Parallel Finite Difference Method for the Poisson Problem

Hikaru N. Belzer

December 6, 2024

Abstract

This report explores the numerical solution of the Poisson problem on a unit square domain using the Finite Difference Method (FDM) and parallel computing. By implementing the solution in C, both serial and parallel computations can be run to approximate the answer and measure performance against varying data sizes. The implementation uses Message-Passing Interface (MPI) to distribute the computation, specifically through blocking and non-blocking communication, which leads to improved efficiency. Performance comparisons reveal significant speedups for the parallel C code. The non-blocking case and blocking case gave very similar results, however, the non-blocking implementation showed improvement, particularly at larger grid sizes, making it the most scalable and efficient approach.

Table of Contents

1 Introduction

Problem Statement

The primary goal of this report is to approximate the elliptic test problem of the Poisson equation on the unit square domain $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$ and perform simulations for moderately fine mesh resolutions, such that they are large enough to fully test numerical convergence.

Poisson Equation and the Finite Difference Method

The Poisson Equation is given by

$$\begin{aligned} -\Delta u &= f(x,y) \text{ for all } (x,y) \in \Omega = (0,1) \times (0,1) \subset \mathbb{R}^2 \\ u &= 0 \text{ for all } (x,y) \in \partial\Omega \end{aligned} \tag{1}$$

The Laplace operator is defined as

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{2}$$

We will check $x_0 = y_0 = 0$ and $x_{N+1} = y_{N+1} = 1$ on the boundary.

Additionally, the function f is given by

$$f(x,y) = -2\pi^2 \text{cos}(2\pi x)\text{sin}^2(\pi y) - 2\pi^2 \text{sin}^2(\pi x)\text{cos}(2\pi y) \tag{3}$$

A closed-form solution as the true solution takes the form

$$u(x,y) = \text{sin}^2(\pi x)\text{sin}^2(\pi y) \tag{4}$$

We are aware that it is important to compute the finite difference error to confirm the convergence of the Finite Difference Method. The finite difference error is defined as the difference between the true solution and the numerical solution that is defined on the mesh points. We know that for sufficiently small h, we can expect that the ratio of errors on consecutively refined meshes will behave like

$$\text{Ratio} = \frac{||u - u_{2h}||}{||u - u_h||} \approx \frac{C(2h)^2}{Ch^2} = 4 \tag{5}$$

Our code implements a matrix-vector product:

$$\vec{v}_{ij} = -\vec{u}_{ij-1} - \vec{u}_{i-1j} + 4\vec{u}_{ij} - \vec{u}_{i+1j} - \vec{u}_{ij+1} \tag{6}$$

Furthermore, in Tables 3.1.1 and 3.1.2 in the Results section, we will provide a column for the computed ratio values. Lastly, the correct norm for the theory of finite differences is the $L^\infty\Omega$ function norm, which is defined by

$$||u - u_h||_{L^\infty(\Omega)} = sup_{(x,y)\in\Omega}|u(x,y) - u_h(x,y)| \tag{7}$$

We compiled the results with blocking and non-blocking MPI to determine that the non-blocking MPI cases are faster at higher nodes, although the overall runtimes were similar.

2 Methods and Implementation

2.1 Serial C: Code Improvement

We will use the Finite Difference Method (FDM) to approximate $u(x, y)$ on a mesh $\Omega_h = \{(x_i, y_i), i, j = 0, ..., N+1\}$ where $x_i = ih, i = 0, ..., N+1, y_i = jh, j = 0, ..., N+1$ with $h = \frac{1}{(N+1)}$.

We will define the finite difference approximation $u_{ij} \approx u(x_i, y_i)$ for all (i, j):

$$-u_{i-1,j} - u_{i,j-1} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1} = h^2 f_{i,j} \quad i, j = 1, ..., N \tag{8}$$

Additionally, we will make use of the CG method, which is implemented in the cg.c file provided in the supplied code from [1]. In our C implementation, we are computing a matrix-vector product, Ax, which makes use of a parallel_dot function. We edited the utilities.c, Ax.c, and main.c files as follows...

2.1.1 Programming serial_dot and parallel_dot

Inside of the utilities.c file, we edited the serial_dot, parallel_dot, and parallel_norm2 functions. First, here is serial_dot:

```
double serial_dot (double *x, double *y, long n)
{
  double dp;
#ifndef BLAS
  long i;
#endif

#ifdef BLAS
  dp = cblas_ddot(n, x, 1, y, 1);
#else
  dp = 0.0;
  // #pragma omp parallel for reduction(+:dp)
  for(i = 0; i < n; i++)
    dp += x[i]* y[i];
#endif

  return dp;
}
```

In this function, we implemented BLAS, which will utilize "long i" and cblas_ddot. The else-condition is reserved for when BLAS is not used and makes use of a traditional for-loop to compute the product.

Next, here is parallel_dot:

```
  double parallel_dot(double *l_x, double *l_y, long l_n, MPI_Comm comm) {

    double l_dot=0.0;
    double dot=0.0;

    l_dot = serial_dot(l_x, l_y, l_n);

  #ifdef PARALLEL
    MPI_Allreduce(&l_dot, &dot, 1, MPI_DOUBLE, MPI_SUM, comm);
  #else
    dot = l_dot;
  #endif

    return dot;
  }
```

In parallel_dot, we initialize new variables, l_dot and dot, to 0.0. Then, we call the serial dot product. If we are running the code in parallel, we use the MPI_Allreduce command to handle the tasks.

Next, here is the parallel_norm2 function:

```
  double parallel_norm2 (double *l_x, long l_n, MPI_Comm comm) {
    return sqrt(parallel_dot(l_x,l_x,l_n,comm));
  }
```

This function simply calls the parallel_dot function and returns the square root of the result.

2.1.2 Implementing the Finite Difference Method

The Ax.c file contains a parallel computation function which uses finite difference methods:

```
for(l_j = 0; l_j < l_N; l_j++) {
  for (i = 0; i < N; i++) {
                   tmp = 4.0 * l_u[i   + N * l_j    ];
    if (l_j > 0)   tmp = tmp - l_u[i   + N * (l_j-1)];
    if (i > 0)     tmp = tmp - l_u[i-1 + N * l_j    ];
    if (i < N-1)   tmp = tmp - l_u[i+1 + N * l_j    ];
    if (l_j < N-1) tmp = tmp - l_u[i   + N * (l_j+1)];
          l_v[i+N*l_j] = tmp;
  }
}
```

The for-loops compute the new values of the vector l_v by applying a finite difference operator to the input vector l_u. The outer loop iterates over the local grid in the vertical direction, and the inner loop iterates over the grid in the horizontal direction. For each grid point, (i, l_j), the function calculates the value of l_v[i + N * l_j], which is the result of applying a finite difference approximation to the current value of l_u[i + N * l_j] and its neighboring values (notice the l_j-1 and l_j+1 values).

Then, in the calculation for each point, (i, l_j), the central value l_u[i + N * l_j] is weighted by a factor of 4, and then adjustments are made based on the values of its neighboring points. By following the shown if-statements, the result is a new value for l_v[i + N * l_j].

2.1.3 Implementing enorminf and Additional Changes

In addition to these files, we have a cg.c file, which is called using these lines in the main.c file:

```
     MPI_Barrier(MPI_COMM_WORLD);
     start = MPI_Wtime();  /* start time */

     cg(l_u, &flag, &relres, &iter, l_r, tol, maxit,
        l_p, l_q, l_n, l_N, N, id, idleft, idright, np, comm, gl, gr);

    MPI_Barrier(MPI_COMM_WORLD);
    end = MPI_Wtime();  /* end time */
```

Directly underneath this in main.c, we implemented enorminf:

```
  // Layout from 5Poisson_HW5a.pdf
  double utrue;
  l_enorminf = 0.0;

  for (l_j = 0; l_j < l_N; l_j++) {
    for (i = 0; i < N; i++) {
      j = l_j + l_ia;
      utrue = pow(sin(M_PI*x[i]), 2)
          * pow(sin(M_PI*y[j]), 2);
      err_ij = fabs(l_u[i+l_j*N] - utrue);
      if (err_ij > l_enorminf)
        l_enorminf = err_ij;
    }
  }
  MPI_Reduce(&l_enorminf, &enorminf, 1, MPI_DOUBLE,
     MPI_MAX, 0, MPI_COMM_WORLD);
```

This part of the code calculates the error between the values of a solution vector (l_u) and a reference solution (utrue), and then finds the maximum error across all processes. The outer loop iterates over the local grid points in the vertical direction, and the inner loop iterates over the grid points in the horizontal direction. For each local grid point, (i, l_j), the code calculates utrue based on the sine functions of the corresponding x[i] and y[j] coordinates, where x[i] corresponds to the horizontal grid point, and y[j] corresponds to the vertical grid point. The true solution is compared to the current approximation l_u[i + l_j * N] at that point, and then the error, err_ij, is calculated. If the error at the current point exceeds the previously recorded max error, l_enorminf, it updates l_enorminf with the new value.

2.2 Parallel C: Code Improvement

2.2.1 Implementing MPI_Send/MPI_Recv

In order to successfully convert the code to parallel, we had to implement MPI_Send and MPI_Recv in a way that would not cause "blocking" or "deadlock." This was accomplished by splitting the processes by whether they were odd or even numbered, and choosing to receive communication first if they were even and send communication first if they were odd. The implementation is shown below:

```
if (id%2 == 0) {
    MPI_Recv(gl,               N, MPI_DOUBLE, idleft,  1, comm, &status);
    MPI_Recv(gr,               N, MPI_DOUBLE, idright, 2, comm, &status);
    MPI_Send(&(l_u[N*l_N - N]), N, MPI_DOUBLE, idright, 1, comm         );
    MPI_Send(&(l_u[    0    ]), N, MPI_DOUBLE, idleft,  2, comm         );
} else {
    MPI_Send(&(l_u[N*l_N - N]), N, MPI_DOUBLE, idright, 1, comm         );
    MPI_Send(&(l_u[    0    ]), N, MPI_DOUBLE, idleft,  2, comm         );
    MPI_Recv(gl,               N, MPI_DOUBLE, idleft,  1, comm, &status);
    MPI_Recv(gr,               N, MPI_DOUBLE, idright, 2, comm, &status);
}
```

2.2.2 Handling Top and Bottom Boundaries

In order to handle the top and bottom boundaries of the sub-matrix, we implemented a series of code in Ax.c that computes the operation at the bottom boundary of the subdomain by initially setting l_j=0, which refers to the first row of the local portion of the matrix l_u. This block relies on the received gl data from the left neighboring process in order to finish the calculation. Similarly, we handle the top boundary by initially setting l_j=l_N-1, which refers to the last row of the local portion of the matrix l_u. This block relies on the received gr data from the right neighboring process to complete the computation.

```
l_j = 0;
for (i = 0; i < N; i++) { //Block B
                tmp = 4.0 * l_u[i     + N*  l_j     ];
  if (id  >      0)  tmp -=   gl[i];
  if (i   >      0)  tmp -=  l_u[i-1   + N*  l_j     ];
  if (i   <    N-1)  tmp -=  l_u[i+1   + N*  l_j     ];
  if (l_j < l_N-1)  tmp -=  l_u[i     + N* (l_j + 1)];
  l_v[i + N * l_j] = tmp;
}

l_j = l_N - 1;
for (i = 0; i < N; i++) { //Block C
                  tmp = 4.0 * l_u[i   + N*  l_j   ];
    if (l_j >     0 )  tmp -= l_u[i   + N* (l_j-1)];
    if (i   >     0 )  tmp -= l_u[i-1 + N*  l_j   ];
    if (i   <   N-1 )  tmp -= l_u[i+1 + N*  l_j   ];
    if (id  <   np-1)  tmp -= gr[i];
    l_v[i + N* l_j] = tmp;
}
```

2.3 Parallel C: Using Non-Blocking MPI

2.3.1 Implementing MPI_Isend/MPI_Irecv

In order to optimize the program for parallel execution, we removed the blocking calls (MPI_Send and MPI_Recv), which caused each process to wait for its send or receive operation to complete before moving on, potentially leading to delays and inefficiencies if some processes finished faster than others. In the updated code, the communication uses non-blocking calls: MPI_Irecv for receiving data into gl and gr, and MPI_Isend for sending data from the boundary elements. This allows communication to occur in the background while computations proceed, potentially reducing wait times.

The changes we made include changing "status" to "statuses[4]" and changing "request" to "requests[4]." We use these to manage non-blocking communication between the processes in our parallel implementation. The requests array tracks two non-blocking receive operations (MPI_Irecv) and two non-blocking send operations (MPI_Isend). Note that we are also using the "&" to access the memory addresses:

```
MPI_Status  statuses[4];
MPI_Request requests[4];

MPI_Irecv(gl            ,N, MPI_DOUBLE, idleft,  1, comm, &(requests[0]));
MPI_Irecv(gr            ,N, MPI_DOUBLE, idright, 2, comm, &(requests[1]));
MPI_Isend(&(l_u[N*l_N-N]),N, MPI_DOUBLE, idright, 1, comm, &(requests[2]));
MPI_Isend(&(l_u[   0   ]),N, MPI_DOUBLE, idleft,  2, comm, &(requests[3]));
```

Aside from this, we added this MPI_Waitall line after Block A to ensure that all communication operations are completed before proceeding:

```
MPI_Waitall(4, requests, statuses);
```

Additionally, we removed Block D from the previous implementation to use these new non-blocking MPI lines.

3 Results

Machine Description

For our experiment, we are modifying N, the tolerance (tol), the maximum number of iterations (iter), the number of nodes, and the number of processes per node. We are using HPCF 2018, which includes both the high_mem and develop partitions. There are 50 compute nodes that total to 1800 cores and more than 19 TB of pooled memory. Specifically, the 50 compute nodes are bdnode[001-008] and cnode[002-029,031-044], and the 2 develop nodes are cnode[001,030], each with two 18-core Intel Xeon Gold 6140 Skylake CPUs (2.3 GHz clock speed, 24.75 MB L3 cache, 6 memory channels, 140 W power), giving a total of 36 cores per node.

Additionally, each node has 384 GB of memory (12 x 32 GB DDR4 at 2666 MT/s), giving a total of 10.6 GB per core and also a 120 GB SSD disk. In terms of the network, the nodes are connected by a network of four 36-port EDR (Enhanced Data Rate) InfiniBand switches (100 Gb/s bandwidth, 90 ns latency).

3.1 Convergence of the Finite Difference Method

We went to source [4] and downloaded driver_cg.m. Then, inside of the MATLAB code, we typed "driver_cg(32);" to test N = 32 and so forth for N = 32, 64, 128, 256, 512, 1024, 2048, and 4096. Then, we assembled the results in a table:

Table 3.1.1: MATLAB Code Results

N	DOF	$\|\|u - u_h\|\|$	Ratio	`#iter`	wall clock time	
					HH:MM:SS	seconds
32	1024	3.0128e-03	3.2809	48	00:00:00	0.01
64	4096	7.7811e-04	3.2875	96	00:00:00	0.02
128	16384	1.9765e-04	3.2891	192	00:00:00	0.04
256	65536	4.9797e-05	3.2891	387	00:00:00	0.19
512	262144	1.2494e-05	3.2881	783	00:00:02	2.07
1024	1048576	3.1266e-06	3.2849	1581	00:00:20	20.73
2048	4194304	7.8019e-07	3.2756	3192	00:03:44	224.88
4096	16777216	1.9366e-07	3.2507	6452	01:02:45	3765.69

After putting all the C code together (as described in Section 2), we ran a study of results for the same N-values plus higher values and assembled the results in a new table:

Table 3.1.2: C Code Results

N	DOF	$\|\|u - u_h\|\|$	Ratio	`#iter`	wall clock time	
					HH:MM:SS	seconds
32	1024	3.0128e-03	3.2809	48	00:00:00	0.00
64	4096	7.7811e-04	3.2875	96	00:00:00	0.00
128	16384	1.9765e-04	3.2891	192	00:00:00	0.01
256	65536	4.9797e-05	3.2891	387	00:00:00	0.08
512	262144	1.2494e-05	3.2881	783	00:00:00	0.75
1024	1048576	3.1266e-06	3.2849	1581	00:00:09	10.01
2048	4194304	7.8019e-07	3.2756	3192	00:01:32	92.62
4096	16777216	1.9366e-07	3.2507	6452	00:12:57	777.44
8192	67108864	4.7392e–08	3.1801	13033	02:04:09	7449.14
16384	268435456	1.1647e–08	3.0994	26316	13:45:03	49503.22

Comparisons

By observing both tables, we see that the iterations columns are identical, which means we can verify that the code is giving accurate results. Additionally, we see that the $||u - u_h||$ columns and Ratio columns contain results that are either identical or very close. The first few decimal places of each value seem to be the same. Also, all of the ratio values are fairly close to 4, which is what we explained in the introduction.

The "wall clock time" column is where we observe differences. Overall, we see that the C code is noticeably faster. In fact, for each trial tested, the time was faster. For smaller N-values, such as 32 and 64, the differences are not as significant because fewer computations are required. However, starting at N=512, we see larger differences. We have 0.75 vs. 2.07 seconds, which is the first large difference. Then, we have 10.01 vs. 20.73 seconds, which is almost half the time. Then, we have a larger difference between 92.62 and 224.88 seconds, and the largest speedup is noticed at N=4096. Here, we have 777.44 vs. 3765.69 seconds, which is about one hour compared to 13 minutes. We can expect to see significant speedup for even larger N-values when using the C code as well.

3.2 Parallel Performance Study with Blocking MPI_Send/MPI_Recv

After finalizing and correcting the serial code, we implemented MPI_Send and MPI_Recv commands, which we described in Section 2.2. We created a table containing N-values ranging from 1024 to 16384 with the wall clock time for each run. The table contains the results for 1 node through 16 nodes with 1 through 32 processes per node for each.

Table 3.2.1

Wall clock time in HH:MM:SS Format:

n = 1024					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:00:09	00:00:11	00:02:20	00:00:01	00:00:01
2 processes per node	00:00:04	00:00:05	00:01:15	00:00:01	00:00:01
4 processes per node	00:00:01	00:00:00	00:00:41	00:00:00	00:00:01
8 processes per node	00:00:01	00:00:00	00:00:00	00:00:00	00:00:01
16 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
32 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

n = 2048					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:01:31	00:00:48	00:00:48	00:00:08	00:00:03
2 processes per node	00:00:44	00:00:22	00:00:08	00:00:03	00:00:05
4 processes per node	00:00:20	00:00:08	00:00:03	00:00:05	00:00:03
8 processes per node	00:00:10	00:00:04	00:00:01	00:00:02	00:00:01
16 processes per node	00:00:07	00:00:02	00:00:01	00:00:01	00:00:01
32 processes per node	00:00:05	00:00:01	00:00:01	00:00:00	00:00:01

n = 4096					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:13:02	00:06:28	00:07:31	00:03:43	00:00:43
2 processes per node	00:06:31	00:03:41	00:01:31	00:00:45	00:00:48
4 processes per node	00:03:13	00:01:31	00:00:48	00:00:19	00:00:24
8 processes per node	00:01:49	00:00:48	00:00:22	00:00:09	00:00:12
16 processes per node	00:01:11	00:00:34	00:00:16	00:00:05	00:00:07
32 processes per node	00:01:03	00:00:29	00:00:19	00:00:03	00:00:03

n = 8192	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	01:49:01	00:57:14	00:26:58	00:31:31	00:15:43
2 processes per node	00:54:04	00:30:14	00:13:25	00:15:24	00:07:40
4 processes per node	00:27:27	00:32:03	00:07:40	00:07:37	00:03:41
8 processes per node	00:41:22	00:07:07	00:03:46	00:04:01	00:02:16
16 processes per node	00:09:53	00:05:03	00:02:35	00:02:32	00:01:33
32 processes per node	00:08:42	00:09:24	00:02:12	00:02:05	00:00:53

n = 16384	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	13:45:03	07:55:03	04:03:17	01:56:09	01:16:51
2 processes per node	07:22:45	03:41:12	02:03:06	00:58:37	01:09:10
4 processes per node	03:38:30	01:52:04	01:02:08	00:30:43	00:32:17
8 processes per node	01:56:44	01:02:52	00:31:02	00:15:23	00:18:47
16 processes per node	01:24:28	00:40:26	00:20:47	00:10:23	00:22:18
32 processes per node	01:10:08	00:35:29	00:17:43	00:19:03	00:13:42

Observations

This table shows the timing results for the parallel implementation of the code. As the size of N increases, it is apparent that the runtime also increases, most considerably at N=8192 and N=16384. For these larger values of N, we observe that increasing the number of nodes as well as the number of processes leads to a considerable speedup in wall clock time. However, the speedup is not as significant for the smaller values of N, specifically 1024 and 2048. For N=4096, N=8192, and N=16384, we can see speedup in all the columns.

Comparing these results to our results obtained earlier in the serial version of the code, and to reiterate the observation above, we find that there is only a substantial benefit to running the code in parallel using multiple nodes and processes when N is considerably large. In the case of N=4096, there is a speedup of around 12 minutes when it is run across 4 nodes running 16 processes as opposed to being run in serial. We expect that the speedup will be even more significant for larger N-values.

3.3 Parallel Performance Study with Non-Blocking MPI_Isend/MPI_Irecv

We created Table 3.3.1 here, which shows the timing results for our new non-blocking implementation. Like Table 3.2.1, this table contains N-values ranging from 1024 to 16384 with the wall clock time for each run. It contains the results for 1 node through 16 nodes with 1 through 32 processes per node for each.

Table 3.3.1

Wall clock time in HH:MM:SS Format:

n = 1024	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:00:09	00:00:03	00:00:02	00:00:00	00:00:01
2 processes per node	00:00:03	00:00:02	00:00:02	00:00:00	00:00:01
4 processes per node	00:00:02	00:00:02	00:00:00	00:00:00	00:00:00
8 processes per node	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00
16 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
32 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

n = 2048	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:01:33	00:00:46	00:00:24	00:00:11	00:00:07
2 processes per node	00:00:44	00:00:19	00:00:07	00:00:03	00:00:07
4 processes per node	00:00:24	00:00:10	00:00:13	00:00:05	00:00:00
8 processes per node	00:00:19	00:00:05	00:00:03	00:00:03	00:00:01
16 processes per node	00:00:08	00:00:03	00:00:00	00:00:00	00:00:00
32 processes per node	00:00:06	00:00:00	00:00:00	00:00:00	00:00:00

n = 4096	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:13:53	00:01:48	00:01:10	00:01:01	00:01:12
2 processes per node	00:07:05	00:01:31	00:00:48	00:00:33	00:00:29
4 processes per node	00:03:35	00:01:45	00:00:50	00:00:17	00:00:23
8 processes per node	00:01:32	00:00:42	00:00:17	00:00:09	00:00:10
16 processes per node	00:01:14	00:00:42	00:00:23	00:00:12	00:00:06
32 processes per node	00:01:02	00:00:31	00:00:18	00:00:03	00:00:03

n = 8192	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	02:04:09	01:02:30	00:27:31	00:32:12	00:15:49
2 processes per node	00:54:00	00:27:41	00:14:36	00:06:52	00:07:35
4 processes per node	00:30:31	00:13:35	00:06:34	00:07:41	00:03:41
8 processes per node	00:15:21	00:07:08	00:03:30	00:04:47	00:02:13
16 processes per node	00:09:54	00:05:02	00:02:24	00:02:23	00:01:08
32 processes per node	00:08:41	00:04:19	00:04:23	00:01:11	00:00:55

n = 16384	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	14:38:23	07:53:12	03:53:57	01:41:22	01:07:09
2 processes per node	08:07:53	04:02:08	01:51:58	00:53:01	01:05:04
4 processes per node	04:06:17	01:51:17	00:55:07	00:31:04	00:32:16
8 processes per node	02:04:47	00:58:38	01:18:36	00:14:51	00:20:18
16 processes per node	01:24:11	00:41:56	00:41:26	00:11:13	00:21:48
32 processes per node	01:10:52	00:35:26	00:18:09	00:10:43	00:09:30

Observations

By looking through these results, it is clear that increasing the number of nodes and the processes per node increases the runtime. For N=1024, it is clear that the parallel results are all faster than the serial result (9 seconds). We see the same pattern for the other N-values as well. Generally, as we move to the bottom-right side of the table, we notice faster runtimes.

Comparing to Table 4.1 in HPCF-2019-1

For N=1024, the results from our table are very similar to the ones in Table 4.1 from HPCF-2019-1. For 1 node, we have identical results, except for 9 seconds vs. 8 seconds for the serial run. The results for 2 nodes are also about the same, with just a one-second difference for 4 processes. The remaining results are also very similar.

For N=2048, the results are all close to each other. The serial runtime is about the same, and the remaining results for 1 and 2 nodes are similar. For 16 nodes, our runtimes are a little slower, but they are fairly close overall.

Across our entire table, we notice times that are slightly faster, but this may be due to differences in trials. If we re-ran these tests many times, we may notice that they are closer to the ones in Table 4.1.

Comparing to Blocking MPI

When looking at both tables, we see that the runtimes are fairly similar for all N-values. For N=1024 and 2048, the results are about the same, but performance gains are clear as more processes per node are used, reducing time to near zero for 32 processes across 16 nodes.

Then, for N=4096, we see a few differences. For 2 nodes, the blocking case took over 6 minutes for 1 process where the non-blocking case only took 01:48. Looking at the 4, 8, and 16 nodes columns, we see a trend where the blocking results are longer, especially when we observe the 1 process per node column.

For N=8192, we see mixed results. For the blocking case, the serial run was a little faster compared to the non-blocking case (01:49:01 vs. 02:04:09). The rest of the column is about the same aside from the 41:22 runtime for 8 processes for the blocking case, which may be an outlier. In the 2 nodes column, the blocking results for 4 processes are larger compared to the non-blocking case (32:03 vs. 13:35). Otherwise, the rest of the tables are similar.

For N=16384, the non-blocking serial result was noticeably longer than the blocking result (14:38:23 vs. 13:45:03). This may have just been the result of one trial, though. If we re-ran the serial result many times, we may see different timings. Throughout the entire 1 node column, we see that the blocking results are a little faster, which contradicts our expectations. We thought that the non-blocking results would be consistently faster. In the 2 nodes column, we see that for 2 processes, the blocking case was faster, and the remaining results were similar. In the 4 nodes column, the blocking results were consistently faster too. In the 8 and 16 nodes columns, we see mixed results where sometimes the blocking code was faster, and sometimes the non-blocking was faster.

In general, we are seeing mixed results. Neither the blocking nor the non-blocking code led to extreme timing differences. However, the results do convey that non-blocking MPI calls can provide slightly shorter runtimes when compared to the blocking calls when the number of nodes and processes per node are larger. By looking closely at the 8 nodes and 16 nodes columns, we notice this. However, we do have some situations where the blocking code is actually faster.

Table 3.3.2: Blocking Time, Speedup, and Efficiency

Wall clock time in HH:MM:SS Format, Observed speedup, and Observed efficiency:

(a) Wall clock time T_p in HH:MM:SS

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
1024	00:00:09	00:00:04	00:00:01	00:00:01	00:00:00	00:00:00
2048	00:01:31	00:00:44	00:00:20	00:00:10	00:00:07	00:00:05
4096	00:13:02	00:06:31	00:03:13	00:01:49	00:01:11	00:01:03
8192	01:49:01	00:54:04	00:27:27	00:41:22	00:09:53	00:08:42
16384	13:45:03	07:22:45	03:38:30	01:56:44	01:24:28	01:10:08

(b) Observed speedup $S_p = T_1/T_p$

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
1024	1.00	2.25	9.00	9.00	19.57	40.91
2048	1.00	2.07	4.55	9.10	13.00	18.20
4096	1.00	2.00	4.05	7.17	11.01	12.41
8192	1.00	2.02	3.97	2.64	11.03	12.53
16384	1.00	1.86	3.78	7.07	9.77	11.76

(c) Observed efficiency $E_p = S_p/p$

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
1024	1.00	1.12	2.25	1.12	1.22	1.28
2048	1.00	1.03	1.14	1.14	0.81	0.57
4096	1.00	1.00	1.01	0.90	0.69	0.39
8192	1.00	1.01	0.99	0.33	0.69	0.39
16384	1.00	0.93	0.94	0.88	0.61	0.37

To briefly describe these results, table (a) has the runtimes for 1 node from the case study we ran, table (b) contains the observed speedup using the formula $S_p = T_1/T_p$, and table (c) contains the observed efficiency using the formula $E_p = S_p/p$.

First, looking at table (b), we see that all the values in the $p = 2$ column are around 2. Then, for $p = 4, 8$, and 16, we have a variety of results. In the $p = 32$ column, we have 40.91 for N=1024, which is significantly larger than the other values. We will see the effect this has in the graph we will observe later.

Next, in table (c), we see more consistent values. In each column, all of the values are around 0 to 1 except for N=1024 and 4 processes. These values are expected because they should be close to 1 to resemble the optimal line, which we will show in the graphs later (in Figure 1).

Table 3.3.3: Non-Blocking Time, Speedup, and Efficiency

Wall clock time in HH:MM:SS Format, Observed speedup, and Observed efficiency:

(a) Wall clock time T_p in HH:MM:SS

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
1024	00:00:09	00:00:03	00:00:02	00:00:01	00:00:00	00:00:00
2048	00:01:33	00:00:44	00:00:24	00:00:19	00:00:08	00:00:06
4096	00:13:53	00:07:05	00:03:35	00:01:32	00:01:14	00:01:02
8192	02:04:09	00:54:00	00:30:31	00:15:21	00:09:54	00:08:41
16384	14:38:23	08:07:53	04:06:17	02:04:47	01:24:11	01:10:52

(b) Observed speedup $S_p = T_1/T_p$

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
1024	1.00	3.00	4.50	9.00	20.00	40.91
2048	1.00	2.11	3.88	4.89	11.62	15.50
4096	1.00	1.96	3.87	9.05	11.26	13.44
8192	1.00	2.30	4.07	8.09	12.54	14.30
16384	1.00	1.80	3.57	7.04	10.43	12.39

(c) Observed efficiency $E_p = S_p/p$

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
1024	1.00	1.50	1.12	1.12	1.25	1.28
2048	1.00	1.06	0.97	0.61	0.73	0.48
4096	1.00	0.98	0.97	1.13	0.70	0.42
8192	1.00	1.15	1.02	1.01	0.78	0.45
16384	1.00	0.90	0.89	0.88	0.65	0.39

Here, the results seem similar to the blocking results. However, in table (b), we notice that the values in the $p = 4$ column are more consistent here. They are all very close to 4. In the blocking case, we had 9.00 for N=1024, which was far off. The remaining results are similar. In table (c), all the results are consistent. Every value is close to 1, and there are no outliers.

Observed Speedup Plots: Blocking-MPI and Non-Blocking MPI

The following plots were generated using a MATLAB script (sample provided in the Appendix) that utilizes the above data. The upper plot is "Observed Speedup (Sp) vs Number of Parallel Processes (p) for Blocking MPI," and the lower plot underneath is "Observed Speedup (Sp) vs Number of Parallel Processes (p) for Non-Blocking MPI":

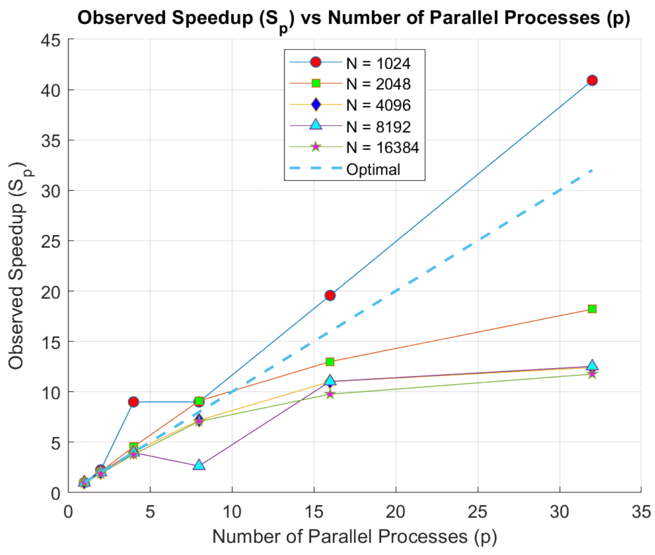

Figure 1: Observed Speedup for Blocking MPI

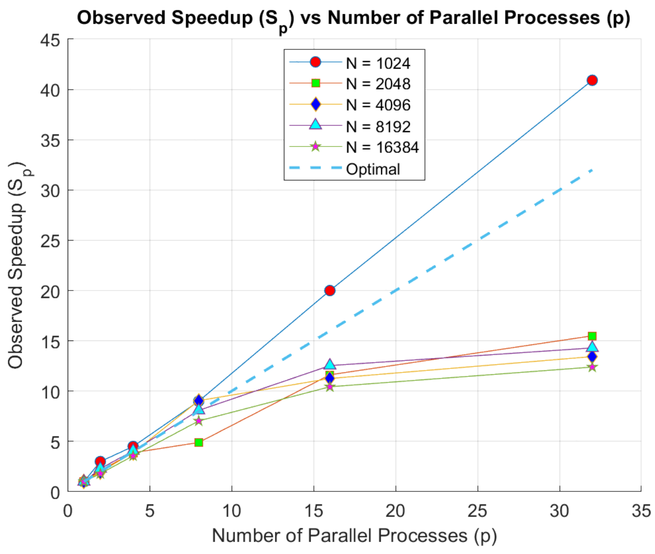

Figure 2: Observed Speedup for Non-Blocking MPI

Analysis: Observed Speedup

First, we will analyze Figure 1: Observed Speedup for Blocking MPI. In this plot, we see that all the curves generally follow the optimal line, although the curves for N=2048, 4096, 8192, and 16384 level off rather than following the optimal line. Another observation to note is the curves for N=4096, 8192, and 16384 appear to be grouped closely together. Additionally, the curve for N=1024 most closely resembles the optimal line, and the curve for N=2048 is between the optimal line and the other closely grouped curves. Lastly, we see that the N=8192 curve deviates from the others at 8 processes, then comes back up again.

When we observe Figure 2: Observed Speedup for Non-blocking MPI, we see that the N=1024 curve follows the optimal line well. It follows the same trend but increases around 16 and 32 processes, making it go slightly above the optimal line. Nevertheless, it signifies to us that our N=1024 tests were very optimal. Then, looking at the other curves (for N=2048, 4096, 8192, and 16384), it is clear that these all follow the same trend. In fact, all of the curves appear to be compact and grouped together. We can see that for 1, 2, 4, and 8 processes per node, all the curves are fairly close to the optimal line. The N=2048 curve does differ more than the rest because it comes down temporarily around 8 processes per node, but it later matches the others. None of these curves exactly match the optimal line, but they all follow the general trend. We can see that around 16 processes per node, the graphs level off in comparison to the optimal line.

Observed Efficiency Plots: Blocking-MPI and Non-Blocking MPI

The following plots were generated using the same MATLAB script that utilizes the above data. The upper plot is "Observed Efficiency (Ep) vs Number of Parallel Processes (p) for Blocking MPI", and the lower plot underneath is "Observed Efficiency (Ep) vs Number of Parallel Processes (p) for Non-Blocking MPI":

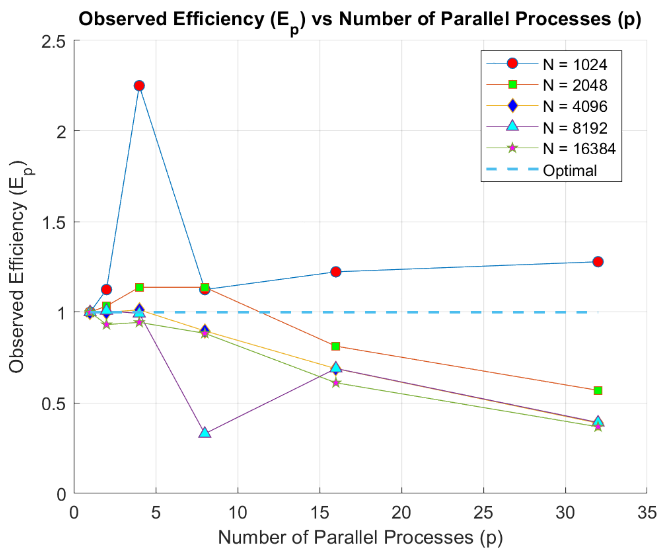

Figure 3: Observed Efficiency for Blocking MPI

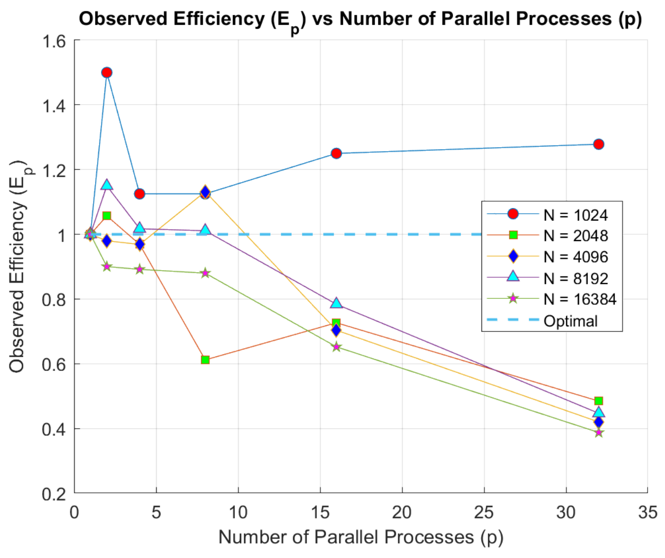

Figure 4: Observed Efficiency for Non-Blocking MPI

Analysis: Observed Efficiency

First, we will look at Figure 3: Observed Efficiency for Blocking MPI. Looking at N=1024, we see that the curve spikes at 4 processes, then comes down and resembles the optimal line. Also, the N=8192 curve dips down at 8 processes, and then comes up afterward. Lastly, we see that the curves for N=2048, 4096, and 16384 are all grouped together like they were in the speedup plot. They decrease as the number of processes increases, so they do not follow the optimal line exactly, but they do follow the general trend. Aside from a couple of spikes, all the curves are fairly horizontal.

By observing Figure 4: Observed Efficiency for Non-Blocking MPI, we see a variety of results. For N=1024, the curve spikes at 2 processes per node, and then levels off and follows the contour of the optimal line afterwards. It does increase in relation to the optimal line, but it still follows the general flat-line shape. Then, looking at the other curves (for N=2048, 4096, 8192, and 16384), we see that the curves are grouped together like they were in the speedup plots, but they all follow a decreasing trend as opposed to following the straight horizontal line. Additionally, we can see that the N=2048 curve deviates from the rest around 8 processes per node, but then follows the other curves afterwards. In general, we can see that all the curves are fairly optimal.

4 Conclusions

From our observations, we observed that the C code had significantly shorter runtimes when compared to the MATLAB code. With each increase in N, we saw noticeably faster runtimes in the C code results, and we noticed that the larger the N-value was, the larger the difference was between the runtime from the MATLAB code and the C code. In summary, the non-blocking code can provide a slight advantage, but the blocking and non-blocking cases gave very similar results.

Recommendation

We recommend using non-blocking communication commands on our system for a numerical problem of this type, specifically, a numerical computing problem that involves the Finite Difference Method for solving partial differential equations, especially when the data size is considerably large. Also, this problem type involves parallel programming and performance optimization using MPI for parallel executions on a cluster of machines.

We conclude that non-blocking (using MPI_Isend and MPI_Irecv commands) is advantageous for problems of this type because it can provide slight speedup when utilizing multiple nodes and processes per node. This slight speedup can lead to a substantial decrease in runtime when considering large data sizes, such as N-values equal to or greater than 16384.

Evidence

By looking at Table 3.3.3: Non-Blocking Time, Speedup, and Efficiency, we saw that the values in the $p = 4$ columns were all very close to 4, meaning we had more consistency here compared to the blocking case.

Moreover, for N=4096, we saw that the runtimes for the non-blocking case were faster in several places. For example, for 2 nodes and 1 process, we had 01:48 for non-blocking vs. 06:28 for blocking. For 4 nodes and 1 process, we had 01:10 for non-blocking vs. 07:31 for blocking. Also, for 8 nodes and 1 process, we had 01:01 for non-blocking vs. 03:43 for blocking.

For N=8192, for 8 nodes and 2 processes, we had 06:52 for non-blocking vs. 15:24 for blocking. Although these are all just individual examples, it does demonstrate that non-blocking MPI may provide a slight advantage, especially when running a program with large N-values. If you are able to run your program in parallel, by using multiple nodes and processes per node, this can provide an advantage.

Since the runtimes for both the blocking and non-blocking cases were very similar, though, and the speedup and efficiency plots closely resembled each other, either method can be used for smaller programs. However, when using large N-values, we recommend using non-blocking MPI, especially if you are going to use N-values that are greater than 16384. We predict that it will lead to more speedup as N is increased beyond this.

Acknowledgments

Department of Mathematics and Statistics: UMBC

This work was completed as part of a class project for MATH 447/627: Introduction to Parallel Computing. For references, we utilized HPCF-2017-3, Allen (UMBC Review), Pacheco for MPI, HPCF-2019-1 and HPCF-2017-3 for the Finite Difference Method, hpcf.umbc.edu, and the Intel MKL documentation. Addtionally, we received help from Dr. Matthias K. Gobbert, who provided parts of the code and continuously helped us revise our report.

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS-0821258, CNS-1228778, OAC-1726023, and CNS-1920079) and the SCREMS program (grant no. DMS-0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See hpcf.umbc.edu for more information on HPCF and the projects using its resources.

References

[1] Kevin P. Allen. Efficient parallel computing for solving linear systems of equations. *UMBC Review: Journal of Undergraduate Research and Creative Works*, 5:8–17, 2004.

[2] Carlos Barajas and Matthias K. Gobbert. Strong and weak scalability studies for the 2-d poisson equation on the taki 2018 cluster. Technical Report HPCF-2019-1, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2019.

[3] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, San Francisco, CA, 1997.

[4] Sai K. Popuri and Matthias K. Gobbert. A comparative evaluation of Matlab, Octave, R, and Julia on maya. Technical Report HPCF-2017-3, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2017. HPCF machines used: Maya. Additional files available online at `https://hpcf-files.umbc.edu/research/papers/PopuriHPCF2017/1_Matlab/`.

Appendix

utilities.c

```
#include "utilities.h"

double serial_dot (double *x, double *y, long n)
{
  double dp;
#ifndef BLAS
  long i;
#endif

#ifdef BLAS
  dp = cblas_ddot(n, x, 1, y, 1);
#else
  dp = 0.0;
  // #pragma omp parallel for reduction(+:dp)
  for(i = 0; i < n; i++)
    dp += x[i]* y[i];
#endif

  return dp;
}

double parallel_dot(double *l_x, double *l_y, long l_n, MPI_Comm comm) {

  double l_dot=0.0;
  double dot=0.0;

  l_dot = serial_dot(l_x, l_y, l_n);

#ifdef PARALLEL
  MPI_Allreduce(&l_dot, &dot, 1, MPI_DOUBLE, MPI_SUM, comm);
#else
  dot = l_dot;
#endif

  return dot;

}

double parallel_norm2 (double *l_x, long l_n, MPI_Comm comm) {

  return sqrt(parallel_dot(l_x,l_x,l_n,comm));
}
```

main.c

```
#include "main.h"

#define SQ(X) ((X)*(X))

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

  int id, np, namelen, idleft, idright;
  int rem, lenm;
  char name[MPI_MAX_PROCESSOR_NAME];
  char filename[64];
  char message[100];
  int iter, maxit, flag;
  long N, l_N, n, l_n, i, j, l_j, l_k;
  long l_ia, l_ib;
  long l_Ntmp, l_ntmp;
  double *l_u, *l_r, *l_p, *l_q;
  double *x, *y;
  double *gl, *gr;
  double h, tol, relres;
  double enorminf, l_enorminf, err_ij;
  double start, end;
  FILE *idfil;
  MPI_Comm comm;
  MPI_Status status;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &np);
  MPI_Comm_rank (MPI_COMM_WORLD, &id);
  MPI_Get_processor_name (name, &namelen);

  if (id > 0)
    idleft = id - 1;
  else
    idleft = MPI_PROC_NULL;
  if (id < np-1)
    idright = id + 1;
  else
    idright = MPI_PROC_NULL;
  comm = MPI_COMM_WORLD;

  /* process command-line inputs: */
  if (argc != 4){
    if (id == 0){
      printf ("Usage: \"poisson N tol maxit\" \n");
      printf (" with integer n, real tol, and integer maxit\n");
    }
    MPI_Abort (MPI_COMM_WORLD, 1);
  }
  N     = (long)(atof(argv[1]));
  tol   =       (atof(argv[2]));
  maxit =  (int)(atof(argv[3]));

  h = 1.0/(N + 1.0);  /*step size*/
  n = SQ(N);  /* n = N^2 */
```

```
/* in the following distribution, the first rem=N%np processes get
 * N/np+1 values, and the remaining np-rem ones get N/np,
 * for a total of rem*(N/np+1) + (np-rem)*(N/np)
 * = rem*(N/np) + rem + np*(N/np) - rem*(N/np)
 * = (rem+np-rem)*(N/np) + rem = np*(N/np) + rem = (N-rem) + rem = N */
rem = N%np; /* remainder when dividing N values into np MPI processes */
if (id < rem) { /* if id is 0, 1, ..., rem-1 then one more value: */
  l_N = N/np + 1;                  /* local number of values */
} else {
  l_N = N/np;                      /* local number of values */
}
l_n = N * l_N;    /* size of local matrix l_u */

/* Notice carefully: start and ending index are coded "C-style",
 * i.e., a for-loop should read: for(l_i=l_ia; l_i<l_ib; l_i++)
 * with a strictly less than comparison. */
rem = N%np; /* remainder when dividing N values into np MPI processes */
if (id < rem) { /* if id is 0, 1, ..., rem-1 then one more value: */
  l_ia = id*l_N;                   /* local starting index */
  l_ib = l_ia + l_N;               /* local ending index */
} else {
  l_ia = rem*(l_N+1)+(id-rem)*l_N; /* local starting index */
  l_ib = l_ia + l_N;               /* local ending index */
}

/* test output: */
if (np <= 8) { /* for more than np=8 this becomes unreadable */
  sprintf(message, "P%03d: l_n=%12ld; l_N=%6ld from l_ia=%6ld to l_ib=%6ld",
                   id, l_n, l_N, l_ia, l_ib);
  if (id == 0) {
    for (i = 0; i < np; i++) {
      if (i > 0)
        MPI_Recv(message,100,MPI_CHAR,i,i,MPI_COMM_WORLD,&status);
      printf("[%3d] %s\n", id, message);
    }
  } else {
    lenm = 1 + strlen(message);
    MPI_Send(message,lenm,MPI_CHAR,0,id,MPI_COMM_WORLD);
  }
}

if (id == 0) {
  printf("N = %6ld, tol = %10.1g, maxit = %d\n", N, tol, maxit);
  printf("n = %12ld, l_N = %6ld, l_n = %12ld\n", n, l_N, l_n);
}

l_u = allocate_double_vector(l_n);  /* initial guess */
l_r = allocate_double_vector(l_n); /* right-hand side in, residual out */
l_p = allocate_double_vector(l_n);
l_q = allocate_double_vector(l_n);
x = allocate_double_vector(N);
y = allocate_double_vector(N);
gl = allocate_double_vector(n/N);
gr = allocate_double_vector(n/N);
```

```
for (i = 0; i < N; i++){
  x[i] = h * (double)(i+1);
}
for (j = 0; j < N; j++){
  y[j] = h * (double)(j+1);
}

/* initializations of the 4 large arrays in the code: */
/* solution vector l_u */
for (l_j = 0; l_j < l_N; l_j++){
  for (i = 0; i < N; i++){
    l_k = i + N*l_j;
    l_u[l_k] = 0.0;
  }
}
/* residual vector l_r = right-hand side l_b */
for (l_j = 0; l_j < l_N; l_j++){
  j = l_j + l_ia; /* l_j + l_N*id; */
  for (i = 0; i < N; i++){
    l_k = i + N*l_j;
    l_r[l_k] = SQ(h)*(-2*SQ(M_PI))*
               (cos(2*M_PI*x[i])*SQ(sin(M_PI*y[j]))
               + cos(2*M_PI*y[j])*SQ(sin(M_PI*x[i])));
  }
}
/* search direction l_p */
for (l_j = 0; l_j < l_N; l_j++){
  for (i = 0; i < N; i++){
    l_k = i + N*l_j;
    l_p[l_k] = 0.0;
  }
}
/* auxiliary vector l_q */
for (l_j = 0; l_j < l_N; l_j++){
  for (i = 0; i < N; i++){
    l_k = i + N*l_j;
    l_q[l_k] = 0.0;
  }
}

for (i = 0; i < N; i++) {
  gl[i] = 0.0;   /*Setting to zero temporary */
  gr[i] = 0.0;
}

// temporary fix: initialize all variables that will come from cg:
flag = -69;
relres = -69.0;
iter = -69;
enorminf = -69.0;

MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();  /* start time */
```

```
  //commented out, until utility functions parallel_dot() and Ax() exist:
  cg(l_u, &flag, &relres, &iter, l_r, tol, maxit,
     l_p, l_q, l_n, l_N, N, id, idleft, idright, np, comm, gl, gr);

  MPI_Barrier(MPI_COMM_WORLD);
  end = MPI_Wtime();  /* end time */

  // Layout from 5Poisson_HW5a.pdf
  double utrue;
  l_enorminf = 0.0;

  for (l_j = 0; l_j < l_N; l_j++) {
    for (i = 0; i < N; i++) {
      j = l_j + l_ia;
      utrue = pow(sin(M_PI*x[i]), 2)
        * pow(sin(M_PI*y[j]), 2);
      err_ij = fabs(l_u[i+l_j*N] - utrue);
      if (err_ij > l_enorminf)
        l_enorminf = err_ij;
    }
  }
    MPI_Reduce(&l_enorminf, &enorminf, 1, MPI_DOUBLE,
               MPI_MAX, 0, MPI_COMM_WORLD);

  if (id == 0) {
    printf("flag = %1d, iter = %d\n", flag, iter);
    printf("relres             = %24.16e\n", relres);
    printf("h                  = %24.16e\n", h);
    printf("h^2                = %24.16e\n", h*h);
    printf("enorminf           = %24.16e\n", enorminf);
    printf("C = enorminf / h^2 = %24.16e\n", (enorminf/(h*h)));
    printf("wall clock time    = %10.2f seconds\n", end-start);
    fflush(stdout);
  }

  MPI_Finalize();

  free_vector(l_q);
  free_vector(l_r);
  free_vector(l_p);
  free_vector(l_u);
  free_vector(x);
  free_vector(y);
  free_vector(gl);
  free_vector(gr);

  return 0;
}
```

Serial Ax.c

```
#include "Ax.h"

void Ax(double *l_v, double *l_u, long l_n, long l_N, long N,
        int id, int idleft, int idright, int np, MPI_Comm comm,
        double *gl, double *gr) {

  long i, l_j;
  double tmp;
  MPI_Status statuses[4];
  MPI_Request requests[4];

  if (np > 1) {
    MPI_Irecv(gl             , N, MPI_DOUBLE, idleft ,  1, MPI_COMM_WORLD, &(requests[0]));
    MPI_Irecv(gr             , N, MPI_DOUBLE, idright,  2, MPI_COMM_WORLD, &(requests[1]));
    MPI_Isend(&l_u[N*(l_N-1)], N, MPI_DOUBLE, idright,  1, MPI_COMM_WORLD, &(requests[2]));
    MPI_Isend(&l_u[0]        , N, MPI_DOUBLE, idright,  2, MPI_COMM_WORLD, &(requests[3]));
  }

  for(l_j = 0; l_j < l_N; l_j++) {
    for (i = 0; i < N; i++) {
                 tmp = 4.0 * l_u[i   + N* l_j   ];
      if (l_j > 0) tmp = tmp - l_u[i   + N*(l_j-1)];
      if (i > 0)   tmp = tmp - l_u[i-1 + N* l_j   ];
      if (i<N-1)   tmp = tmp - l_u[i+1 + N* l_j   ];
      if (l_j<N-1) tmp = tmp - l_u[i   + N*(l_j+1)];
          l_v[i+N*l_j] = tmp;
    }
  }
}
```

Parallel Ax.c (Blocking MPI)

```
#include "Ax.h"

void Ax(double *l_v, double *l_u, long l_n, long l_N, long N,
        int id, int idleft, int idright, int np, MPI_Comm comm,
        double *gl, double *gr) {

  long i, l_j;
  double tmp;
  MPI_Status status;

  for(l_j = 1; l_j < l_N-1; l_j++) { //Block A
      for (i = 0; i < N; i++) {
                     tmp  = 4.0 * l_u[i   + N*  l_j   ];
          if (l_j >     0) tmp -= l_u[i   + N* (l_j-1)];
          if (i   >     0) tmp -= l_u[i-1 + N*  l_j   ];
          if (i   <   N-1) tmp -= l_u[i+1 + N*  l_j   ];
          if (l_j < l_N-1) tmp -= l_u[i   + N* (l_j+1)];
          l_v[i+N*l_j] = tmp;
      }
  }

  if (id%2 == 0) { //Block D
      MPI_Recv(gl,                N, MPI_DOUBLE, idleft,  1, comm, &status);
      MPI_Recv(gr,                N, MPI_DOUBLE, idright, 2, comm, &status);
      MPI_Send(&(l_u[N*l_N - N]), N, MPI_DOUBLE, idright, 1, comm         );
      MPI_Send(&(l_u[    0    ]), N, MPI_DOUBLE, idleft,  2, comm         );
  } else {
      MPI_Send(&(l_u[N*l_N - N]), N, MPI_DOUBLE, idright, 1, comm         );
      MPI_Send(&(l_u[    0    ]), N, MPI_DOUBLE, idleft,  2, comm         );
      MPI_Recv(gl,                N, MPI_DOUBLE, idleft,  1, comm, &status);
      MPI_Recv(gr,                N, MPI_DOUBLE, idright, 2, comm, &status);
  }

  l_j = 0;
  for (i = 0; i < N; i++) { //Block B
               tmp = 4.0 * l_u[i     + N*  l_j     ];
    if (id  >     0)  tmp -=   gl[i];
    if (i   >     0)  tmp -=  l_u[i-1   + N*  l_j     ];
    if (i   <   N-1)  tmp -=  l_u[i+1   + N*  l_j     ];
    if (l_j < l_N-1)  tmp -=  l_u[i     + N* (l_j + 1)];
    l_v[i + N * l_j] = tmp;
  }

  l_j = l_N -1;
  for (i = 0; i < N; i++) { //Block C
                   tmp = 4.0 * l_u[i   + N*  l_j   ];
      if (l_j >     0 )  tmp -= l_u[i   + N* (l_j-1)];
      if (i   >     0 )  tmp -= l_u[i-1 + N*  l_j   ];
      if (i   <   N-1 )  tmp -= l_u[i+1 + N*  l_j   ];
      if (id  <   np-1)  tmp -= gr[i];
      l_v[i + N* l_j] = tmp;
  }
}
```

Parallel Ax.c (Non-Blocking MPI)

```c
#include "Ax.h"

void Ax(double *l_v, double *l_u, long l_n, long l_N, long N,
        int id, int idleft, int idright, int np, MPI_Comm comm,
        double *gl, double *gr) {

  long i, l_j;
  double tmp;
  MPI_Request requests[4];
  MPI_Status statuses[4];


  MPI_Irecv(gl,              N, MPI_DOUBLE, idleft,  1, comm, &(requests[0]));
  MPI_Irecv(gr,              N, MPI_DOUBLE, idright, 2, comm, &(requests[1]));
  MPI_Isend(&l_u[N*(l_N-1)], N, MPI_DOUBLE, idright, 1, comm, &(requests[2]));
  MPI_Isend(&l_u[0],         N, MPI_DOUBLE, idleft,  2, comm, &(requests[3]));

  for (l_j = 1; l_j < l_N-1; l_j++){
    for (i = 0; i < N; i++){
                       tmp = 4.0 *   l_u[i + N*l_j];
      if (l_j > 0)       tmp = tmp -   l_u[i + N*(l_j-1)];
      if (i   > 0)       tmp = tmp -   l_u[i - 1+N*l_j];
      if (i   < N-1)     tmp = tmp -   l_u[i + 1 + N*l_j];
      if (l_j < l_N-1)   tmp = tmp -   l_u[i + N*(l_j+1)];
      l_v[i+N*l_j] = tmp;
    }
  }

 MPI_Waitall(4, requests, statuses);

  l_j = 0;
    for (i = 0; i < N; i++){
                       tmp = 4.0 *   l_u[i + N*l_j];
      if (id  > 0)       tmp = tmp -   gl[i]; // l_u[i + N*(l_j-1)];
      if (i   > 0)       tmp = tmp -   l_u[i - 1+N*l_j];
      if (i   < N-1)     tmp = tmp -   l_u[i + 1 + N*l_j];
      if (l_j < l_N-1)   tmp = tmp -   l_u[i + N*(l_j+1)];
      l_v[i+N*l_j] = tmp;
    }

   l_j = l_N-1;
    for (i = 0; i < N; i++){
                       tmp = 4.0 *   l_u[i +     N*l_j];
      if (l_j > 0)       tmp = tmp -   l_u[i +     N*(l_j-1)];
      if (i   > 0)       tmp = tmp -   l_u[i - 1+  N*l_j];
      if (i   < N-1)     tmp = tmp -   l_u[i + 1 + N*l_j];
      if (id  < np-1)    tmp = tmp -   gr[i]; // l_u[i + N*(l_j+1)];
      l_v[i+N*l_j] = tmp;
    }
}
```

MATLAB Script for Non-Blocking MPI

```
% HW_5d.m
% Code for calculating Sp and Ep values and creating graphs for
% Observed Speedup vs. Number of parallel processes and
% Observed Efficiency vs. Number of parallel processes

% Processes per node
p = [1, 2, 4, 8, 16, 32];

% Time converted to seconds
N_1024 = [9, 3, 2, 1, 0.45, 0.22];
N_2048 = [93, 44, 24, 19, 8, 6];
N_4096 = [833, 425, 215, 92, 74, 62];
N_8192 = [7449, 3240, 1831, 921, 594, 521];
N_16384 = [52703, 29273, 14777, 7487, 5051, 4252];

% Calculate Sp for each N
Sp_1024 = N_1024(1) ./ N_1024;
Sp_2048 = N_2048(1) ./ N_2048;
Sp_4096 = N_4096(1) ./ N_4096;
Sp_8192 = N_8192(1) ./ N_8192;
Sp_16384 = N_16384(1) ./ N_16384;

% Calculate Ep for each N
Ep_1024 = Sp_1024 ./ p;
Ep_2048 = Sp_2048 ./ p;
Ep_4096 = Sp_4096 ./ p;
Ep_8192 = Sp_8192 ./ p;
Ep_16384 = Sp_16384 ./ p;

% Print Sp values with & separator (to 2 decimal places)
fprintf('Sp values:\n');
fprintf('N = 1024: '); fprintf('%.2f & ', Sp_1024(1:end-1)); fprintf('%.2f\n', Sp_1024(end));
fprintf('N = 2048: '); fprintf('%.2f & ', Sp_2048(1:end-1)); fprintf('%.2f\n', Sp_2048(end));
fprintf('N = 4096: '); fprintf('%.2f & ', Sp_4096(1:end-1)); fprintf('%.2f\n', Sp_4096(end));
fprintf('N = 8192: '); fprintf('%.2f & ', Sp_8192(1:end-1)); fprintf('%.2f\n', Sp_8192(end));
fprintf('N = 16384: '); fprintf('%.2f & ', Sp_16384(1:end-1)); fprintf('%.2f\n', Sp_16384(end));

% Print Ep values with & separator (to 2 decimal places)
fprintf('Ep values:\n');
fprintf('N = 1024: '); fprintf('%.2f & ', Ep_1024(1:end-1)); fprintf('%.2f\n', Ep_1024(end));
fprintf('N = 2048: '); fprintf('%.2f & ', Ep_2048(1:end-1)); fprintf('%.2f\n', Ep_2048(end));
fprintf('N = 4096: '); fprintf('%.2f & ', Ep_4096(1:end-1)); fprintf('%.2f\n', Ep_4096(end));
fprintf('N = 8192: '); fprintf('%.2f & ', Ep_8192(1:end-1)); fprintf('%.2f\n', Ep_8192(end));
fprintf('N = 16384: '); fprintf('%.2f & ', Ep_16384(1:end-1)); fprintf('%.2f\n', Ep_16384(end));

% S_p = p represents optimal behavior according to HPCF{2019{1 (pg. 8)
optimal = p;

% Plot Observed Speedup (Sp) vs Number of Parallel Processes (p)
figure;
hold on;
plot(p, Sp_1024, '-o', 'DisplayName', 'N = 1024', 'MarkerFaceColor', 'r'); % Circle
plot(p, Sp_2048, '-s', 'DisplayName', 'N = 2048', 'MarkerFaceColor', 'g'); % Square
plot(p, Sp_4096, '-d', 'DisplayName', 'N = 4096', 'MarkerFaceColor', 'b'); % Diamond
plot(p, Sp_8192, '-^', 'DisplayName', 'N = 8192', 'MarkerFaceColor', 'c'); % Triangle
plot(p, Sp_16384, '-p', 'DisplayName', 'N = 16384', 'MarkerFaceColor', 'm'); % Pentagon
plot(p, optimal, '--', 'DisplayName', 'Optimal', 'LineWidth', 1.5);  % Dashed line
xlabel('Number of Parallel Processes (p)');
ylabel('Observed Speedup (S_p)');
title('Observed Speedup (S_p) vs Number of Parallel Processes (p)');
legend('Location', 'best');
grid on;
hold off;

% Plot Observed Efficiency (Ep) vs Number of Parallel Processes (p)
figure;
hold on;
plot(p, Ep_1024, '-o', 'DisplayName', 'N = 1024', 'MarkerFaceColor', 'r'); % Circle
plot(p, Ep_2048, '-s', 'DisplayName', 'N = 2048', 'MarkerFaceColor', 'g'); % Square
plot(p, Ep_4096, '-d', 'DisplayName', 'N = 4096', 'MarkerFaceColor', 'b'); % Diamond
plot(p, Ep_8192, '-^', 'DisplayName', 'N = 8192', 'MarkerFaceColor', 'c'); % Triangle
plot(p, Ep_16384, '-p', 'DisplayName', 'N = 16384', 'MarkerFaceColor', 'm'); % Pentagon
plot(p, optimal ./ p, '--', 'DisplayName', 'Optimal', 'LineWidth', 1.5);  % Dashed line
xlabel('Number of Parallel Processes (p)');
ylabel('Observed Efficiency (E_p)');
title('Observed Efficiency (E_p) vs Number of Parallel Processes (p)');
legend('Location', 'best');
grid on;
hold off;
```