HW #4
Math/CS 471, Fall 2021

Michael Sands
John Tran

November 6, 2021

Contents

1 Task 1:

1.1 Testing Quadratic Convergence for Serial Code

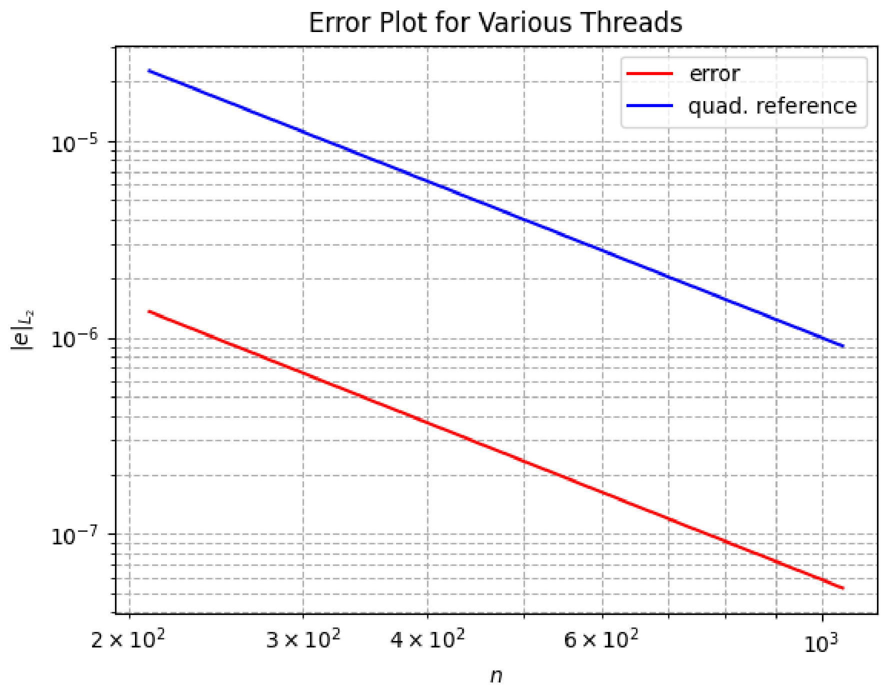


Figure 1: Error Plot: 1 Thread

1.2 Printing Separate Cases From The `A` Matrix

Below contain the three matrix rows for `Option 3` with a grid size of `6` $\times$ `6`.

NOTE: We can use the second-order finite difference stencil shown below to parse through the matrix rows and find the corresponding row for each of the three cases: interior row, corner row, and wall row. In addition, the point location (in terms of the $x-$ and $y-$indices) will be given for each case to better illustrate which point we are referring to. In terms of indexing, we will be starting at 0 – so, for a grid size of `6`, we index 0 to 5 for each dimension.

$$\frac{1}{h^2}\begin{bmatrix}0 & 1 & 0\\ 1 & -4 & 1\\ 0 & 1 & 0\end{bmatrix}$$

For our grid size of `6`, we can fill the rest of the row in the 2D stencil above with zeros until all `6` elements are filled up. However, the way we expand the stencil with zeros are different for each case. A further explanation will be given for each case below.

1.2.1 Case: Interior row

[0. 0. 25. 0. 0. 0. 0. 25. -100. 25. 0. 0. 0. 0. 25. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

Point indices (location): (1, 2)

This row of the `A` matrix contains ...25 ...25 -100 25 ...25 .... Because all 5 of these values are present, the point above, below, to the right, to the left, and the point of interest are all accounted for in the finite difference approximation.

For this case, we can use the entire stencil for three rows in the immediate vicinity of the point (filling the rest in the rows with zeros as mentioned above). Since we've chosen a point in the second row (index 1), we can use the stencil for rows 0 to 2 where the center value of -4 is multiplied by a factor of 25 (by the `poisson.py` script) and aligned with the third column (index 2). So, if we print out the filled out matrix for this particular point:

$$\begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 25 & 0 & 0 & 0\\ 0 & 25 & -100 & 25 & 0 & 0\\ 0 & 0 & 25 & 0 & 0 & 0\end{bmatrix}$$

Essentially, we can get the matrix row given above by flattening out this matrix, starting with row 0 (the last row in this case since the matrix is shown like a grid on the xy-plane) and sequentially adding each subsequent row (going above) after that (i.e., row 1 next above row 0, then row 2 above row 1, and so on).

1.2.2 Case: Corner row

[-100. 25. 0. 0. 0. 0. 25. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

Point indices (location): (0, 0)

This row of the `A` matrix contains -100 25 . . . 25 . . . . Because two 25 values representing two adjacent points in the matrix are not present, we know that this is a corner point in the matrix (i.e., two values not on directly opposite sides of the central -100 value). We also expect that the bottom row of the stencil matrix refers to the bottom left-hand corner of the matrix, further confirming that this is indeed a corner point.

For this case, two of the neighboring points in the stencil (i.e., values with 25) are not used since the point of interest occupies a corner of the filled out matrix (shown below). As opposed to before, only two rows of the stencil are used within the boundary conditions – the row with the -4, and the row above it. The specific rows of the stencil used depend on the coordinates of the specific corner point. So, the matrix for this particular point is:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 25 & 0 & 0 & 0 & 0 & 0 \\ -100 & 25 & 0 & 0 & 0 & 0 \end{bmatrix}$$

We can again get the matrix row given above by flattening out this matrix, starting with row 0 (the last row in this case since the matrix is shown like a grid on the xy-plane) and sequentially adding each subsequent row (going above) after that (i.e., row 1 next above row 0, then row 2 above row 1, and so on).

1.2.3 Case: Wall row

```
[ 25. -100. 25. 0. 0. 0. 0. 25. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Point indices (location): (0, 1)

This row of the `A` matrix contains 25 -100 25 . . . 25 . . . . Since one of the 25 values is not present, we can conclude this is a wall point in the matrix. To say this in a different way, each 25 in the matrix refers to a point either above, below, to the right, or to the left of the point of interest. So, because one of the 25 values is not present, one of the points directly above, below, to the right, or to the left of the point of interest is not present, this means the point of interest must be a wall point and not a corner point.

Since we've chosen point on the South Wall to the right of the lower left corner point – first row (index 0) and second column (index 1) – the 25 value referring to the point that would be below the point of interest is zero. So, if we print out the filled out matrix for this particular point:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 25 & 0 & 0 & 0 & 0 \\ 25 & -100 & 25 & 0 & 0 & 0 \end{bmatrix}$$

As before, we can get the matrix row given above by flattening out this matrix, starting with row 0 (the last row in this case since the matrix is shown like a grid on the xy-plane) and sequentially adding each subsequent row (going above) after that (i.e., row 1 next above row 0, then row 2 above row 1, and so on).

2 Task 2:

2.1 Error Plots: Serial vs. Parallel

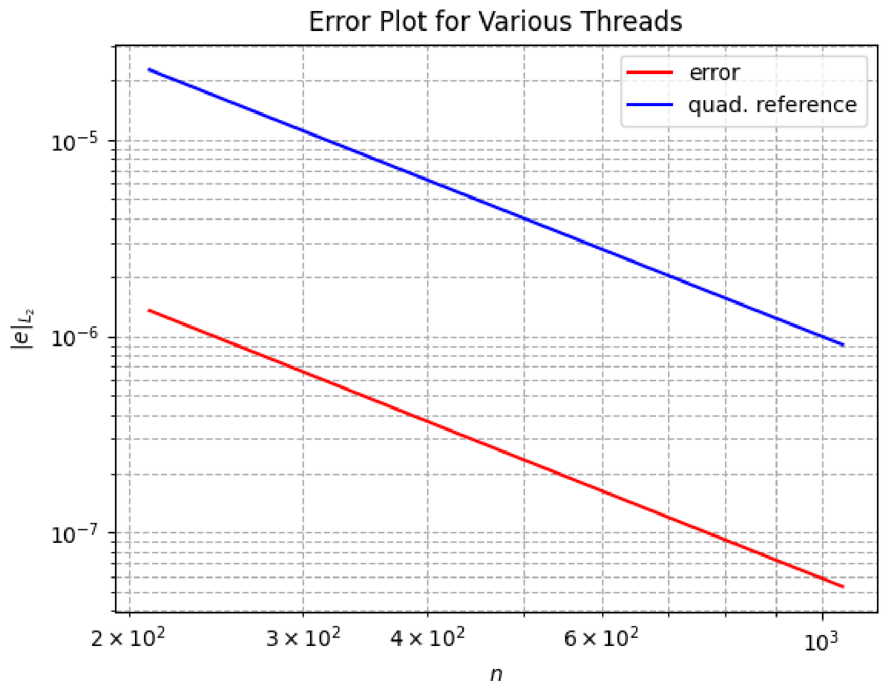


Figure 2: Error Plot: 2 Threads

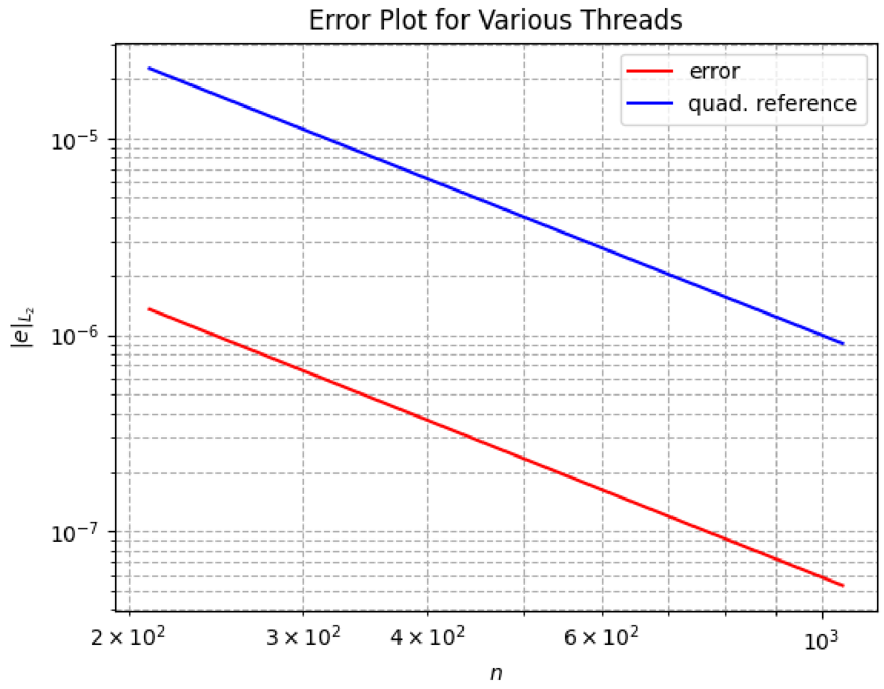


Figure 3: Error Plot: 3 Threads

In Figures 1, 2, and 3, we can see that the number of threads does not affect the rate of convergence for the 2D finite differencing approximation using 1D partitioning. In all the thread cases, the error was found to be quadratic. This should make sense because we are using the same finite differencing scheme for all the thread cases – splitting up the work to multiple threads does not effect the order of the finite differencing approximation.

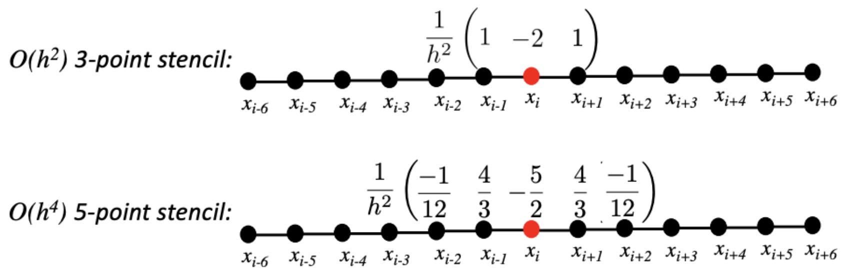
2.2 Timings for 1D Partitioning

	Values of n				
Thread Count	210	420	630	840	1050
1	0.00876427	0.032269	0.08369422	0.14947295	0.19182301
2	0.00398922	0.01915383	0.05043793	0.08485603	0.14407825
3	0.00508285	0.01389909	0.03066707	0.07702518	0.09173799

Table 1: Timing data for `Option 2`

Table 1 displays the computation time for each problem size (value n) compared to the number of threads used. We observed that for increasing number of threads, the overall time decreased. The only deviation in this is seen in $n = 210$. If we compare a thread count of 2 vs. 3, we see that the time cost for $n = 210$ was higher for a thread count of 3, even though a larger number of threads was used. This can be understood by Amdahl's law which describes that the time needed to complete a computation is relative to the number of threads used and the proportion of the code that is parallelizable. In addition, this illustrates an example of overhead when trying to parallelize the serial algorithm because the time required to create and destroy threads increases the time of computation (not present in the serial algorithm). This effect becomes more noticeable for smaller problem sizes.

If we used the fourth order finite difference stencil discussed in class, the halo region for each thread would be twice the size. To clarify, the number of relevant rows and columns on each side of the point of interest would double (i.e., from extending by 1 for our second order finite difference stencil to 2 for the fourth order finite difference stencil), but the actual region for each thread would depend on the grid size, the number of threads used, and the location of the point of interest. The weights are different between the second and fourth order stencils as seen in Figure 4. The increase in halo region makes the 4th order stencil more accurate overall. If we look closely at the weights of the 4th order stencil, we can see that the contribution from further points diminishes with distance from the point of interest as their weights in the stencil decrease.



$O(h^2)$ 3-point stencil:
$\frac{1}{h^2}\begin{pmatrix}1 & -2 & 1\end{pmatrix}$
$O(h^4)$ 5-point stencil:
$\frac{1}{h^2}\begin{pmatrix}\frac{-1}{12} & \frac{4}{3} & -\frac{5}{2} & \frac{4}{3} & \frac{-1}{12}\end{pmatrix}$
x_{i-6} x_{i-5} x_{i-4} x_{i-3} x_{i-2} x_{i-1} x_i x_{i+1} x_{i+2} x_{i+3} x_{i+4} x_{i+5} x_{i+6}

Figure 4: 4th Order Finite Differencing Stencil from Lab 14 pdf

2.3 CARC Strong Scaling Plots

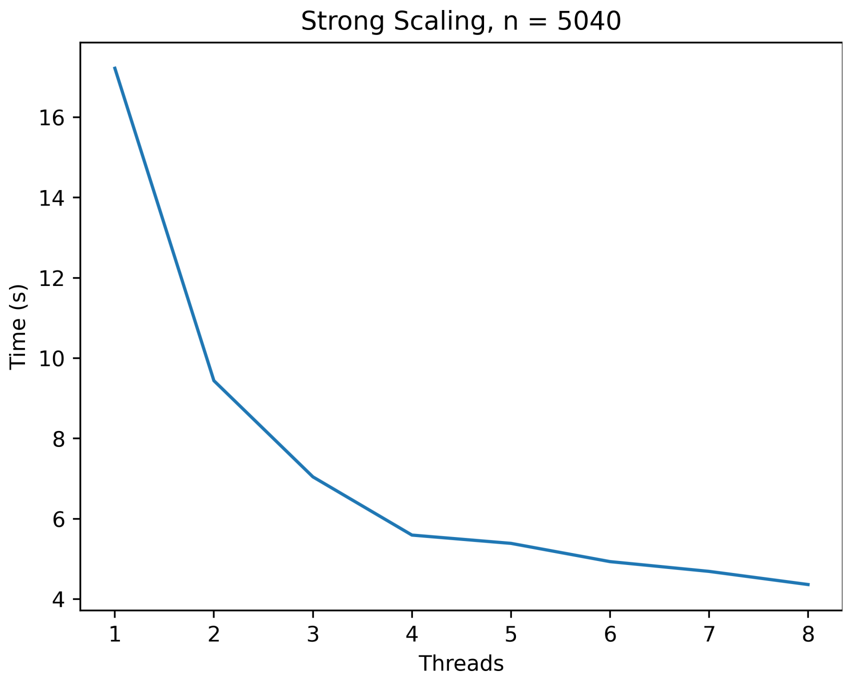


Figure 5: Strong Scaling

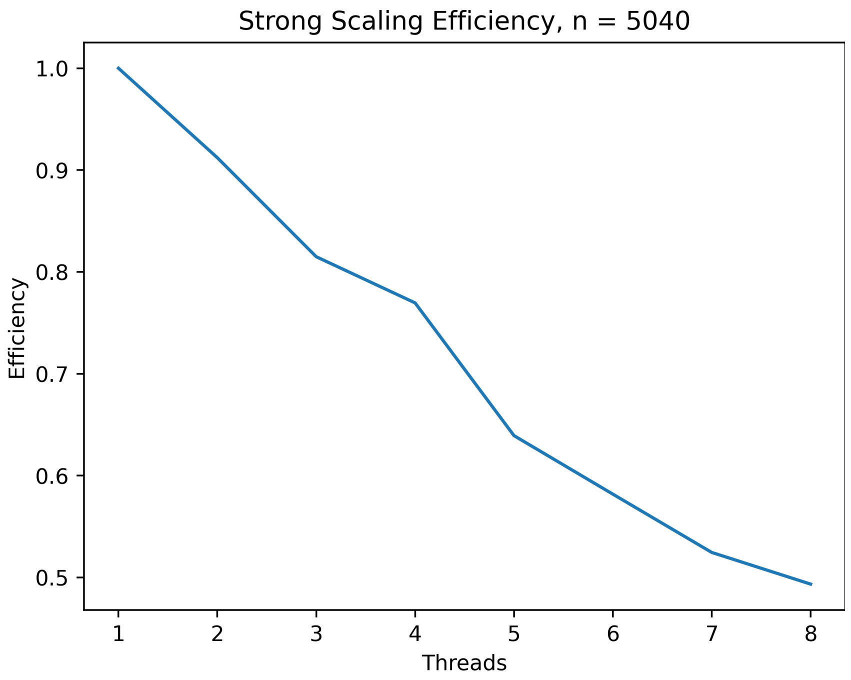


Figure 6: Strong Scaling Efficiency

From our strong scaling plot, as shown in Figure 5, the best number of threads for overall runtime was 8 because it resulted in the shortest runtime. Looking at the strong scaling efficiency, as shown in Figure 6, as the number of threads increases, the efficiency decreases. That said, for thread numbers 1-4, there is a significant increase in performance (i.e., significantly less overall runtime) in the strong scaling plot. The addition of two threads to the serial code reduced the runtime by about a factor of about two. After about 5 threads, we start to see the ”roll over” in the asymptote forming at about $y = 4$ seconds in Figure 5.

We noticed that the efficiency of strong scaling decreases more or less linearly with the number of threads, as shown in Figure 6. We did not observe any increase in runtime from increasing thread counts in Figure 5 as we might eventually expect as the overhead increases.

3 Task 3: Extra Credit

3.1 Serial Test for 2D Partitioning

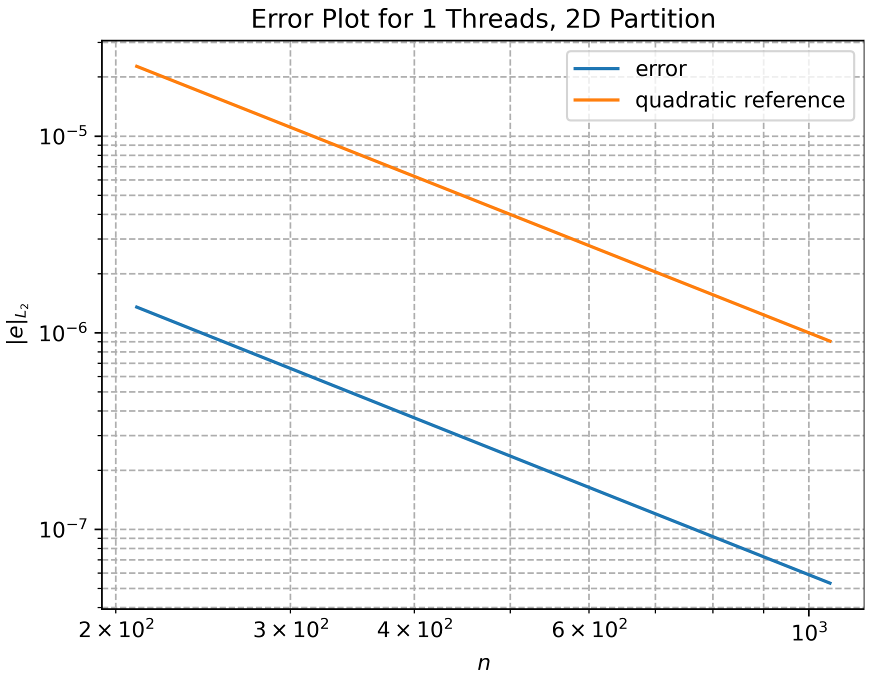


Figure 7: Error Plot: 1 Thread, 2D Partition

Figure 7 is identical to Figures 1, 2, and 3 in 1D partitioning. From this, we can say that the overall order of the finite diferencing algorithm is independent of the method of thread partitioning.

3.2 Printing Separate Cases from the `A` Matrix

For the examples provided below, $n = 6$ and $nt = 9$. We split the submatrices into sizes of 2×2 since $\sqrt{9} = 3$ and each dimension is split into three subsections, for a total of 9 subsections.The dimension of the rows below depend on the halo region for the specific case.

In addition, to make the cases simple, we opted to choose the same case within each submatrix (e.g., we chose the middle point from the middle thread). The example rows chosen were also relative to the halo region of the respective case – since the submatrix has a size of 2×2, there would be no middle or wall points in any of the unextended regions. So, the middle point for the middle thread would use the corresponding row from the `A` matrix, created from the halo intervals of each dimension.

3.2.1 Case: Interior Row

[0. 0. 0. 0. 0. 25. 0. 0. 25. -100. 25. 0. 0. 25. 0. 0.]

Middle submatrix
Submatrix indices: (1, 1)
Point Indices (within submatrix): (2, 1)
Point Indices (within original matrix): $(1 * 2 + 2 = 4,\ 1 * 2 + 1 = 3)$

As we noted for the 1D partitioned case, in the interior row, all values form the stencil (. . . 25 . . . 25 -100 25 . . . 25 . . .) are present. The reasoning behind this is the same as for the 1D partitioned case explained above in Section 1.2.1.

3.2.2 Case: Corner Row

[-100. 25. 0. 0. 25. 0. 0. 0. 0. 0. 0. 0.]

Bottom Left Corner Submatrix
Submatrix indices: (0, 0)
Point Indices (within submatrix): (0, 0)
Point Indices (within original matrix): $(0 * 0 + 0 = 0,\ 0 * 2 + 0 = 0)$

As we noted for the 1D partitioned case, in the corner row, two neighboring values from the stencil are not present. The reasoning behind this is the same as for the 1D partitioned case explained above in Section 1.2.2.

3.2.3 Case: Wall Row

[25. -100. 25. 0. 25. 0. 0. 0. 0.]

South Wall Submatrix
Submatrix indices: (0, 1)
Point Indices (within submatrix): (0, 1)

Point Indices (within original matrix): $(0 * 0 + 0 = 0,\ 1 * 2 + 1 = 3)$

As we noted for the 1D partitioned case, in the wall row, one neighboring value from the stencil is not present. The reasoning behind this is the same as for the 1D partitioned case explained above in Section 1.2.3.

3.3 Parallel Tests for 2D Partitioning

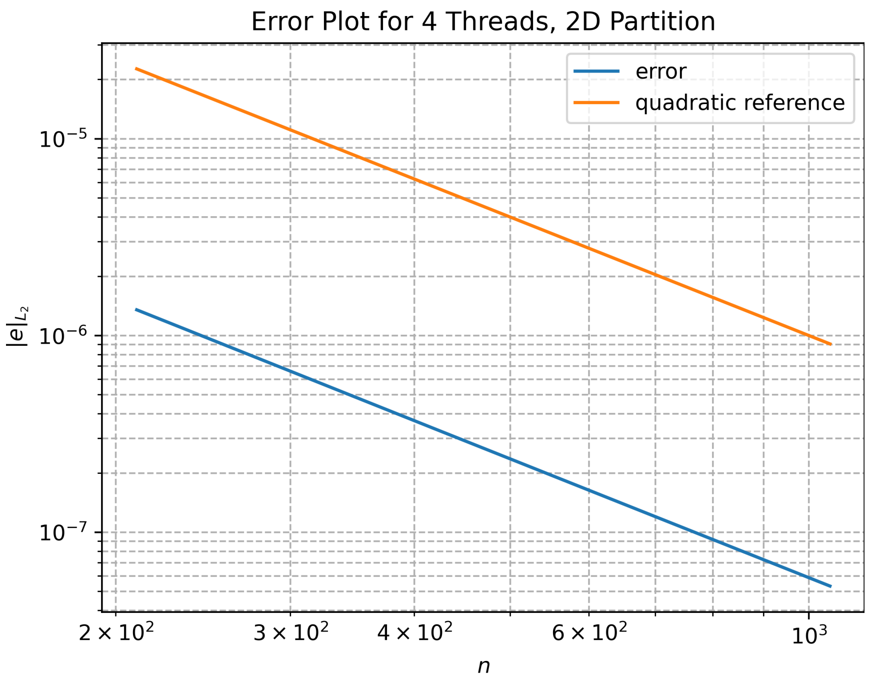


Figure 8: Error Plot: 4 Threads, 2D Partition

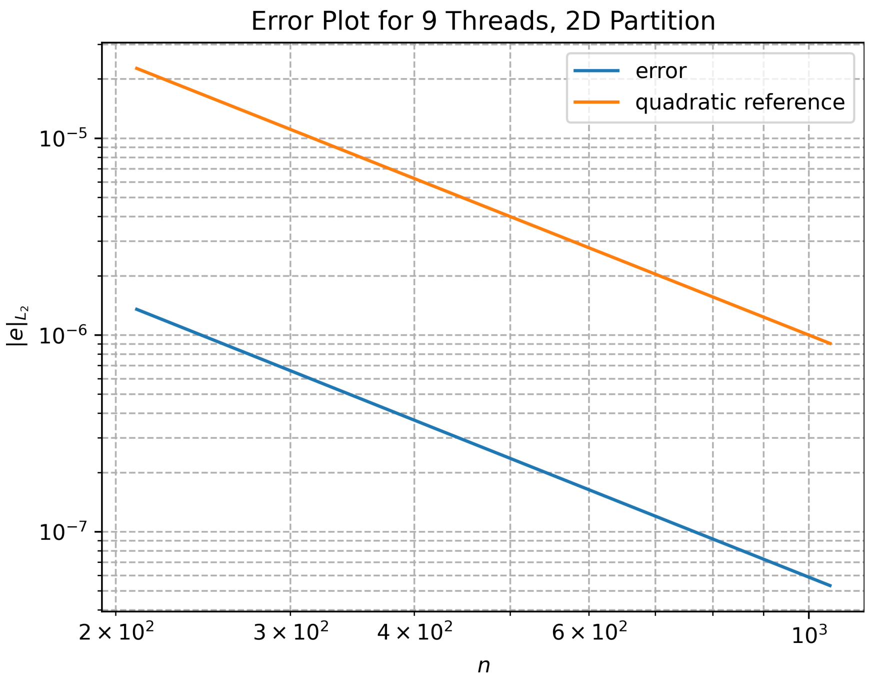


Figure 9: Error Plot: 9 Threads, 2D Partition

In order to hold the 2D partitioning code to the same standards as the 1D partitioning code, we produced Figures 8 and 9 with 4 and 9 threads respectively. These figures are identical to Figure 7 as well as the error plots for 1D partitioning. The fact that the error plots for the 1D and 2D partitioning are identical for various thread counts shows the ability of the 2D partitioning code to produce the same results as the 1D partitioning code.

3.3.1 Timings for 2D Partitioning

	Values of n				
Thread Count	210	420	630	840	1050
1	0.00655413	0.02658796	0.06513715	0.15349674	0.20550489
4	0.01336503	0.0339191	0.08469987	0.087255	0.14559984
9	0.0149889	0.02699089	0.0462029	0.09262633	0.12669301

Table 2: Timing data for `Option 2`, 2D partitioning

For the case of 2D partitioning, the time involved in creating and destroying threads is more noticeable at lower n values. At $n = 210$ we observe that the time involved in computation increases as the number of threads increases. This is largely due to the smaller problem size. We observed a similar effect in Section 2.2 where the time involved with $n = 210$ is greater for 3 threads than it was for 2 threads. Further, Wheeler only has 8 cores per compute node, meaning that the time involved in the 9th thread will be off compared to the first 8. However, as the problem size increases, the time benefits of 2D partitioning become more apparent. At $n = 1050$, as the number of threads increases, the time decreases.

If we implemented the fourth order finite difference stencil, the halo regions in all directions would be twice as long. This is because for each point in the sub-matrix being analyzed, two additional points in each coordinate direction are needed to satisfy the stencil. In our code, we take a rectangular section of the overall matrix that includes the halo region. The halo region and corners of the rectangular section are truncated shortly after they are defined, so there should not be a notable impact on memory in the code. There would also need to be an additional section of code to handle the boundary conditions two rows out from the walls. The overall effects of the fourth order finite difference stencil on the halo region would be more or less the same as discussed for the 1D partitioned case.

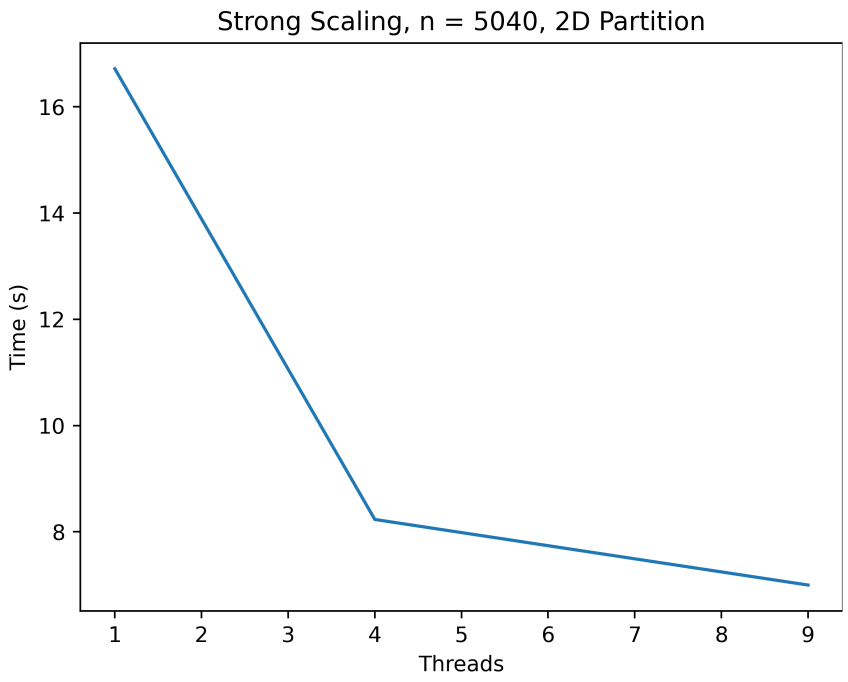


Figure 10: Strong Scaling, 2D Partition

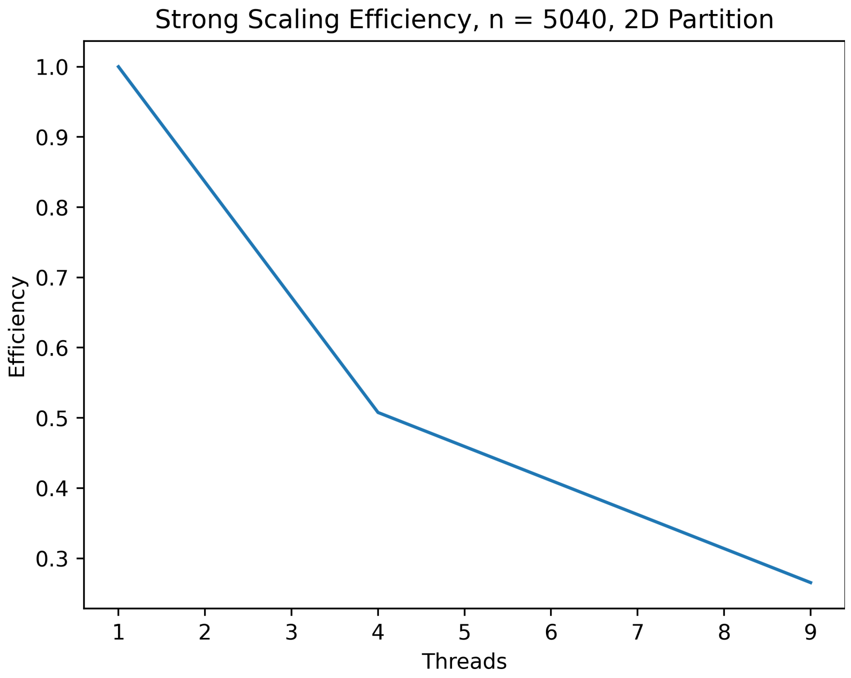


Figure 11: Strong Scaling Efficiency, 2D Partition

For larger problem sizes, 9 threads has the lowest overall run time as seen in Figure 10. However in Figure 11 we can see that the strong scaling efficiency decreases as the number of threads increases. For smaller problem sizes, if the time that it takes to create and destroy threads is comparable to the time required to perform the computation, the lowest run time may appear at lower thread numbers.

We noticed a large performance gain (as seen through the time) between threads 1 and 4. After thread 4, the rate of time decrease as the number of threads increases appears to decrease. This can be thought of as roll over around $Time = 6s$ in Figure 10.