ICS-4020 Programming Parallel Computers
Week 3

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

May 3, 2015

1 Introduction

I implemented the image segmentation (task is1) algorithm and merge sort (task so1) using simple C/C++ code. The benchmarks for the tasks were run on the classroom computer 'drontti'.

The benchmarks for the image segmentation (cp1) are listed on tables 1, 2, 3, 4, for 1, 2, 4 and 8 threads respectively. The critical part for the running time is the calclulations done in the innermost for loops. I tried to move all the code that might slow down the execution. There's a weird looking asm("#dummy") line in the loop. That is to control the loop predictions done by the compiler. The number of divisions versus the running times are plotted in the figure 1.

For the so1, the code was quite straightforward. I implemented my own merge function, since I didn't notice the existence of std::merge at first. The benchmarks the solution running on 1, 2, 4 and 8 cores can be found from the tables 5, 6, 7, 8 respectively. Figures 2, 3, 4, 5 and 6 show the speed up for so1 algorithm versus the input size, for different types of test data.

Best running time for is1 for 400*400 image was 6.453s, and for so1 with 100000000 input size was 2.207s.

2 Results

ny	nx	time
1	1	0.000
1	10	0.000
1	100	0.000
1	200	0.000
1	400	0.001
10	1	0.000
10	10	0.000
10	100	0.002
10	200	0.010
10	400	0.028
100	1	0.000
100	10	0.003
100	100	0.113
100	200	0.392
100	400	1.547
200	1	0.000
200	10	0.005
200	100	0.402
200	200	1.571
200	400	6.217
400	1	0.000
400	10	0.019
400	100	1.616
400	200	6.337
400	400	25.027

Table 1: Benchmarks for the exercise is1, using 1 thread

ny	**nx**	**time**
1	1	0.000
1	10	0.000
1	100	0.000
1	200	0.000
1	400	0.001
10	1	0.000
10	10	0.000
10	100	0.001
10	200	0.005
10	400	0.016
100	1	0.000
100	10	0.001
100	100	0.057
100	200	0.204
100	400	0.787
200	1	0.000
200	10	0.003
200	100	0.204
200	200	0.795
200	400	3.142
400	1	0.000
400	10	0.010
400	100	0.815
400	200	3.192
400	400	12.679

Table 2: Benchmarks for the exercise is1, using 2 threads

ny	**nx**	**time**
1	1	0.000
1	10	0.000
1	100	0.000
1	200	0.000
1	400	0.001
10	1	0.000
10	10	0.000
10	100	0.001
10	200	0.003
10	400	0.010
100	1	0.000
100	10	0.001
100	100	0.033
100	200	0.107
100	400	0.421
200	1	0.000
200	10	0.002
200	100	0.108
200	200	0.423
200	400	1.669
400	1	0.000
400	10	0.005
400	100	0.432
400	200	1.691
400	400	6.684

Table 3: Benchmarks for the exercise is1, using 4 threads

ny	**nx**	**time**
1	1	0.000
1	10	0.000
1	100	0.000
1	200	0.000
1	400	0.001
10	1	0.000
10	10	0.000
10	100	0.001
10	200	0.002
10	400	0.010
100	1	0.000
100	10	0.001
100	100	0.031
100	200	0.104
100	400	0.415
200	1	0.000
200	10	0.001
200	100	0.104
200	200	0.410
200	400	1.629
400	1	0.000
400	10	0.005
400	100	0.414
400	200	1.632
400	400	6.453

Table 4: Benchmarks for the exercise is1, using 8 threads

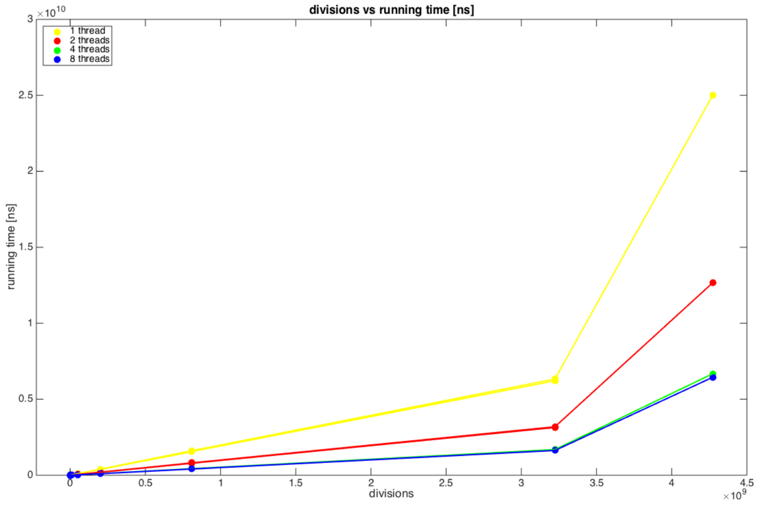

Figure 1: Divisions against running time for is1

data type	**array size**	**time**	**non-parallelized time**
0	1000000	0.061	0.060
1	1000000	0.018	0.018
2	1000000	0.011	0.011
3	1000000	0.011	0.012
4	1000000	0.009	0.010
0	10000000	0.708	0.703
1	10000000	0.209	0.209
2	10000000	0.144	0.145
3	10000000	0.151	0.157
4	10000000	0.110	0.115
0	100000000	8.012	7.958
1	100000000	2.312	2.317
2	100000000	1.733	1.737
3	100000000	1.694	1.759
4	100000000	1.213	1.271
0	200000000	16.600	16.493
1	200000000	4.859	4.874
2	200000000	3.717	3.598
3	200000000	3.505	3.665
4	200000000	2.524	2.646

Table 5: Benchmarks for the exercise so1, using 1 thread

data type	**array size**	**time**	**non-parallelized time**
0	1000000	0.034	0.059
1	1000000	0.013	0.018
2	1000000	0.008	0.011
3	1000000	0.008	0.012
4	1000000	0.007	0.010
0	10000000	0.387	0.703
1	10000000	0.151	0.209
2	10000000	0.117	0.145
3	10000000	0.113	0.157
4	10000000	0.097	0.114
0	100000000	4.350	7.956
1	100000000	1.634	2.321
2	100000000	1.366	1.767
3	100000000	1.263	1.758
4	100000000	1.092	1.274
0	200000000	8.980	16.510
1	200000000	3.408	4.874
2	200000000	2.985	3.744
3	200000000	2.603	3.655
4	200000000	2.174	2.643

Table 6: Benchmarks for the exercise so1, using 2 threads

data type	array size	time	non-parallelized time
0	1000000	0.022	0.060
1	1000000	0.011	0.019
2	1000000	0.010	0.011
3	1000000	0.009	0.012
4	1000000	0.008	0.010
0	10000000	0.249	0.703
1	10000000	0.131	0.210
2	10000000	0.114	0.145
3	10000000	0.105	0.157
4	10000000	0.100	0.115
0	100000000	2.711	7.954
1	100000000	1.505	2.316
2	100000000	1.339	1.754
3	100000000	1.130	1.755
4	100000000	1.057	1.269
0	200000000	5.586	16.492
1	200000000	3.088	4.871
2	200000000	2.773	3.800
3	200000000	2.350	3.660
4	200000000	2.180	2.646

Table 7: Benchmarks for the exercise so1, using 4 threads

data type	**array size**	**time**	**non-parallelized time**
0	1000000	0.019	0.060
1	1000000	0.024	0.020
2	1000000	0.009	0.011
3	1000000	0.009	0.013
4	1000000	0.013	0.010
0	10000000	0.202	0.703
1	10000000	0.158	0.209
2	10000000	0.125	0.145
3	10000000	0.118	0.157
4	10000000	0.123	0.115
0	100000000	2.207	7.956
1	100000000	1.673	2.343
2	100000000	1.501	1.856
3	100000000	1.232	1.761
4	100000000	1.208	1.273
0	200000000	4.512	16.495
1	200000000	3.435	4.871
2	200000000	3.088	3.752
3	200000000	2.440	3.658
4	200000000	2.427	2.648

Table 8: Benchmarks for the exercise so1, using 8 threads

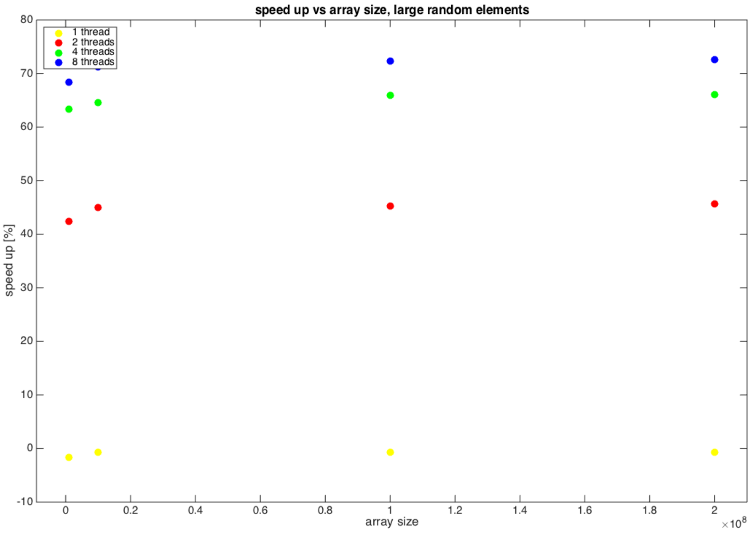

Figure 2: Speed up of so1 vs the input size of the array. Large randomly ordered values.

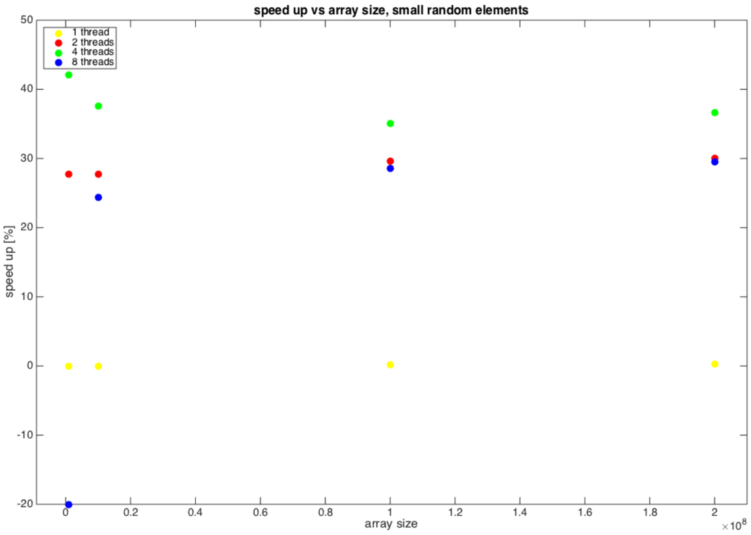

Figure 3: Speed up of so1 vs the input size of the array. Small randomly ordered values.

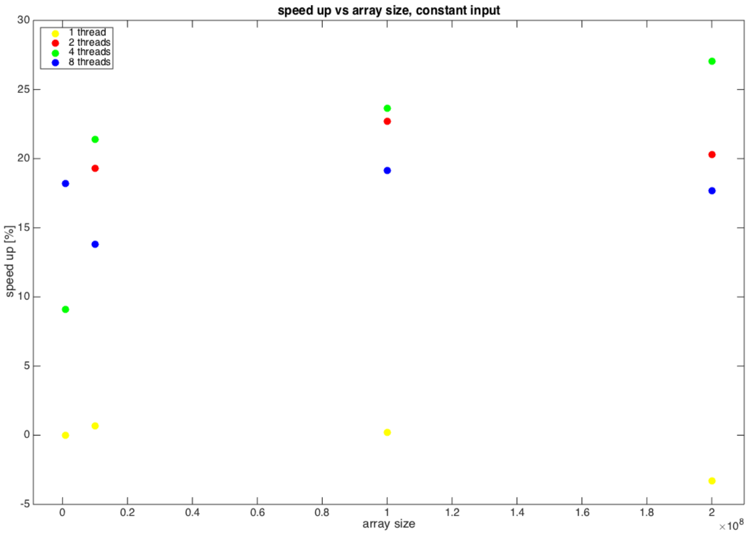

Figure 4: Speed up of so1 vs the input size of the array. Constant input.

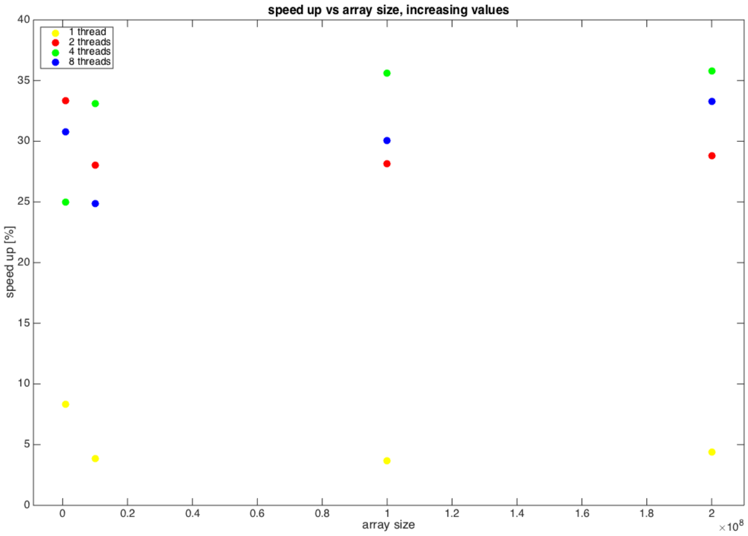

Figure 5: Speed up of so1 vs the input size of the array. Increasing values.

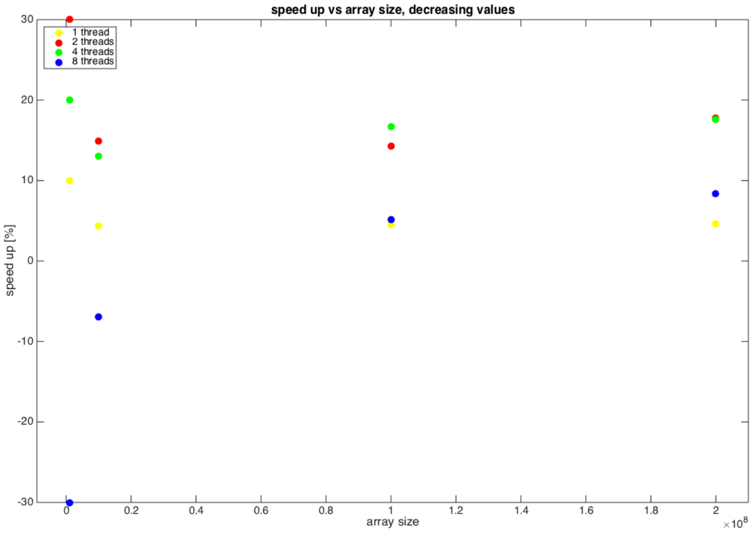

Figure 6: Speed up of so1 vs the input size of the array. Decreasing values.