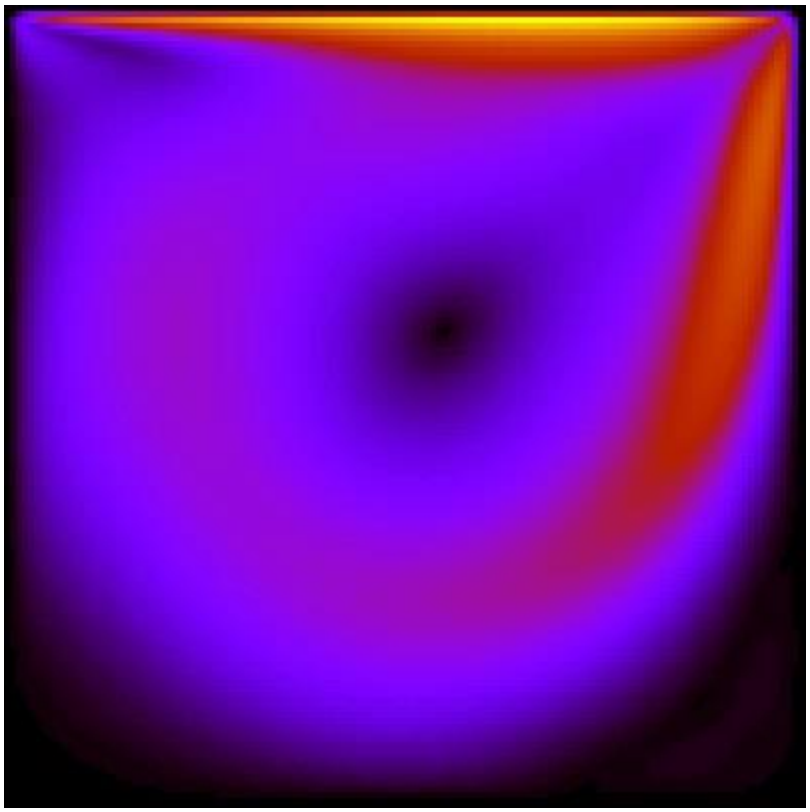
Assignment #1: Porting the lattice Boltzmann code to OpenMP



Name: Christos Mourouzi
User Id: cm16663
Canditate No.: 33747
email: cm16663@my.bristol.ac.uk

Introduction

The 2-D lattice Boltzmann model (BGK method) is currently being used by researchers all over the world for simulating complex fluid flows. This algorithm is extremely memory and computationally intensive. The aim of this project is to optimize and parallelize a C language serial code that implements the aforementioned model using OpenMP. In order to make it run as fast as possible, the code is then implemented utilizing sixteen cores in a Blue Crystal Phase 3 node.

The report begins with a description of the main serial optimizations applied to the code. Then, the parallel implementation of the code is explained and the effectiveness of other optimizations is analyzed. In the end, the time measurements of all three code versions, serial, serial-optimized and parallel are compared.

Serial Optimizations

Profiling the code

In order to find the bottlenecks of the serial code, a profiling with Intel VTune Amplifier 2015 was made. As Figure 1 shows, the functions that took most of the execution time for the 128x128 cells (105 s) were *collision() (80.55 s)*, *av_velocity() (12.7 s)* and *propagate() (11.320 s)*. For that reason, more attention was paid on improving these functions.

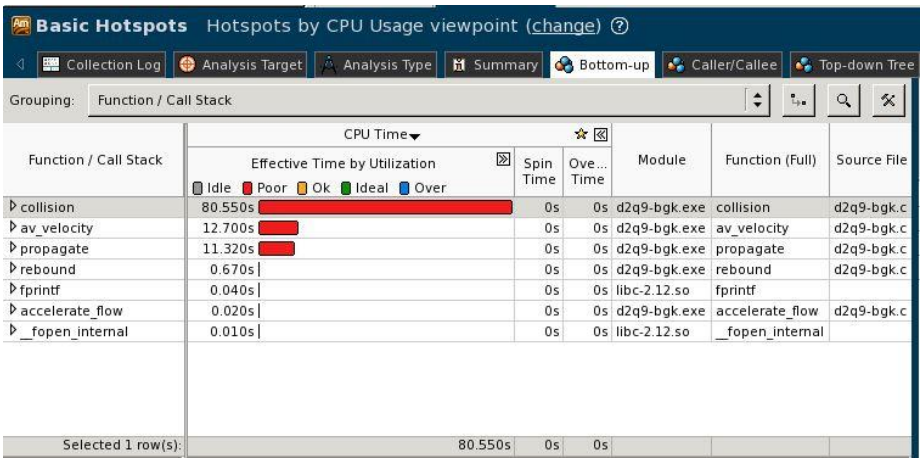


Figure 1. Profiling the code using Intel VTune Amplifier 2015

Serial code changes

- **collision():** *Av_velocity()* and *rebound()* functions were combined with the *collision()* function in order to reduce the execution time by eliminating useless iterations of the same procedure. Also, the two loops were unrolled to one main loop for better indexing of the *tmp_cells* and *cells* arrays. Observing the code, many double precision divisions were repeatedly made, especially for computing the *d_equ* array's coefficients. Dividing numbers of type double costs a lot of CPU cycles so they had to be computed just once. Additionally, using built-in functions such as *sqrt()* and *pow()* for computing mathematical equations, improved the time of the execution. At the end, the average velocity *(av_vels[tt])* of each iteration in the *main()* function is equal to the result of the *collision()* function.
- **propagate():** The only optimization in this function was to transport the calculation of the *y_n* variable and the *y_s* variable out of the inner loop into the outer one. The "if" structure costs

some extra CPU cycles and by checking the "if" conditions effectively reduces the actual execution time.

- **main():** Jumping from one function to another one also costs some extra CPU cycles while running the code. In order to avoid that kind of extra time, the *timestep()* function was removed and was then replaced by the functions that were involved in it (*accelerate_flow()*, *propagate()* and *collision()* run individually in the main loop of the program).

Compiler flags

Compiling the program is also important for the execution time of the code. In the BlueCrystal Phase 3 Supercomputer, Intel processors are included. The gcc compiler was slower than the icc compiler, so the icc one was chosen. Some optimization flags were used to improve compiling performance such as:

- -xSSE3: Generates specialized code for any Intel® processor that supports the instruction set specified by target (SSE3)[1]. Different targets (AVX, SSE4.2 etc) were investigated but the SSE3 had the best performance.
- -fast: Maximizes speed across the entire program[2] and enables many different optimization flags.

Parallel Optimizations

OpenMP is an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism[3]. It uses a thread-based parallelism model in which a master thread creates a team of threads in order to execute a specific region of a program. After the team threads complete the statements in the parallel region synchronize and terminate, only leaving the master thread active.

For the purpose of this project, OpenMP was used to parallelize the intensive loops of the *accelerate_flow()*, *propagate()* and *collision()* functions. In the *main()* function threads are created before *accelerate_flow()* is called and are synchronized after *propagate()* finishes.

A parallel region is created individually in *collision()* to eliminate race conditions between the functions. A reduction clause was used for the *tot_u* and *tot_cells* variables as their values are added by each thread in each iteration.

An attempt to convert the main loop of the program into parallel code was made avoiding the creation or destruction of teams of threads for every time step. The speedup of the execution was enormous but the results were incorrect due to the fact that the reduction clause could not be used in the *collision()* function (*tot_u*, *tot_cells* should be declared out of the loop in the *main()* function in order to be shared variables), causing race conditions and wrong calculations for the *tot_u* and *tot_cells* variables.

In the job submission file OMP_PROC_BIND environment was set to TRUE in order to "bind" the threads to processors and a proc_bind(close) clause was used to "bind" the threads close to the master thread. "Processor affinity"[3] results in better cache utilization as the threads are scheduled by the operating

[1] https://software.intel.com/sites/default/files/compiler_qrg12.pdf
[2] https://software.intel.com/en-us/node/522804
[3] https://computing.llnl.gov/tutorials/openMP/

system and always run on the same processor, without "bouncing" back and forth between different processors for each time slice.

Experiments and results

The different versions of the code were tested for grids including 128x128, 128x256 and 256x256 cells. Execution time of the program and correctness of the results were taken into consideration.

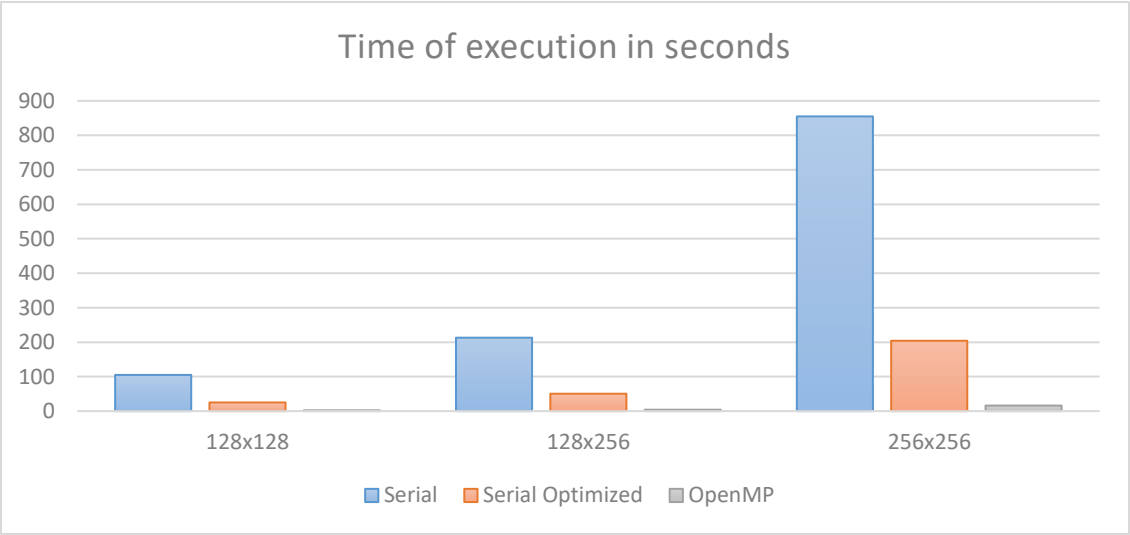


Figure 2. Time of execution of the different versions of the code

Cells	Serial (s)	Serial Optimized (s)	OpenMP (s)	Speedup
128x128	105	25.14	2.36	x10.64
128x256	213	50.46	4.11	x12.23
256x256	855	204.12	**16.11**	**x12.66**

Table 1. Time of execution of the different versions of the code with the speedup being calculated between the optimized serial code and the OpenMP code.

As the table above suggests, the optimized serial code was almost four (x4) times faster than the original one. A major speed-up of up to x12.66 times for the 256x256 cells case was achieved after using sixteen threads in an OpenMP implementation, resulting in an execution time of 16.11 seconds. The optimized serial code and the parallel code, both passed the “check results” scripts successfully with a minimal error of the class of -10^{-11} %.

Conclusion

The final results show how more effective the parallelism optimization of complex mathematic algorithms can be. OpenMP is a simple and powerful tool for achieving this kind of results by taking advantages of the combined compute power of modern multi-core computer processors.