CFD-LAB GROUP 2 – PROJECT REPORT

MANUEL BERGLER, YE TAO, TUSHAR UPADHYAY

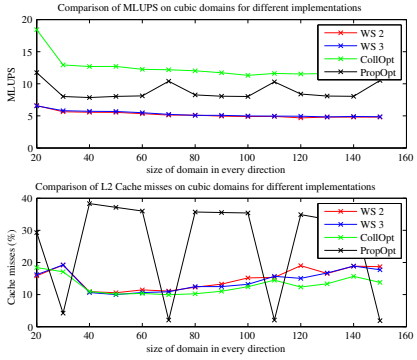
INTRODUCTION

The aim of our project was to make our simulations more efficient. Having had only a parallel efficiency of 0.5 with 4 MPI processes after worksheet 4 and approx. 5 MLUPS on a single core instead of the 8 MLUPS stated on worksheet 2 we were not satisfied with the performance of our code. After tweaking we now achieve a sequential peak performance of 18 MLUPS with a sustained speed of 12 MLUPS. On the LRZ MPP cluster, using all 64 cores, we were able to get up to 216 MLUPS. Although we mainly focused on optimizing the driven cavity scenario, our code is also able to simulate all the scenarios introduced with worksheet 3. For these the numbers only change slightly; in the sequential case we still obtain over 10 MLUPS, the parallel speed scales accordingly.

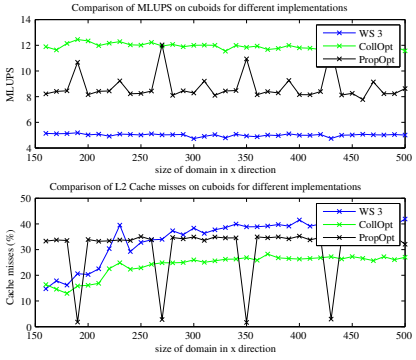
1. SEQUENTIAL OPTIMIZATION

As a first step in improving overall performance we started by optimizing the sequential code. According to [2] we changed the memory layout from the collision optimized (*CollOpt*) to the propagation optimized (*PropOpt*) layout, unrolled every **for**-loop over the 19 discretized velocity directions in order to avoid uneccessary computations – e.g. multiplications with zero components of the c_i – and dealt with streaming and collision in a single set of loops. In addition to that we implemented a vectorized version of the streaming-and-collision function using AVX2 intrinsics. Much to our suprise, the AVX version actually is slower, probably due to the lack of a sufficiently large number of AVX registers.

As shown in figure 1, the performance of the *PropOpt* code heavily depends on the behaviour of the cache, especially for larger computational domains. This is something we cannot observe for our code from assignments 2 and 3. Because of that we decided to enhance the *CollOpt* code as well. In contrast to the results of Wellein et. al. [2] it turned out that on our hardware actually the *CollOpt* code performes better that the *PropOpt* version.



(A) Simulation on cubic domains



(B) Simulation on a cuboid with 50 cells in both y and z direction each, size in x direction as indicated in the plot

FIGURE 1. Performance of our code handed in for assignments 2 and 3 (WS 2 and WS 3) in comparison to the *CollOpt* and *PropOpt* code simulating the driven cavity scenario for 1000 timesteps. Cache misses where measured using PAPI [1], simulations where run on an Intel i5-4200M mobile processor.

# MPI proc.	# oMP threads	time (s)	MLUPS	speedup	efficiency
1	1	362.64	4,736		
	2	255.56	6,720	1,42	0,71
	4	157.67	10.892	2,30	0,57
2	1	167.29	10.266	2,17	1,08
	2	114.75	14.966	3,16	0,79
	4	75.03	22.888	4,83	0,60
4	1	85.50	20.087	4,24	1,06
	2	50.74	33.846	7,15	0,89
	4	33.09	51.900	10,96	0,68
8	1	49.28	34.849	7,36	0,92
	2	32.00	53.668	11,33	0,71
16	1	31.04	55.334	11,68	0,73
32	1	15.20	112.959	23,86	0,75
64	1	7.92	216.938	45,79	0,72

TABLE 1. Scaling and parallel efficiency measured on the LRZ MPP Linux cluster with 16-way AMD Opteron nodes and Infiniband interconnect simulating on a domain of size 256^3.

2. OPENMP

Once we were done with the sequential part we tried to utilize openMP in order to avoid sending MPI messages where several cores have access to the same memory anyway. Unfortunately, the forking and joining of threads in every timestep causes a huge overhead. Intel VTune Amplifier shows that for a domain of size 100^3 and 4 openMP threads the actual computations take less than 50s of CPU time whereas the creation and synchronisation of the threads takes almost 140s. For only 2 threads on the other hand we see that the computation uses approx. 90s of CPU time and the forking only needs 25s. This results in a speedup of 1.7 and a parallel efficiency of 0.85 for 2 threads and a speedup of 1.9 and a parallel efficiency of 0.48 for 4 threads. Thus we don't recommend using more than 2 threads, if any, per process and suggest to split the domain further using additional MPI processes instead - even on NUMA-nodes. Again, these measurements where performed on the Intel i5-4200M CPU, but we see similar results on the LRZ MPP Linux cluster, cf. Table 1

3. NON-BLOCKING COMMUNICATION WITH MPI AND MPI-IO

In the final step we replaced the blocking send and receive functions by their non-blocking counterparts, which allows us to do parts of the streaming and collision while all necessary data is being transferred. Moreover, by using **MPI_Type_indexed** datatypes we no longer need to copy the data we want to send into a send buffer nor extract the data received from a read buffer. Noticing that most of our speed gains get lost by writing seperate ASCII .vtk files with every MPI process, we additionally utilized the **MPI-IO** functions to write a single binary file containing the density and velocity information for every cell in *all* timesteps we want to output. Using an additional script, this file can later be post-processed to retrieve Paraview-compatible output files. As of now we are only able to create .csv files, which are much harder to read correctly with Paraview. Unfortunately, in order for .vtk files to be visualized correctly, the point data needs to be written in the right order so we would need to sort the data in our binary file first.

REFERENCES

[1] Performance application programming interface (http://icl.cs.utk.edu/papi/). 1

[2] G. Wellein, T. Zeiser, G. Hager, and S. Donath. On the single processor performance of simple lattice boltzmann kernels. *Computers & Fluids*, 35(8–9):910 – 919, 2006. Proceedings of the First International Conference for Mesoscopic Methods in Engineering and Science. 1