Final Project: Progress Report 2

CSE 597

Wenliang Sun

Monday, November 5, 2018

Abstract

The convection-diffusion equation is a combination of the diffusion and convection equations, and describes physical phenomena where particles, energy, or other physical quantities are transferred inside a physical system due to two processes: diffusion and convection. Depending on context, the same equation can be called the advectiondiffusion equation, driftdiffusion equation, or scalar transport equation. In this project, we leverage a program to solve a 2D convection-diffusion equation. We have two different solvers: direct solver and iterative solver.

In the first report, we use LU decomposition and Jacobi methods to implement our project. This time we use OpenMP to parallel the Jacobi method. And then we get the profiling and scaling results.

1 Problem of Interest

As we learn in class, the problem is to solve partial differential equation.In my project, I am 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. My project solves the steady 2D convection-diffusion problem: $\frac{\partial \rho u_i \phi}{\partial x_i} = \frac{\partial(\tau \frac{\partial \phi}{\partial x_i})}{\partial x_i} + q_\phi$.

There are also some other methods to solve this problem: the discontinuity-capturing crosswind-dissipation for the finite element solution of the convection-diffusion equation[1], A single cell high order scheme for the convectiondiffusion equation with variable coefficients[2] and etc. We don't discuss too much about these methods in this report.

2 Parallelization

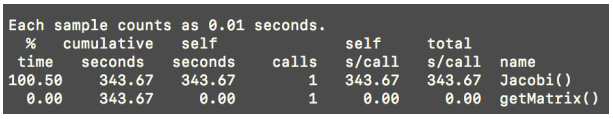
In this paper, we use OpenMP to parallel the Jacobi method. There are several reasons to select this method. The main reason is that using this method makes the parallelization very easy. And the second reason is that OpenMP has minor changes to the original serial code to protect the original code. In additionally, the code is easier to understand when we use OpenMP. The last advantage is that OpenMP allows progressive parallelization. We also learn MPI during classes, but I think it is more difficult to use than OpenMP and its performance will be affected by the network. We can also use Pthreads. However, this approach requires quite a bit of thread-specific code and is not guaranteed to scale reasonably with the number of available processors. The Map-Reduce technique belongs to DISC(Data intensive scalable computing), I think it is not very powerful in HPC field.

In this projecrt, we are going to parallelize the Jacobi method. In this function, we parallelize the "for" loop parts. It is pretty straightforward. According to the inner "for" loops and outer "for" loops, I think the parallelized percentages are 60%. For the initial condition, A-matrix is more complex than b-vector, and the A-matrix is 2 dimensional matrix, but b-vector is one dimensional matrix.

3 Profiling

3.1 Serial

In this project, we generate a 500 * 500 matrix. And then I use this matrix to calculate the result by serial Jacobi method. we select the "gprof" profiling tool to profile our code. The profiling result is shown in figure 1.



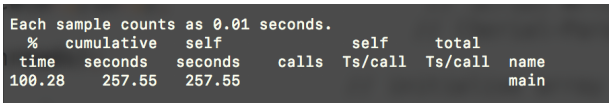
```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.50    343.67   343.67        1   343.67   343.67  Jacobi()
  0.00    343.67     0.00        1     0.00     0.00  getMatrix()
```

Figure 1: Profiling of serial code

According to the figure 1, we can see that the Jacobi function takes most of the executive time, generate matrix function and main function take about 0 second. Totally, this 500 * 500 matrix problem takes about 343.67 seconds. This is what we predicted before. The Jacobi calculation part, the main loops, takes most of time consumption. For the serial code, we think there are no potential steps. Because it is serial, we cannot make it more efficient. But we can increase its efficiency by parallel methods.

3.2 Parallel

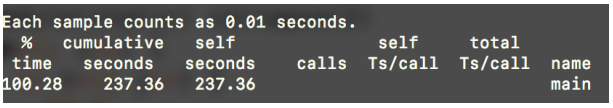
For the parallel code, we also use the 500 * 500 matrix and the tool gprof to evaluate it. Because the main loops part occupies most of the time, so we just write the code in the main function. The profiling result is shown in figure 2.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.28    257.55   257.55                             main
```

Figure 2: Profiling of parallel code

The figure 2 shows us the results of parallel code. As the serial code, the Jacobi calculation part, main loops, costs most of the time. This is what we expected. According to the figure 2, we can see that the parallel code is faster than the serial code. But the it also takes more memory than the serial one. I think the parallel code could be optimized. In our program, we only parallelize the main loops part of the code. However, the matrix generation part is also important. If the data cannot generate as soon as solve it, the performance is poor. So we parallelize the matrix generation part and then profile it in figure 3.



```
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
```

Figure 3: Profiling of optimized code

In figure 3, we can see the performance is better than figure 2. Another method to increase efficiency is that 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.

4 Scaling

4.1 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. The experimental result is as figure 4.

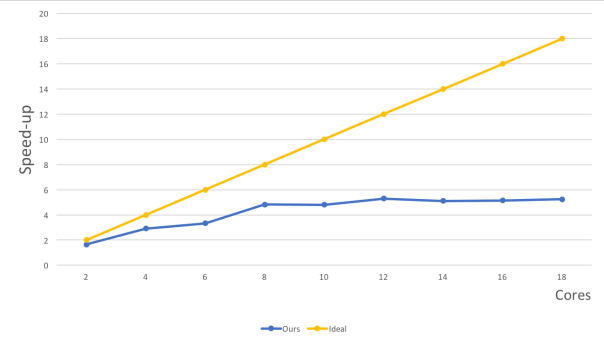


Figure 4: Strong scaling

In this figure, we can see that the speed grows with the cores. According to our results, we think 12 cores are best. So we want to use 16 cores. On the one hand, 16 cores performance is better than the others. On the other hand, in theory, more cores could be faster than fewer cores. Considering the efficiency of resource, we think 8 cores is better. As we know more cores mean more resource wasted. The performance of 8 cores is a balance point. It is as fast as more cores and it is also the most cost-effective choice. If I do some research, if possible, I want to use more cores. Because more cores could get the experimental results faster. In the figure 4, the orange line is ideal situation. So our result is not close to ideal line. But I think it is reasonable, the speed cannot increase indefinitely.

5 Discussion and Conclusions

In this report, we discuss the serial method and parallel method. Firstly, we profile the serial method and then profile the parallel method. After that, we find some potential steps to improve our parallel code and compare their profiling results. In the end, we discuss the strong scaling results. Because of the convenience, we select OpenMP. There are many built-in functions in OpenMP library, so it is very friendly for beginner. There are also some limitations and constraints in the report. The first one is the stable of the node. When we use the same node to run the same code, the results are different. It may influence the experimental results. Secondly, our code has some optimized areas. The last one is that we should add more experiments in the strong scaling part to make the result more reliable. We want to fix these flaws in the future, and try to use MPI or other parallelization methods to calculate the result.

Appendices

A 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.

B Code

- serial.c: it is the serial Jacobi method.
- parallel_original.c: It is the parallel Jacobi method.
- parallel.c: It is the optimized parallel Jacobi method.
- We use the comp-bc node of PSU ACI to run the codes.

  All files are on our github, the address is https://github.com/William0617/CSE597HW2. These above files can be compiled by the command as below:

```
// 1. serial
g++ serial.c -o serial.out -g -pg
./serial.out
gprof serial.out gmon.out
// 2. original parallel
g++ -fopenmp parallel_original.c -o original.out -g -pg
./original.out
gprof original.out gmon.out
// 3. optimized parallel
g++ -fopenmp parallel.c -o parallel.out -g -pg
//parallel.out
gprof parallel.out gmon.out
```

References

[1] Gupta, Murli M., Ram P. Manohar, and John W. Stephenson. "A single cell high order scheme for the convectiondiffusion equation with variable coefficients." International Journal for Numerical Methods in Fluids 4.7 (1984): 641-651.

[2] Codina, Ramon. "A discontinuity-capturing crosswind-dissipation for the finite element solution of the convection-diffusion equation." Computer Methods in Applied Mechanics and Engineering 110.3-4 (1993): 325-342.