A Parallel Iterative Poisson Solver for a Distributed Memory Architecture

Eric Dow
May 11, 2010

Introduction and Motivation

In this project, a parallel iterative Poisson solver is presented and analyzed. The implementation of this solver is tailored for a distributed memory architecture, i.e. one in which each processor has its own private memory. A domain partitioning method based on spectral graph theory is used to distribute the workload among processors. Parallel implementations of two popular iterative solvers, the Jacobi and Gauss-Seidel methods, are presented and compared.

For this project, we consider Poisson’s equation in 2D with homogeneous Dirichlet boundary conditions:

\[-\nabla^2 u = f \quad x \in \Omega\]

\[u = 0 \quad x \in \partial \Omega\]

The function \(f\) is an arbitrary forcing function, and for simplicity we choose to specify \(f = 1\) everywhere on the interior of the domain. Solving Poisson’s equation is a common subproblem in many numerical schemes. A classic example is in the solution of the incompressible Navier-Stokes equations. Most finite difference solvers for the Navier-Stokes equations are based on some form of projection scheme, and require solving a Poisson problem at every time step to determine the pressure field. This is typically the most computationally expensive step of projection schemes. Thus, an efficient Poisson solver is essential.

Typically, the numerical solution of Poisson’s equation requires solving a large sparse system of linear equations. Direct solutions are very efficient for small problem sizes, but for larger problems, especially in higher dimensions, a direct solution becomes impractical. Instead, an iterative scheme is typically used to construct an approximate solution. Unlike direct solvers, iterative solution techniques allow a specified level of accuracy to be achieved. When machine precision is not required, a great deal of computational effort can be spared by using an iterative solver.

Iterative Solvers for Poisson’s Equation

Possibly the most basic method for solving Poisson’s equation is the finite difference method. The solution is represented by discrete values sampled on a regularly spaced grid. Derivatives are discretized via finite difference formulas, and the resulting system of equations is solved for the values of the solution at the grid locations. In order to solve Poisson’s equation, the Laplacian
must be discretized on the grid. A second order accurate discretization of Poisson’s equation in 2D requires a five-point stencil:

\[
\frac{4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}}{h^2} = f_{i,j}
\]

Here, \(h\) is the grid spacing, which is assumed to be constant in both dimensions. It is clear from this stencil that the value of the solution at any particular grid point depends only on the value of its neighbors. This stencil suggests an iterative scheme for solving the resulting system, known as the Jacobi method:

\[
u_{i,j} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} + h^2 f_{i,j})
\]

An initial guess is provided, and the solution at each location is updated using the solution values at the neighboring grid points. This process is repeated until some convergence criteria is met. The Jacobi method saves two copies of the solution, updating the solution \(u_{\text{new}}\) using only the values of \(u_{\text{old}}\). Thus, parallelizing this algorithm is straightforward. Since the values of \(u_{\text{new}}\) do not depend on one another, they can be updated contemporaneously. In practice, the domain is divided into equally sized subdomains, and the values in each subdomain are updated. Each process is responsible for updating some fraction of the solution, and communication is only required at the end of each iteration. Although parallelization is straightforward, this method is rather slow to converge.

The other iterative solver considered in this project is the Gauss-Seidel method. This method converges more quickly than the Jacobi method, and requires less data to be stored. Faster convergence is achieved by using the updated values of the solution as soon as they are available. If the solution at any of the points surrounding grid point \(G\) have already been updated, these updated values are used to update the value at \(G\). There is no longer any distinction between \(u_{\text{old}}\) and \(u_{\text{new}}\), thus only a single copy of the solution must be retained. However, updating these values in parallel is no longer straightforward. For example, assume that Process 1 is responsible for updating grid point \(G\). If grid point \(F\) is adjacent to \(G\) but is updated by Process 2, we would ideally like the value of \(F\) to be sent from Process 2 to Process 1 if \(F\) is updated before \(G\), and vice versa. However, this would require communication between Process 1 and Process 2 before \(G\) is updated to determine whether \(F\) has been updated. Process 1 may be required to wait before continuing, which will in general decrease performance.

One clever solution is to use a red-black ordering of the grid points. The idea is to group the nodes into two disjoint sets, a red set and a black set. The nodes are colored so that any red grid point only has black grid points as its neighbors and vice versa, as in a checkerboard. Below is an example of red-black ordering for a simple rectangular domain.

On more arbitrary domains, a grid point is colored red if the sum of its row and column are odd, and black if the sum is even. The red grid points can be updated using only values from the black set, and vice versa. Most importantly, this operation can be performed in parallel. The disadvantage is that this scheme will require more communication. The same amount of data will need to be sent between processes, but this data must be sent in two separate communications. The first communication occurs once all red values have been updated, and the second once all black values have been updated. Systems in which communication initialization incurs a large overhead will suffer from a loss of performance.
Domain Partitioning

The domain considered for this project is shown below. Homogeneous Dirichlet boundary conditions are enforced along the blue edges. The area inside the triangle is not part of the solution domain.

This geometry was chosen as a representative “nontrivial” domain. Partitioning this domain is complicated by this choice of geometry. Ideally, one hopes to divide the grid points evenly among the various processes in an attempt to balance the workload. Also, one hopes to minimize the “area” that must be shared by any two processes. The values on either side of a cut in the domain must be sent and received during each iteration, and it is therefore desirable to minimize the number of grid points that fall on these boundaries. Spectral partitioning approximately solves the problem of dividing the nodes equally among processes while minimizing the number of grid points that fall on boundaries.

A finite difference grid can be viewed as a graph whose connectivity is defined by the stencil used for approximating derivatives. For the five-point stencil described previously, each grid point is connected to at most four other grid points. Grid points at domain boundaries will be connected
to fewer grid points. The incidence matrix is constructed efficiently by taking advantage of the regularity of the finite difference grid. Since the Laplacian matrix is sparse and symmetric, the Fiedler vector can be calculated very efficiently with the Lanczos algorithm. Once the Fiedler vector is calculated, the domain can be partitioned into two equal subsets. This process can be repeated recursively $n$ times to divide the domain into $2^n$ disjoint sets. Plotted below is the partitioned domain with one and two partitions.

![2 Process Partition](image1.png)  ![4 Process Partition](image2.png)

Figure 3: 2 Process Partition  Figure 4: 4 Process Partition

It is important to note that recursively dividing the domain does not necessarily minimize the number of global edge cuts made. Each division approximately minimizes the number of edge cuts in the subdomain being divided. However, we can not guarantee that this will produce an optimal partition, i.e. a partition that divides the nodes into $2^n$ equal groups with minimal edge cuts.

**Implementation**

Both iterative solvers were implemented in C, and MPI was used for parallelization. Each process was responsible for updating a unique set of grid points. For the Jacobi method, each process updates the values of $u_{\text{new}}$ corresponding to the grid points that it is responsible for. A call to `MPI_Barrier` is made once the process is finished updating the solution. This ensures that every process has finished its iteration before solution values are shared among the various processes.

To share the necessary solution values, `MPI_Sendrecv` was used. Only the required solution values are sent and received between processes. Consider the partition depicted in Figure 2, and assume that Process 1 is responsible for updating the red grid points, and Process 2 is responsible for the blue grid points. At the end of each iteration, it is necessary for Process 1 to send the solution values corresponding to grid points bordering the blue region, and for Process 2 to send the solution values corresponding to grid points bordering the red region. Thus, only the values at these bordering grid points are shared at the end of each iteration. A separate call to `MPI_Sendrecv` is required for every set of shared grid points. In general, $P-1$ calls to `MPI_Sendrecv` are required when using $P$ processes. However, as Figure 4 demonstrates, the spectral partitioning algorithm
used in this project often results in partitions in which each process shares grid points with fewer than \( P - 1 \) other processes. Once all iterations have completed, the results computed by each process are collected at the root process using \texttt{MPI.Gather}.

The Gauss-Seidel method is implemented similarly. The main difference is that the solution values must be shared twice: once after the red nodes have been updated, and again after the black nodes have been updated. The processes must be synchronized before each communication using \texttt{MPI.Barrier} to ensure that the correct values are being shared.

\textbf{Results}

Testing of the iterative solvers was conducted on the Beowulf cluster owned by CSAIL. This cluster is comprised of 16 compute nodes, each with a single 2.4 GHz Intel Xeon Processor with Hyper-threading. The nodes are connected via Gigabit Ethernet. Both solvers were run on 1, 2, and 4 nodes. Two grid sizes, \( 75 \times 75 \) and \( 150 \times 150 \), were used to benchmark performance. The grid sizes refer to the size of the background mesh on which the solution is represented. Below is the solution on the \( 75 \times 75 \) grid after 10000 Jacobi iterations.

![Figure 5: Solution after 10000 Jacobi Iterations](image)

Parallelizing the Jacobi iterative solver was relatively successful. Figures 6 and 7 show the speedup for the \( 75 \times 75 \) grid and the \( 150 \times 150 \) grid, respectively. The dotted red line represents the maximum possible speedup.

It is immediately apparent that performance increase was quite poor for the \( 75 \times 75 \) grid. When 4 processes were used, the runtime was actually greater than for the serial code. This can be explained by the large amount of communication that must be performed at the end of each Jacobi iteration. As seen in Figure 4, the spectral partitioning scheme resulted in a large number of edge cuts. Also, the size of the overlap between subdomains varies greatly. For example, the green and blue subdomains share a much greater number of grid points than the pink and blue subdomains. Since no process can proceed to the next iteration until all other processes have finished sending and receiving the results of the current iteration, this variation in overlap results in a bottleneck.
The performance benefits of parallelization are more apparent in the case of the $150 \times 150$ grid. Doubling the number of grid points in each dimension quadruples the number of grid points in each subdomain. However, the number of grid points shared by any two processes only doubles. The required amount of communication is thus lower relative to the size of the problem solved in each subdomain.

The Gauss-Seidel solver showed similar trends. Figure 8 shows that the parallel solver is slower than the serial solver for both 2 and 4 processes when a $75 \times 75$ grid is used. The $150 \times 150$ grid shows better speedup, but is still slower than the equivalent Jacobi solver. This is a result of the increased amount of communication required by the Gauss-Seidel method.
Conclusions

I was successful in parallelizing both the Jacobi and Gauss-Seidel iterative solvers. The Gauss-Seidel solver is outperformed by the Jacobi solver due to the larger amount of communication that is required during each iteration. However, the decreased memory footprint of the Gauss-Seidel method make it desirable for very large problems. The specifics of the architecture being used and problem size would inform which solver to use.

It is clear from the results presented that parallelization actually leads to a decrease in performance for very small problems. Although only two problem sizes were tested, I believe that the realized speedup would continue to increase for larger problem sizes. This behavior makes a parallel iterative solver a viable choice in practice. As previously mentioned, iterative solvers are typically used for large problems. Since the speedup increases with problem size, parallelization seems appropriate for large problems.
References

Appendix 1: Sample Code

jacobi_ser.c - Serial implementation of Jacobi method

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/types.h>

int ij_to_ind(int i, int j, int N){
    return ( N*(j-1) + i );
}

int *ind_to_ij(int ind, int N, int buf[]){
    buf[0] = ind%N;
    buf[1] = (int) ceil( ( (float) ind ) / ( (float) N ) );
    return buf;
}

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

    clock_t c0, c1;

    // number of gridpoints per dimension
    int N = 75;
    int gridsize = N*N;
    double h = 1/((double)N-1);

    // # of nodes handled by process (set manually)
    int numNodes = 2355;

    double *f = malloc(gridsize * sizeof(double));
    int *Nodes = malloc(numNodes * sizeof(int));

    // read in the forcing vector
    FILE* fp = fopen("f.txt","r");
    char line[256];
    long i = 0;
    while(fgets(line,sizeof line,fp) != NULL){
        f[i] = atof(line);
        i++;
    }
    fclose(fp);

    // read in the nodes
    FILE* fp1 = fopen("Nodes.txt","r");
    i = 0;
while(fgets(line,sizeof line,fp1) != NULL){
    Nodes[i] = atoi(line);
    i++;
}
fclose(fp1);

// initialize solution
double *u_old = malloc(gridsize * sizeof(double));
double *u_new = malloc(gridsize * sizeof(double));
int n;
for (n=0; n < gridsize; n++){
    u_old[n] = 0;
    u_new[n] = 0;
}

c0 = clock();

// do Jacobi Iterations
int n_iter = 10000;
int iter;
for (iter=0; iter < n_iter; iter++){

    // loop through my nodes and update the values
    for (n=0; n < numNodes; n++){

        // figure out the indices of this nodes neighbors
        int buf[2];
        int *ij = ind_to_ij(Nodes[n], N, buf);
        int ip = ij_to_ind(ij[0]+1, ij[1], N) - 1;
        int im = ij_to_ind(ij[0]-1, ij[1], N) - 1;
        int jp = ij_to_ind(ij[0], ij[1]+1, N) - 1;
        int jm = ij_to_ind(ij[0], ij[1]-1, N) - 1;

        // update the solution using u_old values
                                    + h*h*f[ Nodes[n] - 1 ] );
    }

    // copy u_new to u_old
    for (n=0; n < numNodes; n++){
        u_old[Nodes[n] - 1] = u_new[Nodes[n] - 1];
    }
}

} // end of iterations

c1 = clock();
// print how long this took
printf("CPU time: %f\n", (float) (c1-c0)/CLOCKS_PER_SEC);

// write out solution to a file
FILE *rf = fopen("u_ser.txt","wt");
for (n=0; n < gridsize; n++){
    fprintf(rf, "%.10f\n", u_new[n]);
}
fclose(rf);

return 0;
gauss_ser.c - Serial implementation of Gauss-Seidel method

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/types.h>

int ij_to_ind(int i, int j, int N){
    return ( N*(j-1) + i );
}

int *ind_to_ij(int ind, int N, int buf[]){
    buf[0] = ind%N;
    buf[1] = (int) ceil( ( (float) ind ) / ( (float) N ) );
    return buf;
}

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

    clock_t c0, c1;

    // number of gridpoints per dimension
    int N = 75;
    int gridsize = N*N;
    double h = 1/((double)N-1);

    // # of nodes handled by process (set manually)
    int numNodes = 2355;

    double *f = malloc(gridsize * sizeof(double));
    int *Nodes = malloc(numNodes * sizeof(int));

    // read in the forcing vector
    FILE* fp = fopen("f.txt","r");
    char line[256];
    long i = 0;
    while(fgets(line,sizeof line,fp) != NULL){
        f[i] = atof(line);
        i++;
    }
    fclose(fp);

    // read in the nodes
    FILE* fp1 = fopen("Nodes.txt","r");
    i = 0;
    while(fgets(line,sizeof line,fp1) != NULL){
        Nodes[i] = atoi(line);
    }
i++;
}
fclose(fp1);

// initialize solution
double *u = malloc(gridsize * sizeof(double));
int n;
for (n=0; n < gridsize; n++)
{
    u[n] = 0;
}

c0 = clock();

// do Jacobi Iterations
int n_iter = 10000;
int iter;
for (iter=0; iter < n_iter; iter++)
{

    // loop through my nodes and update the values
    for (n=0; n < numNodes; n++)
    {

        // update the even nodes
        if (Nodes[n]%2 == 0)
        {

            // figure out the indices of this nodes neighbors
            int buf[2];
            int *ij = ind_to_ij(Nodes[n], N, buf);
            int ip = ij_to_ind(ij[0]+1, ij[1], N) - 1;
            int im = ij_to_ind(ij[0]-1, ij[1], N) - 1;
            int jp = ij_to_ind(ij[0], ij[1]+1, N) - 1;
            int jm = ij_to_ind(ij[0], ij[1]-1, N) - 1;

            // update the solution using u_old values
                                + h*h*f[ Nodes[n] - 1 ] );
        }
    }

    // loop through my nodes and update the values
    for (n=0; n < numNodes; n++)
    {

        // update the odd nodes
        if (Nodes[n]%2 != 0)
        {

            // figure out the indices of this nodes neighbors
            int buf[2];
            int *ij = ind_to_ij(Nodes[n], N, buf);
int ip = ij_to_ind(ij[0]+1, ij[1], N) - 1;
int im = ij_to_ind(ij[0]-1, ij[1], N) - 1;
int jp = ij_to_ind(ij[0], ij[1]+1, N) - 1;
int jm = ij_to_ind(ij[0], ij[1]-1, N) - 1;

// update the solution using u_old values

} // end of iterations


} // end of iterations


} // end of iterations

c1 = clock();

// print how long this took
printf("CPU time: %f\n", (float) (c1-c0)/CLOCKS_PER_SEC);

// write out solution to a file
FILE *rf = fopen("u_ser_gs75.txt","wt");
for (n=0; n < gridsize; n++){
    fprintf(rf, ".10f\n", u[n]);
}
fclose(rf);

return 0;

}
jacobi_par.c - Parallel implementation of Jacobi method (2 threads)

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

int ij_to_ind(int i, int j, int N){
    return ( N*(j-1) + i );
}

int *ind_to_ij(int ind, int N, int buf[]){
    buf[0] = ind%N;
    buf[1] = (int) ceil( ( (float) ind ) / ( (float) N ) );
    return buf;
}

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

    // number of processes
    int numprocs;
    int pid;

    MPI_Status stat;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&pid);

    // number of gridpoints per dimension
    int N = 75;
    int gridsize = N*N;
    double h = 1/((double)N-1);

    // # of nodes handled by process (set manually)
    int numNodes[2];
    numNodes[0] = 1178;
    numNodes[1] = 1177;
    int maxNodes = 1178; // max # of nodes on any one process

    // # of nodes that need to be sent/received (set manually)
    int numShare[2];
    numShare[0] = 15;
    numShare[1] = 16;

    // tags for data sending/receiving
    int ftag = 1;
    int ntag = 2;
    int stag = 3;
int etag = 4;

double *f = malloc(gridsize * sizeof(double));
int myNodes[numNodes[pid]];
int *Share[numprocs];
int *Nodes[numprocs]; // only process 0 will actually fill this in
double *rbuf;

double t1, t2;

// store the nodes permanently as well
int NodesP[numprocs][maxNodes];

// process 0 reads everything in
if (pid == 0){

    // read in the forcing vector
    FILE* fp = fopen("f.txt","r");
    char line[256];
    long i = 0;
    while(fgets(line,sizeof line,fp) != NULL){
        f[i] = atof(line);
        i++;
    }
    fclose(fp);

    // read in the nodes for each process
    FILE* fp1 = fopen("P1_Nodes.txt","r");
    int Nodes1[numNodes[0]];
    i = 0;
    while(fgets(line,sizeof line,fp1) != NULL){
        Nodes1[i] = atoi(line);
        i++;
    }
    fclose(fp1);

    Nodes[0] = &Nodes1[0];
    for (i=0; i < numNodes[0]; i++){
        NodesP[0][i] = Nodes[0][i];
    }

    FILE* fp2 = fopen("P2_Nodes.txt","r");
    int Nodes2[numNodes[1]];
    i = 0;
    while(fgets(line,sizeof line,fp2) != NULL){
        Nodes2[i] = atoi(line);
        i++;
    }
}

16
fclose(fp2);

    Nodes[1] = &Nodes2[0];
    for (i=0; i < numNodes[1]; i++) {
        NodesP[1][i] = Nodes[1][i];
    }

    // read the nodes that need to be sent/received
    FILE* fp3 = fopen("Nodes_1to2.txt","r");
    int Share1[numShare[0]];
    i = 0;
    while(fgets(line,sizeof line,fp3) != NULL) {
        Share1[i] = atoi(line);
        i++;
    }
    fclose(fp3);

    Share[0] = &Share1[0];

    FILE* fp4 = fopen("Nodes_2to1.txt","r");
    int Share2[numShare[1]];
    i = 0;
    while(fgets(line,sizeof line,fp4) != NULL) {
        Share2[i] = atoi(line);
        i++;
    }
    fclose(fp4);

    Share[1] = &Share2[0];

    // send this info to all the other processes
    for (i=1; i<numprocs; i++) {
        // send the forcing vector
        MPI_Send(&f[0], gridsize, MPI_DOUBLE, i, ftag, MPI_COMM_WORLD);
        // send the nodes that this process will work on
        MPI_Send(&Nodes[i][0], numNodes[i], MPI_INT, i, ntag, MPI_COMM_WORLD);
    }

    // Broadcast the send/receive nodes
    int s;
    for (s=0; s < numprocs; s++) {
        MPI_Bcast(&Share[s][0], numShare[s], MPI_INT, 0, MPI_COMM_WORLD);
    }

    // set the nodes that process 0 will work on
    for (i=0; i < numNodes[0]; i++) {
        myNodes[i] = Nodes[0][i];
    }
if (pid != 0) {
    // receive data from process 0
    MPI_Recv(&f[0], gridsize, MPI_DOUBLE, 0, ftag, MPI_COMM_WORLD, &stat);
    // nodes that this process will work on
    MPI_Recv(&myNodes[0], numNodes[pid], MPI_INT, 0, ntag, MPI_COMM_WORLD, &stat);

    // receive stuff from broadcast
    int s;
    for (s = 0; s < numprocs; s++) {
        Share[s] = malloc(numShare[s] * sizeof(int));
        MPI_Bcast(&Share[s][0], numShare[s], MPI_INT, 0, MPI_COMM_WORLD);
    }
}

// make sure everyone is synched up at this point
MPI_Barrier(MPI_COMM_WORLD);

// initialize solution
double *u_old = malloc(gridsize * sizeof(double));
double *u_new = malloc(gridsize * sizeof(double));
int n;
for (n = 0; n < gridsize; n++) {
    u_old[n] = 0;
    u_new[n] = 0;
}

// start timing
if (pid == 0) {
    t1 = MPI_Wtime();
}

// do Jacobi Iterations
int n_iter = 10000;
int iter;
for (iter = 0; iter < n_iter; iter++) {
    // loop through my nodes and update the values
    for (n = 0; n < numNodes[pid]; n++) {
        // figure out the indices of this nodes neighbors
        int buf[2];
        int *ij = ind_to_ij(myNodes[n], N, buf);
int ip = ij_to_ind(ij[0]+1, ij[1], N) - 1;
int im = ij_to_ind(ij[0]-1, ij[1], N) - 1;
int jp = ij_to_ind(ij[0], ij[1]+1, N) - 1;
int jm = ij_to_ind(ij[0], ij[1]-1, N) - 1;

// update the solution using u_old values
+ h*h*f[ myNodes[n] - 1 ] );
}

// copy u_new to u_old
for (n=0; n < numNodes[pid]; n++){
    u_old[myNodes[n] - 1] = u_new[myNodes[n] - 1];
}

// make sure everyone is synched up at the end of each iteration
MPI_Barrier(MPI_COMM_WORLD);

// collect solution values to be shared
double toSend[numShare[pid]];
for (n=0; n < numShare[pid]; n++){
    // index into solution
    int ind = Share[pid][n] - 1;
    toSend[n] = u_old[ind];
}

int Neighbor[] = {1, 0};
double toRecv[ numShare[Neighbor[pid]] ];

// send and receive
MPI_Sendrecv(&toSend[0], numShare[pid], MPI_DOUBLE, Neighbor[pid], stag,
    &toRecv[0], numShare[Neighbor[pid]], MPI_DOUBLE, Neighbor[pid],
    stag, MPI_COMM_WORLD, &stat);

// insert shared info into solution
for (n=0; n < numShare[Neighbor[pid]]; n++){
    // index in u_old
    int ind = Share[ Neighbor[pid] ][n] - 1;
    u_old[ind] = toRecv[n];
}

} // end of iterations

printf("Process %d done with iterations\n",pid);

// synch up before final send/receive
MPI_Barrier(MPI_COMM_WORLD);
// stop timing
if (pid == 0){
    t2 = MPI_Wtime();
    printf("Elapsed time is %f s\n", t2-t1);
}

// collect solution to be sent
double endSend[ maxNodes ]; // purposely make bigger than needed
for (n=0; n < numNodes[pid]; n++){
    endSend[n] = u_new[ myNodes[n] - 1 ];
}

// allocate space to receive the solution at root
if (pid == 0){
    rbuf = (double *)malloc(numprocs*maxNodes*sizeof(double));
}

// gather results at process 0
MPI_Gather(&endSend[0], maxNodes, MPI_DOUBLE, &rbuf[0], maxNodes, MPI_DOUBLE,
            0, MPI_COMM_WORLD);

// insert results into process 0 solution, write out to a file
if (pid == 0){
    int p;
    for (p=0; p < numprocs; p++){
        // starting index in rbuf for this processes solution
        int st = p*maxNodes;
        for (n=0; n < numNodes[p]; n++){
            // index in global solution array
            int u_ind = NodesP[p][n] - 1;
            u_new[u_ind] = rbuf[st+n];
        }
    }

    // write out solution to a file
    FILE *rf = fopen("u.txt","wt");
    for (n=0; n < gridsize; n++){
        fprintf(rf, ".10f\n", u_new[n]);
    }
    fclose(rf);
}

MPI_Finalize();
return 0;
gauss_par.c - Parallel implementation of Gauss-Seidel method (2 threads)

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

int ij_to_ind(int i, int j, int N){
  return ( N*(j-1) + i );
}

int *ind_to_ij(int ind, int N, int buf[]){
  buf[0] = ind%N;
  buf[1] = (int) ceil( ( (float) ind ) / ( (float) N ) );
  return buf;
}

t main(int argc, char *argv[]){

  // number of processes
  int numprocs;
  int pid;

  MPI_Status stat;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&pid);

  // number of gridpoints per dimension
  int N = 75;
  int gridsize = N*N;
  double h = 1/((double)N-1);

  // # of nodes handled by process (set manually)
  int numNodes[2];
  numNodes[0] = 1178;
  numNodes[1] = 1177;
  int maxNodes = 1178; // max # of nodes on any one process

  // # of nodes that need to be sent/received (set manually)
  int numShare[2];
  numShare[0] = 15;
  numShare[1] = 16;

  // tags for data sending/receiving
  int ftag = 1;
  int ntag = 2;
  int stag = 3;
```

21
int etag = 4;

double *f = malloc(gridsize * sizeof(double));
int myNodes[numNodes[pid]];
int *Share[numprocs];
int *Nodes[numprocs]; // only process 0 will actually fill this in
double *rbuf;

double t1, t2;

// store the nodes permanently as well
int NodesP[numprocs][maxNodes];

// process 0 reads everything in
if (pid == 0){

    // read in the forcing vector
    FILE* fp = fopen("f.txt","r");
    char line[256];
    long i = 0;
    while(fgets(line,sizeof line,fp) != NULL){
        f[i] = atof(line);
        i++;
    }
    fclose(fp);

    // read in the nodes for each process
    FILE* fp1 = fopen("P1_Nodes.txt","r");
    int Nodes1[numNodes[0]];
    i = 0;
    while(fgets(line,sizeof line,fp1) != NULL){
        Nodes1[i] = atoi(line);
        i++;
    }
    fclose(fp1);

    Nodes[0] = &Nodes1[0];
    for (i=0; i < numNodes[0]; i++){
        NodesP[0][i] = Nodes[0][i];
    }

    FILE* fp2 = fopen("P2_Nodes.txt","r");
    int Nodes2[numNodes[1]];
    i = 0;
    while(fgets(line,sizeof line,fp2) != NULL){
        Nodes2[i] = atoi(line);
        i++;
    }
}
fclose(fp2);

Nodes[1] = &Nodes2[0];
for (i=0; i < numNodes[1]; i++) {
    NodesP[1][i] = Nodes[1][i];
}

// read the nodes that need to be sent/received
FILE* fp3 = fopen("Nodes_1to2.txt","r");
int Share1[numShare[0]];
i = 0;
while(fgets(line,sizeof line,fp3) != NULL){
    Share1[i] = atoi(line);
    i++;
}
fclose(fp3);

Share[0] = &Share1[0];

FILE* fp4 = fopen("Nodes_2to1.txt","r");
int Share2[numShare[1]];
i = 0;
while(fgets(line,sizeof line,fp4) != NULL){
    Share2[i] = atoi(line);
    i++;
}
fclose(fp4);

Share[1] = &Share2[0];

// send this info to all the other processes
for (i=1; i<numprocs; i++) {
    // send the forcing vector
    MPI_Send(&f[0], gridsize, MPI_DOUBLE, i, ftag, MPI_COMM_WORLD);
    // send the nodes that this process will work on
    MPI_Send(&Nodes[i][0], numNodes[i], MPI_INT, i, ntag, MPI_COMM_WORLD);
}

// Broadcast the send/receive nodes
int s;
for (s=0; s < numprocs; s++) {
    MPI_Bcast(&Share[s][0], numShare[s], MPI_INT, 0, MPI_COMM_WORLD);
}

// set the nodes that process 0 will work on
for (i=0; i < numNodes[0]; i++) {
    myNodes[i] = Nodes[0][i];
}
if (pid != 0) {

    // receive data from process 0
    MPI_Recv(&f[0], gridsize, MPI_DOUBLE, 0, ftag, MPI_COMM_WORLD, &stat);
    // nodes that this process will work on
    MPI_Recv(&myNodes[0], numNodes[pid], MPI_INT, 0, ntag, MPI_COMM_WORLD, &stat);

    // receive stuff from broadcast
    int s;
    for (s=0; s < numprocs; s++) {
        Share[s] = malloc(numShare[s] * sizeof(int));
        MPI_Bcast(&Share[s][0], numShare[s], MPI_INT, 0, MPI_COMM_WORLD);
    }
}

    // make sure everyone is synched up at this point
    MPI_Barrier(MPI_COMM_WORLD);

    // initialize solution
    double *u = malloc(gridsize * sizeof(double));
    int n;
    for (n=0; n < gridsize; n++) {
        u[n] = 0;
    }

    // start timing
    if (pid == 0) {
        t1 = MPI_Wtime();
    }

    // do Jacobi Iterations
    int n_iter = 10000;
    int iter;
    for (iter=0; iter < n_iter; iter++) {

        // loop through my nodes and update the values
        for (n=0; n < numNodes[pid]; n++) {

            // update the even nodes
            if (myNodes[n]%2 == 0) {

                // figure out the indices of this nodes neighbors
                int buf[2];
int *ij = ind_to_ij(myNodes[n], N, buf);
int ip = ij_to_ind(ij[0]+1, ij[1], N) - 1;
int im = ij_to_ind(ij[0]-1, ij[1], N) - 1;
int jp = ij_to_ind(ij[0], ij[1]+1, N) - 1;
int jm = ij_to_ind(ij[0], ij[1]-1, N) - 1;

// update the solution using u_old values

} // loop through my nodes and update the values
for (n=0; n < numNodes[pid]; n++){

    // update the odd nodes
    if (myNodes[n]%2 != 0){

        // figure out the indices of this nodes neighbors
        int buf[2];
        int *ij = ind_to_ij(myNodes[n], N, buf);

        int ip = ij_to_ind(ij[0]+1, ij[1], N) - 1;
        int im = ij_to_ind(ij[0]-1, ij[1], N) - 1;
        int jp = ij_to_ind(ij[0], ij[1]+1, N) - 1;
        int jm = ij_to_ind(ij[0], ij[1]-1, N) - 1;
    }
} // make sure everyone is synched up at the end of each iteration
MPI_Barrier(MPI_COMM_WORLD);

// collect solution values to be shared
double toSend[numShare[pid]];
for (n=0; n < numShare[pid]; n++){ // index into solution
    int ind = Share[pid][n] - 1;
    toSend[n] = u[ind];
}

int Neighbor[] = {1, 0};
double toRecv[ numShare[Neighbor[pid]] ];

// send and receive
MPI_Ssendrecv(&toSend[0], numShare[pid], MPI_DOUBLE, Neighbor[pid], stag,
        &toRecv[0], numShare[Neighbor[pid]], MPI_DOUBLE, Neighbor[pid],
        stag, MPI_COMM_WORLD, &stat);

// insert shared info into solution
for (n=0; n < numShare[Neighbor[pid]]; n++){ // index in u
    int ind = Share[ Neighbor[pid] ][n] - 1;
    u[ind] = toRecv[n];
}

// loop through my nodes and update the values
for (n=0; n < numNodes[pid]; n++){

    // update the odd nodes
    if (myNodes[n]%2 != 0){

        // figure out the indices of this nodes neighbors
        int buf[2];
        int *ij = ind_to_ij(myNodes[n], N, buf);
int ip = ij_to_ind(ij[0]+1, ij[1], N) - 1;
int im = ij_to_ind(ij[0]-1, ij[1], N) - 1;
int jp = ij_to_ind(ij[0], ij[1]+1, N) - 1;
int jm = ij_to_ind(ij[0], ij[1]-1, N) - 1;

// update the solution using u_old values
    + h*h*f[ myNodes[n] - 1 ] );
}

// make sure everyone is synched up at the end of each iteration
MPI_Barrier(MPI_COMM_WORLD);

// collect solution values to be shared
double toSend1[numShare[pid]];  
for (n=0; n < numShare[pid]; n++){
    // index into solution
    int ind = Share[pid][n] - 1;
    toSend[n] = u[ind];
}

int Neighbor1[] = {1, 0};
double toRecv1[ numShare[Neighbor1[pid]] ];

// send and receive
MPI_Sendrecv(&toSend[0], numShare[pid], MPI_DOUBLE, Neighbor1[pid], stag,
    &toRecv[0], numShare[Neighbor1[pid]], MPI_DOUBLE, Neighbor1[pid],
    stag, MPI_COMM_WORLD, &stat);

// insert shared info into solution
for (n=0; n < numShare[Neighbor1[pid]]; n++){
    // index in u
    int ind = Share[ Neighbor1[pid] ][n] - 1;
    u[ind] = toRecv[n];
}
} // end of iterations

printf("Process %d done with iterations\n",pid);

// synch up before final send/receive
MPI_Barrier(MPI_COMM_WORLD);

// stop timing
if (pid == 0){
    t2 = MPI_Wtime();
printf("Elapsed time is \%f s\n", t2-t1);
}

// collect solution to be sent
double endSend[ maxNodes ]; // purposely make bigger than needed
for (n=0; n < numNodes[pid]; n++){
    endSend[n] = u[ myNodes[n] - 1 ];
}

// allocate space to receive the solution at root
if (pid == 0){
    rbuf = (double *)malloc(numprocs*maxNodes*sizeof(double));
}

// gather results at process 0
MPI_Gather(&endSend[0], maxNodes, MPI_DOUBLE, &rbuf[0], maxNodes, MPI_DOUBLE,
    0, MPI_COMM_WORLD);

// insert results into process 0 solution, write out to a file
if (pid == 0){
    int p;
    for (p=0; p < numprocs; p++){
        // starting index in rbuf for this processes solution
        int st = p*maxNodes;
        for (n=0; n < numNodes[p]; n++){
            // index in global solution array
            int u_ind = NodesP[p][n] - 1;
            u[u_ind] = rbuf[st+n];
        }
    }

    // write out solution to a file
    FILE *rf = fopen("u_par2_gs75.txt","wt");
    for (n=0; n < gridsize; n++){
        fprintf(rf, ".10f\n", u[n]);
    }
    fclose(rf);
}

MPI_Finalize();
return 0;