ID1217 - PDE Solvers

Lucas Kristiansson - lucaskr@kth.se
Victor Naumburg - victorna@kth.se

March 11, 2024

1 Introduction

In this report, we implement a multithreaded solution to a partial differential equation as per Laplace's equation. This problem may also be called the "stationary heat problem" since the Laplace's equation states that the rate of change must be 0, which implies that it merely distributes some values over a grid. In this case, the metaphorical heatpoints from which heat diffuses in the system finds itself at the borders at which the heatpoint intensity is 1, and where the interior to which diffusion may happen is 0. This solution performs such a diffusion iteratively and with discrete changes, thereby only arriving at an approximate end result to what a possible continuous implementation of this problem would result in.

Below in this report, we implement two versions of this problem: using the Jacobi computation method and the multigrid computation method.

2 Software and computing platforms

Implementation and compilation was done against the gcc version 14.2.1 compiler.

Performance evaluations were done on an Arch Linux desktop with an AMD Ryzen 9 3900x CPU.

We're using *timespec_get*() to get high precision wall clock timers for all our performance evaluations. This works well because none of the threads will be kept running past the implicit barriers of our code, so the timers will be accurate, along with being able to use the same timers for all four implementations, so the comparisons will be fair.

3 Design and Implementation

We've created four different PDE solvers for Laplace's equation. They are as follows:

- A sequential Jacobi iteration program
- A parallel Jacobi iteration program
- A sequential multigrid program
- A parallel multigrid program

As per what I was told after a homework presentation, it was merely necessary to do the sequential iteration programs. To conserve space, only the parallel implementations are described here.

3.1 paralleljacobi.c

3.1.1 What it executes and how it executes

The code first allocates the memory for an n by n grid where each entry is a double. The code then sets 0.0 to the entries that are in the interior of the grid, and 1.0 to the entries that are in the border of the grid.

Afterwards, the master thread sets the amount of threads that should operate in the program and then initializes the timer. Then it executes the *JacobiIteration* method with the specified amount of threads, to then turn off the timer.

The *JacobiIteration* method works as follows. The method iterates *numIters* amount of times with a multithreading OpenMP directive *pragma omp parallel for reduction(max:maxDiff) private(diff) schedule(static)*. The *schedule(static)* assures an equal distribution of rows over which a thread may operate, with at most 1 row's difference between the thread working on the least amount of rows and the one(s) working on more rows. The *private(diff)* assigns *diff* variables unique to each thread which in combination with *reduction(max:maxDiff)* makes so the shared *maxDiff* value smoothly updates when a difference greater than the current *maxDiff* arises. Other than the *maxDiff*, the directive also applies the instruction's formula on how to do the Jacobi approximation on each point of the grid.

The method finishes with returning the *maxDiff*, which corresponds to the instruction's epsilon value. Afterwards, the timer is finished and the program proceeds to finding the *maxError* error margin via the method *calculateMaxError*.

3.1.2 Program-level optimizations

Rather self-explanatorily, this program optimizes its code with respect to the sequential implementation by simply using OpenMP's multithreading capacities functionalities.

3.2 parallellmultigrid.c

3.2.1 What it executes and how it executes

In order to effectuate the 4-level V-cycle, parallelmultigrid creates 4 pairs of nested n by n arrays serving as the grid on which the program operates. The pairs complement each other with regards to rendering the grids coarse or finer respectively. This is reflected in the program by the *multigrid* instance containing pointers to *grids* and *newGrids* which are pointers to *Grid* instances. These grid instances in turn are composed of n by n amount of entries where each row is a consecutive segment of dedicated *double* memory segments - i.e. the entries of the grid. This pointer solution's purpose is to allow for variable array size creation as well as to make it possible to easily call the grids in diverse functions. Once the grids' have had their matrix' consecutive double memory segments entries allocated, the program "initializes" each respective grid by setting the border entries' values as 1.0, and the interior entries' values as 0.0.

After that, the program implements the four-level V cycle that uses restriction and interpolation. All of these operations are done with the *numWorkers* amount of threads.

First, the *vCycle* method performs 3 iterations of the following, which corresponds to going from finer to coarser grids: 4 iterations of applying the Jacobi PDE-solver (as per the instruction's specified "Use exactly four iterations on each finer grid") and then transferring that Jacobi-smoothed grid to the adjacent coarser grid, in the process adapting it in accordance to the instruction's *restriction* formula where each point is a reduction of the finer grid's points. In each instance where the Jacobi PDE-solver is used, it is followed up by a *swapGrid* operation which serves to assure that the grid is smoothed by the Jacobi method with finder and finder renditions of the grid.

Secondly, at the coarsest level, the *vCycle* executes *numIters* amount of Jacobi PDE-solver refinement methods.

Finally, the program initiates the steps where it goes from coarsest to finest. This step is done within a for loop that performs the following sentences three times. First, the program creates a copy of the adjacent finer grid. Then, the program interpolates this copy with the current coarse grid. This interpolation operation first maps the coarse grid's contents onto the fine grid. It then refines these mapped entries such that the fine grid exploits its increased grid space to have a more fine distribution of the coarse grids' points. An ensuing nested for

loop adds the initial fine-grained grid's values on the interpolated fine-grained grid's values in order to reduce the error margin. Finally, the jacobi PDE-solver smoothening computation operation is done four times.

3.2.2 Program-level optimizations

Due to an encountered problem of false sharing, the grids' *double* types were replaced by the struct *PaddedDouble*. On most modern CPUs the cache line size is 64 bytes, and the size of a double is 8 bytes. Even if threads don't operate on the same variable, the cache can treat the memory as if they do (bring 64 bytes of data to the cache line...) and cause that data to be swapped back and forth between processors when they need said data. This degrades performance. To prevent this, we created our PaddedDouble struct, which holds our double value and an additional 56 bytes of buffer data, making the PaddedDouble struct exactly 64 bytes big, preventing any false sharing.

4 Performance evaluation

4.1 Results

src	size	iterations	workers	time (s)	max error
jacobi	100	500000	1	32.25	0.00
jacobi	200	150000	1	38.79	0.00
paralleljacobi	100	500000	1	29.21	0.00
paralleljacobi	100	500000	2	16.44	0.00
paralleljacobi	100	500000	3	11.67	0.00
paralleljacobi	100	500000	4	9.28	0.00
paralleljacobi	200	150000	1	35.87	0.00
paralleljacobi	200	150000	2	19.86	0.00
paralleljacobi	200	150000	3	13.31	0.00
paralleljacobi	200	150000	4	10.48	0.00
multigrid	12	300000000	1	30.46	0.00
multigrid	24	100000000	1	35.99	0.00
multigrid	128	3000000	1	28.78	0.00
parallelmultigrid	12	300000000	1	111.97	0.00
parallelmultigrid	12	300000000	2	417.68	0.00
parallelmultigrid	12	300000000	3	NaN	NaN
parallelmultigrid	12	300000000	4	NaN	NaN
parallelmultigrid	24	100000000	1	62.86	0.00
parallelmultigrid	24	100000000	2	164.83	0.00
parallelmultigrid	24	100000000	3	NaN	NaN
parallelmultigrid	24	100000000	4	NaN	NaN
parallelmultigrid	128	3000000	1	33.04	0.00
parallelmultigrid	128	3000000	2	20.85	0.00
parallelmultigrid	128	3000000	3	17.83	0.00
parallelmultigrid	128	3000000	4	17.63	0.00

4.2 Analysis

Analysing the results for the paralleljacobi vs jacobi results, we can see a clear performance gain scaling somewhat linearly with the amount of worker threads. This is unsurprising, since while each iteration has to wait for the previous iterations result before starting (and therefore can't be parallelised), each iteration still has a relatively large problem size with size * size amount of points in a grid. For sizes of 100 or 200 that means 10 000 or 40 000 points to split between threads.

For multigrid vs multigrid results, there were a few issues. A multigrid v-cycle solution is sequential in nature, and so can not be parallelised in of itself. The parts that can be parallelised are the smoothing, restriction, prolongation, and correction steps. This is all well for problems with a large enough matrix (size * size), as we can see in the runs with 128 as the input size (and it scales even better with higher numbers like 256 etc...), but the project requirements

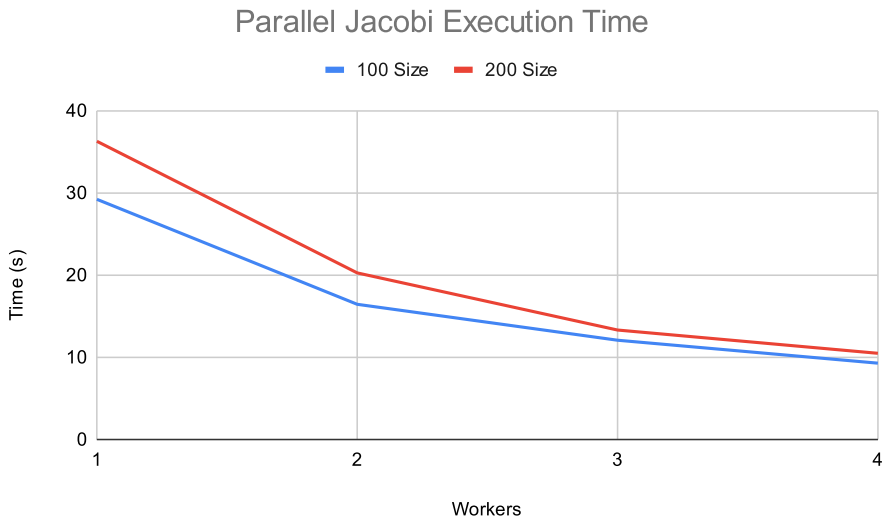

Figure 1: Execution time for various worker amounts for different problem sizes of the parallel Jacobi program

had us run tests with a size of 12 and 24. On top of this, we also had to reach a runtime of 30s by increasing the iteration amount. This combination of factors is not good for performance and not realistic in how a multigrid for this problem would be applied in real life. In an actual problem, we would have a convergence threshold rather than requiring to increase the amount of iterations to run on our coarse grid, but because we have to reach 30s runtime we need an insane 300 000 000 (300 million!) iterations. When trying to run this amount of iterations on a parallel version of multigrid, it takes *more* time, not less, to run the program, and the more workers the longer it takes. This is because the overhead of setting up the threads 300 000 000 times is a lot larger than the actual time solving the problem. As said before, since we use the Jacobi method for smoothing, we can't parallelise the iterations, only the individual iterations themselves. Parallelisation is good for large problem sizes, not small ones like a 12x12 matrix. When excluding the runtime of the smoothing operation on the coarsest grid, our parallel multigrid runs just as fast as the non-parallel one for a problem size of 12 and 24, and slightly faster for problem sizes in the hundreds, the problem is simply the amount of iterations that reaching a 30s runtime requires us to have.

This is also why we haven't ran the 3 and 4 worker variations for multigrid on small sizes, since it takes too long.

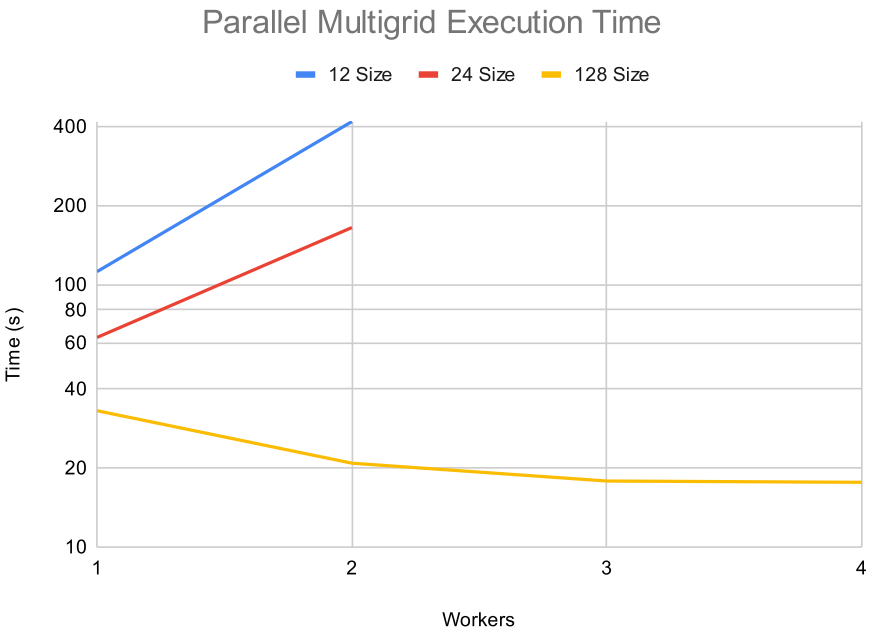

Figure 2: Execution time for various worker amounts for different problem sizes of the parallel multigrid program

5 Conclusions

5.1 Summary

Both the Jacobi and multigrid applications benefit from parallelisation. In the case of the Jacobi method, it benefits on each iteration individually. In the case of the multigrid method, it benefits in the smoothing, restriction, prolongation, and correction steps. Using a convergence threshold is beneficial to prevent unnecessary iterations. Parallelisation works best when problem sizes are substantial enough.

5.2 What has been learnt & problems faced when developing

One interesting insight to come from the multigrid solution is an elaboration is the versatility of pointers. Until this point, at least I (Victor) had not yet thought of any scenario in which having pointers to pointers would be necessary. However, when creating the array of grids, this became a necessity. To avoid becoming triple star programmers (triple pointers), we needed to use some structs to organise our data.

The multigrid method was new to us, and some parts were confusing for a while. For example, the multigrid method with a v-cycle usually has a residual step, which is then used to create the coarse grid to calculate the needed corrections for the fine grid. In the case of Laplace's heat equation on the other hand, we don't have a residual, because the solutions are the harmonic functions. This means it averages out over time, and each point in the grid gets its value from the average of its neighbours.

Another part that was confusing at first was that we at first in the correction step set the fine grid points equal to the interpolated points. This is wrong, since if we do that we would simply get a fine grid which is an interpolated coarse grid. This means a loss in information. Instead what we want to do, is add the interpolated values to the old fine grid values, to bring it towards the solution while keeping the detailed data of the fine grid intact.

False sharing and cache lines were new to us, and how to prevent it was an interesting exercise.

There are many ways to implement a 2d grid on the heap, but what was new to us was how to convert a grid allocated as a 1d array into a grid allocated as a 2d array for easier indexing, while keeping the continuous memory allocation for better spatial memory performance, rather than having each row allocated in different spots.