Laplace PDE Solver

Program Description

This program was developed to solve Laplace's 2d Partial Differential Equation. Two different methods were used to accomplish this, Jacobi Iteration, and Multigrid. Both of these methods were developed in a concurrent shared memory model, and in a sequential format. The multigrid models of tackling the problem built upon the Jacobi Iteration techniques.

Laplace's equation, or the heat equation as it is also known, is the del² operator applied to a function. This essentially takes the second derivative of each and every infinitesimal point of the input space, and solves these for zero. The second derivative of a point, or its rate of change of rate of change, is dependent on the points next to it. Essentially, if there is a large difference between a point and its neighbours, this means a larger second derivative. It is called the heat equation because it is a good model of how heat flows in a gas. Each point moves towards the average of its neighbours. The solution to this is when there is no longer any heat flow on the input space, or when the temperatures have reached equilibrium.

In order to solve this equation, an input space was abstracted to a grid. Or, a small area is assigned a single matrix grid holding an integer value reflecting its "temperature", or some other parameter. Jacobi iteration then consists of repeatedly updating the each grid point to be the average of its direct neighbours. This is done for all grid points, a predetermined number of iterations. Since the boundary values, or the values on the border of the matrix, consist of constant 1's, a solution to the equation would be a grid consisting entirely of 1's. The maximum error of the matrix, or the maximum distance of a gridpoint from its neighbours.

The Sequential Jacobi solver uses 3 additional help methods. PrintMatrix, initializeMatrix, and printtoFile. PrintMatrix and printtoFile prints the current state of the matrix to stdout/filedata.out. Initialize matrix sets the outer rows and columns to all 1's, initializes the matrix to the correct size, and sets interior points to 0.

The concurrent version of this program works in a similar way. The only difference is that the outer for loop of the updateMatrix method, the method which sets the matrix to its new value for next iteration, is preceded by a #pragma parallel for. This is a compiler directive from the OMP framework telling the compiler to split the for loop up into chunks that can be executed in parallel by different threads.

The multigrid solvers of the PDE use the previous solvers. The multigrid sequential program starts with a fine grid, an input space represented with a lot of matrix cells, and iterate using Jacobi iteration, four times. The grid is then restricted. This means that the number of grid points of the matrix are reduced. With each new grid point taking an average of the grid points it's meant to represent. If the matrix is meant to represent an input space, each grid point represents a larger area, and the grid becomes coarser. The grid is then iterated on four times

and reduced another 3 times. When the grid reaches its coarsest incarnation, the grid is iterated upon the amount of times given as command line argument to the program. Then, in corresponding fashion, the grid is interpolated. This means that the grid is sized up, with the new grid points taking on an average of the grid point they represent. This is repeated until the matrix is the same size as in the beginning.
The concurrent version of the multigrid program used the concurrent version of the Jacobi solver. Furthermore, it parallelized the two for loops used during the restriction and interpolation operations also.

Sequential Jacobi Program

Grid Size	Execution time (microseconds)
100	483115
200	1999635

Concurrent Jacobi Program

Number of Workers	Execution Time @ Grid Size 100	Execution Time @ Grid Size 200
1	2851500	3495882
2	2882094	3480504
3	2874840	3526851
4	2851582	3462464
5	2860991	3462660

Sequential Multigrid Program

Grid Size	Execution Time
17	398
33	1046

Concurrent Multigrid Program

Number of Workers	Execution Time @ Grid Size 17	Execution Time @ Grid Size 33

1	463	1179
2	542	995
3	826	940
4	832	1462
5	1441	2389

As these tables tell us, it is not useful to talk about any type of speedup. Almost all of the concurrent programs ran significantly slower, with the best case being a comparable runtime for the multigrid programs at larger input size, with 2 or 3 threads.

The Jacobi Iteration programs do show interesting patterns. If we look at the execution times for the concurrent version of the program, it does not seem to matter much how many threads are employed. The time is roughly constant. However, if we compare the execution time between the sequential and the concurrent programs at the smaller size, the execution time of the concurrent version is absolutely atrocious. It is almost 6 times as slow. If we do this same comparison for the larger input size, the execution time of the concurrent version is still bad, however not nearly as bad. The execution times are of the same order of magnitude, with the concurrent version being approximately twice as slow. This tells an interesting story. Perhaps the concurrent version is so much worse because matrix operations are generally very fast, and the overhead associated with spawning and managing threads is not worth it for smaller matrices. To test this hypothesis, I ran both programs at an input size of 1000, significantly larger than the specified tests. The results confirmed this: at 100 iterations with 4 workers, the concurrent program achieved a speedup of roughly 2.5.

A similar line of reasoning can be taken with the multigrid programs. Since these are done on such small matrices, and since there are so many overheads associated with the program: creating new matrices, restricting, interpolating… perhaps the gains of making the program concurrent do not show themselves until the grid we are doing computations on is huge.