ME 766- High Performance Scientific Computing In the Moment of Heat!

Akshit Shrivastava
Dept. of Metallurgical Engineering
IIT Bombay
akshit@iitb.ac.in

Samyak Shah
Dept. of Electrical Engineering
IIT Bombay
samyakshah@iitb.ac.in

Souvik Pal
Dept. of Metallurgical Engineering
IIT Bombay
18d110011@iitb.ac.in

Rohan Pipersenia
Dept. of Metallurgical Engineering
IIT Bombay
rohanpipersenia@iitb.ac.in

***Abstract*—The theorems concerning the spread of heat find wide applications. They are instrumental when we wish to foresee and regulate temperature with precision, like in the case of greenhouses or cold storage systems. Fourier himself was interested in studying the flow of heat to keep his wine cellar cool. In this project we have solved the problem- 'In the Moment of Heat' from the book SIAM 100-digit challenge- A study in high-accuracy numerical computing by Bornemann, Laurie, Wagon and Waldvogel. The problem asks us to find out the time when the temperature of the centre of a square plate will reach a particular value when the temperature of one of its edges is increased. We plan to solve the problem by implementing the Crank-Nicholson, central difference finite difference scheme over a 2-dimensional square on Python using the Python OpenCL library. We have optimised and parallelised the algorithms wherever possible.**
The problem is a perfect mix of theoretical rigour and an opportunity to implement and visualise via code. We would endeavour to find the exact answer deep after the decimal point in the quickest time.

I. Problem Statement

A square plate $[-1, 1] \times [-1, 1]$ is at temperature $u = 0$. At time $t = 0$ the temperature is increased to $u = 5$ along one of the four sides while being held at $u = 0$ along the other three sides, and heat then flows into the plate according to $u_t = \Delta u$. When does the temperature reach $u = 1$ at the center of the plate?
To twelve digits the wanted solution is t = 0.424011387033.
Note: The Problem statement is one of the ten problems from the book *The SIAM 100 Digit Challenge* [1].

II. Introduction

The project discusses the numerical methods used to solve the heat equation for a 2D Heat Plate. The heat equation is a type of partial differential equation (PDE) which can be solved using numerical methods. To solve the equation, the *Galerkin method* of weighted residuals was applied on the finite elements in the spatial domain and the Implicit Euler method was used to perform the time marching. The objective is to find the time required to reach a particular temperature at the center of the plate.

We also discuss the relevant aspects of parallelisation. The OpenCL kernel methods approach to parallelise certain portions of calculation steps in the Conjugate Gradient method (used to solve the system of linear equations at each time-step) have been discussed in the report. To achieve further reduction in the execution time we will use Numba library's JIT compiler to accelerate the function that we define by just using a simple function decorator!

We will measure the success of our endeavour based on the following metrics:

1) The speedup of the program against the serial code for the same accuracy in the final result.
2) In case the time is fixed, we wish our program to generate the most accurate results (most correct digits after the decimal).

III. Mathematics

First we consider the theoretical solution to the problem. This problem cries out for a Fourier series solution.

We define the various parameters as follows. We take the side $\{x = L, L = 1\}$ as the one maintained at $d = 5$. We denote the square by Ω, the side $x = L$ by Γ_d and the other three sides by Γ_0.

The first step is to get rid of the inhomogenous boundary conditions at Γ_d. Solve for the function $u_0(x, y)$.

$$\begin{cases} -\Delta u_0 = 0 & \text{in } \Omega \\ u_0 = d & \text{on } \Gamma_d \\ u_0 = 0 & \text{on } \Gamma_0 \end{cases}$$

This can be done using Fourier series. Use $u_0(x, y) = \sum_{n=1}^{\infty} u_n(x) sinn\pi \frac{y+L}{2L}$. Here,

$$u_n = \mu_n \frac{sinhn\pi \frac{x+L}{L}}{sinhn\pi},\ \mu_n = \begin{cases} \frac{4d}{n\pi} & \text{if } n \text{ is odd} \\ 0 & \text{if } n \text{ is even} \end{cases}$$

Hence,

$$u_0(0,0) =$$

$$\sum_{n=0}^{\infty}(-1)^n \frac{4d}{(2n+1)\pi} \frac{sinh(2n+1)\frac{\pi}{2}}{sinh(2n+1)\pi}$$

Now let $u = v + u_0$, here v solves the problem:

$$\begin{cases} \frac{\partial v}{\partial t} - \Delta v = 0 & \text{in } \Omega \\ v = 0 & \text{on } \partial\Omega \\ v(x,y,0) = -u_0(x,y) & \text{in } \Omega \end{cases}$$

Solving for v and putting everything together gives the expression for temperature at the center of the plate as a function of time:

$$u(0,0,t) = \frac{20}{\pi}\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1}\frac{sinh(2n+1)\frac{\pi}{2}}{sinh(2n+1)\pi} - \frac{40}{\pi^2}\sum_{q=0}^{\infty}\sum_{p=0}^{\infty}\frac{(-1)^{p+q}(2p+1)e^{-1/4\left((2p+1)^2+(2q+1)^2\right)\pi^2 t}}{(2q+1)\left((2p+1)^2+(2q+1)^2\right)}$$

This theoretical expression needs to be converted to a numerical approximation. Let u_h is the numerical approximation of $u(0,0,t)$, τ is the small time step, h is the small spatial step. Then:

$$u_h(t,x,y) = u_h(t-\tau,x,y) - \frac{\tau}{2h^2}[\ 4u_h(t-\tau,x,y) - u_h(t-\tau,x-h,y) - u_h(t-\tau,x+h,y) - u_h(t-\tau,x,y-h) - u_h(t-\tau,x,y+h) + 4u_h(t,x,y) - u_h(t,x-h,y) - u_h(t,x+h,y) - u_h(t,x,y-h) - u_h(t,x,y+h)]$$

Using $\alpha = \frac{\tau}{2h^2}$ and rearranging the expression,

$$(1+4\alpha)u_h(t,x,y) - \alpha[\ u_h(t,x-h,y) + u_h(t,x+h,y) + u_h(t,x,y-h)u_h(t,x,y+h)] = (1-4\alpha)u_h(t-\tau,x,y) + \alpha[\ u_h(t-\tau,x-h,y) + u_h(t-\tau,x+h,y) + u_h(t-\tau,x,y-h) + u_h(t-\tau,x,y+h)]$$

Flattening the lattice row wise and applying boundary conditions b, we get the following linear system:

$$A_{t+1}u_{h,n+1} = A_t u_{h,n} + 2\alpha b$$

IV. Algorithm

The general Approach to the problem is using the following algorithm which lists the major codeblocks that were adopted:

- Construct the Linear Operators A_t and A_{t+1}.
- Construct Preconditioner P to approximate the inverse of A_{t+1}.
- Secant method is used to approach to the solution.

Each iteration of the secant methods calculates the temperature at the center of the plate reached after an elapsed time t. This is done by applying the finite difference method above successively until the targeted time has been reached.

V. Parallelisation

Throughout this project we have tried to parallelise and accelerate the code and functions as much as possible. Some of the techniques we used hace been covered during the course, the others were self learnt by reading the relevant documentations and journals.

A. Using PyOpenCL

The OpenCL library was used to develop parallel code for two problems, $Y = aX + Y$ (DAXPY). This could be used in the conjugate gradient method where there are few calculations which are performed at every time-step to solve the system of PDE. The parallelisation is beneficial in this case since these can be very large vectors with hundreds of thousands of points. This is specially true for DAXPY since the entire calculation can be parallelised in one step. PyOpenCL is the extension of OpenCL developed for the Python programming language.

- PyOpenCL lets one access GPUs and other massively parallel compute devices from Python. It tries to offer computing goodness in the spirit of its sister project PyCUDA.
- PyOpenCL offers automatic error checking. All errors are automatically translated into Python exceptions for easy comprehension.
- It follows the RAII (Resource Acquisition is Initialization) idiom like C++.

B. Using Numba

We used Numba library and related functions in our project code.

- Numba is an open source JIT (Just-in-time) compiler that translates a subset of Python and NumPy code into fast machine code.
- Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN.
- One doesn't need to replace the Python interpreter, run a separate compilation step, or even have a C/C++ compiler installed. He just has to apply one of the Numba decorators to his Python function, and Numba does the rest.

VI. Experiments and Results

1) The theoretical approach to solve the problem using standard fourier transforms gives a tentative idea of the solution which comes to be t = [0.42, 0.43] seconds, for the center of plate to reach 1 unit temperature. The following plot in Figure 1 depicts the results clearly. It is observed that after a while the temperature value almost remains constant and the change is only seen in the deeper digits of decimal. Appendix A contains the table of the exact values used for the convergence plot.
2) In the next experiment we kept the time step to be 0.001 and the spatial step to be 0.01. The main objective of this experiment was to find the the error that is generated by the code to find the temperature if the exact time is given. The results of this experiment are displayed in Figure 2. It also offers a great visualisation of the temperature flow by means of a heat map.

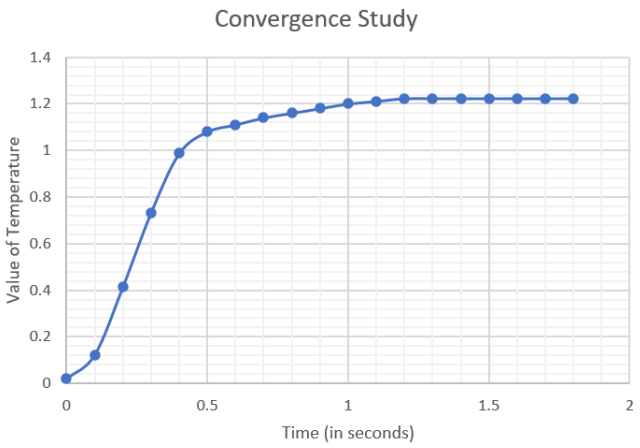

Fig. 1. Evolution of the temperature at the center of the plate

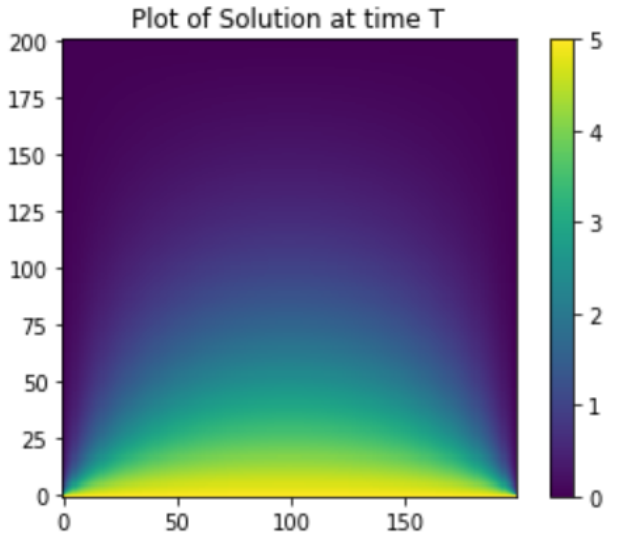

Fig. 2. At time t=0.424011... seconds, the center temperature is 1.0063884459264785 This signifies an error of 6.3884E-03 from the known solution.

3) In the next experiment we did hyperparameter tuning to find the right set of parameters for calculating the Preconditioning matrix which in turn becomes a reference in finding the inverse using conjugate gradient method.The preconditioner for this solver uses the Sparse Incomplete LU decomposition feature of the SciPy library. We found good values for the drop tolerance and fill factor parameters for the generator. The analysis indicated that a drop tolerance of 1e-5 and a fill factor of 18 will provide good convergence results. After hyperparameter tuning we re-run the main function with the time step to be 5e-4 and the spatial step to be 8e-3. We obtained a better result than the previous one where the error was close to 0.63% whereas after hyperparameter tuning the error reduced to 0.21%.
4) In the last experiment of the project, we compare the run time of the serial code and our code which uses effective parallelisation. We note the time at which the code gets a certain number of correct digits behind the decimal point. We will then measure the speedup for each case. We observe that the speedup value increases for a higher number of correct digits behind the decimal point. The speedup value lies between 1.5 to 2.5 for our analysis.

Decimal Places	Serial Code	Parallel Code	Speedup
6	48.64	32.21	1.51
7	77.21	38.88	1.99
8	105.56	50.03	2.11
9	143.1	66.54	2.15
10	207.7	89.1	2.33

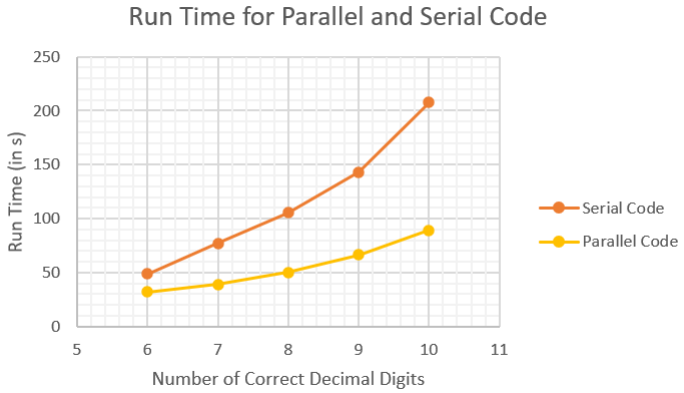

Fig. 3. Run Time for Serial and Parallel code

If the time taken by the serial code to run is t_1 seconds and the time taken by the parallel code to run is t_2 seconds, then:

$$Speedup = \frac{t_1}{t_2}$$

We can thus conclude that our parallel code is a very good option, much better than the serial code if we want to get a very exact answer in a short time.

ACKNOWLEDGMENT

We would like to thank Prof. Shivasubramanian Gopalakrishnan for the references, feedback regarding the experiments and guiding us through the exciting field of High Performance Scientific Computing. His great teaching style made complex concepts crystal clear, even during the tough times of COVID-19.
Last but not the least we would like to thank all the open-source contributors to tools like PyOpenCL and Numba which make high performance computing accessible to programmers of all proficiency levels. They have an important contribution in making research accessible and available to everyone around the globe. Most of our coding was done on Google Colab which offers free GPU access, version control and collaboration features to its users.

APPENDIX A

Below is the table used to make the convergence plot, i.e. 1. Since the plot is only up to 1.7 seconds we have truncated the temperature value to three digits after the decimal, to maintain readability.

Time (in seconds)	Value of Temperature
0	0.02
0.1	0.123
0.2	0.414
0.3	0.732
0.4	0.987
0.5	1.08
0.6	1.11
0.7	1.14
0.8	1.16
0.9	1.18
1	1.2
1.1	1.211
1.2	1.223
1.3	1.223
1.4	1.223
1.5	1.223
1.6	1.223
1.7	1.223
1.8	1.223

APPENDIX B

We were advised by Prof. Shivasubramanian Gopalakrishnan to include details about the data structures we used to capture the high precision values. Initially we used Snowlakes double precision numeric data type with the Python connector to Snowflake. Later, after reading a few Git issues we realized that the given combination generates incorrect results (known bug, yet to be resolved).

We then chose to use the float datatype of Python which uses double precision to store its values. Double precision numbers have 53 bits (16 digits) of precision.

Since the competition at Oxford university which was the inspiration behind the book, required a solution correct up to 10 decimal digits, we have succeeded in our mission.

Had we been born in a different generation and went to Oxford, we would have bagged the prize!

REFERENCES

[1] Bornemann F., Laurie D., Wagon S., Waldvogel J., The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing, February 2003

[2] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fasih, PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation, Parallel Computing, Volume 38, Issue 3, March 2012, Pages 157-174.

[3] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: a LLVM-based Python JIT compiler. In Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15). Association for Computing Machinery, New York, NY, USA, Article 7, 1–6. DOI:https://doi.org/10.1145/2833157.2833162

[4] J. E. Stone, D. Gohara and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," in Computing in Science Engineering, vol. 12, no. 3, pp. 66-73, May-June 2010, doi: 10.1109/MCSE.2010.69.