HW6 Report for Math/CS 471

Jeff and Xiaomeng

11/27/2017

**Abstract**

This is the HW6 Report. First, the report will give a brief introduction of Math background. Second, the report will outline the procedure and methods we used. Third, we will introduce the data results and try to analyze and discuss them. In the end, we will give a conclusion about this Math problem. This time we use Microsoft Word to write this report.

**Introduction**

In a logically rectangular domain, I will compute derivatives and integrals. The main logic is that I can treat this rectangular domain as a reference domain for my problem, denoted by . = . Using a mapping function , I can map any function . This method allows me to compute derivatives and integrals from the reference domain to get the results of real domain, i.e. from to .

1. Differentiation on

is a continuously differentiable function of x and y, where . In order to approximate the partial derivative of u with respect to x and y, I will first write functions and as functions of r and s, where . Then I can use the chain rule and come up with the following equation:

Then the calculation is transferred to reference domain. With the Cartesian grid on the reference element = and use the standard finite difference formulas, I can get as well as . It is easy to use standard finite difference formulas to get , , , with the known mapping results and . Therefore, I can have , , , with the following equation.

Therefore,

1. Integration on

It is accessible to use this reference domain to compute integrals with the equation below:

Where is the surface element

Since the domain relates both r and s to the computation process, it is required to use 2D trapezoidal rule. Similar to 1D trapezoidal rule, 2D will require the computation on the four corners, four boundary lines and the parts in the center. Note that before implement the 2D trapezoidal rule, it is necessary to square every element in the integral and use square root and them to keep everything positive.

Assuming: on

I =

1. Error calculation
2. Strong/Weak scaling

Scalability refers to the ability of a parallel system to demonstrate a proportionate increase in parallel speedup with the addition of more resources.

Strong scaling: for a fixed problem (i.e. in this homework the grid size is unchanged), though increasing the number of threads, the time used to run the problem should be decreased and become 1/ compared to serial computation.

Weak scaling: by increasing from 1 to 16 in this problem, keep the workload of every thread always changing, number of grids equal to , the total work time should be the same with more work completed.

Speedup: ratio of the serial runtime (with only one thread) to the time taken by the parallel algorithm (2-16 threads) to solve the same problem with “” cores.

Efficiency: ratio of speedup to the number of cores.

**Parallel part we added for optimization**

Basic Idea: the moment we import “parallel do” into the code, the moment computer will distribute the loops to each thread to complete. For example, if I want to distribute a number to every position of a 40 by 40 matrix and the thread number equal to 4, when I first loop over columns, the system will ask thread 0 to compete the job for column 1 to 10, and then I loop over rows, the system will ask the same thread to complete the rest of job, i.e., distribute numbers to column 1 to 10, row 1 to 40. Same thing for thread 1 with column 11 to 20, thread 2 with column 21 to 30, thread 3 with column 31 to 40 and all of them deal with row 1 to 40.

In this assignment, we mainly concentrate on the timing of computing function error, so we add the “omp\_get\_wtime()” as well as “parallel do” function in every compute error part. In order to add more parallel contents to the code structure, we also add “parallel do” and “parallel section” to other parts of the main code and subroutines.

**Results and Discussion**

1. The function for this homework:
2. The corresponding Mapping:



Figure1: The mapping of the equation

1. The convergence plot for the problem before parallel computing



Figure2: The plot of effective h vs maximum error as well as

It shows that the codes compiles with a good result and we can now begin to put parallel computing in it.

1. Parallel code result and serial code result comparison

When running on the same machine or same compiler the error values and Effective h values are exactly the same. I compared the two sets of output files with GNU "diff" and it had no differences. and when running from stampede the difference against my serial run was small. It is easy to show that from the following pictures, the convergence results we got from both parallel with OpenMP (thread = 16) and serial code (thread = 1) are the same in my local machine.

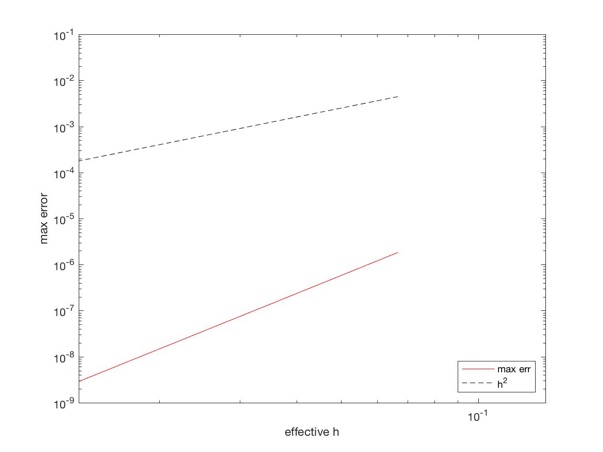
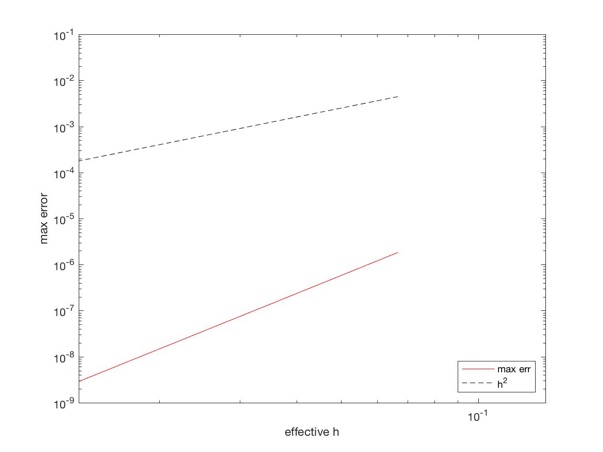


Figure3: Serial mode with thread = 1 (left) and Parallel mode with thread = 16 (right)

1. Adding timing function

In order to use super computer to calculate speedup and efficiency, we add the timing function to the codes where it will calculate the error for different mapping to get convergence. The figures below show us the speedup and efficiency, which proves the accuracy of our parallel codes.

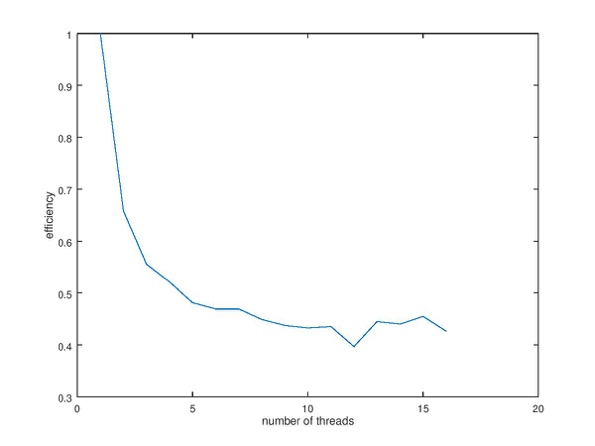
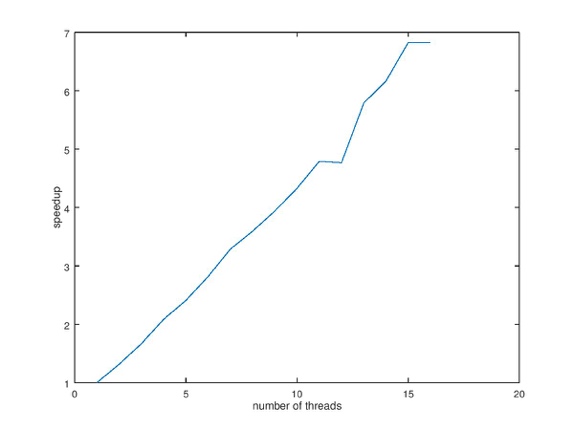


Figure4: Speedup (left) and Efficiency (right) for Error Computing

1. Strong scaling speedup and efficiency (with 1-16 cores)

For strong scaling, what we compare is the time used by the parallel Fortran code when applying the number of threads equal to 1, 2, ,,, ,16. For every experiment with different thread number, we keep the grid size always the same. Therefore, in the following figure, we present the speedup and efficiency results with small grid 20 by 20 and large grid 800 by 800.

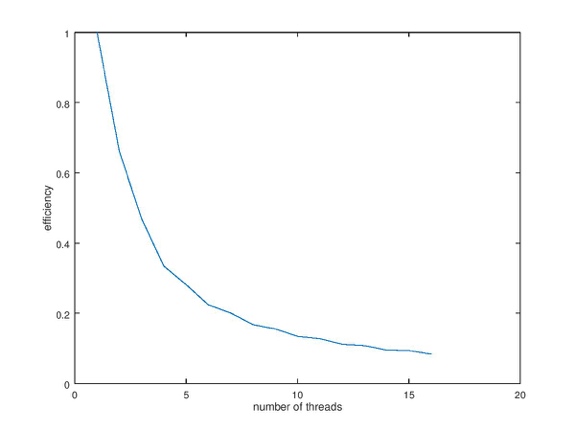
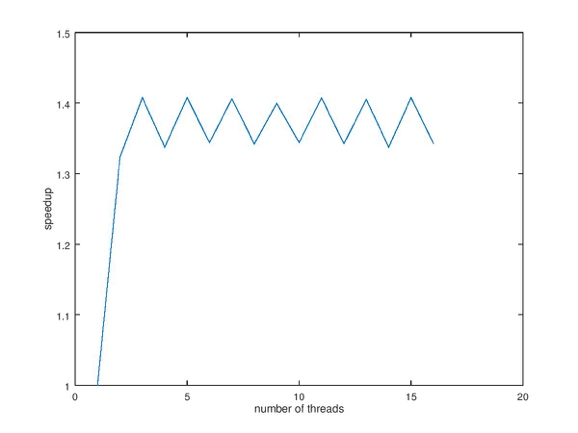


Figure5: Speedup (left) and Efficiency (right) for strong scaling when n = 800

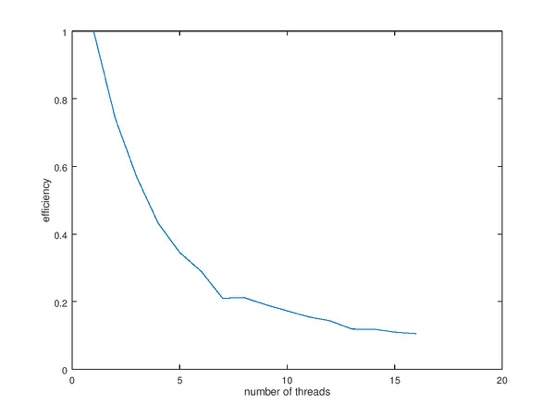
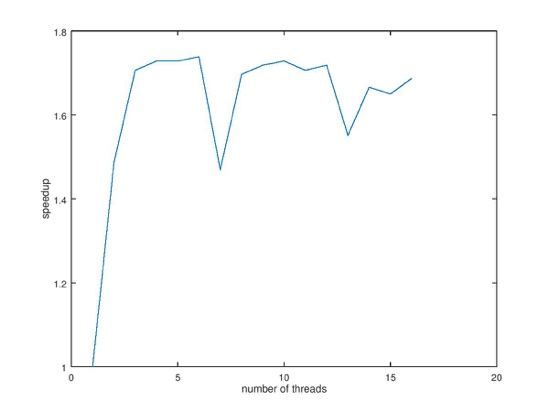


Figure6: Speedup (left) and Efficiency (right) for strong scaling when n = 20

Conclusion: Both Efficiency plots look normal since the efficiency will turn to decrease and then stay constant when there are many threads. However, the plots of Speedup look a little dynamic. We think the increasing trend should happen because in this strong scaling fixed situation, with more and more threads, the computation time should get smaller. But when there are many threads and only n = 20, maybe the unit core calculation time will turn to not change much. After all, no matter how many threads available on supercomputer, the computational time will always be larger than zero. When n = 800 it oscillates much less in magnitude.

1. Weak scaling speedup and efficiency (with 1-16 cores)

For weak scaling, what we compare is the time used by the parallel Fortran code when applying the number of threads equal to 1, 2, ,,, ,16.. For every experiment with different thread number, we keep the grid size growing. So grid when we use a total of threads and n = 200 is the grid size when we only have one thread. As a matter of fact, the number of loops every “parallel do” core could complete is always an integer, so we round the number to get its nearest integer.

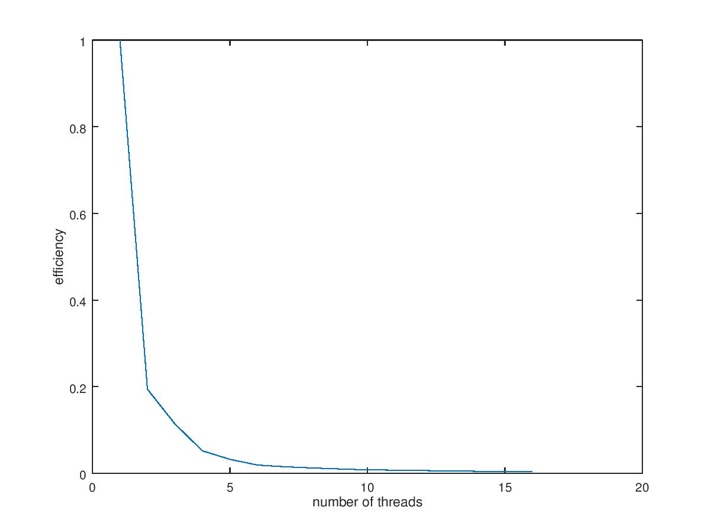
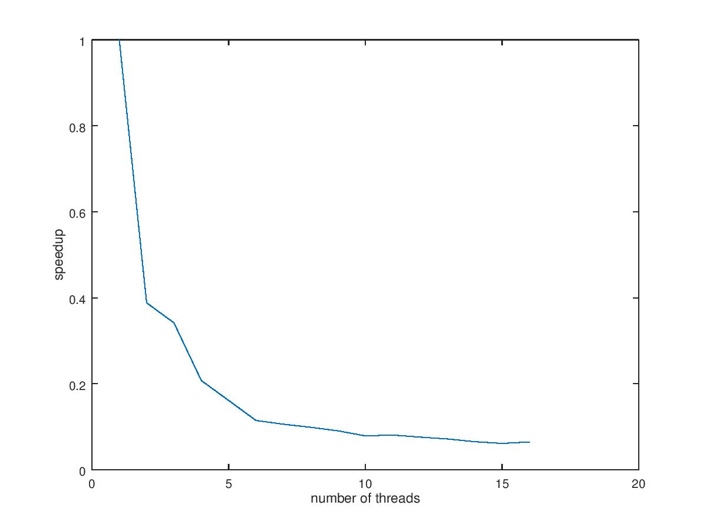


Figure7: Speedup (left) and Efficiency (right) for weak scaling when n = 200

Conclusion: Efficiency plot looks normal since the efficiency will turn to decrease and then stay constant when there are many threads even if the grid size is changing. The speedup gets decreasing all the way. We think that maybe even if there are more threads 1 to 16, the work every thread will do is also increasing with , so the total computational time could be still gets longer and longer if with more cores.

**Discussion**

We added lots of parallelization to the code. Mostly parallel do's but there were some good locations to put sections in as a way to break up the jobs for threads. I think the results have a pretty good correlation that the code is significantly speed up with an increase of threads.