An Exercise in Solving Differential Equations in Parallel on a Grid

Chad Bustard

Abstract

For my final project, I solve the Poisson equation on a grid using OpenMP. I use the Gauss-Seidel method (a chaotic relaxation method) to solve the equation, and I compare the convergence, result determinacy, and speedup of various algorithms utilizing this method. The basic sequential implementation utilizes a nested for loop. I find that a naive parallelization of both for loops increases the runtime significantly, whereas only parallelizing the outer loop leads to the most significant speedup compared to the sequential version. These chaotic relaxation methods, however, have poor result determinacy, meaning they do not lead to the same result each time they are run, even though they converge to the correct solution within the tolerance specified. I find that to get better result determinacy using the wavefront method or a simple implementation of an extra array to store the results leads to worse speedup.

PACS numbers:

I. PROJECT MOTIVATION

The goals of my final project were to learn how to solve a partial differential equation in parallel on a grid. My research in the physics department pertains to solving steady-state (and in the future, time-dependent) galactic wind equations. Galactic winds are essentially galactic-scale versions of the more well-known solar wind. The hot wind particles are driven outward by supernovae or active galactic nuclei (AGN) in a galaxy and serve to expel mass and energy from a galaxy. This is a form of feedback for the galaxy in that, if the galactic outflow expels enough mass from a galaxy, there is no longer enough mass to make stars, and star-formation is quenched. Therefore, these processes are very important for modeling galaxy evolution. The most common big models of galaxy formation utilize OpenMPI, though I believe some are now being converted to CUDA, as well, which was a large reason why I decided to take this course. For this work, I use OpenMP.

Galactic winds are generally described by fluid equations (many times with a complicated magnetic field structure affecting the motion of the wind particles), and these equations are partial differential equations with various boundary conditions. Therefore, this final project will teach me the basics of solving partial differentials on a grid in a parallelized manner and that I can use this knowledge in the future in my own galaxy formation research.

II. METHOD

My work here does not involve the actual fluid dynamics equations used in my research. For simplicity, I've chosen to instead solve the Poisson equation with Dirichlet boundary conditions using a finite difference method, which seems to be a standard exercise. The equation is solved using the Gauss-Seidel method, which updates approximations to the function in question at each grid point until the function converges to some value at each point. The condition for convergence is specified by a required accuracy, i.e. a tolerance, where if the maximum difference in values between two iterations of the function are less than that tolerance, then we say the function has converged. More specifically, the method is as follows:

First, I discretize the domain $0 \leq x \leq 1$, $0 \leq y \leq 1$ using N points in each direction, or N^2 grid points in total.

I then put in the boundary conditions that I want. In the work presented here, I use the same boundary conditions as [3]:

$$u(x, y = 0) = 100 - 200x$$
$$u(x = 0, y) = 100 - 200y$$
$$u(x, y = 1) = -100 + 200x$$
$$u(x = 1, y) = -100 + 200y$$

I then set each grid point inside this boundary to have $u(x, y) = 1$. One can also set these points to have random values.

The Poisson equation I solve is

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Using the Gauss-Seidel method, this amounts to updating each grid point as

$$u^k_{i,j} = 0.25(u^k_{i-1,j} + u^{k-1}_{i+1,j} + u^k_{i,j-1} + u^{k-1}_{i,j+1} - h^2 f_{i,j})$$

Using this set-up, the solution given by Mathematica looks like the following:

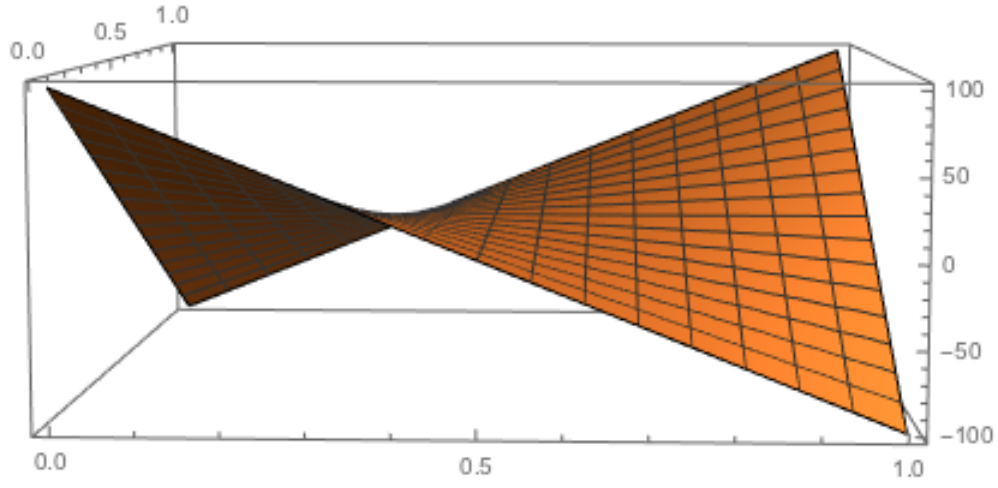

FIG. 1: Solution to the Poisson equation with the specified Dirichlet boundary conditions.

III. COMPARING PARALLEL ALGORITHMS

The key part of the code is the portion where each grid point is updated according to the Gauss-Seidel method. For the sequential method, it looks like the following:

```
do
{
        maximum = 0;
        for(i = 1; i < N; i++) {
                for(j = 1; j < N; j++) {
                        previous = u[i][j];
                        u[i][j] = 0.25*(u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] - pow(h,2)*f[i][j]);
                        difference = fabs(previous-u[i][j]);
                        if (difference > maximum)
                                maximum = difference;
                }
        }
        count ++; //count number of iterations until convergence
} while (maximum > tolerance && count < 1000);
```

FIG. 2: Sequential algorithm

One can see that the variable 'maximum' is accessed at each (i,j), meaning that in a parallel implementation, one has to be careful to lock this variable so only one thread can access it at a time. This, of course, causes a significant slowdown of the code. The most naive parallel implementation is shown below, in which I parallelize both for loops and lock 'maximum'.

```
do
{
        maximum = 0;
#pragma omp parallel for shared(u,maximum) private(difference)
        for(int i = 1; i < N; i++) {
#pragma omp parallel for shared(u,maximum) private(difference)
                for(int j = 1; j < N; j++) {
                        const double previous = u[i][j];
                        u[i][j] = 0.25*(u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] - pow(h,2)*f[i][j]);
                        const double difference = fabs(previous-u[i][j]);
                        omp_set_lock(&maximum_lock);
                        if (difference > maximum)
                                maximum = difference;
                        omp_unset_lock(&maximum_lock);
                }
        }
        count ++; //count number of iterations until convergence
} while (maximum > tolerance && count < 10000);
```

FIG. 3: Naive parallel implementation using two nested parallel sections. This method is slower than the sequential method due to timely synchronization between threads. Also, the basic locking of 'maximum' slows down the algorithm.

This method is considerably slower than the sequential method due to the heavy synchronization between threads and the naive locking of 'maximum'. In my second attempt at a parallel algorithm (parallel 2), I only parallelize the outer for loop, meaning each thread will run through the inner for loop. Because this requires less synchronization between threads, this method is significantly faster than both the naive parallel algorithm and the sequential algorithm. It should be noted that I also utilize the built-in OpenMP reduction to find the maximum. Attempting to increase the speedup by playing with the locking set-up in parallel 1 led to a number of errors, all of which were solved by simply using the reduction.

```
    do {
        maximum = 0;
#pragma omp parallel for shared(u) reduction(max:maximum)
        for (int i = 1; i < N; i++) {
            for (int j = 1; j < N; j++) {
                const double previous = u[i][j];
                u[i][j] = 0.25 * (u[i - 1][j] + u[i + 1][j] + u[i][j - 1] + u[i][j + 1] - pow(h, 2) * f[i][j]);
                const double difference = fabs(previous - u[i][j]);
                if (difference > maximum) maximum = difference;
            }
        }
        count++; //count number of iterations until convergence
    } while (maximum > tolerance && count < 1000);
```

FIG. 4: Second parallel implementation. Here, only the outer loop is parallelized, meaning each thread runs through the entire inner for loop. This method has less synchronization, leading to a significant speedup compared to the naive parallel implementation and the sequential implementation. Using the built-in reduction to find the maximum difference between iterations is also much easier to implement and leads to many fewer issues than trying to do my own reduction, as shown previously using locks on the 'maximum' variable.

A. Race Conditions and Result Determinacy Issues

The two parallel algorithms presented above both converge to the correct solution within the specified tolerance; however, they do not result in consistent answers each time they are run. This is because the value at each grid point is calculated using values from neighboring grid points, which are under control of different threads. This results in race conditions between the threads. Suppose one thread calculates it's grid point and moves on before one of its neighboring points has been updated. Then the next point may be updated using neighboring points that have already been updated instead of points that are on the same

iteration, as it should be in the Gauss-Seidel method. Then, when the program is run again, this may occur for different points, resulting in slightly different solutions each run. This is called poor "result determinacy." Naturally, we should require that parallelizing an algorithm shouldn't lead to different results from the sequential version, but this is not so for the first two parallel versions I have implemented.

The simplest way to avoid this is to define a new array unew(x,y) where one can store the values of u(x,y) at each iteration, thereby ensuring that each u(x,y) is updated using neighboring values on the k-1 -th iteration instead of some that may have been updated by other threads to the k -th iteration. This is shown in the following code fragment (parallel 3):

```
    do {
        maximum = 0;
#pragma omp parallel for shared(u) reduction(max:maximum)
        for (int i = 1; i < N; i++) {
            for (int j = 1; j < N; j++) {
                const double previous = u[i][j];
                u_new[i][j] = 0.25 * (u[i - 1][j] + u[i + 1][j] + u[i][j - 1] + u[i][j + 1] - pow(h, 2) * f[i][j]);
                const double difference = fabs(previous - u_new[i][j]);
                if (difference > maximum) maximum = difference;
            }
        }
        for (int i = 1; i < N; i++) {
                for (int j = 1; j<N;j++) {
                        u[i][j] = u_new[i][j];
                }
        }
        count++; //count number of iterations until convergence
    } while (maximum > tolerance && count < 1000);
```

FIG. 5: Third parallel algorithm that writes each u(x,y) to a new array unew(x,y) in an attempt to get better "result determinacy." At the end of each loop, unew(x,y) must be copied back to u(x,y), leading to significant slowdown of the algorithm compared to the second algorithm (parallel 2).

This third algorithm, besides being slower overall than the second algorithm, also has worse convergence.

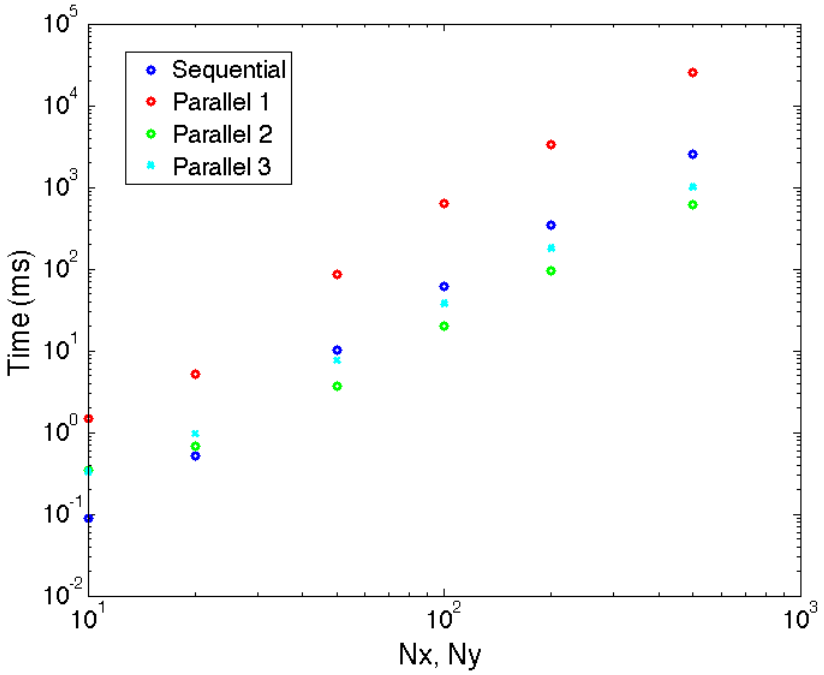

FIG. 6: Comparing times for the naive parallel implementation (parallel 1), the second parallel implementation where only the outer for loop is parallelized (parallel 2), the third parallel implementation where values for $u(x, y)$ are stored in a new array unew(x,y) at each iteration (parallel 3), and the sequential version. For each of these runs, I use 8 processors on Euler, and a tolerance of 0.1. With small grids, of course using 8 processors shouldn't lead to significant speedup. For larger grids, the difference between sequential times and parallel times becomes more noticeable. The second parallel algorithm is the fastest, while the same algorithm using a new array unew(x,y) to store the grid values $u(x, y)$ at each iteration is slower due to the overhead associated with accessing that array and copying values to and from it many times.

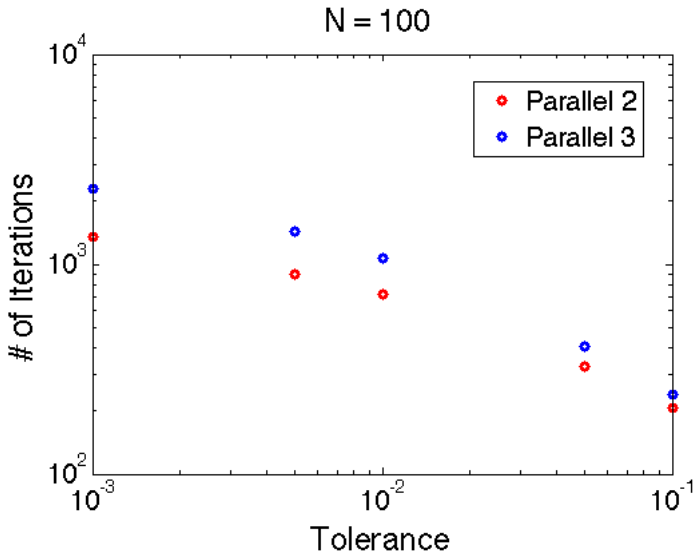

FIG. 7: Plot of the number of iterations required to converge to within various tolerances for the second and third parallel algorithms. The third parallel algorithm, in which I store $u(x, y)$ in a new array unew(x,y) at each iteration, not only is slower due to the increased overhead of copying to and from unew but also converges slower than the second parallel algorithm. Therefore, to get consistent results each time the code is run, which the third parallel algorithm achieves, one has to sacrifice both time and convergence.

B. Wavefront Method

In an attempt to gain considerable speedup while retaining good result determinacy, I also implemented the "wavefront" method, which should undergo the same sequence of operations as the sequential algorithm, thereby giving the same result each run and giving consistency between the sequential version and this parallelized version.

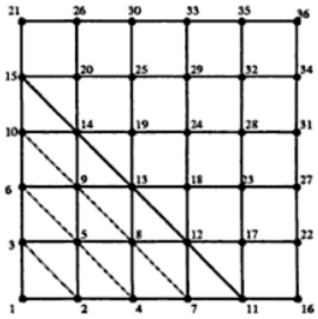

FIG. 8: Image courtesy of [2]. The main code is comprised of two parallel segments. One builds the wave from one corner to the middle of the grid, while the other loop decays the wave from the middle of the grid to the opposite corner.

The main code fragment is given below.

```
do
{
        maximum = 0;
        for(int ix = 1; ix < N; ix++) { //building the wave
  //              innerMaximum[ix] = 0;
#pragma omp parallel for shared(u,ix) reduction(max:maximum)
                for(int i = 1; i < ix + 1; i++) {
                        int j = ix + 1 - i;
                        const double previous = u[i][j];
                        u[i][j] = 0.25*(u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] - pow(h,2)*f[i][j]);
                        const double difference = fabs(previous-u[i][j]);
                        if (difference > maximum) maximum = difference;
                }
        }
        for(int ix = N - 2; ix > 0; ix--) { //decaying wave
#pragma omp parallel for shared(u,ix) reduction(max:maximum)
                for(int i = N-ix; i<N;i++) {
                        int j = 2*N - ix - i - 1;
                        const double previous = u[i][j];
                        u[i][j] = 0.25*(u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] - pow(h,2)*f[i][j]);
                        const double difference = fabs(previous-u[i][j]);
                        if (difference > maximum) maximum = difference;
                }
        }
        count ++; //count number of iterations until convergence
} while (maximum > tolerance && count < 10000);
```

FIG. 9: Wavefront method

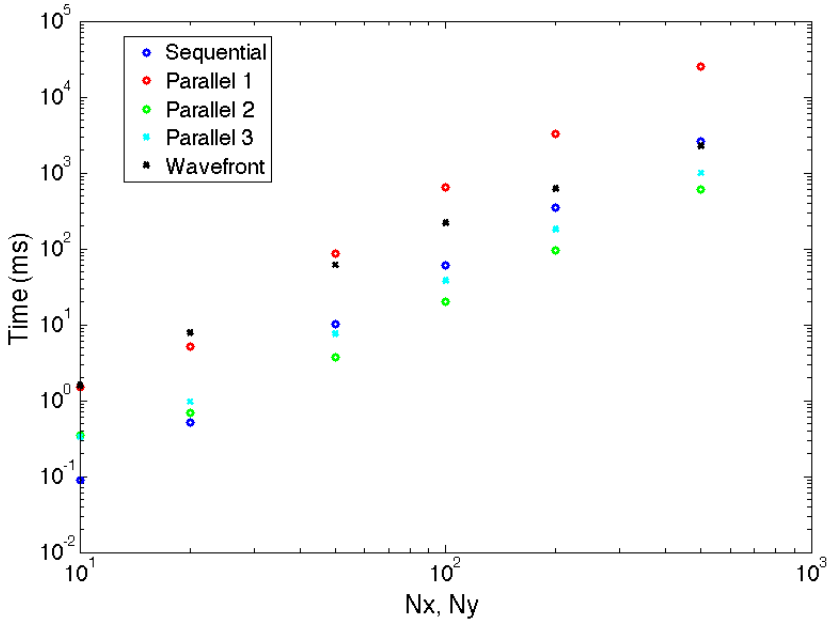

FIG. 10: Time analysis showing all methods tested in this project. The wavefront method gives the same result each run, which is also the same result that the sequential run gives. However, the computation time is longer than that of the second parallel algorithm, which is consistently the fastest implementation but does not give the same result each run.

IV. CONCLUSIONS

My conclusions from this exercise are as follows: the most naive parallel implementation has the worst performance due to the significant synchronization between threads involved in parallelizing each loop. Naively locking the maximum value led to further slowdown. It is much easier and better to use the built-in reduction to find the maximum value. This was implemented in the second parallel algorithm in which I also only parallelized the outer loop, leaving the inner loop to be executed by each thread. This implementation is the fastest method tested; however, the results are not the same each time the program is run. This can be resolved by storing the values at each point in a temporary array or by using the wavefront method, which will give the same result as the sequential method each run. This is very good because parallelizing a portion of sequential code should not lead to different results; however, the speedup is not as impressive as the second parallel algorithm. Therefore, there is a clear trade-off between result determinacy and speedup.

Overall, by doing this project, I became much more aware of how computationally expensive memory accesses can be when using multiple threads, as well as how expensive synchronization can be, as was the case for the very slow, naive parallel implementation. It is also important to be aware of race conditions when using a method, such as the Gauss-Seidel method, that updates grid points using values of neighboring points, which are being executed by other threads. This can lead to result indeterminacy.

In the future, I plan to utilize these methods in my own research, as well as possibly implementing the red-black tiling scheme, which is another frequently used scheme to avoid race conditions.

---

[1] Jianping Zhu *Solving Partial Differential Equations on Parallel Computers*. World Scientific Publishing Co., 1994

[2] Nils Magnus Larsgard *Parallelizing Particle-In-Cell Codes with OpenMP and MPI* Student Thesis, Norwegian University of Science and Technology, 2007 (English)
`http://www.idi.ntnu.no/ elster/master-studs/larsgaard/larsgaard-master.pdf`

[3] `http://www.hpcc.unn.ru/mskurs/ENG/DOC/pp12.pdf`