1.0 Introduction

This report looks into the optimizations implemented on a program that implements the **Lattice Boltzmann** scheme. The times mentioned in the report pertain to testing on the 128x128 parameters and obstacles files. Whenever a line number is mentioned in this report it refers to the supplied program that doesn't have any optimizations.

2.0 Compiler Optimizations

First, we need to check the speed of the un-optimized code. I have submitted the code to *Bcp3* three times and averaged the speed to **105.51s** with the compiler flag *–O3*. This was done with a single processor and single core within it. Now were we to investigate the effect of the different optimizers that *–gcc* offers, we get the results that show that without optimizations the program runs to **227.74s**. However whatever version of the *–O* optimization we use, we get the same speed that averages to **105.56s**. This also includes the *–Ofast* flag which disregards strict standards compliance. If we try to compile the program with *icc* (*-fast* flag) instead of gcc, we get a performance of **36.38s.** We are going to leave the *–fast* flag on for the rest of the coursework unless and otherwise stated. However the use of the *mktemp* on the *icc* compiler causes a security loophole[1] which is not ideal for the program. This is however an issue that does not be dealt here. The different times taken by different compilers are given in the chart to the right. While gcc shows no change in the time it takes depending on the optimizer, icc shows a significant improvement with the *–fast* flag.

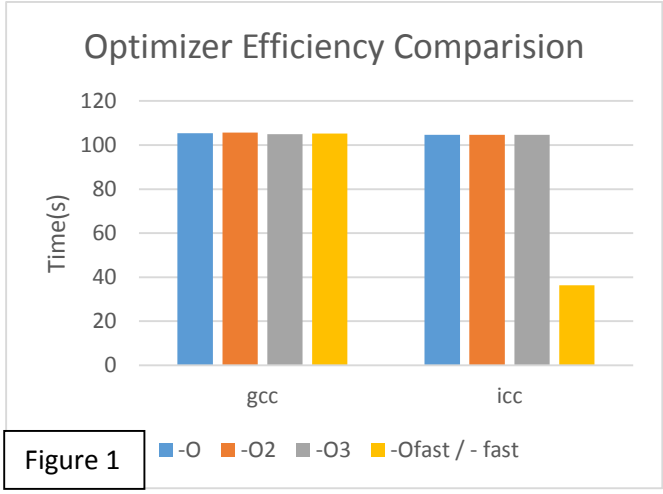


Figure 1

Adding more processors and cores did not speed the program at all, which is understandable as the code is serial and there are no threads. After the program was profiled with *Tau*, *collision, av_velocity and propagate* seem to be the functions that take up most percentage of the time (*timestep* was integrated into the *main* function to get a clearer profile). We will focus on trying to serially optimize these before we approach parallel optimizations.

3.0 Serial Optimizations

3.1 Optimizing Collision

From the first glance many parts of the function are highly inefficient and repeat the same calculation many times. For e.g. – *Line 342* onwards, the code is in the form of the equation:

$$d_{equ[n]} = w1 \times local_{density} \times \left(1.0 + \left(\frac{u[n]}{c_{sq}}\right) + \left(\frac{u[n] * u[n]}{2.0 * c\_sq * c\_sq}\right) - \left(\frac{u\_sq}{2.0 * c\_sq}\right)\right)$$

This can be optimized to (the red parts of the equation can be calculated before the n iterations of the equation):

[1] http://www.gnu.org/software/libc/manual/html_node/Temporary-Files.html

$$d_{equ[n]} = \left[w1 \times local_{density} \times \left(1.0 + \left(\frac{u[n]}{c_{sq}}\right) + \left(\frac{u[n] * u[n]}{2.0 * c\_sq * c\_sq}\right)\right)\right] - \left[w1 \times local_{density} \times \left(\frac{u\_sq}{2.0 * c\_sq}\right)\right]$$

The second optimization for collision would be the storing of the values of *tmp_cells[ii*params.nx +jj].speeds[n]* for the values of *5, 6, 7* and *8* for *n*, right before the computation of the *x* and *y velocity components* (*line 312 onwards*), as this calculation is repeated twice. This lead to a time of **105.58s** on gcc and **37.28s** on icc, thus leading to no speedup or a very small increase in the time needed. When I removed the compilers on the *Makefile*, the time was shortened to **202.37s**. This proves that even though the equations I had added do optimize the program, the *gcc* optimizers are adept at considering repeated calculations and will automatically hoist parts of the equations. Following Knuth's advice, I moved onto trying to combine the functions and thinking on a grander scale instead of looking at the calculations inside them.

3.3 Converting Division into Multiplication and other smaller optimizations

Computationally, multiplication takes lesser time than division and therefore it would be wise to convert all division operators into multiplication. While overloading an operator is possible, a much simpler approach would be to calculate the inverse of the denominator as a constant and then multiply this with the numerator within the few instances of division in the program. This is where we could definitely optimize *Collision* by calculating the inverse of *c*_sq before the iteration and calculate the inverse of *local_density* before we calculate the velocity components (We wouldn't have to divide by local density twice). The new equation is shown below:

$$d_{equ[n]} = w1 \times local_{density} \times \left(1.0 + (u[n] * inv\_c\_sq) + \left(\frac{u[n] * u[n] * inv\_c\_sq * inv\_c\_sq}{2.0}\right) - \left(\frac{u\_sq * inv\_c\_sq}{2.0}\right)\right)$$

This could also be applied to *av_velocity* as it is very similar computationally to *collision*. After this was implemented the program has a run time similar to that as encountered in section **3.2**. Therefore we rule this off as yet another optimization that is handled by *icc -fast*. However were we to compile this program with *gcc –O3*, we get a time of **46.74 s**. This meant that I had chanced onto one of the key optimizations performed by *icc –fast* that isn't performed by the *-O3* on *gcc*.

4.0 Parallel Optimizations

4.1 Parallelizing the Main *for* Loop

The *for* loop that encompasses timestep is the loop that runs for the longest within the program. However it is impossible to parallelize this as each iteration is dependent on the previous one. Therefore I moved on to focusing on the individual for loops inside of the other functions .

4.2 Parallelizing other *for* Loops

I decided to investigate the effect of having a *#pragma omp parallel* on the *for* loop inside of *collision*(which is profiled as the most time consuming function by *Tau*). After setting the number of nodes to 1 and using all 16 processors within the node. When nothing was declared as private possible overheads due to false sharing caused the program to cross the **180s** requested time. When

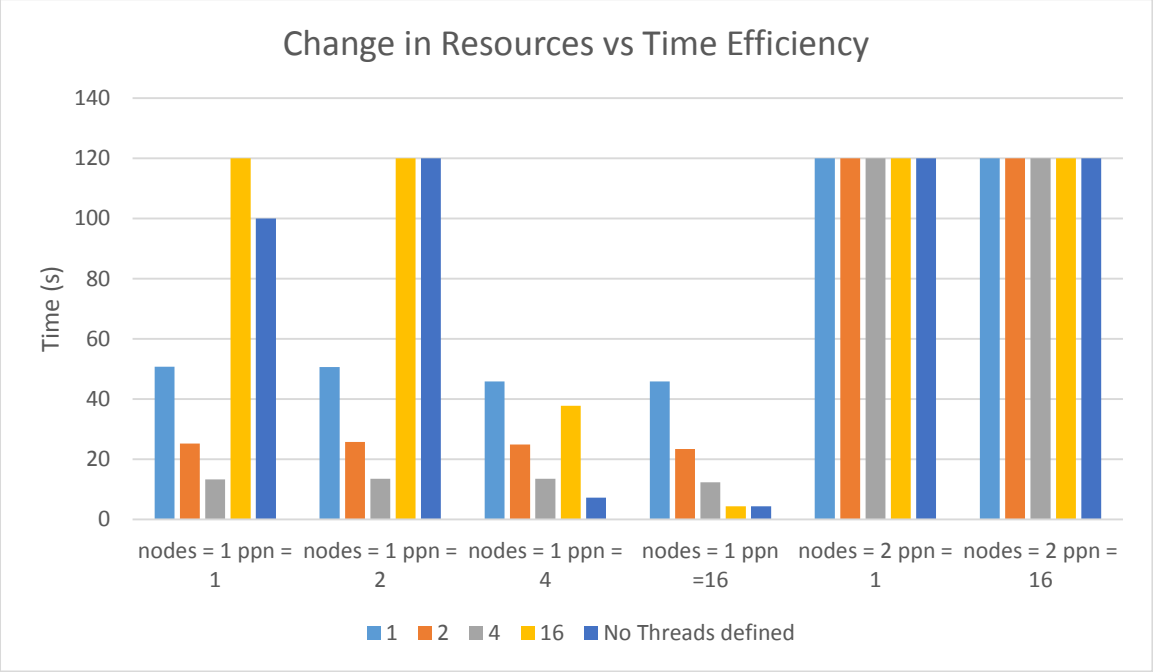
I declared most of the variables being used inside *collision* as privatized variables, a speedup of about 6 seconds was achieved. This was applied to all the functions inside *timestep* besides *av_velocity*, as reduction can be applied to *av_velocity.* This improved the time to **16.43s**. When profiled with *Tau*, it can clearly be seen that the parallel overheads such as *parallel begin/ends* and *for enter/exits* are now the sections that take the longest amount of time. Were we to make it that only one *#pragma omp parallel* encompassed all the parallelized sections of the program the time was reduced to **14.01s**.

4.3 Implementing Reduction into *av_velocity*

I didn't parallelize av_velocity with the other functions as it would be far more efficient to implement reduction with this function. Since the two variables *tot_u* and *tot_cells* are incremented at the end of the innersmost *for* loop, we can place these in reduction. This greatly improved the performance of the program to **4.21s**.

4.4 Investigating Changing Number of Threads and Varying Resources

Since the number resources being requested has always been set to one node and all the processors within it throughout the parallel optimizations, I wanted to investigate the effect of varying these resources together with the variation of the number of threads being allocated (OpenMP has been left to judge the most efficient number of threads till now). The graph below shows all of these being varied (the different colours represent the no. of threads being used):

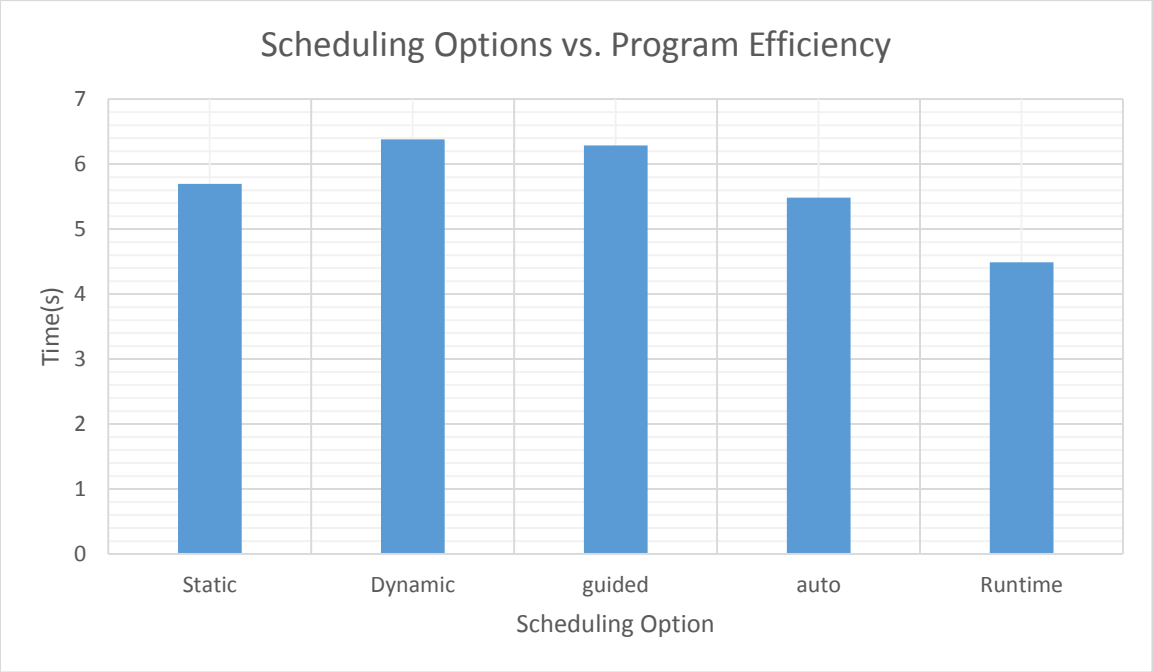


Note that the bars that actually represent **120s** have gone beyond this time and are not being considered in this chart. Since OpenMP does not have support for varying number of nodes, it only makes sense that there is no advantage to increasing the number of nodes being used (The time actually goes beyond **180s** every time this is tried). Besides the tiny variations, there isn't any noticeable difference in the speedup when the number of processors is increased till we start defining the number of threads to be four or more. The fastest time is shared by when the number of threads is not defined and when we define it as **16**. This is the maximum number of processors in a node is sixteen and therefore this is the most efficient. One anomaly noticeable is the fact that when the number of threads aren't defined and **4** processors were allocated to the program, it took

longer than when **2** when processors were allocated. This shows that *OpenMP* isn't always the most efficient when balancing the number of threads being used in respect to the resources allocated.

4.4 Investigating Scheduling

We have yet to investigate the work sharing patterns of the parallel *for* loops. The different options that could be used here were *static*, *dynamic*, *guided*, auto and *runtime*. These have been tested on *collision*. The results are shown below (I have set the *chunksize* to 10 for now to investigate the options where *chunksize* applies):



It seems that when we try to schedule a for loop, the program loses a bit of its efficiency (increasing the *chunksize* on *static* only seemed to increase the amount of time taken). The option *runtime* seems to be running at a similar time to what the program runs on without scheduling and this maybe the default option when nothing is specified.

6.0 Conclusion

Through my investigations it is very clear that while serial optimizations are very important for the program, compilers and their optimizers such as *gcc –Ofast* and *icc –fast* will do most of these optimizations for the user. On the other hand there are many possibilities with OpenMP to parallelize the loops and further optimize the program. Were the entirety of what encompasses *timestep* be merged into one *for* loop, we would further cut back on the begin/end and enter/exit overheads, to make a more efficient program. However, not ever parallelizing tool was useful and it was the case that scheduling for loops with specific a *chunksize*, made the program less efficient. While OpenMP is very good at using the resources available (as shown in Section **4.4**), the inclusion of more than one node had no benefit to the efficiency. Since this is beyond the scope of this programming language, we must look to other languages to improve these aspects of the run time efficiency. My final time on the *128x256 sandwich* case turned out to be **7.41s**.