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

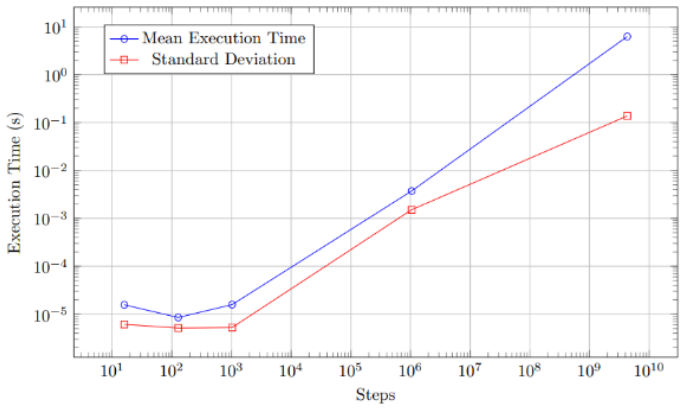

Figure 1: Execution time statistics for different step sizes (5 iterations) in sequential execution.

As shown in Figure 1, the mean execution time generally scales linearly with the number of steps, though an initial dip is observed at small step counts, likely due to fixed overheads and optimizations that impact minimal workloads. As steps increase, execution time grows proportionally, consistent with the workload of a Taylor series approximation. The standard deviation also rises with the step count, indicating greater variability in performance for larger computations. This trend is likely due to system-level factors such as memory access and context switches. For smaller step sizes, both mean execution time and variability remain low, showing consistent performance under lighter loads.

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

We observe that the program achieves a high IPC of 3.58, indicating efficient utilization of CPU resources during execution. 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.

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

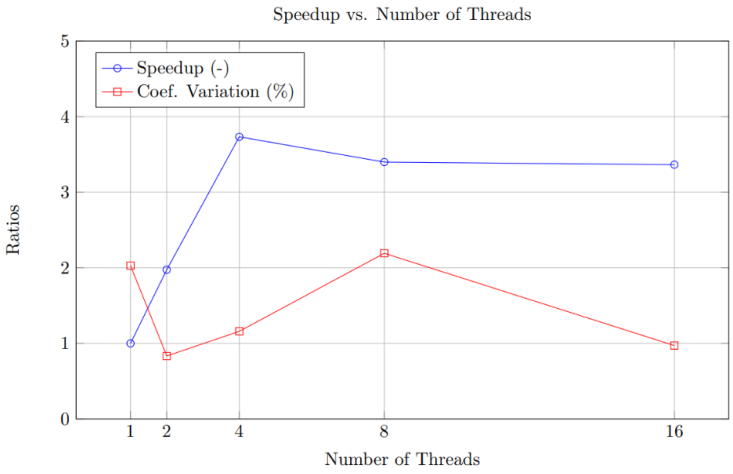

Figure 2: Execution time statistics for different step sizes (5 iterations) in parallel execution.

`Speedup` is a measure that indicates how much faster a parallel process is compared to its one thread version, (Single thread time / multiple threads). In the results, speedup increases significantly up to 4 threads, where optimal performance is observed. However, at 8 threads, the speedup decreases slightly, suggesting that adding more threads does not proportionally improve performance. Beyond 8 threads, `oversubscription` occurs, leading to increased context switching and reduced efficiency. The `coefficient of variation (CV)`, which measures the variability of execution times, starts low with 1 thread. As the count increases to 2 threads, there is a slight drop, followed by a steady increase until reaching a peak at 8 threads. This indicates that while parallelization enhances performance, it also introduces variability in execution time. At 16 threads, the CV decreases slightly but remains above the levels observed with fewer threads, suggesting that performance becomes less predictable as the number of threads increases, emphasizing the importance of careful resource management to optimize efficiency.

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, 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, introducing a different order of operations compared to the strictly sequential summation, leading to slight variations in the final result.

QUESTION 6: Repeat `perf` measurements 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)
Sequential	17,987,511,771	64,438,419,167	28	3.58
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 2: Performance metrics for sequential, 4 and 8 threads with 4294967295 steps

The analysis shows that the number of cycles increases with parallelism, rising further as more threads are added, while the instruction count remains approximately the same. This indicates that although the workload is distributed across multiple threads, the overhead associated with managing those threads leads to an increase in cycles. Additionally, context switches increase significantly with the number of threads due to greater contention for shared resources, which further contributes to performance overhead. These findings highlight the trade-offs between the benefits of parallelization and the inefficiencies introduced by resource contention, leading to diminishing returns in execution efficiency.