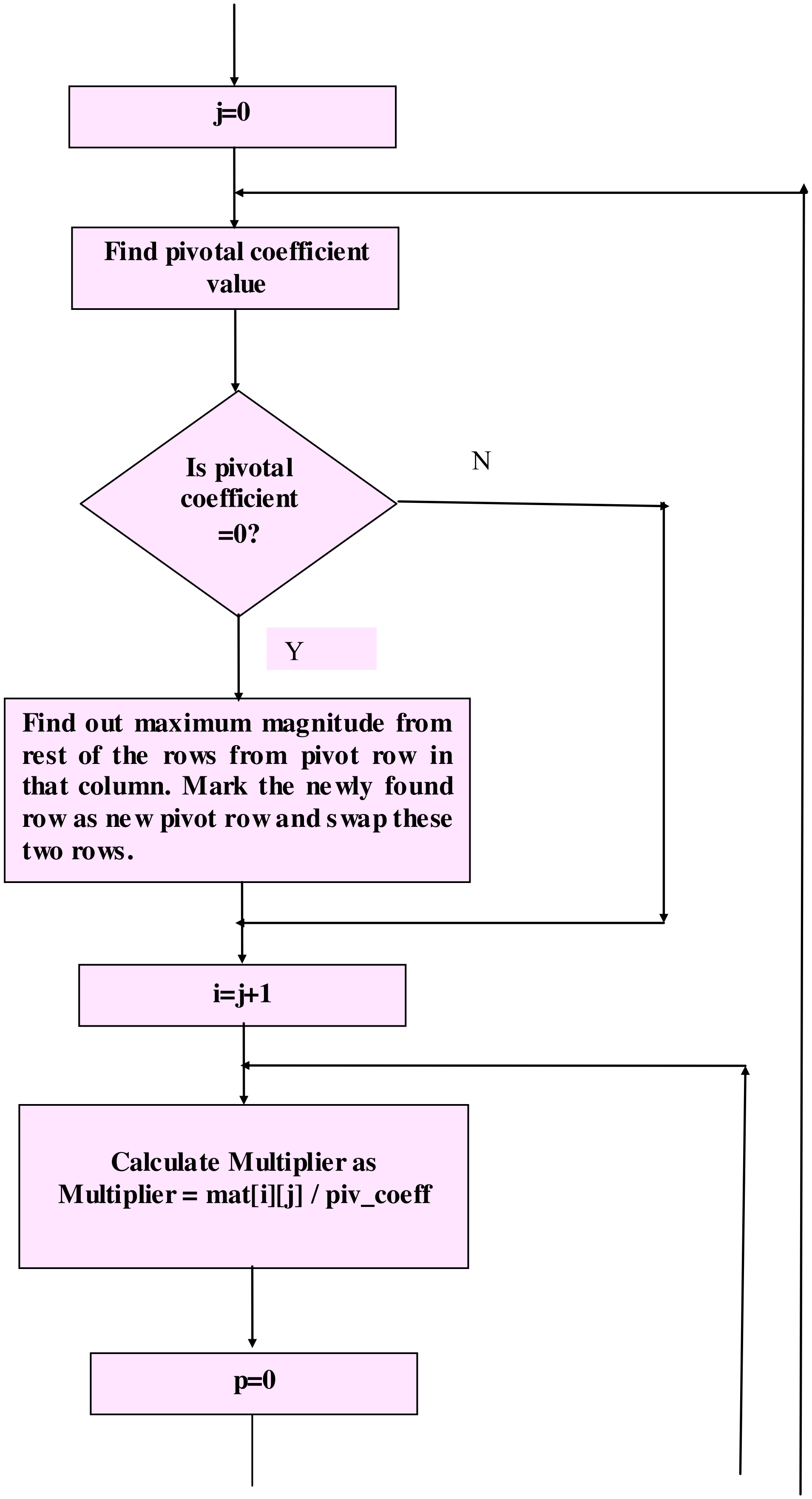
3.1 SEQUENTIAL GEPP

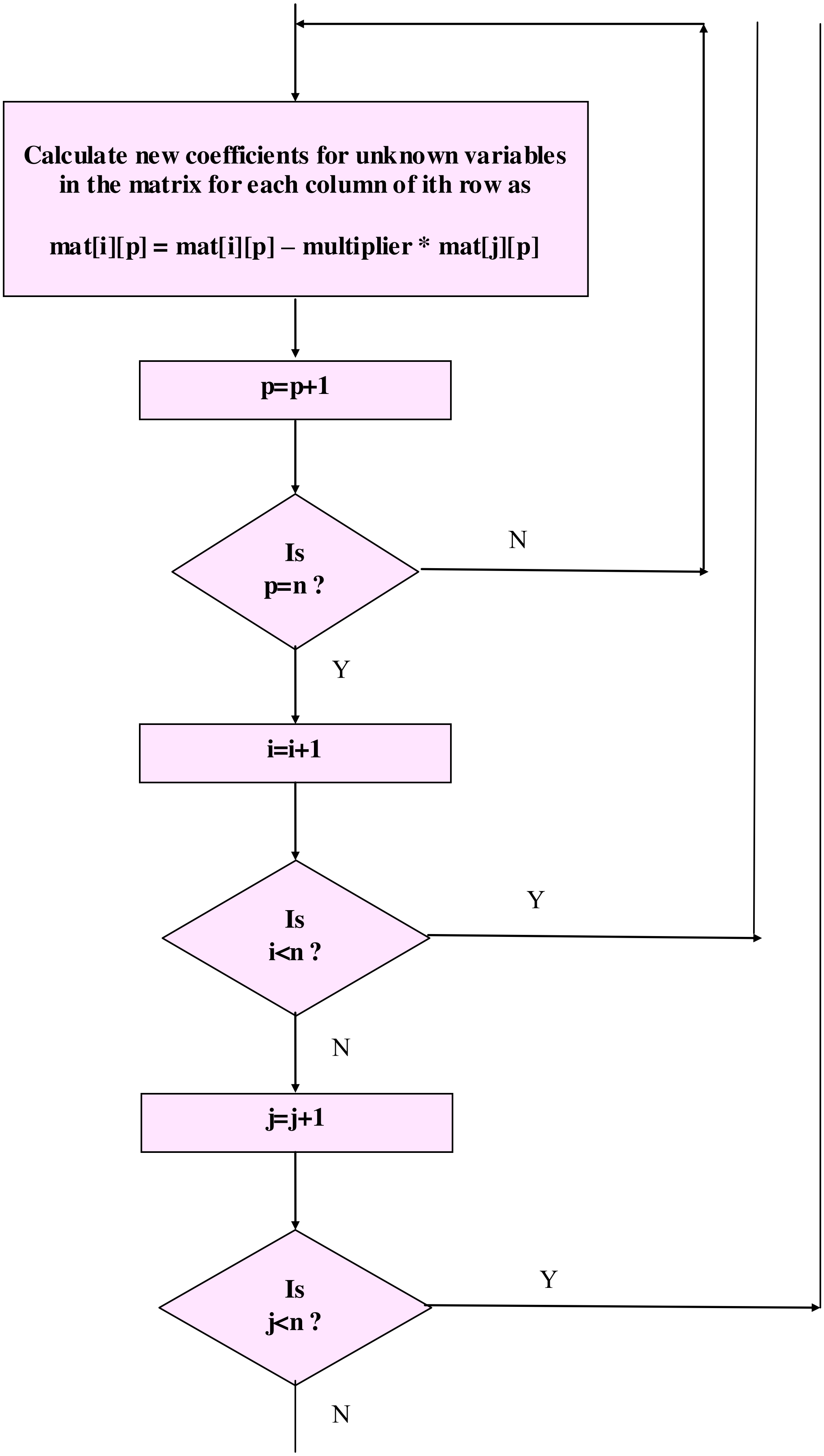
Gaussian elimination to solve a system of n equations for n unknown variables require [n(n+1) / 2] divisions, [$(2n^3 + 3n^2 - 5n)/6$] multiplications, and [$(2n^3 + 3n^2 - 5n)/6$] subtractions, for a total of approximately ($2n^3 / 3$) operations. So it has a complexity of $O(n^3)$. For sequential approach of this algorithm into machine - no need to maintain a separate array to hold vector b, since the manipulation of the elements of b are identical to the manipulations of the elements of A, so b can be adjoined to A and create an augmented matrix with n rows and n+1 columns.

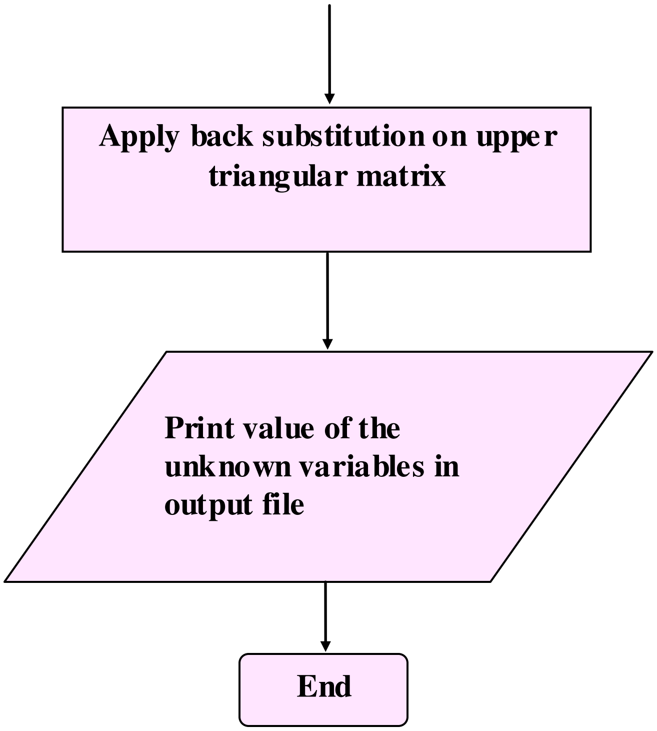
3.1.1 FLOWCHART

Flowchart of the sequential gauss elimination with partial pivoting is as follows :


Start
Get value for n- Number of Unknown Variables
Fill up augmented matrix mat[i][j] with data read from file


j=0
Find pivotal coefficient value
Is pivotal coefficient =0?
N
Y
Find out maximum magnitude from rest of the rows from pivot row in that column. Mark the newly found row as new pivot row and swap these two rows.
i=j+1
Calculate Multiplier as
Multiplier = mat[i][j] / piv_coeff
p=0


Calculate new coefficients for unknown variables in the matrix for each column of ith row as
mat[i][p] = mat[i][p] – multiplier * mat[j][p]
p=p+1
Is p=n ?
N
Y
i=i+1
Is i<n ?
Y
N
j=j+1
Is j<n ?
Y
N


Apply back substitution on upper triangular matrix
Print value of the unknown variables in output file
End

3.1.2 IMPLEMENTATION OF SEQUENTIAL GEPP

The algorithm has been developed by C language in the Linux environment.

int main () :

```
get_mat();

        for(j=0;j<=n-1;j++)
        {
                check_mat(j);
                for(i=j+1;i<n;i++)
                {
                        new_coeff(i,j);
                }
        }
 get_ans();
 print_ans();
```

At first the augmented matrix will be filled up by calling the get_mat ( ) function.

void get_mat ( ) :

The total number of unknown variables n of the linear system is read from a separate file "no_of_var".

```
if ((fp1=fopen("no_of_var","r"))==NULL)
        {
          printf("\nError!Cannot open the File\n");
          exit(1);
        }
        fscanf(fp1,"%d",&n);
        fclose(fp1);
        printf("The No. of Variables is =\t %d\n",n);
```

The augmented matrix is filled up by values which are stored in a input file "input_mat". This input file is created by executing another program which uses rand ().

```
   if ((fp2=(fopen("input_mat","r")))==NULL)
        {
           printf("\n Error! Cannot open the File\n");
           exit(1);
        }
        for(i=0;i<n;i++)
        {
           for(j=0;j<n+1;j++)
               fscanf(fp2,"%f",&mat[i][j]);
        }
   fclose(fp2);
```

After that within a for loop, which execute within the column range starting from 0 up to (n-1) the check_mat () is called.

void check_mat (int) : It checks weather the pivotal coefficient is zero or not. If it is zero at the (n-1)th step then the system is inconsistent, doesn't have any solution.

Otherwise it will find the highest valued coefficient from rest of the rows on that column. Then it will call the swap_rows ().

```
if(mat[c][c]==0)
        {
            if(c==n-1)
                {
                  printf("\nThe system of linear equations
is inconsistent");
                  exit(1);
                }
             for(i=c+1;i<=n-1;i++)
                {
                        if(mat[i][c]>magn)
                        {
                                magn=mat[i][c];
                                p=i;
                        }
                }
             swap_rows(c,p);
```

void swap_rows (int, int) : This function swap the current pivot row with the row which contains the highest magnitude found in the check_mat function.

```
        for(i=0;i<=n;i++) temprow[i]=mat[r1][i];
        for(i=0;i<=n;i++) mat[r1][i]=mat[r2][i];
        for(i=0;i<=n;i++) mat[r2][i]=temprow[i];
```

Now in main() within another sub loop which start with the row value (j+1) to last row, call the new_coeff function.

void new_coeff (int, int) : This function calculate the multiplier value, which will be used to calculate the new coefficients for all the columns for the given row. In this manner every time a new row is found where all the coefficients before the pivotal coefficient becomes zero, and the matrix becomes a upper triangular matrix.

```
multiplier=mat[r][c]/piv_coeff;
          for(i=0;i<=n;i++)
          {
                    mat[r][i]=mat[r][i]-multiplier*mat[c][i];
          }
```

In main (), after completion of the two for loops, we will get the a complete upper triangular matrix with n number of variables. Now the get_ans ( ) is called.

void get_ans ( ) : After getting a upper triangular matrix, the back substitution step is applied to calculate the value of the n variables starting from the last equation that will calculate the value of x_n variable, the calculation will go up until the value of x_1 is got.

```
for(i=n-1;i>=0;i--)
          {
                    ans[i]=mat[i][n]/mat[i][i];
                    for(j=i+1;j<=n-1;j++)
                    {
                           term=term+mat[i][j]*ans[j];
                    }
                    ans[i]=(1/mat[i][i])*(mat[i][n]-term);
                    term=0;
          }
```

At last the print_ans( ) function is called from main( ) function which will store the values of the variables in the output file.

void print_ans( ) :

```
fp=fopen("ans_file","w");
          for(i=0;i<n;i++)
          {
                    fprintf(fp,"x%d =\t",i+1);
                    fprintf(fp,"%f\n\n",ans[i]);
          }
          fclose(fp);
```

CHAPTER 4

VARIOUS PARALLEL GAUSS ELIMINATION TECHNIQUES

Chapter -4

Various Parallel Gauss Elimination Techniques

In this chapter we are going to discuss three popular parallel techniques for Gauss Elimination method. But before going to the detailed discussion we will first compare serial computing to parallel computing and why parallelism on Gauss Elimination.

WHAT IS PARALLEL PROGRAMMING

Parallel programming is a programming technique that allows one to explicitly indicate how different portions of the computation may be executed concurrently by different processors.

4.1. SERIAL VERSUS PARALLEL COMPUTING

Serial computing:

- Executing program on single processor (CPU)
- The CPU performs one operation at a time
- Single Instruction, Single Data (SISD)
- Single task is implemented

Parallel Computing:

- Executing program on more than one processor
- Simultaneous use of multiple computing resources to solve a problem
- Computing resources- Processors, Memory, I/O
- Breaks serial task into multiple tasks
- Works on tasks simultaneously
- Coordinates those tasks

4.2. WHY PARALLELISM ON GAUSS ELIMINATION WITH PARTIAL PIVOTING

Most of the operations occur inside the innermost for loop. A study of the algorithm's data dependences revels that both the innermost for loop and the middle for loop can be executed in parallel. Once the pivot row has been found, the modifications to all unmarked rows may occur simultaneously. Within each row, once the multiplier has been computed, modifications to elements i+1 through n-1 of each row may occur simultaneously. Hence the algorithm is well suited to parallelization.

4.3. OVERVIEW OF VARIOUS TYPES OF PARALLELIZATION OF THE ALGORITHM

There are following types of parallel approaches for the gauss elimination.

4.3.1 ROW ORIENTED APPROACH

In this algorithm first the processes participate in a tournament to determine the pivot row. Then the process controlling the pivot rows broadcasts it to the other processes. After the broadcast step, all of the processes use the pivot row to reduce the portions of the sub matrices they control. Once this has been done, the processes again participate in a tournament to determine the next pivot row. Combining the communication steps this parallel algorithm has overall message latency $\Theta(n \log p)$ and overall message transmission time $\Theta(n^2 \log p)$. The computational complexity is $\Theta(n^3/p)$. The total communication overhead (communications among the primitive tasks lead to the overhead of a parallel algorithm, minimizing parallel overhead is an important goal of parallel algorithm design) across p processes is $\Theta(n^2\ p \log p)$. The scalability (the level of parallelism increases at least linearly with the problem size) function of this parallel system is $(C^2\ p \log^2\ p)$, which shows poor scalability.

4.3.2 COLUMN ORIENTED APPROACH

In this algorithm one primitive task is associated with each column of A and another primitive task with vector b. during iteration **i** of the algorithm, the task controlling column i of A is responsible for finding the candidate element with the largest magnitude. It must only consider rows that have not yet been used as pivot rows. In a single iteration the column oriented algorithm spends Θ(n) time identifying the pivot row. The overall message latency is Θ(n log p) and overall message transmission time is $\Theta(n^2 \log p)$. The agglomeration (how those primitive tasks will run in actual parallel computer of different architectures) of this algorithm ensures that the workload remains balanced as the algorithm progresses. The computational complexity is $\Theta(n^3/p)$, same as row oriented algorithm, so the scalability is also the same.

4.3.3 PIPELINED ROW ORIENTED APPROACH

The RO and CO algorithms are synchronous in the sense that they neatly divide the parallel program's execution into communication and computation phases. The disadvantage of the synchronous approach is that processes are not performing computations during the broadcast steps, and the cumulative time complexity of the broadcast $\Theta(n^2 /p)$, is large enough to ensure the parallel algorithm has poor scalability. This implementation replaces the broadcast step with a series of point-to-point messages being sent around a ring of processes. By pipelining the flow of messages, the parallel algorithm has two advantages. First it facilitates asynchronous execution processes can reduce their portions of the augmented matrix as soon as the pivot rows are available. Second, it allows processes to effectively overlap communication time with computation time. Total message transmission time is $\Theta(n^2)$. The sequential time is $\Theta(n^3)$, and the parallel overhead is Θ(np). Hence the scalability function is *C*. Assuming n is large enough to ensure that message transmission time essentially overlaps with computation time, this parallel system is perfectly scalable.

4.4. PARALLEL PROGRAMMING ISSUES

The main goal of writing a parallel program is to get better performance over the serial version. With this in mind, there are several issues that one needs to consider while designing the parallel code to obtain the best performance possible within the constraints of the problem being solved. These issues are

- Load balancing
- Minimizing communication
- Overlapping communication and computation

Each of these issues are discussed in the following sections.

4.4.1. LOAD BALANCING

Load balancing is the task of equally dividing work among the available processors. This can be easy to do when the same operations are being performed by all the processors or threads (on different pieces of data). It is not trivial when the processing time depends upon the data values being worked on. When there are large variations in processing time, one may be required to adopt a different method for solving the problem.

4.4.2 MINIMIZING COMMUNICATION

Total execution time is a major concern in parallel programming because it is an essential component for comparing and improving all programs. Three components make up execution time

- Computation time
- Idle time
- Communication time

Computation time is the time spent performing computations on the data. Ideally, one would expect that if we have N processors working on a problem, we should be able to finish the job in 1/Nth the time of the serial job. This would be the case if all the processors' time were spent in computation.

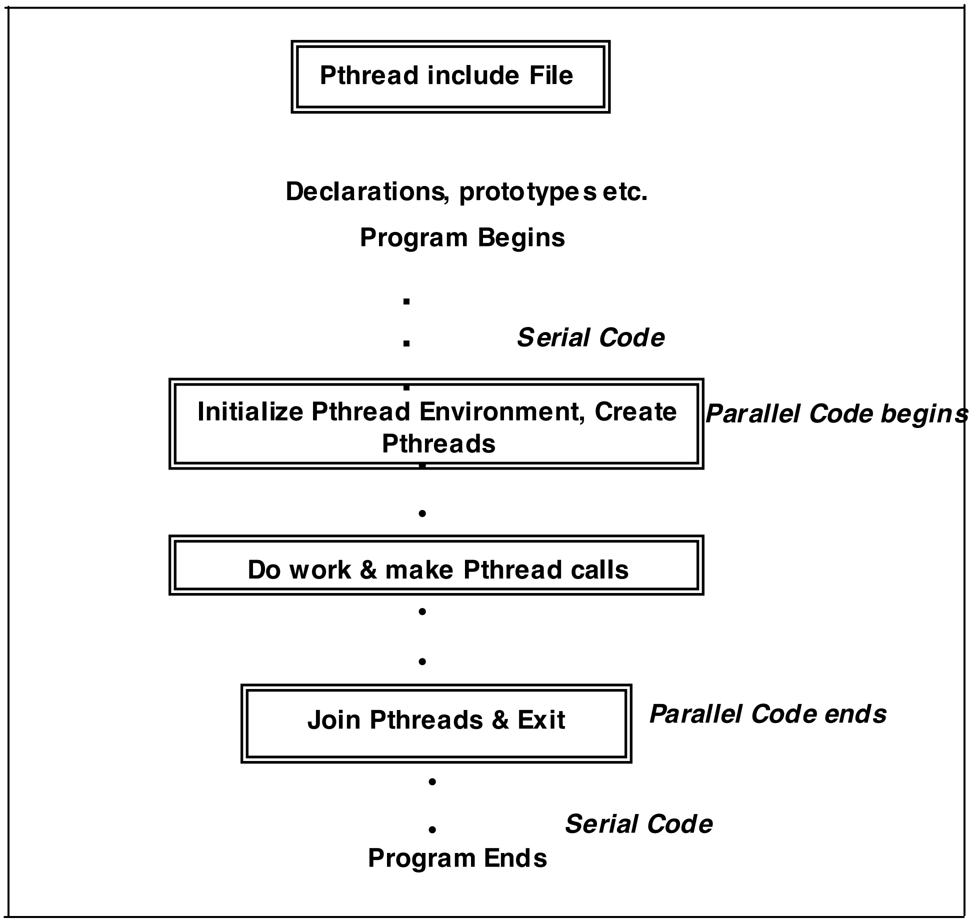
Idle time is the time a processor spends waiting for data from other processors. During this time, the processors do no useful work. An example of this is the ongoing problem of dealing with input and output (I/O) in parallel programs.

Communication time is the time it takes for processors to read from & write into the shared memory. Serial programs do not use inter-process communication. Therefore, we must minimize this use of time to get the best performance improvements.

4.4.3 OVERLAPPING COMMUNICATION AND COMPUTATION

There are several ways to minimize idle time within processors, and one way is overlapping communication and computation. It involves occupying a processor with one or more new tasks while it waits for communication to finish so it can proceed on another task. Careful use of non-blocking communication and data unspecific computation make this possible.

4.5. GENERAL PTHREAD PROGRAM STRUCTURE



CHAPTER-5

OVERVIEW OF POSIX THREADS

5.1 OVERVIEW OF MULTIPROCESSOR ARCHITECTURE

5.2 THREAD AND ITS ARCHITECTURE

5.3 WHAT ARE PTHREADS

5.4 ROLE OF PTHREADS IN PARALLEL PROGRAMMING

Chapter -5

Overview of Pthreads

In shared memory multiprocessor architectures, such as SMPs, threads can be used to implement parallelism. Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads.

This section provides an introduction to concepts, motivations, and design considerations for using Pthreads. Each of the three major classes of routines in the Pthreads API are then covered: Thread Management, Mutex Variables, and Condition Variables.

5.1. MULTI-PROCESSOR SYSTEMS

Multiprocessor systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory and peripheral devices. Therefore they are also known as parallel systems or tightly-coupled systems.

The multiple-processor systems in use today are of two types. Some systems use asymmetric multiprocessing, in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have pre-defined tasks. This scheme defines a master-slave relationship. The most common systems use symmetric multiprocessing (SMP) in which each processor performs all tasks within the operating system's means that all processors are peers; no master-slave relationship exists between processors. The benefit o this model is that many processes can run simultaneously- N processes can run if there are N CPUs without causing a significant deterioration of performance. Virtually all modern operating systems including Windows, Windows XP, Linux and Mac OS X now provide support for SMP.

Multiprocessor systems have three main advantages:

1. **Increased Throughput** : By increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with N processors is however not N.When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead,plus contention for shared resources lowers the speed-up ratio a bit.
2. **Economy of Scale** : Multiple processor systems can cost less than equivalent multiple single-processor systems because they can share peripherals, mass storage and power supplies. If several programs operate on the same set of date, its cheaper to store those data on one disk and to have all the processors share them than to have many computers with local disks and many copies of the data.
3. **Increased Reliability** : If functions can be distributed properly among several processors, then the failure of one processor will not halt the whole system, only slows it down.

5.2. THREAD AND ITS ARCHITECTURE

Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread. To go one step further, we can imagine a main program (a.out) that contains a number of procedures. Then all of those procedures being able to be scheduled to run simultaneously and/or independently by the operating system would describe a "multi-threaded" program.

The independent flow of control is accomplished because a thread maintains its own:

- Stack pointer
- Registers
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals

- Thread specific data.

Before understanding a thread, one first needs to understand a UNIX process. A process is created by the operating system, and requires a fair amount of "overhead". Processes contain information about program resources and program execution state, including:

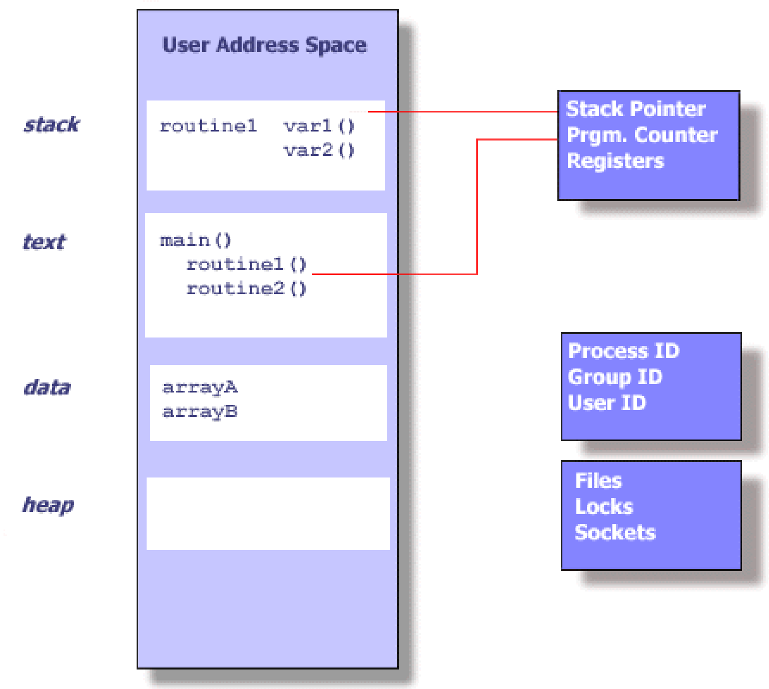
- Process ID, process group ID, user ID, and group ID
- Environment
- Working directory.
- Program instructions
- Registers
- Stack
- Heap
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code.

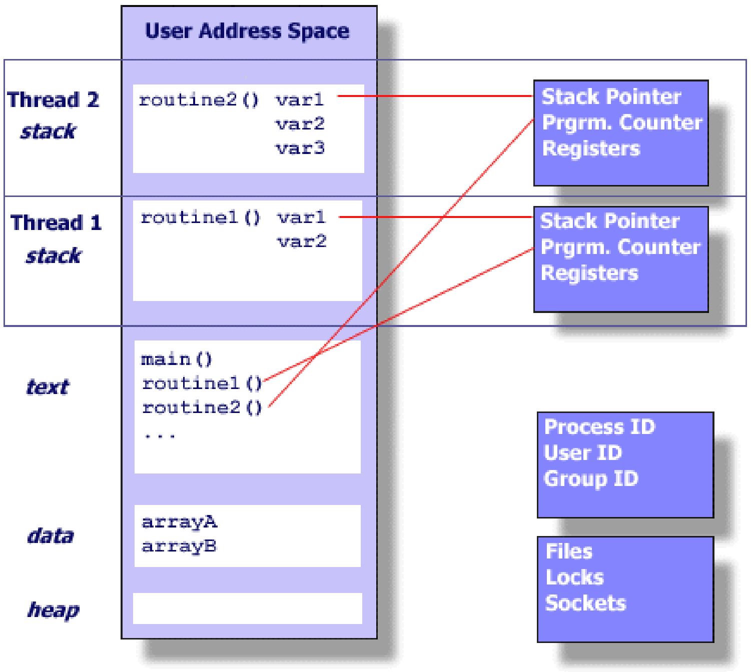
Thus, a thread has the following characteristics:

- Exists within a process and uses the process resources.
- Has its own independent flow of control as long as its parent process exists.
- Duplicates only the essential resources it needs to be independently schedulable
- May share the process resources with other threads that act equally independently.
- Dies if the parent process dies - or something similar.
- Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.
- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads because threads within the same process share resources.
- Two pointers having the same value point to the same data.

- Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.



AN UNIX PROCESS



THREADS WITH AN UNIX PROCESS

5.3. WHAT ARE PTHREADS

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a specification for thread behavior, not an implementation. In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's.

Pthreads are defined as a set of C language programming types and procedure calls, implemented with a `pthread.h` header/include file and a thread library - though this library may be part of another library, such as `libc`.Numerous systems implement the Pthreads specification, including Solaris, Linux, Mac OS X,UNIX etc.

5.4. ROLE OF PTHREADS IN PARALELL PROGRAMMING

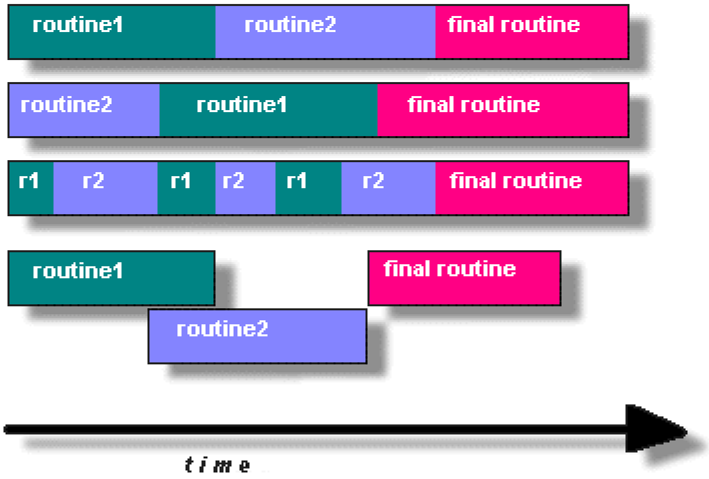
The primary motivation for using Pthreads is to realize potential program performance gains. When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

On modern, multi-CPU machines, pthreads are ideally suited for parallel programming, and whatever applies to parallel programming in general, applies to parallel pthread programs.

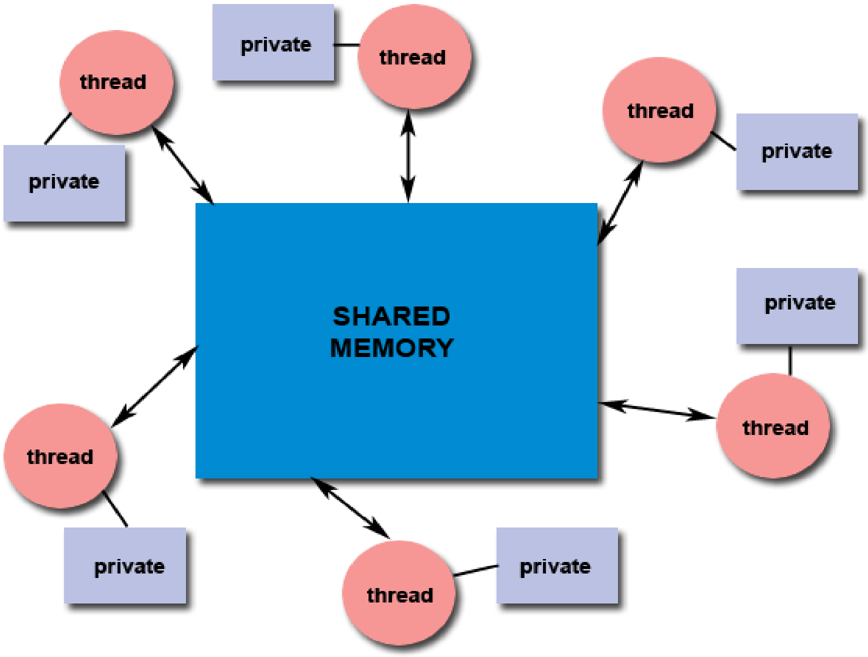
Programs having the following characteristics may be well suited for pthreads:

- Work that can be executed, or data that can be operated on, by multiple tasks simultaneously.
- Block for potentially long I/O waits .
- Use many CPU cycles in some places but not others.
- Must respond to asynchronous events.
- Some work is more important than other work (priority interrupts).

In order for a program to take advantage of Pthreads, it must be able to be organized into discrete, independent tasks which can execute concurrently. For example, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.



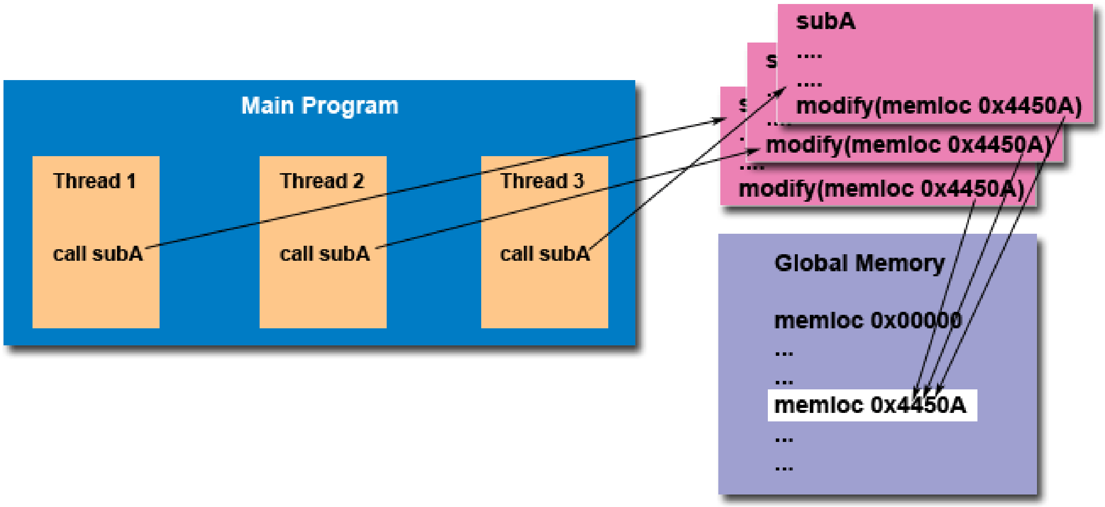
All threads have access to the same global, shared memory. Threads also have their own private data. Programmers are responsible for synchronizing access (protecting) globally shared data.



SHARED MEMORY MODEL

Also, parallel programming using pthreads ensures **thread-safeness.** Thread-safeness in a nutshell, refers an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions. For example, suppose an application creates several threads, each of which makes a call to the same library routine:

- This library routine accesses/modifies a global structure or location in memory.
- As each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time.
- If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe.



THREAD-SAFENESS

CHAPTER 6

DESIGN & IMPLEMENTATION OF PARALLEL GAUSS ELIMINATION WITH PARTIAL PIVOTING

6.1 FLOWCHART
- 6.1.1 COLUMN ORIENTED PGEPP FLOWCHART
- 6.1.2 ROW ORIENTED PGEPP FLOWCHART

6.2 IMPLEMENTATION DETAILS
- 6.2.1 COLUMN ORIENTED IMPLEMENTATION
- 6.2.2 ROW ORIENTED IMPLEMENTATION

6.3 KEY CHALLENGES FACED ON DEVELOPMENT

Chapter 6

Design & Implementation of Parallel Gauss Elimination with Partial Pivoting

This chapter contains the flowcharts for the row oriented and column oriented parallel gauss elimination with partial pivoting. Also contains the implementation details with code of the two different approaches.

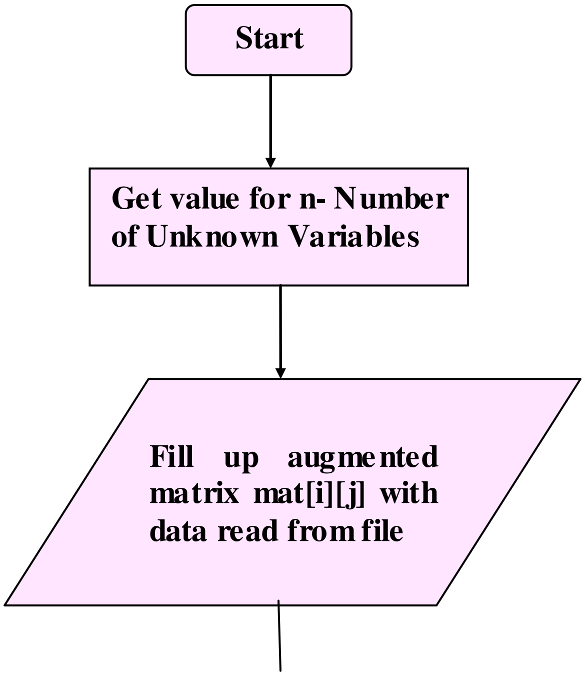
6.1 FLOWCHART

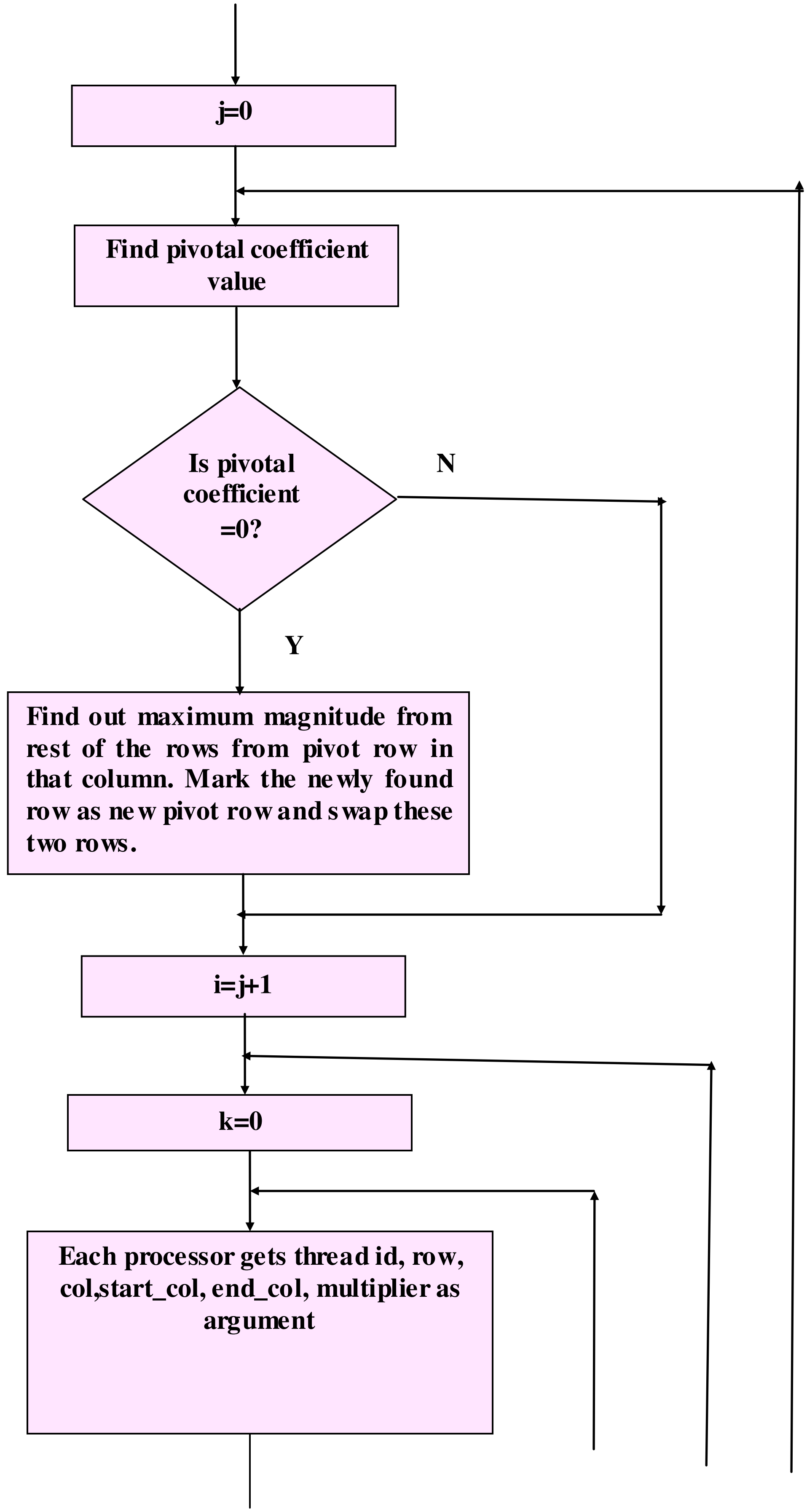
Two types of parallel approaches are developed for the gauss elimination with partial pivoting

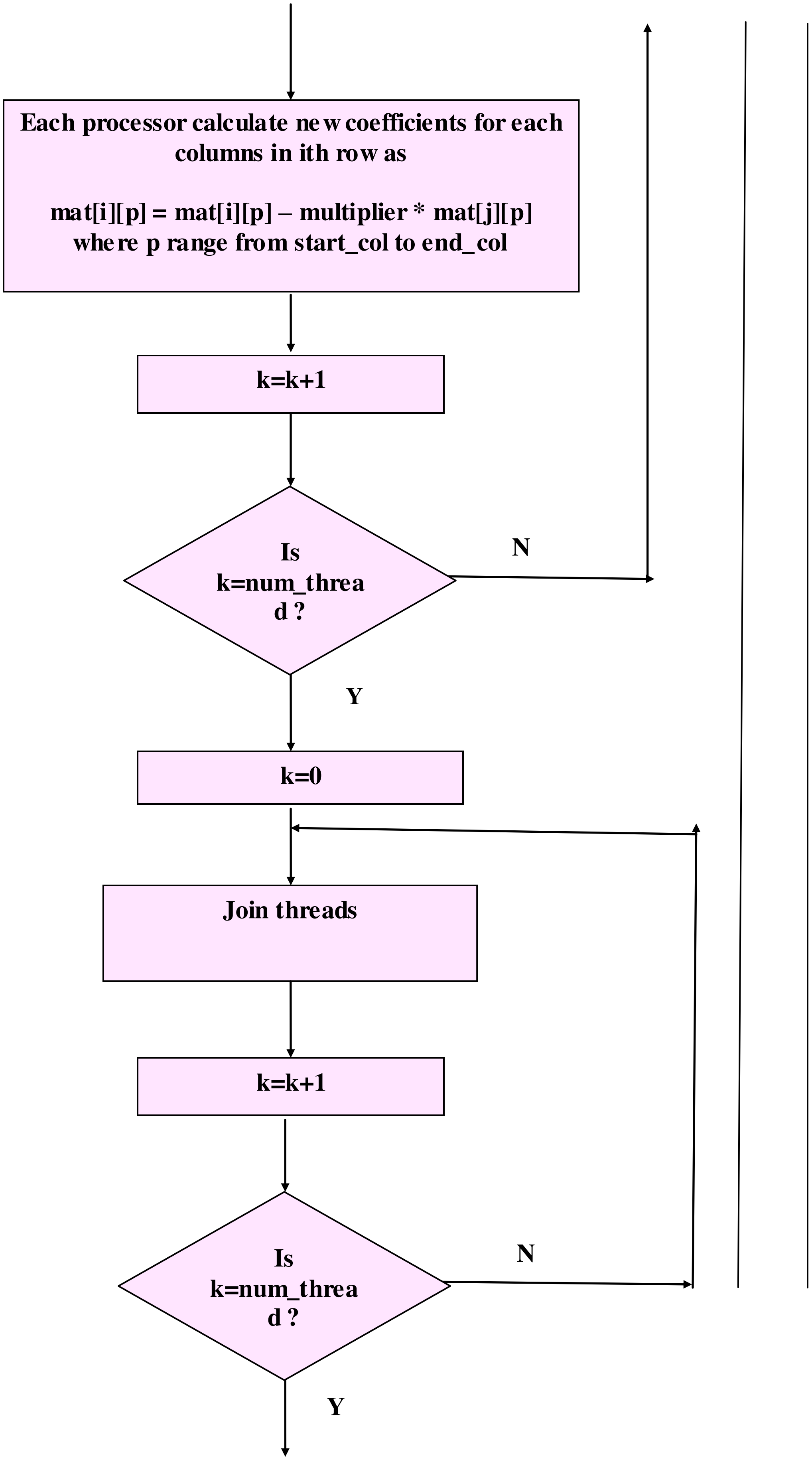
.

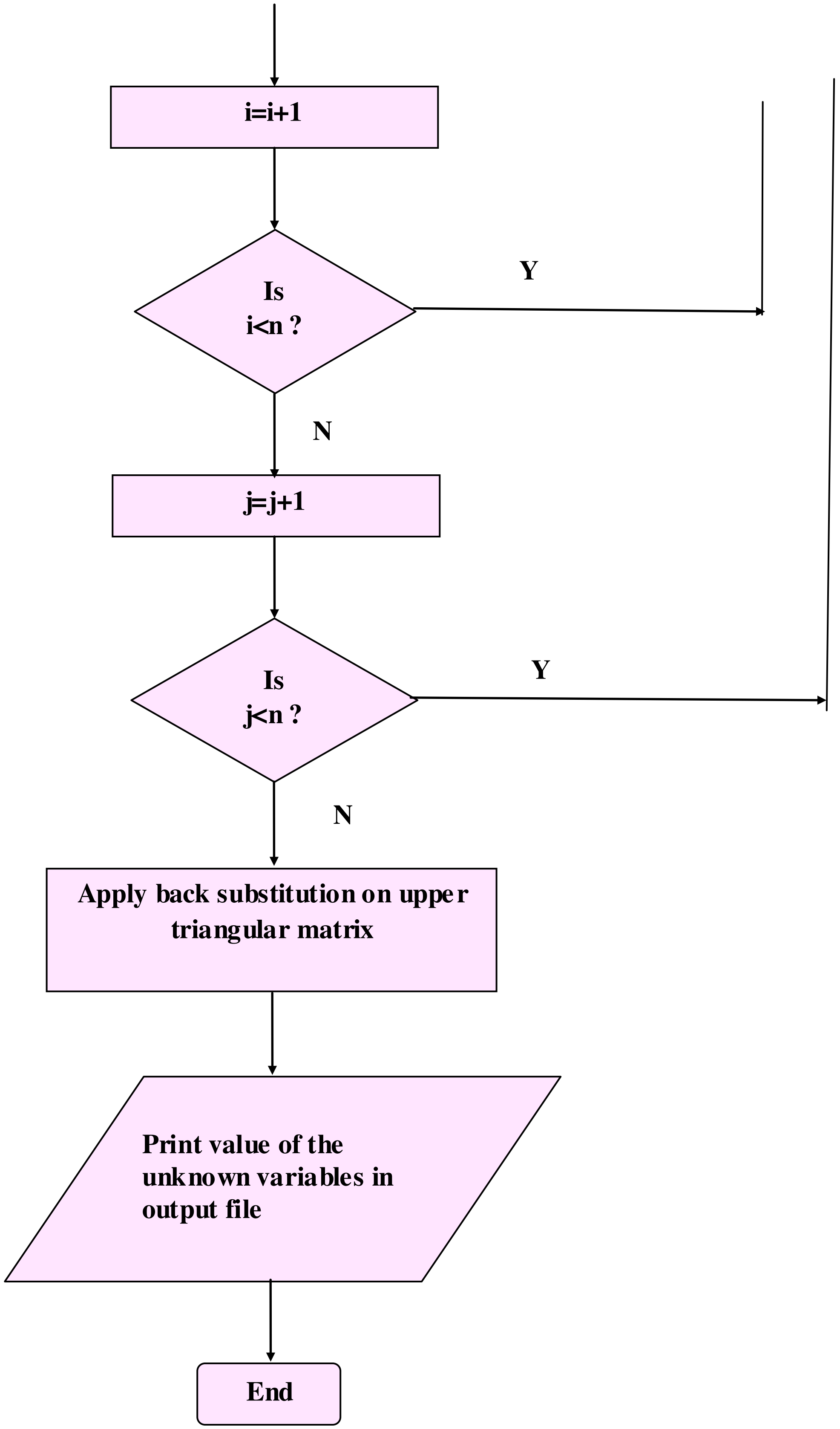
6.1.1 COLUMN ORIENTED PGEPP FLOWCHART

This is the pictorial representation of the program logic that will be further implemented to develop the column oriented parallel GEPP.


Start
Get value for n- Number of Unknown Variables
Fill up augmented matrix mat[i][j] with data read from file

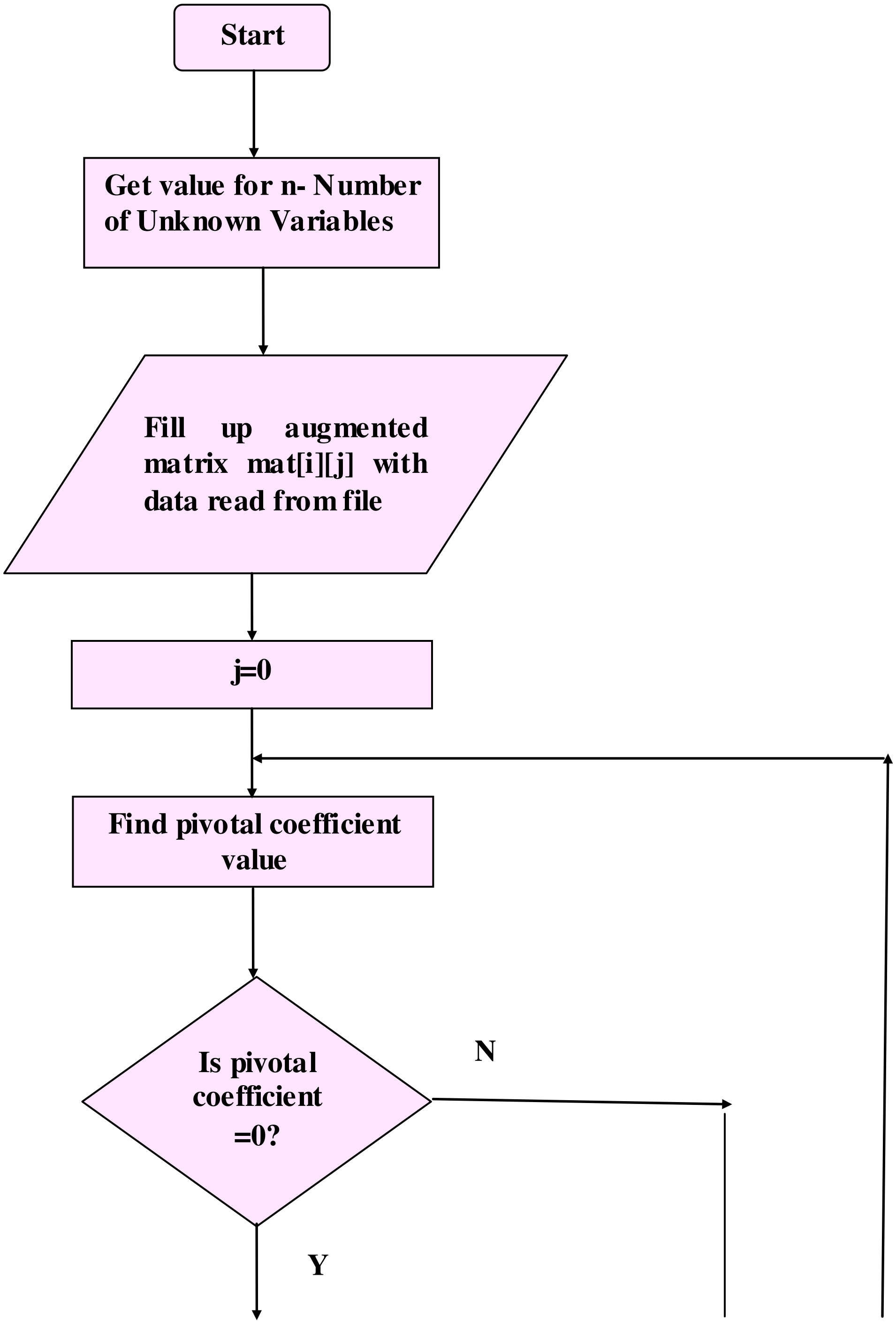

j=0
Find pivotal coefficient value
Is pivotal coefficient =0?
N
Y
Find out maximum magnitude from rest of the rows from pivot row in that column. Mark the newly found row as new pivot row and swap these two rows.
i=j+1
k=0
Each processor gets thread id, row, col,start_col, end_col, multiplier as argument

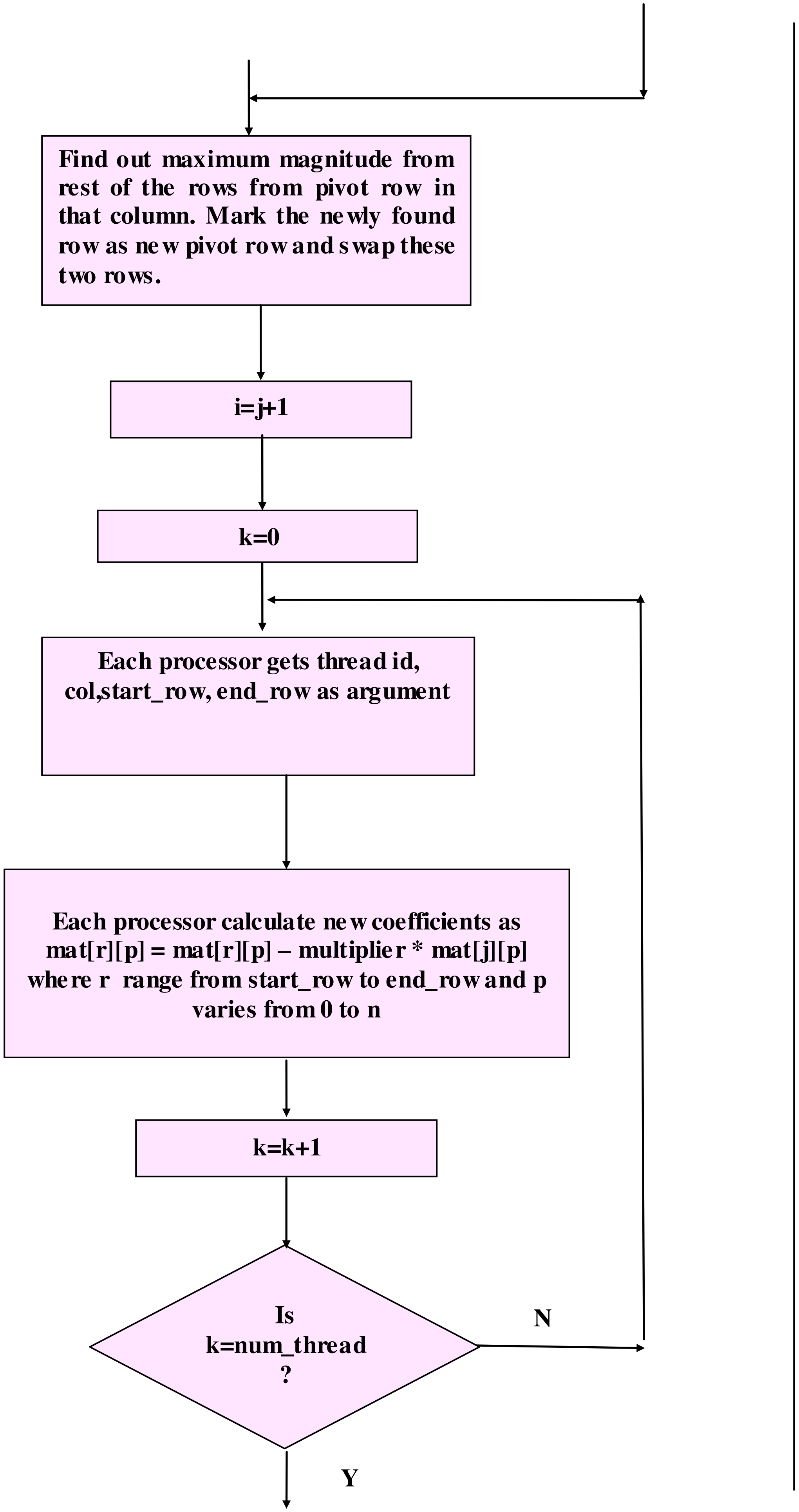

Each processor calculate new coefficients for each columns in ith row as
mat[i][p] = mat[i][p] – multiplier * mat[j][p]
where p range from start_col to end_col
k=k+1
Is
k=num_threa
d ?
N
Y
k=0
Join threads
k=k+1
Is
k=num_threa
d ?
N
Y

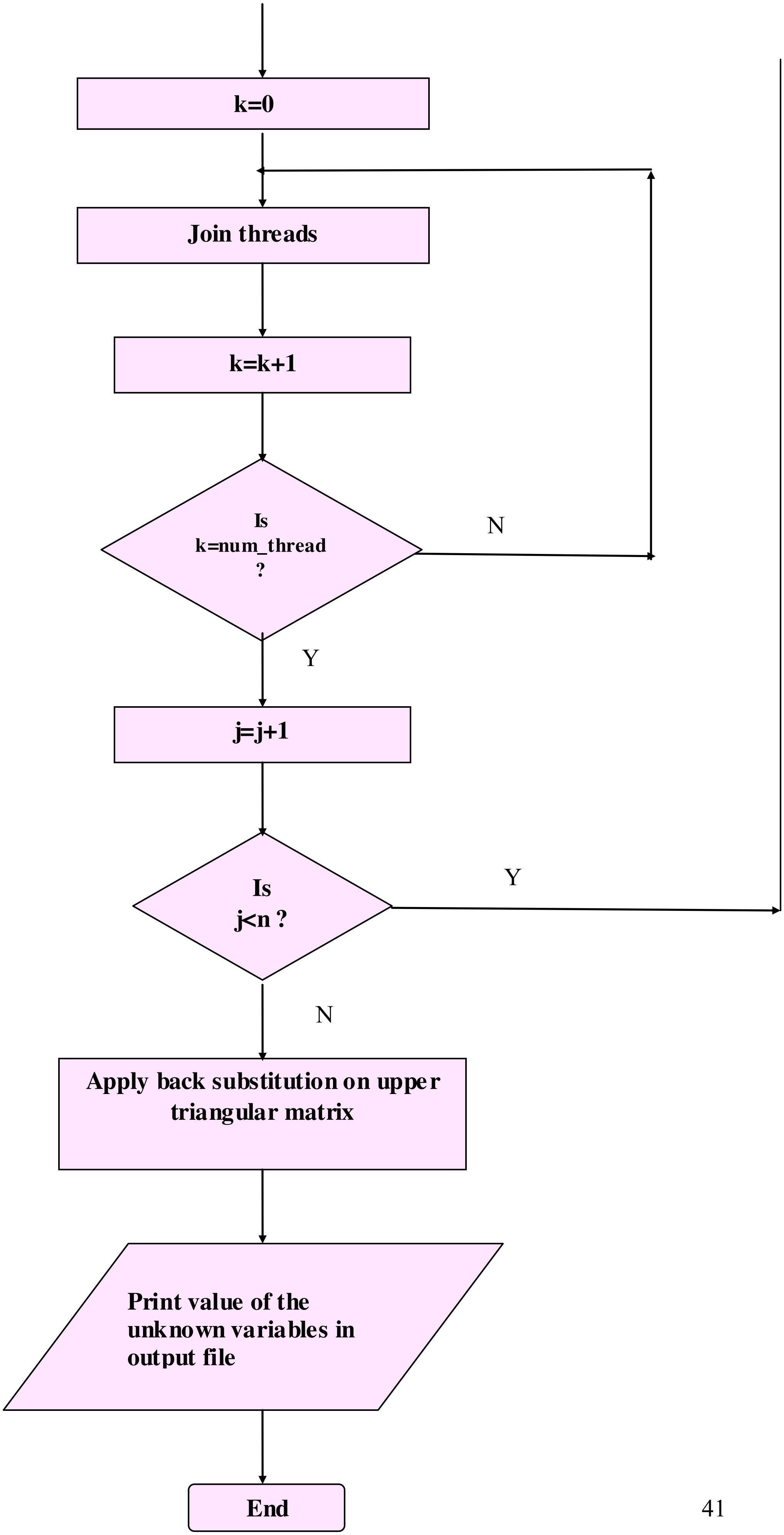

i=i+1
Is
i<n ?
Y
N
j=j+1
Is
j<n ?
Y
N
Apply back substitution on upper triangular matrix
Print value of the unknown variables in output file
End

6.1.1 ROW ORIENTED PGEPP FLOWCHART

This is the pictorial representation of the program logic that will be further implemented to develop the row oriented parallel GEPP.


Start
Get value for n- Number of Unknown Variables
Fill up augmented matrix mat[i][j] with data read from file
j=0
Find pivotal coefficient value
Is pivotal coefficient =0?
N
Y


Find out maximum magnitude from rest of the rows from pivot row in that column. Mark the newly found row as new pivot row and swap these two rows.
i=j+1
k=0
Each processor gets thread id, col,start_row, end_row as argument
Each processor calculate new coefficients as mat[r][p] = mat[r][p] – multiplier * mat[j][p] where r range from start_row to end_row and p varies from 0 to n
k=k+1
Is k=num_thread ?
N
Y


k=0
Join threads
k=k+1
Is
k=num_thread
?
N
Y
j=j+1
Is
j<n ?
Y
N
Apply back substitution on upper triangular matrix
Print value of the unknown variables in output file
End

6.2 IMPLEMENTATION DETAILS

The algorithm has been developed by C language in the Linux environment.

6.2.1 COLUMN ORIENTED IMPLEMENTATION

Following is the data structure created to pass the arguments to the child thread.

```
struct thread_data{
   int  thread_id;
   int  col;
   int start_col;
   int end_col;
   float multiplier;
  };
struct thread_data thread_data_arr[NUM_THREADS];
```

int main():

```
get_mat();
pthread_attr_init ( &pthread_attr );
for(j=0;j<=n-1;j++)
 {
   check_mat(j);
   piv_coeff=mat[j][j];

   for(i=j+1;i<n;i++)
    {
      for(k=0;k<NUM_THREADS;k++)
       {
        tread_data_arr[k].thread_id = k;
        thread_data_arr[k].row = i;
        thread_data_arr[k].col = j;
        thread_data_arr[k].start_col=
(k*((n+1)/NUM_THREADS));
        thread_data_arr[k].end_col=
((k*((n+1)/NUM_THREADS))+((n+1)/NUM_THREADS));
        thread_data_arr[k].multiplier= mat[i][j]/piv_coeff;

ret_val[k]=pthread_create(&pthread_hndl[k],&pthread_attr,ne
w_coeff_thread,(void *)&thread_data_arr[k]);
```

```
      if(ret_val[k]!=0)
          printf ("Error in pthread_create\n");
    }
       for(k=0;k<NUM_THREADS;k++)
          {
ret_val[k]=pthread_join(pthread_hndl[k],NULL);
       if(ret_val[k]!=0)
           printf ("Error in pthread_join\n");

         }
     }
}
get_ans();
print_ans();
pthread_exit(NULL);
```

In the main function of the program, get_mat, check_mat, swap_rows, get_ans and prints_ans functions work as same as the sequential program. new_coeff_thread function works as the child thread for the program. Within the row wise inner loop, threads are created and child thread is being called. this portion of the program runs parallel. The value of the start_col and end_col variables are calculated depending on the thread id. Before that the programs behaves sequentially. Joining of the threads follows afterward. After joining back substitution and prints_ans functions work sequentially.

new_coeff_thread(void *threadarg):

```
loc_data = (struct thread_data *) threadarg;
tid = loc_data->thread_id;
r = loc_data->row;
c = loc_data->col;
s_col= loc_data->start_col;
e_col= loc_data->end_col;
multipl= loc_data->multiplier;

for(i=s_col;i<e_col;i++)
 {
  mat[r][i]=mat[r][i]-multipl*mat[c][i];
 }
 pthread_exit ((void*) 0);
```

Each processor gets data from the threadarg and calculate new coefficients for the unknown variables of the all columns that starts from s_col upto e_col for the rth row.

6.2.2 ROW ORIENTED IMPLEMENTATION

Following is the data structure created to pass the arguments to the child thread.

```
struct thread_data{
   int  thread_id;
   int  col;
   int start_row;
   int end_row;
   };
```

int main():

```
get_mat();
pthread_attr_init ( &pthread_attr );
for(j=0;j<=n-1;j++)
{
  check_mat(j);
  i=j+1;
  p=((n-i)/NUM_THREADS);

  for(k=0;k<NUM_THREADS;k++)
   {
      thread_data_arr[k].thread_id = k;
      thread_data_arr[k].col = j;
      thread_data_arr[k].start_row = i;
      if(k!=(NUM_THREADS-1))
        {
        thread_data_arr[k].end_row = (i+(p-1));
        i=((i+(p-1))+1);
        }

        else
        {
          thread_data_arr[k].end_row = (n-1);
        }
        ret_val[k]=pthread_create(&pthread_hndl[k],&pthread
_attr,new_coeff_thread,(void *)&thread_data_arr[k]);
        if(ret_val[k]!=0)
          printf ("Error in pthread_create\n");
      }
 for(k=0;k<NUM_THREADS;k++)
   {
     ret_val[k]=pthread_join(pthread_hndl[k],NULL);
        if(ret_val[k]!=0)
          printf ("Error in pthread_join\n");
   }
  }
get_ans();
print_ans();
pthread_exit(NULL);
```

Same as the column oriented parallel program up to the call of the check_mat function is done sequentially. There is only initialization of the variable i instead of the for loop which represents the value of row. Variable p calculate the value number of rows that a single processor can process. After that the program behave parallel, by creating the threads and calling the child thread. new_coeff_thread is the child thread function. Each thread will get the start_row and end_row which are calculated by i, p and n. After creating, need to join the threads. At last the back substitution and print_ans functions behave sequentially same as column oriented parallel program.

new_coeff_thread(void *threadarg):

```
loc_data = (struct thread_data *) threadarg;
tid = loc_data->thread_id;
c = loc_data->col;
s_row= loc_data->start_row;
e_row= loc_data->end_row;

piv_coeff=mat[c][c];
for(r=s_row; r<e_row; i++)
 {
    multiplier=mat[r][c]/piv_coeff;
    for(i=0;i<=n;i++)
     {
      mat[r][i]=mat[r][i]-multiplier*mat[c][i];
      }
    }
pthread_exit ((void*) 0);
```

Within the 1st for loop the multiplier is calculated for each row that starts from s_row upto e_row. After that another for loop is created where the variable range from 0 to n, this loop actually calculate the new coefficients for the unknown variables for each column of the rth row.

6.3 Key challenges faced on development

The key challenges faced during this program development that were as follows :

In the sequential Gauss elimination algorithm there are two for loops present in the main program. One for column another for row, where row wise for loop in nested. Within these loops some functions are compulsory that have to be done sequentially. So the application of parallelism was not very easy going.

For column oriented approach of the parallelism we keep both the loop in main, total number of columns are divided in to number of threads. Since we worked on augmented matrix, the matrix format is (n X n+1). This (n+1) not necessarily always divided by total number of threads. So this calculation has been done directly by the thread id which was the main challenge.

For the row oriented approach of the parallelism initial loop is kept within main, only the row wise nested loop is eliminated, and the total number of rows for every j are sent to threads. To do this the main challenge was that how to divide the number of rows equally for each processor where for every j row is subtracted by 1. For every column, row is uniformly decreasing by 1. Another challenge was that when the number of row is lesser than the number of processor how to distribute them within processors. In that situation we decreased the number of threads according to the number of the rows.

CHAPTER 7

RESULTS, CONCLUSION & FURTHER WORK

7.1 EXPERIMENTAL RESULTS

7.2 CONCLUSIONS

7.3 FURTHER WORKS

Chapter 7

Results Conclusion & Further Work

This chapter contains the experimental results that we have after execution of the programs; on the basis of the experimental results the conclusion has been drawn. And a discussion on what the possible future works are possible for this project.

7.1 EXPERIMENTAL RESULTS

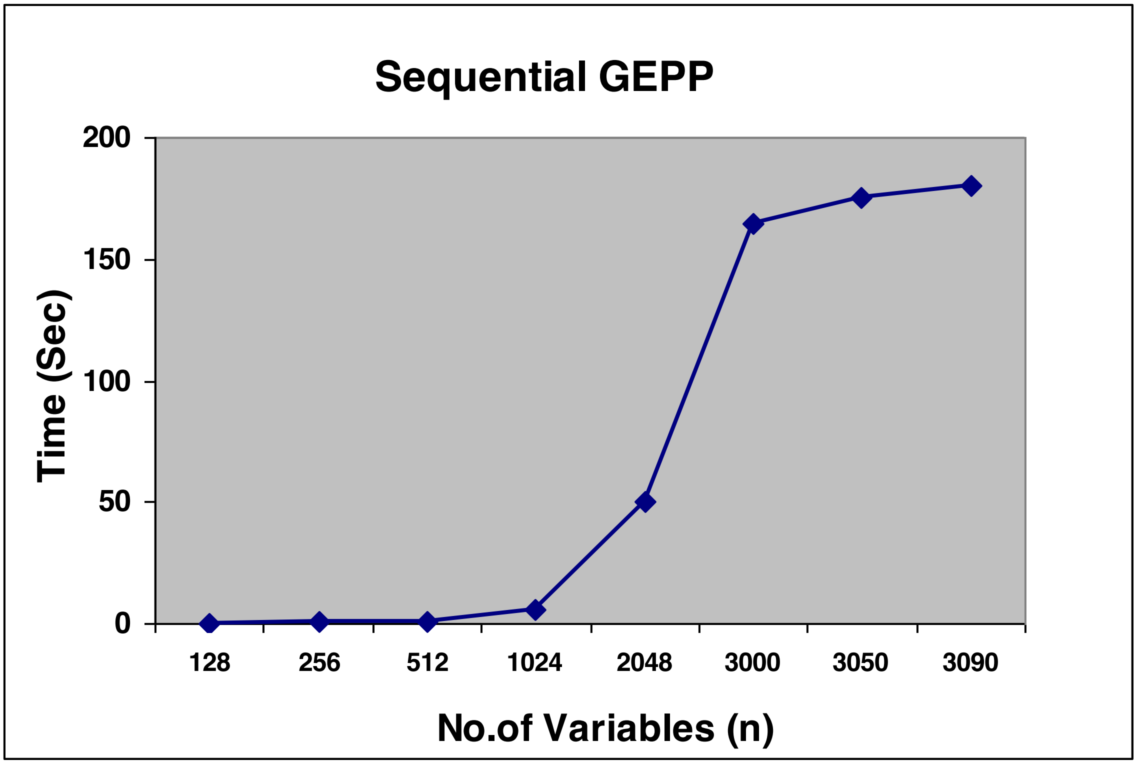
Following are the experimental results that we got after executing the programs sgepp.c, pgepp_pth_col.c and pgepp_pth_row.c which are the programs for sequential GEPP, column oriented GEPP and row oriented GEPP respectively. We run the programs with different number of variables (n) and note down in table format how much time it taken. Time is measured in seconds. To get the pictorial vision these tables or part of these tables are represented in graphical form.

(A) Time Table for the **Sequential Gauss Elimination with Partial Pivoting** program:

No. of Variables (n)	Time (Sec.)
128	0
256	1
512	1
1024	6

2048	53
3000	165
3050	175
3090	180

TABLE 1

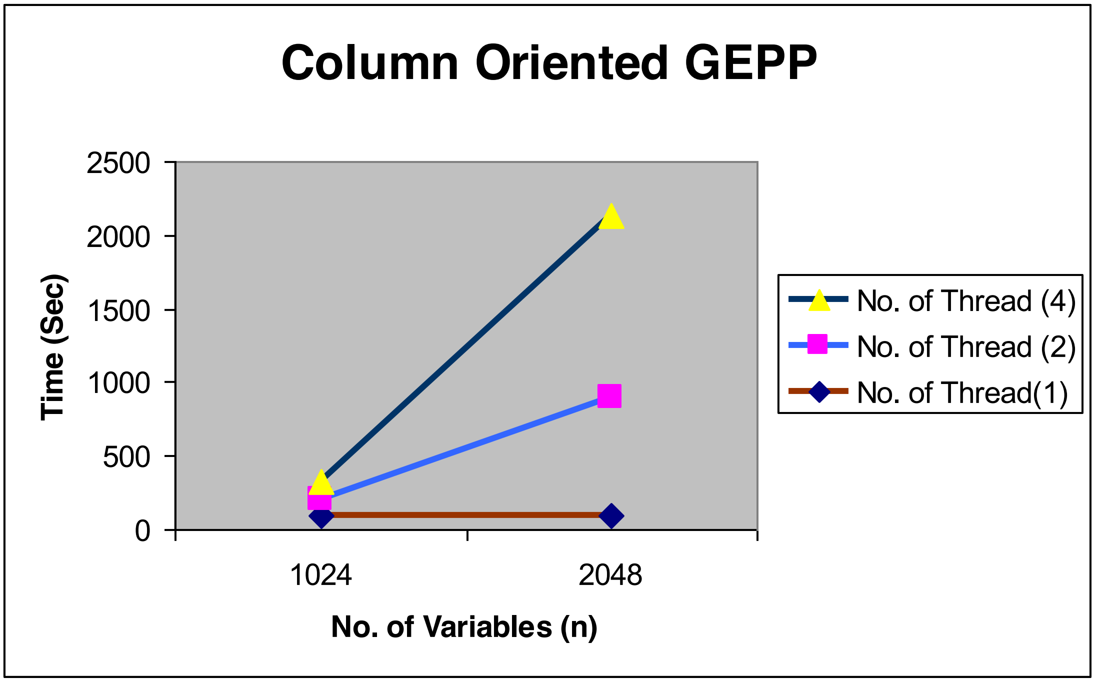


GRAPH 1

(B) Time table for **Column Oriented Parallel Gauss Elimination with Partial Pivoting** program:

No of Variable (n)	Time (Sec)		
	No. of Thread (1)	No. of Thread (2)	No. of Thread (4)
128	0	0	1
256	2	2	4
512	3	5	12
1024	97	103	122
2048	100	800	1230

TABLE 2

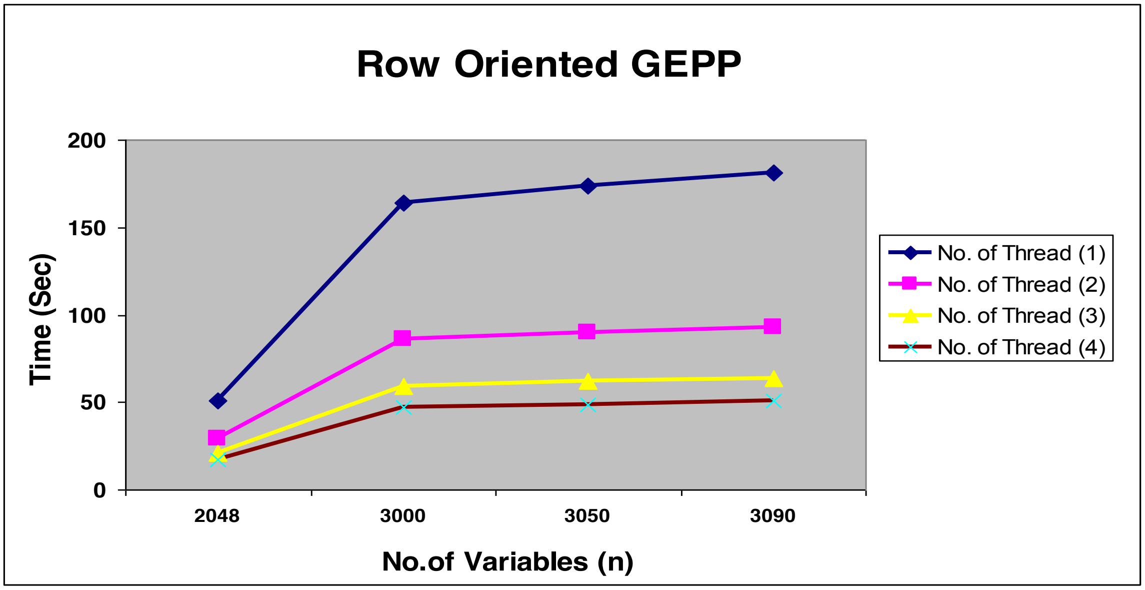


GRAPH 2

(C) Time table for **Row Oriented Parallel Gauss Elimination with Partial Pivoting** program:

No of Variable (n)	Time (Sec)			
	No. of Thread (1)	No. of Thread (2)	No. of Thread (3)	No. of Thread (4)
256	0	0	0	0
512	0	1	0	0
1024	6	4	4	5
2048	51	29	21	17
3000	164	86	59	47
3050	174	90	62	49
3090	181	93	64	51

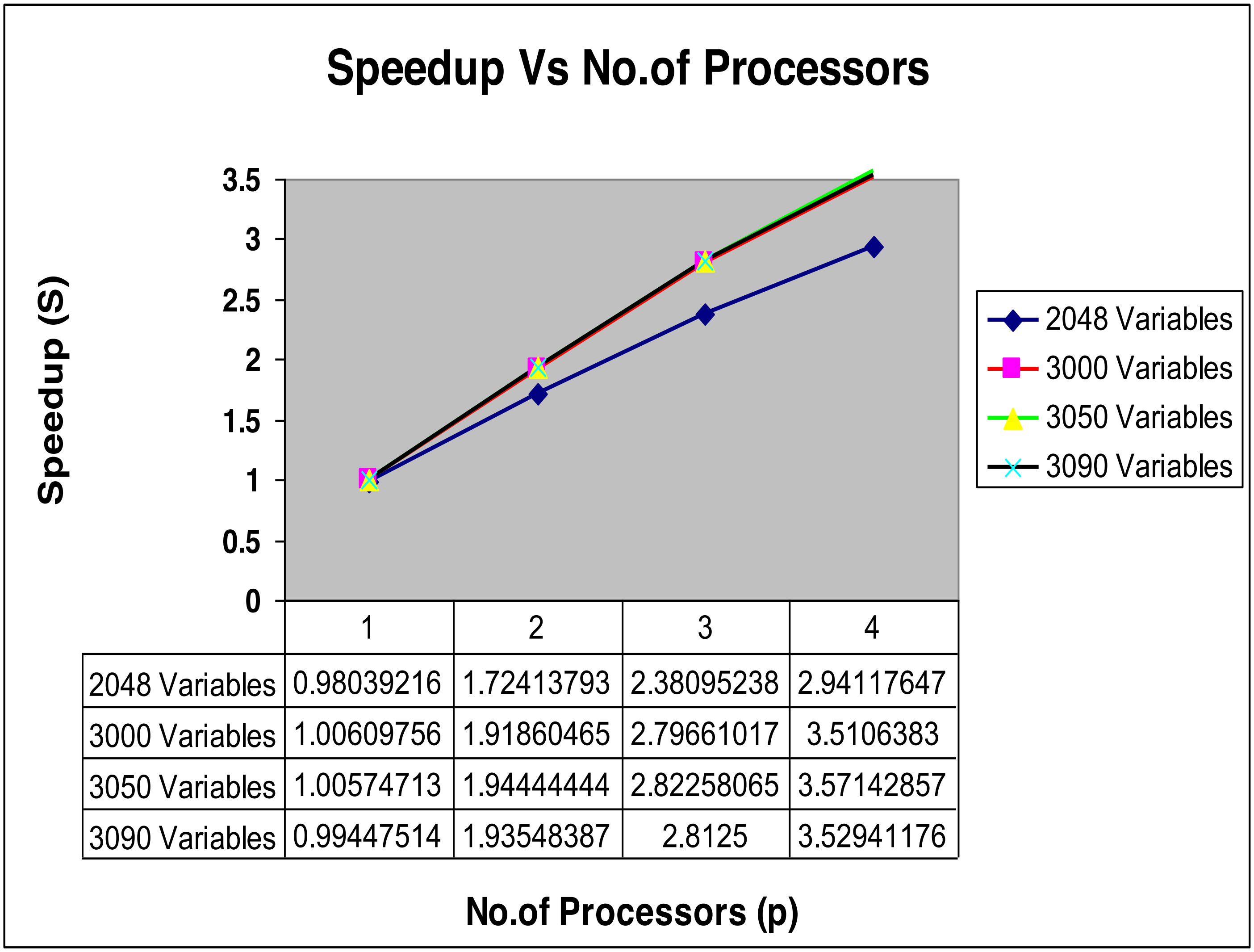
TABLE 3



GRAPH 3

(D) **Speedup** : The ratio between the time needed for the most efficient sequential algorithm to perform a computation and the time needed to perform the same computation on a machine incorporating pipelining and/or parallelism.

$$S = Tseq / Tpar.$$

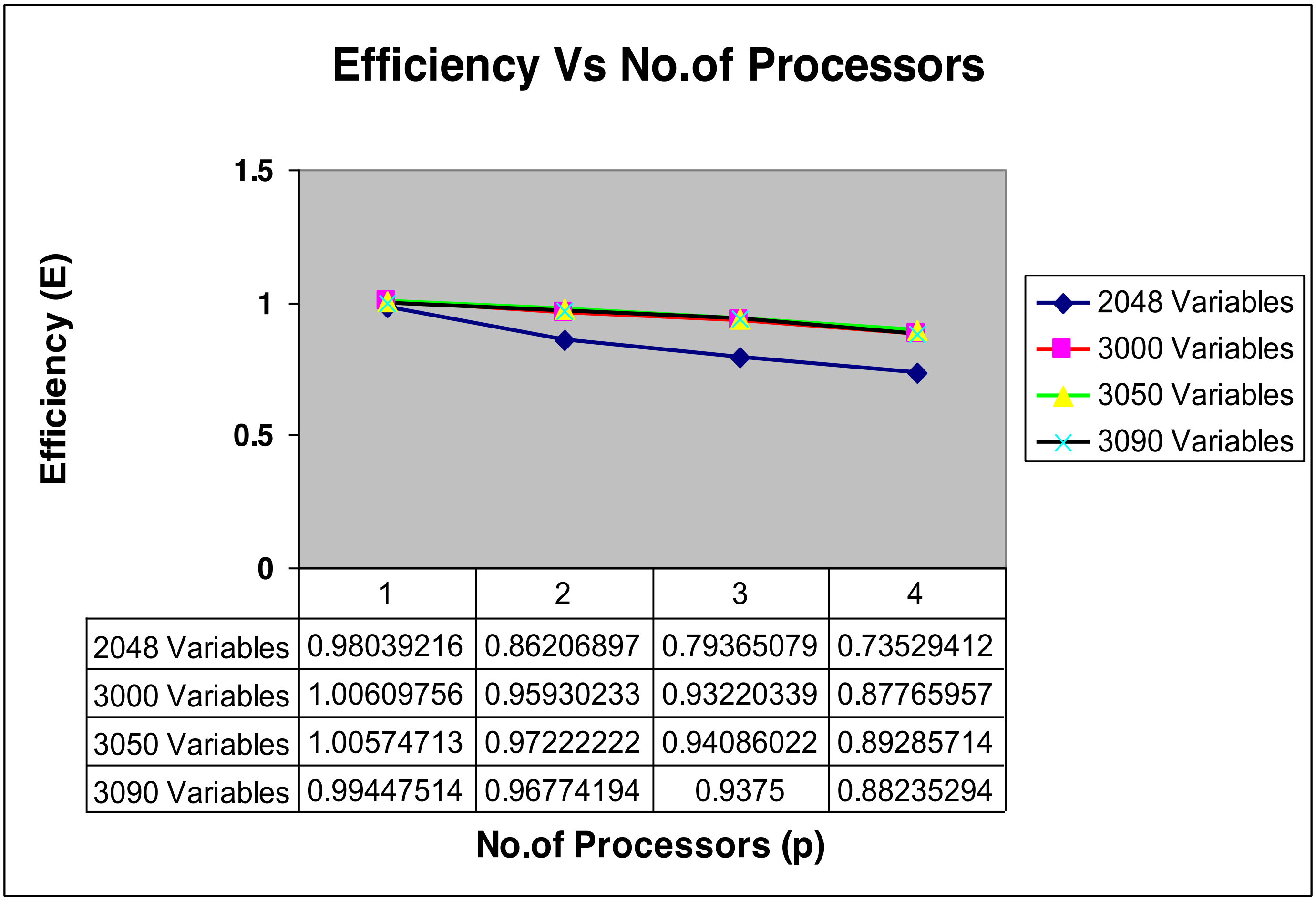


	1	2	3	4
2048 Variables	0.98039216	1.72413793	2.38095238	2.94117647
3000 Variables	1.00609756	1.91860465	2.79661017	3.5106383
3050 Variables	1.00574713	1.94444444	2.82258065	3.57142857
3090 Variables	0.99447514	1.93548387	2.8125	3.52941176

GRAPH 4

(E) **Efficiency:** The measure of how efficiently the parallel processing is done in terms of algorithm and/or architecture.

E = S / P



	1	2	3	4
2048 Variables	0.98039216	0.86206897	0.79365079	0.73529412
3000 Variables	1.00609756	0.95930233	0.93220339	0.87765957
3050 Variables	1.00574713	0.97222222	0.94086022	0.89285714
3090 Variables	0.99447514	0.96774194	0.9375	0.88235294

GRAPH 5

7.2 CONCLUSIONS

From the TABLE 1 and GRAPH 1 we can conclude that execution time for the sequential GEPP is directly proportionate to the number of variables (n).

From TABLE 2 and GRAPH 2 we can see that the time of execution of the program increases when the number of processors (p) and number of variables (n) are increased. That is not required for parallelization of any program. So we can conclude that column oriented GEPP is not an appropriate method for parallelization of GEPP when number of variables (n) is much greater than number of processor (p).

On the other hand from TABLE 3, GRAPH 3 we can see, when the number of processors (p) is increased the execution time for a particular number of variable (n) is uniformly decreasing, which is required for parallelization. From the GRAPH 4, we can see speedup becomes more ideal with the increasing number of variables (n). From GRAPH 5 we can see efficiency is also more ideal when the number of variables (n) is increasing. So we can conclude that the row oriented approach for parallelization of Gauss Elimination with Partial Pivoting is superior process.

7.3 FURTHER WORKS

- In future the row oriented parallel approach of GEPP can be tested with more number of variables and with more number of processors for a better measurement.
- Within the program the back substitution function can be done as parallel to get better result.
- Gauss elimination is used in so many numerical and science algorithms as a part where this parallelization can be used to get the faster result.

APPENDICES

APPENDIX A

SYSTEM DESCRIPTION

1. HARDWARE CONFIGURATION

Intel® Pentium ® 4 CPU 3.00 GHz
With HT technology (4 logical processors)
3.00 GHz, 512 MB of RAM.
Cache 1 MB L2.

2. OPERATING SYSTEM

Secure Shell

3. SOFTWARE CONFIGURATION

The language used in this program is C.

APPENDIX B

PTHREAD HEADER FILE DESCRIPTION

The Pthreads API is defined in the ANSI/IEEE POSIX 1003.1 - 1995 standard. Unlike MPI, this standard is not freely available on the Web - it must be purchased from IEEE. Pthreads are defined as a set of C language programming types and procedure calls, implemented with a **`pthread.h`** header/include file and a thread library - though this library may be part of another library, such as `libc`. The pthread.h header file is described below:

Syntax

#include <pthread.h>

Description

The **pthread.h** header defines the following symbols:

PTHREAD_CANCEL_ASYNCHRONOUS
PTHREAD_CANCEL_ENABLE
PTHREAD_CANCEL_DEFERRED
PTHREAD_CANCEL_DISABLE
PTHREAD_CANCELED
PTHREAD_COND_INITIALIZER
PTHREAD_CREATE_DETACHED
PTHREAD_CREATE_JOINABLE
PTHREAD_EXPLICIT_SCHED
PTHREAD_INHERIT_SCHED
PTHREAD_MUTEX_DEFAULT
PTHREAD_MUTEX_ERRORCHECK
PTHREAD_MUTEX_NORMAL
PTHREAD_MUTEX_INITIALIZER
PTHREAD_MUTEX_RECURSIVE
PTHREAD_ONCE_INIT
PTHREAD_PRIO_INHERIT
PTHREAD_PRIO_PROTECT
PTHREAD_PROCESS_SHARED
PTHREAD_PROCESS_PRIVATE

PTHREAD_RWLOCK_INITIALIZER
PTHREAD_SCOPE_PROCESS
PTHREAD_SCOPE_SYSTEM

The following are declared as functions and may also be declared as macros. Function prototypes must be provided for use with an ISO C compiler.

```
int pthread_attr_destroy (pthread_attr_t *);
int pthread_attr_getdetachstate (const pthread_attr_t *, int *);
int pthread_attr_getguardsize (const pthread_attr_t *, size_t *);
int pthread_attr_getinheritsched (const pthread_attr_t *, int *);
int pthread_attr_getschedparam (const pthread_attr_t *, struct
sched_param*);
int pthread_attr_getschedpolicy (const pthread_attr_t *, int *);
int pthread_attr_getscope (const pthread_attr_t *, int *);
int pthread_attr_getstackaddr (const pthread_attr_t *, void **);
int pthread_attr_getstacksize (const pthread_attr_t *, size_t *);
int pthread_attr_init (pthread_attr_t *);
int pthread_attr_setdetachstate (pthread_attr_t *, int);
int pthread_attr_setguardsize (pthread_attr_t *, size_t);
int pthread_attr_setinheritsched (pthread_attr_t *, int);
int pthread_attr_setschedparam (pthread_attr_t *, const struct
sched_param *);
int pthread_attr_setschedpolicy (pthread_attr_t *, int);
int pthread_attr_setscope (pthread_attr_t *, int);
int pthread_attr_setstackaddr (pthread_attr_t *, void *);
int pthread_attr_setstacksize (pthread_attr_t *, size_t);
int pthread_cancel(pthread_t);
void pthread_cleanup_push (void (*)(void*), void *);
void pthread_cleanup_pop (int);
int pthread_cond_broadcast (pthread_cond_t *);
int pthread_cond_destroy (pthread_cond_t *);
int pthread_cond_init (pthread_cond_t *, const pthread_condattr_t *);
int pthread_cond_signal (pthread_cond_t *);
int pthread_cond_timedwait (pthread_cond_t *, pthread_mutex_t *, const
struct timespec *);
int pthread_cond_wait (pthread_cond_t *);
int pthread_condattr_destroy (pthread_condattr_t *);
int pthread_condattr_getpshared (const pthread_condattr_t *, int *);
int pthread_condattr_init (pthread_condattr_t *);
int pthread_condattr_setpshared (pthread_condattr_t *, int);

int pthread_create (pthread_t *, const pthread_attr_t *, void
*(*)(void*), void *);
int pthread_detach (pthread_t);
int pthread_equal (pthread_t, pthread_t);
void pthread_exit (void *);
int pthread_getconcurrency (void);
int pthread_getschedparam (pthread_t, int *, struct sched_param *);
void *pthread_getspecific (pthread_key_t);
int pthread_join (pthread_t, void **);
int pthread_key_create (pthread_key_t *, void (*)(void*));
int pthread_key_delete (pthread_key_t);
int pthread_mutex_destroy (pthread_mutex_t *);
int pthread_mutex_getprioceiling (const pthread_mutex_t *, int *);
int pthread_mutex_init(pthread_mutex_t *,const pthread_mutexattr_t *);
int pthread_mutex_lock (pthread_mutex_t *);
int pthread_mutex_setprioceiling (pthread_mutex_t *, int, int *);
int pthread_mutex_trylock (pthread_mutex_t *);
int pthread_mutex_unlock (pthread_mutex_t *);
int pthread_mutexattr_destroy (pthread_mutexattr_t *);
int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t*,int *);
int pthread_mutexattr_getprotocol (const pthread_mutexattr_t *, int *);
int pthread_mutexattr_getpshared (const pthread_mutexattr_t *, int *);
int pthread_mutexattr_gettype (pthread_mutexattr_t *, int *);
int pthread_mutexattr_init (pthread_mutexattr_t *);
int pthread_mutexattr_setprioceiling (pthread_mutexattr_t *, int);
int pthread_mutexattr_setprotocol (pthread_mutexattr_t *, int);
int pthread_mutexattr_setpshared (pthread_mutexattr_t *, int);
int pthread_mutexattr_settype (pthread_mutexattr_t *, int);
int pthread_once (pthread_once_t *, void (*)(void));
int pthread_rwlock_destroy (pthread_rwlock_t *);
int pthread_rwlock_init(pthread_rwlock_t *,pthread_rwlockattr_t *);
int pthread_rwlock_rdlock(pthread_rwlock_t *);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *);
int pthread_rwlock_trywrlock(pthread_rwlock_t *);
int pthread_rwlock_unlock(pthread_rwlock_t *);
int pthread_rwlock_wrlock(pthread_rwlock_t *);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *);
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *, int *);
int pthread_rwlockattr_init(pthread_rwlockattr_t *);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *, int);
pthread_t pthread_self(void);
int pthread_setcancelstate(int, int *);
int pthread_setcanceltype(int, int *);
int pthread_setconcurrency(int);
int pthread_setschedparam(pthread_t, int *,
const struct sched_param *);
int pthread_setspecific(pthread_key_t, const void *);
void pthread_testcancel(void);
```

Inclusion of the **pthread.h** header will make visible symbols defined in the headers **sched.h** and **time.h**.

APPENDIX C

PTHREAD CALLS & COMMANDS USED

The Pthreads API contains over 60 subroutines. This chapter will focus on a subset of these - specifically, those which have been used in the PGEPP program. For portability, the `pthread.h` header file should be included in each source file using the Pthreads library.

All identifiers in the threads library begin with **pthread_** as follows:

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects.
pthread_mutex_	Mutexes.
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables.
pthread_condattr_	Condition attributes objects.
pthread_key_	Thread_specific data keys.

DATA TYPES:

pthread_t	
pthread_attr_t	
pthread_mutex_t	*mutex*= PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t	*mutexattr*; // must initialize dynamically
pthread_condattr_t	*condattr*; // must initialize dynamically
pthread_cond_t	cond=PTHREAD_COND_INITIALIZER;

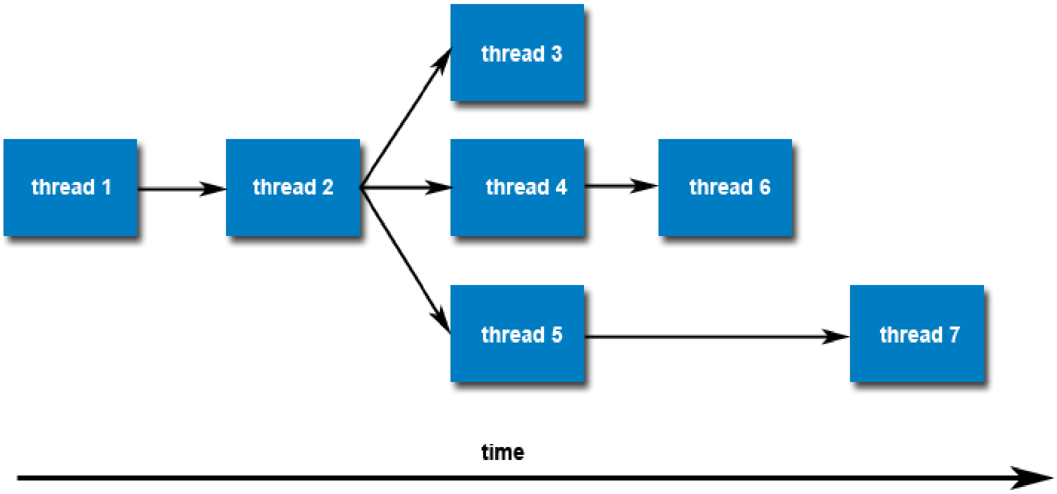
FUNCTION PROTOTYPES:

1. **CREATING THREADS** - Creates a thread and makes it available for execution.

int pthread_create *(pthread_t *thread, const pthread_attr_t *attr, void*(*routine)(void*), void* arg);*

Initially, the `main()` program comprises a single, default thread. All other threads must be explicitly created by the programmer. `pthread_create` creates a new thread and makes it executable. This routine can be called any number of times from anywhere within the code. The **`pthread_create`** arguments are:

- **`thread`** : An opaque, unique identifier for the new thread returned by the subroutine.
- **`attr`** : An opaque attribute object that may be used to set thread attributes. One can specify a thread attributes object, or NULL for the default values.
- **`start_routine`**: the C routine that the thread will execute once it is created.
- **`arg`** : A single argument that may be passed to *start_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.



2. TERMINATING THREADS – Terminates a thread when it is no longer needed

int pthread_exit *(void* status);*

There are several ways in which a Pthread may be terminated:

- The thread returns from its starting routine (the main routine for the initial thread).
- The thread makes a call to the `pthread_exit` subroutine (covered below).
- The entire process is terminated due to a call to either the exec or exit subroutines.

- `pthread_exit` is used to explicitly exit a thread. Typically, the `pthread_exit()` routine is called after a thread has completed its work and is no longer required to exist.
- If `main()` finishes before the threads it has created, and exits with `pthread_exit()`, the other threads will continue to execute. Otherwise, they will be automatically terminated when `main()` finishes.
- The programmer may optionally specify a termination *status*, which is stored as a void pointer for any thread that may join the calling thread.

3. PASSING ARGUMENTS TO THREADS VIA STRUCTURE –

Its always easy to use structure rather than array because different data types can be declared. In pthread programming argument passing is possible via structure. Following is data structure that is used in row oriented GEPP to pass the argument from main to child thread named new_coeff-thread ( ).

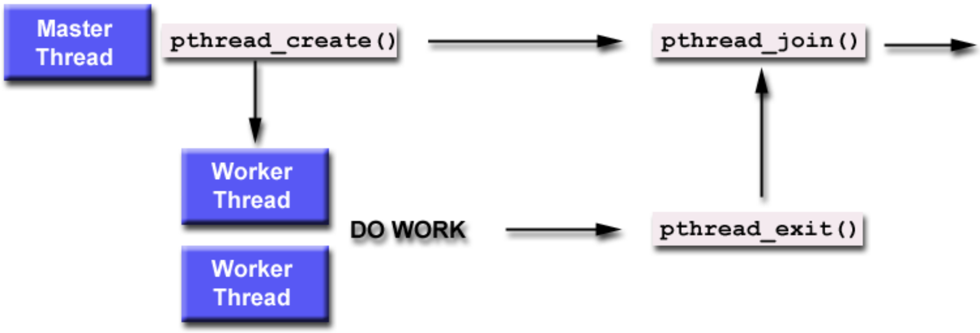
```
struct thread_data{
int  thread_id;
int  col;
int start_row;
int end_row;   };

struct thread_data thread_data_arr[NUM_THREADS];
```

```
void *new_coeff_thread(void *threadarg)
{
     struct thread_data *loc_data;
     ……………
        loc_data = (struct thread_data *) threadarg;
        tid = loc_data->thread_id;
        c = loc_data->col;
        s_row= loc_data->start_row;
        e_row= loc_data->end_row;
      …………………
}

int main(){
……………
  thread_data_arr[k].thread_id = k;
  thread_data_arr[k].col = j;
  thread_data_arr[k].start_row = i;

  if(k!=(x-1))
   {
     thread_data_arr[k].end_row = (i+(p-1));
     ……….
}
```

4. <u>JOINING OF THREADS</u> - Joining is one way to accomplish synchronization between threads.

int pthread_join *(pthread_t thread, void ** status);*

The `pthread_join()` subroutine blocks the calling thread until the specified `threadid` thread terminates. The programmer is able to obtain the target thread's termination return `status` if it was specified in the target thread's call to `pthread_exit()`. A joining thread can match one `pthread_join()` call. It is a logical error to attempt multiple joins on the same thread. Two other synchronization methods are mutexes and condition variables.



<u>“WAITING” FOR THREAD COMPLETION USING JOIN()</u>

APPENDIX-D

COMPILING & EXECUTING PROGRAMS

1. COMPILE PROCEDURE

Before compiling the sgepp.c/ pgepp_pth_col.c / pgepp_pth_row.c we need to put the value for n which indicates the number of variables in the file “no_of_var”. Then have to run input_create.c so that the input_mat file is created.

Sequential and the two parallel programs are compiled in the following way:

```
gcc  sgepp.c –o sgepp
```

```
gcc  pgepp_pth_col.c –o pgepp_th_col -lpthread
```

```
gcc  pgepp_pth_row.c –o pgepp_th_row -lpthread
```

2. RUN PROCEDURE

Then the programs are executed. There are no command line arguments. We just run the programs with the program name.

```
./sgepp
```

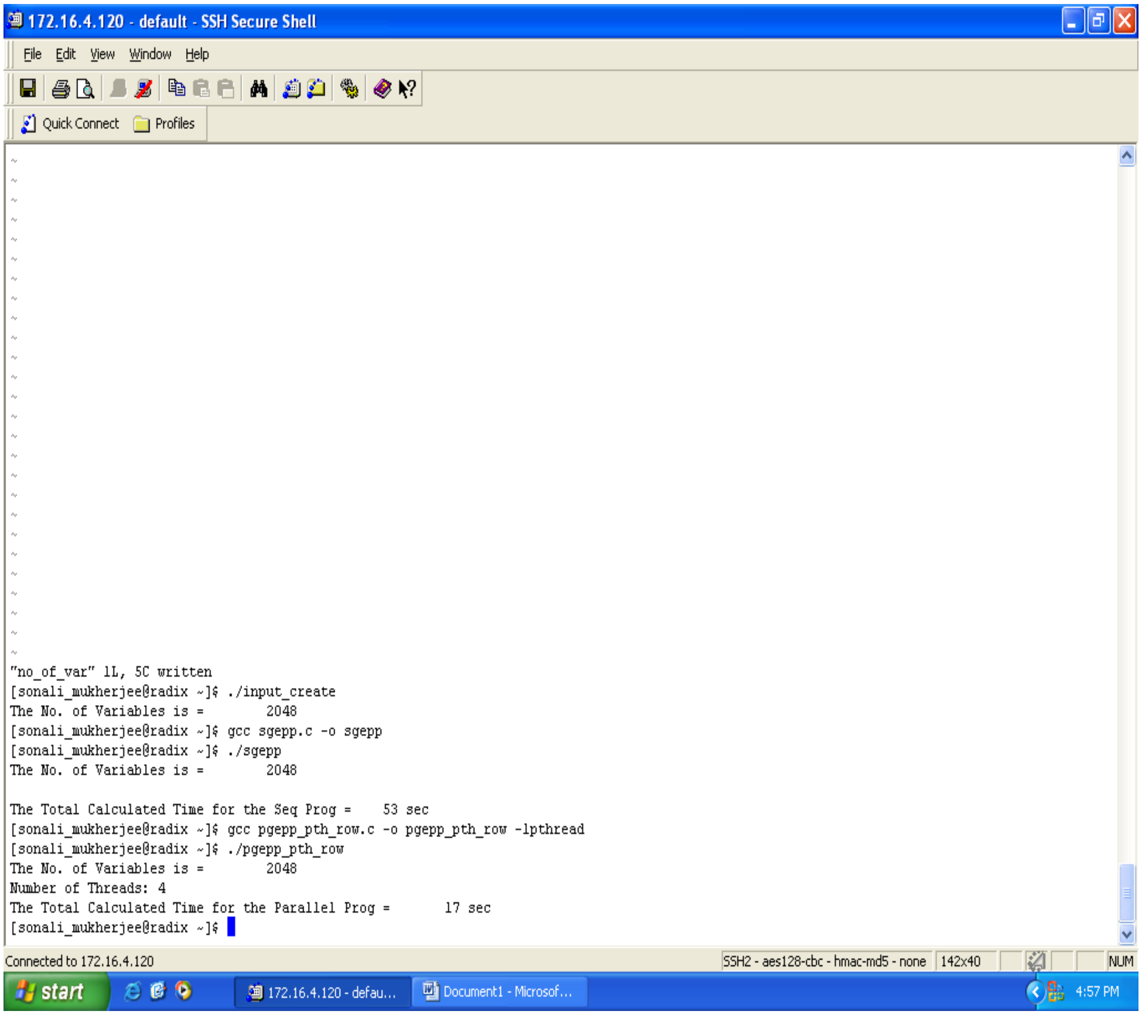
```
./pgepp_pth_col
```

```
./pgepp_pth_row
```

The solution of the unknown variables of the linear system will be stored in the file named as ans_file.

3. SCREEN LAYOUT

Following is the screen layout for compiling and executing sgepp.c and pgepp_pth_row.c programs for 2048 number of variables.



```
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"no_of_var" 1L, 5C written
[sonali_mukherjee@radix ~]$ ./input_create
The No. of Variables is =        2048
[sonali_mukherjee@radix ~]$ gcc sgepp.c -o sgepp
[sonali_mukherjee@radix ~]$ ./sgepp
The No. of Variables is =        2048

The Total Calculated Time for the Seq Prog =    53 sec
[sonali_mukherjee@radix ~]$ gcc pgepp_pth_row.c -o pgepp_pth_row -lpthread
[sonali_mukherjee@radix ~]$ ./pgepp_pth_row
The No. of Variables is =        2048
Number of Threads: 4
The Total Calculated Time for the Parallel Prog =       17 sec
[sonali_mukherjee@radix ~]$ 
```

REFERENCE

Reference

[1] "*Introduction to Numerical Analysis*" by Amtritava Gupta, Subhas Chandra Bose. Academic Publishers.

[2] "*Parallel Computing Theory & Practice*" by Michale J. Quinn. Tata McGraw-Hill Publishing Company Ltd.

[3] Parallel Programming in C with MPI and OpenMP by Michale J. Quinn. Tata McGraw-Hill Publishing Company Ltd.

[4] Materials on Numerical Analysis http://www.scholarpedia.org/article/Numerical_analysis

[5] Materials on Linear System at http://en.wikipedia.org/wiki/System_of_linear_equations

[6] *Parallel Linear Algebra Methods* Published in 2006 by John von Neumann Institute for Computing. http://www.fz-juelich.de/nic-series/volume31

[7] *Applications of Parallel Computers Solving Linear Systems arising from PDEs*–I. James Demmel. www.cs.berkeley.edu/~demmel/CS267_2002_Poisson_1

[8] *Materials on Parallel Computing* http://www.top500.org http://www.llnl.gov/computing/tutorials/parallel_comp

[9] Design of Parallel algorithms – linear equations by Jari Prras. http://www.it.lut.fi/kurssit/02-

[10] Materials on Gauss Elimination Method - http://en.wikipedia.org/wiki/Gaussian_elimination

[12] Materials on Gauss Elimination with Partial Pivoting - http://www.nasc.snu.ac.kr/sheen/nla/html/node8.html

[13] Gauss Elimination with Partial Pivoting – Chapter 04.06 http://numericalmethods.eng.usf.edu/mtl/com/04sle/mtl_com_sle_txt_gaussian.doc

[14] CS 267 Dense Linear Algebra: Parallel Gaussian Elimination by James Demmel www.cs.berkeley.edu/~demmel/cs267_Spr08

[15] Pthread Materials on - https://computing.llnl.gov/tutorials/pthreads/

[16] Code help for Parallel Gauss Elimination - http://code.google.com/p/parallel-gaussian-elimination/updates/list