Phys Enph 479/879 Assignment #4: Parallelization and MPI - Solving the 2D Heat Equation

Michael Jafs[1, *]

[1] *Department of Physics, Engineering Physics and Astronomy, Queen's University, Kingston, ON K7L 3N6, Canada*

(Dated: March 25, 2022)

We numerically solve the 2D diffusion equation in parallel using MPI4Py, a python package capable of parallel programming. We outline the typical approach of finite differences for the case of the 2D diffusion equation and show how this approach can be parallelized. We compare run times for our code while varying both the number of processes and nodes used. We find a direct correlation between the number of processes used on a single node and the speed-up time for a single process to run. Finally, we consider the solution visually by graphing several time steps and examine how the solution changes over a small period of time.

I. INTRODUCTION

There are a variety of differential equations that show up in the physical sciences which can be solved numerically. This is because they typically allow a user to discretize an array of space and time points that a computer can then handle one at a time. In this report, we consider the famous diffusion equation, which describes the rate of diffusion (in many cases heat transfer) through a medium or material given a set of initial conditions. When one defines a set of initial and boundary conditions, it is possible to model how these conditions will diffuse or propagate through the material for later times. An equation which provides knowledge of the temperature gradient, as a result of a temperature on a boundary not only provides interesting mathematical cases, but can provide essential information to those studying material sciences.

The diffusion equation's simple descritizability also allows for the construction of a parallel code used to perform the necessary computations. In parallel programming, the work required for computation is divided up into smaller tasks and distributed across multiple CPU's. The intent being an almost linear speed up in computation time. In this report, we investigate the times taken to numerically solve the diffusion equation with a code written for use with parallel programming. For the smaller and less memory/computationally intensive case examined here, running code in parallel is not at all essential, however, for much larger tasks this technique can prove extremely useful. As we will show, the total time for computations is reduced significantly when increasing the number of CPU's which share the work. The report is structured as follows. In Section II we provide the theory and necessary background for the numerical approach used to solve the heat equation and explain how our code has been parallelized. Section III gives time comparisons when altering the number of CPU's used and provides a graphical representation of the solution. Finally, Section IV provides concluding remarks.

II. THEORY

The equation we consider is the two-dimensional (2D) diffusion equation (otherwise referred to as the "heat equation"):

$$\frac{\partial U}{\partial t} = D\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}\right), \tag{1}$$

where D is the diffusion coefficient [1]. Eq. 1 describes a function U which changes in both space and time. It is similar in form to the wave equation, however typical solutions to the heat equation usually result in rapid changes at the beginning of some time interval t, while at later times, the solution becomes smooth and U is often unrecognizable [2]. Since Eq. 1 is a function of both space and time, a useful numerical approach to finding a solution is to descretize the spacial parameters and work with grid points in which we perform calculations on. We assume $U(x, y, t)$ is a discrete function $u_{i,j}^{(n)}$, in which $x = idx$, $y = jdy$, and $t = ndt$ [1]. This descretization step allows one to write out the derivatives in Eq. 1 as finite difference approximations, thus transforming a differential equation into an algebraic one. A simple approach uses a forward difference scheme in time and a central difference in both space dimensions:

$$\frac{u_{i,j}^{(n+1)} - u_{i,j}^{(n)}}{\Delta t} = D\left(\frac{u_{i+1,j}^{(n)} - 2u_{i,j}^{(n)} + u_{i-1,j}^{(n)}}{(\Delta x)^2} + \frac{u_{i,j+1}^{(n)} - 2u_{i,j}^{(n)} + u_{i,j-1}^{(n)}}{(\Delta x)^2}\right). \tag{2}$$

Having made this shift to a descritized equation, it is straightforward to show that the system at a time step $n+1$ can be calculated from the time step at n from:

$$u_{i,j}^{(n+1)} = u_{i,j}^{n} + D\Delta t\left(\frac{u_{i+1,j}^{(n)} - 2u_{i,j}^{(n)} + u_{i-1,j}^{(n)}}{(\Delta x)^2} + \frac{u_{i,j+1}^{(n)} - 2u_{i,j}^{(n)} + u_{i,j-1}^{(n)}}{(\Delta x)^2}\right). \tag{3}$$

* 17msj2@queensu.ca

Eq. 3 can then be solved using a vectorized function which takes an initial array and steps through time, forming the solution in space (on the grid). Given a set of boundary and initial space conditions, the discrete solution $u_{i,j}^{(n)}$ can be solved for. It is lastly necessary to note that the separation in values for time must be sufficiently small to ensure the process does not become unstable. This condition appears as [1]:

$$\Delta t = \frac{1}{2D} \frac{(\Delta x \Delta y)^2}{(\Delta x)^2 + (\Delta y)^2} \tag{4}$$

A. Parallelizing the Heat Equation

Solving the diffusion equation in 2D, for a moderate grid size and small number of time steps is not particularly computationally intensive. However, the jump from two, to three dimensions and/or investigating the solution for a large number of time steps or grid points quickly requires much more time and memory, when performing calculations. For example, a grid in 2D with side lengths of 100 points has a total of 10,000 grid points, but in 3D, this becomes one million grid points. For a large number of time steps, this requires even more memory and computational power. One solution to speeding up this process is to not run the calculation routine in serial but in parallel. This implies using a package capable of parallel programming, which allows the user to divide up tasks in the computations, so that a network of computers can solve parts of the routine separately. The goal being a linear (proportional to the number of processes) speed up in computation time. The ability to discretize the heat equation and write the solution as a grid of spacial points, creates an effective arena for writing and testing the speed up potential, of a parallel code.

In order to perform calculations in parallel, one must first determine how they will divide up the tasks comprising the calculations in the routine. In the case of the 2D heat equation, a simple way is to divide up the spacial grid vertically into smaller grids so that each separate process does work on a seperate smaller grid. This is the method our results in the following section have exploited. We used an MPI interface known as MPI4Py [3] (an MPI version for python), to assist with parallel computations on the Frontenac cluster at the Center for Advanced Computing (CAC). Therefore, we solved the heat equation in a python environment, with the assistance of multiple processes at the CAC to speed up the processing time.

III. RESULTS

Our simulations involved solving the heat equation in 2D with a parallel code. By dividing up the original grid vertically into smaller grids, we allowed the specified number of processes (specified when running our job on the cluster), to each work on a separate smaller grid. We designed our routine to use a "buffer" like system to deal with the communication of grids with one another across processes. Specifically, each grid was given an extra array above and below, which stored the information about the last array in the grid above and the first array in the grid below respectively.

We considered an example case of a "plate" with dimensions of width = height = 20.48 mm. Next we imposed initial conditions such that $U_0 = 2000\cos^4(4r)$ if $r < 5.12$ and $U_0 = 300$ K if $r \geq 5.12$, where r acts like the radius and is measured from the start of the domain. The form of these initial conditions can be seen in the upper left subplot in Figure 1. We further imposed boundary conditions such that $U = 300$ K at the edges of the domain and used a diffusion coefficient of $D = 4.2$ mm^2/2.

A. Time Comparisons

Solving Eq. 1 numerically for a given set of initial conditions can provide unique information about the way in which heat propagates through a specific material, however since we are running our code in parallel, we would first like to investigate the speed up times that are possible as a result of using MPI. For a grid size of 1024x1024 and 1000 time steps, we bench marked our code for the cases of 1, 2, 4, 8, and 16 processes on a single node, and then for 2 nodes, with 8 processes per node. The results for these scaling experiments can be found in Table I.

The speed up of the times as measured within the routine across a single process ("time to solve") are decreasing, which is to be expected. As the number of processes increases, we are dividing up the total work amongst a larger number of cores. Therefore, the work for each core should decrease. The fact that the times measured across process 0 are asymptotically decreasing indicates that for a larger number of processes, we cannot achieve a significant increase in the speed up potential found using MPI. Due to limitations in the reservations used when running the jobs, our executions were limited to the case of 16 process's on a single node, however the trend in Table I is clearly logarithmic, or at least slowly decreasing for a large number of processes. Since the communication between nodes in a cluster is slower than the communication between cores in the same node, we expect a slight increase in the time for an individual process to complete it's own task. We found an increase of ∼0.1 second for process 0 to complete the task when running one extra node for 9 processes. The values in Table I are averaged over three trials, so it is likely that this slow-down is a result of increasing the number of nodes and not random fluctuations in time.

The system + user time is the total CPU time spent during the entire computation. In other words, this is the cumulative time for each CPU, since we are using multiple CPU's when running parallel code. We have found that when increasing the number of processes for a single node, the system + user time did not vary widely for 1, 2, and 4 processes, however, we saw a noticeable increase in this number when using 8 and 16 processes. Since this time rep-

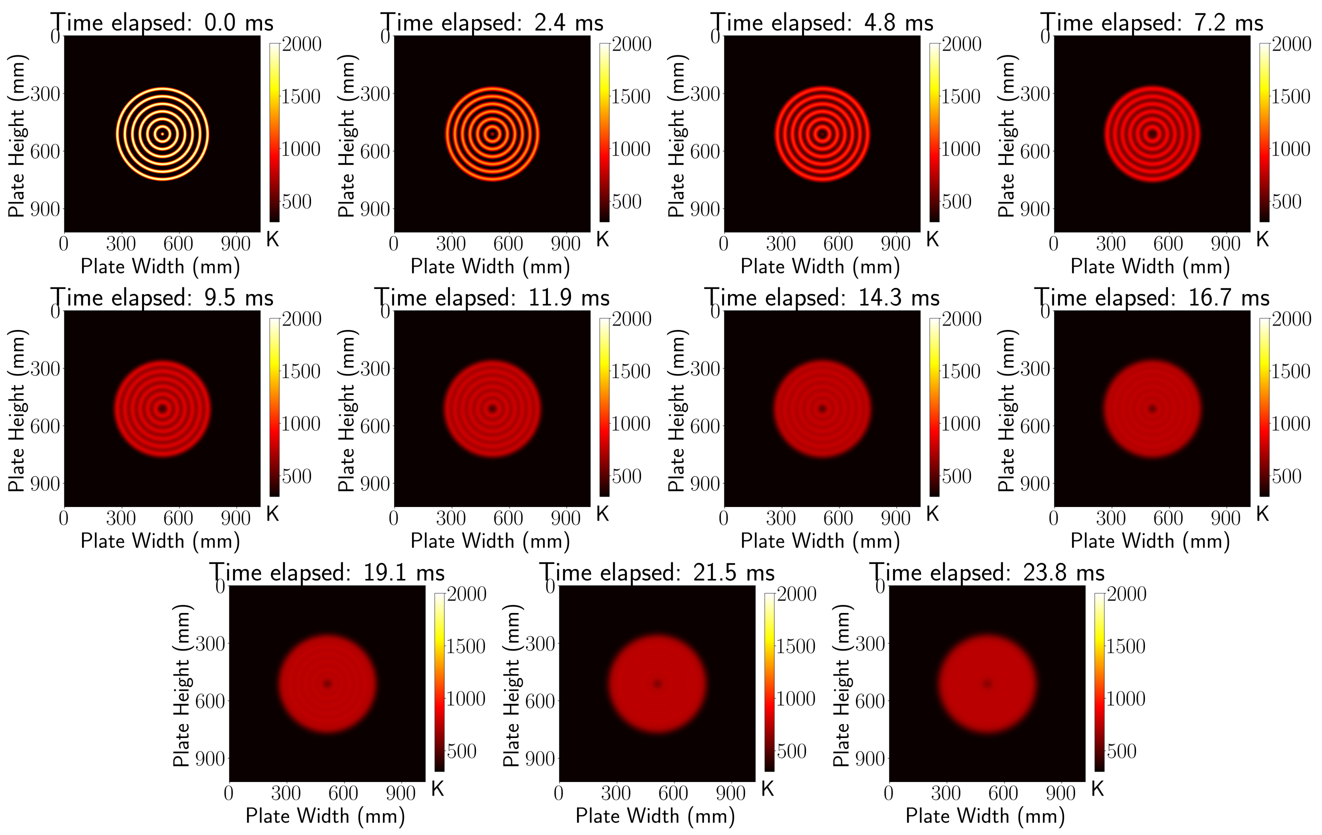

Figure 1. Solution $u_{i,j}^{(n)}$ as a function of time for 1000 time steps. The plate is symmetric with side length 20.48 mm. The grid size is 1024x1024. The diffusion coefficient is taken to be $D = 4.2$ mm^2/2.

resents the sum of the times for the total computations, we found when dividing the system + user time by the number of processes used, there was a noticeable decrease in computation time for an increase in processes. Additionally, by dividing the system + user time by the number of processes used, we found the times to be much closer to the "time to solve" times. Contradictory to what we expected, we found that the (system + user)/(processes) for 2 nodes gives a time of 3.563 seconds, while for 1 node, the same computation gives a time of 3.804 seconds. It is possible that if we continued our observations to include 4 + nodes that we would find an increase in (system + user)/(processes) time, though this is not what we have found for the case of only 2 nodes as seen at the present time.

Nodes	Processes	System + User time (s)	Time to solve (s)
1	1	25.243	22.384
1	2	18.682	7.243
1	4	23.112	4.422
1	8	30.433	2.517
1	16	41.416	1.345
2	8	28.506	2.609

Table I. Run times corresponding to the number of processes and nodes used when running the job. System and user are measured using the linux timer with the "time" call when running the job, whereas the "time to run" is calculated across a single process (process 0) using the timeit module within python. Times are taken as a mean average of three separate trials each.

B. Visualizing the Solution

It is worth taking a look at the solution to the heat equation for the initial conditions outlined at the beginning of this section, to make clear what the computations we were timing in the last section represent. After running our computations in parallel (though this need not be the case for plotting), we had our routine save snap shots at every 100 time steps. This amounted to ms time stamps that can be seen in the title of each subplot in Figure 1. We ran our computations for 1000 time steps of dt which gave us a total run time of 23.8 ms. The size of dt was limited by imposing the Courant Friedrichs Lewy condition (Eq. 4), so for a moderate time step size of 1000, we examined a short time frame for fluctuation of the solution $u_{i,j}^{(n)}$. The behaviour demonstrated by solving the heat equation is obvious even under these conditions. Figure 1 shows the hard ripples present in the initial conditions at time $t =$ 0ms washing out over time. In fact, at a time of 23.8 ms (bottom right subplot in Figure 1), the initial condition is barely recognizable. Additionally, the change in color from yellow, to a dark red indicates the change in temperature of the plate over time. Indeed, the color of the circles at a time of 0 ms as compared to the color bar show that the initial condition of U_0 having a temperature of 2000 is in fact satisfied. In only 20 ms, the plate has cooled to roughly 1000 K or slightly less. Therefore, our solution represents the behaviour we expect when solving the heat equation.

IV. CONCLUSIONS

We have provided the mathematical discourse to understand a simple approach to solving the heat equation numerically. By using a finite difference approximation, we were able to discretize the heat equation in both space and time so that our computations were done over a grid with user-defined dimensions. We provided the motivation for a code written for use with a parallel programming package such as MPI and showed how one might structure a spatial 2D array, so that parallel computations can be executed.

Using a grid size of 1024x1024 and realistic parameters, we solved the heat equation in parallel using the package MPI4Py. This resulted in a noticeable speed-up of computation time when the number of processes (number of CPU's used) increased. We also found that increasing the number of nodes slowed the computational time across a single process. Due to reservation limitations, we were limited to using a maximum of 16 processes and 2 nodes. However, our data still shows the trends we expect when it comes to the speed-up/slow-down effects present when running parallel code over multiple processes and nodes.

We finally plotted the solution in space for 1000 time steps to visualize our circular initial condition and to verify our boundary conditions. We also examined how the temperature gradient changed over time and showed that over a small time span of $\sim$ 20ms, a large decrease in temperature is possible, where at late times the solution is slowly varying.

References

[1] "The two-dimensional diffusion equation," .
[2] "The 1d diffusion equation¶," .
[3] "Tutorial," .