Author : Lucien Iseli, Loris Pilotto
SCIPER : 274999, 262651

Report for labo 1:

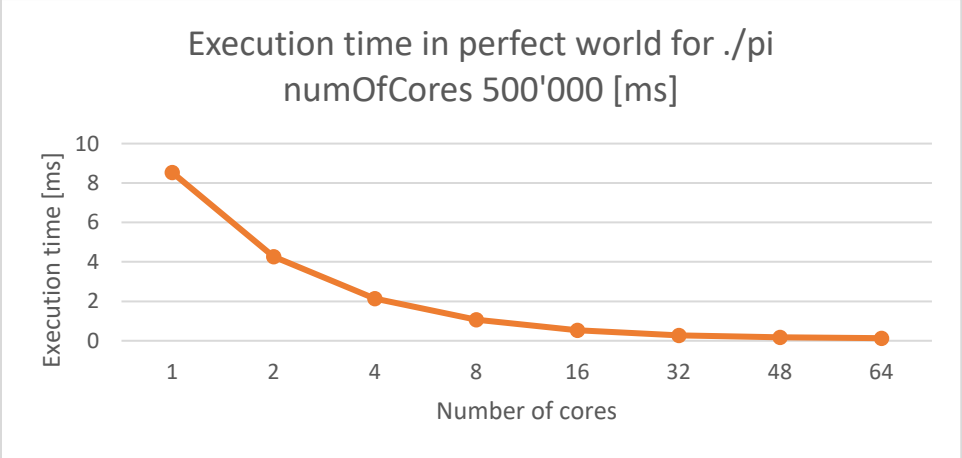
1)

a) First, there is a serial phase to initialize finals variables and an array (of the size of the number of threads) used later to store each result of the threads independently.
Then we have a parallelized part that creates the RNG for each thread and counts how many points are inside the unit circle in pi.c and that computes the area of the random rectangle in integral.c independently in each thread(each thread loops samples/num_threads times) and store the result in the array. We parallelized the for loop because it is possible since every iteration is independent. Speeding the for loop is the most efficient because it is the biggest part of the program (Amdahl's law).
Last, we have a serial phase that adds all independent counts into a total and then computes the final result with what we calculated in the for loop.

b) In pi.c: In the first phase, there is a single operation (the assignment is not an operation) which is a division so it dominate the execution time even if the initialisation phase hasn't a big influence on the overall execution time. The parallelized for loop has 2 multiplications, 3 addition (with the i++ in the for loop) at most, 2 comparison at each iteration. Thus, the most expensive operation is the multiplication as it is the most expensive and there is no other operation used a lot more. The third phase of computing pi has 2*num_threads additions(with the i++ in the for loop), 1 division and 1 multiplication, division is way slower than multiplication and additions but we make a lot of addition (depending on the number of threads) so the operation that dominate the execution phase can be either the division or the addition.
Integral.c is very similare to pi.c.

c)The same reasoning applies for both pi.c and integral.c
The initialization phase takes constant time :O(1).
The for loop takes O(samples/number of threads) time because every operation in its body takes constant time and every thread should take an equal portion of the for loop (only computing f(x), in integral.c, may take longer, depending on f, but in our case it is identity function).
The computing result phase takes O(number of threads) time because the last calculus can be done in constant time and the body of the for loop is a single addition which takes constant time.
So the execution time of the program is: O(max{ samples/number of threads, number of threads}).
So we see that if we put an enormous number of threads the program will actually run slower.

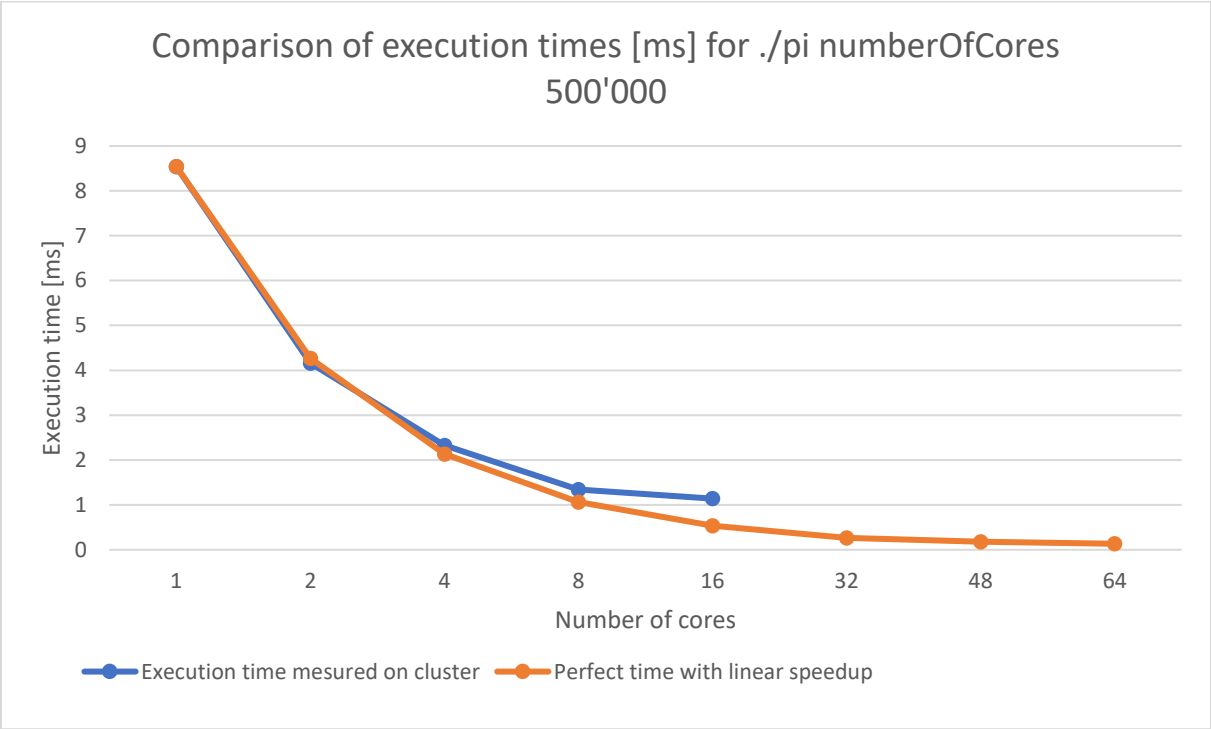
d)The speedup of the multithreaded program should be close to linear with a slope of 1. (Every time we double the number of cores it divide the time of execution by two so the speedup is multiplied by two)



Author : Lucien Iseli, Loris Pilotto
SCIPER : 274999, 262651

2)
We only put the numbers up to 16 cores, because we can not run with 32, 48 or 64 cores on the cluster because we get an error if we try too many cores on one node: “sbatch: error: Batch job submission failed: Requested node configuration is not available”. However, we get enough information to see how it scales in reality. We can see that we don’t get any major speedup after 8 cores.

Number of cores	Execution time [ms] on SCITAS	Speedup compared to 1 core
1	8.54	1
2	4.16	2.05
4	2.33	3.67
8	1.34	6.37
16	1.14	7.49
32		
48		
64		



3)We can see that the practical speedup is slower than the theoretical speedup and it is mostly due to the fact that in the theory we supposed that the for loop took 100% of the execution time but it is not the case in reality, so with parallelism we are able to divide the time of the for loop by the number of threads but not the first and the last phase. So Amdahl’s law starts to kick in when we add many cores, because the loop takes less and less time, thus we speedup a part of the program that’s getting shorter.