Aidan Hood (ah16165@bristol.ac.uk) **Candidate number - 34164** **Student number - 1623238**

Advanced High performance Computing Assignment 1

1. Introduction

There were two major aims to this assignment. The first aim was to implement serial optimisations to a Lattice Boltzmann simulation program, including advanced vectorization optimisations. The second aim was to convert our optimised code to OpenMP enabled code and investigate the scalability across all cores on one Blue Crystal Phase 4 node when run with OpenMP. All times referred to in this report are timed for the smallest simulation size and were tested on node 091 for consistency. Other problem sizes had consistent times with the ballparks, except for the OpenMP implementation.

2. Serial Optimizations

The baseline time of the original code with no optimisations was 38.4s. With the serial optimizations detailed in this section of this report, times of 21.8s were achieved which beat the benchmark times.

The first optimisation to be applied was the addition of compiler flags onto the icc compiler, which resulted in a 1.2 times speedup when compared to the default gcc compiler with no optimisation flags.The flags used are as follows: -Ofast, -fast, -xHOST. The Ofast flag induces floating point, mathematics and basic vectorization optimizations, whereas the fast flag added the ipo and static compiler options which resulted in interprocedural optimization and a prevention of linking with shared libraries respectively. The xHOST flag induced the compiler to generate instructions that matched the processor used, which would have little effect for the serial optimizations, but is critical for the advanced vectorization which is discussed further in this report.

The most significant serial optimization excluding advanced vectorization was the fusioning of multiple loops. The original code has several functions which perform operations in a double loop which iterates across the whole grid. Notably, the ‘collision’ , ‘propagation’ and ‘av_velocity’ functions were the functions which took the most runtime when profiling with Gprof. These were condensed to one double loop and hence only one iteration across the whole gird, with the operations of each of the former functions being combined into one fusioned function. This saved runtime due to less memory accesses per iteration and less floating point operations per iteration, especially in the case of the ‘av_velocity’ function which had a lot of repeated operations when compared to the ‘collision’ function.

The loop fusion optimization was applied jointly with a pointer swap optimization, allowing the removal of an unnecessary copy between cells and tmp_cells. This resulted in each iteration of the algorithm to read from one array, perform calculations, and write to the other array in one step. The algorithm was then applied again, but with the cells and tmp_cells pointers swapped. This removed a significant amount of unnecessary memory accesses, hence speeding up the program. The joint effect of these optimizations was a 1.5 times speedup when compared to the unchanged code with optimization flags.

3. Vectorization

The implementation of advanced vectorization had the greatest speedup of all serial optimisation. The vectorized code runs serially for 8.8s on average, with a speedup of 2.5 times compared to the previously optimised serial code. These matched the benchmark times.

Advanced vectorization was achieved through a number of steps. These included using the xHOST compiler flag, using “restrict” to label non aliasing pointers and converting the t_speed struct to SoA format from AoS format. _mm_malloc(), __assume_aligned() and assume() were also used in order to induce data alignment onto 64 byte boundaries and then to inform the compiler so that it can vectorize the aligned data.

The replacement of conditional statements with ternary operators within the critical fusion loop, as well as manually unrolling smaller loops, was also required in order to induce the compiler to vectorize the critical loop.

4. OpenMP Implementation

A #pragma omp parallel for reduction(+:tot_u,tot_cells) was placed around the critical fusioned loop , which induced OpenMP to split the iteration space into segments which were worked upon by each thread. Alongside this, the initialisation of the cells and tmp_cells were also parallelised in exactly the same way in order to ensure each thread touches the same data in initialisation and the fusioned loop. This is in order to assign data into memory on the same core that is going to operate on the data This reduces the number of slower memory accesses from memory in other sockets - hence speeding up the program. In order to ensure that this was applied correctly, the threads were pinned via assigning the OMP_PROC_BIND environment variable to true. All together, the OpenMP implementation ran for 3.4s.

Additionally, the timings were recorded when the initialisation routine was not parellised in order to investigate the effects of having NUMA aware code and effective thread pinning. The results were that the times were approximately 1.1 times faster with the initialization being parraleised and the thread pinning

being implemented correctly. However, there was an additional benefit of the timings of the thread pinned code were much more consistent than that of the former code.

5. Scalability

The vectorised solution was only achieved in the final moments before the deadline. As a result all comparisons between the OpenMP code and serial code will be for the non vectorized versions of both implementations.

The OpenMP implementation performed considerably better than the serially optimised implementation. Compared to the serial times, the OpenMP times when applied to 28 threads were 6.3 times faster. This can be observed in Table 1 and Figure 1. Table 1 shows the average times for the OpenMP implementation scaled from 1 thread to 28 threads. This is better visualised in Figure 1 which is a graphical representation of the same results.

As shown by Figure 1, there is an initial rapid speedup as the number of threads are increased. This is expected, as both the number of floating point operations per thread decrease, as well as the number of memory accesses per thread. The graph plateaus due to this problem being a classical example of Amdahl's law which shows that no problem can be parralised to infinity, and the speedup is limited by the serial parts of the program.

6. Conclusion

The conclusions to take from the OpenMP results are evident; parallelisation is extremely effective to reduce runtime, but is limited by the serial segments of the algorithm. This is clearly shown in Figure 1. Additionally, besides basic OpenMP implementation, parallelising initialization routines and pinning threads is critical to allow the program to be 'NUMA aware' and take advantage of data being initialized on the same core as it is going to be operated on.

Serial optimization conclusions are also evident. Compiler flags did not make a significant impact on runtime. This is most likely due to the optimizations which they try and implement, such as vectorization, being too hard for the compiler to implement on it's own on a complex program such as the Lattice Boltzmann program. Additionally, fusioning the loops over the iteration space and implementing a pointer swap had a somewhat significant effect as many operations and memory accesses were saved, especially due to the high iteration count seen in this program.

However, the most interesting, and significant, conclusions can be drawn from the vectorised code results. It is interesting to see how much additional assistance the programmer has to give to the compiler, such as aligning data or adapting structs, to see proper vectorization of the critical loops. Additionally, the speedup generated serially when the code is properly vectorized is very large and interesting that it is considerably greater than any other serial optimisation.

Figure 1 - A graph showing the relationship between run time and number of threads in the OpenMP implementation

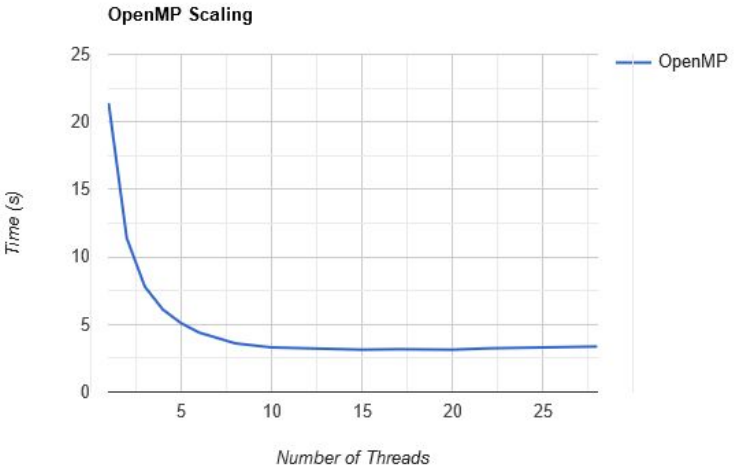

Table 1 - A table showing the results of timing the OpenMP program running on one thread, through to 28 threads

Number of Threads	Time (s)
1	21.4
2	11.4
3	7.8
4	6.1
5	5.1
6	4.4
8	3.6
10	3.3
12	3.2
15	3.1
17	3.1
20	3.1
22	3.2
25	3.3
28	3.4

Aidan Hood (ah16165@bristol.ac.uk) **Candidate number - 34164** **Student number - 1623238**