

**OPERATING SYSTEM**

**SLOT- E2**

**PROF. SHAIK NASEERA**

**PROJECT TITLE**

**PARALELLIZATION OF POISON EQUATION IN 2D SPACE USING JACOBI ITERATION**

**Project Report**

BY

DHRUV SIROHI – 15BCE0146

SHIVAM ANAND – 15BCE0169

KANAV SETHI – 15BCE0311

TANISHKA SHOREY – 15BCE0739

**Abstract**

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same or lesser time. Parallelization can be achieved by dividing the program into multiple threads, these threads are individually executed during executing of the program in separate parts of the processor.

The project works at applying these principles of parallelization to the Poisson equation and attempt to compare the results of a sequential execution and a parallel execution. The parallelization of the Poison Equation has been achieved us the Jacobi’s Iteration Method.

**Key Words**

Poisson Equation; Jacobi’s Iteration; Parallelization; Threads; openMP.

**Introduction**

The Poisson equation

−∇2u = f (1.1)

is the simplest and the most famous elliptic partial differential equation. The source (or load) function f is given on some two- or three-dimensional domain denoted by Ω ⊂ R2 or R3. A solution u satisfying (1.1) will also satisfy boundary conditions on the boundary ∂Ω of Ω; for example

αu + β ∂u ∂n = g on ∂Ω, (1.2)

where ∂u/∂n denotes the directional derivative in the direction normal to the boundary ∂Ω (conventionally pointing outwards) and α and β are constant, although variable coefficients are also possible.

The combination of (1.1) and (1.2) together is referred to as a boundary value problem. If the constant β in (1.2) is zero, then the boundary condition is of Dirichlet type, and the boundary value problem is referred to as the Dirichlet problem for the Poisson equation.

The version of Poisson's equation being solved is for a 2D space or Rectangular region,



over the rectangle {**0 <= X <= 1**}, {**0 <= Y <= 1**}, with exact solution

**U(x,y) = sin ( pi \* x \* y )**

so that

**F(x,y) = pi^2 \* ( x^2 + y^2 ) \* sin ( pi \* x \* y )**

and with Dirichlet boundary conditions along the lines x = 0, x = 1, y = 0 and y = 1. (The boundary conditions will actually be zero in this case, but we write up the problem as though we didn't know that, which makes it easy to change the problem later.)

We compute an approximate solution by discretizing the geometry, assuming that DX = DY, and approximating the Poisson operator by

**( U(i-1,j) + U(i+1,j) + U(i,j-1) + U(i,j+1) - 4\*U(i,j) ) / dx /dy**

Along with the boundary conditions at the boundary nodes, we have a linear system for U. We can apply the Jacobi iteration to estimate the solution to the linear system.

**System Design**

**Programming language**: C

**Library**: omp.h

We will be implementing parallel computing with the help of threads.

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.

Threads have the ability to speed up the program provided they are used carefully. Threads are popular way to improve application through parallelism. Threads operate faster than processes due to following reasons:

* 1. Thread creation is much faster.
  2. Context switching between threads is much faster.
  3. Threads can be terminated easily
  4. Communication between threads is faster.

Another important step is task decomposition. Task decomposition is most likely the backbone of parallel computing. Factors like dependency are taken into consideration before dividing the tasks and distributing them among threads.

Our aim is to decompose the equations involved in Jacobi iterations and form task that can be executed parallelly. If done right, we expect to see faster execution of our program as a whole.

**Literature Review**

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time.

The computing industry changed course in 2005 when Intel followed the lead of IBM’s Power 4 and Sun Microsystems’ Niagara processor in announcing that its high-performance microprocessors would henceforth rely on multiple processors or cores. The new industry buzzword “multicore” captures the plan of doubling the number of standard cores per die with every semiconductor process generation starting with a single processor. This was stated by “Moores” law. Moreover, actual processor performance has increased faster than Moore’s law would predict, because processor designers have been able to harness the increasing numbers of transistors available on modern chips to extract more parallelism from software. The microprocessor industry is leading the way to multicore architectures; however, the full benefit of these architectures will not be harnessed until the software industry fully embraces parallel programming. [1] [2]

Linear systems Ax = b arose from industrial applications are usually large and sparse. It is very common that those systems involve millions of unknowns. With such a magnitude, forming parallel threads to solve the equation only makes sense. The most popular methods nowadays used in industry are iterative methods. [3] [6]

The general framework of an iterative process is as simple as this: first, an initial assumption-solution is generated intuitively for the vector-solution x(0). Then, using this assumption the algorithm provides us with a possible solution x(1). Now the role of the solution x(1) becomes the input for the next possible solution. This process goes repeatedly, providing an array of vector-solutions x(0), x(1), x(2)…, until we get into a satisfactory solution. Jacobi iteration follows this framework. [4]

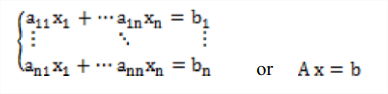
Jacobi Iterative method is one of the most efficient techniques for solving above stated equations. This method obtained more popularity in the field of scientific computing due to its mathematical simplicity. This solver is implemented in several platforms such as GPU, FPGA etc. [5]

Poisson Equation has made it’s way into many engineering applications. One direct way of solving Poisson equation is by Fourier Analysis. [7] But such a summation process is a very slow one. The Poisson equation discretized onto a finite differenced spatial grid can be in essence transformed to a system of linear equations, which can be described by a sparse matrix. Numerical mathematics provides various methods with which the solution can be found. These methods can be divided into two groups: the direct and the iterative methods. Iterative solvers are particularly useful in the PIC method, because at each time step the solution from the previous time step can be used as the initial estimate of the new solution. The importance of this benefit, however, strongly depends on fluctuations of the density between successive time steps. [8] [9]

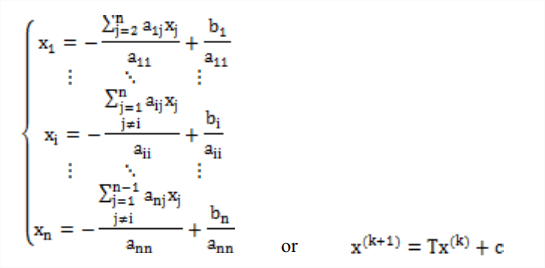
A 2D Poisson Equation can be discretized into a form of a grid. Boundary conditions are also needed. An (n+1)-by-(n+1) grid on Omega = the unit square, where h=1/(n+1) is the grid spacing is used. Any point U(i,j) on the grid is said to be the approximate solution at x=i\*h and y=j\*h. Jacobi, SOR (Successive OverRelaxation) and CG (Conjugate Gradients) can be thought of as performing most nearest neighbor operations on the grid.  Jacobi is easy to implement in parallel because each grid point can be treated independently. [10]

**Implementation Plan**

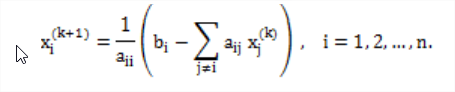
* Jacobi puts borders between iterations; values of the vector-solution x are calculated only from the vector-solution of the previous iteration. Hence, it can be parallelized easily.
* When solving linear equations of the form:



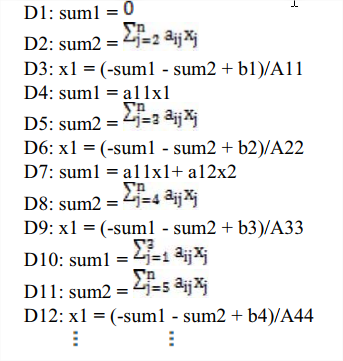
In order to describe the method procedure firstly we set the system in more proper form like the following:



Hence, the formula in terms of its elements would look like the following:

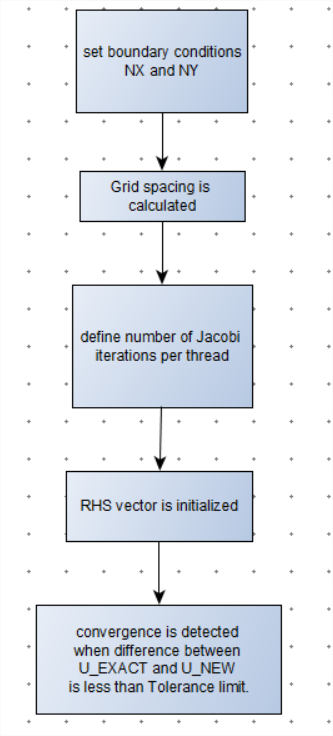


* Second step is to decompose data based on the above equation:



Thus, we can see that, there are independent sub problems like: 1, 2, 4, 5, 7, 8, 10, 11, etc. and those dependent: 3 dependents from 1 and 2, 6 from 4 and 5 etc. This way the process of parallelization looks more feasible and every process can do many of these tasks (threads) depending of the mapping topology.

**Implementation Flow Chart**



**Implementation**

**Parallel Coding in openMP**

# include <stdlib.h>

# include <stdio.h>

# include <math.h>

# include <time.h>

# include <omp.h>

# define NX 232

# define NY 232

int main ( int argc, char \*argv[] );

double r8mat\_rms ( int nx, int ny, double a[NX][NY] );

void rhs ( int nx, int ny, double f[NX][NY] );

void sweep ( int nx, int ny, double dx, double dy, double f[NX][NY],

int itold, int itnew, double u[NX][NY], double unew[NX][NY] );

void timestamp ( void );

double u\_exact ( double x, double y );

double uxxyy\_exact ( double x, double y );

int main ( int argc, char \*argv[] )

/\*

POISSON\_OPENMP is a program for solving the Poisson problem.

This program uses OpenMP for parallel execution.

The Poisson equation

- ( d/dx d/dx + d/dy d/dy ) U(x,y) = F(x,y)

- DEL^2 U(X,Y) = F(X,Y)

is solved on the unit square [0,1] x [0,1] using a grid of NX by

NY evenly spaced points. The first and last points in each direction

are boundary points.

The boundary conditions and F are set so that the exact solution is

U(x,y) = sin ( pi \* x \* y )

so that

- DEL^2 U(x,y) = pi^2 \* ( x^2 + y^2 ) \* sin ( pi \* x \* y )

The Jacobi iteration is repeatedly applied until convergence is detected.

For convenience in writing the discretized equations, we assume that NX = NY.

\*/

{

int converged;

double diff;

double dx;

double dy;

double error;

double f[NX][NY];

int i;

int id;

int itnew;

int itold;

int j;

int jt;

int jt\_max = 20;

int nx = NX;

int ny = NY;

double tolerance = 0.000001;

double u[NX][NY];

double u\_norm;

double udiff[NX][NY];

double uexact[NX][NY];

double unew[NX][NY];

double unew\_norm;

double wtime;

double x;

double y;

dx = 1.0 / ( double ) ( nx - 1 );

dy = 1.0 / ( double ) ( ny - 1 );

timestamp ( );

printf ( "\n" );

printf ( "POISSON\_OPENMP:\n" );

printf ( " C version\n" );

printf ( " A program for solving the Poisson equation.\n" );

printf ( "\n" );

printf ( " Use OpenMP for parallel execution.\n" );

printf ( " The number of processors is %d\n", omp\_get\_num\_procs ( ) );

# pragma omp parallel

{

id = omp\_get\_thread\_num ( );

if ( id == 0 )

{

printf ( " The maximum number of threads is %d\n", omp\_get\_num\_threads ( ) );

}

}

printf ( "\n" );

printf ( " -DEL^2 U = F(X,Y)\n" );

printf ( "\n" );

printf ( " on the rectangle 0 <= X <= 1, 0 <= Y <= 1.\n" );

printf ( "\n" );

printf ( " F(X,Y) = pi^2 \* ( x^2 + y^2 ) \* sin ( pi \* x \* y )\n" );

printf ( "\n" );

printf ( " The number of interior X grid points is %d\n", nx );

printf ( " The number of interior Y grid points is %d\n", ny );

printf ( " The X grid spacing is %f\n", dx );

printf ( " The Y grid spacing is %f\n", dy );

/\*

Set the right hand side array F.

\*/

rhs ( nx, ny, f );

/\*

Set the initial solution estimate UNEW.

We are "allowed" to pick up the boundary conditions exactly.

\*/

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

if ( i == 0 || i == nx - 1 || j == 0 || j == ny - 1 )

{

unew[i][j] = f[i][j];

}

else

{

unew[i][j] = 0.0;

}

}

}

unew\_norm = r8mat\_rms ( nx, ny, unew );

/\*

Set up the exact solution UEXACT.

\*/

for ( j = 0; j < ny; j++ )

{

y = ( double ) ( j ) / ( double ) ( ny - 1 );

for ( i = 0; i < nx; i++ )

{

x = ( double ) ( i ) / ( double ) ( nx - 1 );

uexact[i][j] = u\_exact ( x, y );

}

}

u\_norm = r8mat\_rms ( nx, ny, uexact );

printf ( " RMS of exact solution = %g\n", u\_norm );

/\*

Do the iteration.

\*/

converged = 0;

printf ( "\n" );

printf ( " Step ||Unew|| ||Unew-U|| ||Unew-Exact||\n" );

printf ( "\n" );

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

udiff[i][j] = unew[i][j] - uexact[i][j];

}

}

error = r8mat\_rms ( nx, ny, udiff );

printf ( " %4d %14g %14g\n", 0, unew\_norm, error );

wtime = omp\_get\_wtime ( );

itnew = 0;

for ( ; ; )

{

itold = itnew;

itnew = itold + 500;

/\*

SWEEP carries out 500 Jacobi steps in parallel before we come

back to check for convergence.

\*/

sweep ( nx, ny, dx, dy, f, itold, itnew, u, unew );

/\*

Check for convergence.

\*/

u\_norm = unew\_norm;

unew\_norm = r8mat\_rms ( nx, ny, unew );

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

udiff[i][j] = unew[i][j] - u[i][j];

}

}

diff = r8mat\_rms ( nx, ny, udiff );

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

udiff[i][j] = unew[i][j] - uexact[i][j];

}

}

error = r8mat\_rms ( nx, ny, udiff );

printf ( " %4d %14g %14g %14g\n", itnew, unew\_norm, diff, error );

if ( diff <= tolerance )

{

converged = 1;

break;

}

}

if ( converged )

{

printf ( " The iteration has converged.\n" );

printf ( "\n" );

printf ( " Step ||Unew|| ||Unew-U|| ||Unew-Exact||\n" );

printf ( "\n" );

printf ( " %4d %14g %14g %14g\n", itnew, unew\_norm, diff, error );

}

else

{

printf ( " The iteration has NOT converged.\n" );

}

wtime = omp\_get\_wtime ( ) - wtime;

printf ( "\n" );

printf ( " Elapsed seconds = %g\n", wtime );

/\*

Terminate.

\*/

printf ( "\n" );

printf ( "POISSON\_OPENMP:\n" );

printf ( " Normal end of execution.\n" );

printf ( "\n" );

timestamp ( );

return 0;

}

double r8mat\_rms ( int nx, int ny, double a[NX][NY] )

/\*

Input, int NX, NY, the number of rows and columns in A.

Input, double A[NX][NY], the vector.

Output, double R8MAT\_RMS, the root mean square of the entries of A.

\*/

{

int i;

int j;

double v;

v = 0.0;

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

v = v + a[i][j] \* a[i][j];

}

}

v = sqrt ( v / ( double ) ( nx \* ny ) );

return v;

}

void rhs ( int nx, int ny, double f[NX][NY] )

/\*

Purpose:

RHS initializes the right hand side "vector".

Discussion:

It is convenient for us to set up RHS as a 2D array. However, each

entry of RHS is really the right hand side of a linear system of the

form

A \* U = F

In cases where U(I,J) is a boundary value, then the equation is simply

U(I,J) = F(i,j)

and F(I,J) holds the boundary data.

Otherwise, the equation has the form

(1/DX^2) \* ( U(I+1,J)+U(I-1,J)+U(I,J-1)+U(I,J+1)-4\*U(I,J) ) = F(I,J)

where DX is the spacing and F(I,J) is the value at X(I), Y(J) of

pi^2 \* ( x^2 + y^2 ) \* sin ( pi \* x \* y )

Parameters:

Input, int NX, NY, the X and Y grid dimensions.

Output, double F[NX][NY], the initialized right hand side data.

\*/

{

double fnorm;

int i;

int j;

double x;

double y;

for ( j = 0; j < ny; j++ )

{

y = ( double ) ( j ) / ( double ) ( ny - 1 );

for ( i = 0; i < nx; i++ )

{

x = ( double ) ( i ) / ( double ) ( nx - 1 );

if ( i == 0 || i == nx - 1 || j == 0 || j == ny - 1 )

{

f[i][j] = u\_exact ( x, y );

}

else

{

f[i][j] = - uxxyy\_exact ( x, y );

}

}

}

fnorm = r8mat\_rms ( nx, ny, f );

printf ( " RMS of F = %g\n", fnorm );

return;

}

void sweep ( int nx, int ny, double dx, double dy, double f[NX][NY],

int itold, int itnew, double u[NX][NY], double unew[NX][NY] )

/\*

Purpose:

SWEEP carries out one step of the Jacobi iteration.

Discussion:

Assuming DX = DY, we can approximate

- ( d/dx d/dx + d/dy d/dy ) U(X,Y)

by

( U(i-1,j) + U(i+1,j) + U(i,j-1) + U(i,j+1) - 4\*U(i,j) ) / dx / dy

The discretization employed below will not be correct in the general

case where DX and DY are not equal. It's only a little more complicated

to allow DX and DY to be different, but we're not going to worry about

that right now.

Parameters:

Input, int NX, NY, the X and Y grid dimensions.

Input, double DX, DY, the spacing between grid points.

Input, double F[NX][NY], the right hand side data.

Input, int ITOLD, the iteration index on input.

Input, int ITNEW, the desired iteration index

on output.

Input, double U[NX][NY], the solution estimate on

iteration ITNEW-1.

Input/output, double UNEW[NX][NY], on input, the solution

estimate on iteration ITOLD. On output, the solution estimate on

iteration ITNEW.

\*/

{

int i;

int it;

int j;

# pragma omp parallel shared ( dx, dy, f, itnew, itold, nx, ny, u, unew ) private ( i, it, j )

for ( it = itold + 1; it <= itnew; it++ )

{

# pragma omp for

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

u[i][j] = unew[i][j];

}

}

# pragma omp for

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

if ( i == 0 || j == 0 || i == nx - 1 || j == ny - 1 )

{

unew[i][j] = f[i][j];

}

else

{

unew[i][j] = 0.25 \* (

u[i-1][j] + u[i][j+1] + u[i][j-1] + u[i+1][j] + f[i][j] \* dx \* dy );

}

}

}

}

return;

}

void timestamp ( void )

/\*

Purpose:

TIMESTAMP prints the current YMDHMS date as a time stamp.

\*/

{

# define TIME\_SIZE 40

static char time\_buffer[TIME\_SIZE];

const struct tm \*tm;

time\_t now;

now = time ( NULL );

tm = localtime ( &now );

strftime ( time\_buffer, TIME\_SIZE, "%d %B %Y %I:%M:%S %p", tm );

printf ( "%s\n", time\_buffer );

return;

# undef TIME\_SIZE

}

double u\_exact ( double x, double y )

/\*

Purpose:

U\_EXACT evaluates the exact solution.

Parameters:

Input, double X, Y, the coordinates of a point.

Output, double U\_EXACT, the value of the exact solution

at (X,Y).

\*/

{

double pi = 3.141592653589793;

double value;

value = sin ( pi \* x \* y );

return value;

}

double uxxyy\_exact ( double x, double y )

/\*

Purpose:

UXXYY\_EXACT evaluates ( d/dx d/dx + d/dy d/dy ) of the exact solution.

Parameters:

Input, double X, Y, the coordinates of a point.

Output, double UXXYY\_EXACT, the value of

( d/dx d/dx + d/dy d/dy ) of the exact solution at (X,Y).

\*/

{

double pi = 3.141592653589793;

double value;

value = - pi \* pi \* ( x \* x + y \* y ) \* sin ( pi \* x \* y );

return value;

}

# undef NX

# undef NY

**Serial Coding**

# include <stdlib.h>

# include <stdio.h>

# include <math.h>

# include <time.h>

# define NX 232

# define NY 232

int main ( int argc, char \*argv[] );

double r8mat\_rms ( int nx, int ny, double a[NX][NY] );

void rhs ( int nx, int ny, double f[NX][NY] );

void sweep ( int nx, int ny, double dx, double dy, double f[NX][NY],

double u[NX][NY], double unew[NX][NY] );

void timestamp ( void );

double u\_exact ( double x, double y );

double uxxyy\_exact ( double x, double y );

int main ( int argc, char \*argv[] )

/\*

Purpose:

POISSON\_SERIAL is a program for solving the Poisson problem.

This program runs serially. Its output is used as a benchmark for

comparison with similar programs run in a parallel environment.

The Poisson equation

- DEL^2 U(X,Y) = F(X,Y)

is solved on the unit square [0,1] x [0,1] using a grid of NX by

NX evenly spaced points. The first and last points in each direction

are boundary points.

The boundary conditions and F are set so that the exact solution is

U(x,y) = sin ( pi \* x \* y )

so that

- DEL^2 U(x,y) = pi^2 \* ( x^2 + y^2 ) \* sin ( pi \* x \* y )

The Jacobi iteration is repeatedly applied until convergence is detected.

For convenience in writing the discretized equations, we assume that NX = NY.

\*/

{

int converged;

double diff;

double dx;

double dy;

double error;

double f[NX][NY];

int i;

int it;

int it\_max = 100000;

int j;

int nx = NX;

int ny = NY;

double tolerance = 0.000001;

double u[NX][NY];

double u\_norm;

double udiff[NX][NY];

double uexact[NX][NY];

double unew[NX][NY];

double unew\_norm;

double x;

double y;

dx = 1.0 / ( double ) ( nx - 1 );

dy = 1.0 / ( double ) ( ny - 1 );

timestamp ( );

printf ( "\n" );

printf ( "POISSON\_SERIAL:\n" );

printf ( " C version\n" );

printf ( " A program for solving the Poisson equation.\n" );

printf ( "\n" );

printf ( " -DEL^2 U = F(X,Y)\n" );

printf ( "\n" );

printf ( " on the rectangle 0 <= X <= 1, 0 <= Y <= 1.\n" );

printf ( "\n" );

printf ( " F(X,Y) = pi^2 \* ( x^2 + y^2 ) \* sin ( pi \* x \* y )\n" );

printf ( "\n" );

printf ( " The number of interior X grid points is %d\n", nx );

printf ( " The number of interior Y grid points is %d\n", ny );

printf ( " The X grid spacing is %f\n", dx );

printf ( " The Y grid spacing is %f\n", dy );

rhs ( nx, ny, f );

clock\_t timewala = clock();

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

if ( i == 0 || i == nx - 1 || j == 0 || j == ny - 1 )

{

unew[i][j] = f[i][j];

}

else

{

unew[i][j] = 0.0;

}

}

}

unew\_norm = r8mat\_rms ( nx, ny, unew );

for ( j = 0; j < ny; j++ )

{

y = ( double ) ( j ) / ( double ) ( ny - 1 );

for ( i = 0; i < nx; i++ )

{

x = ( double ) ( i ) / ( double ) ( nx - 1 );

uexact[i][j] = u\_exact ( x, y );

}

}

u\_norm = r8mat\_rms ( nx, ny, uexact );

printf ( " RMS of exact solution = %g\n", u\_norm );

converged = 0;

printf ( "\n" );

printf ( " Step ||Unew|| ||Unew-U|| ||Unew-Exact||\n" );

printf ( "\n" );

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

udiff[i][j] = unew[i][j] - uexact[i][j];

}

}

error = r8mat\_rms ( nx, ny, udiff );

printf ( " %4d %14g %14g\n", 0, unew\_norm, error );

for ( it = 1; it <= it\_max; it++ )

{

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

u[i][j] = unew[i][j];

}

}

/\*

UNEW is derived from U by one Jacobi step.

\*/

sweep ( nx, ny, dx, dy, f, u, unew );

/\*

Check for convergence.

\*/

u\_norm = unew\_norm;

unew\_norm = r8mat\_rms ( nx, ny, unew );

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

udiff[i][j] = unew[i][j] - u[i][j];

}

}

diff = r8mat\_rms ( nx, ny, udiff );

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

udiff[i][j] = unew[i][j] - uexact[i][j];

}

}

error = r8mat\_rms ( nx, ny, udiff );

printf ( " %4d %14g %14g %14g\n", it, unew\_norm, diff, error );

if ( diff <= tolerance )

{

converged = 1;

break;

}

}

if ( converged )

{

printf ( " The iteration has converged.\n" );

printf ( "\n" );

printf ( " Step ||Unew|| ||Unew-U|| ||Unew-Exact||\n" );

printf ( "\n" );

printf ( " %4d %14g %14g %14g\n", it, unew\_norm, diff, error );

}

else

{

printf ( " The iteration has NOT converged.\n" );

}

printf ( "\n" );

printf ( " Elapsed seconds = %.2f\n", (double)(clock() - timewala)/CLOCKS\_PER\_SEC);

printf ( "\n" );

printf ( "POISSON\_SERIAL:\n" );

printf ( " Normal end of execution.\n" );

printf ( "\n" );

timestamp ( );

return 0;

}

double r8mat\_rms ( int nx, int ny, double a[NX][NY] )

/\*

Purpose:

R8MAT\_RMS returns the RMS norm of a vector stored as a matrix.

Parameters:

Input, int NX, NY, the number of rows and columns in A.

Input, double A[NX][NY], the vector.

Output, double R8MAT\_RMS, the root mean square of the entries of A.

\*/

{

int i;

int j;

double v;

v = 0.0;

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

v = v + a[i][j] \* a[i][j];

}

}

v = sqrt ( v / ( double ) ( nx \* ny ) );

return v;

}

void rhs ( int nx, int ny, double f[NX][NY] )

/\*

Purpose:

RHS initializes the right hand side "vector".

Discussion:

It is convenient for us to set up RHS as a 2D array. However, each

entry of RHS is really the right hand side of a linear system of the

form

A \* U = F

In cases where U(I,J) is a boundary value, then the equation is simply

U(I,J) = F(i,j)

and F(I,J) holds the boundary data.

Otherwise, the equation has the form

(1/DX^2) \* ( U(I+1,J)+U(I-1,J)+U(I,J-1)+U(I,J+1)-4\*U(I,J) ) = F(I,J)

where DX is the spacing and F(I,J) is the value at X(I), Y(J) of

pi^2 \* ( x^2 + y^2 ) \* sin ( pi \* x \* y )

Parameters:

Input, int NX, NY, the X and Y grid dimensions.

Output, double F[NX][NY], the initialized right hand side data.

\*/

{

double fnorm;

int i;

int j;

double x;

double y;

for ( j = 0; j < ny; j++ )

{

y = ( double ) ( j ) / ( double ) ( ny - 1 );

for ( i = 0; i < nx; i++ )

{

x = ( double ) ( i ) / ( double ) ( nx - 1 );

if ( i == 0 || i == nx - 1 || j == 0 || j == ny - 1 )

{

f[i][j] = u\_exact ( x, y );

}

else

{

f[i][j] = - uxxyy\_exact ( x, y );

}

}

}

fnorm = r8mat\_rms ( nx, ny, f );

printf ( " RMS of F = %g\n", fnorm );

return;

}

void sweep ( int nx, int ny, double dx, double dy, double f[NX][NY],

double u[NX][NY], double unew[NX][NY] )

/\*

Purpose:

SWEEP carries out one step of the Jacobi iteration.

Discussion:

Assuming DX = DY, we can approximate

- ( d/dx d/dx + d/dy d/dy ) U(X,Y)

by

( U(i-1,j) + U(i+1,j) + U(i,j-1) + U(i,j+1) - 4\*U(i,j) ) / dx / dy

The discretization employed below will not be correct in the general

case where DX and DY are not equal. It's only a little more complicated

to allow DX and DY to be different, but we're not going to worry about

that right now.

Parameters:

Input, int NX, NY, the X and Y grid dimensions.

Input, double DX, DY, the spacing between grid points.

Input, double F[NX][NY], the right hand side data.

Input, double U[NX][NY], the previous solution estimate.

Output, double UNEW[NX][NY], the updated solution estimate.

\*/

{

int i;

int j;

for ( j = 0; j < ny; j++ )

{

for ( i = 0; i < nx; i++ )

{

if ( i == 0 || j == 0 || i == nx - 1 || j == ny - 1 )

{

unew[i][j] = f[i][j];

}

else

{

unew[i][j] = 0.25 \* (

u[i-1][j] + u[i][j+1] + u[i][j-1] + u[i+1][j] + f[i][j] \* dx \* dy );

}

}

}

return;

}

void timestamp ( void )

/\*

Purpose:

TIMESTAMP prints the current YMDHMS date as a time stamp.

\*/

{

# define TIME\_SIZE 40

static char time\_buffer[TIME\_SIZE];

const struct tm \*tm;

time\_t now;

now = time ( NULL );

tm = localtime ( &now );

strftime ( time\_buffer, TIME\_SIZE, "%d %B %Y %I:%M:%S %p", tm );

printf ( "%s\n", time\_buffer );

return;

# undef TIME\_SIZE

}

double u\_exact ( double x, double y )

/\*

Purpose:

U\_EXACT evaluates the exact solution.

Parameters:

Input, double X, Y, the coordinates of a point.

Output, double U\_EXACT, the value of the exact solution

at (X,Y).

\*/

{

double pi = 3.141592653589793;

double value;

value = sin ( pi \* x \* y );

return value;

}

double uxxyy\_exact ( double x, double y )

/\*

Purpose:

UXXYY\_EXACT evaluates ( d/dx d/dx + d/dy d/dy ) of the exact solution.

Parameters:

Input, double X, Y, the coordinates of a point.

Output, double UXXYY\_EXACT, the value of

( d/dx d/dx + d/dy d/dy ) of the exact solution at (X,Y).

\*/

{

double pi = 3.141592653589793;

double value;

value = - pi \* pi \* ( x \* x + y \* y ) \* sin ( pi \* x \* y );

return value;

}

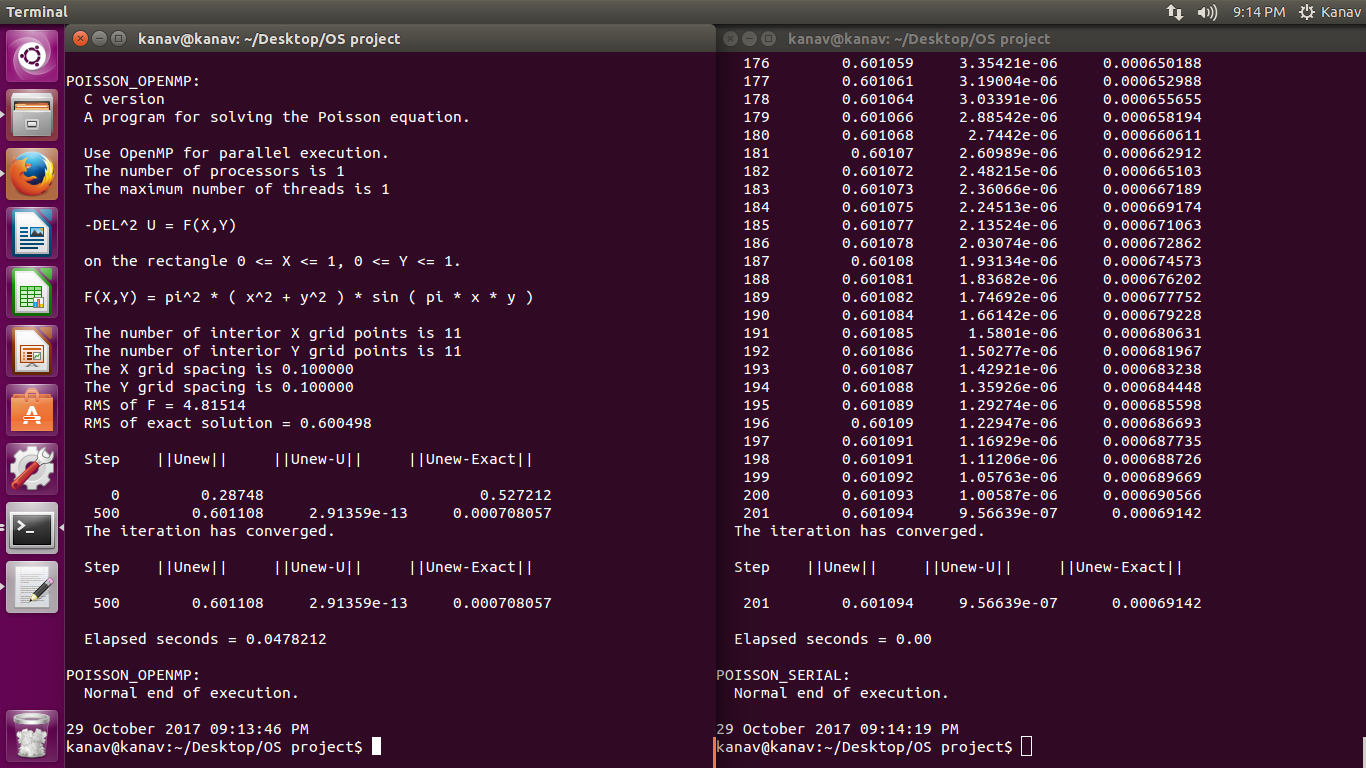
# undef NX

# undef NY

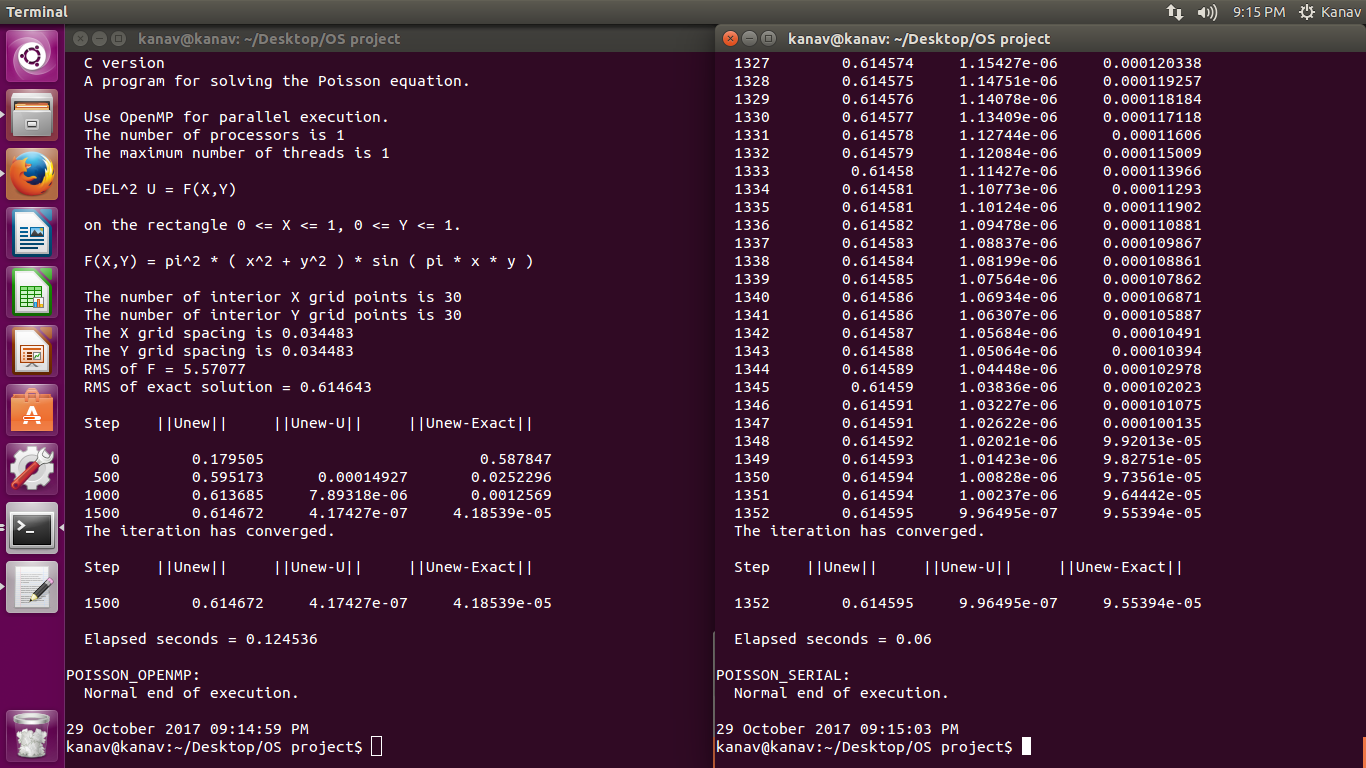
**Results and Discussion**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No.** | **Grid Dimensions** | **Execution time in secs (Parallel)** | **Execution time in secs (Serial)** |
| 1. | 11 x 11 | 0.0478 | 0.00 |
| 2. | 30 x 30 | 0.124536 | 0.06 |
| 3 | 83 x 83 | 2.3832 | 2.25 |
| 4. | 161 x 161 | 23.5992 | 30.54 |
| 5. | 232 x 232 | 105.52 | 130.97 |

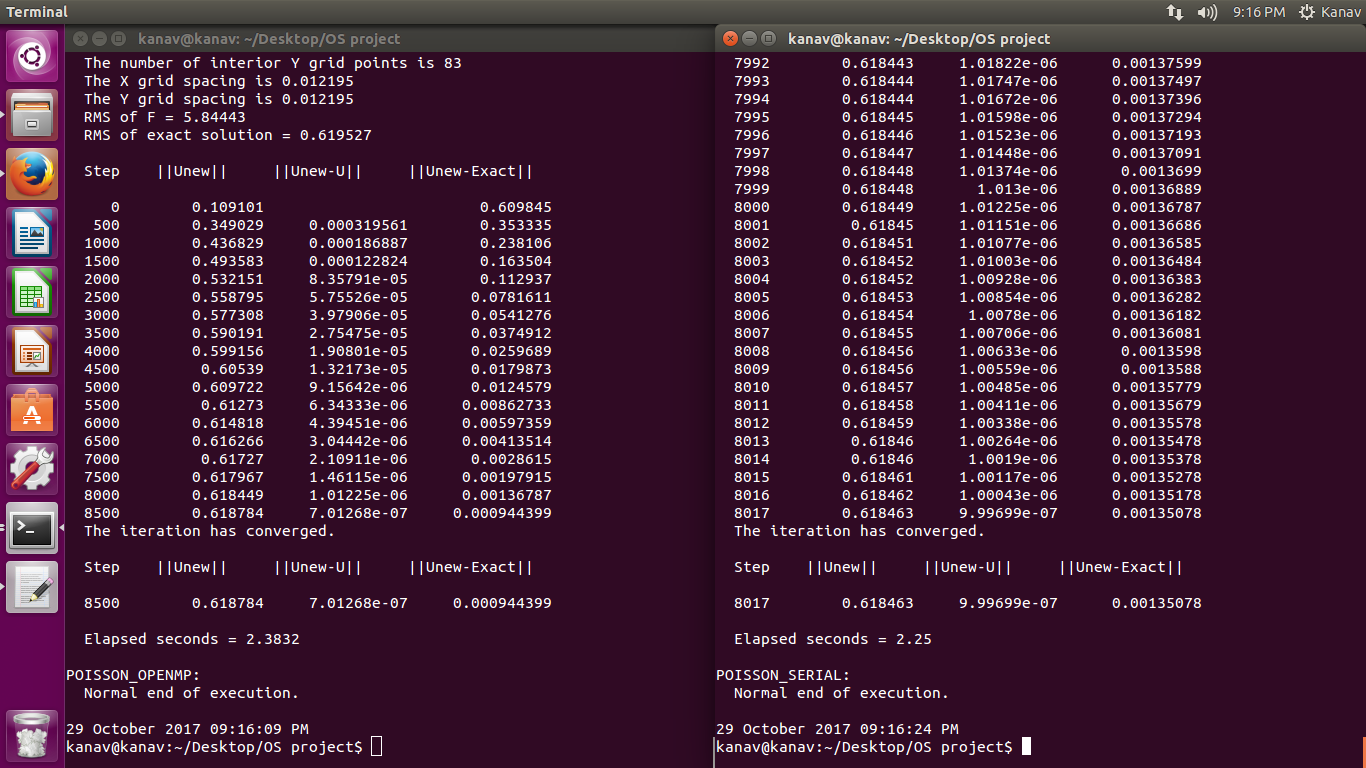
**Output Screenshots**



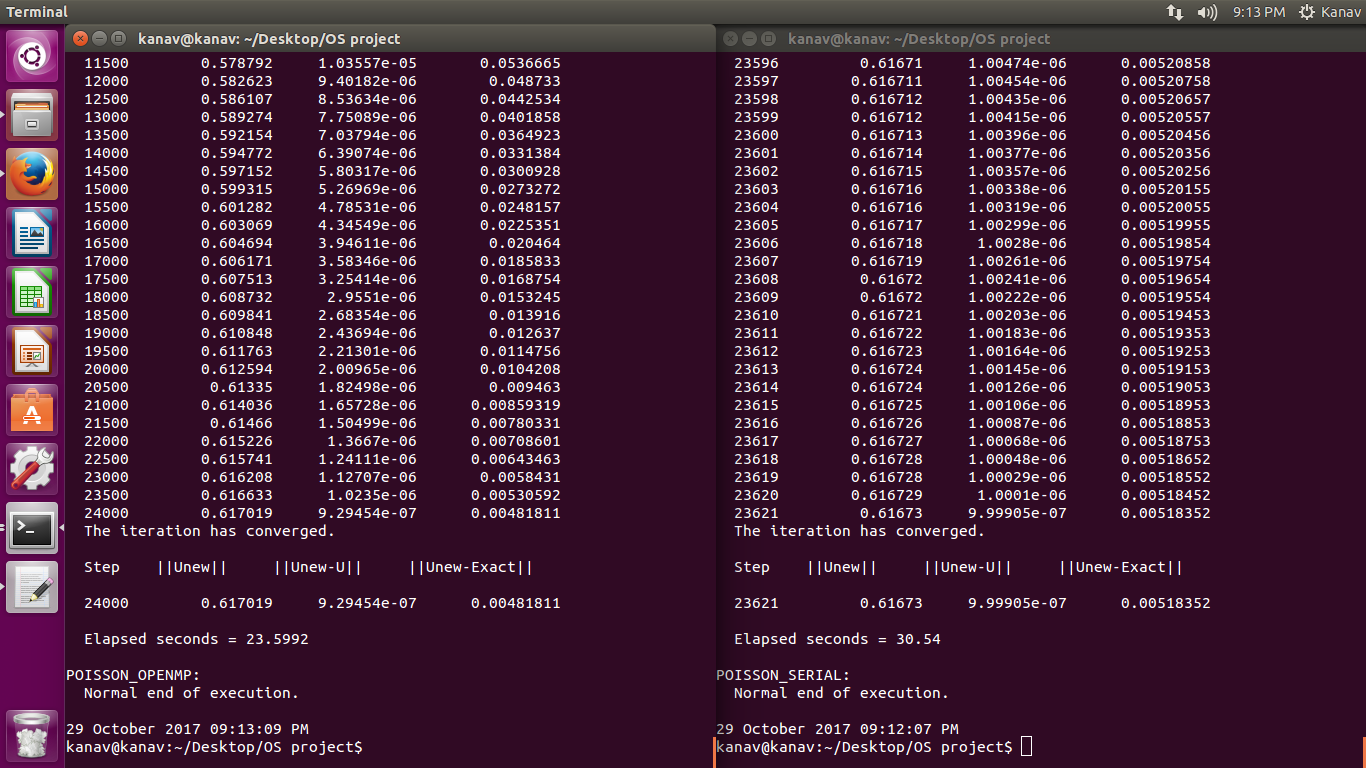
**Output for grid size 11 x 11**



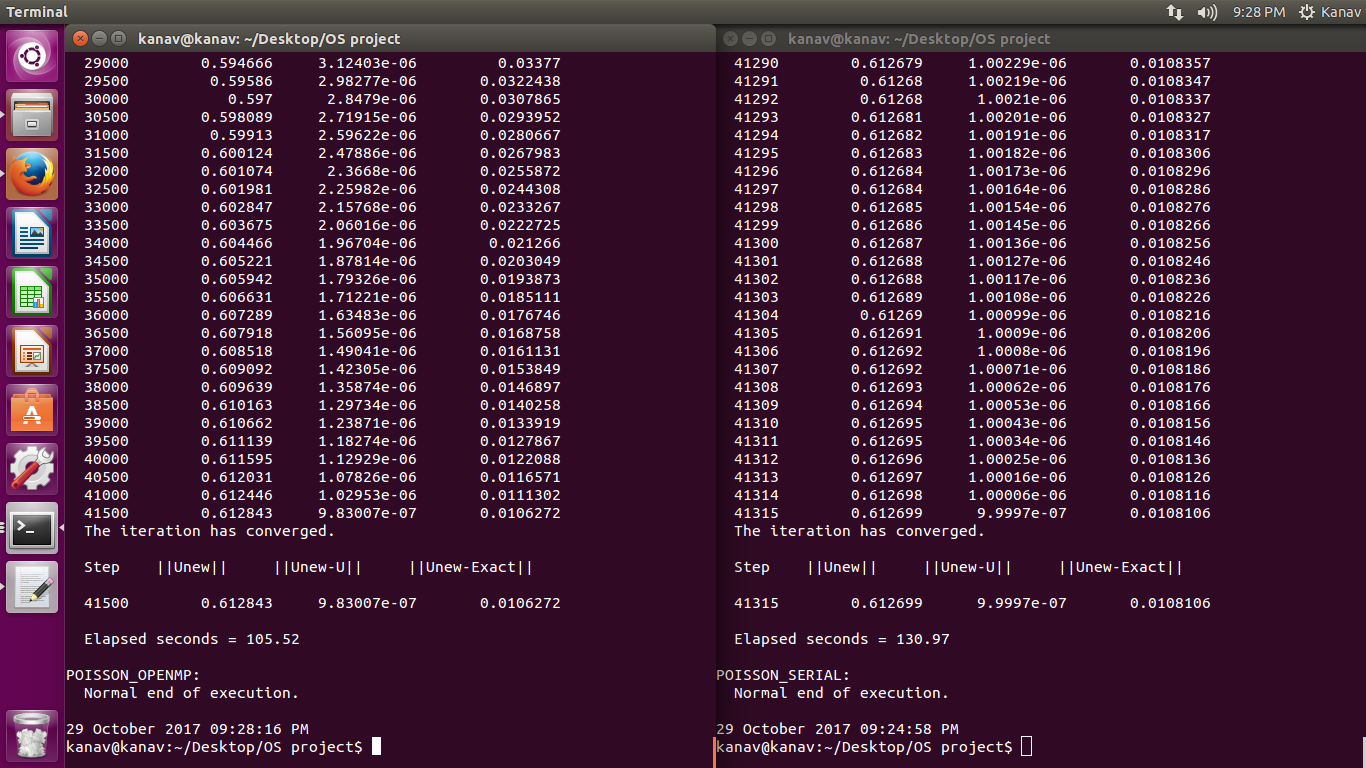
**Output for grid size 30 x 30**



**Output for grid size 83 x 83**



**Output for grid size 162 x 162**



**Output for grid size 232 x 232**

**Conclusion**

It is clearly seen from the above-mentioned results that the parallel approach used to solve the Poisson Equation using Jacobi iteration is a more efficient form of execution for larger defined Grid Dimensions. This also holds true to the core value of Parallelization, which is to provide both time and resource efficient solutions to complex and bulky problems.

The code provides high time efficiency for large Grid Dimensions, reducing execution time by using parallel threads.

**Future Enhancements**

Attempt will be made to identify core applications of our time efficient parallelization method and its necessary adaptations related to it for successful execution.

**References**

1. Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick. (2006). The Landscape of Parallel Computing Research: A View from Berkeley. *EECS Department: Technical Report No.* UCB/EECS-2006-183.
2. Kunle Olukotun, Lance Hammond. (2005). The Future of Microprocessors. *ACM ISSN: 1542-7730 EISSN: 1542-7749 doi.*
3. Dr. Man-Chung Yeung, Mr. Jon Held. (2005). A Comparison of Some Iterative Methods in Scientific Computing. *Department of Mathematics University of Wyoming, Summer Research Apprentice Program.*
4. Edmond Jajaga. (2012). MPI parallel implementation of Jacobi. *ICT Innovations Web Proceedings ISSN 1857-7288.*
5. Manashwi Tamuli, Shreyasee Debnath, Swanirbhar Majumdar, Ashok Ray. (2015). A Review on Jacobi Iterative Solver and it’s Hardware based performance analysis. *Research Gate Publication 278329438.*
6. Rashid Mehmood, Jon Crowcroft. (2005). Parallel iterative solution method for large sparse linear equation systems. *Technical Report UCAM-CL-TR-650 ISSN 1476-2986*.
7. R.W. Hockney. (1965). A Fast Direct Solution of Poisson's Equation Using Fourier Analysis. *Journal of the ACM*, 12(1). 95-113.
8. Pek´arek and R. Hrach Charles. (2006). A Comparison of Advanced Poisson Equation Solvers Applied to the Particle-In-Cell Plasma Model Z. *WDS'06 Proceedings of Contributed Papers*, Part III, 187–192.
9. Yiming Li. (2003). A parallel monotone iterative method for the numerical solution of multi-dimensional semiconductor Poisson equation. *Computer Physics Communications,* 153 (2003) 359–372.
10. (1996). https://people.eecs.berkeley.edu/~demmel/cs267/lecture24/lecture24.html
11. MPI Parallel Implementation of Jacob