HW #6
Math/CS 471, Fall 2021

Michael Sands
John Tran

December 15, 2021

Contents

1 Problem to be Solved

In this project, we design a program to solve an ordinary differential equation, specifically the heat equation, using the Message Passing Interface (MPI) and backwards Euler's method. We demonstrate that the parallelized version of our code produces identical results to the serial version of our code, and we present the parallel speedup or our code through weak and strong scaling experiments.

2 Serial Code:

In this project, we modeled the second derivative of Equation 1. We chose this u_{exact} because it has non-zero boundary conditions. Analytically, this second derivative is shown in Equation 2. The resulting Dirichlet boundary condition function is presented in Equation 6 where Omega is defined through Equation 4 and its partial derivative is defined through Equation 5. We derived Equation 2 through the heat equation in Equation 3.

$$u_{exact}(t,x,y) = cos(\pi t)cos(\pi x)cos(\pi y) \tag{1}$$

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

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + f(t,x,y), \quad x,y \in \Omega \tag{3}$$

$$\Omega := (x,y)\ such\ that\ x \in [0,1], y \in [0,1] \tag{4}$$

$$\partial\Omega := (x,y)\ such\ that\ x = 0,1, y = 0,1 \tag{5}$$

$$u(t,x,y) = u_{exact}(t,x,y) = g(t,x,y)\ such\ that\ (x,y) \in \partial\Omega \tag{6}$$

In our heat equation experiment, we used the Jacobi method for inverting the matrix in backwards Euler. We used Equation 7, the ratio of the final and initial residuals, to check the convergence of each step. We also set a maximum iteration limit of 400 in case Jacobi fails to converge. In our

experiments, we needed 293 iterations for Jacobi to converge to a discretization error with tol $= 1 * 10^{-10}$. As shown below, this value of the tolerance produced a quadratic convergence plot similar to Figure 1 in the homework pdf.

$$\frac{\|b - Gx^{(k)}\|}{\|b - Gx^{(0)}\|} \leq tol \tag{7}$$

2.1 Initial Guess:

In our code, we took our initial guess at time = 0 to be the exact solution in Equation 8 at time = 0. From here, backwards Euler via the Jacobi method modeled the function until the final time T.

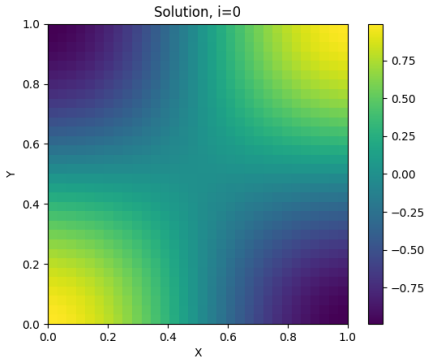
$$u(t = 0, x, y) = u_{exact}(t = 0, x, y) \tag{8}$$

2.2 Effect of Final Time:

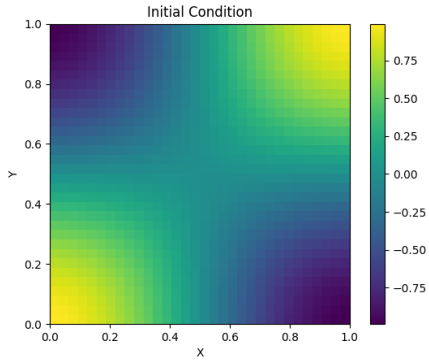
We note that the specific function we chose for this project is non-zero at the boundary conditions, allowing for a more interesting fitting, but it also sees a sharp drop off at time = 0.5. To illustrate the ability of our code to fit the chosen equation, we illustrate its output at T = 0.5, where the final image is numerically zero, and at T = 0.75 where the final image is on a much larger scale.

2.2.1 T = 0.5

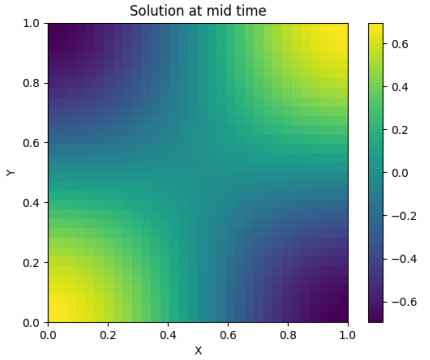
At T = 0.5, in Equation 1, the exact solution that we are modeling, goes to zero. This is also apparent from the last frames of Figure 1 where the scale of our solution in Figure 1e dramatically decreases with the exact solution in Figure 1f.



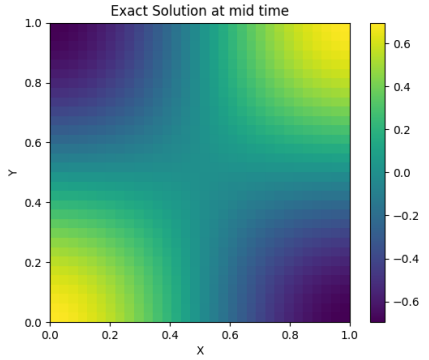
(a) i = 0



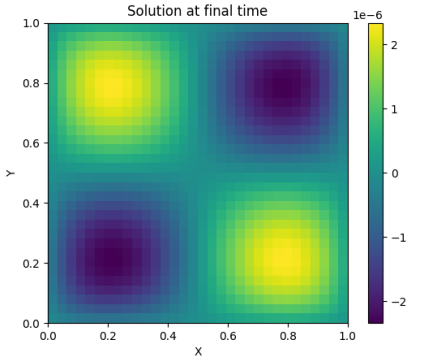
(b) Initial Condition



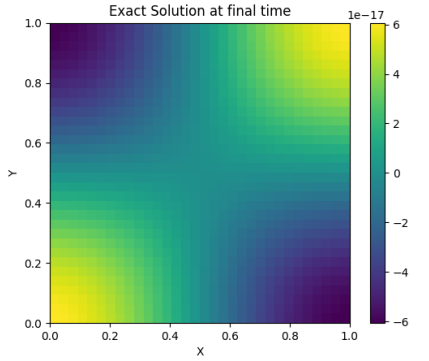
(c) Middle Time Solution



(d) Middle Time Exact



(e) Final Time Solution



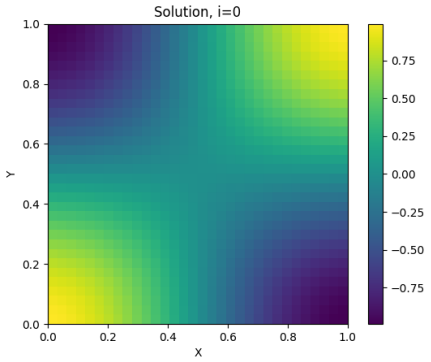
(f) Final Time Exact

Figure 1: Example plot for $T = 0.5$

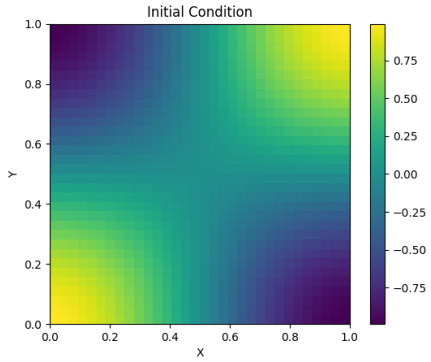
2.2.2 T = 0.75

At T = 0.75, we observe similar results in Figure 2. However, in this case, the scales of the final plots line up much more clearly. Because the cosine of 0.75π is a negative value, the the final time plots in Figure 2 appear to be mirrored compared to the rest.

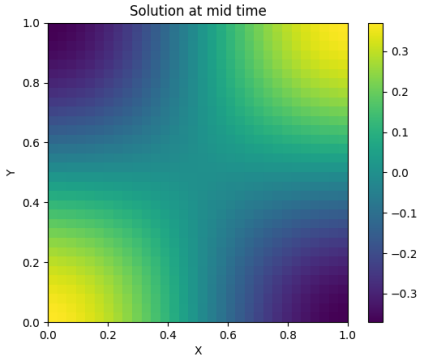
Note: We found the the discrepancy between the two times could be due to the behavior of our u_{exact} near the final time T since the value of u_{exact} should be numerically zero (or close to it). The t term in u_{exact} should evaluate to zero ($cos(0.5 * \pi) = 0$). Since we used the guess for the next time iteration $i + 1$ as the result u from the previous iteration i where u_i is used as the guess for u_{i+1}, the rate of convergence of our solution will be dependent on our *choice of final time* T and *the number of time steps* n_t. The way u_{exact} changes to the final time T from the previous time iteration changes quite dramatically (well, any nonzero value that suddenly drops to 0 would result in an inexact solution since a definite tolerance is used). This means we can only get arbitrarily close to 0 *relative* to our guess (previous iteration result), and with a smaller ht (bigger value of n_t), the guess used for our final time iteration at $t = T$ will be that much closer to 0 (as observed in Figure 6e).



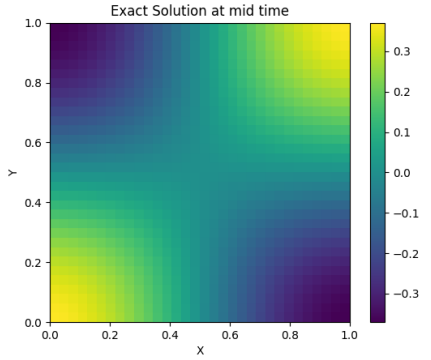
(a) i = 0



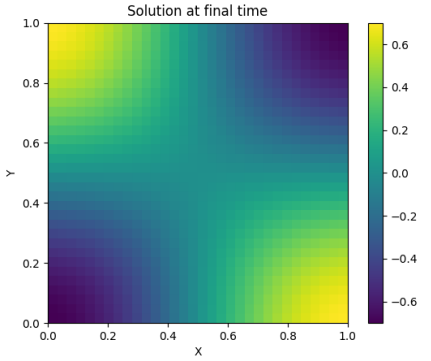
(b) Initial Condition



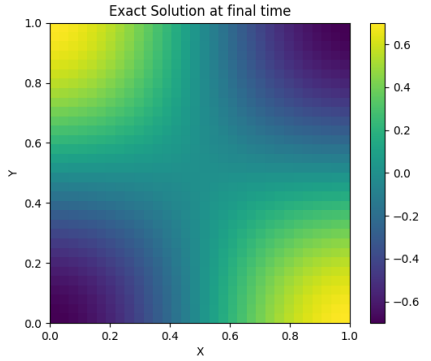
(c) Middle Time Solution



(d) Middle Time Exact



(e) Final Time Solution



(f) Final Time Exact

Figure 2: Example plot for $T = 0.75$

2.3 Error

$$\|e\|_{L_2([0,1])\times[0,1]} = \left(\int_{[0,1])\times[0,1]} e(T,x,y)^2 dxdy)\right)^{1/2} \tag{9}$$

In order to confirm that our code is functioning correctly, we used the L_2 norm, Equation 9, to plot the convergence rate. We would expect to see a quadratic convergence if our code works correctly, and in Figures 3 and 4, that is exactly what we observe. Note that in Figure 3, we observe a slight bowing of the computed error. As our solution is numerical, this bowing is not unexpected and does not disprove the correctness of our code. Also, we noted that the error is shifted downwards for Figure 3 compared to Figure 4. This is largely due to the fact that both plots use the same value of h over a different length of the time. Therefor the spacial h is relatively larger in the case of T = 0.75 and therefore the error is not as small.

For h and ht, because the rate of convergence for h is $O(h^2)$ and the rate of convergence for h_t is $O(h_t)$, h_t must be reduced faster than h to observe the quadratic convergence. To do this, we decreased h_t by a factor or 4 where h was decreased by a factor of 2 for each next point. This kept the ratio of h/h_t constant.

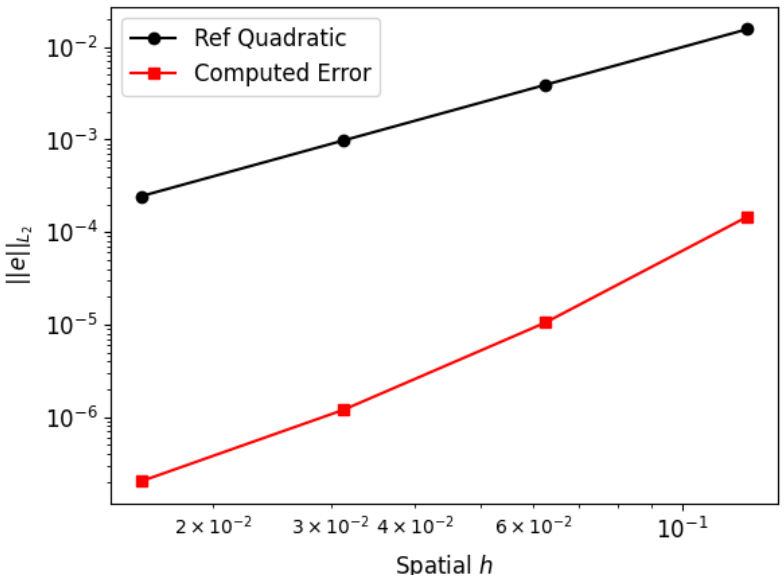


Figure 3: Quadratic Reference Curve: T = 0.5

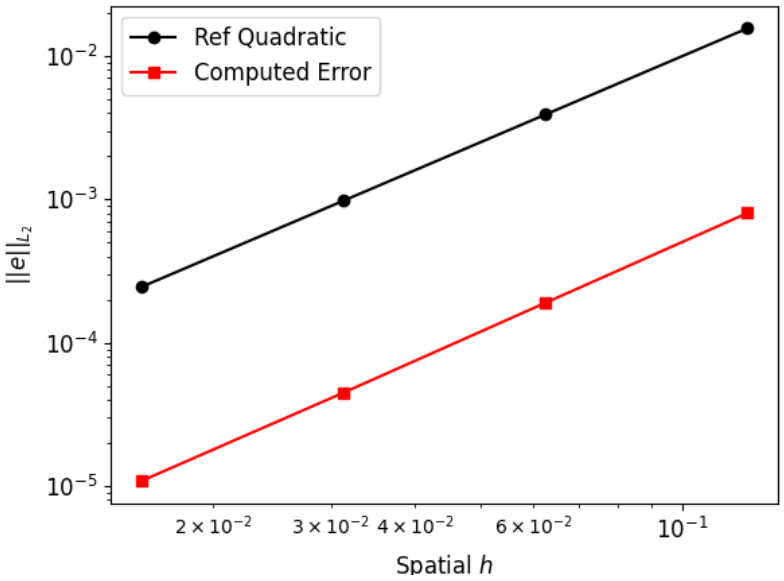


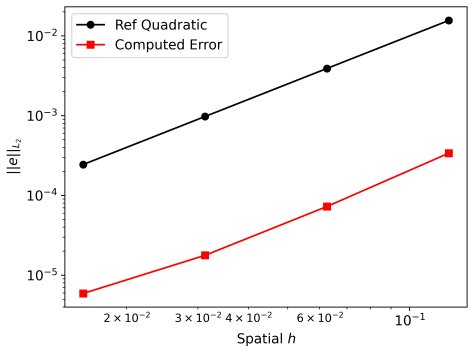
Figure 4: Quadratic Reference Curve: T = 0.75

2.4 Effect of Tolerance:

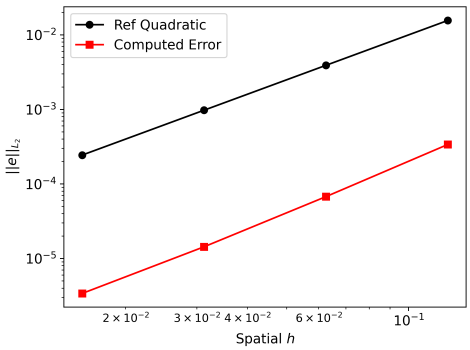
In our serial and parallel studies, we noted that our code, on average, took longer to run than the professors in the sample plots provided. We attribute this to a difference in the tolerance used. In our initial serial and parallel studies, we used a tolerance of $1 * 10^{-15}$. However, as is shown in Figure 5, quadratic convergence is still observed at lower values of the tolerance. We noted a significant speedup of our code using a tolerance of $1*10^{-10}$ compared to a tolerance of $1 * 10^{-15}$. Both of these tolerances are observed to produce the quadratic convergence expected, so we are not concerned about a loss of accuracy from the new tolerance.

We note that quadratic convergence appears to be lost at a tolerance of $1 * 10^{-4}$ where in Figure 5a, there is a slight kink in the leftmost data point. For this reason, we conclude that a tolerance of $1*10^{-10}$ is more than sufficient to preserve quadratic convergence while maintaining a reasonable solution time.

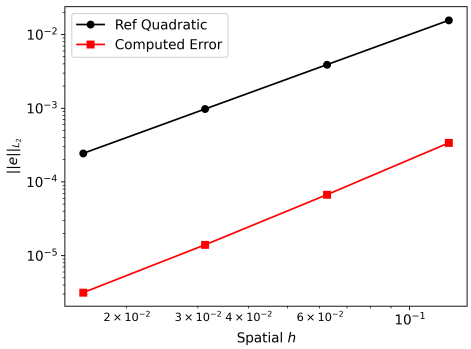
Note: The tolerances used for strong and weak scaling were also tested independently as they are more computationally expensive.



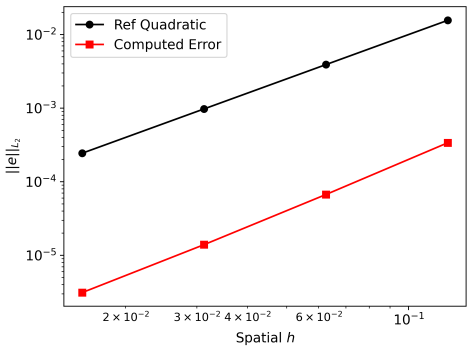
(a) tol $= 1 * 10^{-4}$



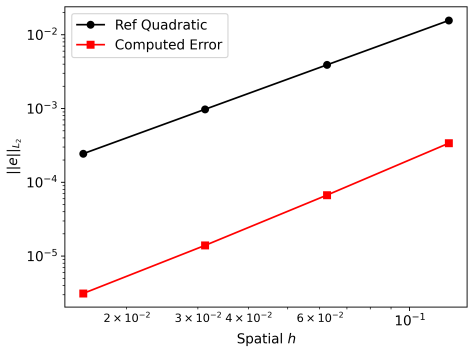
(b) tol $= 1 * 10^{-5}$



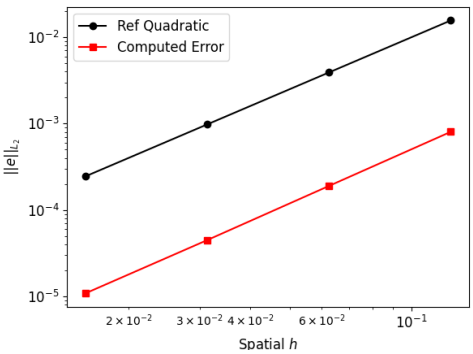
(c) tol $= 1 * 10^{-6}$



(d) tol $= 1 * 10^{-8}$



(e) tol $= 1 * 10^{-10}$



(f) tol $= 1 * 10^{-15}$

Figure 5: Illustration of effect of tol on the observed quadratic convergence

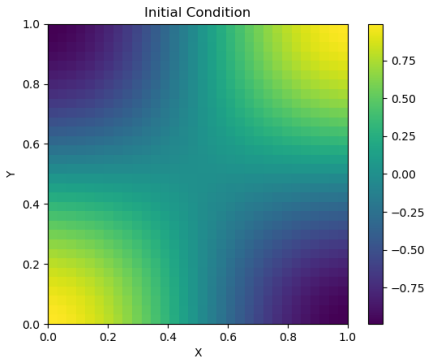
3 Parallel Code

3.1 Parallel Mat-Vec

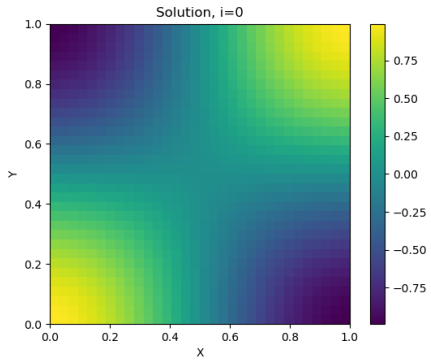
After confirming the quadratic convergence of our serial code, we parallelized the Jacobi method inside of backwards Euler. As is noted below and as we would expect, the rate of convergence is not altered by the parallelization of the Jacobi method.

Looking at the results of the check_matvec(), we note that the function returns 0 for interior points, -1 for points on the walls, and -2 for corner points. These values were expected and observed in the output of our code. Because the corner points are on the first and last processor, we expected to see all -2 values on processors 0 and nprocs - 1. We did observe this to be the case. We also expected that all non -2 values on processors 0 and nproxs - 1 should be -1 as they are all wall points. This was also observed in our code. Finally, we expected that all other processors should have two -1 values and the rest of the values in each of these processors should be 0 as they are interior points. This was also observed in our code.

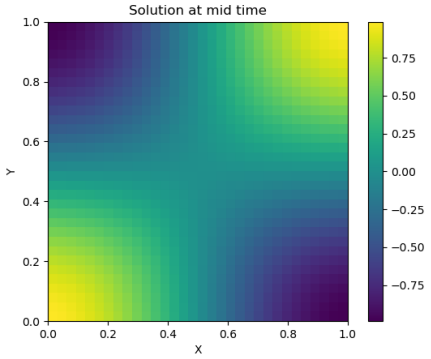
3.2 Comparison to Serial Results



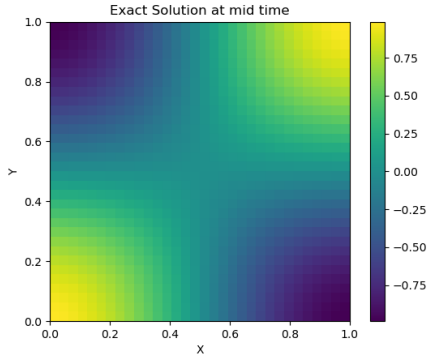
(a) i = 0



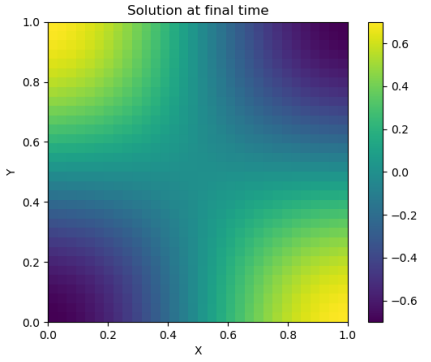
(b) Initial Condition



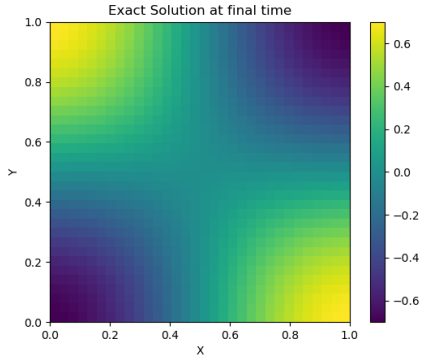
(c) Middle Time Solution



(d) Middle Time Exact



(e) Final Time Solution



(f) Final Time Exact

Figure 6: Example plot for $T = 0.75$ in Parallel

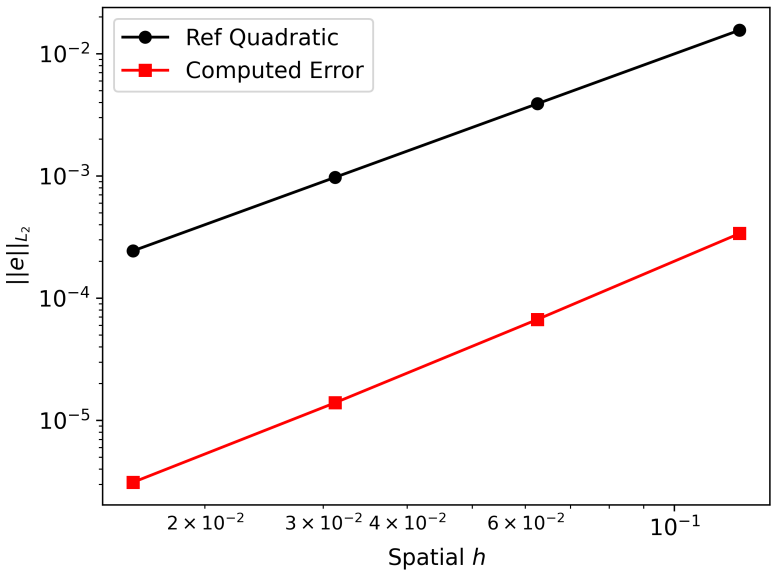


Figure 7: Quadratic Convergence Plot from the Parallel Code

The quadratic convergence plot for the parallel code is almost identical to the quadratic convergence plot for the serial code at T = 0.75. We note that the error is shifted slightly downwards in the case of the parallel code. Because the final time is the same, the h for both convergence plots are also the same. This means that the shifting is most likely due to a difference in the tolerance used for each case. In the serial code, we used a tolerance of $1 * 10^{-15}$ where as in the parallel code, we used a tolerance of $1 * 10^{-10}$. In Figure 5, these figures are exactly what we would expect given their respective tolerances. In both cases, we observe a quadratic reduction in error with the reduction of h, so we are confident that our code works the same for both the serial and parallel case.

3.3 Scaling

3.3.1 Weak Scaling

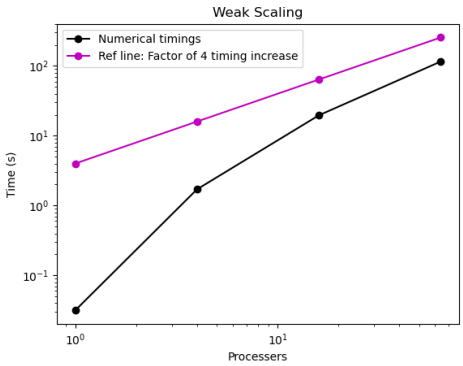
For this strong scaling experiment, we used the following variables:

Problem sizes = $[48, 48 * 2, 48 * 2^2, 48 * 2^3]$

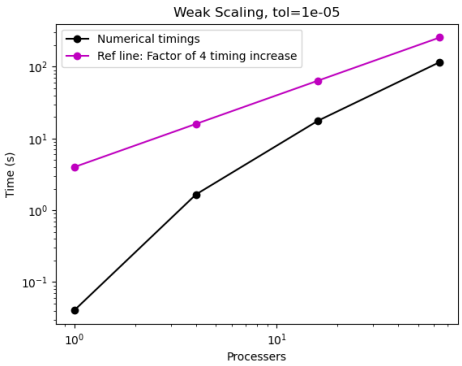
Processor counts = $[1, 4, 16, 64]$
Final time = $1/36$

NOTE: In order to maintain the h_t/h ratio, we varied n and n_t commensurately in order to maintain a constant final time T. n changed by a factor of 2 for each test case, and n_t changed by a factor of 4. The number of processors should change proportionally to n_t so that when n_t reduces by a factor of 4, so does the number of processors. We checked this with the professor, and he mentioned the effect is the same and that this deviation from the directions was OK.

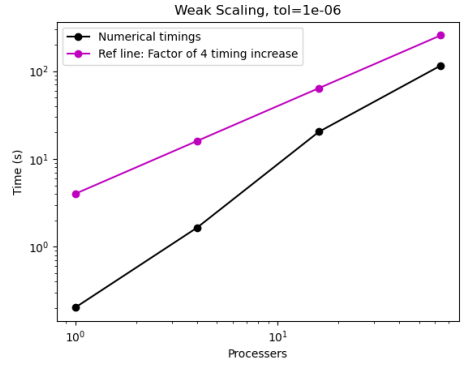
Weak scaling measures the change in solution time with a changing number of processors while the problem size per processor is held constant. For this weak scaling study, we used a problem size of about 2000, specifically 48^2 grid points per processor. The results of our weak scaling experiment can be seen in Figure 8. We note that our timings increase slightly more than the linear reference curve in the case of all tolerances tested. This is largely due to added overhead from each additional processor in the experiment. In this weak scaling experiment, we also tested tolerance values of, $1*10^{-4}$, $1*10^{-5}$, $1*10^{-6}$, $1*10^{-8}$, $1*10^{-10}$, and $1*10^{-15}$ for the Jacobi method. We note that our plots best match the plots provided in the homework pdf for tolerance values of $1 * 10^{-10}$ and $1 * 10^{-15}$.



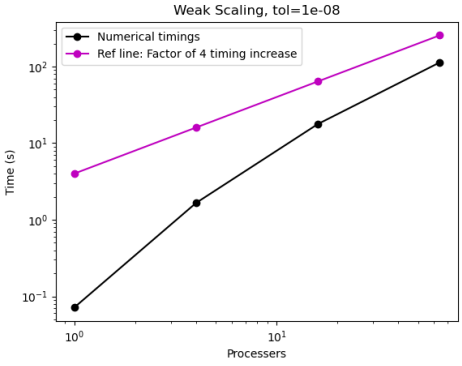
(a) tol $= 1 * 10^{-4}$



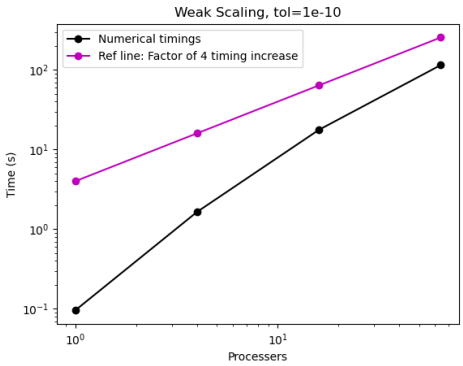
(b) tol $= 1 * 10^{-5}$



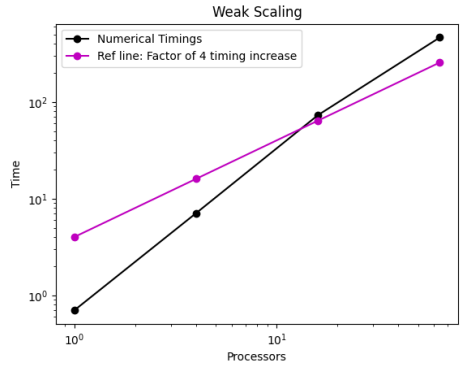
(c) tol $= 1 * 10^{-6}$



(d) tol $= 1 * 10^{-8}$



(e) tol $= 1 * 10^{-10}$



(f) tol $= 1 * 10^{-15}$

Figure 8: Illustration of effect of tol on Weak Scaling

We note that the shape of the weak scaling plots varies over the different values of tolerance. Also, the timing scale also varies with different tolerances. This was expected as for reduced tolerance, less time will be needed for each iteration, effectively shifting the weak scaling curve downwards.

3.3.2 Strong Scaling

For this strong scaling experiment, we used the following variables:

Problem size = 512
Processor count = $[2, 4, 8, 16, 32, 64]$
Final time = 1/64

Strong scaling measures the change in solution time with a fixed problem size and variable numbers of processors. In this project, we used 1024 time steps ($nt = 1024$) and a 512×512 grid in space ($n = 512$) to produce Figure 9. We note that the strong scaling plot does not uniformly decrease. However, our strong scaling plot does not uniformly decrease in the same way as the sample plot in the homework pdf. We would expect that in the last few data points, the strong scaling plot should level off and go up. The leveling off is due to to the overhead from each processor being comparable to the time benefit from adding the processor, ie the leveling off is due to the parallel limit being reached and the remaining time cost being dominated by a serial portion of code. As the overhead from the added processors becomes more notable, the solution time may go up in the end. However, in our figures, we observed an odd trend in the middle points of our strong scaling experiment. This is most likely due to buffer size issues that we observed in our MPI tests. Because the problem size per processor is so small at 64 processors, we did not observe the same buffering issue in the strong scaling plot for the last data point. We can observe from this plot that there is about a 2x speedup from the use of of higher numbers of processors by looking at the difference between the first and last points in the figure (we presume the last data point is where the figure would level off from the data that we were able to obtain).

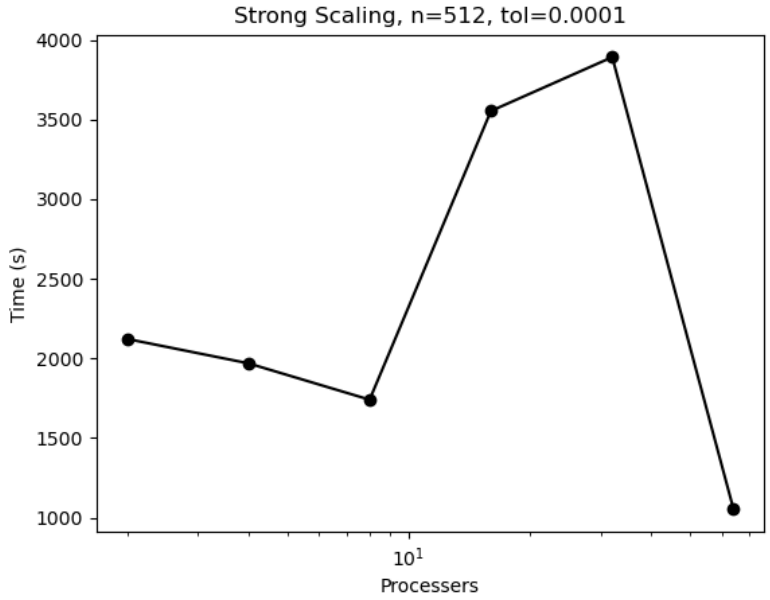


Figure 9: Strong Scaling, tol = 0.0001, 512 Grid Size

3.3.3 Serial Time:

From Amdahl's law and Figure 9, we can derive the serial time, T_s. This relates to the scalability limits because a portion of the code is not parallelizable. So, for a fixed problem size, eventually the benefit of increasing the number of processors does not outweigh the cost of the overhead from the additional processor. We calculate the serial time below.

$$T_p = (1-P)T_s + P\frac{T_s}{N_p} = T_s((1-P) + \frac{P}{N_p}) \tag{10}$$

$$\frac{T_s}{T_p} = \frac{T_s}{T_s((1-P) + \frac{P}{N_p})} = \frac{1}{(1-P) + \frac{P}{N_p}} \tag{11}$$

In this equation, P is the portion of the program that can be parallelized, T_s is the time required to run the program serially. N_p is the number of processors available. And, T_p is the time required to execute a program in parallel.

At an ideal speedup, this solution can be written as:

$$N_p = \frac{T_s}{T_p} \tag{12}$$

From Equation 12, we can solve for T_s (T_1 in the homework pdf) by using the portion of the graph where the timings begin to level off, and the initial timing of the figure in serial. In the homework pdf, the sample curve began to level off at 16 to 64 processes. However, with our unusual plot, we decided to use 64 processes as that resulted in about a two times speedup, as mentioned above. From our timing for 64 processes, we found it be 1053 seconds for the overall execution time (rounded to the nearest second). We found the serial time, T_s, to be $N_p \cdot T_p = 64 \cdot 1053 = 67392$ s.

4 Extra Credit:

4.1 Strong Scaling 128 x 128 Grid:

To better visualize the cost of scaling on small problem sizes, we repeated our strong scaling test on a 128 x 128 (Figure 10) grid instead of a 512 x 512 grid. The general shape of the figure is the same. We observe that there are a few outlier points. From what we observe for 16 and 32 processes, the sudden spike in overall time could be attributed to the increased communication costs. As a refresher, we used the capital `MPI` commands for send and receive (`MPI.Send()` and `MPI.Receive()`) which used a buffer instead of generic `Python` objects. As a result, the increased amount of data that was communicated (i.e., the $n = 512$ case and to a certain extent for the $n = 128$ case) contributed more to the overall runtime. For our implementation, we had to break up the send and receive messages up into smaller chunks. So, this utilized a blocking scheme (similar to cache optimization) to help reduce the message size while increasing the total number of messages being sent and received. For the plots below, we used a chunk size of 400 `float.64` values (specifically, related to the local portion of u being sent to neighboring processes, like during the parallel matrix-vector multiplication).

With the increased number of processors, more processes have to communicate with each other, and since the `MPI.Send()` and `MPI.Receive()` in `Python` are blocking commands, all the other processes have to wait for each other before proceeding. In addition, the case of 16 processes mark the start

of inter-node communication, leading to much longer communication and wait times. However, the 64 processes case with the greatly reduced time, from we can observe and deduce, could be explained by the reduced local portion needed to be communicated. So even if there is inter-node communication, the buffer limit on the `MPI.Send()` and `MPI.Receive()` might have pushed below the threshold and lead to increased performance.

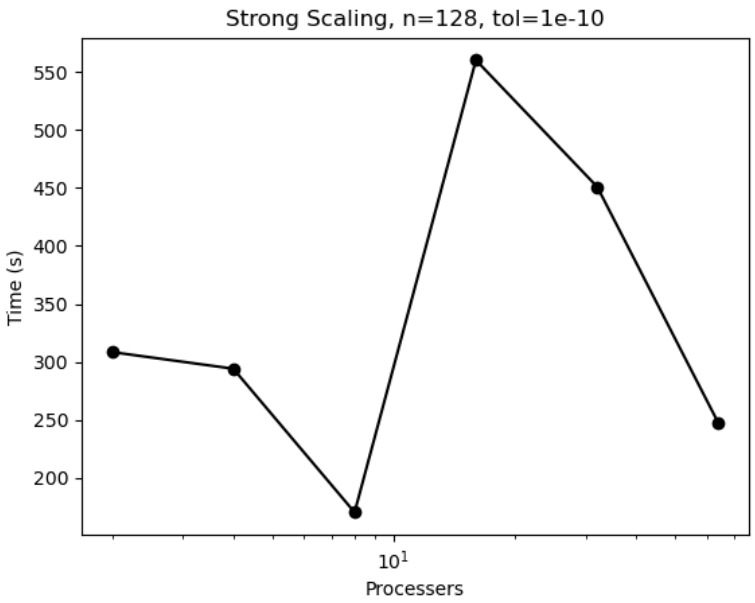


Figure 10: Strong Scaling, tol $= 1 * 10^{-10}$, 128 Grid Size

We observed a clearer example of strong scaling in the larger problem size as factors such as overhead were relatively less impactful in the figures produced. For this reason, the strong scaling on the larger problem size is a better example of strong scaling. With the smaller problem size, the overhead plays a more notable part in the overall run time.

4.2 Weak Scaling 10,000 Points:

We repeated our weak scaling experiment for a 100 by 100 grid size. The resulting plot is shown below. We observed the same general figure shape and trends as in the weak scaling for the smaller problem size. It is clear that there is a steeper slope for this weak scaling experiment compared to that of the smaller problem size. However, this speaks more to the problem size than the scaling. In general, the larger problem size shows better scaling as the overhead from the processors does not have as notable an impact on the results found.

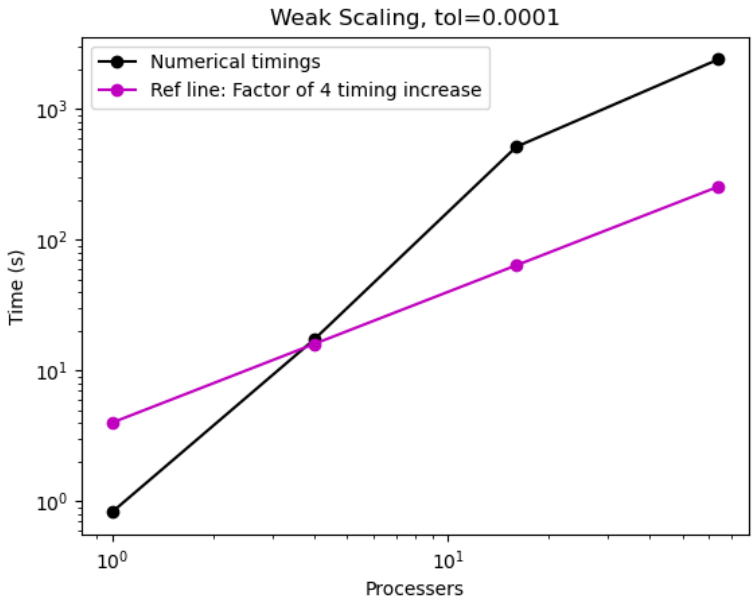


Figure 11: Weak Scaling on Large Problem Size