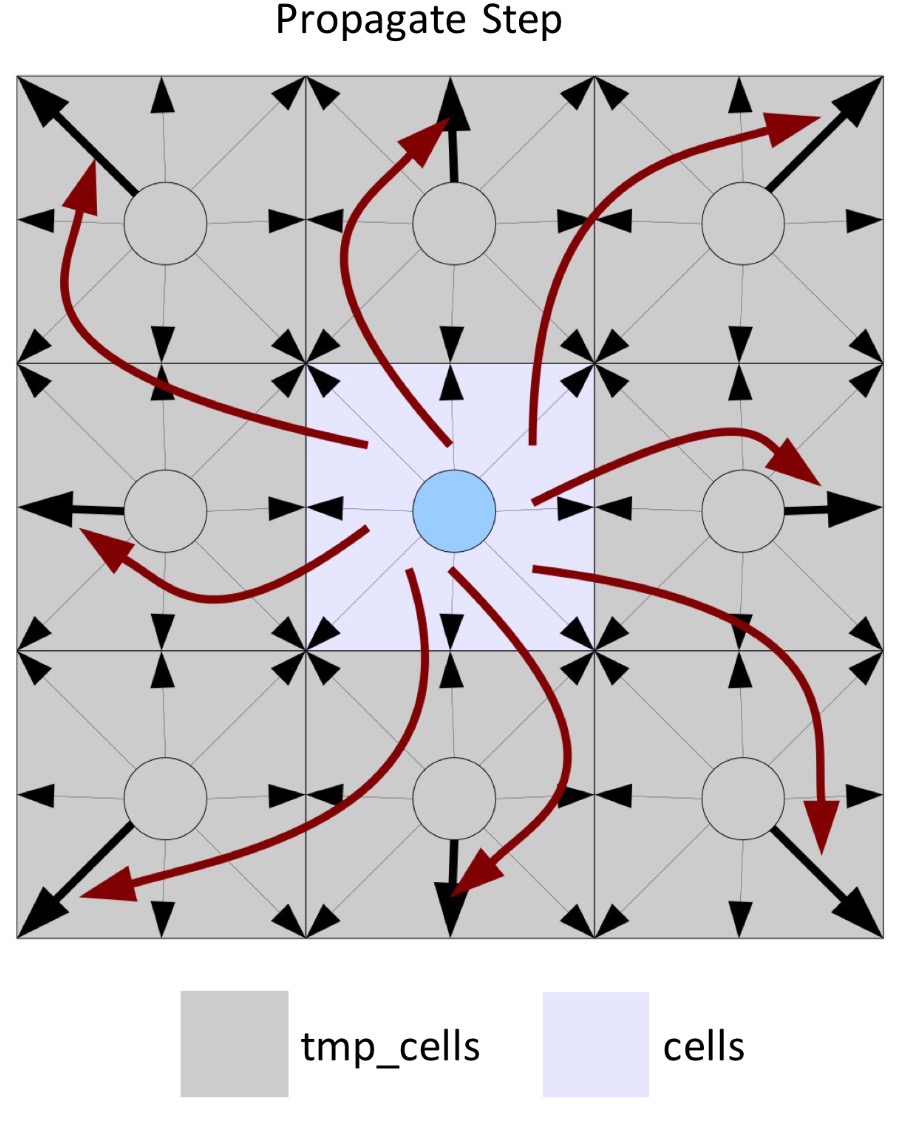
**Serial optimisations and OpenMP Report**

The report will discuss serial and parallel optimisation methods used to speed up a template Lattice Boltzman code. Serial optimisations were applied and maximised before attempting to run the program in parallel using OpenMp. The code has been designed to run on a single node on Blue Crystal Phase 3 (2 x Intel E5-2670).

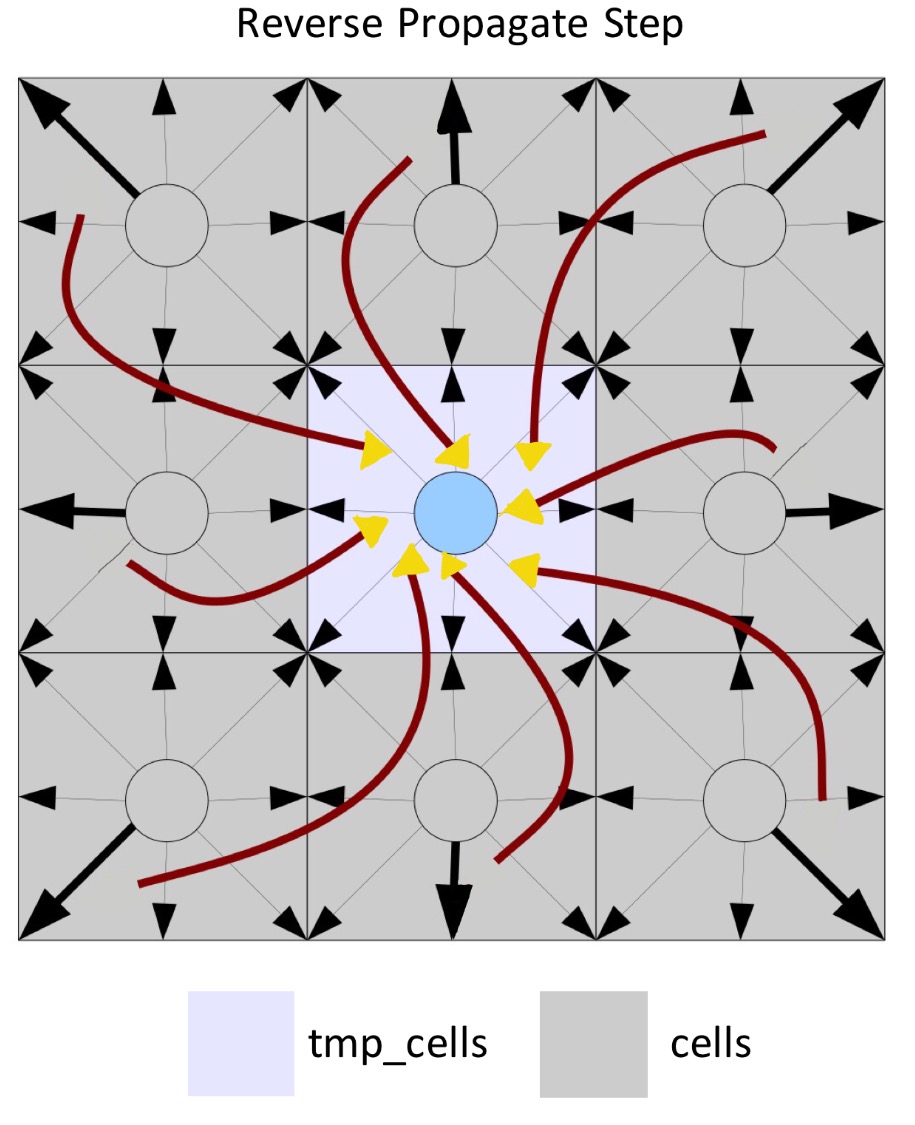
Note that all timings will refer to the 128x256 input file. This eliminated the possibility that an optimisation would only work on an n by n input type.

Serial Optimisations

The time of the program without any optimisations using only the GCC compiler with ‘-std=c99’ flag was 493s. After adding the ‘-O3’ optimisation flag, the time decreased to 213s. Comparing this to the ICC compiler with the ‘-xHOST’ flag, it ran faster again at 209s. The difference in time between the two compilers is most likely because GCC is deigned produces extremely portable code that runs on a multiple x86 architectures. Whereas the ICC compiler is specifically designed for intel CPUs and therefore will have more specific optimisations for Intel architectures. However, the optimisation flags on both compilers clearly make a very significant difference, producing around 2.3x speedup on each. Without any optimisation flags the compilers aim is to reduce the cost of compilation and to make debugging produce the expected results; turning on these flags reduces both of these objectives. It produces optimisations such as loop unrolling, common-sub expression elimination, constant propagation, vectorisation, function inclining.

The functions propagate(), rebound(), collision() and av\_velocity() are called on each iteration in ‘timestep()’ and they all repeat the same nested for loop, operating on the same data structures. As both rebound() and collision() read from *tmp\_cells* and write to the *cells*, they are only sequentially dependence. Therefore, rebound() could be moved into collision() making sure the code from rebound() precedes that in collision(). The nested for loop that was present in both function is now only iterated over once, resulting in a speed up to 207s. The same relationship occurs between collision() and av\_velocity(), they are only sequentially dependent. Therefore, the two can be combined, resulting in a speed up to 204s. In fact, they both calculate the velocity magnitude using values u\_x, u\_y and local density, however collision() calculates these values before the relaxation step and av\_velocity() after. If you only use the values calculated in collision() before the relaxation step and store these in the av\_vals array instead, the tests still pass. Effectively we include the velocity magnitude for the 1st iteration before the relaxation step (a value which was not included in the original implementation) but miss the very last iteration. This could account for the slight increase in ‘Total av\_vels difference’ which increased by 0.003\*e^-9. However, given we have significantly reduced the amount of computation necessary (now only calculating u\_x, u\_y, and local density once) and the error is negligible, this is acceptable.

The nested for loop was now only in propagate() and collision() but trying to combine the two proved challenging due to the data dependences. Propagate calculates the indexes of the surrounding squares, assigning a speed from the current cell in the cells array, to each corresponding index in the tmp\_cells array, as shown in the ‘Propagate Step’ diagram (right). To resolve this dependency, I needed to alter the data aggregation method such that the propagate function only writes to the current index (ii \* params.nx + jj) of tmp\_cells, the same as in collision(). To do this, you need to reverse the procedure as shown in the ‘Reverse Propagate Step’ diagram (right). Instead of tmp\_cells[index + shift].speed() = cells[index + shift].speed(j), we now have tmp\_cells[index].speed(i) = cells[index + shift].speed(j)

With this change only, before combining the two functions, it resulted in a speed up to 147s. **WHY**?

Moving propagate() into collision() was a trivial task and resulted in a speed up to 139s; at this point all calculations were contained within one nested for loop, with the exception of accelerate\_flow(). However, the relationship between temp cells and cells appeared to overlap. On each iteration, the code from propagate() assigns speeds from the surrounding cells in the cells array into a single index in tmp\_cells (as previously explained). Following this, the same index of tmp\_cells is used to calculate u\_x, u\_y and the relaxation step. This means that the assignment of cells to tmp\_cells in the propagate step can effectively be bypassed. Therefore we can replace every occurrence of tmp\_cells[index].speed(i) with cells[index + shift].speed(j). In each iteration, we are now only reading from cells and writing to tmp\_cells. However, this is a problem because the cells aray will never be updated. To resolve this, you can swaps the points of cells and tmp\_cells on each iteration of timestep.

This means we are only reading from cells and writing to tmp\_cells in each iteration. However, the scratch space is not being updated. To overcome this, we can swap the pointer between tmp\_cells and cells so that tmp\_cells contains the scratch space again.

Interestingly trying to combine accelerate\_flow() and collision() resulted in a slower time. As accelerate\_flow() only iterates over the top row of the grid, the number of iterations in comparison to collision() is much smaller and not computationally expensive. When combined with collision(), the conditional statement necessary inside the nested for loop most likely creates more overhead than putting the computation in a separate function that precedes collision().

The calculations for axis and diagonal speeds, contained in the d\_equ array in collision() appeared to be very repetitive. After substituting in u\_x and u\_y from the ‘directional velocity components’, (u[NSPEEDS]) and ‘velocity squared’ (u\_sq), it was clear that the equations could be reduced. The method used was to algebraically manipulate the equations by hand using pen and paper. Here is an example reduction:

before: (1.0 + u\_y / c\_sq + (u\_y \* u\_y) / (2.0 \* c\_sq \* c\_sq) - u\_x \* u\_x + u\_y \* u\_y / (2.0 \* u\_x \* u\_x + u\_y \* u\_y));

After: c\*(4 + u\_x \* 12 + (u\_x \* u\_x) \* 648 \* d1- (216 \* d1 \* (u\_x \* u\_x + u\_y \* u\_y)))

The first attempt at minimising the equations worked well, the speed reduced to 79.3s. The large reduction in speed is most likely down to the huge number of calculations the program no longer has to compute. These calculations occur in the inner most for loop and on a 128x256 input, these would be calculated 4000\*128\*256 number of times, any reduction will have a significant effect on run speed. Given this, the key to speeding up this area of the code, is to create repeated (exactly the same) calculations, so that the program only has to compute a subexpression once. Therefore, the calculations were re-written with as many repeated expression as possible the reduced form constraint. In 8 / 9 calculations in the d\_equ array, the following sub exprerssions occurred (u\_x \* u\_x), (u\_y \* u\_y), and in 4/9 (u\_x + u\_y), (u\_x - u\_y). This meant that each repeated sub expression is only calculated once on each loop iteration and the memory location of the value is referenced on every subsequent occurrence. This reduced the run time further to 65.3s.

Computational expense is approximately ordered in the following way, where the integers represent relative weights:

*(1 - Addition/Subtraction) < (4 – Multiplication) < (10 - Division/Modulo) < (50 – sqrt, pow)*

Therefore, reducing the number of expensive computation will help speed up the program. To reduce the number of divisions, a ‘static const’ was initialised outside the nested for loop in collision(). Using ‘static’ means the value is maintained between functions calls, this is useful because collision() is called 4000 times and it would seem inefficient to dynamically allocate a value which stays the same on every iteration. All divisions were replaced with multiples of the declared static const and factorised such that only one division occurs per calculation. Any small multiplications were replaced for the equivalent addition or subtraction; this resulted in a speed increase to 60.2s.

Vectorisation is essential to decreasing the number of computations performed by the CPU, given that the hardware has multiple processing elements (SIMD) this means it can perform the same operation on multiple data points simultaneously. After looking at the ICC compiler vectorisation report it stated that there were vector depedences between the two data structures cells and tmp\_cells, specifically FLOW and ANTI dependence, and so the loop could not be vectorised. However, in my implementation of the collision, this isn’t true. Data is only written to tmp\_cells but never read, and only read from cells but never written. I attempted to manually indicate to the compiler that it should ignore these dependancies, however the attempts did not work. I then changed the ICC compiler version from 15 to 16 and the compiler vectorised the loop. The speed then decreases to 46.3s. The reason for the speed up is down to the utilisation of the SIMD architecture that enables the processing of multiple data with a single instruction. The Intel E5-2670 chip set has a **AVX unit** with vector registers of 256 bit. This effectively means four double-precision floating-point values can be operated on in parallel.

**Parallel Optimisations**

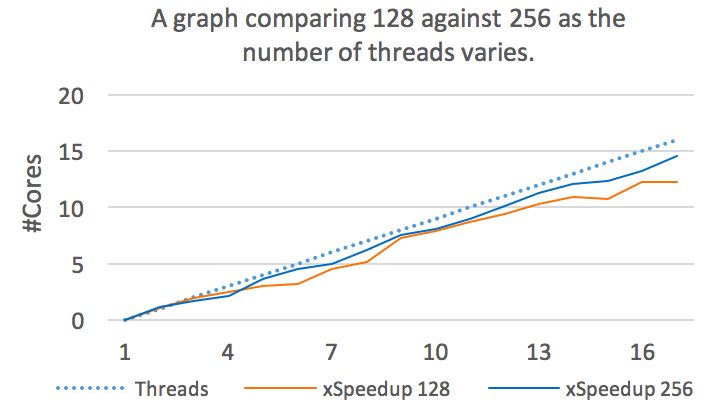
To select an appropriate location to start a parallel section, VTune was used to profile the code. As over 80% of computation is completed in the collision function, putting a “#pragma omp parallel for” around the outer for loop seemed sensible. However, this ran at 105s, about 2.4x slower than the serial equivalent. In theory, ‘parallel for’ creates a new team of threads, and assigns each team to handle different portions of the loop. However, it became apparent that the two shared variables between each thread, tot\_cells and tot\_u, created a critical region. And as such, each thread was having to acquire and release all the locks created by this region (mutex). The solution to this was to use a reduction clause, this meant a private copy for each variable is created in each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the original globally shared variable. This reduced the time to 4.213s.

It’s important to note that the variables in the reduction clause are now private to each thread and by definition the loop counters are also private. Given that everything else is by definition, shared amongst the threads, there seemed little use for either the ‘private’ or ‘shared’ clause in openMP code. Another improvement occured by specifying the num\_threads clause. As 16 is the number of cores available on a node and loop\_iterations%16 = 0 is always true on the given inputs, using the num\_threads(16) will optimise the workload distribution.

Memory and thread distribution is key to parallel programming and in OpenMP this is somewhat controlled using the scheduling clause. After experimenting with the dynamic sechduller, where the chunks are handled on a first-come first-serve basis, there was no speed up. This is because dynamic scheduling is more suited to uneven workloads and in the implementation almost every thread will have the same amount of computation. After testing a number of different chunk sizes, 4 returned the fastest time - 5.06s, but this is slower than not specifying a scheduling type.

The same test was repeated but with this time with the static scheduling clause. The best time occurred with a chunk size of 16, giving a time of 3.9s. Static scheduling divides the iteration space up into the chunk size specified. At most one chunk is distributed to each thread in a round robin fashion. Clearly this is more suited to even workloads and thus will work better with this program. As the program is running on the 128x256 input file, the outer loop completes 256 iterations. It makes sense that 16 is optimum as to find the most efficient chunk size (assuming even workload) we can divide the #max\_iters / #threads = Chunk size, 256/16 = 16. You would expect 8 to be more efficient on the 128x128 input file. However despite this, the fastest time 3.85s, came from not specifying the chunk size, this meant the iteration space was broken up into roughly equal size at run time.

Interestingly, when the chunk size was small, for both the dynamic and static scheduling, the times were much higher. This is because as the chunk size decreases, the number of times a thread needs to fetch work from the work queue increases. Thereby increasing overhead and reducing performance.

The graph shows a comparison between the 128x128 (orange line) and 256x256 (blue line) grid inputs as the the number of cores/threads varies, with the dotted line representing perfect scaling. Both the blue and orange line scale linearly, which shows that the overheads in the OpenMP implementation also scales at the same rate. This could be a good and a bad feature. On the one hand, it shows that the code is very portable and could be used on nodes with a different number of cores (if cache sizes are similar). However, it might also suggest there is potential improvement, whereby the code runs more optimally when using more cores but has more overheads when that number reduces. It also shows that cache thrashing and workload distribution are well maintained even when varying the number of cores. Both lines are very close to perfect scaling with the blue line achieving 14.5x speed up when running on 16 cores with the 256x256 input. Interestingly, the orange line is below the blue line at almost every point. As the 128 grid has 4x less computation it is more likely to diverge away from perfect scaling as it must create and destroy threads more frequently.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **128x128** | **128x256** | **256x256** |
| **Serial** | 22.98s | 46.34s | 185.41s |
| **Parallel** | 1.88s | 3.38s | 12.75s |
| **xSpeedup** | 12.2x | 13.7x | 14.5x |

3.385339 (s)

256x256

Elapsed time: 12.752264 (s)

Elapsed time: 185.410273 (s)

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)

* Cell fusion (not working)
* Elapsed time: 60.316837 (s)

* Swapped temp and cells. Tmp cells now collect from cells

Elapsed time: 66.909001 (s)

* All in 1 for loop, subbed in position of original cells into collision cals and made tmp cells the new ‘cells’. I then reassigned the pointer – cells = tmp cells in the main for loop.

Elapsed time: 56.981620 (s)

* ICC

Elapsed time: 46.345664 (s)

1. As the chunk size shrinks, the number of times a thread needs to retrieve work from the work queue increases. As a result, the overhead of going to the work queue increases, thereby reducing performance and possibly offsetting the benefits of load balancing.

Note that shared data, in my case the cells array, is read-only in the loop and so does not lead to false sharing.

Open mp:

* <http://stackoverflow.com/questions/10850155/openmp-for-schedule>
* #pragma omp parallel for

Elapsed time: 105.691866 (s)

* relaised that shared variables were causing slow down so used reduction on reduction(+:tot\_cells,tot\_u)
* the reduction clause, the compiler creates private copies of the variable sum for each thread, and when the loop completes, it adds the values together and places the result in the original variable sum.
* A private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

Elapsed time: 4.613546 (s)

* collapse(2)

Elapsed time: 5.593933 (s)

* omp parallel for **simd** reduction(+:tot\_cells,tot\_u)

Elapsed time: 4.611253 (s)

* #pragma omp parallel for simd reduction(+:tot\_cells,tot\_u) num\_threads(16)

Elapsed time: 4.600349 (s)

* schedule(dynamic)- After each iteration, the threads must stop and receive a new value of the loop variable to use for its next iteration

Elapsed time: 7.446625 (s)

* schedule(dynamic, 2)

Elapsed time: 6.107569 (s)

* schedule(dynamic,3)

Elapsed time: 5.335499 (s)

* schedule(dynamic, 4)

Elapsed time: 5.068675 (s)

* schedule(dynamic, 5)

Elapsed time: 5.817931 (s)

* schedule(dynamic, 6)

Elapsed time: 5.510437 (s)

* schedule(dynamic, 7)

Elapsed time: 6.019383 (s)

* schedule(dynamic, 16)

Elapsed time: 5.313900 (s)

* schedule(static, 4)

Elapsed time: 5.197671 (s)

* schedule(static, 2)

Elapsed time: 7.021678 (s)

* schedule(static, 16) when iteration is 256 then 256/#cores should give optimum chunk size however default distribution seems to do a better job.

Elapsed time: 4.594036 (s)

* schedule(static, 32)

Elapsed time: 8.433023 (s)

* schedule(static) - When no chunk\_size is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. Note that the size of the chunks is unspecified in this case.

Elapsed time: 4.542880 (s)

* icc compiler

Elapsed time: 3.455430 (s)

* -fast flag

Elapsed time: 3.385339 (s)

256x256

Elapsed time: 12.752264 (s)

Elapsed time: 185.410273 (s)

Info on schedulling:

1. the chunk size is too small to show any gain when divided among threads.
2. the opening and closing of a parallel region inside a loop may hurt performance.
3. As the chunk size shrinks, the number of times a thread needs to retrieve work from the work queue increases. As a result, the overhead of going to the work queue increases, thereby reducing performance and possibly offsetting the benefits of load balancing.
4. For dynamic scheduling, the chunks are handled with the first-come, first-serve scheme, and the default chunk size is 1
5. Static works well because every iteration of the loop has to do the same amount of work.

Info on private and shared:

1. By default, all the variables in a parallel region are shared, with three exceptions. First, in parallel for loops, the loop index is private
2. Second, variables that are local to the block of the parallel region are private.
3. And third, any variables listed in the private, firstprivate, lastprivate, or reduction clauses are private. The privatization is done by making a distinct copy of each of these variables for each thread.

* schedule(guided)- This scheduling policy is similar to a dynamic schedule, except that the chunk size changes as the program runs. It begins with big chunks, but then adjusts to smaller chunk sizes if the workload is imbalanced.

Elapsed time: 5.245766 (s)

* shared(cells, tmp\_cells, obstacles) not params as this is default given exist within for loop

Elapsed time: 4.604510 (s)

Vectorization not possible as not in correct for a[n] = b[n] + c[n], each iteration has calculations which rely on values contained within an array and storing in another. This makes vecotrization not possible.

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

My first step was to outline and document the functions in the program using gdb to trace the execution path step by step. as well as testing the speed on blue crystal – 493.07s. Note that all tests will be made on the 128x256 file to prevent an optimisation only working on a n by n input 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 ‘-xHOST’ which further reduced the time to 209.43, both were a dramatic improvement.

I followed the execution trace using the gdb debug tool and applied 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.