CS471 Homework 6

Mike Adams
Thomas Bowidowicz

December 15, 2021

1 Introduction

This project tasked us with implementing and modeling a heat equation experiment using backward Euler for our time stepping. The equation we were modeling was:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + f(t, x, y), x, y \in \Omega \tag{1}$$

where the Dirichlet boundary conditions are:

$$u(t, x, y) = u_{exact}(t, x, y) = g(t, x, y), x, y \in \partial\Omega \tag{2}$$

and the time domain being defined by:

$$T \in [0, T], T = 0.5 \tag{3}$$

The term u(t,x,y) determined the temperature in the square room, with $u_{exact}(t, x, y)$ determining the initial starting condition, the function g(t,x,y)) being the temperature at the walls, and the function f(t,x,y) being the forcing function that determines how much heat was being injected into or removed from the room. We implemented this in a serial version, a parallel version using MPI message passing, and then ran multiple scaling studies on the CARC supercomputer to determine the efficiency of our code and the speed up that we achieved by parallelization.

2 Task 1: Serial Implementation

2.1 What

We were asked to update the following function so that the boundary conditions were non zero:

$$u(t, x, y) = sin(\pi t)sin(\pi x)sin(\pi y) \tag{4}$$

We were able to update our trigonometric function to provide a non zero boundary condition by adding some constant values within the sine functions. This provided the following function, which was also used as the boundary condition function, $g(t, x, y)$, on the edges of the unit box:

$$u_{exact}(t, x, y) = sin(\pi * t + a) * sin(\pi * x + b) * sin(\pi * y + c) \tag{5}$$

$$a = 0.25, b = 0.5, c = 0.75 \tag{6}$$

Next, we were able to solve for the forcing function and got the following function:

$$f(t, x, y) = \pi cos(\pi t+a)sin(\pi x+b)sin(\pi y+c)+2\pi^2 sin(\pi t+a)sin(\pi x+b)sin(\pi y+c) \tag{7}$$

where a = 0.25, b = 0.5, c = 0.75.

Finally, we completed the skeleton code that was provided for the serial implementation of the model by completing the Jacobi function to solve the linear system $Gx = b$ with the backward euler iteration matrix being $G = (I - h_t A)$ and A being the Poisson operator. Our Jacobi function iterates until the general residual fell below our chosen tolerance of $1e-6$ with the general residual being $\frac{\|b-Gx^{(k)}\|}{\|b-Gx^{(0)}\|}$.

We also set a maximum iteration limit of 250 iterations so that we would have a hard limit on run time during our debugging phase. Finally, we used a small spatial grid of 8x8 to test our serial implementation and then created a convergence plot to determine if it was working as expected. We measured the convergence rate with the L2 norm of the error e and the final time T using the midpoint rule:

$$\| e \|_{L2([0,1]x[0,1])} = \left(\int_{[0,1]x[0,1]} e(T,x,y)^2 dxdy \right)^{\frac{1}{2}} \tag{8}$$

$$= h \left(\sum_{(x,y)\in((0,1),(0,1))} \left(e\left(T,x,y\right)\right)^2 \right)^{\frac{1}{2}} \tag{9}$$

This plot was compared to the plot provided in the instructions and determined to be converging quadratically as expected.

2.2 How

To find our forcing function f(t,x,y), we derived the value from the exact function as follows:

$$u_{exact}(t,x,y) = sin(\pi*t+a)*sin(\pi*x+b)*sin(\pi*y+c), a = 0.25, b = 0.5, c = 0.75 \tag{10}$$

and solved for:

$$\frac{du}{dt} = \pi cos(\pi t + a)sin(\pi x + b)sin(\pi y + c) \tag{11}$$

$$u_x = \pi cos(\pi x + b)sin(\pi t + a)sin(\pi y + c) \tag{12}$$

$$u_{xx} = -\pi^2 sin(\pi t + a)sin(\pi x + b)sin(\pi y + c) \tag{13}$$

$$u_{yy} = -\pi^2 sin(\pi t + a)sin(\pi x + b)sin(\pi y + c) \tag{14}$$

$$\frac{du}{dt} - u_{xx} - u_{yy} = \pi cos(\pi t+a)sin(\pi x+b)sin(\pi y+c) + 2\pi^2 sin(\pi t+a)sin(\pi x+b)sin(\pi y+c) \tag{15}$$

where a = 0.25, b = 0.5, c = 0.75.

With regards to the implementation of Backward Euler, the following is the function in our serial code as we implemented it:

```
def euler_backward(A, u, ht, f, g):
    I = speye(A.shape[0], format='csr')
    G = I - ht*A

    b = u + (ht*g) + (ht*f)
    tol = 1e-6
    maxiter = 250

    return jacobi(G, b, u, tol, maxiter)
```

Our implementation of the Jacobi method was as follows:

```
def jacobi(A, b, x0, tol, maxiter, n):
    D = A.diagonal()

    # compute initial residual norm
    r0 = ravel(b - A*x0)
    r0 = sqrt(dot(r0, r0))

    # Max number of iterations
    # maxiter = 250

    # Generates the identity matrix
    I = speye(A.shape[0], format='csr')

    # Generates the D inverse
    Dinv = diags(1.0/D, format='csr')

    # Generates D inverse * b
    Db = Dinv*b

    # Generates the (I-Dinv*G)
    Iterm = I - Dinv*A

    x = x0

    # Start Jacobi iterations
    for k in range(maxiter):
        x = Iterm*x + Db

        # Calculates the residual norm of the kth x
```

```
        rk = ravel(b - A*x)
        rk = sqrt(dot(rk, rk))
        #print("The current rk is ", rk)

        residual = rk/r0

        if (k == maxiter - 1 and residual > tol):
            print("Jacobi did not converge at K=", k)
        if (residual <= tol):
            #print("K: ", k)
            break

    return x
```

The following error convergence plot was generated over 4 pairs of (nt, n) values: (8, 8), (32, 16), (128, 32), (512, 64)

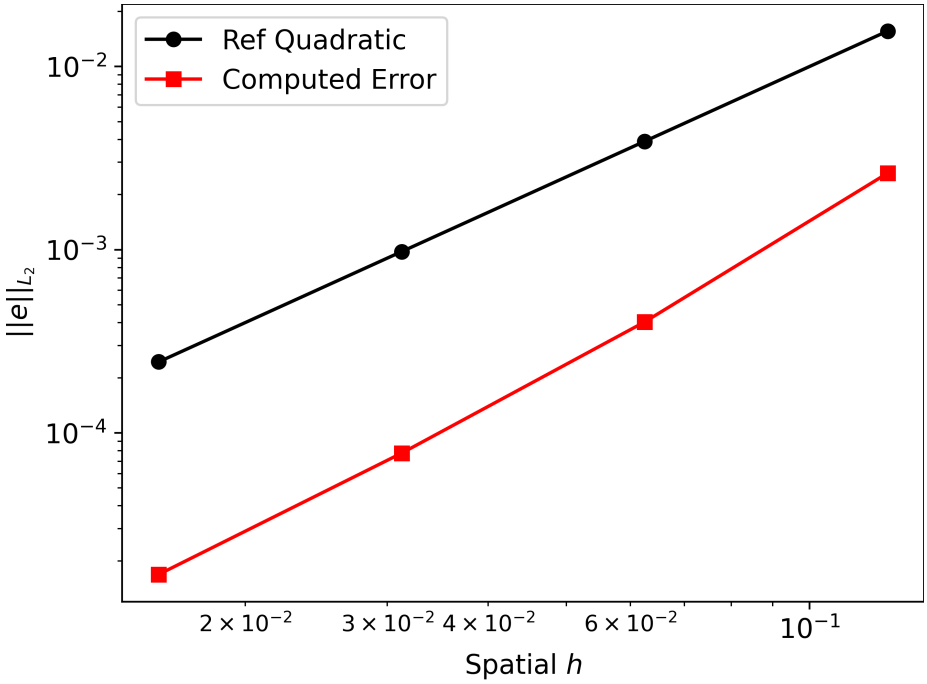

Figure 1: Plot for the serial implementation of error convergence for the spatial discretization.

2.3 Why

With regards to the convergence plot, we are plotting the error versus the decreasing spatial grid size, h. To observe the desired quadratic convergence, both

h_t and h^2 need to decrease in relation to each other. Thus, since Euler decreases at a $O(h_t)$ rate, h_t needed to decrease at a faster rate than h^2. A factor of 4 produced the expected results.

We chose our max iterations value of 250 based on the information provided by Professor Jacob Schroder in his own version of the code. Depending on the problem, our Jacobi solves took 150-225 iterations. We arbitrarily chose a tolerance of $1e^{-6}$ because it provided a decent degree of accuracy without having to worry about getting close to the machine precision. We chose the initial guess for the Jacobi method to be the solution to the previous time step. This is sufficient as long as h_t is small.

It can clearly be seen in Figure 1 above that the observed convergence rate of the error is on the order of $\mathcal{O}(h^2)$

3 Task 2: Parallel Implementation

3.1 What

This task asked us to implement that same model in a parallel manner utilizing MPI so that it could be run on the CARC supercomputer. This meant discretizing and agglomerating the problem into smaller problems that could be tackled by various processors, using MPI to pass the values of the halo regions and x values back and forth before completing the calculation, and then paring off the repeat values from the answer.

After we got the parallel implementation working we ran it against the same 8x8 spatial grid problem size as the serial version for 1, 2, and 4 tasks and got the same answer as the serial version which was expected.

3.2 How

To parallelize our code, we started by importing MPI from mpi4py and initializing the comm, getRank, and numRanks values that would be used throughout the program. Next, we modified the general loop of the program to include the calculations for the halo regions, adjusted the size of A to only be part of a given task, and adjusted the size of the solution matrix u so that it was only storing the values from the problem given to a specific task. The bulk of the changes were made in the Jacobi function, which contained all of the MPI commands. This included using the MPI send and receive commands to pass the initial values of x from each task so we could compute the initial local, and then global residual. It also included MPI instructions being added to the main Jacobi loop to send and receive the values needed by each task to process the x matrix. The following is our updated Jacobi function for the parallelized code:

```
def jacobi(A, b, x0, tol, maxiter, n):
    # This useful function returns an array containing diag(A)
    D = A.diagonal()

    # compute initial residual norm
    if (numranks != 1):
         if (rank == 0):
             comm.Send([x0[-(2*n):-n], MPI.DOUBLE], dest=1, tag=77)
         elif (rank == (numranks - 1)):
             comm.Send([x0[n:2*n], MPI.DOUBLE], dest=rank-1, tag=77)
         else:
             comm.Send([x0[-(2*n):-n], MPI.DOUBLE], dest=rank+1, tag=77)
             comm.Send([x0[n:2*n], MPI.DOUBLE], dest=rank-1, tag=77)

         if (rank == 0):
             comm.Recv([x0[-n:], MPI.DOUBLE], source=rank+1, tag=77)
         elif (rank == (numranks - 1)):
             comm.Recv([x0[0:n], MPI.DOUBLE], source=rank-1, tag=77)
         else:
             comm.Recv([x0[0:n], MPI.DOUBLE], source=rank-1, tag=77)
             comm.Recv([x0[-n:], MPI.DOUBLE], source=rank+1, tag=77)
    sol0 = A*x0

    r0 = ravel(b - sol0)
    if (rank == 0):
        if (numranks != 1):
            r0 = r0[0:-n]
        else:
            r0 = r0
    elif (rank == (numranks - 1)):
        if (numranks != 1):
            r0 = r0[n:]
    else:
        r0 = r0[n:-n]

    r0r0 = array([dot(r0, r0)])
    globalr0r0 = array([0.0])
    comm.Allreduce(r0r0, globalr0r0, op=MPI.SUM)
    globalr0 = array([0.0])
    globalr0[0] = sqrt(globalr0r0[0])

    # Max number of iterations
    # maxiter = 250

    # Generates the identity matrix
    I = speye(A.shape[0], format='csr')
```

```
# Generates the D inverse
Dinv = diags(1.0/D, format='csr')

# Generates D inverse * b
Db = Dinv*b

# Generates the (I-Dinv*G)
Iterm = I - Dinv*A

x = x0

# Start Jacobi iterations
for k in range(maxiter):
    #print("rank ", rank, "x befOre ", x)
    if (numranks != 1):
        if (rank == 0):
            comm.Send([x[-(2*n):-n], MPI.DOUBLE], dest=1, tag=77)
        elif (rank == (numranks - 1)):
            comm.Send([x[n:2*n], MPI.DOUBLE], dest=rank-1, tag=77)
        else:
            comm.Send([x[-(2*n):-n], MPI.DOUBLE], dest=rank+1, tag=77)
            comm.Send([x[n:2*n], MPI.DOUBLE], dest=rank-1, tag=77)

        if (rank == 0):
            comm.Recv([x[-n:], MPI.DOUBLE], source=rank+1, tag=77)
        elif (rank == (numranks - 1)):
            comm.Recv([x[0:n], MPI.DOUBLE], source=rank-1, tag=77)
        else:
            comm.Recv([x[0:n], MPI.DOUBLE], source=rank-1, tag=77)
            comm.Recv([x[-n:], MPI.DOUBLE], source=rank+1, tag=77)


    x = Iterm*x + Db

    if (numranks != 1):
        if (rank == 0):
            comm.Send([x[-(2*n):-n], MPI.DOUBLE], dest=1, tag=77)
        elif (rank == (numranks - 1)):
            comm.Send([x[n:2*n], MPI.DOUBLE], dest=rank-1, tag=77)
        else:
            comm.Send([x[-(2*n):-n], MPI.DOUBLE], dest=rank+1, tag=77)
            comm.Send([x[n:2*n], MPI.DOUBLE], dest=rank-1, tag=77)

        if (rank == 0):
            comm.Recv([x[-n:], MPI.DOUBLE], source=rank+1, tag=77)
```

```
            elif (rank == (numranks - 1)):
                comm.Recv([x[0:n], MPI.DOUBLE], source=rank-1, tag=77)
            else:
                comm.Recv([x[0:n], MPI.DOUBLE], source=rank-1, tag=77)
                comm.Recv([x[-n:], MPI.DOUBLE], source=rank+1, tag=77)

        sol = A*x

        rk = ravel(b - sol)
        if (rank == 0):
            if (numranks != 1):
                local_rk = rk[0:-n]
            else:
                local_rk = rk
        elif (rank == (numranks - 1)):
            if (numranks != 1):
                local_rk = rk[n:]
        else:
            local_rk = rk[n:-n]

        rkrk = array([dot(local_rk, local_rk)])
        globalrkrk = array([0.0])
        comm.Allreduce(rkrk, globalrkrk, op=MPI.SUM)
        globalrk = array([0.0])
        globalrk[0] = sqrt(globalrkrk[0])

        residual = (globalrk[0])/(globalr0[0])

        if (k == maxiter - 1 and residual > tol):
            #print('rkrk  ', rkrk)
            #print("local rk  ", local_rk)
            #print("b ", b, " sol ", sol)
            if (rank == 0):
                print("Jacobi did not converge at K=", k)
        if (residual <= tol):
            if (rank == 0):
                print("Jacobi converged at K: ", k)
            #print("rkrk ", rkrk)
            #print("local rk  ", local_rk)
            #print("b ", b, " sol ", sol)
            break

    return x
```

3.3 Why

Our parallel version of our code did not see any change in convergence for Jacobi from serial to parallel as expected. Figure 1 and Figure 2 both show our error convergence for the serial and parallel version of our code and are almost identical.

4 Task 3: Scaling Studies

4.1 What

First, we were asked to demonstrate that our MPI parallelization produced the same result, with the same order of accuracy, as our serial implementation featured in Task 1. This involved plotting the convergence of our error norm versus spatial grid sizes.

Next, we were asked to plot the initial condition, approximate final solution at time T, and exact solution using MPI in order to show that our distributed solution can be recombined in order to produce a visualization of the entire solution.

In this section, we were then tasked with performing two scaling studies on the CARC supercomputer using our parallelized code from task 2. One was a strong scaling study where the problem size was fixed and various quantities of MPI processes were used. The second was a weak scaling study where the problem size and number of MPI processes increased commensurately.

4.2 How

First, we implemented a test study on CARC using 1 node and 8 processors over all problem sizes to ensure that we were getting the same convergence rates as was seen in Task 1's serial version of the code. A global sum of each rank's portion of the error was done via MPI. The pbs script for this CARC job named hw6_test.pbs can be found in our course repository. A sample plot using 8 MPI processes on one Wheeler node is below:

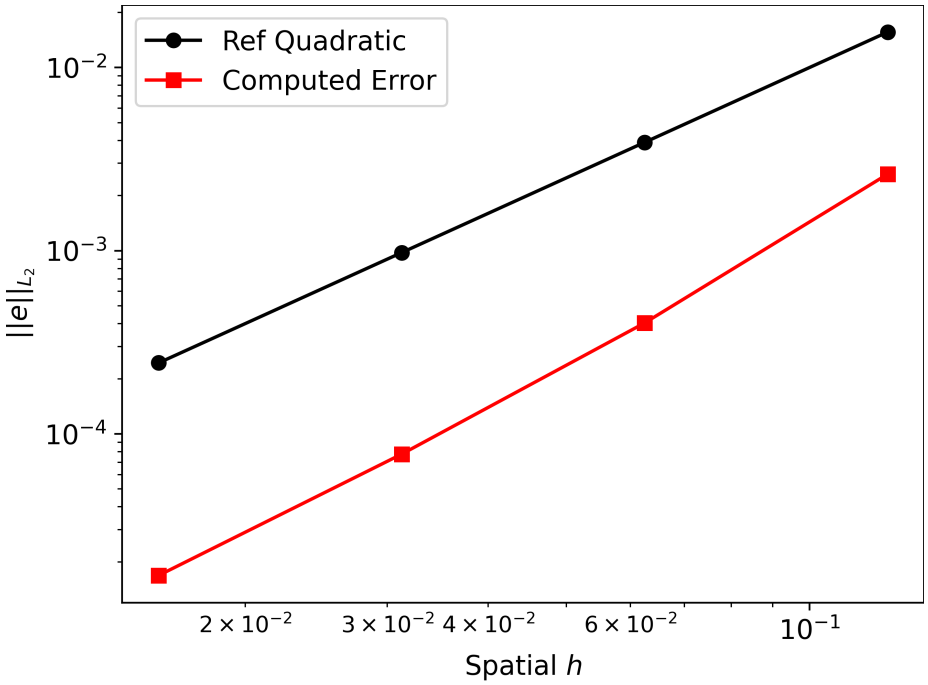

Figure 2: Plot for the parallel 8 process implementation of error convergence for the spatial discretization.

Note: similar plots for convergence of the error for 2 and 4 MPI processes can be found in our course repository.

We carried out a weak scaling study by determining the spatial grid size, the time step size, the number of processors being used and then using the T = 0.03 provided in the skeleton code. For the weak scaling study, we wanted to maintain a good h_t/h^2 ratio of 4.0 which helped to give us the n and nt values we needed. From there, we used those values to determine the processor sizes and ended up running on four sizes, 1, 4, 16, and 64 processors using their respective values: (nt, n) = (16, 48), (69, 96), (276, 192), (1106, 384). The Unix time command was used to time each run 5 times in order to obtain an average wall time. These were repeated five times each in our pbs script and provided the following results:

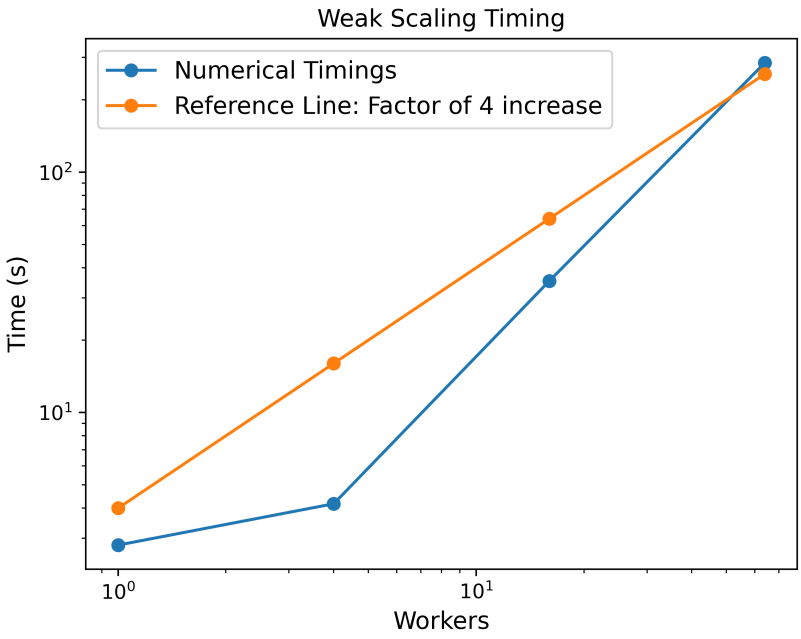

Figure 3: Plot for the weak scaling study compared against a reference line that is increasing by a factor of 4

Finally, we carried out a strong scaling study by using 1024 time steps and a spatial grid size of 512x512 for processor sizes 2, 4, 8, 16, 32, and 64 with an end time of 0.01555. The file was run 5 times for each number of processors and resulted in the following plot:

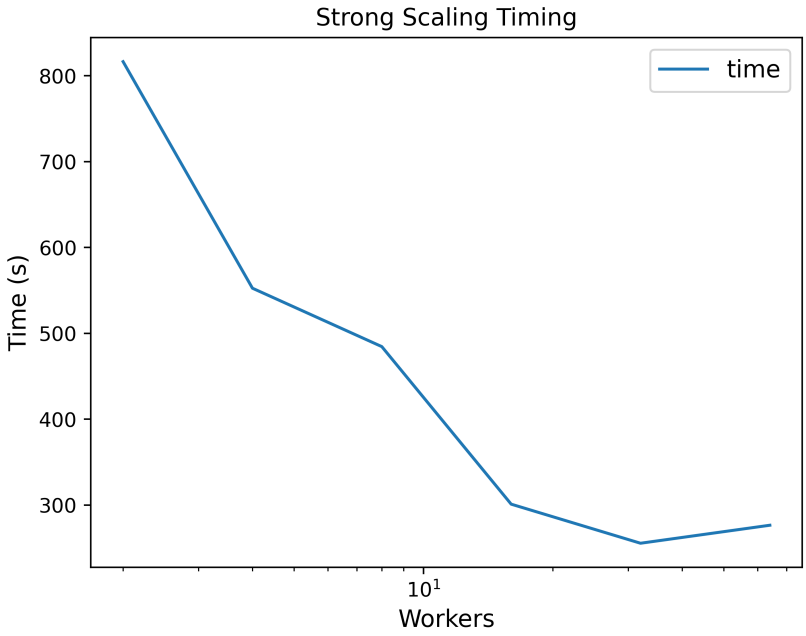

Figure 4: Plot for the strong scaling study

We were further asked to speculate on the stagnation of the decreases in problem time with more processors in the strong scaling study and to examine the weak scaling results.

Next, a plot of a general solution at the final time T was generated. This was to illustrate how the solution evolved over time. The MPI gather() function was used to collect the local portions of each rank's solution into a global solution array on rank 0's MPI process as follows (contributions are ordered by MPI rank index):

```
if (rank == 0):
    if (numranks != 1):
        local_u0 = u[0,0:-n]
        local_uf = u[nt-1,0:-n]
        local_uef = ue[nt-1,0:-n]
    else:
        local_u0 = u[0]
```

```
        local_uf = u[nt-1]
        local_uef = ue[nt-1]
elif (rank == (numranks - 1)):
    if (numranks != 1):
        local_u0 = u[0,n:]
        local_uf = u[nt-1,n:]
        local_uef = ue[nt-1,n:]
else:
    local_u0 = u[0,n:-n]
    local_uf = u[nt-1,n:-n]
    local_uef = ue[nt-1,n:-n]

recvu0arr = comm.gather(local_u0, root=0)
recvufarr = comm.gather(local_uf, root=0)
recvuefarr = comm.gather(local_uef, root=0)

if (rank == 0):

    globalu0 = concatenate(recvu0arr, axis=0)
    globaluf = concatenate(recvufarr, axis=0)
    globaluef = concatenate(recvuefarr, axis=0)
```

Standard matplotlib plotting code was used to plot globalu0 (the initial condition), globaluf (the approximate solution at time T), and globaluef (the exact solution at time T). A sample set of plots for a problem size of $(Nt, N, T) = (512, 64, 0.5)$ can be seen below:

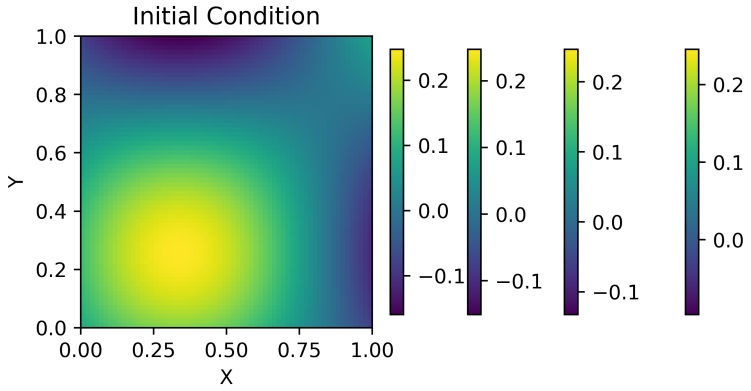

Figure 5: Plot for the initial condition at time $t = 0$ using 8 MPI processes.

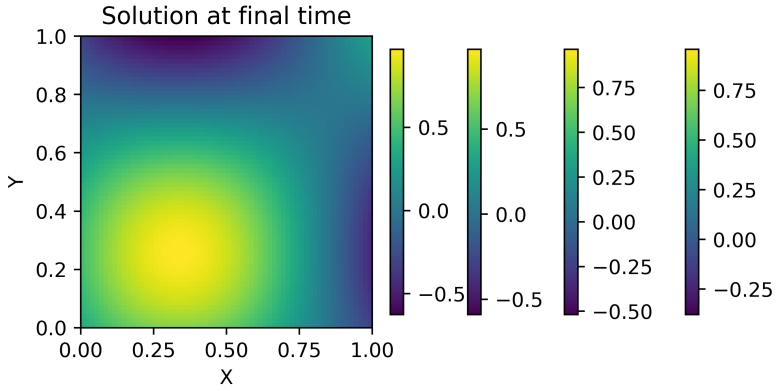

Figure 6: Plot for the approximate solution at time $t = T$ using 8 MPI processes.

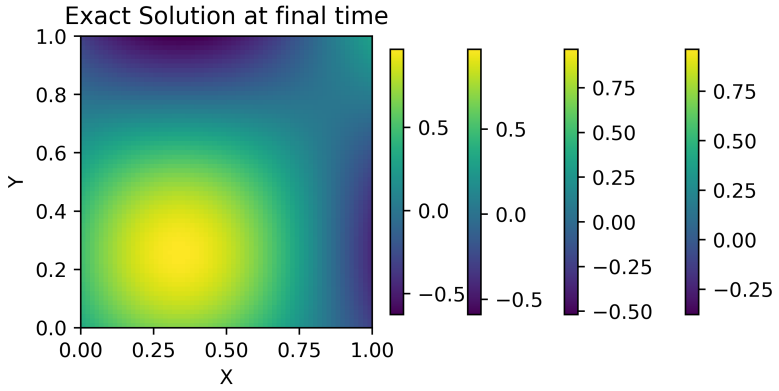

Figure 7: Plot for the approximate solution at time $t = T$ using 8 MPI processes.

The three figures above were generated by a run of 8 MPI processes on one Wheeler node. Exact duplicates of these plots for 1, 2, 4, 8 MPI process counts can be found in the LoboGit repository.

4.3 Why

By examining Figures 1 and 2, it is observed that our MPI parallelization functions properly. The error is converging at a rate of $\mathcal{O}(h^2)$ as the computed error remains parallel to the reference quadratic slope. This agrees with the theoretical error bound of the finite difference method.

In Figure 4, we can see the results from the strong scaling study. As expected, the parallelization never achieves optimal efficiency. The timings decrease until using 64 processes instead of 64. This increase is most likely due to the fact of increased numbers of communication between processes. Since a 1-D domain decomposition was used, the number of communications for the halo regions increase. Communication over the network is orders of magnitude slower than communication between the processes within a node. Thus, since 32 processes uses 4 Wheeler nodes and 64 processes use 8 Wheeler nodes, two times the amount of extra slow network communication was needed.

Since the strong scaling problem size has a $\frac{800}{540} \approx 1.48$ speedup when using 2 to 4 processes, 1.48 will likely be a reasonable upper bound on the speedup seen from 1 to 2 processes since efficiency tends to decrease as the number of parallel workers increases. Using Amdahl's Law,

$$S_{1->2} = \frac{T_s}{T_2} \tag{16}$$

$$1.48 \geq \frac{T_s}{800s} \tag{17}$$

$$T_s \leq (800s)\, 1.48 \tag{18}$$

$$T_s \leq 1184s \tag{19}$$

we conclude that T_s for our strong scaling problem size is near 1184 seconds, but, in theory, could be as low as 800 seconds. T_s is the serial runtime for this problem size. This is a crucial parameter in parallel computing.. Computational efficiency is defined as the measure of how much speedup we get when compared to the optimal case where there is no overhead. It is defined in terms of T_s as: $\epsilon = \frac{T_s}{NT_p}$, where T_p is the time taken to solve the specified problem with N parallel workers.

As seen in Figure 3, timings increase as the process counts increase for 4, 16, and 64 processes. In the ideal case, the timings remain constant as the number of processes increase, so computational overhead, such as communication between processes, is driving the results from this study. It is apparent, though, that from 1 to 4 processes, overhead is minimal since the plot is a flat line in that range.

It can clearly be seen from Figures 5-7 that the solution range to the heat equation, given our defined forcing function $f(t, x, y)$, is a translated circle. This makes sense as we added b and c to πx and πy within the two respective sine functions given in the assignment handout.

It should also be noted that when comparing the solution at times 0 and T, it can be seen that the values within the circle increased in intensity, while the surrounding dark regions become even less intense.

5 Acknowledgements

We would like to thank the UNM Center for Advanced Research Computing, supported in part by the National Science Foundation, for providing the research computing resources used in this work.