James Hahn
CS1645
High Performance Computing

Verification

Below are the results of the verification test with source term $S(x, y) = xe^y$ and boundary conditions $T(0, y) = 0$, $T(2, y) = 2e^y$, $T(x, 0) = x$, and $T(x, 1) = xe$. The exact solution is $T = xe^y$. The contour map (Figure 1) can be found below plotted on the domain $0 \leq x \leq 2$ and $0 \leq y \leq 1$ with one processor. Additionally, Figure 2 displays the contour map with four processors on a grid size of 400x200. A convergence rate of 10^{-12} was used on an iteration-by-iteration basis. All results in this report are from Jacobi algorithm requested in the project description.

Grid Size	**L-∞ Error**
5x5	1.0929×10^{-3}
11x11	1.8886×10^{-4}
21x21	4.8004×10^{-5}
41x41	1.2026×10^{-5}
51x51	7.6982×10^{-6}
101x101	1.9238×10^{-6}
201x201	4.7511×10^{-7}

The discretization is verified since the above tests were ran on successively finer grids. The algorithm was also tested on odd-dimension grids, such as 11x5, 21x11, and 41x21. The algorithm still worked and converged. It is important to note these lopsided dimensions will not necessarily work with more than 1 processor because the project specifications clarify each dimension is evenly divisible by the number of processors in that dimension. For example, if there are 3 dimensions in the y-dimension and 2 processors in the x-dimension, the grid dimensions must be divisible by 3 in the y-dimension and divisible by 2 in the x-dimension or the program will exit with an error.

Figure 1
Contour Map of verification test with 1 processor

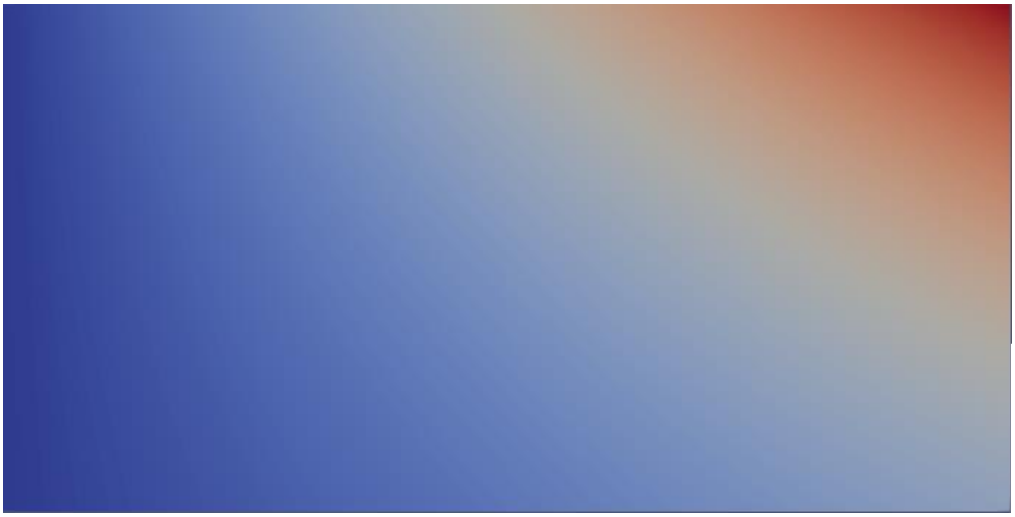
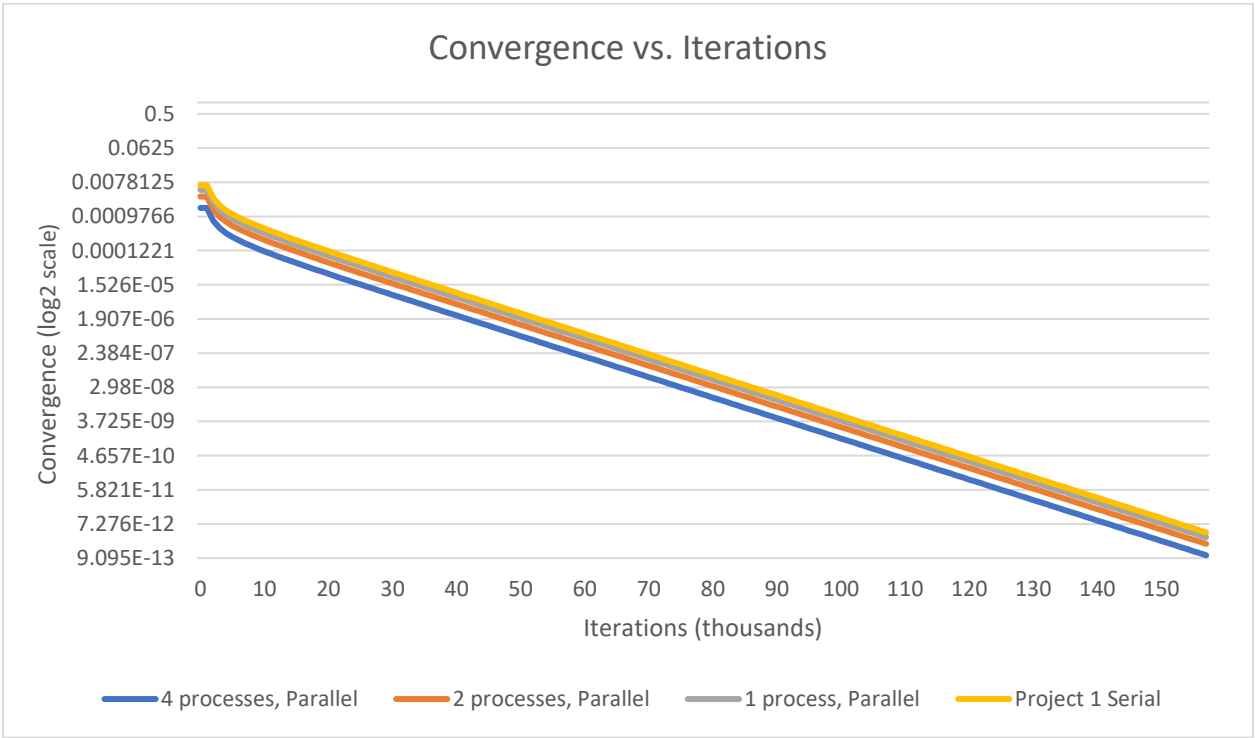


Figure 2
Contour Map of verification test with 4 processors

Convergence Analysis

The convergence of the algorithm was analyzed. On a grid size of 200x200, the simulation is ran in a serial manner, parallelized with one process, parallelized with two processes, and parallelized with four processes. A log scale was used for the convergence.



As one can see, the program's convergence does not depend on the number of processors used for the program. This is because only the Jacobi algorithm is implemented, so every iteration T_k depends solely on the previous iteration's T_{k-1} values. In contrast, using Gauss-Seidel or a hybrid Gauss-Seidel/Jacobi algorithm as mentioned in the project description, is hypothesized to have a positive effect on the convergence since Gauss-Seidel typically converges faster than

Jacobi, but implementation is a bit trickier and more memory is required by the algorithm to store both the previous and current iteration's values. Additionally, one will notice the convergence values were converted by a $\log_2$ scale, so the convergence does converge quicker at the beginning of the simulation and much slower toward the end, as indicated by the linear line discovered by the transformation. To further explain this phenomenon, doubling the number of iterations halves the speed of convergence, as well as halving the importance of convergence because the difference of convergence from 10 to 20 iterations and convergence values of 2.0×10^{-3} and 1.0×10^{-3} is much more significant than shifting from 50k to 100k iterations and convergence values of 2.0×10^{-9} and 1.0×10^{-9}.

Performance

As grid size increases, so did the time taken to solve Poisson's equation. The grid sizes were pushed to significant and experimentally viable sizes. Time was analyzed with the built-in MPI_Wtime() function purely around the algorithmic portion of the code to reduce noise from other functions, such as computing error at the end or setting initial border values.

When the simulation is ran with 2 processors, the partitioning scheme is 1 processor in the x-dimension and 2 processes in the y-dimension. When 4 processors are used, 2 processors are placed in each direction.

Grid Size	Time (seconds)	Processes
50x50	1.32436	1
50x50	0.734489	2
50x50	0.449535	4
100x100	21.3279	1
100x100	10.5581	2
100x100	5.6621	4
200x200	337.456	1
200x200	169.355	2
200x200	82.8084	4

Clearly, converting this problem into a multi-processing problem drastically reduces computational time. As the number of processes increases, the average runtime across the processes decreases. Even more precisely, when the number of processes doubles, the runtime almost halves. As such, the runtime decreases almost proportionally to the increase in processors. Generally, the time does not always

Effect of Partitioning Scheme

As a user can specify the number of processes in both the x and y directions, they can select many different combinations of partitions. In fact, the number of partitions they can choose is the number of unique pairs of factors of the number of processors. For example, if 12 processors are chosen, 12 has factor pairs of (1, 12), (2, 6), (3, 4), (4, 3), (6, 2), (12, 1) where (np_x, np_y) represents the combination of np_x processors in the x direction and np_y processors in the y direction. As such, with 12 processors, a user may partition their grid six different ways. To

analyze this, a grid size of 400x200 was chosen. Simulations were ran with 4 processes on three different schemes:

- Full block partitioning with $np_x > 1$ and $np_y > 1$
- Stripping in x-direction with $np_y = 1$
- Stripping in y-direction with $np_x = 1$

For full block partitioning, there were 2 processors in the x-direction and 2 processors in the y-direction. Below, the convergence and runtime are analyzed:

Grid Size	Partitioning Scheme	Time (seconds)	Iterations	L-∞ Error
400x200	Full block partitioning	277.314	246820	4.76256×10^{-7}
400x200	Stripping in y-direction	273.695	246820	4.76256×10^{-7}
400x200	Stripping in x-direction	269.195	246820	4.76256×10^{-7}

One can clearly see the L-∞ error stays consistent across any partitioning scheme. This makes sense; the values should converge to the ground-truth at the same rate. In addition, the number of iterations to achieve this convergence remains consistent across all schemes. The largest difference between the three partitioning schemes is the computation time.

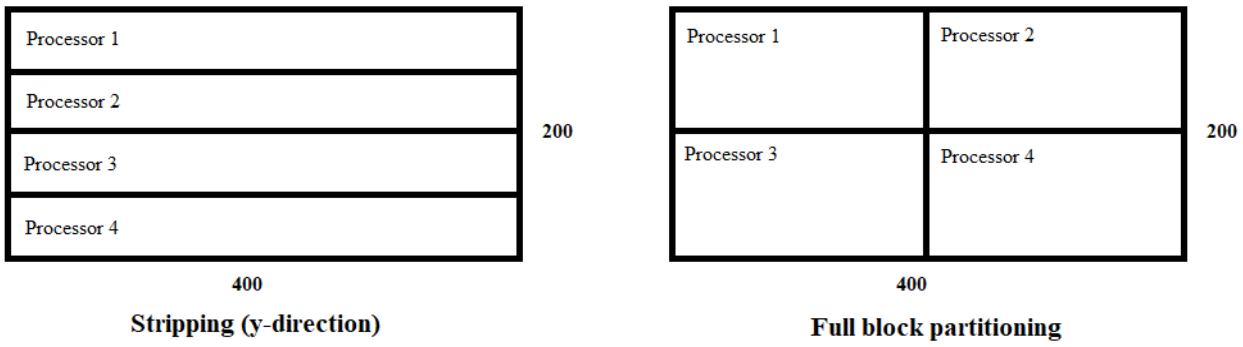


Figure 3 Demonstrating the difference between the stripping and full block partitioning schemes

First, it makes sense the stripping schemes take less computation time compared to full block partitioning. In full block partitioning, as seen in Figure 3, each processor must grab its border values from two processors (i.e. processor 1 grabs its right border values from processor 2 and bottom border values from processor 3). As such, for every iteration, the process requires eight send and receive requests across just four processors. Meanwhile, with stripping (in x-direction or y-direction), each iteration requires six send and receive requests across all four processors. This saves 25% of computation time spent on sending and receiving across every iteration. Thus, stripping will always produce lower computational times.

Secondly, stripping in the y-direction takes slightly less time than stripping in the x-direction due to the nature of this example. The previous paragraph mentioned having to send and receive a total of six requests across four processors. The reason stripping in the y-direction results in a

lower time is because every send and receive request only sends 200 values, compared to the 400 values required for the opposing scheme.

As such, results are consistent and make sense.

Algorithm Presentation

Clearly, the project handout contains a brief, general, vague description of the algorithm and Poisson equation. After further investigation, the Jacobi algorithm was implemented for this project to solve any solution given the source term, domain constraints, and boundary conditions.

As noted in the handout, the second derivative can be used to solve the true values of the solution. It turns out, the following equation, after solving for $T_{i, j}$, can be used to approximate the desired value:

$$(T_{i-1, j} - 2T_{i, j} + T_{i+1, j}) / (\Delta x)^2 + (T_{i, j-1} - 2T_{i, j} + T_{i, j+1}) / (\Delta y)^2 = S_{i, j}$$

Which ends up being:

$$[(\Delta y)^2 T_{i-1, j} + (\Delta y)^2 T_{i+1, j} + (\Delta x)^2 T_{i, j-1} + (\Delta x)^2 T_{i, j+1} - (\Delta y)^2 (\Delta x)^2 S_{i, j}] / (2(\Delta y)^2 + 2(\Delta x)^2) = T_{i, j}$$

So, this above equation was used to approximate the value of $T_{i, j}$ on each iteration, based on its neighbors and underlying source term, which was always provided for all coordinates.

An important note is this algorithm does not iterate over the boundaries because they are important boundary conditions to keep the algorithm stable. As such, $T_{i, j}$ is only a concern for all interior points of the grid.

One important challenge in the implementation of this algorithm, since it is utilizes parallel processors, is to ensure each processor maintains an MxN matrix containing only that processor's data and ensuring some processors iterate over all M values and others iterate over M-1 values. For example, in a four processor scheme, processor 0 possesses the overall top and left boundary values of the matrix. Thus, when updating values for the Jacobi algorithm, for loops must iterate from 1…M and 1…N, but processor 1, which possesses the top and right boundary values, iterates from 1…M and 0…N-1. Finally, processor 2, with the left and bottom boundary values, iterates from 0…M-1 and 1…N. Maintaining the integrity of these boundary conditions across any scheme and any number of processors is imperative or the algorithm is at risk of never converging of converging with poor values, thus producing high error.

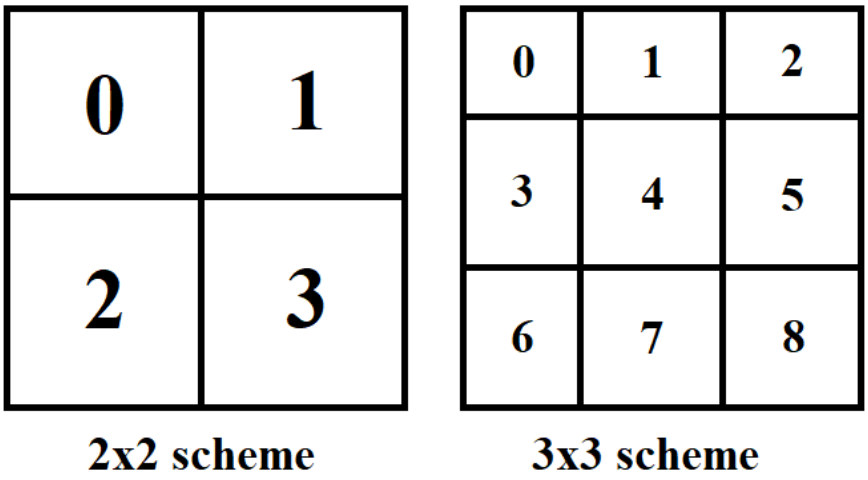


Figure 4
Contour Map of verification test with 1 processor

Additionally, another challenge is sending the correct boundary values from processor to processor. In a four processor 2x2 scheme, such as displayed in Figure 4, processor 0 only has two neighbors to send its right and bottom values to, processor 1 and 2. Meanwhile, it must obtain its right and bottom border values from processors 1 and 2. This becomes quite a bit more complicated in higher level schemes, such as on the right of Figure 4 with a 3x3 scheme. Processor 0, once again, has two neighbors, processors 1 and 3. However, processor 4 has four neighbors, processors 1, 3, 5, and 7. Ensuring awareness of *how many* neighbors, *which* neighbors, and *purpose* of each neighbor is valuable. Failure of maintaining this may result in failure to converge or high error.

In the end, to find the error between the 'solved' grid and the 'true' grid, given in the verification test, we calculated the L-∞ norm (maximum difference between 'solved' value and 'true' value in the grid).

Satisfactory results were generated, as the grid both converged and produced useful contour maps related to the underlying solution values.