HW4

Deep Patel
Jacob Hurst

October 27, 2018

1 What

The overall goal of this assignment is to implement and experiment with threaded programming shared memory in Python to compute numerical derivatives. We conducted a strong scaling study for computing numerical derivatives. The scaling studies were carried out on CARCs compute cluster. This is because the computations are demanding and CARC compute nodes have sufficient cores for the scaling studies.

2 How

Objective: After completing this assignment we will have written a Python script to carry out numerical derivation on our function using multiple threads shared memory, measure error by taking the l2norm, generate data regarding error as our problem size increases timing/efficiency as our number of threads increases.

The parallel computation was carried out on Wheeler at CARC and our local machine. We used our local machine for options 2 and 3 (less computationally demanding) to generate plots on error as problem size increases and Wheeler for option 1 (computationally demanding) to carry out our strong scaling study (timing/efficiency as number of threads increases).

On our local machine, we ran *python hw4skeleton.py* to run our script. On CARC, *hw4.pbs* was used to submit script to be run with the Wheeler scheduler.

Our function of interest is

$$f(x,y) = cos((x+1)^{1/3} + (y+1)^{1/3}) + sin((x+1)^{1/3} + (y+1)^{1/3})$$

in the domain of the unit square.

$$\|e_{L_2}\| = \left\{ \int_0^1 e^2 dx \right\}^{1/2} \tag{1}$$

The error is taken as the l2norm as shown in equation 1, which is approximated in our code as $\sqrt{h^2 \sum e^2}$, where $e = f_{approx.} - f_{exact}$ and equivalently as

$$e(h) = \left\{ \int_{[0,1]X[0,1]} h^2 \left(\frac{\partial^2 u_{approx}}{\partial x^2} - \frac{\partial^2 u_{exact}}{\partial x^2} + \frac{\partial^2 u_{approx}}{\partial y^2} - \frac{\partial^2 u_{exact}}{\partial y^2} \right)^2 dxdy \right\}^{1/2} \tag{2}$$

There are four code files: $hw4.pbs$, $hw4_skeleton.py$, $threading_example.py$, and $poisson.py$.

2.1 Preliminary Threading example

The threadingexample.py file was used to test the threading of the code which resulted in the desired effect: reduction of computation time with increase in number of threads.

For number of threads = 1, min time taken is: 0.219505786896. For number of threads = 4, min time taken is: 0.118069887161.

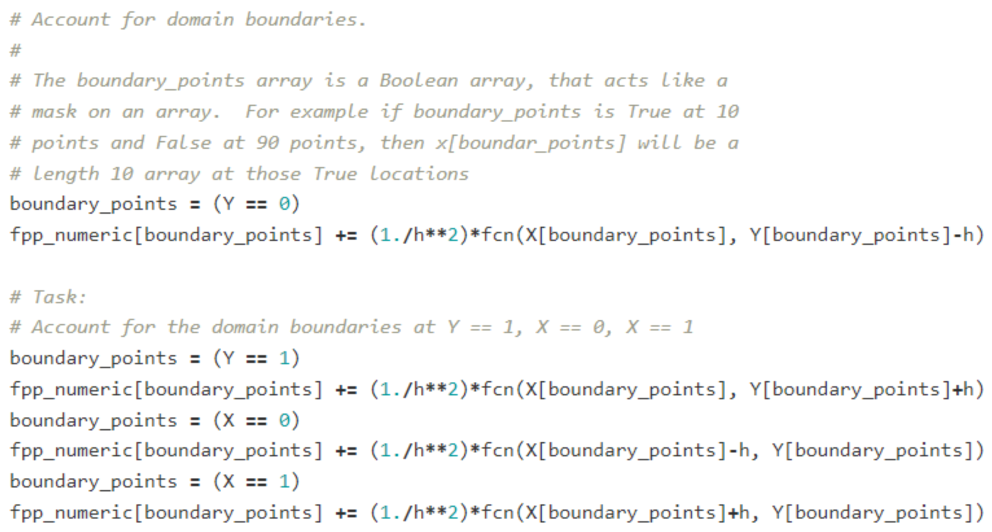
3 Second Order Differential stencil setup

The Second Order Differential stencil was set up to compute the numerical derivative of the function. Thereafter, the derivative is compared to the exact analytic derivative in order to compute the error evaluated by equation 2.

The provided *poisson.py* function was used to generate the CSR matrix which was then scaled by h^2.

The midpoint rule is used to compute the integral in equation 2. An l2norm function is used to do this: which can be understood by equation 1. In essence, this integral is like a norm over our domain. We are squaring the value of the function at each grid point, summing, and then taking a square-root. Its very similar to the Euclidean norm, except we scale by h.

The domain boundaries are accounted for by using the code as shown below in Figure 1



```
# Account for domain boundaries.
#
# The boundary_points array is a Boolean array, that acts like a
# mask on an array.  For example if boundary_points is True at 10
# points and False at 90 points, then x[boundar_points] will be a
# length 10 array at those True locations
boundary_points = (Y == 0)
fpp_numeric[boundary_points] += (1./h**2)*fcn(X[boundary_points], Y[boundary_points]-h)

# Task:
# Account for the domain boundaries at Y == 1, X == 0, X == 1
boundary_points = (Y == 1)
fpp_numeric[boundary_points] += (1./h**2)*fcn(X[boundary_points], Y[boundary_points]+h)
boundary_points = (X == 0)
fpp_numeric[boundary_points] += (1./h**2)*fcn(X[boundary_points]-h, Y[boundary_points])
boundary_points = (X == 1)
fpp_numeric[boundary_points] += (1./h**2)*fcn(X[boundary_points]+h, Y[boundary_points])
```

Figure 1: This figure shows the creation of a "ghost" boundary of points around the domain boundaries.

Moreover, different grid size options were used to verify the quadratic convergence of the error. In figure 2 a serial thread log-log plot of the error is provided as it is reduced for increasing n using the grid sizes of option == 2: NN = 210*arange(1,6). NN is an array of grid sizes to loop over. e.g. if NN equals [6,12] and num-thread=[1,2] then there will be total of 4 combinations of grid sizes and threads. arange(1,6) gives [1,2,3,4,5] which is used for increasing grid size.

The verification for quadratic convergence is verified for 2 and 3 threads in 3 and 4 respectively.

For the 36 x 36 grid size, the selected output of the CSR matrix, A is provided: a row corresponding to the top right corner point:

A.shape = (36, 36)

A[0,:].indices = [0 1 6]

A[0,:].data = [-100. 25. 25.]

We see that the left-most point of the leading diagonal of A, corresponding to index 0 is the center of the stencil: -100, the point right to it is 25 corresponding to index 1, the point that is n-2 (4) indices to the right in A is the point directly below the center of the stencil and has value 25.

Moreover, a row corresponding to a point in the center of the domain results in:

A[16,:] indices = [10 15 16 17 22];

A[16,:] data = [25. 25. -100. 25. 25.]

We see that for the inner point, the value corresponding to the leading diagonal matrix (index of 16 for row 16) is -100, the 25 to the right and left corresponding to indices 15 and 17 are the points to the right and left of the stencil center. The points that are n-2 (4) indices above and below of of the center point correspond to stencils 10 and 22, and have values of 25.

4 Parallelizing the code by 1D domian partition

Let k be the local thread number, nt be the global number of threads, and n be the number of points in each dimension. Then, each thread 'owns' the [k(n/nt) : (k+1)(n/nt)] rows of the domain.

In addition to the local row-ranges we computed the halo region for each thread. A halo region essentially expands a threads local domain to include enough information from neighboring threads to carry out the needed computation. Each thread's output is to assign entries in *fpp-numeric* over that threads local range of rows.

4.1 Similarity of parallel and serial code result for varying threads

We demonstrate that our parallel code gives the same result as the serial code, for thread numbers 1 through 4 by regenerating a log-log plot of the error for each thread number. The problem sizes used were specified by option 2 in the provided homework skeleton.

As seen in figures 2 through 4, the increase in number if threads has negligible effect on the reduction of error.

We observe that the error reduced by an order of magnitude quadratically with increase in grid size.

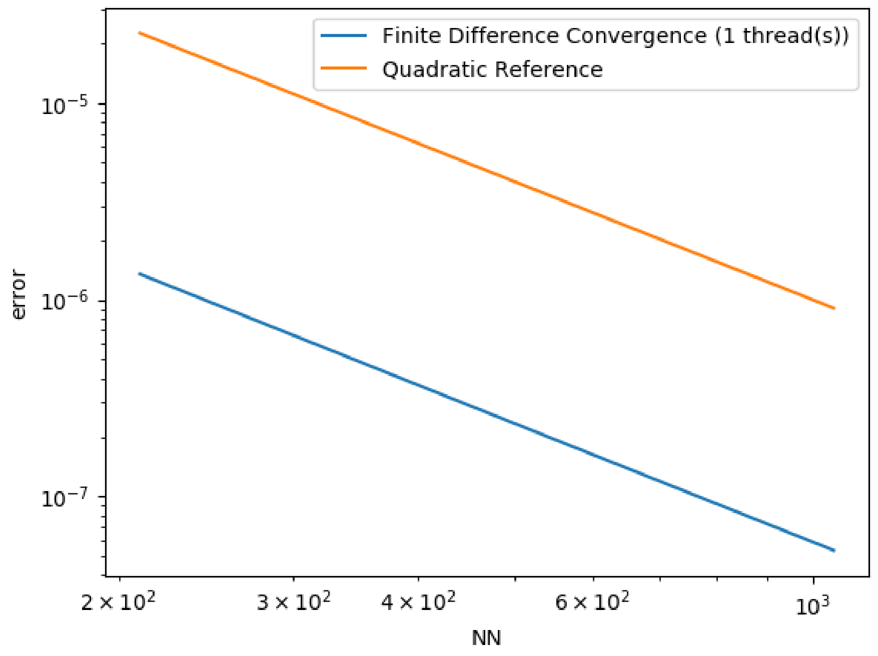


Figure 2: This figure shows reduction in error as our problem size increases on 1 thread.

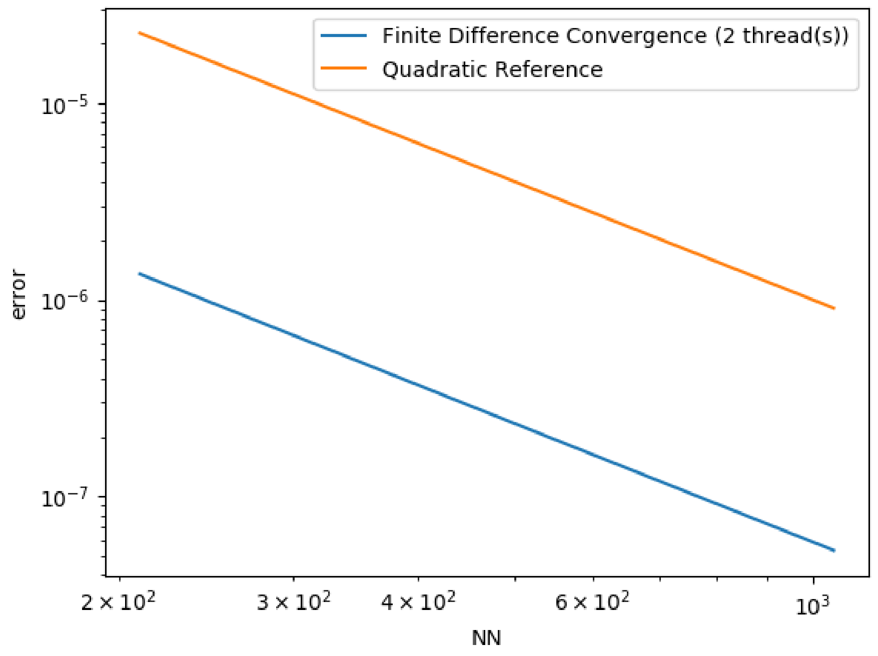


Figure 3: This figure shows reduction in error as our problem size increases on 2 threads.

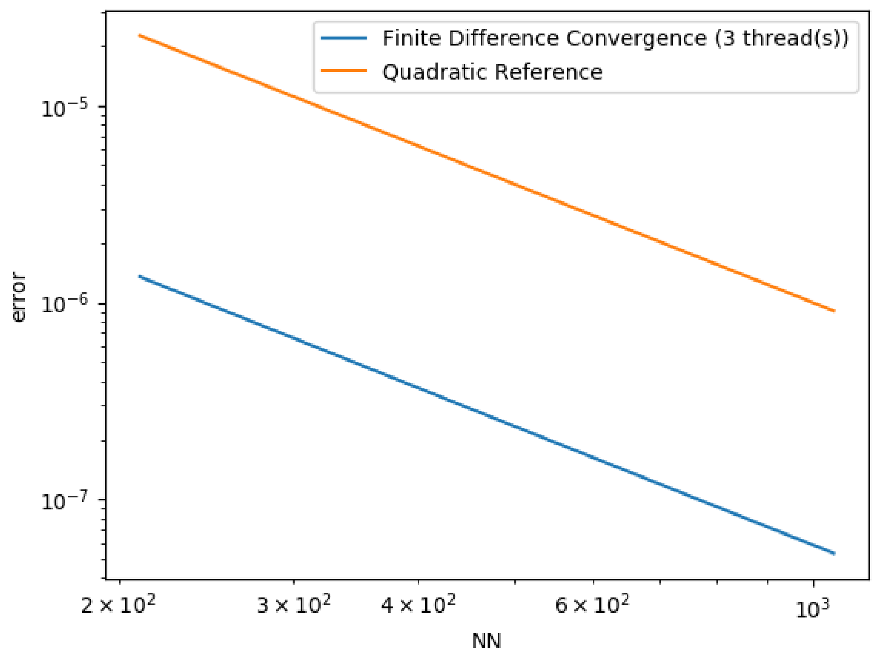


Figure 4: This figure shows reduction in error as our problem size increases on 3 threads.

4.2 Preliminary timing of matrix-vector product

Some preliminary timing information is generated by inserting timing statements to time your computations for varying number of threads. The instructor-indicated locations in the skeleton code for your timing locations were used. The calculation of interest is parallelization of a matrix-vector product, which is the heart of many, many numerical simulations. The timing results are shown below.

1. Minimum time on 1 thread: 0.659749078751 (option 2 with 10 timing samples).
2. Minimum time on 2 threads: 0.3865599823s (option 2 with 10 timing samples).

3. Minimum time on 3 threads: 0.331065883636s (option 2 with 10 timing samples).

We observe via the preliminary timings that increasing our threads from 1 to 2 decreases our time by 41.4 percent and increasing our threads from 1 to 3 decreases our time by 49.8 percent.

4.3 Effect of increasing Halo region

To carry out this matrix-vector, you must have access to information (here function values) for one row above, and one row below in the domain. This is denoted with the gray in Figure 5. Then, a matrix-vector with addition, as depicted in Figure 5, will give that thread's local contribution to the global computation.

Therefore, using the fourth order stencil will increase the stencil points by one in all directions: top, bottom, right and left. However, the halo region will only only be affected by the addition of data dependency of points in the top and bottom of center. If each thread owns four rows, as shown in Figure 5, the halo region will increase from 4 points above and 4 points below to 8 points above and 8 points below. This will increase data dependency from two threads (for second order stencil) to four threads (for fourth order stencil).

$$\frac{1}{h^2}\begin{bmatrix}
-4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4
\end{bmatrix}
\begin{bmatrix}
u_{0,0} \\ u_{1,0} \\ u_{2,0} \\ u_{3,0} \\ u_{0,1} \\ u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{0,2} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \\ u_{0,3} \\ u_{1,3} \\ u_{2,3} \\ u_{3,3}
\end{bmatrix}
+ \frac{1}{h^2}
\begin{bmatrix}
g_{0,-1} + g_{-1,0} \\ g_{1,-1} \\ g_{2,-1} \\ g_{3,-1} + g_{4,0} \\ g_{-1,1} \\ 0 \\ 0 \\ g_{4,1} \\ g_{-1,2} \\ 0 \\ 0 \\ g_{4,2} \\ g_{-1,3} + g_{0,4} \\ g_{1,4} \\ g_{2,4} \\ g_{4,3} + g_{3,4}
\end{bmatrix}$$

Figure 5: This figure shows the halo region (gray) showing data dependency of the highlighted thread (blue).

5 Scaling Study

5.1 Scalability

Scalability refers to the ability of a parallel system (software and/or hardware) to demonstrate a proportionate increase in parallel speedup with the addition of more resources.

Perfect scaling results in a runtime reduction of 1/NN.

We measure performance in timing and efficiency on the range of 1 to 8 threads. In Figure 6 we observe that from 3 to 6 threads we observe a nearly level time of 7.5. From 6 to 8 threads we observe a slight increase to a time of 8.

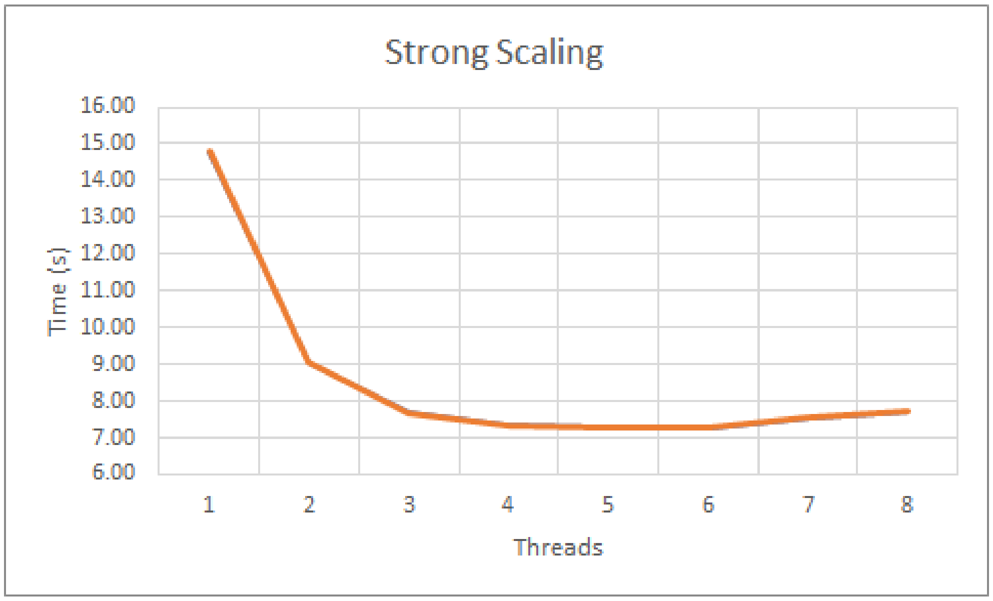


Figure 6: This figure shows the reduction of computation time of the matrix-vector operation with increase of threads and the resulting constant time at higher than 4 threads.

5.2 Efficiency

Parallel efficiency measures how far you deviate from ”ideal” speedup. You can measure efficiency for weak or strong scaling.

In Figure 7 We observe the efficiency by threads and can notice the curve begins to level out towards the halfway mark (4 threads).

This is reflected in the efficiency plot, when we reach the 30 percent mark on efficiency by threads, we observe a leveling of the curve of our function as number of threads increases.

It appears the available work to parallelize for option 1 (largest problem size) gets swamped by machine limitations in this range, particularly, at 4 threads.

Memory bandwidth is diminishing above thread 5 because the overhead in creating threads and allocating a task for these threads is beginning to outweigh the speedup offered by the use of these threads.

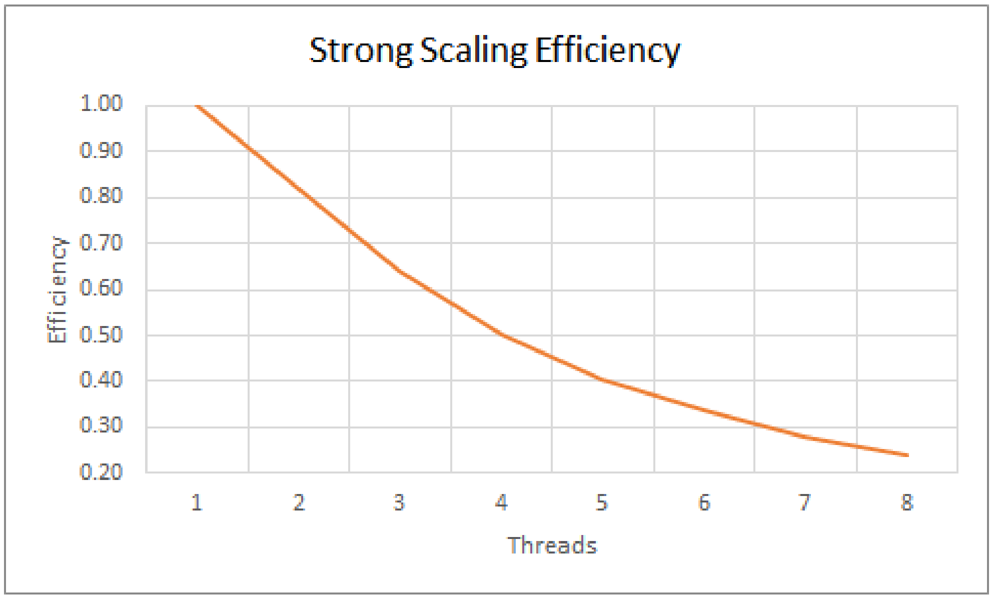


Figure 7: This figure shows the reduction of scaling efficiency with increase of thread size