ICS-4020 Programming Parallel Computers
Week 2

Kristian Hartikainen (222956)
kristian.hartikainen@aalto.fi

April 26, 2015

1 Introduction

I implemented the correlate function by using simple C code, with one additional C++ function to calculate the actual medians. The task itself is pretty straight forward, and the code is commented such that it should be sufficient as a documentation. The benchmarks for the task were run on the classroom computer 'drontti'.

The benchmarks for the naive solution (cp1) are listed on table 1. The most time consuming part of the algorithm is the multiplication of the correlation matrix with its own transpose. For ny*nx matrix, the matrix multiplication takes ny*ny*nx/2 + ny*nx/2 multiplication operations. The running time against the count of multiplications is plotted in the figure 1.

For the cp2, I used simple OpenMP pragmas to parallelize the for loops. The benchmarks for the cp2 solution running on 2, 4 and 8 cores can be found from the tables 2, 3, 4, 5 respectively. Figure 2 shows the running times against the multiplication count for 2, 4, and 8 cores. The result from cp1 is shown as a reference, with black markers.

Benchmarks for running the vectorized algorithm on 8 cores are shown in tables 6, and the running time versus multiplication count in figure 2. The results for cp2 are shown in magenta and black.

Benchmarks for the algorithm using cache blocking are shown in tables 7 and 7, for 2x2 and 3x3 block size respectively. Nanoseconds per multiplication are plotted in igures 4 and 5.

The best running time for 4000x4000 matrix was 1.782s, using 3x3 cache blocking. After 3x3 blocks the nanonseconds per multiplication started to grow, due to the lack of L1 memory. I couldn't get the compiler to unroll my for-loops automatically, so I had to implement the cache blocking (uroll the loops) manually. Thus I don't have graphs/data for larger block sizes. My in progress implementation of the non-hard-coded cache blocking can be found on my branch called 'block-sizing'.

2 Results

ny	nx	time
4000	1	0.011
4000	10	0.071
4000	100	0.699
4000	1000	6.880
4000	2000	13.588
4000	4000	27.161

Table 1: Benchmarks for the exercise cp1

ny	nx	time
4000	1	0.011
4000	10	0.070
4000	100	0.701
4000	1000	6.876
4000	2000	13.623
4000	4000	27.156

Table 2: Benchmarks for the exercise cp2, using 1 thread

ny	nx	time
4000	1	0.006
4000	10	0.036
4000	100	0.354
4000	1000	3.476
4000	2000	6.944
4000	4000	13.788

Table 3: Benchmarks for the exercise cp2, using 2 thread

ny	nx	time
4000	1	0.003
4000	10	0.019
4000	100	0.193
4000	1000	1.834
4000	2000	3.665
4000	4000	7.282

Table 4: Benchmarks for the exercise cp2, using 4 threads

ny	nx	time
4000	1	0.004
4000	10	0.017
4000	100	0.170
4000	1000	1.753
4000	2000	3.521
4000	4000	7.057

Table 5: Benchmarks for the exercise cp2, using 8 threads

ny	nx	time
4000	1	0.005
4000	10	0.010
4000	100	0.059
4000	1000	0.713
4000	2000	1.713
4000	4000	3.839

Table 6: Benchmarks for the exercise cp3, using 8 threads

ny	nx	time
4000	1	0.007
4000	10	0.010
4000	100	0.040
4000	1000	0.509
4000	2000	0.961
4000	4000	2.047

Table 7: Benchmarks for the exercise cp4, using 8 threads, and 2*2 block size

ny	nx	time
4000	1	0.009
4000	10	0.011
4000	100	0.030
4000	1000	0.309
4000	2000	0.635
4000	4000	1.782

Table 8: Benchmarks for the exercise cp4, using 8 threads, and 3*3 block size

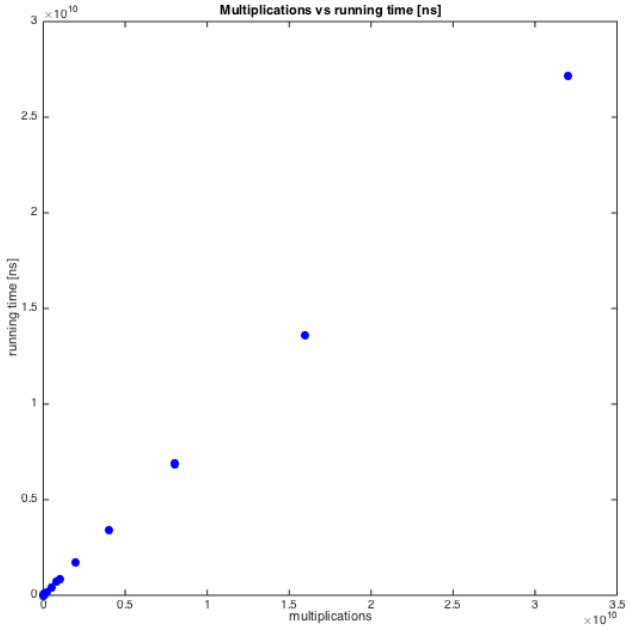

Figure 1: Multiplication per second, using naive algorithm implemented in cp1

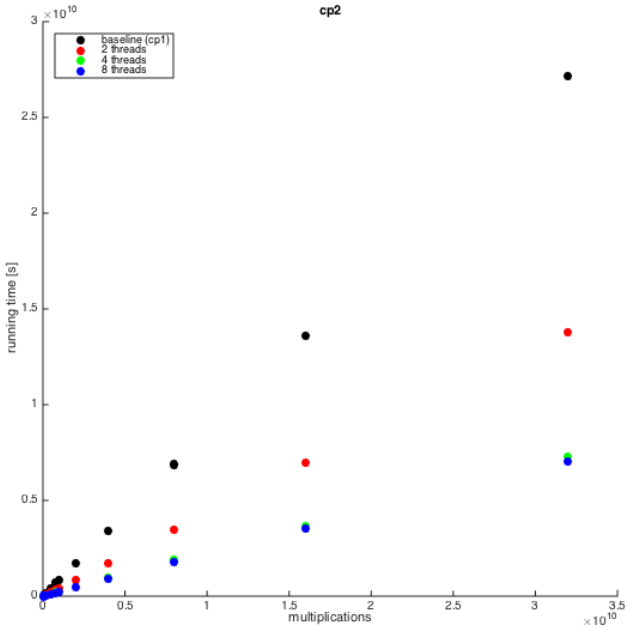

Figure 2: Multiplication per second, using multipthreaded algorithm implemented in cp2

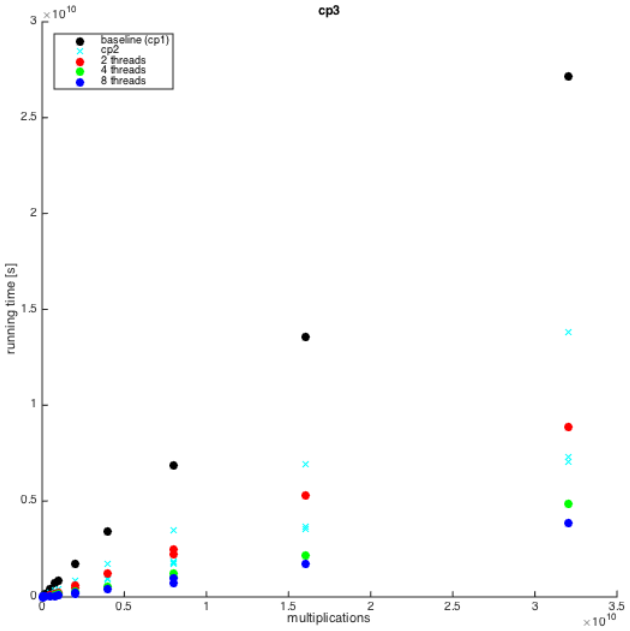

Figure 3: Multiplication per second, using vectorized algorithm implemented in cp3

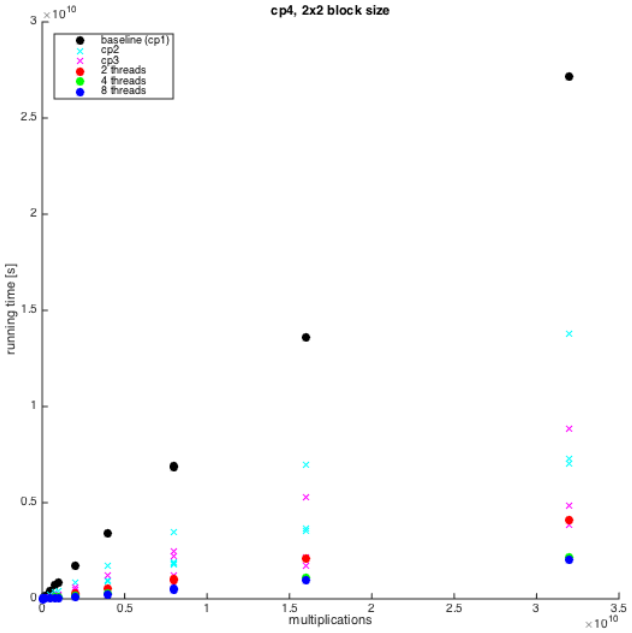

Figure 4: Multiplication per second, using cache blocking (block width 2) algorithm implemented in cp4

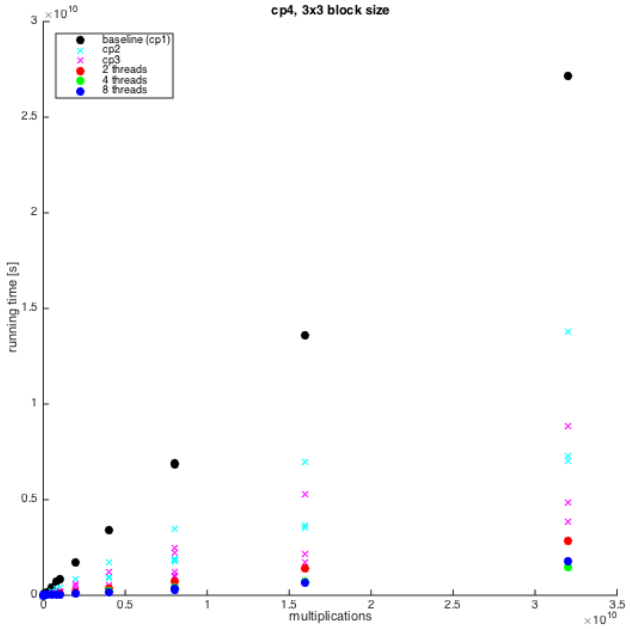

Figure 5: Multiplication per second, using cache blocking (block width 3) algorithm implemented in cp4