Laboratory 3 - Report

Group Members:
Javier Almario, NIP: 962449
Alvaro Provencio, NIP: 960625

1 Question 1: What is the goal of the std::stoll function?

Given one numeric string the function converts to long long int type [1].

2 Question 2: Does the execution time scales linearly with the number of steps?

From the plot (a) we can say yes, for large enough N, the execution time scales linearly with the number of steps. For smaller N, the times are "noisy" and flat, because at small size This is an expected behavior considering that the number of steps is directly the number of terms in the sequence.

3 Question 3: What is the CPI of the program?

We measured with $perf$ the number of cycles and instructions with 4294967295 steps, then from the output, we get $Cycles = 198,271,243,060$ and $Instructions = 99,165,703,392$ This means, that on average, each instruction takes 2 CPU cycles to complete.

4 Question 4: Scalability Analysis

In this test, we executed every situation (number of threads) 3 times, so that we are showing the mean of those 3 executions for each situation on plot (b). On all the cases we got a coefficient of variation between 0.2 and 2%, which shows a low variation on execution times for all the experiments.

The time spent to make the calculation reduces with the number of threads, being a bigger reduction for 2 or 4 threads as it reduces the workload almost by half. But the performance does not improve much with 8 and 16 threads, where the workload of each thread is not that much and the CPU needs to take more time managing them.

5 Question 5: Do you obtain the same exact pi approximation value?

Result for 4294967295 steps. They differ in the 10th decimal digit.:

- **Sequential:** Time: 50718.1 ms pi value: 3.141592653356969973.
- **With 8 threads:** Time: 11494.03 ms pi value: 3.1415926538226246

This behavior happens because floating points only maintain the commutative property when rounding, not the associative, and since the summation of all the steps is different in the sequential and parallel programs, there are different rounding errors on each case.

6 Question 6

The code with std::async implemented does not require to share any variable among the asynchronously launched functions, each task can get its inputs as normal function arguments and returns its result using $std :: future$ which holds a shared state, and $std :: async$ allowing to run the code in parallel.

We ran *pi_taylor_parallel_async* for 4294967295 steps and 1, 2, 4, 8, and 16 threads, the same way as in the parallel exercise and plotted it in (c).

7 Question 7

We obtained the measurements shown in table 1, as we increase the number of threads beyond the number of CPU cores, the system needs to manage more threads, which leads to increased context-switching. The sequential run has the most cycles overall, but it is also running at a higher execution time. Finally, Parallel runs have almost identical instructions, and sequential run executes fewer instructions overall, but because it runs on a single thread.

Metric	**4 Threads (Parallel)**	**8 Threads (Parallel)**	**1 Thread (Sequential)**
Cycles	69,573,804,004 (3.698 GHz)	70,323,099,409 (3.411 GHz)	198,695,241,914 (3.979 GHz)
Context-Switches	160 (8.504 /sec)	957 (46.423 /sec)	911 (18.245 /sec)
Instructions	73,156,057,774	73,174,420,824	99,266,858,722

Table 1: Comparison of Cycles, Context-Switches, and Instructions for Different Configurations

What is the impact of moving from long double to float in terms of execution time and accuracy between pi_taylor_parallel_async and pi_taylor_parallel_async_kahan?

The program with float, the result of pi is equal to 3.141593 and with long double is equal to 3.141592653822624859, We get less decimal digit for float case 7 because float is 32-bit single-precision, with long float 80-bit extended precision we got 18 digits, The float type provides much less precision than long double which is correct up to at least the 9th decimal place.

Program	**Context-switches**	**Cycles**	**Instructions**	**Time Elapsed (s)**
`Async kahan (8)`	590	70,269,412,469	73,163,350,253	2.56
`Async (8)`	2,062	229,224,612,832	99,286,807,009	9.14

Table 2: Comparison of Context-switches, Cycles, Instructions, and Time Elapsed for two programs using 8 threads.

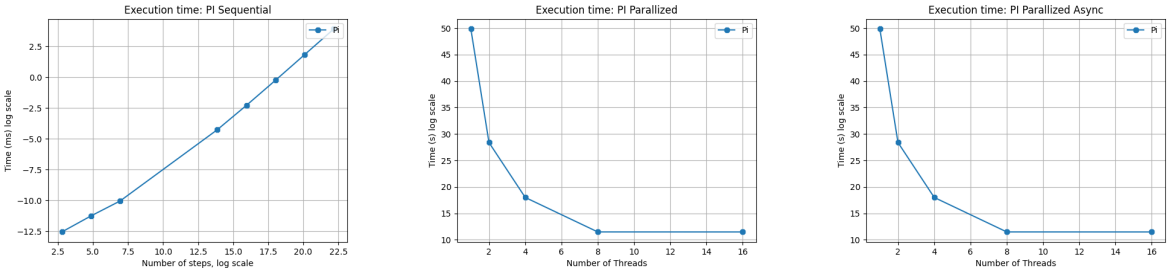
In terms of execution, we have observed that Async with kahan is slightly faster than Async and has less cycles as well, computing the IPC for each one, we have for Async kahan $IPC = 1.04$ and Async $IPC = 0.43$, showing that Async kahan executes multiple instructions per cycle, suggesting better instruction-level parallelism or vectorization.

8 Optional β

- **8 Threads:** Threads time: 0.5-0.98 ms . Total execution time: 1.47 ms
- **64 threads:** Threads time: 0.018-0.031 ms but with some others around 0.05 or 0.08 ms. Total execution time: 3.86 ms

With 8 threads we see that some of them take longer or lower than the others but without much disparity. But for 64 threads we see that some of them take double the time or more as the others, this is because the scheduler may interrupt some threads. All this work that the scheduler has to do is reflected on the total execution time, that is more than 2 times the other.

9 Results



(a) Sequential (b) Parallel (c) Async

Figure 1: Result of each program.

References

[1] std::stoll, `https://en.cppreference.com/w/cpp/thread/async.html`, 2025.

[2] Branch predictor, `https://en.wikipedia.org/wiki/Branch_predictor?utm_source=chatgpt.com`, 2025.