Valentin Gegoux

Multicore Computing
Project #1, Problem 1

I/ Introduction

This document aims to observe and compare the performance of three programs that implement the same algorithm but use different multithreading strategies. These programs compute the number of prime numbers between 1 and an *end number* specified by the user.

II/ How to use

Compile and run the program by typing one of the following command lines within the `problem1/` directory of the project:

```
$ java pc_static_block.java <number of cores> <end number>
$ java pc_static_cyclic.java <number of cores> <end number>
$ java pc_dynamic.java <number of cores> <end number>
```

III/ Testing environment

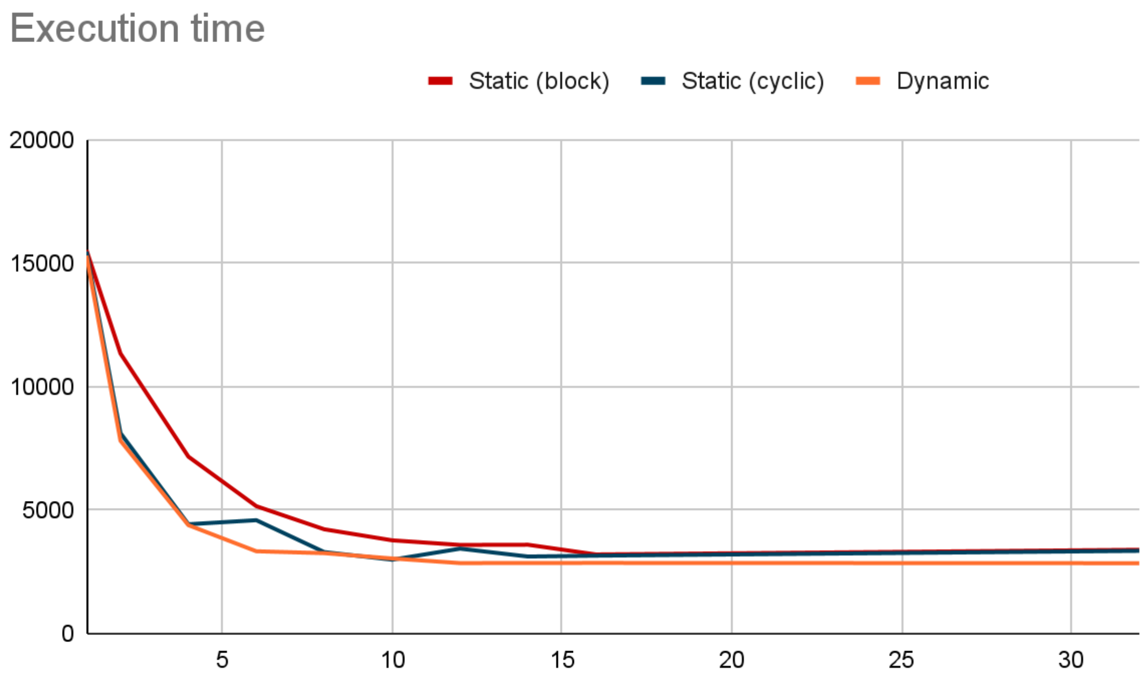
RAM: 18 Go
CPU Type: 11-core Apple M3 Pro (6 performance + 5 efficiency)
Hyperthreading: No (1 thread per core)
Clock Speed: Up to 4.06 GHz (P-cores)
Architecture: ARM64 (Apple Silicon)
OS: Sequoia 15.3.2

IV/ Measurements

The end number tested is **900000**. A high value has been chosen because the testing machine is very powerful. This choice helps to observe a clear difference between results.

Execution time (in ms)

exec time (ms)	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**32**
static (block)	*15524*	*11341*	*7165*	*5160*	*4225*	*3778*	*3591*	*3599*	*3206*	*3395*
static (cyclic) [task size: 10 numbers]	*15469*	*8117*	*4428*	*4593*	*3302*	*2993*	*3440*	*3122*	*3156*	*3351*
dynamic [task size: 10 numbers]	*15306*	*7810*	*4392*	*3335*	*3258*	*3043*	*2856*	*2860*	*2863*	*2851*

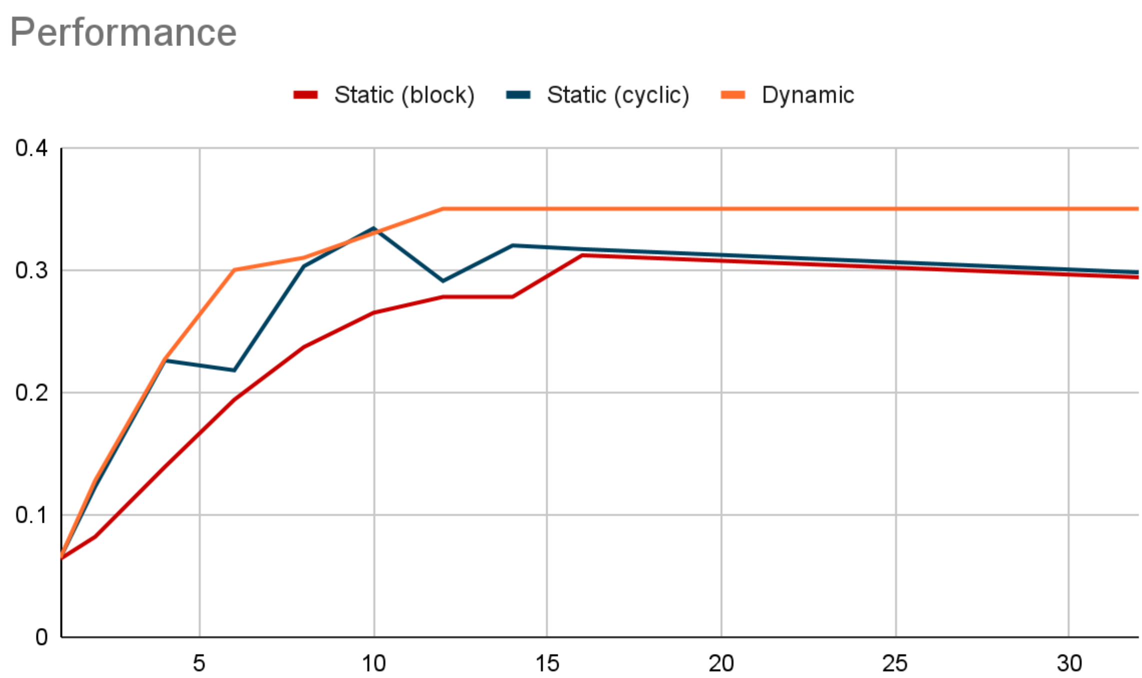


Graph showing the execution time (vertical axis) depending on the number of threads (horizontal axis)

Performance

In order to ease readability, I multiplied the performance by 1000. It doesn't affect the relationship between each value.

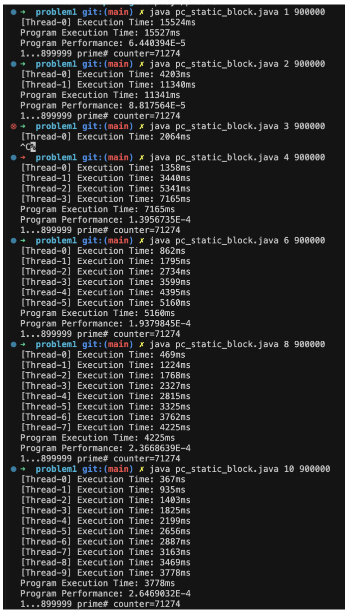
performance (1/exec time)	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**32**
static (block)	*0.064*	*0.082*	*0.139*	*0.194*	*0.237*	*0.265*	*0.278*	*0.278*	*0.312*	*0.294*
static (cyclic) [task size: 10 numbers]	*0.065*	*0.123*	*0.226*	*0.218*	*0.303*	*0.334*	*0.291*	*0.32*	*0.317*	*0.298*
dynamic [task size: 10 numbers]	*0.065*	*0.128*	*0.227*	*0.3*	*0.31*	*0.33*	*0.35*	*0.35*	*0.35*	*0.35*

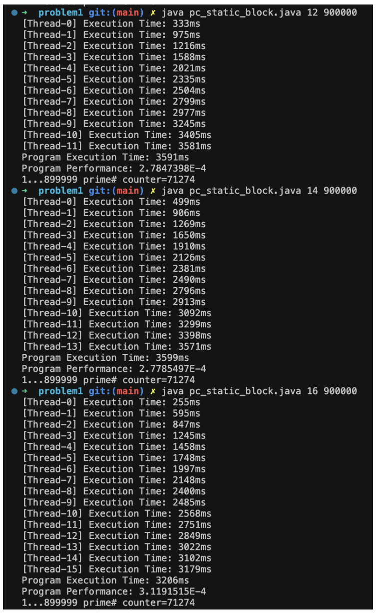


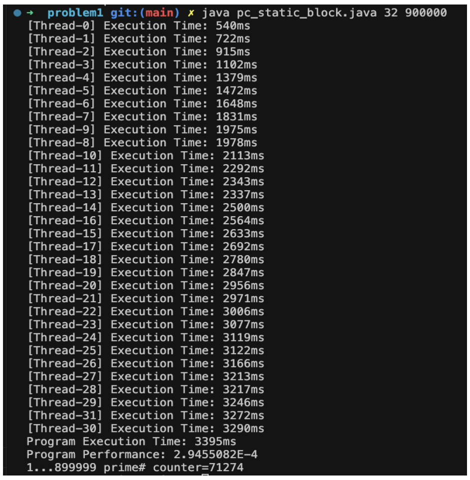
Graph showing the performance (vertical axis) depending on the number of threads (horizontal axis)

V/ Screenshots

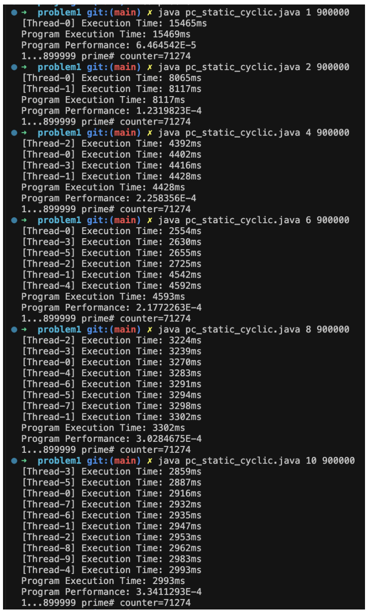
Static block

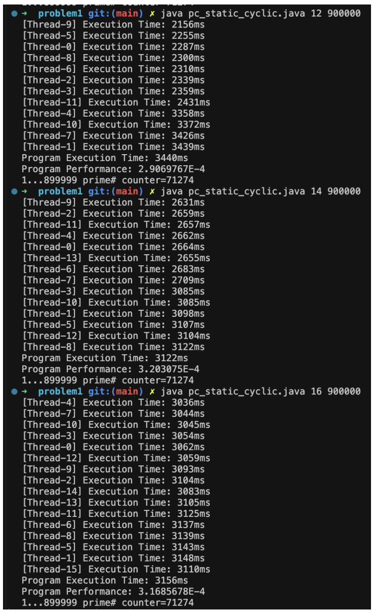


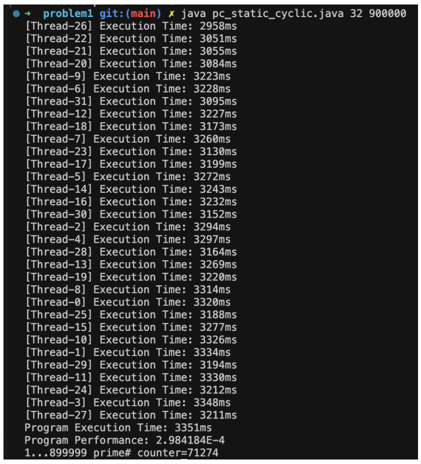




Static cyclic







Dynamic

```
● ➜  problem1 git:(main) ✗ java pc_dynamic.java 1 900000
  [Thread-0] Execution Time: 15301ms
  Program Execution Time: 15306ms
  Program Performance: 6.533386E-5
  1...899999 prime# counter=71274
● ➜  problem1 git:(main) ✗ java pc_dynamic.java 2 900000
  [Thread-1] Execution Time: 7806ms
  [Thread-0] Execution Time: 7806ms
  Program Execution Time: 7810ms
  Program Performance: 1.2804098E-4
  1...899999 prime# counter=71274
⊗ ➜  problem1 git:(main) ✗ java pc_dynamic.java 3 900000
  ^C%

● ➜  problem1 git:(main) ✗ java pc_dynamic.java 4 900000
  [Thread-1] Execution Time: 4388ms
  [Thread-3] Execution Time: 4388ms
  [Thread-2] Execution Time: 4388ms
  [Thread-0] Execution Time: 4388ms
  Program Execution Time: 4392ms
  Program Performance: 2.276867E-4
  1...899999 prime# counter=71274
● ➜  problem1 git:(main) ✗ java pc_dynamic.java 6 900000
  [Thread-3] Execution Time: 3331ms
  [Thread-5] Execution Time: 3331ms
  [Thread-1] Execution Time: 3331ms
  [Thread-2] Execution Time: 3331ms
  [Thread-0] Execution Time: 3331ms
  [Thread-4] Execution Time: 3331ms
  Program Execution Time: 3335ms
  Program Performance: 2.9985007E-4
  1...899999 prime# counter=71274
● ➜  problem1 git:(main) ✗ java pc_dynamic.java 8 900000
  [Thread-3] Execution Time: 3253ms
  [Thread-1] Execution Time: 3253ms
  [Thread-4] Execution Time: 3253ms
  [Thread-5] Execution Time: 3253ms
  [Thread-6] Execution Time: 3253ms
  [Thread-2] Execution Time: 3253ms
  [Thread-0] Execution Time: 3253ms
  [Thread-7] Execution Time: 3253ms
  Program Execution Time: 3258ms
  Program Performance: 3.0693677E-4
  1...899999 prime# counter=71274
● ➜  problem1 git:(main) ✗ java pc_dynamic.java 10 900000
  [Thread-1] Execution Time: 3036ms
  [Thread-4] Execution Time: 3036ms
  [Thread-8] Execution Time: 3035ms
  [Thread-9] Execution Time: 3035ms
  [Thread-3] Execution Time: 3036ms
  [Thread-0] Execution Time: 3036ms
  [Thread-2] Execution Time: 3036ms
  [Thread-7] Execution Time: 3036ms
  [Thread-6] Execution Time: 3036ms
  [Thread-5] Execution Time: 3036ms
  Program Execution Time: 3043ms
  Program Performance: 3.2862308E-4
  1...899999 prime# counter=71274
```

```
● ➜  problem1 git:(main) ✗ java pc_dynamic.java 12 900000
  [Thread-3] Execution Time: 2849ms
  [Thread-6] Execution Time: 2849ms
  [Thread-7] Execution Time: 2849ms
  [Thread-5] Execution Time: 2849ms
  [Thread-0] Execution Time: 2849ms
  [Thread-11] Execution Time: 2846ms
  [Thread-4] Execution Time: 2849ms
  [Thread-1] Execution Time: 2849ms
  [Thread-10] Execution Time: 2849ms
  [Thread-2] Execution Time: 2849ms
  [Thread-9] Execution Time: 2849ms
  [Thread-8] Execution Time: 2849ms
  Program Execution Time: 2856ms
  Program Performance: 3.5014006E-4
  1...899999 prime# counter=71274
● ➜  problem1 git:(main) ✗ java pc_dynamic.java 14 900000
  [Thread-10] Execution Time: 2851ms
  [Thread-13] Execution Time: 2827ms
  [Thread-8] Execution Time: 2852ms
  [Thread-7] Execution Time: 2852ms
  [Thread-2] Execution Time: 2852ms
  [Thread-3] Execution Time: 2852ms
  [Thread-6] Execution Time: 2852ms
  [Thread-0] Execution Time: 2854ms
  [Thread-5] Execution Time: 2852ms
  [Thread-4] Execution Time: 2854ms
  [Thread-11] Execution Time: 2841ms
  [Thread-1] Execution Time: 2852ms
  [Thread-9] Execution Time: 2854ms
  [Thread-12] Execution Time: 2832ms
  Program Execution Time: 2860ms
  Program Performance: 3.4965036E-4
  1...899999 prime# counter=71274
● ➜  problem1 git:(main) ✗ java pc_dynamic.java 16 900000
  [Thread-4] Execution Time: 2858ms
  [Thread-13] Execution Time: 2843ms
  [Thread-9] Execution Time: 2858ms
  [Thread-12] Execution Time: 2848ms
  [Thread-7] Execution Time: 2856ms
  [Thread-3] Execution Time: 2856ms
  [Thread-8] Execution Time: 2856ms
  [Thread-11] Execution Time: 2853ms
  [Thread-2] Execution Time: 2858ms
  [Thread-14] Execution Time: 2831ms
  [Thread-15] Execution Time: 2820ms
  [Thread-6] Execution Time: 2858ms
  [Thread-0] Execution Time: 2858ms
  [Thread-10] Execution Time: 2855ms
  [Thread-5] Execution Time: 2856ms
  [Thread-1] Execution Time: 2857ms
  Program Execution Time: 2863ms
  Program Performance: 3.4928398E-4
  1...899999 prime# counter=71274
```

```
● ➜  problem1 git:(main) ✗ java pc_dynamic.java 32 900000
  [Thread-4] Execution Time: 2847ms
  [Thread-8] Execution Time: 2843ms
  [Thread-26] Execution Time: 2754ms
  [Thread-2] Execution Time: 2846ms
  [Thread-30] Execution Time: 2747ms
  [Thread-27] Execution Time: 2756ms
  [Thread-11] Execution Time: 2839ms
  [Thread-25] Execution Time: 2756ms
  [Thread-31] Execution Time: 2747ms
  [Thread-20] Execution Time: 2773ms
  [Thread-24] Execution Time: 2763ms
  [Thread-18] Execution Time: 2786ms
  [Thread-29] Execution Time: 2753ms
  [Thread-22] Execution Time: 2776ms
  [Thread-28] Execution Time: 2757ms
  [Thread-23] Execution Time: 2768ms
  [Thread-6] Execution Time: 2847ms
  [Thread-9] Execution Time: 2846ms
  [Thread-17] Execution Time: 2787ms
  [Thread-16] Execution Time: 2799ms
  [Thread-21] Execution Time: 2775ms
  [Thread-19] Execution Time: 2785ms
  [Thread-14] Execution Time: 2824ms
  [Thread-1] Execution Time: 2844ms
  [Thread-5] Execution Time: 2846ms
  [Thread-10] Execution Time: 2841ms
  [Thread-7] Execution Time: 2843ms
  [Thread-13] Execution Time: 2831ms
  [Thread-3] Execution Time: 2844ms
  [Thread-15] Execution Time: 2814ms
  [Thread-0] Execution Time: 2844ms
  [Thread-12] Execution Time: 2834ms
  Program Execution Time: 2851ms
  Program Performance: 3.5075413E-4
  1...899999 prime# counter=71274
```

VI/ Analysis of the results

We can observe that for all multithreading strategies, the performance grows as the number of threads increases until it reaches 12 to 16 threads. The shape of the curves look similar to a logarithm function. This ceiling is probably due to the limited number of physical cores (11), in addition to overhead and memory bandwidth limitations. Indeed, there's no hyperthreading on the testing machine, so each core can't run multiple threads simultaneously.

However, we can notice differences of results between static block, static cyclic, and dynamic programs:

- **Static block multithreading** performs decently as the number of threads increases. But due to uneven workload distribution between threads, it shows the slowest results compared to the two other strategies. As you can see on the screenshots, the difference of execution time is huge between threads (255 ms against 3179 ms for 16 threads). The first threads calculate small numbers so they finish quickly, while the last ones need to calculate high numbers which takes much more time. When the first ones are done, they enter an idle state and become useless. This loss of computation resources explains the low performances of this strategy.

- **Static cyclic multithreading** shows better performance than the static block strategy. By distributing small chunks of work (in this case, blocks of 10 numbers), it achieves a more balanced workload. Each thread receives portions of both small and large numbers to process, which evens out the computation time. The execution times are much closer between threads (3036 ms against 3148 ms for 16 threads). Idle time is reduced which improves overall CPU utilization. However, we can observe some very unstable results: the performance drops at 6 threads before gaining again until 10 threads. Then it drops again at 12 threads. It's likely that efficiency cores took heavy numbers when these drops happened. E-cores being slower than p-cores, it would explain these time loss. This uncontrollable distribution introduces randomness in execution time: it's unstable.

- **Static multithreadings** decrease from 16 to 32 threads due to overhead. 11 cores have to switch between too many threads. This efficiency loss increases with the number of threads used.

- **Dynamic multithreading** is supposed to be the best at minimizing idle time. Despite potential overhead, this strategy indisputably shows best performances. The workload is even for all threads, their execution time is almost identical - to within a few milliseconds. All the available resources are used, it's ideal load balancing. The results are way more flatten then static ones: they show high stability. Execution times are less random because all potential resources are used no matter the distribution. We can observe a slow down of performance from 6 threads because the 6 p-cores are used. The 5 e-cores, less powerful, provide smaller increases until they reach the total number of threads used. Without surprises, it remains constant from this point: all available cores are used.

We can conclude that the dynamic strategy seems to show the highest and most stable performances for this algorithm. However, the testing machine being more powerful than average, overhead is likely minimized. Testing in another environment could help to check the impact of this cost, which may influence our choice of strategy depending on the running machine.