Final Project for Math/CS 471

Xiaomeng and Jeff

12/17/2017

Abstract

This is the Final Project Report. First, the report will give a brief introduction of Math background. Second, the report will outline the procedures 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

The wave equation in the conservative form in two dimensions:

$$u_{tt} = (a(x,y)u_x)_x + \left(a(x,y)u_y\right)_y + f(x,y,t), \qquad (x,y) \in D = [-1,1]^2, t \in [0,2],$$

augmented with initial conditions

$$u(x,y,0) = f_1(x,y), \; u_t(x,y,0) = f_2(x,y), \qquad (x,y) \in D,$$

and Dirichlet boundary conditions

$$u(x,y,t) = g(x,y,t), \qquad (x,y) \in \partial D.$$

Here, $a(x,y) = 1 + \sin(x)\cos(y)$ is the given wave speed, and $u = u(x,y,t)$ is the wave solution to be computed. In general, a partial differential equation together with its corresponding initial and boundary conditions is called an initial-boundary value problem.

Part I. Serial computation

1. Use the method of manufactured solutions and manufacture an initial-boundary value problem for which the true solution is known:

   First, we manufacture a solution for the PDE problem, e.g. a trigonometric function:

   $u(x,y,t) = \sin(\omega t - k_x x)\sin(k_y y)$, (this is not the solution we want to get for project)

2. Then we will bring the method of lines and finite differencing on uniform grids:
   a. The PDE problem is discretized in the spatial domain D into N grid point to obtain a semi-discrete problem leaving the time continuous. This gives us a system of N ODEs in time.
   b. The semi-discrete problem (i.e. the system of ODEs) is then discretized in time to obtain a fully discrete solution in both space and time.
   c. We need to verify the convergence, i.e. we need to verify that the approximate solution converges to the true solution with the advertised theoretical rate. In the project, we can set $h = h_x = h_y$ and Δt = ch, and consider the maximum error at a fixed time t_k:

$$\varepsilon(h) = \max\left|u(x_i, y_j, t_k) - u_{i,j}^k\right|$$

   we want to make sure that $\varepsilon(h) \sim \vartheta(h^2)$. For this we will need to measure the error for various h values and verify that it is proportional to h^2.

3. Second, we adjust the initial-boundary value problem so that this is indeed its solution. We need to plug this manufactured solution into the problem and obtain the forcing $f(x,y,t)$, the initial data $f_1(x,y)$ and $f_2(x,y)$ as well as the boundary data $g(x,y,t)$. For the u shown as above, we would get:

$$\begin{aligned} f(x,y,t) &= \left(k_x^2 + k_y^2 - \omega^2\right)\sin(\omega t - k_x x)\sin(k_y y), \\ f_1(x,y) &= -\sin(k_x x)\sin\left(k_y y\right), \\ f_2(x,y) &= \omega\cos(k_x x)\sin\left(k_y y\right), \\ g(x,y,t) &= \sin(\omega t - k_x x)\sin(k_y y). \end{aligned}$$

4. Third, we will apply the method to this particular problem so that we have full control on the error. After we make sure that the error to this manufactured problem converges correctly, we can apply it to the actual problem in hand. In order to have a smooth transition from solving the manufactured problem to solving the actual problem, we should build a library of functions or subroutines so that we can easily modify the forcing and initial boundary data.
5. Fortran code to implement a second-order accurate finite difference method for solving manufactured problem by the method of lines. We would set $nx = ny$ and $hx = hy$.
6. Finally, we need to verify the code woks well by verifying the convergence rate of the maximum approximation error over the domain D at time 2. Plot (log-scale) the error versus a decreasing sequence of grid spacing and show the error decreases with the expected rate: 2.

Part II. Parallelization

1. Using MPI library to make the serial Fortran code parallel. A one-dimensional parallelization strategy should be employed by splitting the computational domain D into $nprocs$ vertical slabs. This is a one-dimensional domain decomposition with $px\_\max = npocs$ processors ranging from the processor number $px = 1$ to $px = px\_max$. The communication between processors will be similar to the 1D wave equation.
2. Apply the code to the same manufactured problem in part I. From checking the convergence of the maximum error in approximate solution at time t=2, we should be able to find: error decays with the expected rate 2, which could prove our code is working correctly.
3. 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 the case where 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 $1/N_p$ compared to serial computation.

Weak scaling: by increasing N_p from 1 to 8 in this problem, keep the workload of every thread always changing, number of grids equal to $\sqrt{N_p} \times n$, 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 "l" cores.

Efficiency: ratio of speedup to the number of cores.

Some backgrounds for MPI:
MPI_Init(int* argc, char*** argv)
Initialize the MPI execution environment.

MPI_Comm_size(MPI_Comm communicator, int* size)

Determines the size of the group associated with a communicator.

MPI_Comm_rank(MPI_Comm communicator, int* rank)
Determines the rank of the calling process in the communicator.

MPI_BARRIER(MPI_Comm communicator)
Blocks until all processes in the communicator have reached this routine.

MPI_REDUCE(void* send_data, void* recv_data, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm communicator)
Reduces values on all processes to a single value.

MPI_SEND(void* data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator)
Performs a blocking send.

MPI_RECV(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status)
Blocking receive for a message.

MPI_SENDRECV
Sends and receives a message.

MPI_WTIME
Returns an elapsed time on the calling processor.

MPI_FINALIZE
Terminates MPI execution environment.

Results and Discussion
Part I. Serial computation

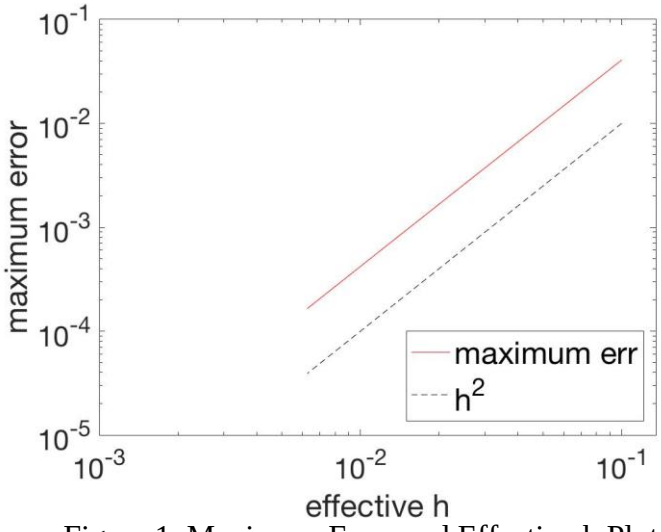

Figure 1. Maximum Error and Effective h Plot

It is easy to understand from Fig. 1 that the convergence is good and code works correctly at t = 2. We decide to calculate the slop of the curve to show the error decreases with the expected rate. Choose two points $(H_1, E_1) = (0.1, 0.0407)$ and $(H_2, E_2) = (0.0063, 0.0002)$. Therefore:

$$s = \frac{E_1 - E_2}{H_1 - H_2} = \frac{\log(0.0407 \div 0.0002)}{\log(0.1 \div 0.0063)} = 2$$

Basic idea: we translated the MATLAB code wave2D.m into Fortran, which is provided by the Professor and then output the error values and plot them in MATLAB. The results from MATLAB code and Fortran code are exactly the same, which proves the correctness of the Fortran code.

Part II. Parallelization

Basic idea: so according to the MPI library, we add as much MPI as we can. Generally, as soon as we add the MPI, we look at the procedure's outputs and see if it is the same as wave2D.m's results. Unfortunately, our MPI-parallelized Fortran code does not provide the same results as wave2D.m's when we get to Wave2D subroutine's later part, so we only stop here and begin the strong/weak parallelization on supercomputer. Therefore, the result for the Error and Convergence might be incorrect.

For strong scaling, we tried from 1 to 8 threads on supercomputer for all of the six grid number situations (100, 125, 150, 200, 250, 300) and use speedup as well as efficiency to analyze all of the six cases; For weak scaling, we use one thread with grid number = 100, two threads with grid number = $\sqrt{2} \times 100$, three threads with grid number = $\sqrt{3} \times 100$, four threads with grid number = $\sqrt{4} \times 100$,,,,,,,,seven threads with grid number = $\sqrt{7} \times 100$, which is in a total of seven cases. We also implement speedup as well as efficiency to analyze all of the seven time numbers.

a. Strong Scaling

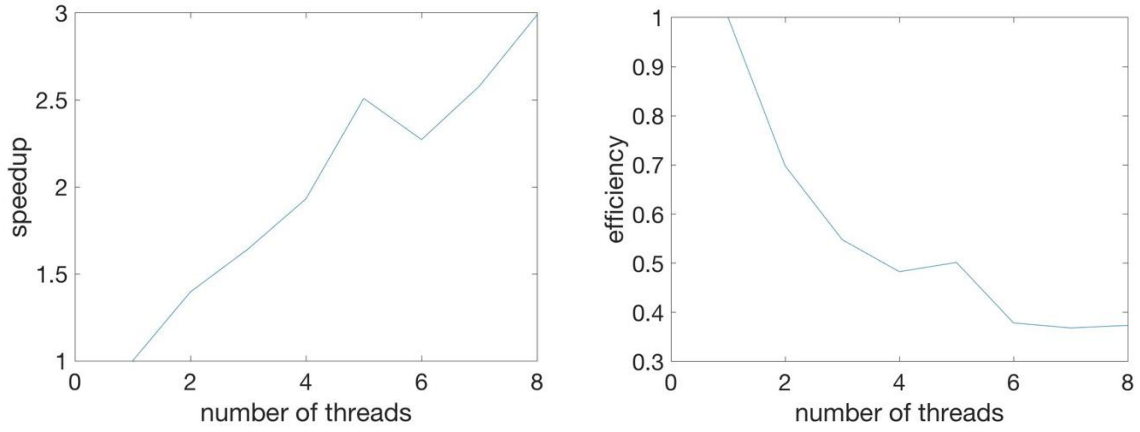

Figure 2. Strong scaling plot of Speedup (left) and Efficiency (right) with grid number = 100

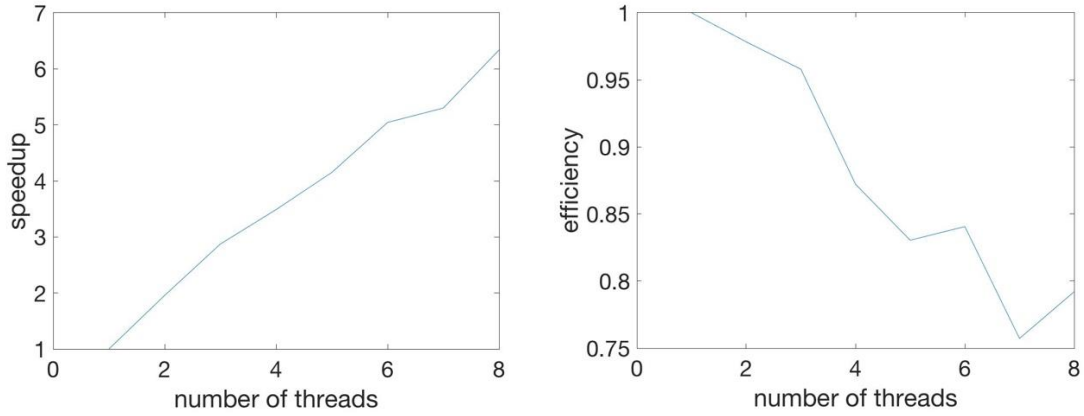

Figure 3. Strong scaling plot of Speedup (left) and Efficiency (right) with grid number = 125

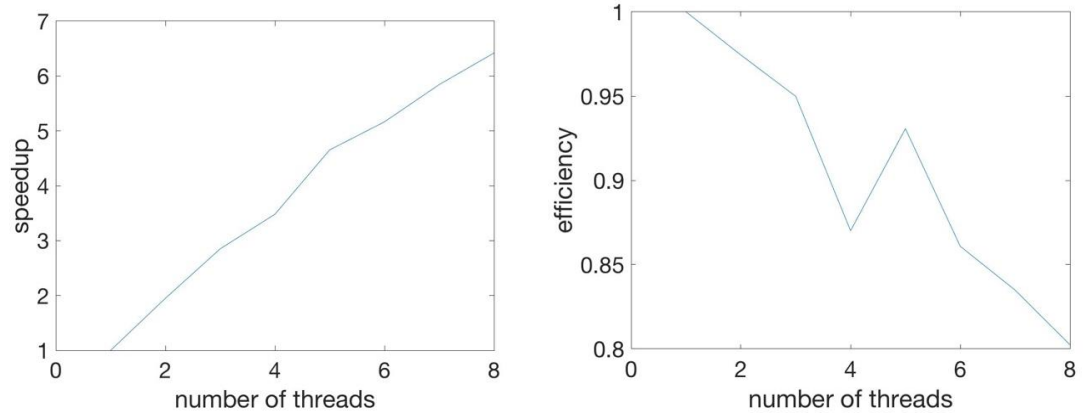

Figure 4. Strong scaling plot of Speedup (left) and Efficiency (right) with grid number = 150

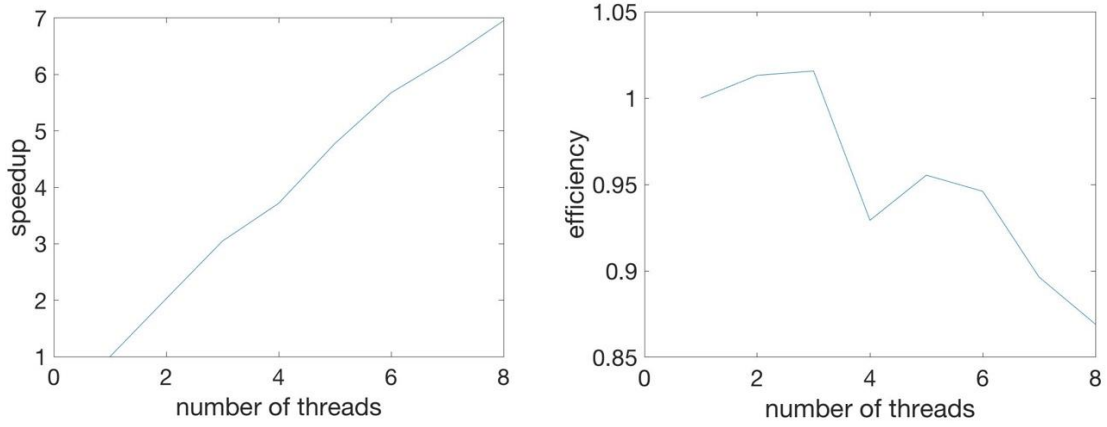

Figure 5. Strong scaling plot of Speedup (left) and Efficiency (right) with grid number = 200

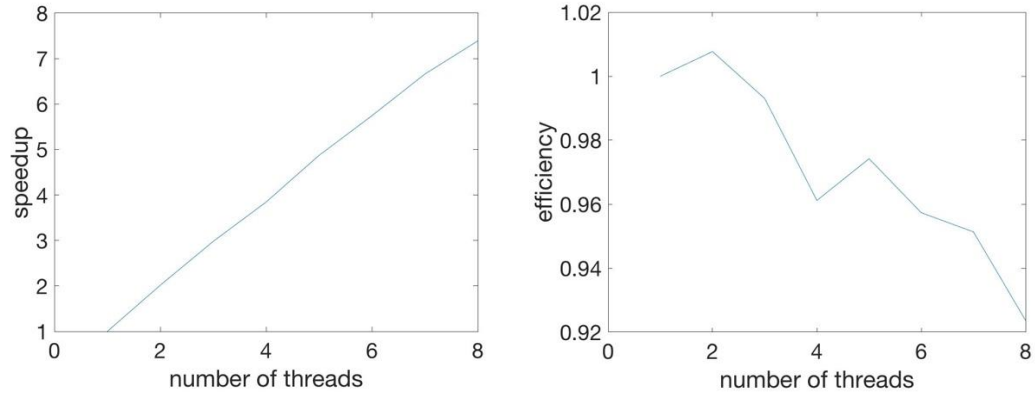

Figure 6. Strong scaling plot of Speedup (left) and Efficiency (right) with grid number = 250

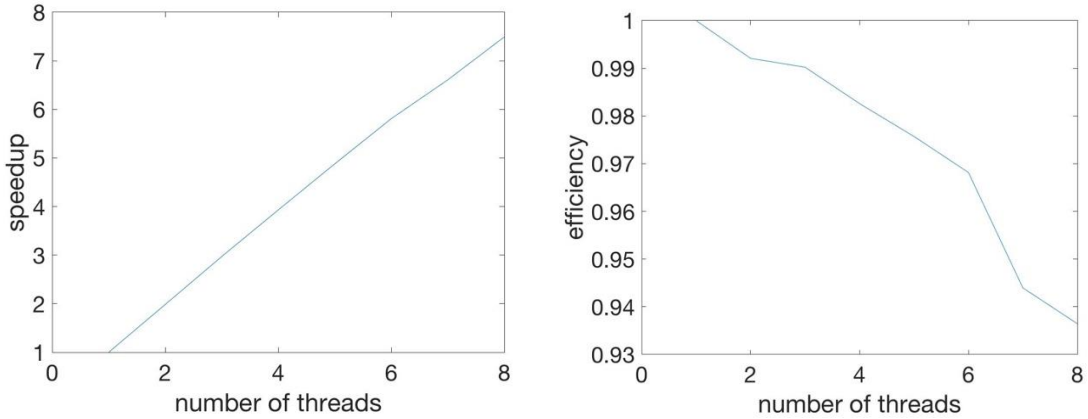

Figure 7. Strong scaling plot of Speedup (left) and Efficiency (right) with grid number = 300

Conclusion: It is easy to show that speedup increases with the number of threads used increasing. However, the efficiency decreases with the number of threads used increasing, which is normal for us. With grid number increasing from 100 to 300, the maximum speedup gets bigger and the minimum efficiency also gets bigger.

b. Weak Scaling

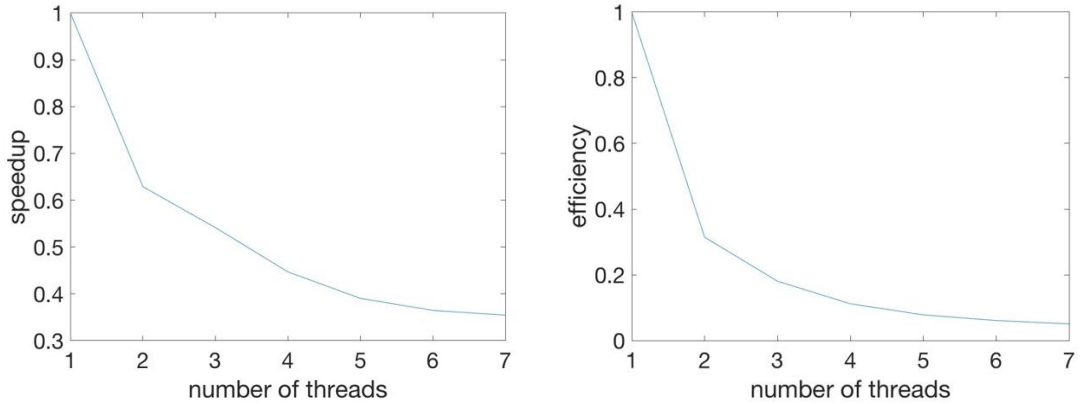

Figure 8. Weak scaling plot of Speedup (left) and Efficiency (right) with grid number and number of threads increasing.

Conclusion: we use one thread with grid number = 100, two threads with grid number = $\sqrt{2} \times 100$, three threads with grid number = $\sqrt{3} \times 100$, four threads with grid number = $\sqrt{4} \times 100$,,,,,,,,seven threads with grid number = $\sqrt{7} \times 100$. Observing from Fig 8, 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 8, the work every thread will do is also increasing with $\sqrt{l} \times n$, so the total computational time could be still getting longer and longer even if with more cores.

Discussion

We did not get nearly as far as we would have liked to. We only got to breaking the grid up in one direction. We could improve this by breaking the grid up into two dimensions. We could have also improved performance by utilizing Openmp as well.