Assignment 1

(02158) Concurrent Programming

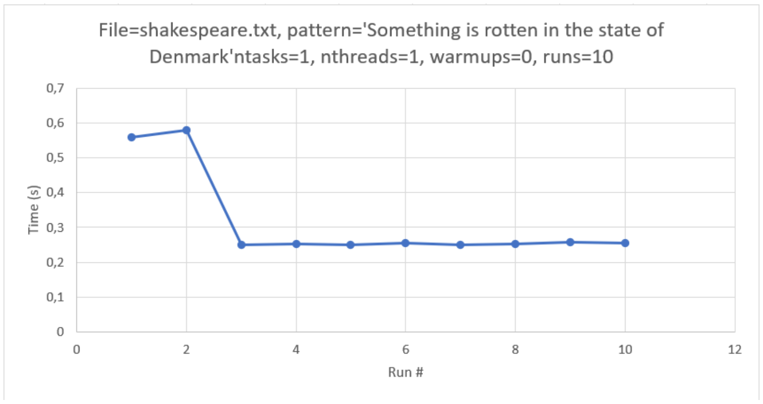
Anton Lage
s191270

Daniel Schütt
s194592

September 20, 2022

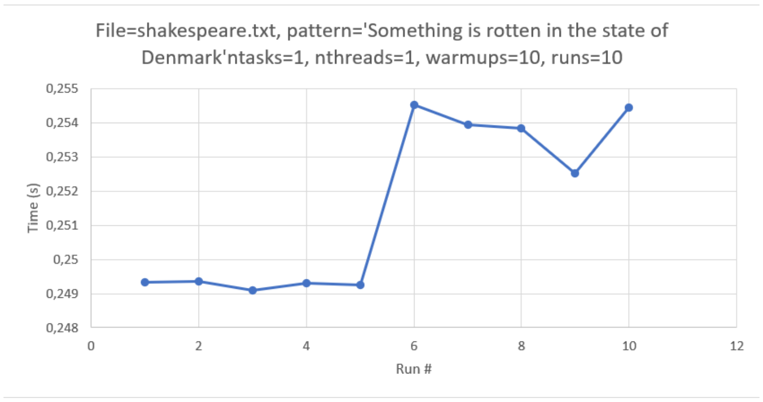
Problem 1

The entirety of this report's results was tested on an Intel Core i7-8750H CPU @ 2.20 GHz (6 cores, 12 threads) . Firstly the program was run with the following parameters:



It is clearly visible that the run time drastically decreases after the first couple of runs. Other than the drastic decrease, the time differences are not that large.

If we instead run the program with the same parameters, albeit with warmup rounds, then the following graph is our result:



It is worth to note that the amount of warmup rounds required is dependent on the CPU of the machine which runs the program. The reason why warmup rounds increase the speed of the results is due to the CPU caching relevant information appearing multiple times, thus speeding up the search.

Problem 2

In this problem we've modified the program to submit multiple tasks to the taskpool. Before executing the program, we need to divide the search up between the different tasks, to ensure that none of the text is searched multiple times.

character arrays

To ensure that the tasks aren't searching the same text, we've divided the tasks as following: Each of the tasks get a part of the text to search. task N gets the following section of text to search

From character number $\frac{len}{tasks} * N - (patternLength - 1)$

to character number $\frac{len}{tasks} * (N + 1) - (patternLength - 1) + rest$

len is the length of the entire text.

tasks is the total number of tasks.

N is the current task.

patternLength is the length of the pattern we are searching for.

rest is the result of the following calculationg: $len \bmod ntasks$

$\frac{len}{tasks} * N$ is the tasks starting character. on every task which isn't the first, we need to look back to ensure that we do not miss a pattern, due to splitting the text in the middle of a pattern. An example of this is the text "ABCDEF" where the search pattern is "CDE". This would create 2 tasks. Task 1 would search "ABC", task 2 searches "DEF". Due to the offset, task 2 will start before "DEF", which makes the task find the pattern, which would otherwise not be found.

For the end character, we add the *rest* variable. Rest is required in the case of a text which has an uneven amount of characters. If we have a text of 10 characters and 4 tasks, then each task cannot search 2.5 characters, and will thus search 2 each. This leaves 2 characters at the end of the text which isn't searched. *rest* ensures that the last 2 characters will be searched.

We hereby split the work to be done almost evenly between the tasks. The only exception is the last task, which will in some cases do more work. This could be ammended by spreading the *rest* out to equally as many tasks.

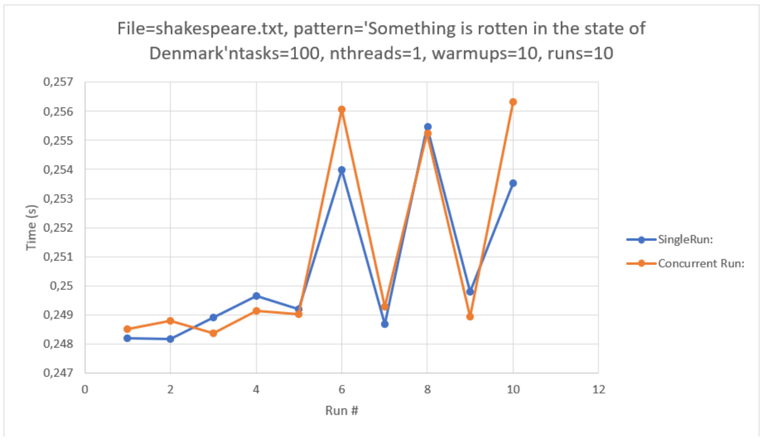
The Speedup Notion

The speedup notion is equal to $\frac{averagesingletasktime}{averagemultitasktime}$. Thus it calculates how much faster or slower the multitasking is. If the speedup is less than 1, singletasks are faster. If the speedup is greater than 1, then multitasking is faster.

If the speedup is equal to the number of threads (or tasks), its called linear speedup. However, this is rarely the case, since parrelization of programs usually introduce some overhead, like memory allocation for new threads, and contexts switching.

We've found that there is little to no difference between using multiple tasks or using a single task. This is due to the program still running on a single thread, and thus the maschine cannot work concurrently.

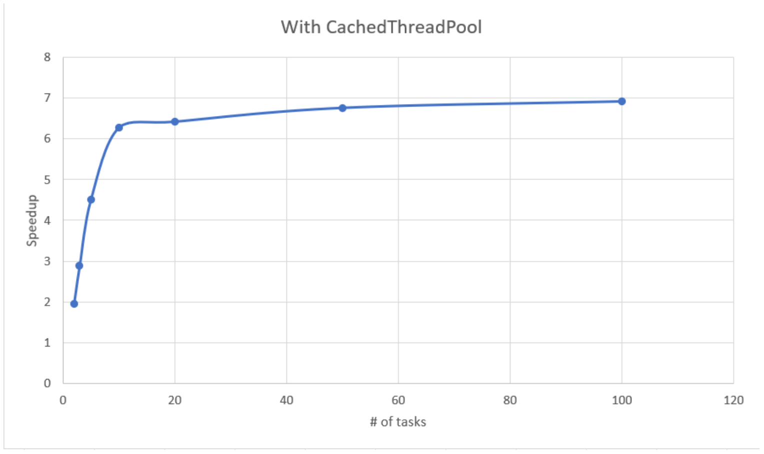
It can be seen in the following graph



We've found there is an upper limit on the amount of tasks where the speedup is about 1. If the program creates a large number of tasks, such as $>$ 10000 we see a slowdown. The creation of tasks takes a certain amout of time, and the retrieval of results takes a certain amount of time. So if we have many tasks, there will be a slowdown.

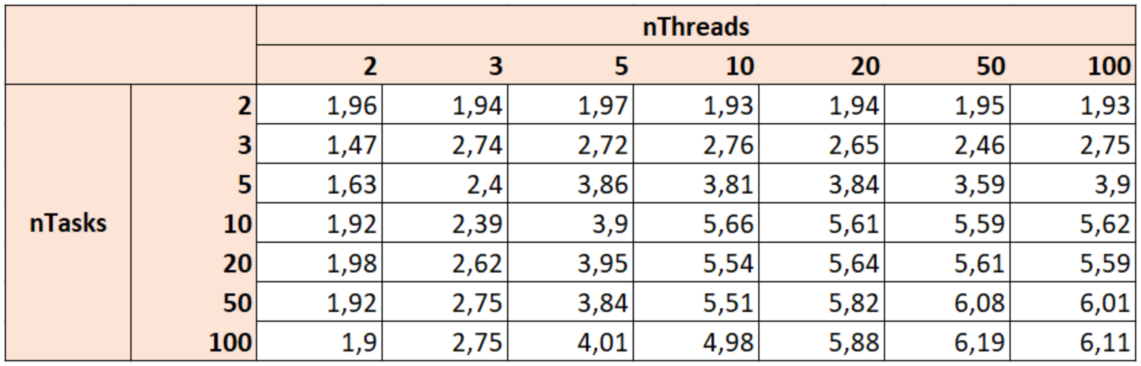
Problem 3

When using a cached thread pool, the executor generates one thread per task. This is quickly becomes wastefull, since the CPU can only handle a finite number of threads at a time. Therefore, we see a noticable speedup when increasing the amount of tasks (i.e threads) from one to two to three etc, but when we start hitting the limit of the CPU's concurrency capability, the speedup levels off, and we struggle to get a speedup greater than 7, no matter how many tasks we use. At some point, it even start to decline, since too many task introduce too much overhead.



Problem 4

When using a fixed thread pool, we do not see a difference in perfomance compared to using a cached thread pool. This is because when we start more threads than the CPU can handle at a time, surplus threads have to wait anyway. However, there are benefits in using a fixed thread pool. The amount of resources a system have to allocate is less for say 10 threads than 100, and also less for 100 task than 1000 threads. Therefore, we can optimize for both speed and resource usage by allocating threads equal to the amount of threads the CPU can handle at a time (12 for the system we use), and then splitting the job into many more tasks. For this use case however, there is no real benefit in using many more tasks, but it is easy to image usecases for which it would be beneficial. Handling request for a server for example, where each task is not directly related, as it is here.



		nThreads						
		2	**3**	**5**	**10**	**20**	**50**	**100**
nTasks	**2**	1,96	1,94	1,97	1,93	1,94	1,95	1,93
	3	1,47	2,74	2,72	2,76	2,65	2,46	2,75
	5	1,63	2,4	3,86	3,81	3,84	3,59	3,9
	10	1,92	2,39	3,9	5,66	5,61	5,59	5,62
	20	1,98	2,62	3,95	5,54	5,64	5,61	5,59
	50	1,92	2,75	3,84	5,51	5,82	6,08	6,01
	100	1,9	2,75	4,01	4,98	5,88	6,19	6,11

Figure 1: Relationship between number of tasks and number of threads

The table displays the speedup of the program in regards to the number of threads and tasks. If we have a greater number of tasks than threads, then the speedup doesn't increase, as the program cannot run all the tasks concurrently. If we have a larger amount of threads than tasks, or equally many threads and tasks, each task should then be occupied by a thread, ensuring maximum efficiency. There is a limit to the speedup, as the CPU only has a limited amount of threads. This means that at some point, more threads merely equal more overhead.