2016201278 Jinyeong Wang

Final Project

1. Compute poisson equation using Jacobi iteration method.
2. Use at least 500 grid nodes.
3. Run more than 5 cases with different Np.
4. Compare execution time, L2 error.

- Poisson equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \cos[\pi x]\sin[y] + \pi^2\cos[\pi x]\sin[y]$$

- While

$$0 \le x \le 2 \; and \; 0 \le y \le 2$$

- Exact Solution

$$u_{exact}(x, y) = \cos[\pi x]\sin[\pi + y]$$

- Terminate compute when

$$\sum |u_m^{k+1} - u_m^k| < 10^{-4}$$

- Grid nodes are
  - 500
  - 525
  - 550
  - 575
  - 600
- Dirichlet boundary condition

$$\begin{cases} u(0, y) = \sin[\pi + y] \\ u(2, y) = \sin[\pi + y] \\ u(x, 0) = 0 \\ u(x, 2) = \cos[\pi x]\sin[\pi + 2] \end{cases}$$

2016201278 Jinyeong Wang

1. Derive discretized equation

   A. Poisson equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \cos[\pi x]\sin[y] + \pi^2\cos[\pi x]\sin[y]$$

   B. Discretize

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} = \cos[\pi x]\sin[y] + \pi^2\cos[\pi x]\sin[y]$$

   C. Adopt Jacobi iteration method

$$u_{i,j}^{k+1} = \frac{\Delta y^2\left(u_{i+1,j}^k + u_{i-1,j}^k\right) + \Delta x^2\left(u_{i,j+1}^k + u_{i,j-1}^k\right) - \Delta x^2 \Delta y^2 f_{i,j}}{2(\Delta x^2 + \Delta y^2)}$$

$$while\ \ f_{i,j} = \cos[\pi(i\Delta x)]\sin[j\Delta y] + \pi^2\cos[\pi(i\Delta x)]\sin[j\Delta y]$$

   D. Adopt problem condition and simplify equation

$$u_{i,j}^{k+1} = \left(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k - \Delta^2 f_{i,j}\right) * 0.25$$

2. Pseudo code

Parallelize memory indexing

```
N_fix = N * N
While( N_fix % size )
   N_fix++
rows_p = N_fix / size
If( rank == size-1 )
   rows_p = N * N - N_fix / size   * (size - 1)
U = memory allocation( N * N )
U_p = memory allocation( rows_p )
counts = memory allocation( size )
displs = memory allocation( size )
Sum = 0
for( i, 0 to size - 1 )
   counts[ i ] = N_fix / size
   displs[ i ] = sum
   sum += counts[ i ]
counts[ size − 1 ] = N * N − sum
displs[ size − 1 ] = sum
```

Boundary condition

```
for ( index, 0 to counts[rank] )
   pos = index + displs[rank]
   i = pos % N
   j = pos / N
   if( pos is not boundary )
      U_p[pos] = 0.0
   else if ( 0,y )
      U_p[pos] = sin(pi*delta*j)
   else if ( 2,y )
      U_p[pos] = sin(pi*delta*j)
   else if ( x,0 )
      U_p[pos] = 0.0
   else if ( x,2 )
      U_p[pos] = cos(pi*delta*i)*sin(pi+2.0)
```

Jacobi iteration method

```
Tol = 0.0001
Error = 1.0
Repeat
   MPI_Allgatherv( U_p to U )
   For ( index = 0 ; index < counts[rank] ; index++ ){
      Pos = index + displs[rank]
      I = pos % N
      J = pos / N
      X = delta * i
      Y = delta * j
      F = cos(pi*x)sin(y)+pi*pi*cos(pi*x)*sin(y)
      If ( pos is not boundary )
         U_p[index] = (U[pos+1]+U[pos-1]+U[pos+N]+U[pos-N]-delta*delta*f)*0.25
   Error_p = 0.0
   For ( index = 0 ; index < counts[rank] ; index++ ){
      Pos = index + displs[rank]
      Error_p += fabs(U_p[index] – U[pos])}
   MPI_Allreduce(Error_p to Error, sum)
Untill Error < Tol
```

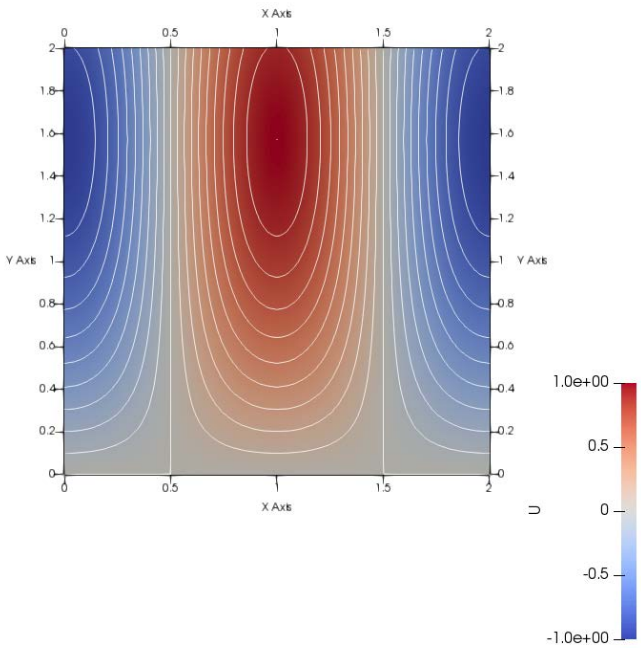
3. Result

   A. Capture of results

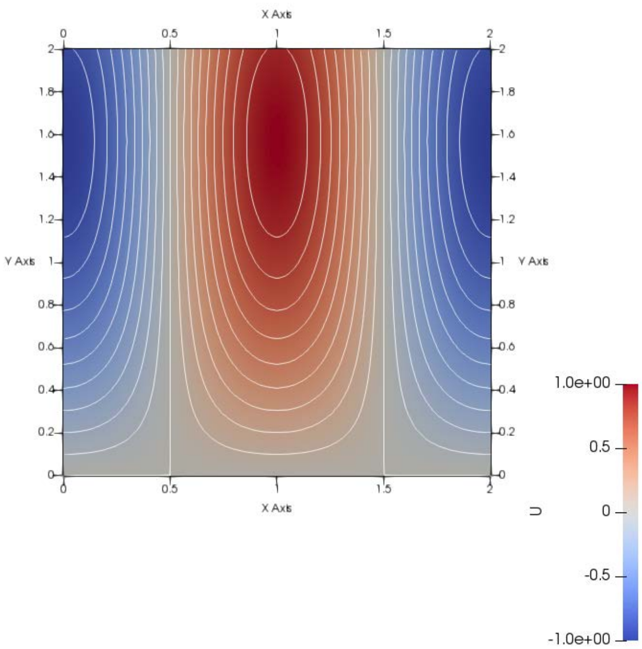
- Elliptic, N=500, Exact
- Elliptic, N=500, L2=0.0000000391478, np=1, t=4555.590961, Jacobi
- Elliptic, N=500, L2=0.0000000391478, np=2, t=2383.236201, Jacobi
- Elliptic, N=500, L2=0.0000000391478, np=3, t=1703.346945, Jacobi
- Elliptic, N=500, L2=0.0000000391478, np=4, t=1290.001672, Jacobi
- Elliptic, N=500, L2=0.0000000391478, np=5, t=1086.956939, Jacobi
- Elliptic, N=500, L2=0.0000000391478, np=6, t=954.962086, Jacobi
- Elliptic, N=525, Exact
- Elliptic, N=525, L2=0.0000000382156, np=1, t=5476.488216, Jacobi
- Elliptic, N=525, L2=0.0000000382156, np=2, t=2950.183918, Jacobi
- Elliptic, N=525, L2=0.0000000382156, np=3, t=2062.157977, Jacobi
- Elliptic, N=525, L2=0.0000000382156, np=4, t=1570.662798, Jacobi
- Elliptic, N=525, L2=0.0000000382156, np=5, t=1317.836718, Jacobi
- Elliptic, N=525, L2=0.0000000382156, np=6, t=1169.407257, Jacobi
- Elliptic, N=550, Exact
- Elliptic, N=550, L2=0.0000000372543, np=1, t=6655.835934, Jacobi
- Elliptic, N=550, L2=0.0000000372543, np=2, t=3568.105374, Jacobi
- Elliptic, N=550, L2=0.0000000372543, np=3, t=2486.185437, Jacobi
- Elliptic, N=550, L2=0.0000000372543, np=4, t=1902.978690, Jacobi
- Elliptic, N=550, L2=0.0000000372543, np=5, t=1596.473933, Jacobi
- Elliptic, N=550, L2=0.0000000372543, np=6, t=1434.189380, Jacobi
- Elliptic, N=575, Exact
- Elliptic, N=575, L2=0.0000000362858, np=1, t=7971.567872, Jacobi
- Elliptic, N=575, L2=0.0000000362858, np=2, t=4261.728387, Jacobi
- Elliptic, N=575, L2=0.0000000362858, np=3, t=2976.539548, Jacobi
- Elliptic, N=575, L2=0.0000000362858, np=4, t=2282.359113, Jacobi
- Elliptic, N=575, L2=0.0000000362858, np=5, t=1923.139598, Jacobi
- Elliptic, N=575, L2=0.0000000362858, np=6, t=1741.285618, Jacobi
- Elliptic, N=600, Exact
- Elliptic, N=600, L2=0.0000000353238, np=1, t=9247.522723, Jacobi
- Elliptic, N=600, L2=0.0000000353238, np=2, t=5080.173374, Jacobi
- Elliptic, N=600, L2=0.0000000353238, np=3, t=3598.920527, Jacobi
- Elliptic, N=600, L2=0.0000000353238, np=4, t=2773.763258, Jacobi
- Elliptic, N=600, L2=0.0000000353238, np=5, t=2351.110034, Jacobi
- Elliptic, N=600, L2=0.0000000353238, np=6, t=2152.320612, Jacobi

B. Contour

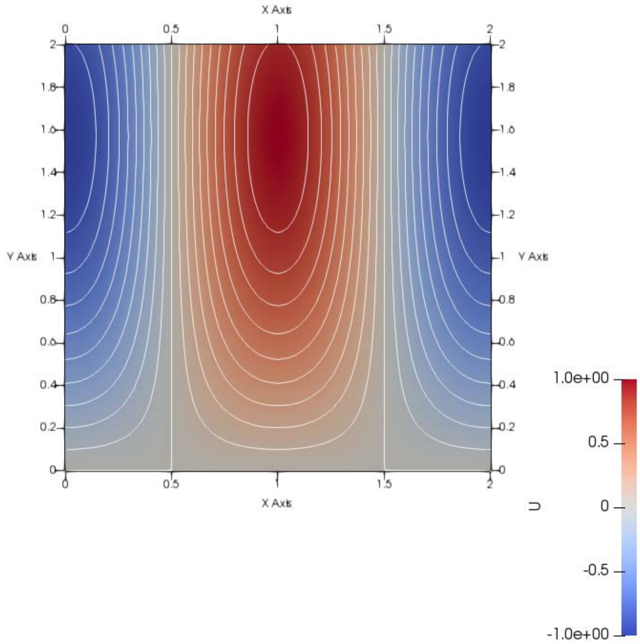
i. Grid nodes = 500



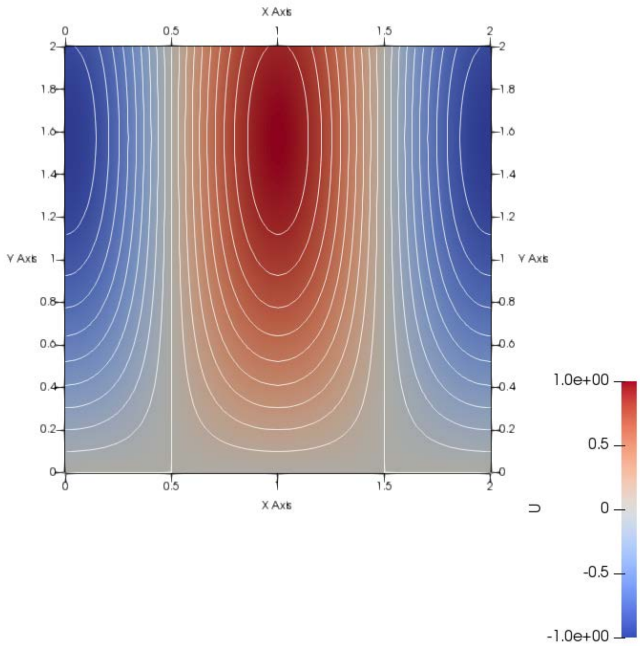
ii. Grid nodes = 525



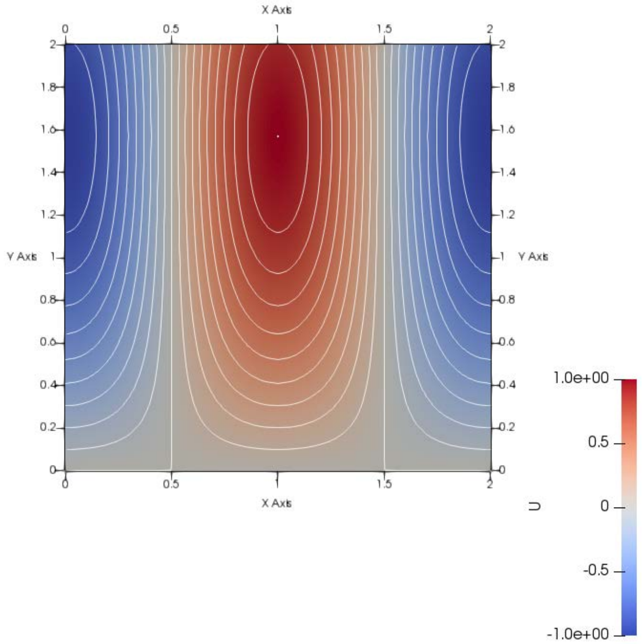
iii. Grid nodes = 550



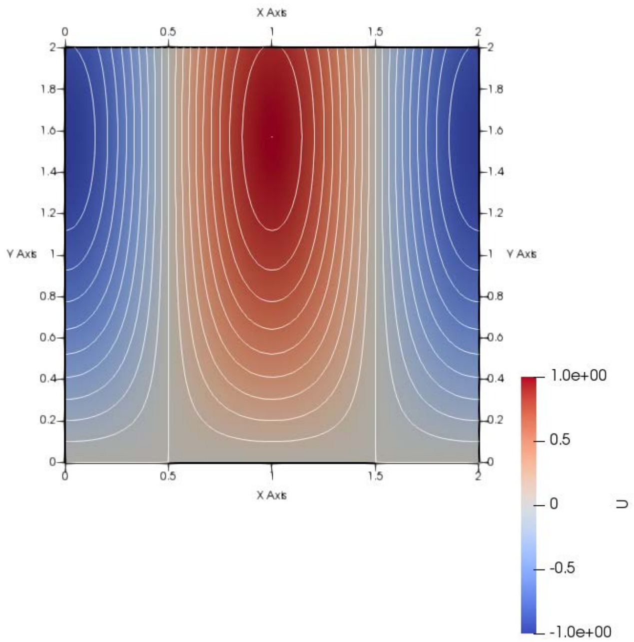
iv. Grid nodes = 575



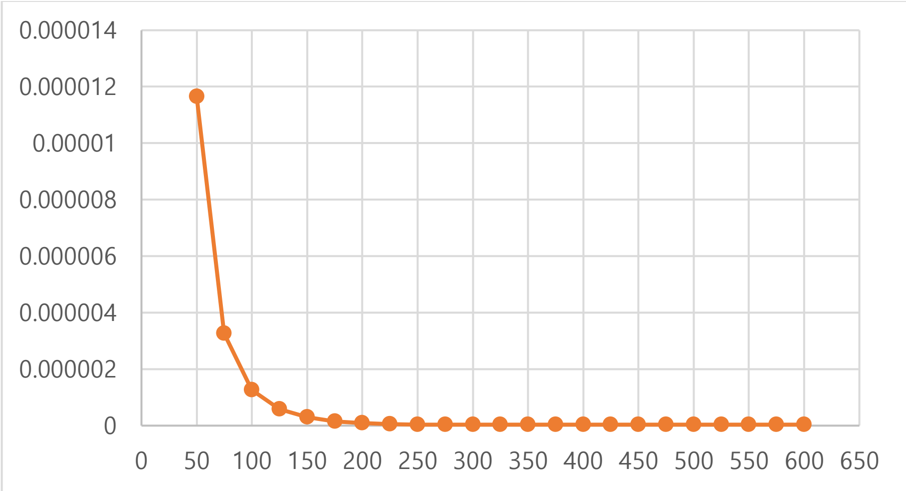
v. Grid nodes = 600



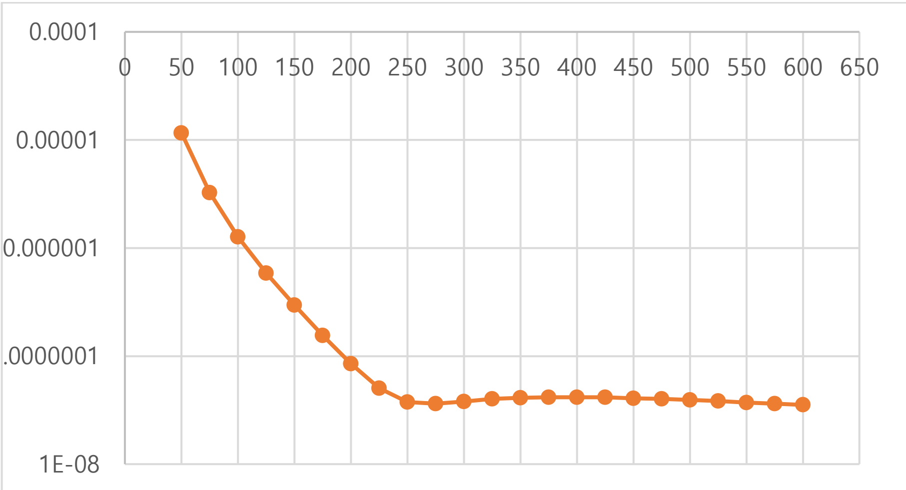
vi. Exact solution



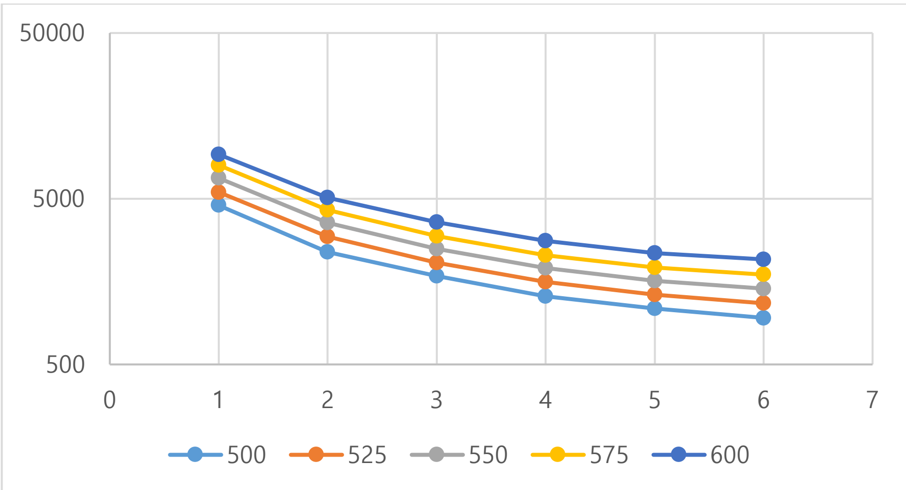
C. L2 error vs Grid nodes



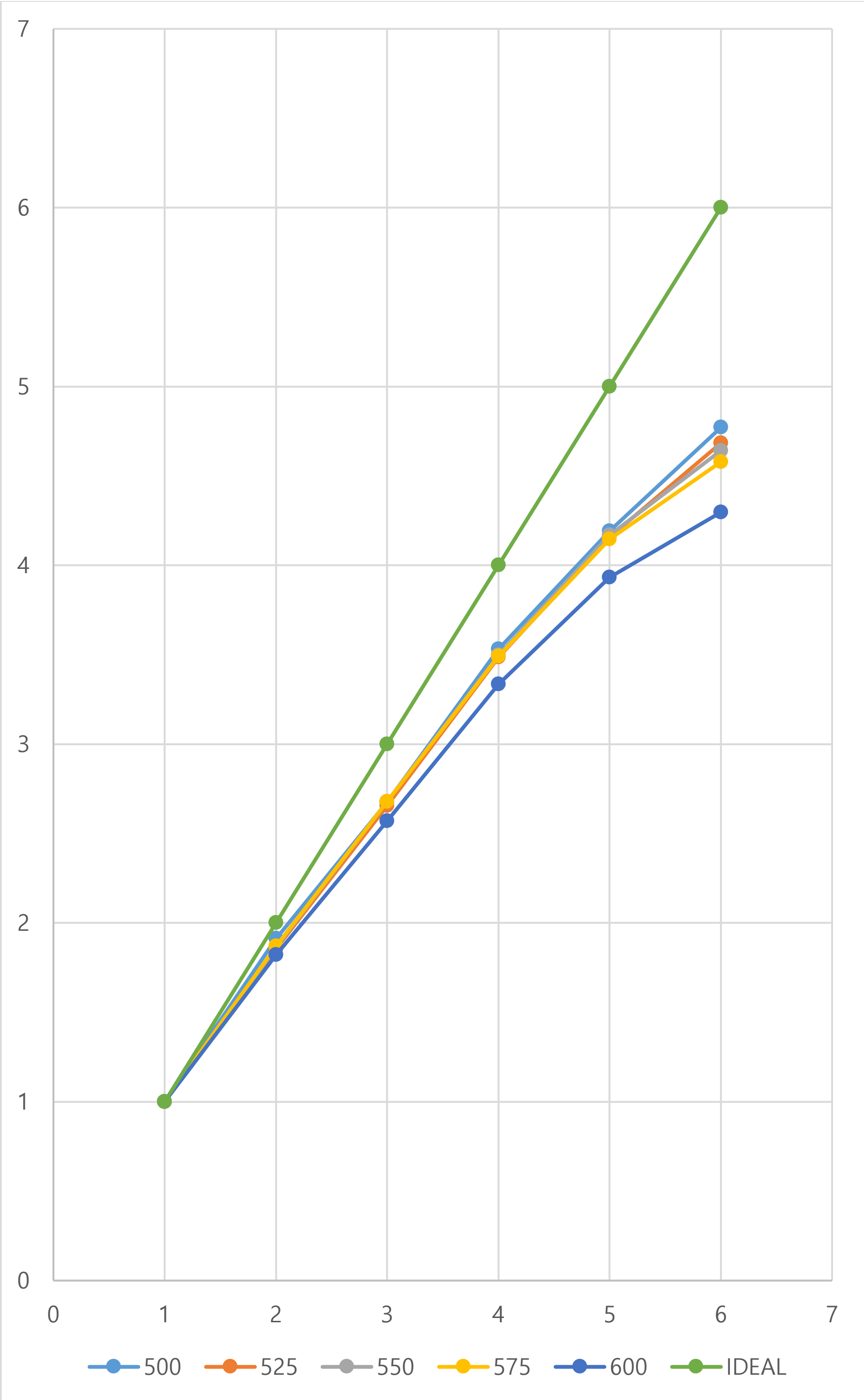
D. Log(L2) vs Grid nodes



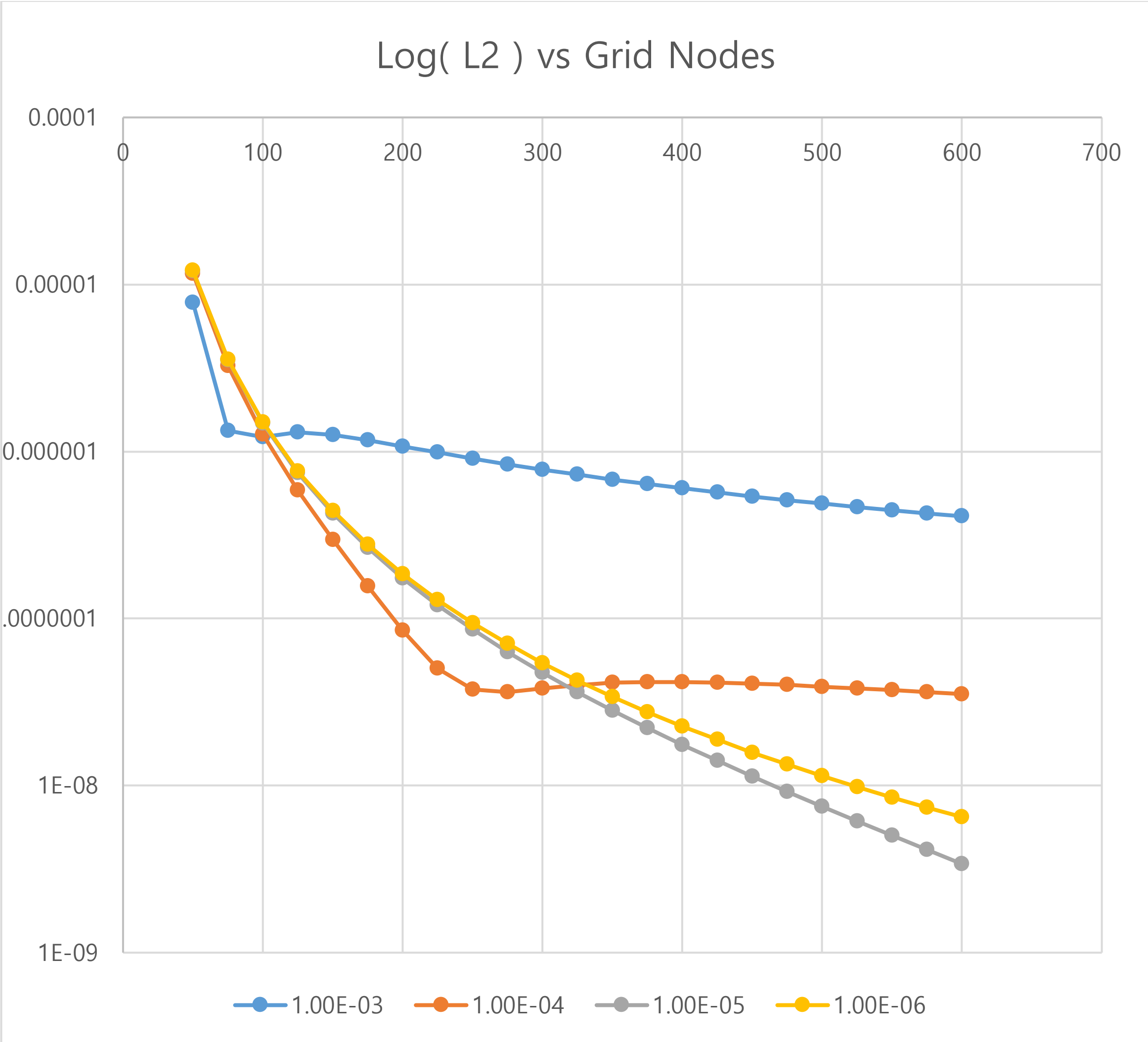
E. Log(Computation time) vs Number of processors



F. Speedup



G. Log(L2) vs Grid nodes for different L1 error tolerance



4. Conclusion

Regardless of the number of processors, the L2 error is same for each grid nodes.

In terms of grid size from 50 to 250, error decrease by reduce of round-off error. But when grid size is more than 250, error does not decrease because of truncation error.

But this phenomenon caused by L1 error tolerance. When L1 error tolerance is 1e-5 and 1e-6, L2 error shows gradually decrease.

The computation time is different each time. But it can't explain low scalability of grid size 600. In main computation process, there exist 2 loops which can reduce to 1 loop. If modify and re-run the code, computation time could be different from result shown.

5. Codes

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>
#define PI 3.14159265359
#define Tol 0.0001
void Elliptic(int N, int rank, int size);
void FileWriter(double *U, int N, double delta, double time, int size);
double L2Error(double *U, int N, double delta);
void ExactWriter(int N, double delta);
int main(int argc, char **argv)
{
    int rank, size, N;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(rank == 0)
    {
        fflush(stdin);
        printf("It will Elliptic, Jacobi Method ₩n");
        printf("Input N₩n");
        scanf("%d", &N);
    }
    MPI_Bcast(&N,1,MPI_INT,0,MPI_COMM_WORLD);
    Elliptic(N, rank, size);
    MPI_Finalize();
}
void Elliptic(int N, int rank, int size)
{
    int NN = N*N, N_fix = N*N;
    int rows_p, index, i, j, pos, iter=0;
    double tic, toc, x, y, f, L1, delta = 2.0/((double)(N-1));
    double Error=0.0, Error_p=0.0;
    while(N_fix%size!=0)
        N_fix++;
    rows_p = (int)(N_fix/size);
    if(rank == size-1)
        rows_p = NN-rows_p*(size-1);
    double *U = (double *)malloc(NN*sizeof(double));
    double *U_p = (double *)calloc(rows_p, sizeof(double));
    int *counts = malloc(sizeof(int)*size);
    int *displs = malloc(sizeof(int)*size);
    int sum = 0;
    for( i = 0 ; i < size-1 ; i++ )
    {
        counts[i] = (int)(N_fix/size);
        displs[i] = sum;
        sum += counts[i];
    }
    counts[size-1] = NN-sum;
    displs[size-1] = sum;
    for( index = 0 ; index < counts[rank] ; index++ )
    {
```

```
        pos = index + displs[rank];
        i = (int)(pos % N);
        j = (int)(pos / N);
        if( pos != j * N + i )
        {
            printf("Wrong index₩n");
        }
        if( i != 0 && j != 0 && i != N-1 && j != N-1 )
            U_p[index] = 0.0;
        else if( i == 0 )   // 0,y
            U_p[index] = sin(PI+delta*j);
        else if( i == N-1 ) // 2,y
            U_p[index] = sin(PI+delta*j);
        else if( j == 0 )   // x,0
            U_p[index] = 0.0;
        else if( j == N-1 ) // x,2
            U_p[index] = cos(PI*delta*i)*sin(PI+2.0);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == 0 )
        tic = MPI_Wtime();
    Error = 1.0;
    while(Error > Tol)
    {
        iter++;
        MPI_Allgatherv(U_p, counts[rank], MPI_DOUBLE, U, counts, displs, MPI_DOUBLE,
MPI_COMM_WORLD);
        for( index = 0 ; index < counts[rank] ; index++ )
        {
            pos = index + displs[rank];
            i = (int)(pos % N);
            j = (int)(pos / N);
            x = delta * i;
            y = delta * j;
            f = cos(PI*x)*sin(y)+PI*PI*cos(PI*x)*sin(y);
            if( i != 0 && j != 0 && i != N-1 && j != N-1 )
                U_p[index] = (U[pos+1]+U[pos-1]+U[pos+N]+U[pos-N]-delta*delta*f)*0.25;
        }
        Error_p = 0.0;
        for( index = 0 ; index < counts[rank] ; index++ )
        {
            pos = index + displs[rank];
            Error_p += fabs(U_p[index] - U[pos]);
        }
        MPI_Allreduce(&Error_p,&Error,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
        if( rank == 0 )
            printf("₩rN=%d, iter=%d, L1=%.13lf",N,iter,Error);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == 0 )
    {
        toc = MPI_Wtime();
        printf("₩n");
    }
    MPI_Gatherv(U_p, counts[rank], MPI_DOUBLE, U, counts, displs, MPI_DOUBLE, 0,
```

```
MPI_COMM_WORLD);
    if( rank == 0 )
    {
        FileWriter(U, N, delta, toc-tic, size);
        ExactWriter(N, delta);
    }
    free(U);
    free(U_p);
    free(counts);
    free(displs);
}
void FileWriter(double *U, int N, double delta, double time, int size)
{
    double L2 = L2Error(U, N, delta);
    FILE *Solve;
    char Solname[100];
    int i,j,pos;
    double x,y;
    sprintf(Solname, "Elliptic, N=%d, L2=%.13lf, np=%d, t=%lf, Jacobi.csv",N,L2, size,
time);
    Solve = fopen(Solname, "w");
    fprintf(Solve, "X,Y,U₩n");
    for( j = 0 ; j < N ; j++ )
    {
        for( i = 0 ; i < N ; i++ )
        {
            pos = j * N + i;
            x = delta*i;
            y = delta*j;
            fprintf(Solve, "%lf,%lf,%lf₩n", x, y, U[pos]);
        }
    }
    fclose(Solve);
    return;
}
double L2Error(double *U, int N, double delta)
{
    double error = 0.0, x, y;
    int i,j;
    for( j = 0 ; j < N ; j++ )
    {
        for( i = 0 ; i < N ; i++ )
        {
            int pos = j*N+i;
            x = delta*i;
            y = delta*j;
            double abs_error = fabs(U[pos]-cos(PI*x)*sin(PI+y));
            error += abs_error * abs_error;
        }
    }
    error = sqrt(error)/(double)(N*N);
    return error;
}
void ExactWriter(int N, double delta)
{
```

```
    FILE *fp;
    char name[50];
    int i, j;
    double x, y, Exact;
    sprintf(name, "Elliptic, N=%d, Exact.csv", N);
    fp = fopen(name, "w");
    fprintf(fp, "X,Y,U₩n");
    for( j = 0 ; j < N ; j++ )
    {
        for( i = 0 ; i < N ; i++ )
        {
            x = delta*i;
            y = delta*j;
            Exact = cos(PI*x)*sin(PI+y);
            fprintf(fp, "%lf,%lf,%lf₩n", x, y, Exact);
        }
    }
    fclose(fp);
}
```