Team: Tobias Hosp, Marcel Huber, Thomas Klotz

Exercise 1

- Speedup of a parallel program is defined as:

$$S(p) = \frac{Execution\ time\ using\ one\ processor}{Execution\ time\ using\ p\ processors} = \frac{t_s}{t_p}$$

- Amdahl´s law is formally define as:

$$S(p) = \frac{t_s}{ft_s + (1-f)t_s/p} = \frac{1}{f + (1-f)/p}$$

$$\begin{gathered} with \\ t_s \ldots serial\ execution\ time \\ f \ldots fraction\ of\ the\ non\ parallelizable\ part\ of\ the\ program \\ p \ldots number\ of\ processors \end{gathered}$$

Amdahl´s law describes the maximum speedup a program can experience dependent on the proportion of non parallelizable code.

This is significant because it tells us when further speedup is impossible and thus prevents us from wasting resources.

- Theoretical speedup for 10% unparallelizable code, 6 cores:

$$S(6) = \frac{1}{0.1 + \frac{(1-0.1)}{6}} = \frac{1}{0.1 + \frac{0.9}{6}} = 4$$

and for infinite cores:

$$S(\infty) = \lim_{p \to \infty} \frac{1}{0.1 + \frac{0.9}{p}} = \frac{1}{0.1 + 0} = 10$$

- Theoretical speedup for 20% unparallelizable code, 6 cores:

$$S(6) = \frac{1}{0.2 + \frac{(1-0.2)}{6}} = \frac{1}{0.2 + \frac{0.8}{6}} = 3$$

and for infinite cores:

$$S(\infty) = \lim_{p \to \infty} \frac{1}{0.2 + \frac{0.8}{p}} = \frac{1}{0.2 + 0} = 5$$

- Speedup of 10 can be archived with:

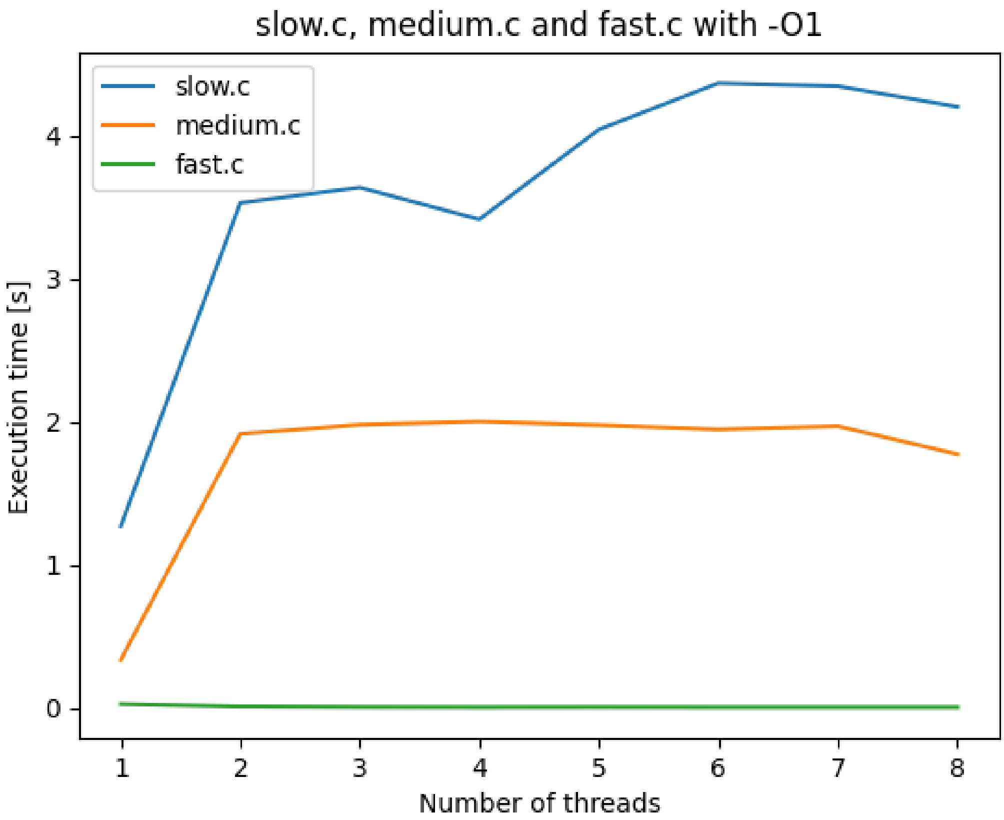
$$f = \frac{\frac{p}{S} - 1}{p - 1} = \frac{\frac{64}{10} - 1}{64 - 1} \approx 8,57\%$$

Exercise 2

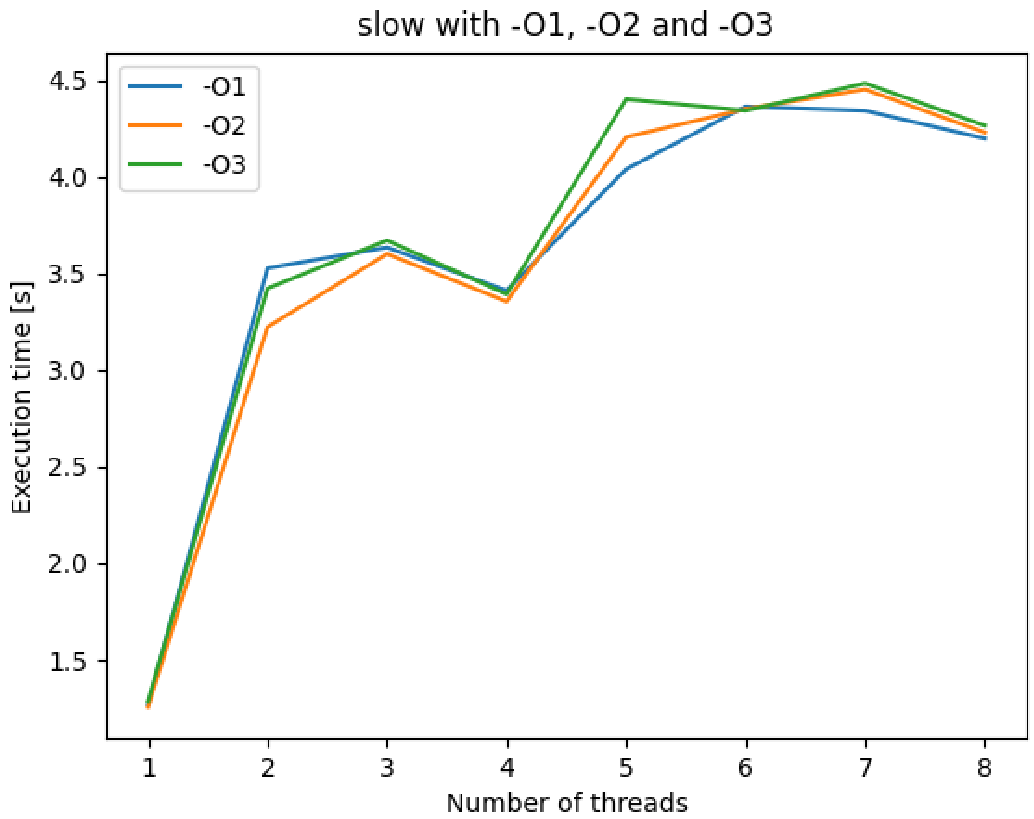
All programs were executed with number of threads from 1 to 8 and with three different optimization flags (O1, O2, O3). Compiling, execution and plotting was done by a self-made python script. We used gcc version 10.2.0.

Plots

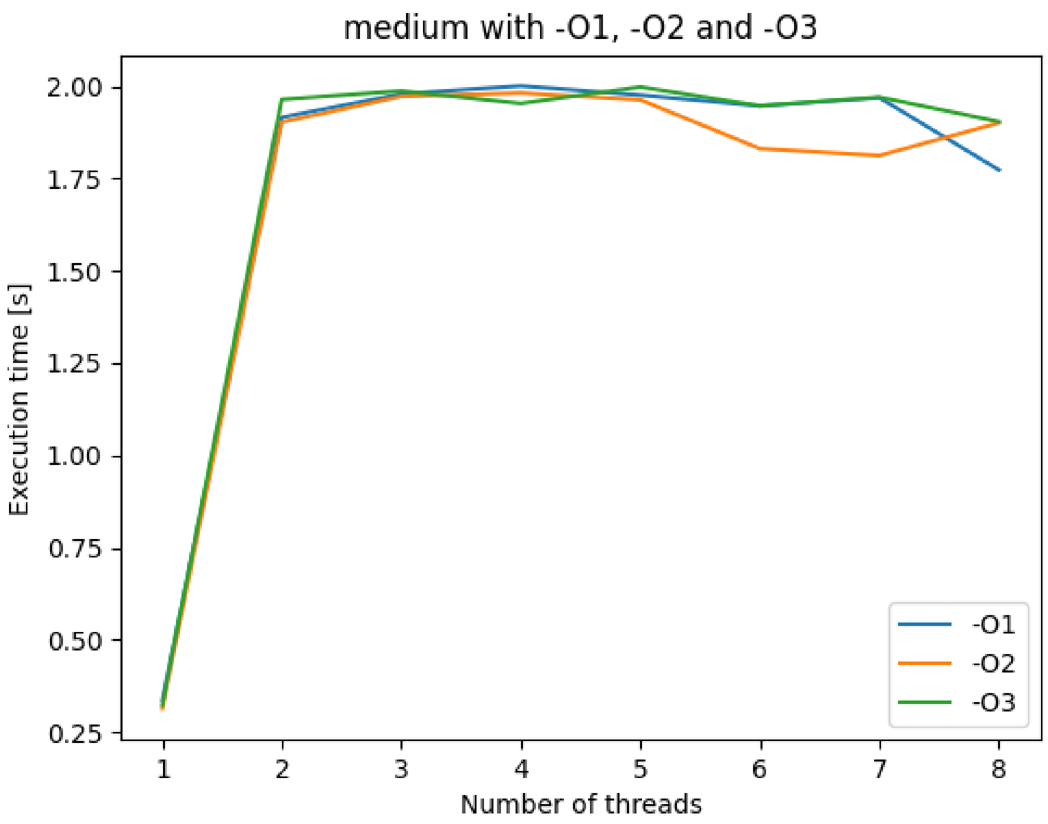
- Plot for comparison between the execution time of the 3 programs with optimization flag O1



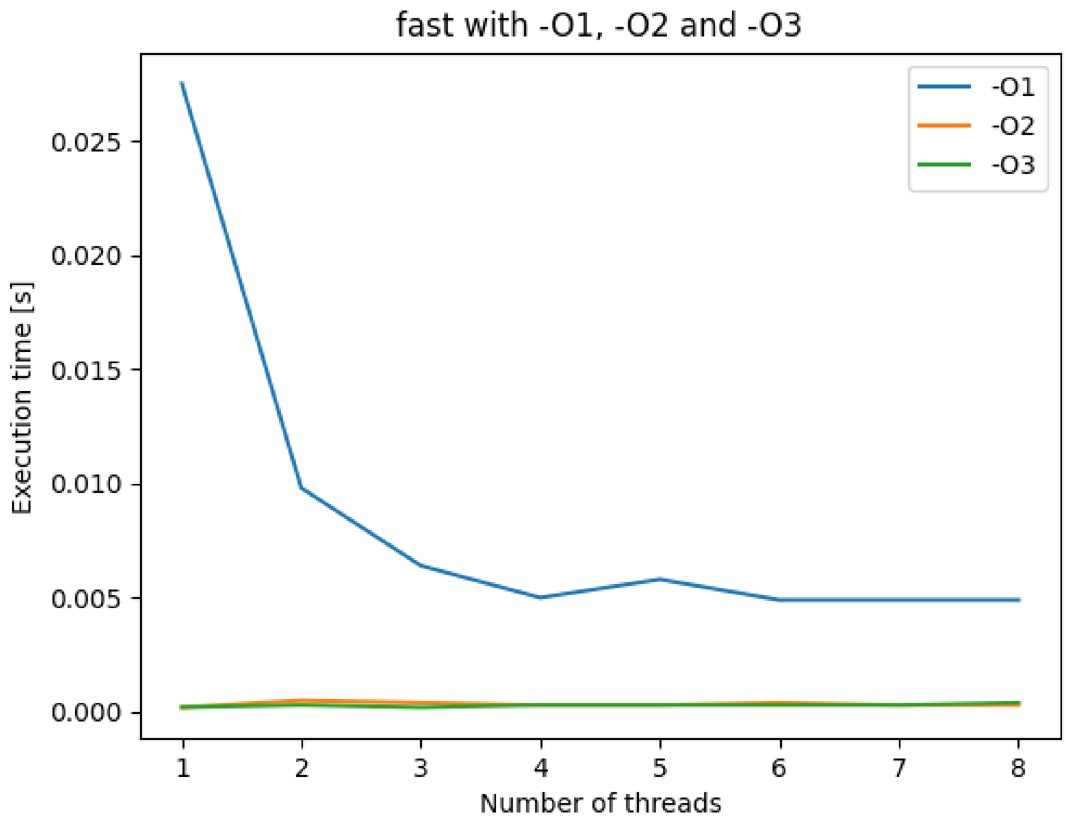
- Plot for the slow variant with all three optimization flags



- Plot for the medium variant with all three optimization flags



- Plot for the fast variant with all three optimization flags



Data Tables

- Data for all three variants with O1

num_threads	slow.c	medium.c	fast.c
1	1.2696	0.3361	0.0275
2	3.5286	1.9153	0.0098
3	3.6359	1.9788	0.0064
4	3.4131	2.0007	0.005
5	4.0409	1.9758	0.0058
6	4.3649	1.9465	0.0049
7	4.3439	1.9687	0.0049
8	4.2	1.773	0.0049

- Data for slow.c

num_threads	slow.c -O1	slow.c -O2	slow.c -O3
1	1.2696	1.256	1.2864
2	3.5286	3.2237	3.4237
3	3.6359	3.6026	3.6728
4	3.4131	3.3567	3.3943
5	4.0409	4.207	4.4029
6	4.3649	4.3525	4.3442
7	4.3439	4.4537	4.4854
8	4.2	4.2314	4.2671

- Data for medium.c

num_threads	medium.c -O1	medium.c -O2	medium.c -O3
1	0.3361	0.314	0.3234
2	1.9153	1.9025	1.9642
3	1.9788	1.9722	1.987
4	2.0007	1.9815	1.9532
5	1.9758	1.963	1.9977
6	1.9465	1.8307	1.9469
7	1.9687	1.8119	1.9702
8	1.773	1.9005	1.9039

- Data for fast.c

num_threads	fast.c -O1	fast.c -O2	fast.c -O3
1	0.0275	0.0002	0.0002
2	0.0098	0.0005	0.0003
3	0.0064	0.0004	0.0002
4	0.005	0.0003	0.0003
5	0.0058	0.0003	0.0003
6	0.0049	0.0004	0.0003
7	0.0049	0.0003	0.0003
8	0.0049	0.0003	0.0004

Conclusion

- The slow variant gets slower and slower if you execute it with more threads. We assume that this might happen, because the programs synchronisation is very inefficient, so it actually gets slower due to thread-blocking, etc.
- The medium variant runs the fastest with one thread, if the number of threads is increased the execution time raises too. If you use 2 or more threads the execution time is sligthly higher but stays constant.
- Generally speaking fast.c gets faster with more threads.
- It looks like the optimization isn't really useful for slow.c and medium.c. For fast.c -O2 and -O3 have a significant impact on performance.
- Except of some negligble peaks the measured values do not differ much during multiple executions.

LCC2-Measurements

Everything compiled with -O3.

num_threads	slow	medium	fast
1	2.0240	0.7119	0.0
8	58.3543	3.0898	0.0047