Domain Decomposition as a Method to Solve the Poisson Problem

Brenton Peck

A thesis presented for the degree of Bachelor of Science in Applied Mathematics

BOISE STATE UNIVERSITY

Department of Mathematics
Boise State University
United States of America

May 1, 2018

Acknowledgments

This thesis would not have been possible without the support of Dr. Donna Calhoun and Scott Aiton who have answered my seemingly innumerous questions with grace. I am especially indebted to my adviser Dr. Grady Wright who has been incredibly supportive of my research and who has constantly given advice and direction in both my academics as-well-as my career goals.

I am grateful to everyone with whom I have have had the pleasure of working with while researching and writing this thesis. Each professor that I have interacted with in the math department at Boise State University has provided me with extensive personal and professional guidance as I finish my degree and begin life as both a graduate and first-time father.

During my time as a student, no one has been more important to me than my family. I would like to thank my parents for their constant love, support, and understanding. Most importantly, I wish to thank my wife, Teri, for her unending love and support and for understanding the amount of time that goes into doing research and completing a degree.

Contents

Chapter 1

Introduction

In high performance computing, one often deals with very large matrices that would be exorbitantly expensive to compute using direct solvers. One method of handling these matrices is to decompose them into smaller matrices that can then be approximated iteratively and in parallel. This method is known as domain decomposition. There are two ways of decomposing a domain into smaller sub-domains, overlapping and non-overlapping as-well-as two methods of evaluating these, node-centered and cell-centered. For this paper we will assume Dirichlet-Dirichlet boundary conditions.

In the first method, we will first use the node centered method and split the domain into smaller sub-domains that overlap each other such that the values at the boundaries between the sub-domains are equivalent. This method leaves the sub-domains coupled at the boundary and each of the sub-domains must communicate values to its adjacent sub-domains. There is some parallelism that is achieved using this method in a staggered fashion; however, the more sub-domains we decompose to, the more wait time there is for the sub-domains furthest from the first sub-domain to be approximated due to each sub-domain having to wait to start being approximated until receiving boundary values from the previous sub-domain. We can parallelize this method further by not communicating values between the sub-domains, but rather making an initial guess at the boundary value and then iterating until a Neumann boundary condition on the interior boundaries has been satisfied to within a given tolerance. This allows all of the sub-domains to be approximated simultaneously instead of in a staggered fashion.

The second method decouples the sub-domains entirely allowing us to achieve full parallelism in solving the sub-domains. This is done by dividing the domain into sub-domains such that they share the boundary or boundaries without overlapping. If we solve for the values on the boundaries, then we can turn the single Dirichlet-Dirichlet boundary problem into an arbitrary number of smaller Dirichlet-Dirichlet problems. These boundary values can be found by solving the Schur-complement system.

Chapter 2

Motivations and Background

I will be examining the application of domain decomposition methods specifically applied to the Poisson problem which is a type of elliptic problem particularly important to the realm of physics. The Poisson equation $\nabla^2 u = f$ involves the Laplacian and a function that only depends on its space variables [1]. This allows us to use a central-difference approximation for a given system.

While there are many applications of the Poisson problem, in my research we are approximating it to model pressure as part of the over-arching goal of modeling wind through mountainous regions. Attempting to model weather without any surface interactions is a challenging problem by itself. When we begin to introduce more complex geography, the problem becomes much more complex. These large problems result in very large matrices which we would like to avoid computing using direct solvers. By decomposing these rather large problems into smaller de-coupled problems, we can achieve massive parallelization that enables us to approximate the original system in a comparably small amount of time.

A further method, which I will not discuss in detail, we utilize in my research to reduce the size of the problem is called adaptive mesh refinement. This is motivated by fluid dynamics, where the frictional force causes the highest rate of change in pressure to occur at boundaries where the fluid interacts with another object. In our situation, this means the the pressure is going to change the most where the wind interacts with the mountain. This means that we can gain a fairly accurate approximation of the pressure at points away from the surface of the mountain using significantly larger interval widths which make for much smaller problems. As we approach the surface of the mountain, however, we need much smaller interval widths to achieve the same level of accuracy. This can be done by starting with a courser mesh size as needed away from the mountain and further sub-dividing up portions of that mesh until we have a fine enough mesh to approximate the problem at the boundary of the mountainside.

Chapter 3

Iterative Methods

The most natural place to start, although not the easiest to motivate, is the 1-dimensional problem. I will show that the equations arising from an overlapping domain decomposition using a second-order central-difference approximation along a 1-dimensional domain interface with a node centered approach ensures that the values shared at the boundary of two sub-domains are equivalent. Subsequently, I will demonstrate that this extends naturally to 2-dimensional problems.

We start by analyzing figure 3.1 which depicts a line and decomposing it into two sub-domains in the subsequent figure 3.2. Nodes u_0 and u_8 are ghost nodes used to solve along the line from u_1 to u_7. Each node can be approximated using the central-difference approximation in equation 3.1.

$$u''(x_j) \cong \frac{1}{h^2}[u_{j-1} - 2u_j + u_{j+1}] \tag{3.1}$$

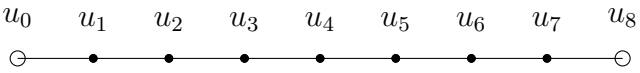

Figure 3.1: 1-D line before domain decomposition.

In figure 3.2, the domain has been decomposed into two parts, Ω_0 and Ω_1, that share the boundary $u_4^0 = u_0^1$. Each of the sub-domains then have two ghost nodes such that on each iteration u_{-1}^1 receives its value from u_3^0 and u_5^0 receives its value from u_1^1.

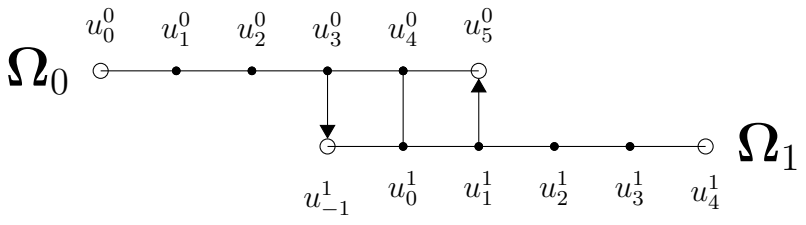

Figure 3.2: 1-D line after domain decomposition.

This decomposition leads to the following equations on the boundary when solv-

ing the Poisson problem where f is the exact solution at each point.

$$u_3^0 - 2u_4^0 + u_5^0 = f_4^0$$
$$u_{-1}^1 - 2u_0^1 + u_1^1 = f_0^1$$

From these equations we know that $f_4^0 = f_1^1$ and since at each iteration u_{-1}^1 receives its value from u_3^0 and u_5^0 receives its value from u_1^1 we see that the equations become $u_3^0 - 2u_4^0 + u_1^1 = f_4^0 = u_3^0 - 2u_0^1 + u_1^1$ and thus $u_4^0 = u_0^1$.

Now we will attempt the 2-D case. We start by making a box and decomposing it into two sub-domains as seen in the subsequent figures. I have adopted the notation of u_{xy} where x is the x-coordinate and y is the y-coordinate. Nodes along $x = 0$ and 4 as-well-as nodes along $y = 0$ and 8 are ghost nodes. We can approximate each node using the central-difference approximation

$$u''(x_j) \cong \frac{1}{h^2}[u_{i-1,j} + u_{i+1,j} - 4u_{i,j} + u_{i,j-1} + u_{i,j+1}] \tag{3.2}$$

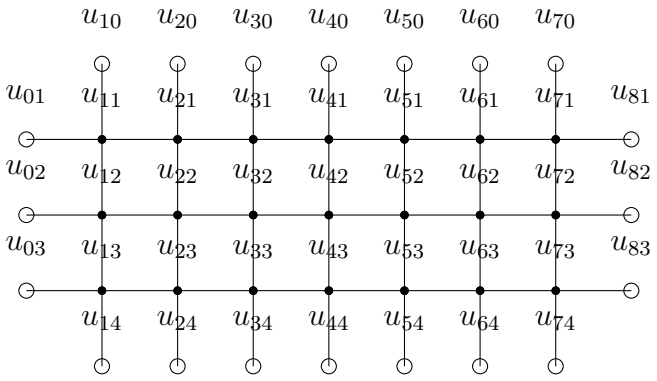

Figure 3.3: 2-D box before domain decomposition.

Similar to the 1-D case, the domain has been decomposed into two parts, Ω_0 and Ω_1, that share the boundary $u_{4y}^0 = u_{0y}^1$. Each of the sub-domains then have ghost nodes such that for each j, $u_{-1,j}^1$ receives its value from $u_{3,j}^0$ and $u_{5,j}^0$ receives its value from $u_{1,j}^1$ every iteration.

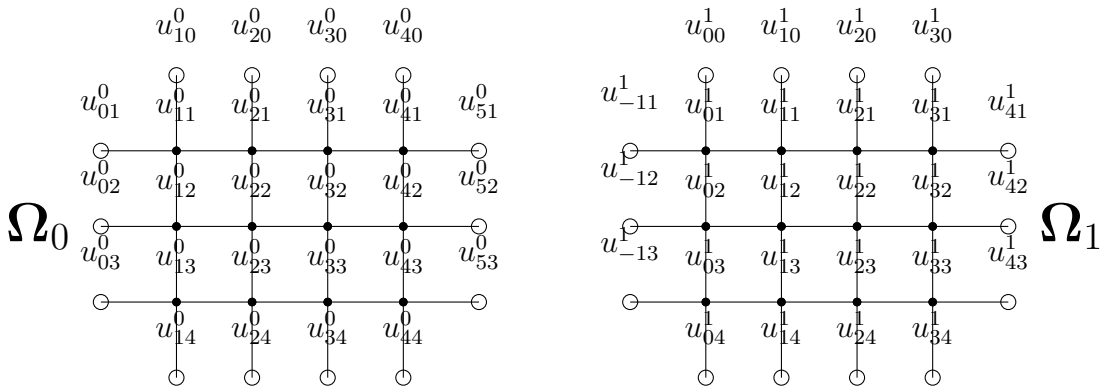

Figure 3.4: 2-D box after domain decomposition.

The resulting decomposition yields the following equations on the boundary for the Poisson problem.

$$\begin{aligned}
u^0_{31} + u^0_{51} + u^0_{40} + u^0_{42} - 4u^0_{41} &= f^0_{41} \\
u^1_{-11} + u^1_{11} + u^1_{00} + u^1_{02} - 4u^1_{01} &= f^1_{01} \\
u^0_{32} + u^0_{52} + u^0_{41} + u^0_{43} - 4u^0_{42} &= f^0_{42} \\
u^1_{-12} + u^1_{12} + u^1_{01} + u^1_{03} - 4u^1_{02} &= f^1_{02} \\
u^0_{33} + u^0_{53} + u^0_{42} + u^0_{44} - 4u^0_{43} &= f^0_{43} \\
u^1_{-13} + u^1_{13} + u^1_{02} + u^1_{04} - 4u^1_{03} &= f^1_{03}
\end{aligned}$$

From these equations we know that for each j, $f^0_{4,j} = f^1_{0,j}$ and since at each iteration $u^1_{-1,j}$ receives its value from $u^0_{3,j}$ and $u^0_{5,j}$ receives its value from $u^1_{1,j}$ we see that after reducing, the equations become:

$$\begin{aligned}
u^0_{40} + u^0_{42} - 4u^0_{41} &= u^1_{00} + u^1_{02} - 4u^1_{01} \\
u^0_{41} + u^0_{43} - 4u^0_{42} &= u^1_{01} + u^1_{03} - 4u^1_{02} \\
u^0_{42} + u^0_{44} - 4u^0_{43} &= u^1_{02} + u^1_{04} - 4u^1_{03}
\end{aligned}$$

Furthermore, since we have boundary conditions, the ghost nodes on the boundary are known to be equivalent. That is, $u^0_{40} = u^1_{00}$ and $u^0_{44} = u^1_{04}$ which gives the following system of equations.

$$u^0_{42} - 4u^0_{41} = u^1_{02} - 4u^1_{01} \tag{3.3}$$

$$u^0_{41} + u^0_{43} - 4u^0_{42} = u^1_{01} + u^1_{03} - 4u^1_{02} \tag{3.4}$$

$$u^0_{42} + 4u^0_{43} = u^1_{02} + 4u^1_{03} \tag{3.5}$$

Now if we both solve 3.3 and 3.5 for u^1_{02} and u^0_{42} and plug into 3.4, as-well-as subtract 3.5 from 3.3 we get

$$\begin{aligned}
u^0_{41} + u^0_{43} &= u^1_{01} + u^1_{03} \\
-u^0_{41} + u^0_{43} &= -u^1_{01} + u^1_{03}
\end{aligned}$$

Adding and subtracting these equations together and then plugging into 3.4 we arrive at the final result we are looking for.

$$\begin{aligned}
u^0_{41} &= u^1_{01} \\
u^0_{42} &= u^1_{02} \\
u^0_{43} &= u^1_{03}
\end{aligned}$$

For the sake of thoroughness, we will extend this to 4 sub-domains in 4 quadrants as shown in the next two figures. Similar to our proof that the boundary values between Ω_0 and Ω_1 are equivalent, it can be shown that the boundary values between Ω_0 and Ω_2 are equivalent as are those between Ω_1 and Ω_3. What remains then is to show that the boundary value shared by all 4 nodes is equivalent for each; that is, $u^0_{43} = u^1_{03} = u^2_{40} = u^3_{00}$.

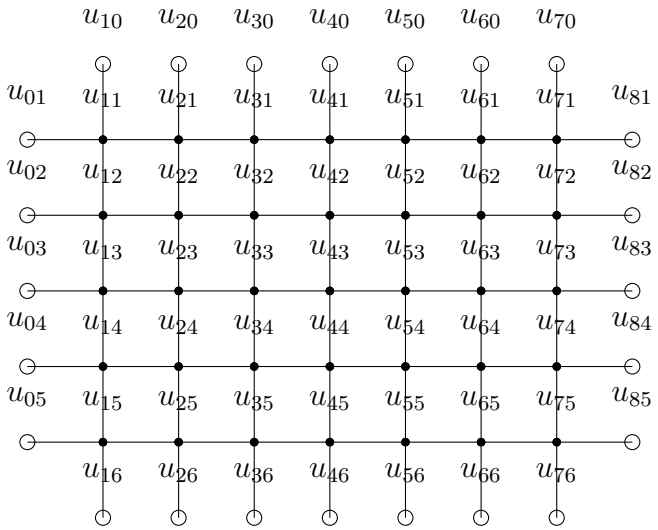

Figure 3.5: 2-D box before domain decomposition.

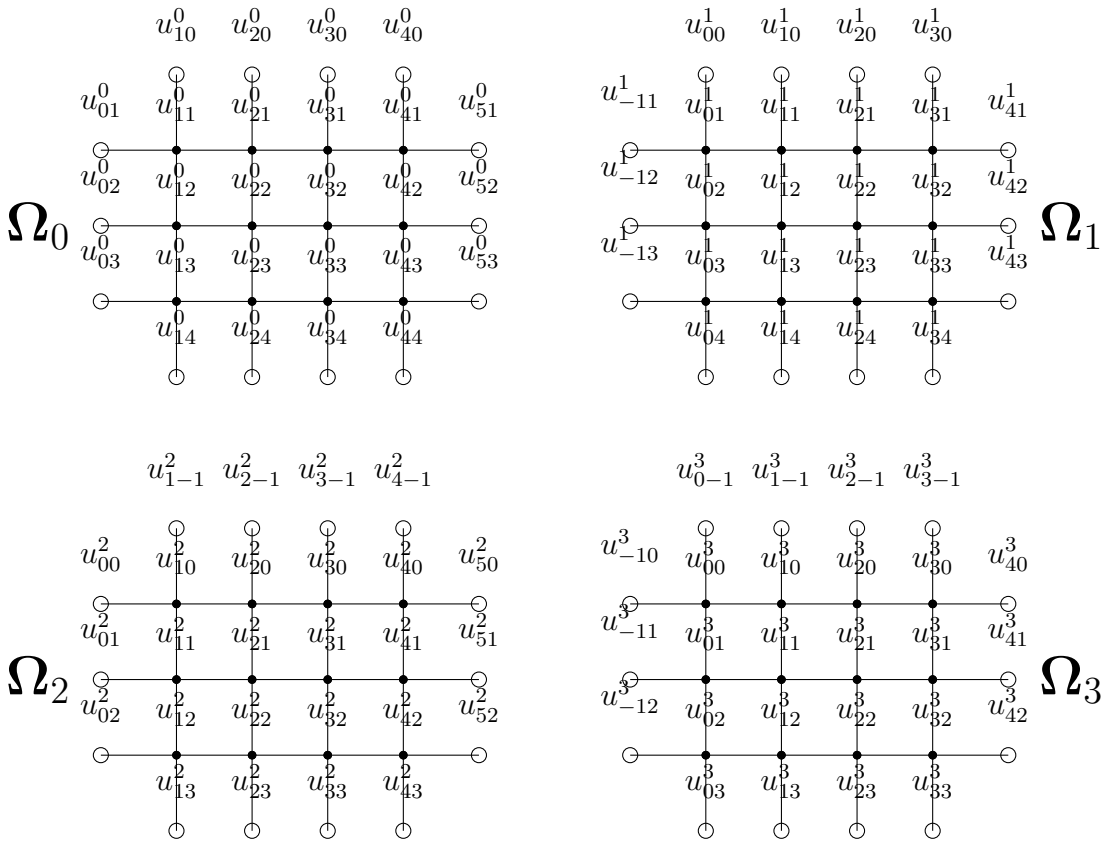

Figure 3.6: 2-D box before domain decomposition.

Taking the values of the ghost nodes from their adjacent sub-domains and knowing that the function is equivalent on $f^0_{43}, f^1_{03}, f^2_{40}$, and f^3_{00}, we obtain the following system of equations.

$$\begin{aligned} & u^0_{42} + u^2_{41} + u^0_{33} + u^1_{13} - 4u^0_{43} \\ = {} & u^1_{02} + u^3_{01} + u^0_{33} + u^1_{13} - 4u^1_{03} \\ = {} & u^0_{42} + u^2_{41} + u^2_{30} + u^3_{10} - 4u^2_{40} \\ = {} & u^1_{02} + u^3_{01} + u^2_{30} + u^3_{10} - 4u^3_{00} \end{aligned}$$

Similarly, we know that ghost nodes sharing a boundary have the same value as shown below.

$$\begin{aligned} u^0_{53} = {} & u^2_{50} = u^1_{13} = u^3_{10} \\ u^0_{44} = {} & u^1_{04} = u^2_{41} = u^3_{01} \\ u^2_{4-1} = {} & u^3_{0-1} = u^0_{42} = u^1_{02} \\ u^1_{-13} = {} & u^3_{-10} = u^0_{33} = u^2_{30} \end{aligned}$$

Plugging these in, we get that $u^0_{43} = u^1_{03} = u^2_{40} = u^3_{00}$ which is what we expected.

Chapter 4

Motivating Further Methods

Using an overlapping domain method works well for some situations; however, there are situations that arise in which we would like to de-couple the sub-domains entirely. To motivate this method I will switch to a cell-centered approach. Starting in 1-D again we can visualize this using figure 4.1 which depicts a straight line with nodes at the center of each cell. We use the same central difference approximation we showed in equation 3.1. The system boundaries are indicated at $u|_{x=0}$ and $u|_{x=1}$. For the non-overlapping case we define the boundary between two sub-domains as Γ. The value of u on the boundary is then γ, so I have indicated the interior boundary Γ at $u|_\Gamma = \gamma$.

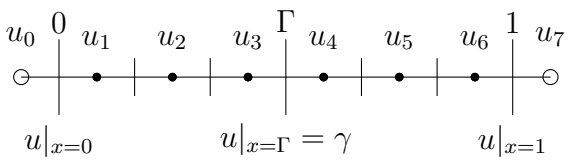

Figure 4.1: 1-D line before domain decomposition.

We can define γ to be the average of its neighbor nodes as shown in equation 4.1.

$$\gamma = \frac{u_j + u_{j+1}}{2} \tag{4.1}$$

In figure 4.2 we have decomposed the domain into two sub-domains, Ω_0 and Ω_1. Unlike the node-centered case, the two sub-domains do not share any nodes or receive any values for ghost nodes. Instead we enforce equation 4.1 as a Dirichlet boundary condition between the two sub-domains.

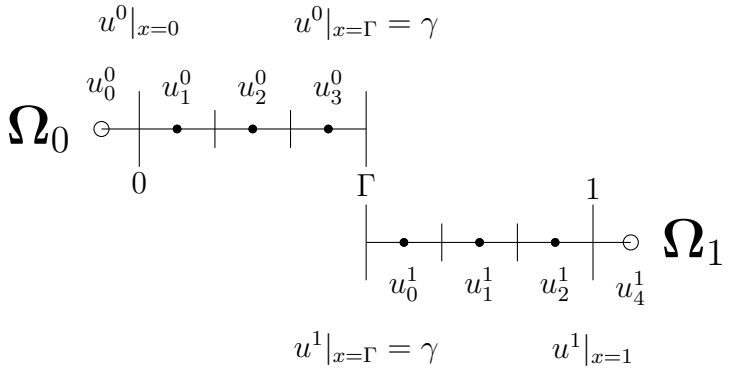

Figure 4.2: 1-D line after domain decomposition.

The matrix system using the 3-point stencil before decomposition is shown below.

$$\begin{bmatrix} -3 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & & & \ddots & & & \\ & & & & 1 & -2 & 1 \\ & & & & & 1 & -3 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix}$$

From figure 4.2 we can see that on the boundary we have the following system of equations.

$$f_n^0 = u_{n-1}^0 - 2u_n^0 + u_0^1 \tag{4.2}$$
$$f_0^1 = \quad u_n^0 - 2u_0^1 + u_1^1 \tag{4.3}$$

$$\gamma = \frac{u_n^0 + u_0^1}{2} \tag{4.4}$$

Solving 4.4 for u_0^1 and u_n^0 and plugging into 4.2 and 4.3 respectively we obtain the system of equations below. These let us construct the subsequent decomposed matrix.

$$f_n^0 = u_{n-1}^0 - 3u_n^0 + 2\gamma$$
$$f_0^1 = \quad u_1^1 - 3u_0^1 + 2\gamma$$

$$\left[\begin{array}{ccccc|ccccc|c} -3 & 1 & & & & & & & & & 0 \\ 1 & -2 & 1 & & & & & & & & 0 \\ & & \ddots & & & & & & & & 0 \\ & & 2 & -2 & 1 & & & & & & 0 \\ & & & 1 & -3 & & & & & & 2 \\ \hline & & & & & -3 & 1 & & & & 2 \\ & & & & & 1 & -2 & 1 & & & 0 \\ & & & & & & & \ddots & & & 0 \\ & & & & & & & 1 & -2 & 1 & 0 \\ & & & & & & & & 1 & -3 & 0 \\ \hline & & & & 2 & 2 & & & & & -4 \end{array}\right] \left[\begin{array}{c} u_1^0 \\ \\ \vdots \\ \\ u_n^0 \\ \hline u_1^1 \\ \\ \vdots \\ \\ u_n^1 \\ \hline \gamma \end{array}\right] = \left[\begin{array}{c} f_1^0 \\ \\ \vdots \\ \\ f_n^0 \\ \hline f_1^1 \\ \\ \vdots \\ \\ f_n^1 \\ \hline 0 \end{array}\right]$$

This shows that if we know the γ values, we can completely decouple the system into smaller systems that we can solve much faster and in parallel.

Chapter 5

Conclusion

Appendix A

Code

Listing A.1: Very basic direct solver in Matlab for the 1-D Poisson Problem.

```
function [x,err] = PoisSolver(f,a,b,n)

h = (b-a)/n;
j = 1:n;
xj = a + (j-0.5).*h;
A = diag([-3,-2*ones(1,n-2),-3]) + diag(ones(1,n-1),1) ...
    + diag(ones(1,n-1),-1);
A
B = h.^2*f(xj);
B = B'
x = (A\B);
x
g = @(xj) sin(4*pi*xj);
err = g(xj)'-x;

end
```

Listing A.2: Jacobi node-centered 1-D solver in C for the Poisson Problem using Method 1.

```
#include "hmwk3.h"
#include <demo_util.h>
#include <mpi.h>
#include <math.h>
#include <stdio.h>

// Define the function
double foo(double x){
    return -1*(2*M_PI)*(2*M_PI)*cos(2*M_PI*x);
}

double exact(double x){
    return cos(2*M_PI*x);
}
```

```
int main(int argc, char** argv){
    // Initialize MPI
    int rank, nprocs;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    // Initialize global variables
    int itermax = 5; // quick break for when itermax is not
       ↪ specified
    int p, M, N, err;
    double tol;

    // Read command line
    read_int(argc, argv, "-n", &N, &err);
    read_int(argc, argv, "--itermax", &itermax, &err);
    read_double(argc, argv, "--tol", &tol, &err);

    // Set intervals
    M=N/nprocs;
    double sintvl = 1.0/nprocs;
    double a = sintvl*rank;
    double h = 1.0/N;

    // Initialize vectors
    double u[M];
    double prevu[M];
    double f[M];
    int i;
    for(i=1; i<M; i++){
        u[i] = 0;
        f[i] = h*h*foo(a+i*h);
    }

    // Initialize Boundary Conditions
    if(rank == 0){
        u[0] = 1;
    }
    if(rank == nprocs-1){
        u[M] = 1;
    }
    // Iterate over Jocobi Iterations
    int iter;
    double diff, largest_diff, ri, world_diff;
    for(iter=1; iter<itermax; iter++){
        largest_diff = 0;
```

```
        for(i=0; i<M+1; i++){
            prevu[i] = u[i];
        }
        //perform a Jacobi iteration, keeping track of
           ↪ largest difference
        for(i=1; i<M; i++){
            u[i] = -0.5*(f[i] - prevu[i-1]-prevu[i+1]);
            diff = fabs(u[i] - prevu[i]);
            if(diff>largest_diff){largest_diff = diff;}
        }
        //find largest difference and break if below tol
        MPI_Allreduce(&largest_diff, &world_diff, 1,
           ↪ MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
        if(world_diff<tol){break;}

        //Synchronize end conditions
        MPI_Request left_request,right_request;
        if(rank!=0){
            //send left boundary
            MPI_Isend(&(u[1]), 1, MPI_DOUBLE, rank-1, 0,
               ↪ MPI_COMM_WORLD,&left_request);
            //receive right boundary
            MPI_Recv(&(u[0]), 1, MPI_DOUBLE, rank-1, 0,
               ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }else{
           u[0] = 1;
        }
        if(rank != nprocs-1){
            //send right boundary
            MPI_Isend(&(u[M-1]), 1, MPI_DOUBLE, rank+1, 0,
               ↪ MPI_COMM_WORLD,&right_request);
            //recieve left boundary
            MPI_Recv(&(u[M]), 1, MPI_DOUBLE, rank+1, 0,
               ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }else{
          u[M] = 1;
        }

    }
    double max_error=0;
    for(i=1;i<M;i++){
        double error = fabs(u[i]-exact(a+i*h));
        if(error>max_error)
            max_error=error;
    }
    double world_error;
    MPI_Allreduce(&max_error, &world_error, 1, MPI_DOUBLE,
       ↪ MPI_MAX, MPI_COMM_WORLD);
```

```
    if(rank==0){
        for(i=0; i<M+1; i++){
            printf("%.19g\n",u[i]);
        }
        printf("%d\n",iter);
        printf("%.19g\n",world_diff);
        printf("%.19g\n",world_error);
    }

    MPI_Finalize();
return 0;
}
```

Bibliography

[1] Stanley J. Farlow. *Partial Differential Equations for Scientists and Engineers.* Dover, 2015.