High Performance Computing

Assignment 1: OpenMP

Haohan Feng
ip23949@bristol.ac.uk

Friday 1st March, 2024

Abstract

The first assignment of the high performance computing lab is about optimising the lattice Boltzmann methods (LBM) code with OpenMP. However, some serial optimisation and vectorization before using OpenMP can also help with approaching the target time.

1 Introduction

LBM is a method used in fluid simulation, which requires a high level of computing power. As a result, we need to optimise the code to make it run faster serially and parallel. High performance computers provide fast CPUs and GPUs and large RAMs to deal with different kinds of jobs, and in this lab we use BlueCrystal Phase 4's two 14-core CPUs at 2.4 GHz and 128 GiB of RAM on one node to compute.

2 Tasks

The lab task was divided into four parts: compiler and flags, serial optimisation, vectorization and OpenMP.

2.1 compiler and flags

GCC and ICC are two popular compilers available on BlueCrystal Phase 4. With the same compiler flags *-std=c99 -Wall -Ofast -mtune=native*, ICC(2020-u4) spent much less time than GCC(10.4.0) on three scaling of grids, as is shown in the Table 1. Therefore I used ICC(2020-u4) as my compiler and -*Ofast -mtune=native -xAVX -qopenmp* as my compiler flags for future optimization.

Table 1: ICC and GCC on -Ofast

	128	256	1024
ICC	23.7s	190.0s	820.8s
GCC	32.0s	245.3s	924.81s

2.2 serial optimisation

After reading the code, It could be found that the total time is made up of two parts: Init time and Compute time. Compared to the compute time, the Init time is extremely small, enabling us to only focus on the other part. The Compute time consists iterations of timestep() and av_velocity() and timestep consists of 4 functions, accelerate_flow(), propagate(), rebound() and collision().

2.2.1 loop fusion

The five functions whose running time are counted into compute time all contains one iteration over the whole grid except accelerate_flow(), which gives us an opportunity to melt the other four functions into one. Besides, the arithmetic in collision() and av_velocity() is nearly the same. The only difference between them is that the operand of collision() is tmp_cells while av_velocity() cells. Therefore we can change the role of tmp_cells and cells when the loop begins to call av_velocity(). In this way we do not need to calculate twice in a single loop. The iteration gap now should be changed to 2 from 1.

2.2.2 pointer swap

pointer swap means removing unnecessary copying between tmp_cells[] and cells[]. In each loop, we only focus on the 9 speeds of one cell, so it is unnecessary to copy between the whole grids of cells[] and tmp_cells[]. We just need to set an array called tmp_speeds[NSPEEDS] to store the speeds temporarily. This cuts down on the loading and storing operations between memories and caches, which will save much time.

2.2.3 conclusion

As is shown in Table 2, after serial optimisation, the total time is reduced by 2.11%, 1.26%, 6.34% on 128, 256 and 1024 grids respectively. The reason why the progress was slight is that ICC has already optimised it using -Ofast with the original code. Comparing with GCC on -Ofast, the progress is more apparent.

Table 2: optimised serial(all on -Ofast)

	128	256	1024
original(GCC)	27.0s	217.2s	924.8s
original(ICC)	23.7s	190.0s	820.8s
optimised(ICC)	23.2s	187.6s	768.8s

2.3 Vectorization

Compilers vectorise the code to help the CPU implement *SIMD*. However, they need manual help since there are so many reasons for a compiler not vectorizing certain section of the code. What I do to help compilers enable efficient vectorisation are:

- adding -xAVX into the compiler flags
- adding ”restrict” and ”const” everywhere I can in timestep(), so that the compiler knows that pointers to cells[] and tmp_cells[] won't alias (overlap) in the inner loop
- Changing the data layout from array of structures (AoS) to structure of arrays (SoA) to suit vectorising the inner loop. This is the most challenging part throughout this assignment from my perspective, for it requires a comprehensive knowledge of pointers in C program.
- replacing malloc() and free() with _mm_malloc(, alignment) and _mm_free(), and adding __assume_aligned(pointer, alignment) inside routines about to use certain pointers for array access inside important inner loops. These make compiler know that data is aligned on the requested boundary like 64 bytes or 32 bytes.
- using #pragma omp simd or #pragma ivdep to force the compiler to vector(the former one is conflicted with #pragma omp parallel for reduction(), as is said in the report:
  LOOP BEGIN at d2q9-bgk.c(295,5)
  remark #15316: simd loop was not vectorized: scalar assignment in simd loop is prohibited, consider private, lastprivate or reduction clauses [d2q9-bgk.c(364,9)]
  remark #15552: loop was not vectorized with ”simd”)

As is shown in Table 3, vectorization indeed minimize the time by quite some margin. Additionally, Using 32 bytes as the alignment takes a slight advantage over 64 bytes, which may result from data's neater layout in memories and caches.

Table 3: Vectorization(all with ICC on -Ofast)

	128	256	1024
original	23.7s	190.0s	820.8s
serial+vec	9.9s	80.7s	379s

2.4 OpenMP

To enable the code running on the multi-cores, we need to add a *#pragma omp parallel for* clause before the main loop. When a thread encounters the parallel construct, a team of threads is created to execute the parallel region. And after each loop, tot_cells and tot_u are accumulated, which means the variables could be protected by reduction(+:tot_cells, tot_u) in shared memories. Besides, we need to export OMP_NUM_THREADS=28, OMP_PLACES=cores and OMP_PROC_BIND=spread in env.sh.

Table 4: OpenMP(ICC -Ofast)

	128	256	1024
original	23.7s	190.0s	820.8s
serial+vec+OMP	1.4s	12.6s	53.0s
serial+OMP	1.1s	7.4s	26.9s

As is shown in Table 4, using OpenMP based on the serial optimisation and vectorization version minimise the total running time to a higher degree. However, it is surprising that using OpenMP directly on the serial optimisation version is even quicker than adding the vectorization. Why vectorization exerts a negative influence on the OpenMP optimisation? I think it is because the vectorization is not efficient. Besides, when using schedule(dynamic), the total time, 54.4, is twice as using schedule(static) or default. I think it is because static scheduling improves locality in memory access, and also the job is workload-balanced, which suits the static scheduling best.

As is shown in Table 5, the running time gradually reduces as OpenMP threads climbs, but the rate of speed increase is not proportional to the threads increase, indicating that the more threads, the less efficiency OpenMP is.

Table 5: parallel scaling from 1 to 28 OpenMP threads

1024x1024	1	6	11	16	21	26	28
time/s	718.4	119.9	66.0	45.8	35.4	28.5	27.0
rate	1	6.0	10.9	15.7	20.3	25.2	26.6

3 Conclusion

The assignment challenges me with C program and some new knowledge, but with the help with all the resources provided, it is not very difficult. Through the lab, I learned some basic techniques to analyses code such as *gprof* and *Vtune* and mastered how to make code running on multi cores.