Project Report

Shiva Besharat Pour, Nora Alnaami

March 18, 2018

1 Introduction

Briefly describe the problem and what your report will show.

Partial differential equations (PDE) are used to model different phenomenas such as weather, heat, electrodynamics, airflow over wings etc. In order to solve PDE, Laplace equations in two dimensions are used (Laplace equation in two dimensions). Different iterative methods exist for computing Laplace equations in second dimension such as Jacobi iteration, Gauss-seidel, successive over-relaxation and mutligrids. In this lab, Jacobi iterations and multigrids will be used to solve Laplace equations. Also, both of the methods will be presented computed sequenctially and in parallel using openMP. The performance of each program will be calculated for different grid sizes and for different amount of processes working in parallel for the parallel programs. The maximum error in final values is also computed for each program. The main goal of this lab is to develop and to evaluate efficient parallel programs for a grid computation to solve partial differential equations as well as conducting performance experiments and analyzing the performance results.

2 Programs

Describe your programs, stating what each does and how. Explain the program-level optimizations you have implemented.

The programs were implement and tested with respect to their efficiency on a Macbook Pro, running the Operating System macOS High Sierra version 10.13.2. This MacBook's processor is a 3,1 GHz Intel Core i5 with 2 cores. Program 1, Sequnetial Jacobi : This program uses Jacobi iteration method to Solve Laplace equation. It updates a grid point value using the average of its 4 neighboring values, top, left, bottom right. It does so for user-defined amount of iterations and finally presents the maximum error in final values as well as printing the matrices to a seperate file. The efficiency implementations are as follows: - To calculate every grid point, multiplication by 0.25 is used instead of division by 4, because fewer machine cycles are needed for multiplication than division.
- A counter is used to count the iterations instead of using the difference computation to terminate the iterations. The difference is calculated only once after the grid computation.
- Instead of copying the grid which takes time, 2 grids are being used at a time and both are updated at each iteration of the for loop which required for the iterations to be divided by 2. This avoids the need for update.
- To calculate maximum difference, instead of using math functions such as max and absolute value which are timetaking, a simple if-statement is used instead.

Program 2 , Parallel Jacobi: This program is very similar to program, in fact just a copy-paste. However, it uses openmp to paralleize the jacobi iterations and a barrier function as necssary. The difference is calculated only once from a master thread.

Program 3, Sequential Multithread: It uses 8 sepearate grids allocated on the heap. Thi is to use the 4-level V-cycle as well as avoiding updating. The performance optimizations are mainly

similar to those in program 1. Except memory requirements are of course changed. Jacobi, restriction and interpolation functions are implemented as described in the book, lecture 18, as well as the task description itself. They are then used in the main function to run the V-cycle execution using fine and coarse grid points.

Program 4 , Parallel Multithread: Optimizations and implementation is very similar to program 3 except an added barrier function taken from program 2, as well as parallelization using openmp.

3 Performance Evaluation

Present the results from the timing experiments. Use tables to present the raw data and graphs to show speedups and comparisons. Also explain your results. Do not just present the output data! What do the results show? Why?

Grid Size	Time(sec)
100	0.001989
200	0.006938

Table 1: Program 1 with 20 iterations.

Number of threads	Grid Size	Time(sec)
1	100	0.001836
1	200	0.0065
2	100	0.006616
2	200	0.011881
3	100	0.004571
3	200	0.010133
4	100	0.008485
4	200	0.013659

Table 2: Program 2 with 20 iterations.

size	time
12	0.087954
24	0.346831

Table 3: Program 3 with 20 iterations.

number of threads	grid size	time
1	12	0.086768
1	24	0.331217
2	12	0.057168
2	24	0.199943
3	12	0.055026
3	24	0.177504
4	12	0.053245
4	24	0.175313

Table 4: Program 4 with 20 iterations.

The results show that generally the parallel implementation of both multigrid and jacobi iterations have a slowdown compared to the sequential versions. One might think this is unusual but this is in fact very expected considering the fact that parallelism aims to speedup independent iterations in iterative algorithms such as the ones implemented in these programs. However the iterations in these programs are all dependent on each other, since every grid point value is updated based on the old values of its neighbouring grid points and this effect simulates along the entire grid. Therefor parallelism not only speeds up the provcess, rather slows it down because it works just as if everything was pretty much sequential because every thread has to wait for other threads to do their work in order to preceed, at several stages(the barrier). Thus it becomes as if an extra delay was added to the sequential implementation and thus the execution time becomes longer.

4 Conclusion

Briefly summarize what your report has shown, and describe what you have learned, and any problems that you have faced when implementing the project.

Parallelism improves speedup when it is independent calls that are being parallelized. This is regardless of whether the program is iterative or recursive. It is the independent recursive calls or iterations that can actually be done simultaneously that speeds up the program execution when they are being executed buy several threads at the same time. However in algorithms such as jacobi and multigrid iterations where there is data-dependency between every single iteration and in fact, the entire algorithm is fundamentally data-dependent parallelism will result in slow down. This is because iterative calls cannot be executed simultaneously due to data-dependency and therefore regardless of number of threads being used, they still have to wait for each other's results which adds extra delay due to synchronization, to the program.

References

Andrews, G. (n.d.). Foundations of multithreaded, parallel, and distributed programming. Reading, Mass. [u.a.]: Addison-Wesley.