**Serial optimisations and OpenMP Report**

b.A description of your Serial optimisations and OpenMP design;

c.Comparisons of your parallel performance vs. serial performance;

d.Analysis of effectiveness of different optimisations you tried;

e.Make it clear what your best performance is for the “256x256” case;

Serial Optimisations and OpenMP design

The first step was to outline and document the functions in the program as well as testing the speed on blue crystal – 493.07s (all tests will be made on the 128x256 file). I then looked into complier optimisation flags, I tested the GCC flag ‘-O3’ which reduced the runtime to 213.11s, and also the ICC flag ‘-XSSE’ which further reduced the time to 209.43, both were a dramatic improvement.

I followed the execution trace using the gdb debug tool and began to apply some basic serial optimisations. At this point I went straight into optimising the program without having fully appreciated where the majority of the overhead lies. The following optimisations have very little effect. There were a number of lines in the code that could be trivially hoisted, these were found in accelerate\_flow and propagate. I removed the call to timestep and placed the functions in the original for loop. I removed the return value from all the function in the main for loop included the av\_velocity return, this meant instead of this function parsing a double back to the for loop, I instead filled the av\_vels array within the function, parsing it as a pointer. I also removed all the “ii \* params.nx + jj” array index calculations and stored the value in a local variable. Unfortunately all of these improvements helped very little.

I then decided to take a completely different approach to the code. I profiled the code using gprof, where it because apparent (although I already assumed) that the majority of the work occurred in the collision function.

Parallel vs serial performance

Effectiveness of optimisations

Best performance (‘256x256’)

List of optimisations attempted:

Without –O3 flag:

Elapsed time: 493.078442 (s)

With –O3 flag

Elapsed time: 213.117505 (s)

1. Hoisting code (Repeated hence wasted calculations)
   1. Collisions function – c\_sq, w0, w1, w2. However saw no time saving, compiler most likely already hoists variables at compile time.
   2. Propagate() removed two calculation out of inner for loop
   3. Combined for loop in initialise()
2. Computation reductions
   1. Removed ii \* params.nx + jj from all for loops and made it one calculation at the start.
   2. Reduced repeated commutation in collision
   3. Could make index global variable
   4. Looked into loop unrolling but then notes that –O3 flag does this
   5. Changed mult for addition where possible

Elapsed time: 213.083247 (s)

1. Compiler change icc with -XSSE optimisation flag

Elapsed time: 209.422428 (s)

* Reduced equations in collision, removing last for loop – relation step (below time)- array d\_aq

Elapsed time: 195.605971 (s)

* Combined rebound and collision

Elapsed time: 194.543988 (s)

* + Changed mult for addition where possible

Elapsed time: 194.479301 (s)

* Tried \_\_builtin\_prefetch but no speed up
* Combined accelerate flow and propagate but slowed program
* Combined av\_velocity and collision. WHY

Elapsed time: 195.290540 (s)

* Removed call to timestep made no difference
* Removed return values
* Removed returning double in av\_velocity and parsed pointer to array instead. But made no difference
* Simplified collision maths equations

Elapsed time: 109.510725 (s)

* Combining av\_velocity and collision and removed 2nd u\_x, u\_y calculation because it’s a repition of the collsison calculation

Elapsed time: 85.857503 (s)

* Moving / local density into constant about. So division only done once. Then multiply by constant

Elapsed time: 79.017142 (s)

* Removing any multiplication with decimal values and swapping with a constant. constant was a static cont = 1/36. Static const will be saved and not recalculated.

Elapsed time: 75.121300 (s)

* Removing NSPEEDS global

Elapsed time: 75.096699 (s)

* Removed other gobal file name – expect to help when in parallel

Elapsed time: 75.085925 (s)

* Removing params.nx/ny and params.omega
* Removed params in function call

Elapsed time: 73.905988 (s)

1. Cache Thrashing
   1. If data is constantly being loaded into cache and out again in the same code block. The data being accessed is too large to be stored within local cache.
   2. Cash thrashing -> optomisation of temporal locality
   3. Tried removing ‘cells[index].speeds[k]’ repetition by storing t\_speed current\_cell = cells[index] in a local variable and accessing by current\_cell.speed[k] however this produced no speed improvement. Because…. Cells array in cache so access is fast anyway
   4. In terms of serieal optomisations, Tiling will not help as the code never reuses memory accesses during a 128\*128 iteration,
2. Vectorisation
3. Changed params to pointer and parsed address between functions but made not difference
4. Combining for loops:
   1. In initialise(), there exists two sets of double for loops that iterate over the same value. Therefore combine computation in a single for loop.

Things to try:

* Allocate array memory the free memory
* For collapse(2) - opemp

Code Structure:

Main()

Initialise()

* Get values from input files
* Initialise values in array
* For nx, for ny

For(params.maxIters)

Timestep()

* Accelerate\_flow()
  + For nx
* Propagate()
  + For ny, for nx
* Rebound()
  + For ny, for nx
* Collision()
  + For ny, for nx

Av\_velocity()

* For ny, for nx

Nodes = 2 x 2.6GHz 8-core Intel E5-2670 (SandyBridge) chips (a total of 16 cores), 4GB of RAM per core (64GB total)

L1 cache = 8 x 32KB instruction, 8 x 32KB data = 256KB

L2 cache = 8 x 256 KB = 2048KB

L3 cache = 20MB

B = byte

t\_speed array = each array index is 9xDouble (8 bytes) = 72B

length(t\_speed array[16384]) = 128\*128 = 16,384

malloc(t\_speed array[16384]) = 1,179,648

size(t\_speed array[16384]) = 72\*16384 = 1,179,648KB = 1.179648GB

malloc(obstacles\_ptr [16384]) = 65536

To utilise cache:

* Each loop of inner For loop = 128 entries of array
  + 128\*72 = 9216 KB
* Utilising L3 cache = 20000KB/72 = 277.78
  + L3 cache holds 277 entries
* Utilising L2 cache = 2048KB/72 = 28.4
  + L2 cache holds 28 entries
* Utilising L1 cache = 32000/72 = 444.44
  + L3 cache holds 444 entries
* Conservative as other data will be stored in cache

Af():

Cells in cache

Obstacles in cache

Prop():

Cells in cache

Tmp pulled from memory

Rebound:

Cells in cache

Tmp cells in cache

Obstacles in cache

Collision:

Cells in cache

Obstacles in cache