CAAM 520: COMPUTATIONAL SCIENCE II
HOMEWORK 2.

WEI WU

1. Introduction

In this project, we build upon the previous project: we parallelize our linear system solver using MPI.

1.1. **Mesh Grid Partition.**

We partition the mesh grid by the rows, illustrated by the picture below.

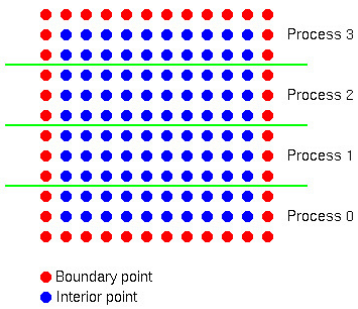

Figure 1. Mesh grid partition

However, I made the design decision to only partition the interior nodes. Let N be the number of interior nodes in a single row, and let s be the number of processors used. I try to divide the interior nodes by rows evenly among all processors. If rm is the remainder of N divided by s, I allocate extra rm rows to processor 0. Two extra rows of ghost nodes, one at the bottom and the other at the top, are allocated for the local u.

```
/*
Design decision: allocate extra N%size rows to processor 0,
and N/size + 2 to all other processes.
*/
int local_row_size = N/size + 2;
if (rank == 0){
  local_row_size += N%size;
}

double *local_u = (double*) calloc((local_row_size)*(N+2),sizeof(double));
double *f = (double*) calloc((local_row_size)*(N+2), sizeof(double));
```

```
double h = 2.0/(N+1);

for (int i = 0; i < N+2; ++i){
  for (int j = 0; j <
    local_row_size; ++j){
    const double x = -1.0 + i*h;
    double y = -1.0 + j*h;
    if (rank > 0)
      y += (rank - 1)*(N/size)*h + (N/size + N%size)*h;
    f[i + j*(N+2)] = sin(PI*x)*sin(PI*y) * h*h;
  }
}
```

1.2. **Parallel weighted Jacobi.** A very high level pseudocode of the parallel weighted Jacobi:

Algorithm 1 My algorithm

1: **procedure** MYALGO($N, tol, size, rank$)
2: **while** $global\_res \geq tol$ **do**
3: Send up unless $rank = size - 1$, then receive from $rank - 1$
4: Send down unless $rank = 0$, then receive from $rank + 1$
5: Compute u_local using weighted Jacobni; compute $local\_res$
6: Compute $global\_res$
7: **return** u_local

The above pseudocode corresponds to the following code snippet:

```
/*
Ghost nodes are stored on
u[0,...,N+1]
and
u[(local_row_size-1)*(N+2),...,(local_row_size)*(N+2)-1]
Note that for the bottom and top, the exterior nodes will be
ghost nodes.
*/
double *unew = (double*)calloc((N+2)*(local_row_size),sizeof(double));

double w = 1.0;
double invD = 1./4.; // factor of h cancels out

double global_res2 = 1.0;
unsigned int iter = 0;
MPI_Status status;
while(global_res2>tol*tol){

  double local_res2 = 0.0;
  for (int i=0; i<= N+1; ++i){
    /*Send up unless at the top, then receive from below*/
    if (rank < size - 1)
```

```
    MPI_Send(&u[i + (local_row_size-2)*(N+2)], 1, MPI_DOUBLE,
      rank + 1, 0, MPI_COMM_WORLD);
    if (rank > 0)
      MPI_Recv(&u[i], 1, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD, &status);
    /*Send down unless at the bottom, then receive from above*/
    if (rank > 0)
      MPI_Send(&u[i + 1*(N+2)], 1, MPI_DOUBLE,
      rank-1, 1, MPI_COMM_WORLD);
    if (rank < size - 1)
      MPI_Recv(&u[i + (local_row_size-1)*(N+2)], 1, MPI_DOUBLE,
    rank + 1, 1, MPI_COMM_WORLD, &status);
  }

// update interior nodes using Jacobi; does not touch ghost nodes
// int global_id;
for(int i=1; i<=N; ++i){
  for(int j=1; j<=local_row_size-2; ++j){
    const int id = i + j*(N+2); // x-index first
    const double Ru = -u[id-(N+2)]-u[id+(N+2)]-u[id-1]-u[id+1];
    const double rhs = invD*(f[id]-Ru);
    const double oldu = u[id];
    const double newu = w*rhs + (1.0-w)*oldu;
    local_res2 += (newu-oldu)*(newu-oldu);
    unew[id] = newu;
  }
}

for (int i = 0; i < (N+2)*(local_row_size); ++i){
  u[i] = unew[i];
}

++iter;

// calcualte global residual
MPI_Allreduce(&local_res2, &global_res2, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
```

1.3. **Debugging.** I did not use dbg to debug. I extensively used printf and return 0 to single out chunks of code. I checked correctness of the program on a small problem with N = 2, just as last time.

2. Results on XPS 13 9370

For a residual $< 1e-6$, N = 10. The max error and number of iterations stays the same. Depending on the number of processors, the total time could be either higher or lower than using a single processor. See tables belows for some of the results.

TABLE 1. N = 10, tol = 1e-6

Num. Processors	Iter	Max Error	Time(s)
1	64	0.00137	0.00137
2	64	0.00137	0.000769138
3	64	0.00137	0.003733
4	64	0.00137	0.000908077
5	64	0.00137	0.00152731
6	64	0.00137	0.00202286
7	64	0.00137	0.00161999
8	64	0.00137	0.14174

TABLE 2. N = 100, tol = 1e-6

Num. Processors	Iter	Max Error	Time(s)
1	4395	6.13072e-06	0.88749
2	4395	6.13072e-06	0.635089
3	4395	6.13072e-06	0.650189
4	4395	6.13072e-06	0.567912
5	4395	6.13072e-06	0.609207
6	4395	6.13072e-06	0.589206
7	4395	6.13072e-06	0.580673
8	4395	6.13072e-06	0.650094