Final Project Report

CSE 597

Wenliang Sun

Friday, December 7, 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 advection-diffusion equation, drift-diffusion 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. Firstly, we use the serial code to implement the basic functions. After that, we use OpenMP to parallelize the Jacobi method. And then we get the profiling and scaling results. The results show that parallel code has a better performance than the serial code. Additionally, we use the Intel MKL external library to parallelize the code. This method is effective and efficient. It increases performance about tenfold. After these experiments, we discuss the possible further work. It includes advanced decomposition, other libraries, parallel I/O libraries and etc. In the end, we show the limitation of our project and image use the future technologies, such as quantum computing, to improve our code performance.

1 Problem of Interest

The mathematical models that reflect the laws governing the movement of soil water and salt are mostly nonlinear partial differential equations. Before the 1960s, analytical methods were used to study the analytical or semi-analytical solutions of equations, but the analytical methods were only applicable to the problem of soil water movement under simple conditions. In order to study soil water and salt transport under more complex conditions, the most effective method at present is to use numerical calculation methods. 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.

The effective numerical solution to the convection-diffusion problem has always been an important research content in computational mathematics. The numerical methods for solving the convection-diffusion equation are mainly finite difference method (FDM), finite element method (FEM), finite volume method (FVM), and finite analytical method. (FAM), Boundary Element Method (BEM), Spectral Method (SM). Actually, there are two common methods to solve this equation: the discontinuity-capturing crosswind-dissipation for the finite element solution of the convection-diffusion equation, a single cell high order scheme for the convectiondiffusion equation with variable coefficients and etc. We don't discuss too much about these methods in this report.

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

1.1 Numerical Set-up

For the 2D convection diffusion equation, we assume: x∈[0,1] and y∈[0,1] and $\rho = 100$, $\tau = 0.1$, $q_\phi = 0$, $u_x = x$, and $u_y = -y$. The boundary conditions are ϕ=0 on y = 1, $\phi = \phi$(y) = 1-y on x=0, $\frac{\partial \phi}{\partial x} = 0$ on x = 1, and $\frac{\partial \phi}{\partial y}$ on y = 0.

According the above conditions, we can discretize the problem. We use a small size matrix as an simple example. And then divide the area into a 5*5 matrix as figure 1.

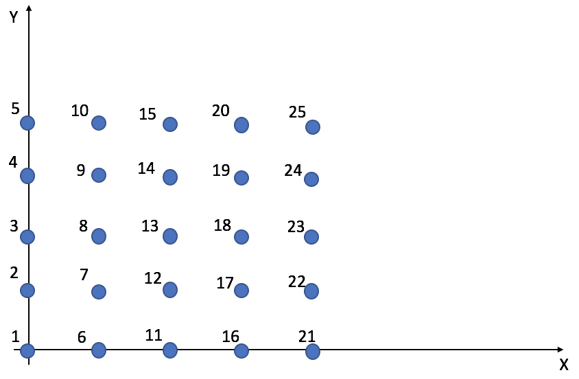


Figure 1: This is an example of a 5*5 matrix

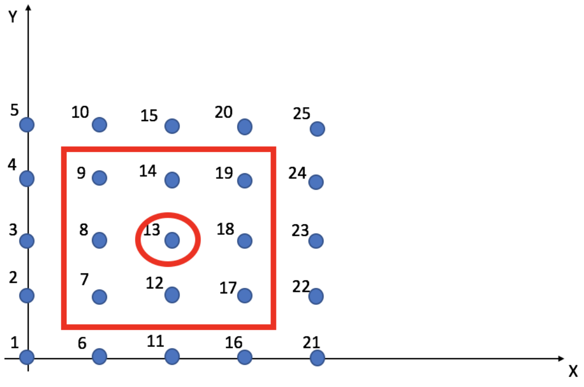


Figure 2: This is an example of an interior node

Firstly, we calculate the interior nodes. In the figure 2, the interior nodes are in the red circle. We use node 13 as an example. The other interior nodes' calculation are same. For the node 13, we leverage the Second-order Central Difference Scheme(CDS2) to solve it. According to the node 8, 18, 12, 14, we can get: $(\rho u_x/2h - \tau/h^2)\phi_{18} + (-\rho u_x/2h - \tau/h^2)\phi_8 + 4\tau/h^2\phi_{13} + (\rho u_y/2h - \tau/h^2)\phi_{14} + (-\rho u_y/2h - \tau/h^2)\phi_{12} = q_\phi$

In this equation, "h" stand for the distance between two nodes. Through this method, we can calculate all the interior nodes. And we can use the coefficients to build A matrix. There are nine interior nodes, so it could fill in nine lines in A matrix, and the same position of b matrix is "0".

Secondly, we should solve the boundary problem. For this example, there are four boundaries: left, right, up and down. The left boundary is a constant, it is "1-y". So we can fill in the A matrix with "0", and set the node's position in A matrix as "1". The corresponding position in b matrix is "1-y". The up boundary is similar as the left boundary, the difference is that the corresponding position in b matrix is "0".

But the right boundary is different, they are Newman Boundary Condition. We use Second-order Backward Difference Scheme(BDS2) to solve it. This method means that we use the two nodes behind it. We use node 23 as an example. We can get the equation:

$$\frac{\phi_1 3-4\phi_{18}+3\phi_{23}}{2\Delta x} = f(y)|_{23}$$

Because the $\frac{\partial \phi}{\partial x} = 0$ on x = 1, so we can get: $\phi_{13} - 4\phi_{18} + 3\phi_{23}$=0. And then we fill in A matrix with these coefficients. And the corresponding position of b matrix is "0". In the end, we can generate the A matrix and b matrix by merging the two parts as above.

2 Solvers

2.1 Direct Solver

In our project, we choose LU decomposition solver as the direct approach. As we learned in class, this type of solver decompose A matrix into L and U matrix. In the real world, there may be lots of calculations to solve the whole problem. So the matrix decomposition method is much more convenient than other methods such as Gauss elimination.

We use the following optimization flags:

- -O2 - Enabling the optimizations while not inducing a significant increasing of compiling time.
- -mavx - Enabling the optimizations for vectorized data to speed-up matrix operations.

According to the experiments, a 100 * 100 matrix takes about 40 seconds for the matrix decomposition. This matrix takes about 100 MB for saving the matrices L and U, and about another 30 MB to save the original matrix A. If the matrix is 5000 * 5000, the matrix could be 2500 times larger than currently constructed. It takes around 100GB memory space. It cannot be accepted. Additionally, the matrix decomposition takes about 200 days!

2.2 Iterative Solver

We use Jacobi method as iterative solver in our project. According to the Dr. Adam's code, we implement our Jacobi solver. Generally speaking, the Jacobi solver is based on the convergence. It also could be used in the future parallel implementations. In my test case, the two solvers seems like have similar time consumption. But according to the algorithm complexity, the iterative solver must be better than the direct solver when the problem scale is huge. For the memory problem, because my the scale of problem is small, so it just costs several kilobytes.

For the above test cases, the iterative solver takes less time than the direct solver. For the random, good guess, all zeros cases, the convergence is satisfying, and guessed initial value is slightly better than zero initial value, which is then better than random initial value. The result is in accordance with the expectation. Because the Jacobi method is too fast, we cannot measure the exact value. Therefore we get the result is around 20 ms. Memory allocated in this procedure is hard to track with the in-program tracking method. The memory allocation for a production problem is still considerably small, for the size of array still grows in O(N), and it around takes 3 MB for a 5000 * 5000 size problem.

2.3 Solver Comparison

For our certain problem, the iterative method has a better performance. The convergence of iterative methods varies widely among different problems. And the Jacobi iterative method are used to solve large sparse matrices.

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.

3 Parallelization

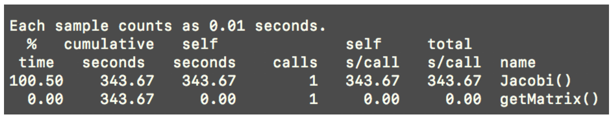
In this paper, we use OpenMP to parallel the Jacobi method. There are several reasons to select this method. The main reason is that using this method makes the parallelization very easy. And the second reason is that OpenMP has minor changes to the original serial code to protect the original code. 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.

3.1 Profiling

3.1.1 Serial

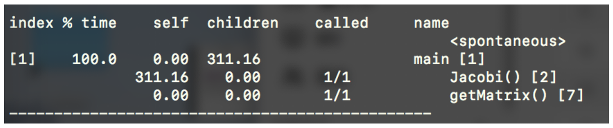
In this project, we generate a 500 * 500 matrix. And then I use this matrix to calculate the result by serial Jacobi method. we select the "gprof" profiling tool to profile our code. The profiling result is shown in figure 1.



```
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 3: 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. And then, we change the compilers flag from -O2 to -O3. Below is the updated profile with the new optimization flag.



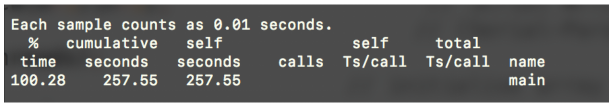
```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00  311.16                 main [1]
              311.16    0.00       1/1           Jacobi() [2]
                0.00    0.00       1/1           getMatrix() [7]
-----------------------------------------------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

Figure 4: The profile results of serial code with improved compiling flag.

According to the experimental result, we can the here is a slight improvement in the speed. For the same matrix, he time consumed decreases from 343.67 s to 311.16 s, shortened by around 9%. But we think that there must be some other reasons to influence the result, such as the load of the server, so we can ignore the slight improvement.

3.1.2 Parallel

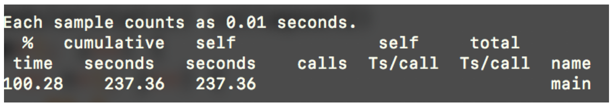
For the parallel code, we also use the 500 * 500 matrix and the tool gprof to evaluate it. Because the main loops part occupies most of the time, so we just write the code in the main function. The profiling result is shown in figure 2.



```
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 5: 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 6: 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.

3.2 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.

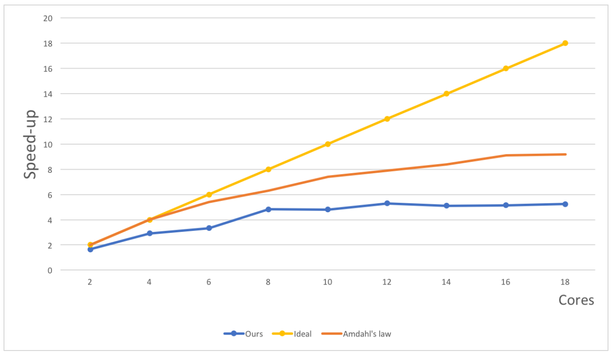


Figure 7: Strong scaling

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. 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.

4 Coupling to an External Library

4.1 Coupling

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. These operations include general matrix-vector multiplication, spare matrix-vector multiplication, symmetric matrix multiplication and etc. Additionally, Intel MKL is efficient, portable, and widely available.

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.

4.2 Potential Coupling

Except Jacobi method, we believe that the LU decomposition can be replaced with other libraries. The Basic Linear Algebra Subprograms (BLAS) is a specification that requires a set of low-level routines for performing linear algebra operations. The BLAS also includes many basic algebra operations. So it can be used in LU decomposition code.

For the LU decomposition part, we believe that Gaussian Elimination could also implement the code. Since it is an algorithm in linear algebra that can be used to solve the 2D convection-diffusion equations, find the rank of the matrix, and find the inverse matrix of the reversible square matrix. When used in a matrix, the Gaussian elimination method produces a row ladder. The PETSc library can be use in our code too. PETSc is a suite of data structures and routines for the scalable, parallel solution of scientific applications modeled by partial differential equations. It supports MPI, and GPUs through CUDA or OpenCL, as well as hybrid MPI-GPU parallelism. PETSc also contains the Tao optimization software library. Additionally, for a huge matrix, the parallel I/O library is also useful. Because I/O operations occupy lots of time of the overall performance. For a better performance, we can also add more GPUs to accelerate the program.

In the future, we are going to study the parallel I/O libraries. As we all know, I/O operations are more expensive than CPU calculations. If we want to further improve our application's performance, we have to leverage the parallel I/O approach. The matrix read and result write can be parallel. We plan to use the HDF5 model to make our code has a better performance.

5 Discussion and Conclusions

In this project, we describe the basic idea of the 2D convection-diffusion equation, and then solve the equation by direct solver and iterative solver. We also have discussed the method of discretization, the construction of A matrix and b matrix and the comparison of these two methods. Additionally, we discuss the serial method and parallel method. We leverage OpenMP in our Jacobi method. And then we profile the serial method and profile the parallel method. After that, we find some potential steps to improve our parallel code and compare their profiling results. We also discuss the strong scaling results. There are many built-in functions in OpenMP library, so it is very friendly for beginner. Finally, we couple to an external library. We analyze some popular external libraries and compare them. According to our code, we decide to use Intel MKL. We develop the Jacobi method with Intel MKL. And then we show the profiling result. We also discuss the potential coupling. It includes advanced decomposition, solver libraries, parallel I/O libraries and accelerators.

In conclusion, we find that the parallel code is faster than the serial code. The improvement depends on the code and hardware. External libraries provide more useful functions and routines increase the performance of the code. However, there are also some limitations of our project. The first one is the code.The first one is the stable of the node. When we use the same node to run the same code, the results are different. So it must influence the profiling and scaling results. The second one is the external library. We are not familiar with it, so there are some bugs in the code. The last limitation is that our parallelization is insufficient. We try our best to parallelize the code, but there are also some parts not be parallelized. In the future, we are going to fix the above problems. After that, we want to use MPI to parallelize the LU decomposition code and then get the profiling and scaling results.

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. In the future, maybe we can use quantum computing to calculate the equation. As we know, the quantum computer are more faster than current computers. Using it in parallel computation field, it must be a revolution. We can solve huge scale of matrix problems. In the end, we have some suggestions for people who want to learn the parallel computation. Taking a course about the parallel computation is needed. During the class, we can interact with instructors. After class, we can read some papers and books to consolidating the foundation. Last but not least, write some code to verify our idea is correct. Using this approach, we can understand the technique deeply.

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

Our code is pulished on GitHub. Its address is `https://github.com/William0617/CSE597HW3`. The mkl_solver.cpp is the code that call the Intel MKL library. And other codes can be found in `https://github.com/William0617/CSE597HW1` and `https://github.com/William0617/CSE597HW2`. We use the comp-bc node of PSU ACI to run the codes.

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.