HW# 4

Aaron Segura
Frederick Lee

October 29, 2018

1 Task 1: Compute $e(h)$ Using Skeleton Code and Second-Order Finite Difference Stencil

1.1 What

The overall goal of this assignment is to implement and experiment with threaded programming or shared memory in Python. In the following tasks, we will conduct a strong scaling study for computing numerical derivatives and integrals. To do this, we will first develop code in Python on our local machines and then carry out the strong scaling study in UNM's CARC (Center for Advanced Research Computing).

In the following tasks, we will consider the following function:

$$u(x,y) = \cos\left(\sqrt[3]{x+1} + \sqrt[3]{y+1}\right) + \sin\left(\sqrt[3]{x+1} + \sqrt[3]{y+1}\right) \quad (1)$$

Letting the domain be the unit square $\Omega := (x,y) \in [0,1] \times [0,1]$ and the number of grid points in each dimension be the same in both dimensions x and y. We will denote the number of grid points in a dimension as n, such that $h = \frac{1}{n-1}$ giving us a total of n^2 grid points. In this homework we will measure the error e as a function of h such that

$$e(h) = \left(\int_0^1 \left(\frac{\partial^2 u_{approx}}{\partial x^2} - \frac{\partial^2 u_{exact}}{\partial x^2} + \frac{\partial^2 u_{approx}}{\partial y^2} - \frac{\partial^2 u_{exact}}{\partial y^2} \right) dx \right) \quad (2)$$

In other words, we will be measuring the difference between the exact derivative and the approximate numerical derivative.

To accomplish this, we will be using the course provided **poisson()** function to generate a Compressed Sparse Row (CSR) matrix while scaling this matrix by h^2. As a side note, the Poisson equation is an elliptic type partial differential equation that takes the form of $\Delta u = f$. This formula has broad applications in mechanical engineering and theoretical physics such as describing the potential field created by a charge or mass density distribution when the potential field is known then one can calculate the electrostatic or gravitational field, respectively. It is a generalization of Laplace's equation. Here, the **poisson()** function will generate a matrix which will be used in the numerical approximation of the second derivative of our function $u(x,y)$.

Additionally, it should be noted that in our problem, we do not assume any initial condition since the equation is not time dependent and so we

will approach it as a boundary value problem. Also, the midpoint rule will be used as a method of approximating an integral to obtain an approximate value of e using the implemented **l2norm** function. Then, we will account for boundary conditions and implement this for 1 thread, using grid size options of 2 and then 3 in later tests.

Here, we will use the following second-order finite difference stencil to approximate the points that are not considered boundary points:

$$\frac{1}{h^2}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{3}$$

with $1/h^2$ being a correction factor used in approximating the derivative. In other words, this stencil is expanded to account for our domain size Ω in such a way that when we a use it in the matrix vector product with the vector composed of points $u(x, y)$ we will generate a numerical approximation of $\Delta u(x, y)$ as shown here:

$$\frac{1}{h^2}\begin{bmatrix} -4 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 1 & -4 & 1 & \ddots & & & & \vdots \\ 0 & 1 & -4 & 1 & \ddots & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & 1 & -4 & 1 & 0 \\ \vdots & & & & \ddots & 1 & -4 & 1 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} g(x,y) \\ u(x,y) \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ u(x,y) \\ g(x,y) \end{bmatrix} = \begin{bmatrix} \Delta u(x,y) \\ \Delta u(x,y) \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \Delta u(x,y) \\ \Delta u(x,y) \end{bmatrix} \tag{4}$$

Scaled Matrix Vector Product Approximation of $\Delta u(x, y)$
**Note: Not all entries of the sparse matrix are included

Another way to represent the two dimensional five point stencil used here where $f(x, y)$ is any function with (x, y) inputs in 2D space is shown below:

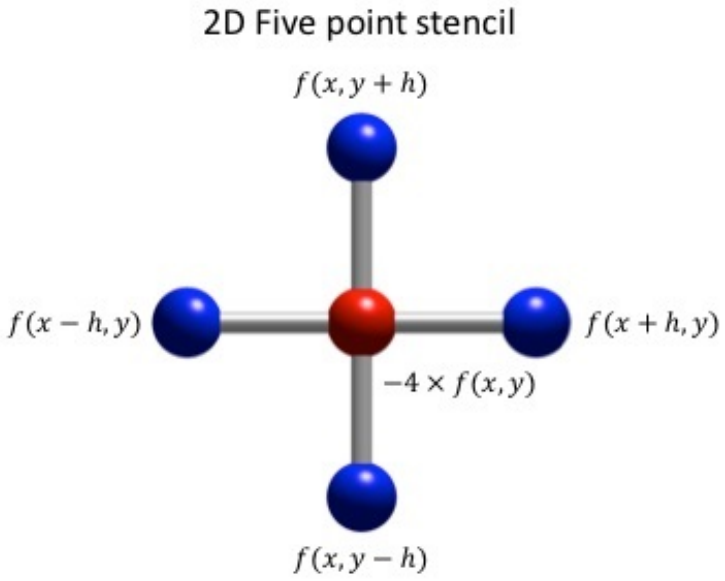


Figure 1: Five Point Stencil in 2D

We will apply this stencil to our uniform grid where the size of each square within the grid is h by h. In other words, we will apply the stencil to the following points: $(x-h, y), (x+h, y), (x, y-h), (x, y+h), (x, y)$. In this case, we evaluate our function $u(x, y)$ at the following points, we form a pattern known as a quincunx, which is a geometric pattern consisting of five points arranged in a cross with four of the points forming a square or rectangle with the fifth at its center. This process is reiterated for each of our grid points within Ω. As stated previously, this stencil will then be used to numerically approximate the Laplacian of our function $u(x, y)$ as shown in the following equation:

$$\Delta u(x, y) \approx \frac{u(x-h, y) + u(x+h, y) + u(x, y-h) + u(x, y+h) - 4u(x, y)}{h^2} \tag{5}$$

So for our domain Ω and the equation $\Delta u(x, y) = f$ to be solved, Ω is subject to Dirichlet boundary conditions. To do this we first generate a uniform Cartesian or equally spaced grid as depicted below.

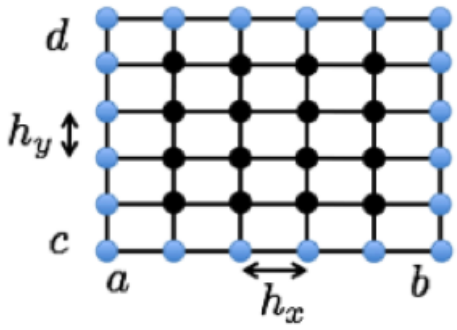


Figure 2: Graphical Depiction of Our Grid

Here, the blue points in the grid represent the outer boundary points in Ω and the black points represent the inner points in Ω. Also, we can evaluate the exact value of $\Delta u(x, y)$ by the following equation:

$$\Delta u(x,y) = -\frac{sin\left(\sqrt[3]{x+1}+\sqrt[3]{x+1}\right)}{9(x+1)^{4/3}} - \frac{sin\left(\sqrt[3]{y+1}+\sqrt[3]{y+1}\right)}{9(y+1)^{4/3}} +$$

$$\frac{2sin\left(\sqrt[3]{x+1}+\sqrt[3]{x+1}\right)}{9(x+1)^{5/3}} + \frac{2sin\left(\sqrt[3]{y+1}+\sqrt[3]{y+1}\right)}{9(y+1)^{5/3}} -$$

$$\frac{cos\left(\sqrt[3]{x+1}+\sqrt[3]{x+1}\right)}{9(x+1)^{4/3}} - \frac{cos\left(\sqrt[3]{y+1}+\sqrt[3]{y+1}\right)}{9(y+1)^{4/3}} -$$

$$\frac{2cos\left(\sqrt[3]{x+1}-\sqrt[3]{x+1}\right)}{9(x+1)^{5/3}} - \frac{2cos\left(\sqrt[3]{y+1}+\sqrt[3]{y+1}\right)}{9(y+1)^{5/3}} \tag{6}$$

This equation is equivalent to $\frac{\partial^2 u_{exact}}{\partial x^2} + \frac{\partial^2 u_{exact}}{\partial y^2}$ which will be used in calculating $e(h)$ along with the numerical approximation $\frac{\partial^2 u_{approx}}{\partial x^2} + \frac{\partial^2 u_{approx}}{\partial y^2}$. Here, we note again that the boundary condition $g(x, y)$ is $u(x, y)$ evaluated at the boundary point as represented by the blue points in Figure 2. After computing these values we are now ready to compute $e(h)$.

1.2 How

To compute $e(h)$, we used the code skeleton provided by the instructor of this course which includes several useful function skeletons as well as the

appropriate structure for the specified compute job. The portions of code that were implemented by us were marked as "Tasks" in several commented sections in the skeleton file. Here, we were required to insert the appropriate code to accomplish a specific task when prompted to by each comment.

In implementing each portion of the main computation loop, we set the number of timings to 10 for each thread count. This command takes the fastest timing out of 10 different timings when running the computation for each thread and stores this value in the specified array. This process will be implemented for each problem size combination i.e. for each of the different grid sizes as required by the homework tasks. In other words, this number in particular was chosen in an attempt to minimize time for each run of this program while maintaining some robustness in terms of countering operating system noise in measuring and comparing runs of this code.

With regards to the $compute\_fd$ function in the code skeleton, which was also populated with helpful task comments, the focus for this portion of the code implementation task was on ensuring that this function would work properly for a single-threaded run. Note that the $l2norm$ function skeleton provided necessary implementation info for computing the L2 norm integral of the error data that gets passed in as arguments along with the our specified grid spacing.

Additionally, we accounted for boundary conditions after the $compute\_fd$ is called during the main computation loop. This use of a boolean masking-like array allows for the masking of constructs in the Python $NumPy$ library and is showcased in the provided boundary condition code. This applies the boundary condition operation to each grid point whose Y value is along the upper boundary of our target domain. The general structure of this code was adhered to in handling the other three groups of grid points that lie along a boundary of the target domain. Finally, in regards to the serial version of our code, the data tables named *error* and *timings* are saved in the output files named *error.txt* and *timings.txt*, respectively.

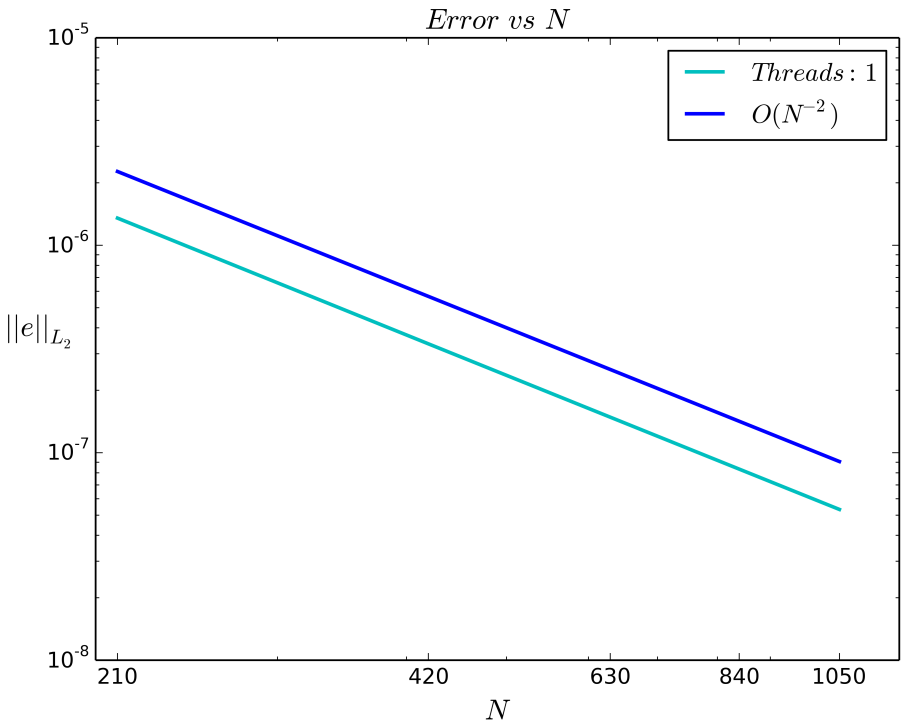


Figure 3: L2Norm Error vs Problem Size along 1 Dimension N

In another defined task, we interpreted the output of our given **poisson()** function located within the file *poisson.py* from the class repository. Here, we chose the grid size to be 3×3 where in which we get the following output for our CSR matrix in the Python 2.7 interpreter:

```
>>> from poisson import poisson
>>> A = poisson((3,3),format='csr')
>>> A.todense
matrix([[-4.,  1.,  0.,  1.,  0.,  0.,  0.,  0.,  0.],
        [ 1., -4.,  1.,  0.,  1.,  0.,  0.,  0.,  0.],
        [ 0.,  1., -4.,  0.,  0.,  1.,  0.,  0.,  0.],
        [ 1.,  0.,  0., -4.,  1.,  0.,  1.,  0.,  0.],
        [ 0.,  1.,  0.,  1., -4.,  1.,  0.,  1.,  0.],
        [ 0.,  0.,  1.,  0.,  1., -4.,  0.,  0.,  1.],
        [ 0.,  0.,  0.,  1.,  0.,  0., -4.,  1.,  0.],
        [ 0.,  0.,  0.,  0.,  1.,  0.,  1., -4.,  1.],
        [ 0.,  0.,  0.,  0.,  0.,  1.,  0.,  1., -4.]])
```

So, for a 3×3 grid of domain points, we see that each row in the **poisson()** matrix corresponds to a grid point. Looking at the stencil matrix for the upper left corner point in this $3x3$ grid, we see the following:

```
>>> A.todense[0]
matrix([[-4.,  1.,  0.,  1.,  0.,  0.,  0.,  0.,  0.]])
```

And in the following we are looking at the stencil matrix for the center point:

```
>>> A.todense[4]
matrix([[ 0.,  1.,  0.,  1., -4.,  1.,  0.,  1.,  0.]])
```

So here we can see that all five relevant stencil coefficients are captured in this stencil matrix, requiring no additional calculation.

1.3 Why

In this task we have successfully computed the Euclidean norm of the error $e(h)$ in the unit square domain $\Omega := (x, y) \in [0, 1] \times [0, 1]$ for the function $u(x, y) = \cos\left(\sqrt[3]{x+1} + \sqrt[3]{y+1}\right) + \sin\left(\sqrt[3]{x+1} + \sqrt[3]{y+1}\right)$. As you can see from the log-log plot of the L2 norm error is reduced for increasing N using the grid sizes.

Additionally, use of the *poisson.py* file function *poisson* was integral to numerically calculating the Laplacian of our target function. The above python interpreter commands and output illustrate the way in which the stencil matrices are calculated for a given desired grid size. In our example case of the 3×3 grid of domain points, we saw that each row in the **poisson()** matrix corresponds to a grid point. It should also be noted that for the corner boundary case an additional two function evaluations are needed to arrive at the correct Laplacian approximation. Also, we can see that all five relevant stencil coefficients are captured in this stencil matrix, requiring no additional calculation. This is due to the fact that this stencil matrix's corresponding grid point is in the center of our grid. Also, the first two entries in the above python interpreter output correspond to the stencil coefficients for the target point and its neighbor point to the right. The fourth entry corresponds to the stencil coefficient for the point just 'below' our target point.

In summary, from our plot of the L2-norm error values for problem sizes of option 2 along with a reference quadratic plot, we can clearly see that we converge to the actual value by the $O(h^2)$. This can be stated since, as we have established previously, the relationship relating h to N is such that $h = \frac{1}{(1-N)}$ and our plot shows that we converge in the manner of $O(N^{-2})$. Therefore, we can say that we have an order of $O(h^2)$ accuracy using the 5-point stencil approach in approximating the value of $\Delta u(x, y)$.

2 Task 2: Using Python Threading Module to Parallelize Our Serial Code

2.1 What

Now we will use the Python threading module to parallelize our serial code using nt threads to compute the value of $e(h)$, where nt is a list of thread counts to be employed in our main computation file $hw4_driver.py$. It is important to note that parallel machines are usually grouped into two categories: Shared Memory and Distributed Memory. In shared memory, the computer architecture is such that all processors have direct access to common physical memory as a global address space and multiple processors can operate independently but share the same memory resources. In Distributed Memory, computer architecture is structured such that each processor has its own local memory and operates independently, and the memory addresses in

one processor do not map to other processors.

The Shared Memory machines can classified as Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA). In UMA, we have identical processors that have equal access times to memory and have cache coherency. In other words, if one processor updates a location in shared memory then all the other processors know about the update immediately. Whereas in NUMA, we link multiple UMA machines and so not all the processors have equal access time to all memories. So, in this case, the processor can access its own memory faster than non-local memory. In other words, memory access across links is slower and NUMA has a significant overhead in maintaining cache coherency across shared memory.

In the Distributed Memory case, there is no concept of global address space across processors and the distributed memory system requires a communication network to connect between processor memory. Here, the concept of cache coherency does not apply since the changes made by each processor to its local memory does not have an effect on other processors. In this case, the programmer must explicitly define data utilization parameters such as how and when to communicate data between processors.

For Wheeler, the supercomputer used in this assignment, we are using a hybrid of the two memory cases when the thread number size exceeds the number of processors within each compute node. For the purposes of this homework, we will not exceed thread numbers greater than CARC's compute nodes i.e. we will not use thread sizes greater than 8 to prevent oversubscribing.

2.2 How

Parallelization of the main compute procedure was implemented by populating areas in the main computation loop skeleton marked with "Task" with the appropriate thread start and join commands that iterate through the list of thread objects. Here, the $compute\_fd$ function was reworked to account for multiple threads, as its first implementation accounts only for thread counts of 1, ignoring task domain partitioning entirely. In this part, the main point of difficulty involved ensuring that the desired output locations for the computation results for each thread were being referenced correctly. Another point of difficulty that we encountered stemmed from the observation that, while the documentation for the function assumes that the number of threads divides evenly into the number of rows in the problem do-

main, there are problem sizes in the skeleton that are not divided into evenly. For example, in the option where the problem size vector "NN" is a list of the first five multiples of 210, the thread count of 4 does not divide evenly into odd multiples of 210. So, to account for this, a fix was implemented for the cases where even divisions of problem size into thread count do not exist.

To demonstrate that our parallelization of the $compute\_fd$ function remains consistent among different threads, we present the table of $L2norm$ error values below:

	$L2Norm\ Error\ by\ Problem\ Size\ (s)$				
$Threads$	$N^2 = 210^2$	$N^2 = 420^2$	$N^2 = 630^2$	$N = 840^2$	$N = 1050^2$
1	$1.351660e-06$	$3.348363e-07$	$1.483641e-07$	$8.333115e-08$	$5.329387e-08$
2	$1.351660e-06$	$3.348363e-07$	$1.483641e-07$	$8.333115e-08$	$5.329383e-08$
3	$1.351660e-06$	$3.348362e-07$	$1.483641e-07$	$8.333113e-08$	$5.329388e-08$
4	$1.351660e-06$	$3.348363e-07$	$1.483641e-07$	$8.333114e-08$	$5.329360e-08$

Table 1: L2 Norm Error By Problem Size N^2 and Thread Count 1 through 4

This table illustrates how the convergence of the $l2norm$ of the absolute difference of the numerical and analytical solution approach 0 as we increase the problem size N. Here, we expect that this term converges to 0, meaning that we can closely approximate the actual value, on the order of $O(h^2)$. In this case, $h = \frac{1}{n-1}$ where n is the number of grid points along one dimension and so for the 2 dimensional problem domain N, the number of grid points to take into account for the second dimension, is N^2.

It is interesting to note that the error values among each thread of equal problem size are not entirely equivalent in that we obtain slightly different values depending on thread size. Unfortunately, it was not clear why this was the case, though, upon consulting the course professor, the error differences, which fluctuate roughly between values 10^{-13} and 10^{-16}, were determined to come from the way our implementation handles boundary conditions.

It may be the case that it is possible that there are some library routines that are threaded and being run underneath $SciPy$ away from our view. If this is the case, then it is possible that some computations are being performed in a different order which is affecting the final output. Specifically, the python library $NumPy$ comes equipped with useful Boolean masking-like features that were used in dealing with the boundary condition handling. The provided code skeleton included code that handled one boundary of our domain whose structure we used to handle the other three boundaries. It

is not known precisely why this non-determinism exists in using this masking, though it had been hypothesized that $NumPy$ may be handling these masking-like function calls in a concurrent way, resulting in non-deterministic behavior when considering the non-associativity of floating-point arithmetic operations.

In any case, for this problem, we chose to not re-implement the boundary condition handling in a strictly serial way, although this non-determinism in our numerical approximation should be eliminated in future finite differencing applications, especially those that would necessitate high degrees of precision.

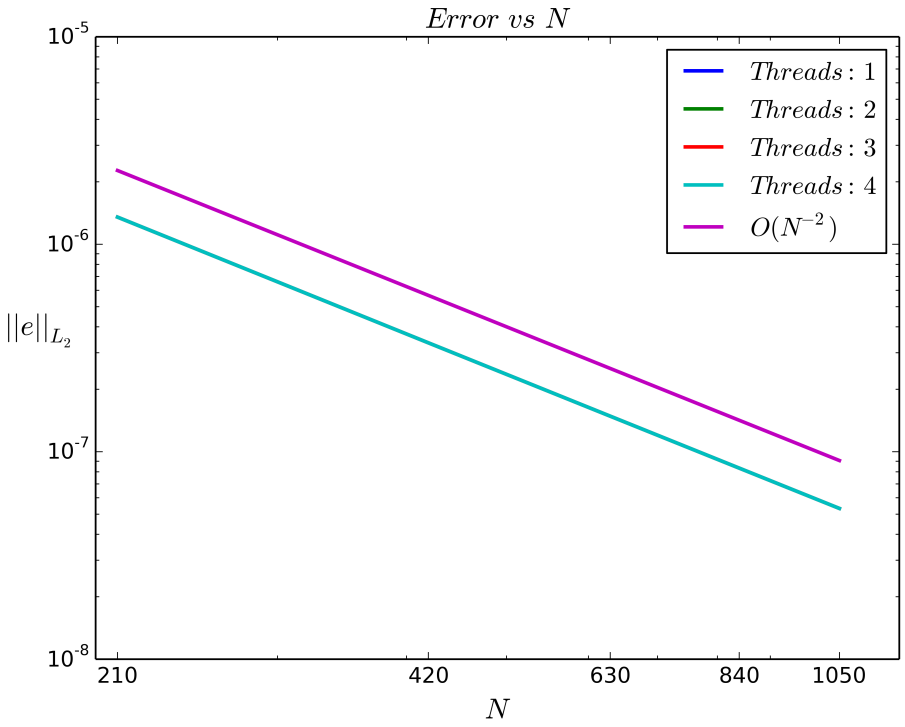


Figure 4: L2-Norm Error vs N by Thread Size

Above we have included a plot of the previous error table. Note: Only 2 distinct plots can be visualized due to the insignificantly small differences in error value approximations among thread options. Here, we cannot visualize each thread value on the log-log scale provided while still maintaining view of the reference plot which is suggesting our order of convergence for each thread to be $O(N^{-2})$ or $O(h^2)$.

	Compute Times by Problem Size (s)				
Threads	$N = 210^2$	$N = 420^2$	$N = 630^2$	$N^2 = 840^2$	$N^2 = 1050^2$
1	1.349401e-02	6.261992e-02	1.503711e-01	2.738800e-01	4.250271e-01
2	1.336598e-02	5.196810e-02	1.250830e-01	2.275031e-01	3.717098e-01
3	1.071000e-02	4.719782e-02	1.084061e-01	2.053111e-01	3.361671e-01
4	1.235199e-02	4.519701e-02	1.073129e-01	1.952641e-01	3.184271e-01

Table 2: Compute Time By Problem Size N and Thread Count 1 through 4

Since the 5-point stencil gave us an order of approximation of $O(h^2)$ for the L2 norm, one wonders if it is worth going to higher order to take $\Delta u(x, y) = f$ and truncate the finite difference approximation using center differences at the next order such that:

$$\Delta u(x,y) \approx \frac{1}{h^2}\Big[(4/3)u(x+h,y) + (4/3)u(x-h,y) + (4/3)u(x,y+h) +$$

$$(4/3)u(x,y-h) - (1/12)u(x+2h,y) - (1/12)u(x-2h,y)$$

$$-(1/12)u(x,y+2h) - (1/12)u(x,y-2h) - 5u(x,y)\Big] \tag{6}$$

Although there will be an error of $O(h^4)$, this is not a popular method because it renders too many points as near boundary points on a square grid requiring additional computations increasing computation time and decreasing time efficiency. In other words, this method will be more computationally expensive when solving. Here, we have provided a representation of the 9-point stencil that would be used in this case:

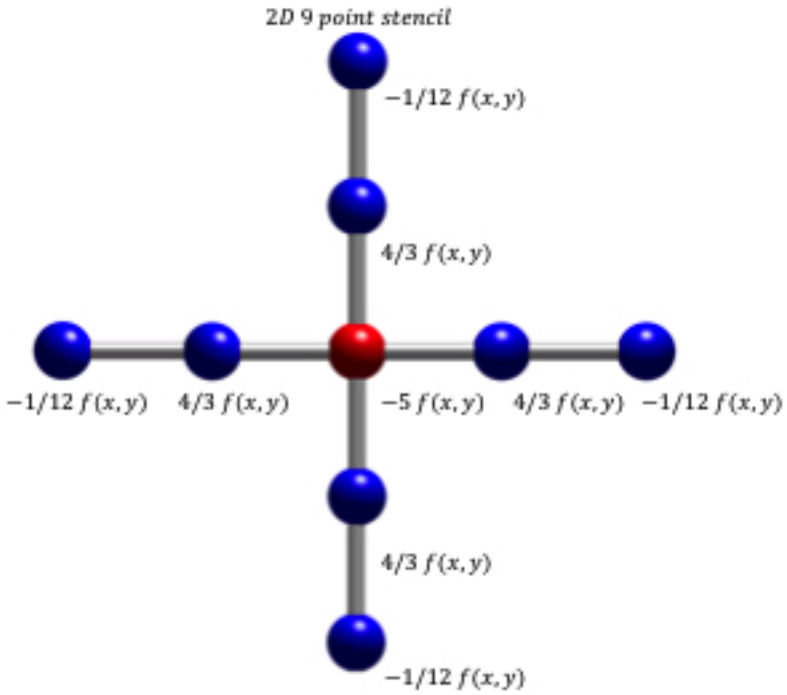


Figure 5: 2D 9-point stencil

Interestingly, the 9-point formula where we use a computational cell that includes all adjacent points, or points one could consider as having a Chebyshev distance of 1 with respect to the target point, in the 2D domain case is also $O(h^4)$, similar to the previous 9-point stencil. This formula takes the form of

$$\Delta u(x,y) \approx \frac{1}{h^2}\Big[(2/3)u(x+h,y) + (2/3)u(x-h,y) + (2/3)u(x,y+h) +$$

$$(2/3)u(x,y-h) + (1/6)u(x+h,y+h) + (1/6)u(x-h,y+h) +$$

$$(1/6)u(x+h,y-h) + (1/6)u(x-h,y-h) - (10/3)u(x,y)\Big] \qquad (6)$$

So here we can improve the convergence rate by this method of including the other nearest neighbor points around the point $u(x,y)$ while not increasing the number of boundary conditions to account for. The previous method mentioned is also desirable in this case.

2.3 Why

Parallelization of some computation should not change the results of the computation. As we have mentioned above, we have observed non-deterministic behavior in our runs that can be attributed to latent $NumPy$ implementation details. Taking into account that we might want to ensure accuracy in our results, further work could be done in eliminating this non-determinism in our implementation of this computation. Thankfully, despite this, we have observed a preservation of quadratic error convergence through parallelization of our code.

By looking at the timing table above, we can see that there is no clear compute time reduction through parallelization for problem size $N = 210^2$. This can be attributed to operating system noise i.e. the set of processes being handled by a computer's operating system that may interfere with a program's execution time. There are a number of strategies to combat this noise. One of these strategies is already employed and involves taking multiple measurements of the same compute job which saves the minimal run time out of the set of run times. Another obvious strategy would include minimizing the number of active user accessible processes. Also, a constant time cost operations such as thread start up might have a noticeable effect in the run time, especially considering the small initial problem size of $N = 210^2$ as well as the fact that no compute time in the timing table was above half of a second.

In conclusion, from the results shown in the timing table, we observed that as the problem size increases there is a decrease in compute time. For most cases, as thread count increases, we also observe a decrease in compute time. For example, in the case where problem size $N = 1050^2$, there is a clear decrease as thread count increases. However, we do observe an unaccountable increase in compute time for problem size $N = 210^2$ using 4 threads. This may be due to the possible issues mentioned previously where the processing speed is negatively influenced by noise or other unseen processes. In the next section, we will carry out a strong scaling study to more rigorously analyze the behavior of compute time as problem size increases for each thread count option.

3 Task 3: Strong Scaling Study Using One Node of Wheeler at CARC

3.1 What

In the context of high performance computing, the two most common notions of scalability are namely strong scaling and weak scaling. In strong scaling, we determine how the solution time varies with the number of processors for a fixed *total* problem size. In weak scaling, we determine how the solution time varies with the number of processors for a fixed problem size *per processor*. In this task, we will focus on strong scaling, comparing how the solution time changes as the number of threads used to carry out the computation increases, for a single problem size $N = 5040^2$

Strong scaling is an appropriate parameter to measure in instances where the program may have a long run time or, in other words, is CPU-bound. A possible goal may be for us to find the optimal number of threads or processors that allows for the computation to complete in a reasonable amount of time while not wasting too many cycles due to the parallel overhead associated with parallel computing. So, by conducting a scalability study, we can determine the parallel efficiency or how far we deviate from the "ideal" speed up using weak or strong scaling.

To calculate strong scaling efficiency, we use the following equation:

$$E_p = \frac{T_0}{T_p p} \tag{6}$$

with the amount of time required to complete one work unit with 1 processor T_0 and the amount of time to complete the same unit of work with p processors is T_p. Here, a work unit is the calculation of the numerical approximation of our target function's second derivative or Laplacian.

3.2 How

We carried out the strong scaling study on a single compute node within the Wheeler compute cluster at the Center of Advanced Research Computing (CARC). For each compute node at the Wheeler CARC machine, there is a maximum total of 8 processing core units. In comparison to running the study on a local machine, such as a personal computer which typically has

either 2 or 4 processing cores, the biggest difference is in carrying out this computation is doing so via a remote compute cluster. This involves the use of a Portable Batch System (PBS) batch script to schedule compute jobs in this case on the CARC Wheeler machine. The file used to execute our scaling study, *hw4.pbs* was given to us but the course instructor and can be found in the class assignment repository. The only necessary edit that needed to be made in the given file was changing the name of the python file that was used to carry out the computation. In this case, our file was named *hw4_driver.py*.

For the strong scaling study, our problem size N was equal to 5040^2, which means that, for each thread count option, we evaluated $e(h)$ using approximately *25.4 million* grid points. Since the Wheeler compute nodes have a maximum of 8 processing cores, the upper bound for the thread count will be 8. We iterate from a minimum thread count of 1 to a maximum of 8, keeping the minimal run time for each option out of 10 runs for each thread count option. Here, we have include plots of both the computation time verses the number of threads created to parallelize the job as well as the efficiency verse thread count.

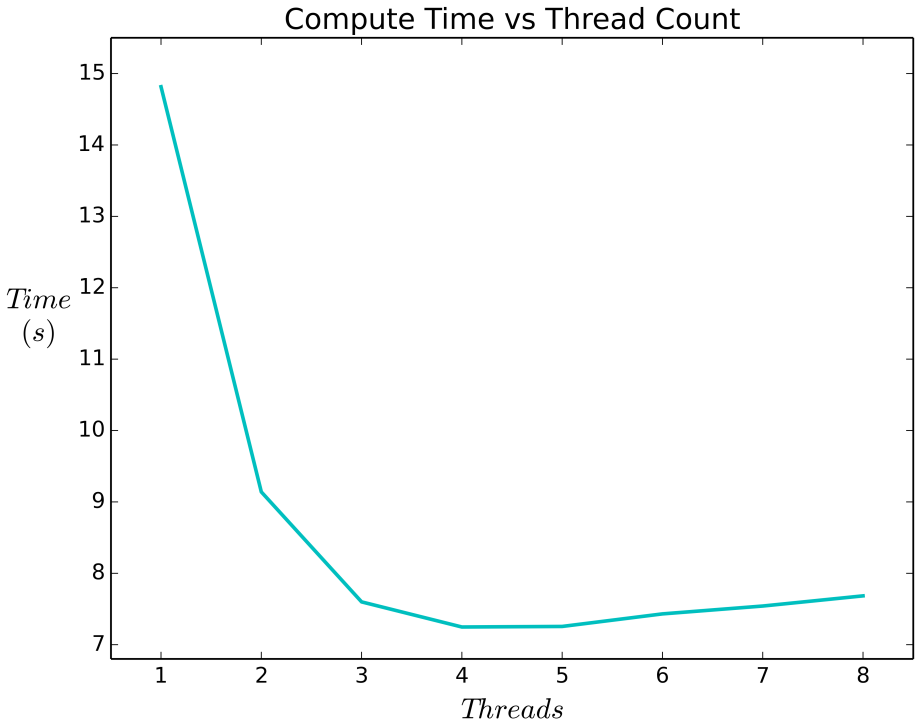


Figure 6: Compute Time vs Thread Count

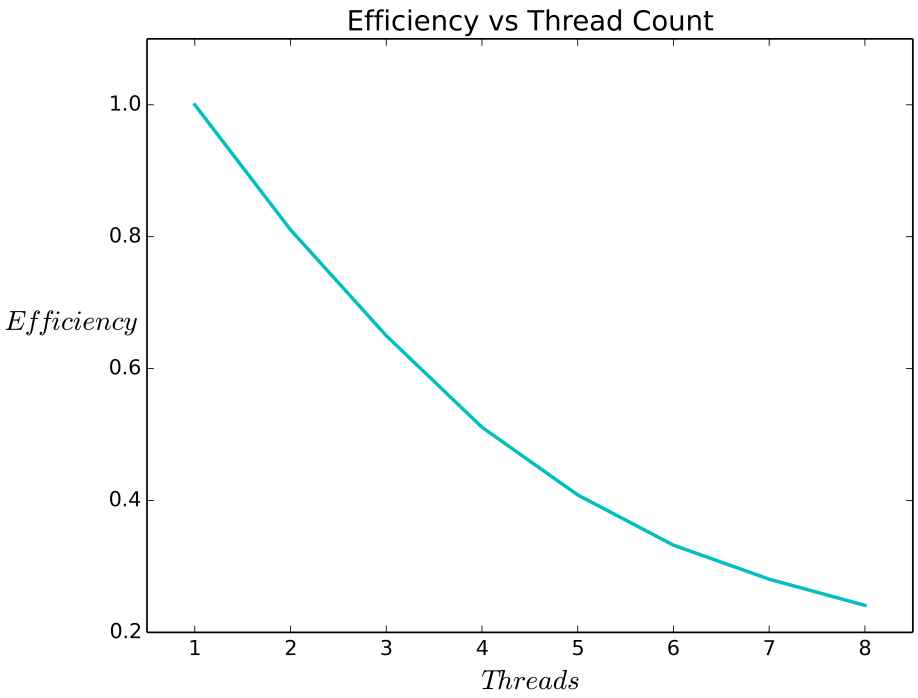


Figure 7: Efficiency vs Thread Count

3.3 Why

In our compute time plot, it is apparent that parallelization is immediately advantageous, offering a compute time difference of approximately 6 seconds when increasing thread count from 1 to 2. The compute time continues to decrease until, interestingly enough, the minimal compute time out of all runs is reached at 4 threads. When the number of threads is more than 4, compute time slightly *increases*, reaching a local maximum with a thread count of 8. An ideal speedup would not reduce the efficiency value and efficiency would remain at 1.

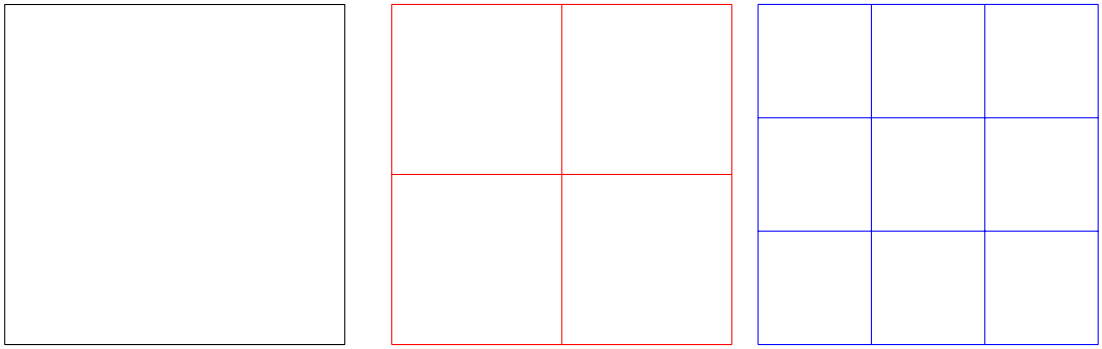
Observing ideal efficiency corresponds to reaching the theoretical computate speed-up through parallelization as defined by having an efficiency value of 1 for all thread count options. In practice, it is rare that the efficiency is always calculated to be a value of 1 for all threads counts. In our

efficiency plot, we see that, as the number of threads increases, the efficiency value decreases, steadily deviating from the ideal efficiency value of 1. This is consistent with the fact that machine limitations will generally result in a deviation from the ideal speed-up of computation. Additionally, there is a significant portion of our code that was not parallelized, which also resulted in deviation from ideal speed-up. Here, a thread count of 8 has the lowest efficiency value, while the highest parallel efficiency value belongs to the case where 2 threads are used in carrying out the computation.

It is not clear why parallelization stopped resulting in a decrease of computing time. For this problem, we do not require or specify that any thread should communication information to another thread. Additionally, there are no issues with the number of physical cores being outnumbered by the number of active threads being run. This phenomenon where the number of threads exceeds the number of compute cores is known as thread oversubscription. The possible reasons that may explain why the compute time begins to increase while using 5 or more threads could be that there are hardware limitations that negatively affect compute time as well as the unseen NumPy implementation code. The details of this notion are beyond the scope of this assignment and would be more appropriately discussed by using computer hardware and performance tests.

4 Task 4: Extra Credit - 2D Domain partitioning

4.1 Display of a potential 2D Domain partitioning



$Threads = 1$ $Threads = 4$ $Threads = 9$

Figure 8: 2D Domain partitioning for $Threads = 1, 4$, and 9

Above, we have provided depictions of the 2D domain partition case for square Thread sizes of $1, 4$, and 9. This is how one would approach designing each thread domain given the thread sizes. However, we did not implement this procedure due to time constraints. We did consider this problem and speculate that there will be oversubscribing in the $Threads = 9$ case but without implementation we cannot comment on time efficiency.

5 Summary

In this homework we have implemented and experimented with threaded programming in Python. We did this in the context of computing the L2 norm of $e(h)$. To accomplish this, we needed to use a second-order finite difference stenciling approach to approximate the points of $\Delta u(x, y)$ to obtain values of $e(h)$ which required computing numerical derivatives and integrals. After doing this for the serial case, we then parallelized our code and subsequently conducted a strong scaling study. To do this, we first developed Python code on our local machines and then used the course provided code which allowed us to carry out a parallelized version of our code. The strong scaling study was conducted using compute nodes of UNM's CARC Wheeler

supercomputer and helped us to determine the efficiency of computation with respect to time.

Potential future directions on this topic include exploring parallelization which requires more than one compute node, in this case when the thread size is greater than 8. Also, gauging the memory and compute time costs of using different stencil methods, as mentioned in section 2.2. Additional topics of interest include exploring finite differencing with functions of more than 2 variables, other transcendental and polynomial functions, along with different approaches to numerical differentiation such as differential quadrature.

All in all, we found that we get reduced efficiency when we parallelize our code to calculate $e(h)$ but benefit from the shortened compute time as indicated by our strong scaling study. Here, we see that parallelization comes at the cost of writing code that is more complex, but appears to be quite useful in real-world applications. When working with large data sets or problem sizes, we may often rely on the time reduction benefits that come with parallelization to compute jobs in a reasonable time frame.

6 References

http://people.bu.edu/andasari/courses/Fall2015/LectureNotes/Lecture14$_2$7Oct2015.*pdf*

https://www.sharcnet.ca/help/index.php/Measuring$_P arallel_S caling_P erformance$

http://math.unm.edu/ schroder/2018$_F all_4$71/lec15$_1$0$_9$2018.*pdf*

https://math.oregonstate.edu/ restrepo/475B/Notes/sourcehtml/node52.html

Differential Quadrature and Its Application in Engineering: Engineering Applications, Chang Shu, Springer, 2000, ISBN 978-1-85233-209-9