Homework 3 MATH/CS 471, Fall 2023

Owen Pannucci[1] and Liam Pohlmann[2]

[1]University of New Mexico, Department of Arts and Sciences, Applied Mathematics
[2]University of New Mexico, Department of Nuclear Engineering, Nuclear Engineering and Mathematics of Computation

November 6, 2023

Part I

Problem

The following equation is given as the continuous function value:

$$u(x,y) = \cos\left((x+1)^{1/3} + (y+1)^{1/3}\right) + \sin\left((x+1)^{1/3} + (y+1)^{1/3}\right) \tag{1}$$

To compare our numerical finite difference analysis (FDA) values, we compute the analytic second derivatives as:

$$\frac{\partial^2 u}{\partial x^2} = -\frac{2}{9}(x+1)^{-\frac{5}{3}} \sin\left((x+1)^{\frac{1}{3}} + (y+1)^{\frac{1}{3}}\right) - \frac{1}{9}(x+1)^{-\frac{4}{3}} \cos\left((x+1)^{\frac{1}{3}} + (y+1)^{\frac{1}{3}}\right) \tag{2}$$

and:

$$\frac{\partial^2 u}{\partial y^2} = -\frac{2}{9}(y+1)^{-\frac{5}{3}} \sin\left((x+1)^{\frac{1}{3}} + (y+1)^{\frac{1}{3}}\right) - \frac{1}{9}(y+1)^{-\frac{4}{3}} \cos\left((x+1)^{\frac{1}{3}} + (y+1)^{\frac{1}{3}}\right) \tag{3}$$

By specifying a grid size (see next section), we generate a linear system of the form:

$$A\mathbf{u} + \mathbf{f} = D\mathbf{u} \tag{4}$$

where D is the second derivative operator on $\mathbf{u}$.

Method

To begin, a desired grid-size of (n x n) for the problem and thread count for parallelization are initialized. The problem is then split up based on threading. Each thread is responsible for the computation of 1/k of the matrix given k threads. To compute the k$^{\text{th}}$ thread, the thread will be passed into a function, `compute_fd`, which takes that thread number (k), the total number of threads (nt), the number of grid points in the x and y directions (n), the function defined in (1), and a vector to write into, `fpp_num`.

Inside this function, the bounds of the problem for which the k$^{\text{th}}$ thread is responsible for are defined as $k * (n/nt)$ for the start and $(k+1) * (n/nt)$ for the end. Each thread computes the second derivatives at every point between its start and end. We must also define a halo[1] region as one row above and below the end and start respectively in order to grab information

[1]We define this as the region extending one data point outside the domain of interest. The purpose of this region is to provide the necessary missing data for the computation of the thread.

surrounding the start and end bounds that are required for their computation.

The matrix A, which is generated in CSR format, is made using the provided `poisson.py` function. This function created a matrix of size (n^2 x n^2) where the i$^{\text{th}}$ row of this matrix is multiplied by the column vector of solutions to (1) to obtain the second derivative at the ith point. The column vector of solutions to (1) are calculated only for the required points inside the halo region bounds. Matrix A is sparse matrix with -4 along the diagonal and 1 to the direct left and right of the diagonal. There is also a 1 at the n$^{\text{th}}$ position to the left and right of the diagonal. We can only guarantee that the -4 along the diagonal exists as some of the 1's will not depending on the position of the point to be calculated. For example, a point in the bottom left corner of the domain will not have a 1 anywhere to the left of the diagonal in that row because that would correspond to information outside the domain. We therefore must look to the boundary conditions to complete the calculation. Matrix A is scaled by $\frac{1}{h^2}$ where $h = \frac{1}{n-1}$. The output of this function is truncated so only points within the `start` and `end` bounds are returned.

As mentioned before, points along each wall of the grid are calculated differently because there is no halo region that we can use to grab information needed in the computation. We can observe this by printing the matrix rows of A corresponding to a corner, a wall, and an interior point:

A[0] $\rightarrow$ (X=0,Y=0) bottom left corner
(0, 0) -99.99999999999999
(0, 1) 24.999999999999996
(0, 6) 24.999999999999996
A[3] $\rightarrow$ (X=3, Y=0) middle of south wall (0, 2) 24.999999999999996
(0, 3) -99.99999999999999
(0, 4) 24.999999999999996
(0, 9) 24.999999999999996
A[14] $\rightarrow$ (X=2, Y=2) interior point
(0, 8) 24.999999999999996
(0, 13) 24.999999999999996
(0, 14) -99.99999999999999
(0, 15) 24.999999999999996
(0, 20) 24.999999999999996

First, notice all values of a are scaled by $\frac{1}{h^2}$, where h = 0.2. For the $A[0]$ print, only the three values are available because this point is located in the corner, and the information of the points directly to the left (0, -1) and bellow (0, -6) do not exist. Similarly, for the $A[3]$ print, the information for the point below (0, -3) does not exist. Finally, for the interior point, we have all the information of the surrounding points, so all 5 points are used in the A matrix multiply. To solve the problem for walls and corners, we must use the fact that this problem has Dirichlet boundaries $g(x, y) = u(x, y)$ for $x, y \in \delta\Omega$. We get $A \times u + \frac{1}{h^2}g$ where $g = 0$ for the interior points.

After obtaining the second derivative of the function numerically, we look to analyze the error using a scalable method. In this case, we look to the *L2-norm*, which is defined as:

$$\|u\|_{L_2} = \left(\int_{[0,1]\times[1,0]} u^2\,dx\,dy\right)^{\frac{1}{2}} \approx h\left(\sum_{i,j} u_{i,j}^2\right)^{\frac{1}{2}}, \tag{5}$$

where u is the difference between the numerical solution and analytical solution. The analytic solution is found by taking the second derivative by hand and plugging in all (X,Y) points generated by Python.

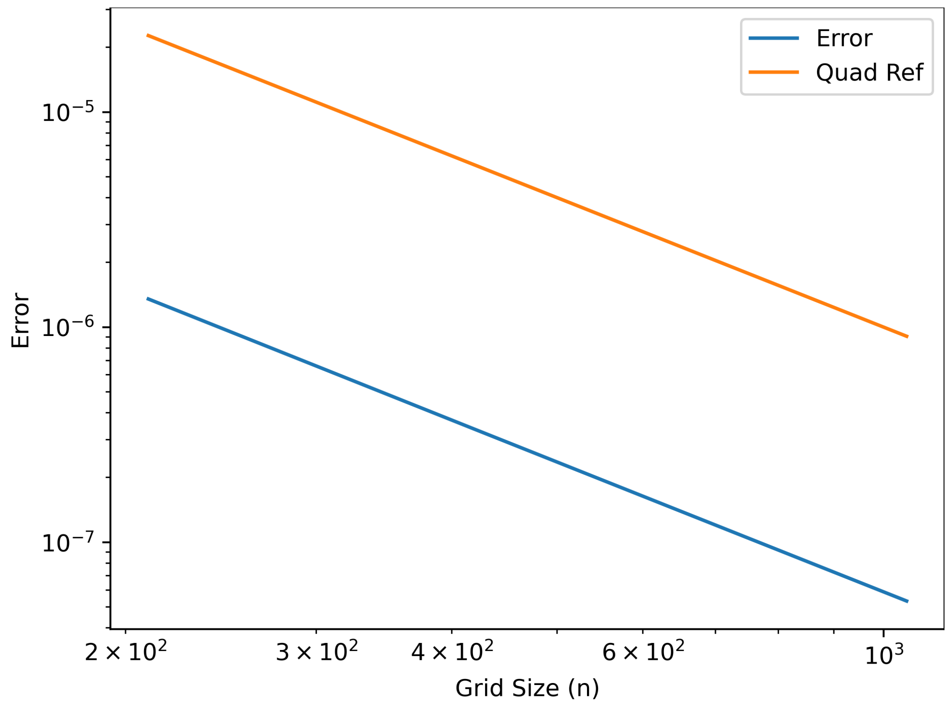
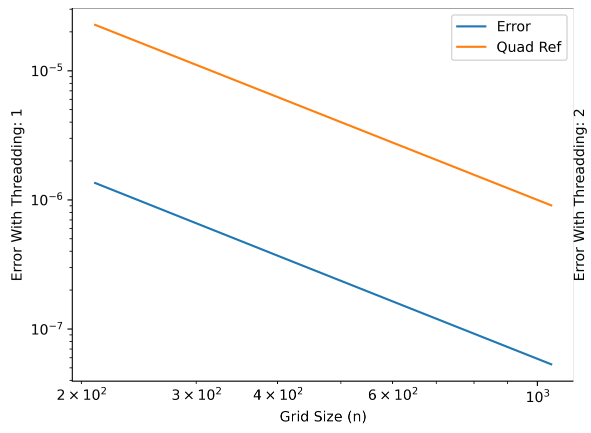


Figure 1: Error convergence plot of n vs $|e|_{L_2}$

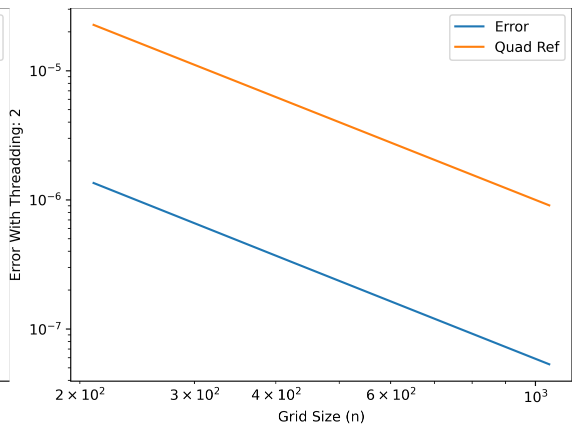
In the Figure (1) log-log plot, we can observe that the error converges quadratically as the grid-size (n) increases.

Part II

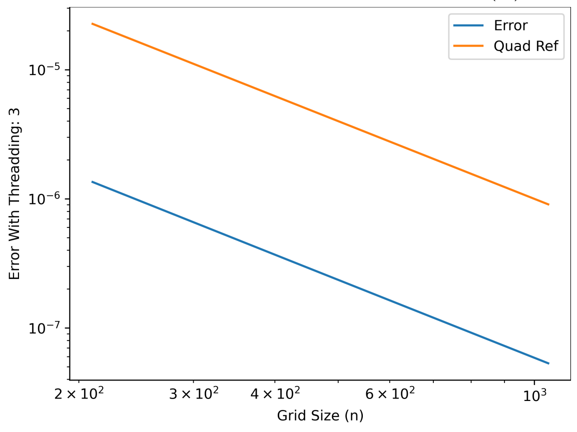
To be certain the parallelization of the code works, Part I was run for 1, 2, and 3 threading options. We can verify by confirming the error plots between the different threading options are the same in Figure (2).



(a) 1 thread



(b) 2 threads



(c) 3 threads

Figure 2: Error convergence plots of n vs $|e|_{L_2}$

Each (grid-sized, thread count) is run 5 times, and the minimum timing is recorded as the timing for that set.[2]

Observe the timings obtained on a local machine in Table (1). In each column, the time for the problem to run decreases as the threading count increases. Now, we carry out a strong scaling study by running more threads and a larger problem size on one node of Hopper at

[2]We assume the minimum value is 'good enough' because background processes on local machines will inevitably vary the timings.

	n=210	**n=420**	**n=630**	**n=840**	**n=1050**
Threads: 1	0.02213584	0.05143981	0.13824956	0.24345801	0.37211507
Threads: 2	0.01548173	0.03584316	0.08650753	0.15513095	0.25120930
Threads: 3	0.01253371	0.03267983	0.06768428	0.12080858	0.19737838

Table 1: Timings (s) for different problem sizes and threading

CARC. First, we want to observe the speedup of the program for an increasing number of threads. Figure (3) was made using grid size n = 2520. As we can see for each additional

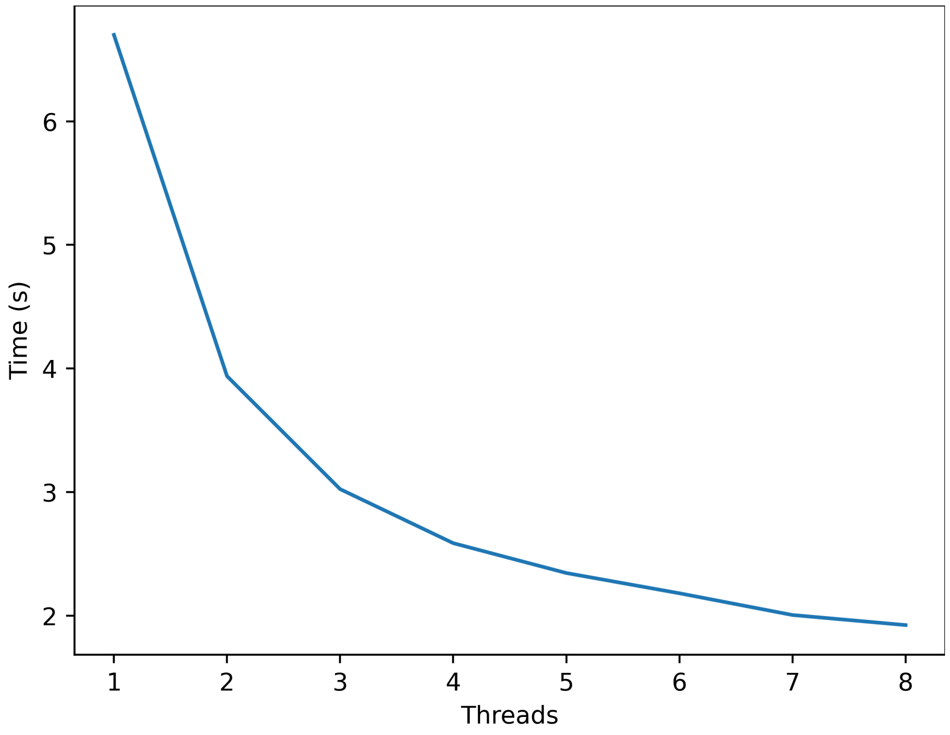


Figure 3: Strong scaling: run time vs number of threads

thread, the run time decreases. We can see that threading is very useful in this problem because we are able to reduce run time from 6.70 seconds (1 thread) to only 1.92 seconds (8 threads). Also note that the parallelization of this problem is not exactly "embarrassingly parallel" because there are some computations that are not able to be sped up by increasing threading count. We can observe this by plotting efficiency in Figure (4) defined as:

$$E_p = \frac{T_1}{T_p p} \tag{6}$$

Let E_p be the efficiency using p threads, and T_p be the execution time taken using p threads.

Each additional thread does decrease the run time, however, the efficiency of the threading

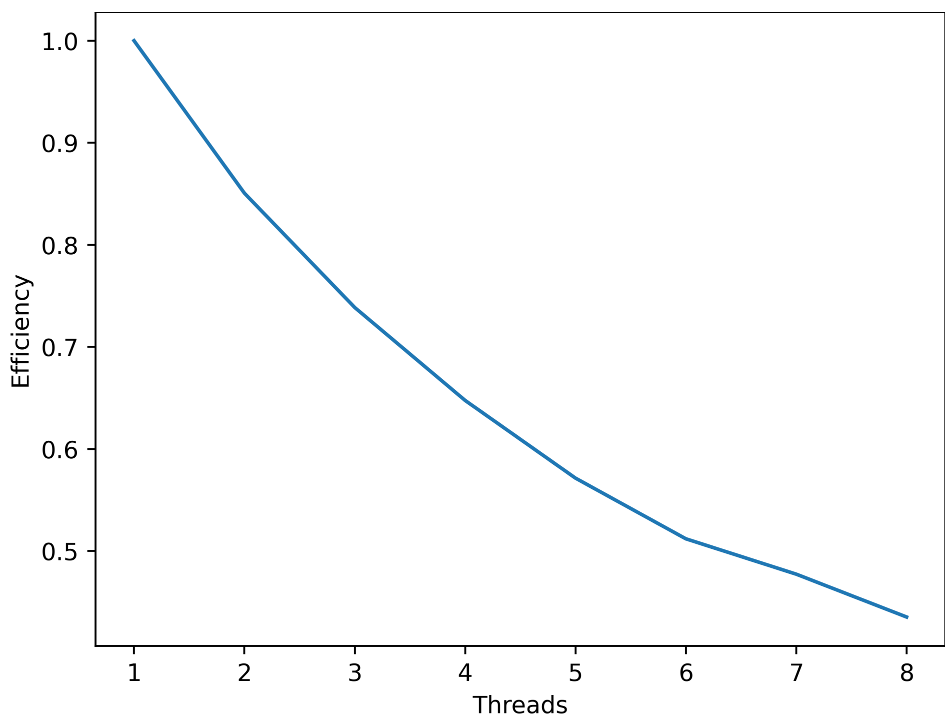


Figure 4: Strong scaling efficiency of threading

decreases as well. This means that while the best number of threads for overall run time is 8 (the max), there is a loss in performance when increasing the thread count. We can see this because the difference in time for 7 to 8 threads is only .08 seconds. The strong scaling in our plots does not roll over (increase for larger threads), however it is possible as the timings for 6–8 threads are very close and the relative influence of background processes is more significant. On any random trial, the background processes could make the timing longer for 8 threads than 6 or 7 threads.