Parallelization of 2D Convection-Diffusion Equation Solver

Wenliang Sun
wzs51@psu.edu
The Pennsylvania State University

Problem Statement

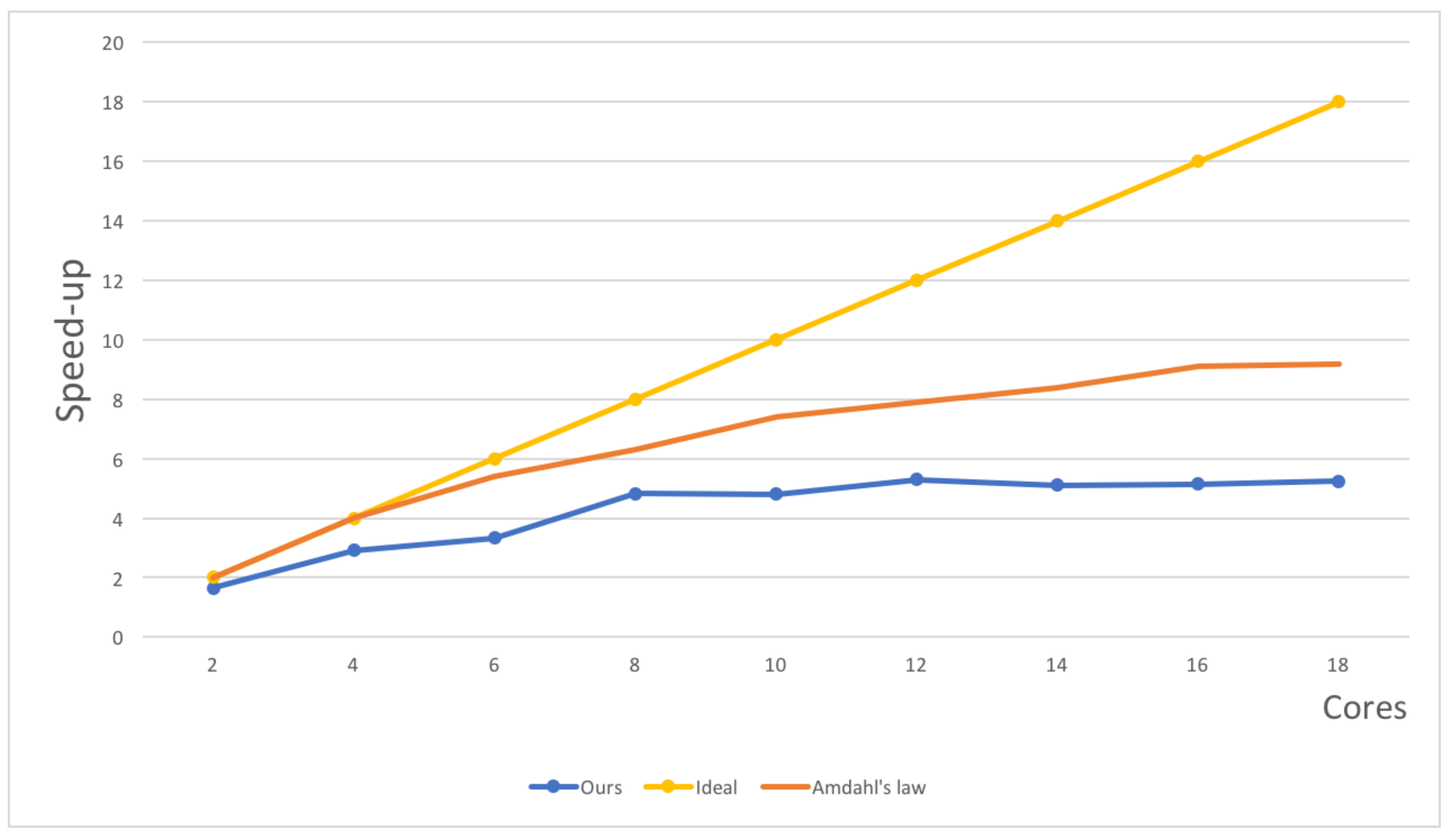
In recent years, due to the rapid development of industry, the industrial heavy metal emissions have increased, and heavy metals containing polluting gases, polluting liquids and industrial residues have had a huge impact on the environment. The root of the problem lies in finding the source of pollution, and the treatment of the source of pollution can solve the fundamental problem. The transport of heavy metal pollutants in the soil can be attributed to the migration of soil solute. The solution to the transport model of soil solute is to solve the convection-diffusion equation.
In our project, we are going to solve a 2D convection-diffusion problem. As a very important branch of partial differential equations, convection-diffusion equations have been widely used in many fields, such as fluid mechanics, gas dynamics, etc. Because convective diffusion equations are difficult to obtain analytical solutions through analytical methods, so through various numerical methods to solve the convection diffusion equation plays an important role in numerical analysis. Our project solves the steady 2D convection-diffusion
problem:

$$\frac{\partial \rho u_i \phi}{\partial x_i} = \frac{\partial (\tau \frac{\partial \phi}{\partial x_i})}{\partial x_i} + q_\phi .$$

For the 2D convection diffusion equation, we assume: x∈[0,1] and y∈[0,1] and $\rho = 100$, $\tau = 0.1$, $q_\varphi = 0$, $u_x = x$, and $u_y = -y$. The boundary conditions are $\varphi=0$ on $y=1$, $\varphi=\varphi(y)=1-y$ on $x=0$, $\partial\varphi/\partial x = 0$ on $x= 1$, and $\partial\varphi/\partial y$ on $y=0$. The problem could beconstructed as a linear equation ultimately, with the form of Ax = b.

Strong Scaling

In this part, we fix the problem size, 500 * 500 matrix, and then add the processors. We record the time consumption for each experiment and then calculate the speed-up.



In this plot, the point start from 2 cores. Starting from N = 2 makes it easy to get the expected computation time from Amdahls law. We could also see that using 10 cores is not a good choice for the parallel code profiling from this plot. Because it gives a large overhead of communication time between processes. However, this does not influence the validity of those discussions. We can also see that the speed grows with the cores.

Serial: Direct and Iterative Solver

Our direct and iterative solvers are used for serial version of code, and only iterative part has been parallelized in our project. We use the LU decomposition as direct solver. So we can solve the Ax = b equation. And the iterative solver leverages the Jacobi iterative method. Because our system matrix A is diagonally dominant, this iterative method converges unconditionally, and is reliable for most usual inputs.

There are some differences between the two approaches. For production problems, they are very likely to be constrained by time consumption of direct solver on first run. Therefore before the memory exhausts, it is very likely that the computation time becomes unacceptable. For iterative solver, the major constraint is time (iteration cycle numbers), for the memory consumption is of the same order of magnitude of the consumption for saving an input field. To make the solvers parallel, the partial-pivoting LU-decomposition method might need modifications in order to run on different cores or even nodes.

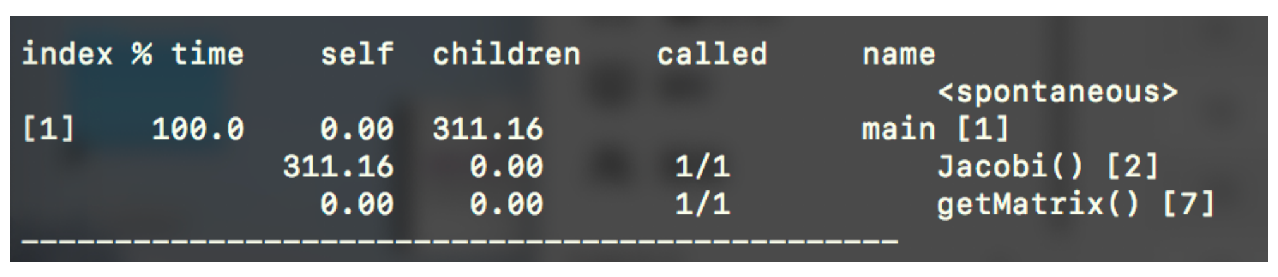
Library: Intel MKL

We select the Intel MKL external library to modify the Jacobi method. Generally speaking, we replace the matrix operations of Jacobi method with the library. The reason we use Intel MKL library is that this library has many advantages. The first advantage is free for student. As we know, many external library, such as IMSL, is commercial. It means that we have to pay for it. If we just use Intel MKL for our non-commercial research, it is free. The second advantage is speed. The Intel MKL is fast, and it is not complicated to use it to solve determinants and matrix inverse operations. It also supports some key operations.

After selecting the library, we should import this library in our code. And then use the internal functions to replace some matrix operations in our code. According to the experimental results, we can know that the MKL increase our performance. It means the code become faster than before. Intel MKL can use the Intel Xeon-phi co-processor to do parallel operations on hundreds of GPU cores. The speed is fast, and MKL automatically performs the operation offload, which is more convenient than the general GPU.

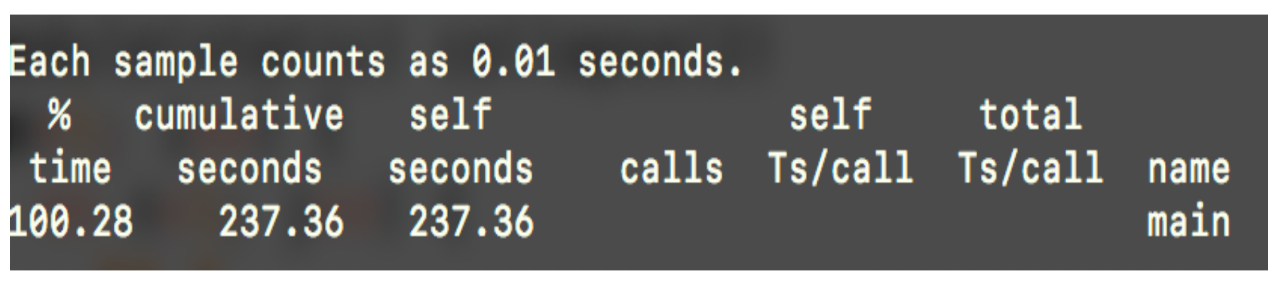
Parallelization: OpenMP

We select OpenMP to parallel our Jacobi method. There are several reasons to select this method. The main reason is that using this method makes the parallelization very easy. And the second reason is that OpenMP has minor changes to the original serial code to protect the original code. In addi- tionally, the code is easier to understand when we use OpenMP. The last advantage is that OpenMP allows progressive parallelization.
For the serial code, the iterative solver itself costs around the whole time. Here is a profiling figure as below:



```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00  311.16                 main [1]
              311.16    0.00       1/1           Jacobi() [2]
                0.00    0.00       1/1           getMatrix() [7]
-----------------------------------------------
```

Therefore the portion could be accelerated by parallelization is large for the solver. According to the second figure, we can see that the parallel code is faster than the serial code. To increase efficiency, we add the threads numbers in the code. In our program, the default threads are 4. We can modify the code to add the threads to 16 or more if necessary.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.28    237.36   237.36                             main
```

Acknowledgements

This project would not have been possible without the support of Dr. Adam Lavely, Dr. Christopher Blanton and my classmate Yueze Tan. I am especially indebted to my friends Tianyuan Wei and Xingzhao Yun. Tianyuan Wei is an Earth and Mineral Science student, he gave me his class notes and taught me how to solve the PDE step by step; Xingzhao Yun is good at C and CPP, he taught me the basic syntax. They worked actively to provide me the help to complete the Ax = b problem. I also appreciate the CS 267 course, University of California, Berkeley. I learned a lot from it.

References

- [1] Gupta, Murli M., Ram P. Manohar, and John W. Stephenson. ”A single cell high order scheme for theconvectiondiffusion equation with variable coefficients.” International Journal for Numerical Methodsin Fluids 4.7 (1984): 641-651.
- [2] Codina, Ramon. ”A discontinuity-capturing crosswind-dissipation for the finite element solution ofthe convection-diffusion equation.” Computer Methods in Applied Mechanics and Engineering 110.3-4(1993): 325-342.

Published Code Location

- https://github.com/William0617/CSE597HW1
- https://github.com/William0617/CSE597HW2
- https://github.com/William0617/CSE597HW3