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
* While
* Exact Solution
* Terminate compute when
* Grid nodes are
  + 500
  + 525
  + 550
  + 575
  + 600
* Dirichlet boundary condition

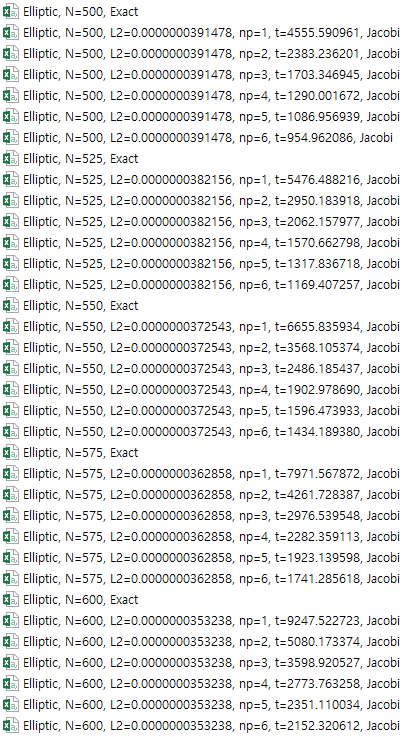
1. Derive discretized equation
   1. Poisson equation
   2. Discretize
   3. Adopt Jacobi iteration method
   4. Adopt problem condition and simplify equation
2. Pseudo code

|  |
| --- |
| Parallelize memory indexing |
| 1. N\_fix = N \* N 2. While( N\_fix % size ) 3. N\_fix++ 4. rows\_p = N\_fix / size 5. If( rank == size-1 ) 6. rows\_p = N \* N - N\_fix / size \* (size - 1) 7. U = memory allocation( N \* N ) 8. U\_p = memory allocation( rows\_p ) 9. counts = memory allocation( size ) 10. displs = memory allocation( size ) 11. Sum = 0 12. for( i, 0 to size - 1 ) 13. counts[ i ] = N\_fix / size 14. displs[ i ] = sum 15. sum += counts[ i ] 16. counts[ size – 1 ] = N \* N – sum 17. displs[ size – 1 ] = sum |

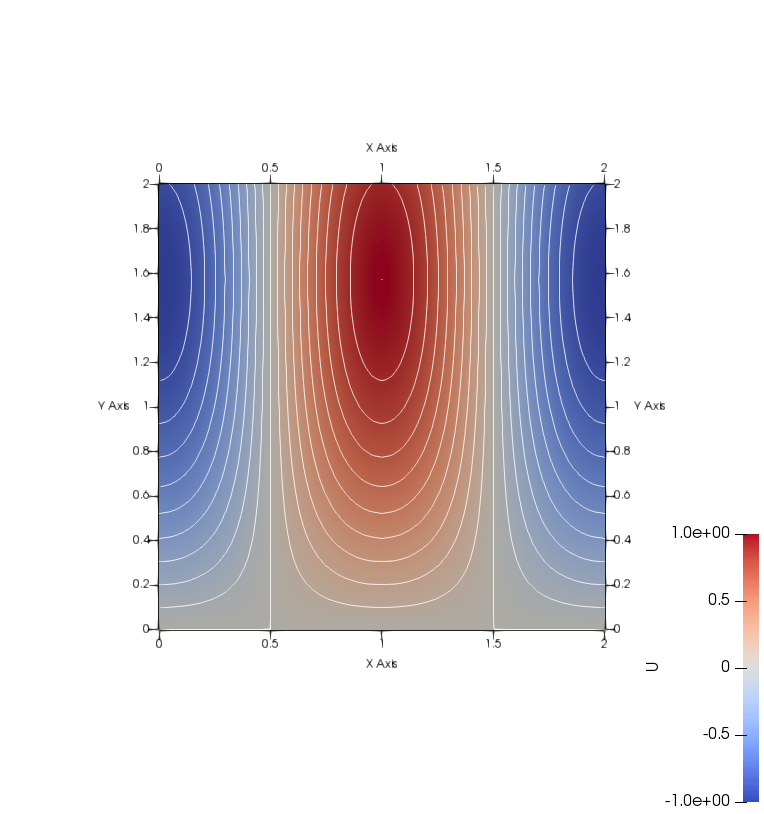
|  |
| --- |
| Boundary condition |
| 1. for ( index, 0 to counts[rank] ) 2. pos = index + displs[rank] 3. i = pos % N 4. j = pos / N 5. if( pos is not boundary ) 6. U\_p[pos] = 0.0 7. else if ( 0,y ) 8. U\_p[pos] = sin(pi\*delta\*j) 9. else if ( 2,y ) 10. U\_p[pos] = sin(pi\*delta\*j) 11. else if ( x,0 ) 12. U\_p[pos] = 0.0 13. else if ( x,2 ) 14. U\_p[pos] = cos(pi\*delta\*i)\*sin(pi+2.0) |

|  |
| --- |
| Jacobi iteration method |
| 1. Tol = 0.0001 2. Error = 1.0 3. Repeat 4. MPI\_Allgatherv( U\_p to U ) 5. For ( index = 0 ; index < counts[rank] ; index++ ){ 6. Pos = index + displs[rank] 7. I = pos % N 8. J = pos / N 9. X = delta \* i 10. Y = delta \* j 11. F = cos(pi\*x)sin(y)+pi\*pi\*cos(pi\*x)\*sin(y) 12. If ( pos is not boundary ) 13. U\_p[index] = (U[pos+1]+U[pos-1]+U[pos+N]+U[pos-N]-delta\*delta\*f)\*0.25 14. Error\_p = 0.0 15. For ( index = 0 ; index < counts[rank] ; index++ ){ 16. Pos = index + displs[rank] 17. Error\_p += fabs(U\_p[index] – U[pos])} 18. MPI\_Allreduce(Error\_p to Error, sum) 19. Untill Error < Tol |

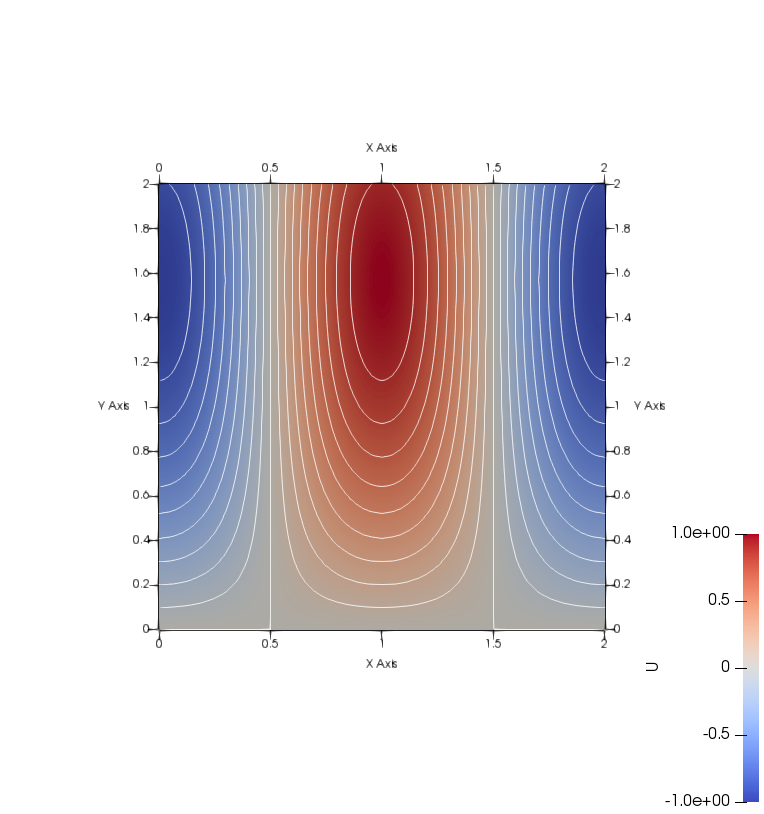
1. Result
   1. Capture of results



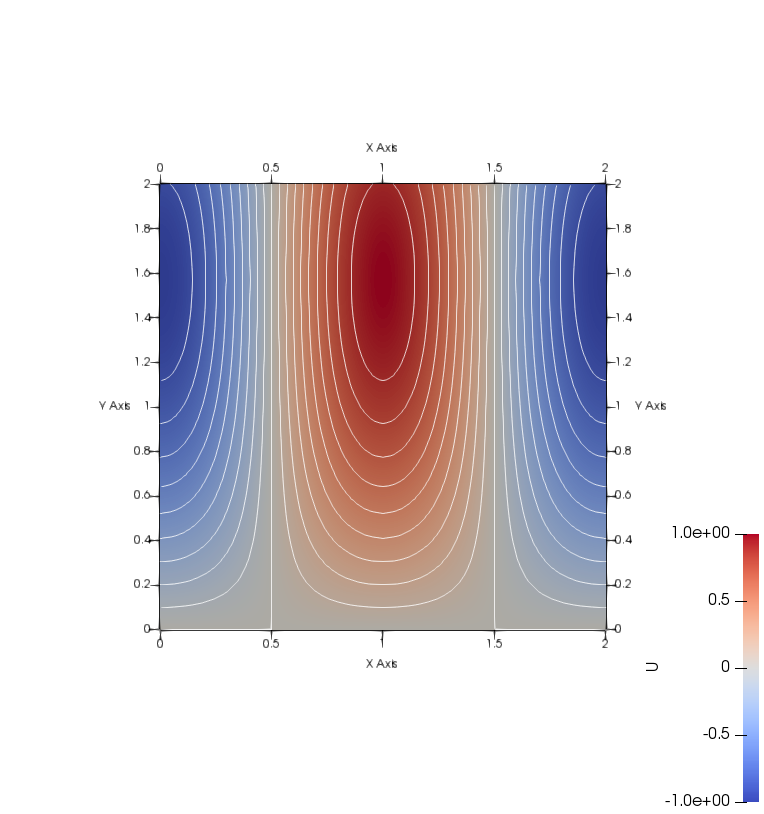
* 1. Contour
     1. Grid nodes = 500



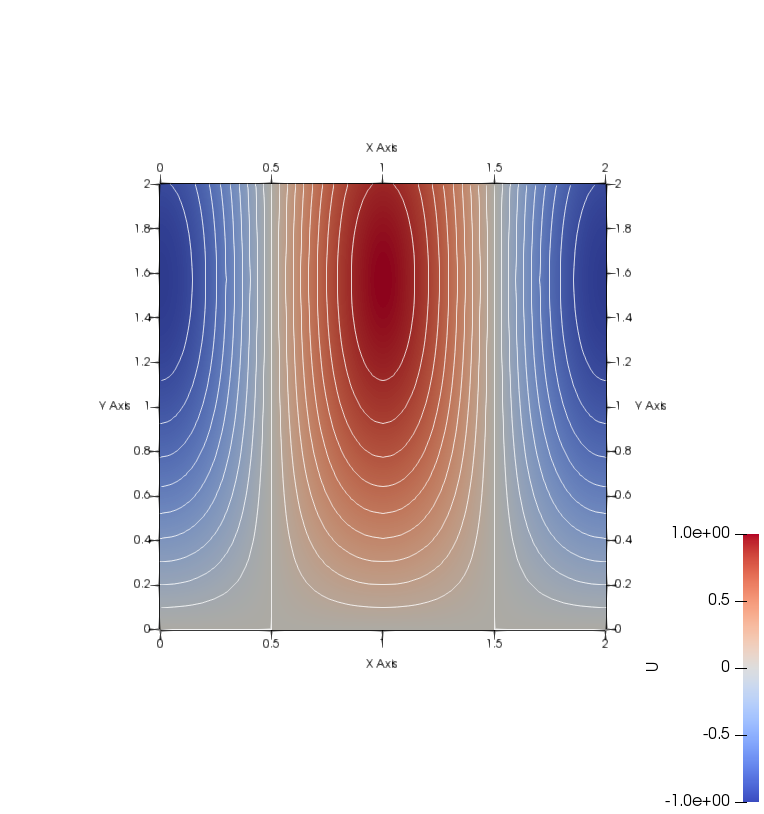
* + 1. Grid nodes = 525



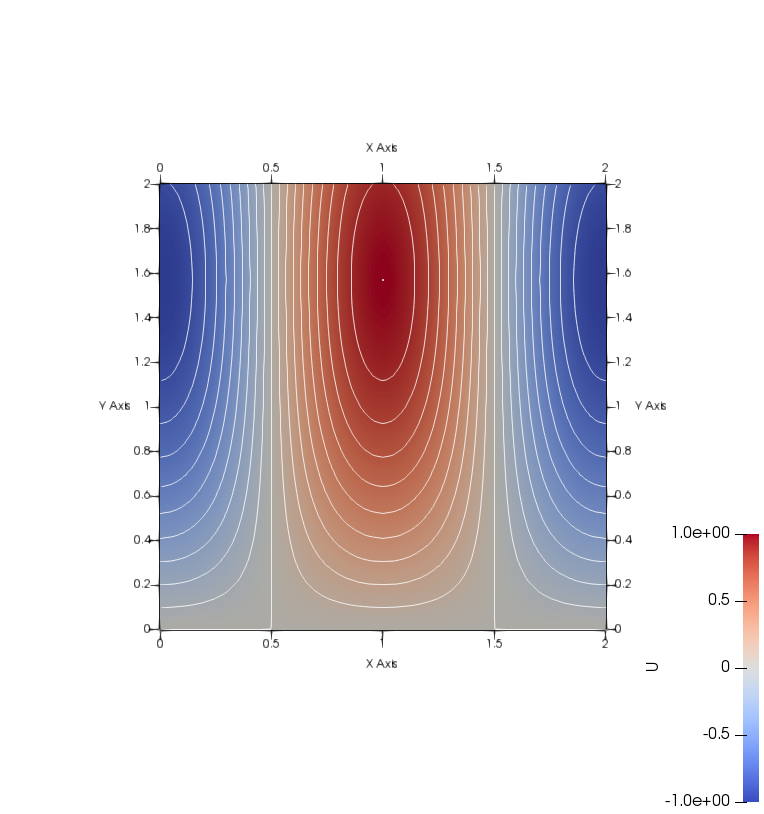
* + 1. Grid nodes = 550



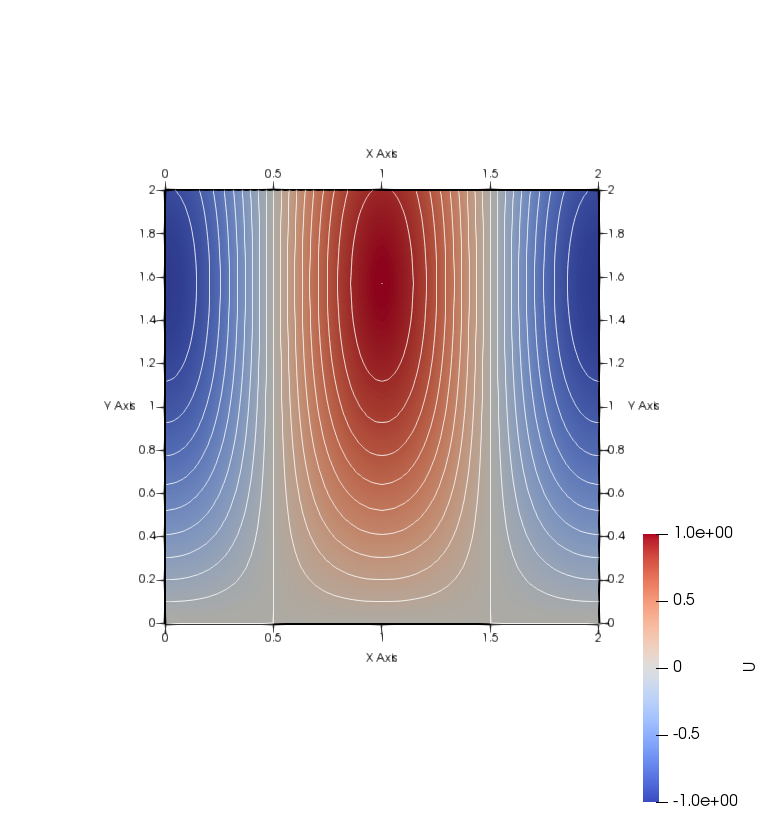
* + 1. Grid nodes = 575



* + 1. Grid nodes = 600



* + 1. Exact solution



* 1. L2 error vs Grid nodes
  2. Log( L2 ) vs Grid nodes
  3. Log( Computation time ) vs Number of processors
  4. Speedup
  5. Log( L2 ) vs Grid nodes for different L1 error tolerance

1. 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.

1. 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);  } |