Diego Bielsa Monterde (776564), Alejandro Báscones Gállego (801469)

Programming And Architecture Of Computers
Laboratory 3

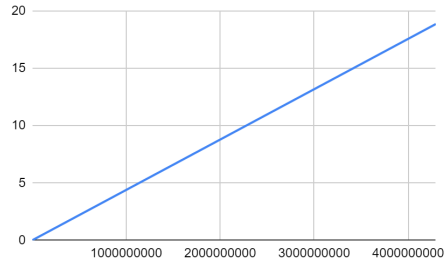
Sequential
Function std::stoll

The goal of std::stoll is to convert string type arguments into long long int, doing that we can use the number of steps given by the user.

Time Scale

After executing the sequential code for the different number of steps, we obtain the following times that follow the linear behavior that can be observe in the graph:

	Number of steps				
	16	**128**	**1024**	**1048576**	**4294967295**
Mean execution time (s)	1.59e-6	2.16e-6	6.26e-6	4.85518e-3	18.837257
Deviation (s)	1.2e-8	1.9e-8	9.8e-8	9.87e-4	0.546
Coefficient of variation	7.547e-3	8.796e-3	0.0156	0.2033	0.028985
Result	3.079153537750244140625	3.1333779525756835 9375	3.140615701675415 0390625	3.141595363616943 359375	3.141596794128417 96875
Error	0.062438115839548 097	0.007812872833957 2605	0.000976303086377 7618	1.012753340256942 e-7	2.015331479844225 2e-8



Hardware performance

In the following table we can see the number of cycles, number of instructions and number of context-switches given by the instruction perf while running the sequential code with 4294967295 steps:

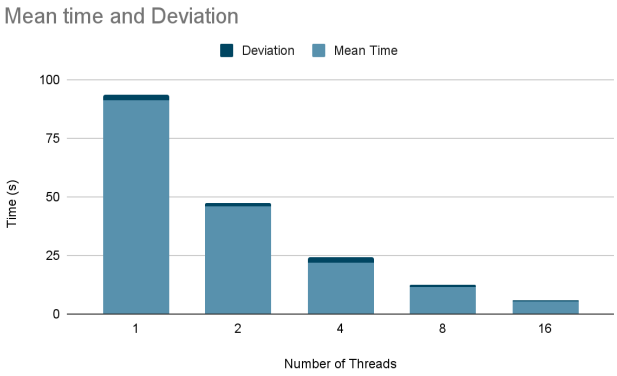
Number of cycles	49105702380
Number of instructions	38665655349
Number of context-switches	10

Parallel

Scalability Analysis

For the following analysis the program "*pi_taylor_parallel*" has been executed for 4294967295 steps and 1, 2, 4, 8, and 16 threads. Before getting the conclusions, the following table and graph show performance measurements as well as the pi approximation obtained for each execution (100 executions per amount of cores).

	Number of threads				
	1	**2**	**4**	**8**	**16**
Mean exec. time (s)	1m31.23s	46.02	22.04	11.38	5.56
Dev.(s)	2.34	1.23	1.12	0.67	0.45
Coeff. of var.	0.025	0.026	0.0508	0.0588	0.0809
Result	3.14159265581537283651414327323436 7				
Error	2.266303024429033e-12				



As the program has been executed in "pilgor" (it has 96 cores), the scalability looks like the ideal one because when the amount of cores is doubled, time decreases to half thanks to the even workload per core. It can be easily seen in the plot, in which each bar decreases to half.

Value variation from sequential to 8 threads parallel

The value obtained with 8 threads and 4294967295 steps and the one obtained in a sequential way with the same number of steps is a little bit different. The reason is based on the floating point operations, specifically in the precision loss as the number grows. Binary representation causes this loss, because the higher the number is, the more bits you need to represent the larger number parts, losing then bits to represent the precision in decimals. Therefore, when there are 8 threads each one is adding low numbers to each component of the output vector (its size is the same as the number of threads), so the used numbers tend to be lower than the sequential algorithm that adds them all to the same variable; this causes a higher precision when 8 threads are used.

Hardware resources

	4	**8**
Number of cycles	205.851.523.747	202.038.273.739
Number of instructions	484.291.326.707	474.094.911.163
Number of context-switches	77	118

Regarding the number of cycles and instructions, the parallel algorithm has many more than the sequential algorithm. This behavior is given by the amount of context-switches, which is higher in the algorithm with threads as they have to "compete" for getting the core. This competition makes each thread get and leave the core multiple times (number of context-switches) which is expensive due to the relaunch of the thread in such a core. Besides, thread management also implies a cost, which raises the number of cycles and instructions.