Assignment 3

Introduction to Parallel Programming.

David Padilla Orenga
Ignacio Pastore Benaim

Question 1

What is the goal of the `std::stoll` function?

The `std::stoll` function in C++ converts a string (or C-style string) representing a number into an integer of type `long long int`. This function is particularly useful for processing command-line arguments, where inputs are typically strings. In our program, `std::stoll` reads the number of steps and threads provided as command-line arguments, allowing for their interpretation as integer values needed to define the calculation workload and parallelization level.

Question 2

Does the execution time scale linearly with the number of steps?

Steps	Execution Time (ms)	Value Obtained
16	0.0156 ± 0.006	3.200365515409547529
128	0.0085 ± 0.005	3.149344475124620055
1024	0.0157 ± 0.005	3.142568263113741388
1048576	3.732 ± 1.514	3.14159360726320027
4294967295	6256.498 ± 137.659	3.141592653356970138

Table 1: Execution time statistics for different step sizes (in milliseconds)

Figure 1: Log-log plot of Mean Execution Time and Standard Deviation vs. Steps

As shown in Table **??**, the mean execution time generally increases as the number of steps grows. The log-log plot reveals that execution time does not scale perfectly linearly with the number of steps. This is particularly noticeable with large step sizes, where execution time increases more than linearly. This deviation from linear scaling is likely due to overhead factors in computation and memory handling, which become more significant at larger step counts. Additionally, the variability, represented by the standard deviation, also tends to increase as the number of steps grows, indicating that larger computations introduce more variation in execution time since larger calculations intensify the load on system resources, making them more susceptible to fluctuations in performance due to factors like cache misses, context switches, and other system-level activities that introduce variability.

Question 3

Please measure with `perf` the number of cycles, instructions, and context-switches for the application with 4294967295 steps.

Cycles	Instructions	Instructions per Cycle (IPC)	Context Switches
17,987,511,771	64,438,419,167	3.58	28

Table 2: Performance metrics for the sequential version of `pi_taylor_sequential` with 4294967295 steps

From these results, we observe that the program achieves a high IPC of 3.58, indicating efficient utilization of CPU resources during execution. These results highlight that the program maintained a steady execution flow with minimal context switching, contributing to consistent performance. The high number of cycles and instructions reflects the large computation workload associated with approximating π to high precision. The low number of context switches (28) and CPU migrations (6) suggests minimal interruptions, which is logic for sequential programs like our with consistent workloads.

Question 4

Run `pi_taylor_parallel` for 4294967295 steps and 1, 2, 4, 8, and 16 threads, and perform a scalability analysis, including a plot and some measurement of the variability of the program, such as the coefficient of variation.

Threads	Execution Time (s)	Coef. Variation (%)
1	5.6990 ± 0.1157	2.03
2	2.8865 ± 0.0240	0.83
4	1.5266 ± 0.0177	1.16
8	1.6768 ± 0.0368	2.19
16	1.6940 ± 0.0165	0.97

Table 3: Scalability results for different thread counts (5 iterations)

The results show that execution time decreases significantly as the number of threads increases up to 8 threads, aligning with the maximum number of available CPU cores. Beyond 8 threads (e.g., 16 threads), we observe minimal improvement in execution time, which can be attributed to oversubscription, where more threads than available cores leads to increased context switching and reduced efficiency.

The coefficient of variation (CV) generally increases as the thread count rises, with a noticeable increase at 8 threads. This trend indicates higher variability in execution time with parallelism, especially at higher thread counts, likely due to resource contention. Overall, the program demonstrates good scalability up to the core count limit, after which oversubscription reduces the efficiency gains.

Question 5

If you compare the parallel version with 8 threads and the same number of steps (4294967295) to the sequential version, do you obtain the same exact π approximation value?

In comparing the π values computed by the sequential and parallel versions (with 8 threads) for 4294967295 steps, a slight difference is observed. The sequential version computes π as 3.141592653356970138, whereas the parallel version approximates π as 3.141592653822625016. This discrepancy arises due to the non-associative nature of floating-point addition. In floating-point arithmetic, addition is not associative, meaning that the order in which terms are added affects the final result. In the parallel implementation, each thread computes a partial sum over a subset of the terms in the Taylor series, and these partial sums are combined at the end. This approach introduces a different order of operations compared to the strictly sequential summation, leading to slight variations in the final result. These rounding errors accumulate differently based on the summation order, causing a minor discrepancy in the computed value of π.

Question 6

Please repeat `perf` measurements of the number of cycles, instructions, and context-switches for `pi_taylor_parallel` with 4294967295 steps, and 4 and 8 threads to perform the comparison with the sequential version.

Threads	**Cycles**	**Instructions**	**Context-Switches**	**Instructions per Cycle (IPC)**
4	19,792,257,258	64,440,408,243	325	3.26
8	38,431,180,728	64,494,714,007	4,695	1.68

Table 4: Performance metrics for 4 and 8 threads in `pi_taylor_parallel` with 4294967295 steps

This comparison indicates that while increasing the thread count to 8 results in only a minor improvement in execution time, it introduces much higher context switching due to oversubscription, where more threads than available CPU cores leads to increased contention for resources. The higher cycle count and reduced IPC in the 8-thread version highlight the reduced efficiency from managing more threads than CPU cores available, illustrating the impact of oversubscription on performance.