HPCSE (151-0116-00L), Project part 1 | Beat Hubmann | Mar 02, 2019

1 Task 1

Done; nothing to report.

2 Task 2

Done; nothing to report.

3 Task 3

Done; nothing to report.

4 Task 4

4.1 Subtasks a-d)

Do complete these subtasks all together and after checking with a TA first if allowed to do so, I temporarily rewrote `auxiliar.cpp` to accept command parameters for grid count, upcycle relaxations and downcycle relaxations. I then wrote a few lines of Python code (shown below) to perform a grid search on the parameter ranges some initial reading and manual expermenting made seem promising. When it became obvious queueing times on Euler were to long to do this on short notice, I ran the grid search locally on my personal machine with the smaller-sized problem 1. To finalize my choice of parameters, I manually timed the top ten results from the grid search on the target problem 2 and came up with `gridCount=5, downRelaxations=1, upRelaxations=7` as the tuple consistently delivering the best runtimes below 5s.

```
import subprocess
p = 1
avg = 3
timings = {}
for gc in range(2, 6):
        for ur in range(1, 6):
                for dr in range(1, 11):
                        result = {}
                        time = 0
                        for _ in range(avg):
                                output = subprocess.check_output(
                                        ['./heat2d', '-p', str(p), '-v', '-gc', str(gc),
                                         '-ur', str(ur), '-dr', str(dr)])
                                for row in output.decode('utf-8').split('\n'):
                                        if ':␣' in row:
                                                key, value = row.split(':␣')
                                                result[key.strip()] = value.strip('s')
                                time += float(result['Running␣Time'])
                        time /= avg
                        timings['gc=␣' + str(gc) + '␣ur=␣' + str(ur) +
                                '␣dr=␣' + str(dr)] = time
                        print(gc, ur, dr, time)
[print(key , '␣:␣' , value, 's') for (key, value) in sorted(timings.items() ,  key=lambda x: x[1]  ) ]
```

4.2 Subtask e)

Given an initial discretisation of a problem, translating the problem to a coarser grid alllows a 'good initial approximate guess' at the solution to be found cheaply in terms of computation (i.e. quickly in terms of time). Refining the initial guess towards the final accepted solution on the finer original grid then can be done faster compared to starting from scratch on the fine grid and approaching a solution iteratively. This basic idea can be applied recursively leading to the stacking of multiple grids and traversing between them.

4.3 Subtask f)

Based on the parameters discovered above, using multiple Jacobi steps only improved convergence speed when done in one direction. Nevertheless, more Jacobi steps at a given grid level help to smoothe out the solution at that level so that the next level won't have to do it at a possibly smaller step size.

4.4 Subtask g)

Initial experiments with a W-cycle showed significantly worse convergence times. Assuming that the problem size likely is too small to profit from such complications, further investigation was abandoned.

5 Task 5

5.1 Subtask a)

Wherever possible throughout the code, loops where interchanged so that the fastest running index (typically `j`) gets to be on the innermost loop. Loop fusion was applied wherever possible except for the `applyProlongation` routine: Measurements showed a difference of 0.1 - 0.2s while making the code almost unreadable. As the target runtime could be reached easily with plenty of margin otherwise, I decided to refrain from fusing those loops just to manually deal with all the edge cases in an ugly way.

5.2 Subtask b)

Cache access can be improved by allocating the pointers to the grid rows just after allocating the double pointers so that the data gets contiguously placed in memory.

5.3 Subtask c)

Based on experience from loop tiling applied in the course HPC1, I know that results are quite fickle and often not intuitively reproduced. Given the relatively modest problem size and the ample margin with regards to the given convergence time target, I gave up further experiments after a tile length of 8 (as a seemingly reasonable guess for dealing with `doubles`) showed no promising results. This might have been due to the fact that initial memory allocation was not aligned and thus any potential tiling benefits were eaten away by memory address jumps.

6 Task 6

6.1 Subtask a)

I replaced all divisions by multiplications. Furthermore, squared factors were changed to be precalculated by simple multiplication outside the loop.

7 Task 7

7.1 Subtask a-b)

All hints given in the vectorization report were implemented. The following loops were excluded from vectorization:

- Saving the solution: Not performance relevant as only done once.
- Allocating memory: Inner `j` depending on outer loop; only done once.
- `while` loop until convergence: Number of iterations not know; unable to optimize.

7.2 Subtask c)

My local installation of `icc` somehow didn't want to play nice with Intel intrinsics' aligned memory allocation. As no time was left for interactive development on Euler and local experiments with the functionally similar but portable `posix_memalign` didn't show any tangible benefits, implementing alignment wasn't pursued any further. In theory, aligned SIMD operations would offer better performance, however reading through Intel's documentation on the subject, current-day CPUs and compilers seem to have done away with the necessity to do this manually.