Lab 3 -Introduction to Parallel Programming

Diego Martínez Baselga

`735969@unizar.es`

David Morilla Cabello

`822899@unizar.es`

Universidad de Zaragoza

Report for the course:

PROGRAMMING AND ARCHITECTURE OF COMPUTING SYSTEMS

Grupo Arquitectura de Computadores

20th November 2020

1 Introduction

The lab introduces the concepts of basic parallel programming. The proposed problem is to obtain a π approximation using Taylor series. In order to perform an comparison, the parallel and sequential codes are required. Their performance in term of execution time depending on the algorithm step and number of threads are studied. Other metrics as the number of cycles, instructions and context-switches are analyzed for the different versions.

Additionally, some questions are asked regarding floating point numeric precision. With this regard, an additional exercise requires to implement the Kahan's summation algorithm in order to reduce the error. Finally, a question about load balancing is studied and discussed.

The system used to do the project is Pilgor, which belongs to the EINA. It is a Linux system with 96 cores of 2.6 GHz, 64 bits ARM processors, divided into 2 sockets with 2-SMT 24 core processor. For the last two questions, a personal computer is used too. This computer is a Linux system with 4 cores of 2.3 GHz, 64 bits Intel i5 processor.

2 Results

In this section the results are computed and discussed.

2.1 Sequential π computation

2.1.1 Code 1: Sequential

The sequential code for the π value computation is based on a function with a simple for loop with an accumulator that stores the sum of every term. The loop computing the approximated value of π using the following Taylor series:

$$\pi = 4\sum_{i=0}^{N} \frac{(-1)^i}{2i+1}$$

The function `std::chrono` was used to measure the time that the computation takes.

2.1.2 Question 1

The function `std::stoll` function is used to convert the parameters of the `main` function. The `main` function receives the parameters as an array of strings in the variable `argv`. However, the parameters are used as numeric values. The name `stoll` is for "string to long long" and it converts an input string to `long long` type.

2.1.3 Question 2

After having implemented the sequential version of the program, its execution time is measured and analyzed. The program executes a loop iterating in the number of steps. Therefore, its execution time scales linearly with the number of steps, O(n). The time for each N value with logarithmic scaled axis are represented in Figure 1 and the error of the result with respect π are shown in Figure 2. In the case of the times, the regression is not accurate due to the fact that the noise of the last measure greatly changes the rest of the points. For this reason, we checked intermediate values to validate our hypothesis. They are shown in Figure 3 with the tendency line, the equation and the R^2 value.

2.2 Parallel π computation

2.2.1 Code 2: Parallel threaded version

The parallel threaded version receives the number N and the number of threads. With this information, divides the N value in different work "chunks" with nearly the same number of elements and therefore number of operations. Hence, the work is balanced between the different threads. For N steps and T threads, $N \bmod T$ threads must do $\frac{N}{T} + 1$ iterations, while the rest of the threads must do $\frac{N}{T}$. Each thread saves its own accumulator in a vector in order to ensure mutual exclusion between the threads when they read and write the data. When all the threads are finished, the value of the accumulators are summed and multiplied by four to obtain the π value.

2.2.2 Question 3

For this question, an event based profiling is performed. The metrics that are analyzed are context-switches, cycles and instructions. They are computed with the tool `perf stat`, and the results obtained are shown in Table 1. There are no context switches, which mean that the program is not interrupted, as it is executed in one core and the scheduler assigns it to that core for all the duration of the program. The number of instructions and cycles are high, as the program requires a big number of operations. The CPU executes 1.86 instructions per cycle.

2.2.3 Question 4

The execution time of the parallel version of the program has been measured for 4294967295 steps, for 1, 2, 4, 8 and 16 number of threads. To perform a proper analysis, the execution for each number of threads has been performed 5 times, the average and the variance coefficient are computed. The results are shown in Figure 4 and Table 2. The variance coefficient of the sequential version is much lower, as creating threads and assigning them to cores increases the variance of the duration of the program. It may be seen from the metrics gathered that as the number of threads is multiplied by two, the execution time is approximately divided by two.

The time of each thread was also gathered and analyzed for one run of the experiment. It can easily be seen how the time taken by each thread is similar. Still, the total execution time for that specific experiment and each number of threads was really similar to the thread which took the larger time to complete as expected. Even though they are quite balanced, it is interesting to notice that the later threads (the ones taking larger step values) take more time than the initial. It could be due to the fact that the later threads are assigned larger step values, so they work with larger floating points number.

2.2.4 Question 5

Floating points are commonly characterized by four numbers: a radix or base, a precision and two extremal exponents. They are also defined by an integral significand and the exponent. There are different floating-points systems used to represent numbers and not every number may be represented using a particular system. In general, the results of floating-points operations are not exactly representable and the result has to be rounded.

This problem repercutes in additions and products as associativity and distributivity are lost. In our problem, in order to compute the result in a parallel way, associativity is assumed. The programs perform the summations in different order. In the sequential algorithm, the Taylor series is performed in a single summation:

$$\pi = 4\sum_{i=0}^{N}\frac{(-1)^i}{2i+1}$$

In the parallel version, different chunks of the summation are assigned to each thread. The equivalent operation is:

$$\pi = 4\left(\sum_{i=0}^{n_1}\frac{(-1)^i}{2i+1} + \sum_{i=n_1+1}^{n_2}\frac{(-1)^i}{2i+1} + ... + \sum_{i=n_j+1}^{N}\frac{(-1)^i}{2i+1}\right)$$

Where n_i is the limit of each chunk. Each summation is calculated separately and their results are added.

Therefore, both methods accumulate a different rounding error (associativity is not fulfilled) and the result will be different.

2.2.5 Question α

In order to solve the rounding problem with sums, the Kahan summation algorithm is used. It stores in variables the negative rounding error using the difference between the accumulated variable and the small summed value. In the next iteration, the error is added to the next small summed value in order to incorporate it in the big accumulated value. Thus, the error is compensated and the remaining error only depends on the floating-point precision.

In the last iteration for each thread, the remaining error is stored in an array. When the sum by each thread is finished, this error is summed again with the Kahan summation. This value is used to compensate the error produced when summing the result by every thread as these results will also differ in order of magnitude. An alternative to avoid this last step would be to reorganize the chunk assignment in a way that the threads compute a similar order of magnitude summation. For example, assigning one value per each number of threads to a different thread.

The result of the computation is a more accurate π approximation than without the Kahan summation, as it may be seen in Table 3. The π obtained with the sequential method, using `long double` as data type, is used as the correct value, and it is compared to the result of the parallel algorithm using Kahan's algorithm and the standard version, using `float` as the data type and 4294967295 steps.

However, the execution time increases heavily, as the number of arithmetic operations performed is much bigger. The execution time of the standard parallel version with a `float` data type used for the experiment was 6197865334 ns, whereas for the Kahan version, it was 20662423323 ns.

2.2.6 Question 6

An event based profiling is performed similarly as in Section 2.2.2. This time, it has been performed with 4 threads, in Table 4, and 8 threads, 5.

The used system has a big number of cores (Pilgor), which allows to execute the experiments without context-switches. Another possibility is that `perf` is silently failing to count context switches because the user is not root [1]. In a machine with less cores, the sequential version would be likely to have no context switches too, but as the number of threads increases, the number of

[1]Extracted from answer in
`https://unix.stackexchange.com/questions/439260/why-does-perf-stat-show-0-context-switches`

context switches will increase; as there would be more processes than cores trying to be executed. In order to validate the hypothesis that a larger number of threads would create more context switches, a relaxed version of the experiment was run in computer with less resources. This is clearly illustrated in Section 2.2.7, where the 32 threads experiment (performed in a personal computer) have many context switches.

The number of instructions and number of cycles is lower as the number of threads increases. However, the instructions per cycle rate decreases. This is counter-intuitive, as an increment in the number of threads would increase the number of instructions and cycles in their creation and processing. The reason why this happen could be that the function executed by the threads (the loop) is optimized by the compiler. This optimization results in more instructions but very efficient. Therefore, as the number of threads increase, the number of optimized instructions executed will decrease. Therefore, the total number of instructions will be smaller but the instructions per cycle rate will be smaller too.

2.2.7 Question β

To perform this task, the personal computer described in 1 with 4 cores was used. In this case, the cores are not exclusively executing this experiment, so the threads need to compete with other processes to be executed.

The parallel version of the program was been executed for 4 threads, the number of cores of the machine; and 32, 8 times the number of cores. The experiment was repeated 50 times, obtaining the execution times for each of them. The mean execution time is 6406242.52 and standard deviation is 3094833.871 for the 4 threads experiment. In the case of 32 threads, the mean execution time is 7092202.4 and the standard deviation is 3483344.391. For each thread, the duration, starting and ending time of one execution are measured. They are shown in Table 6 and Table 7. Starting and ending times are measured using `timeofday()`, and execution times are measured using `std::chrono`, as it was done in the previous experiments.

The machine is able to execute 4 threads at the same time at most. Therefore, increasing the number of threads does not increase the efficiency of the program. Thus, the time taken to execute the program for 4 and 32 threads is similar. However, for 4 threads is more efficient than for 32 threads because the program needs to execute more instructions and do more context switches, which means increasing the execution time. The standard deviation is high because the machine is executing other processes at the same time and that may increase the difference between the executions. In Table 6, it may be seen that all 3 processes start at the same time, and the other one needs to wait to be executed, as the scheduler does not allow to be in the fourth core. In the Table 7, it can also be seen that there are threads with a delayed start time. In addition, more than 4 threads are been executed at the same time interval. This is because the scheduler is making them sleep and execute doing context switches, which makes the program slower. There are threads that have a much bigger duration, which is due to a context switch.

Appendix

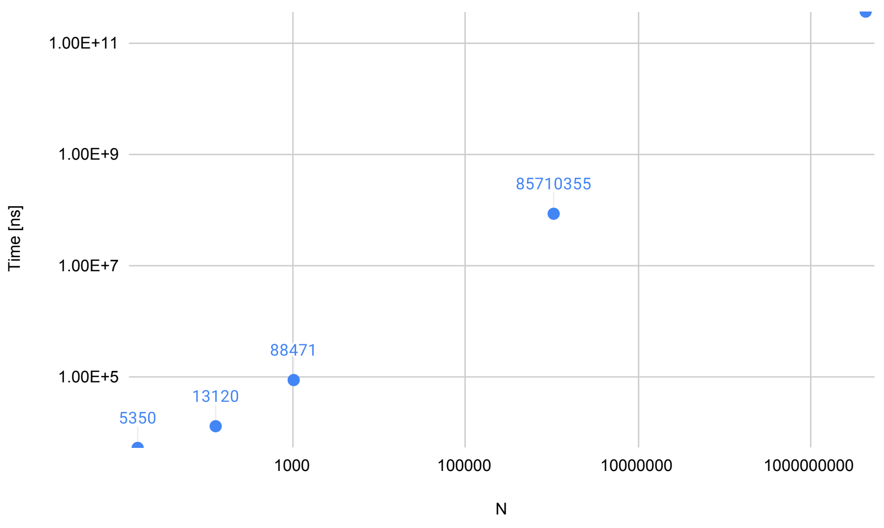

Figure 1: Execution time for sequential π approximation for the requested number of steps.

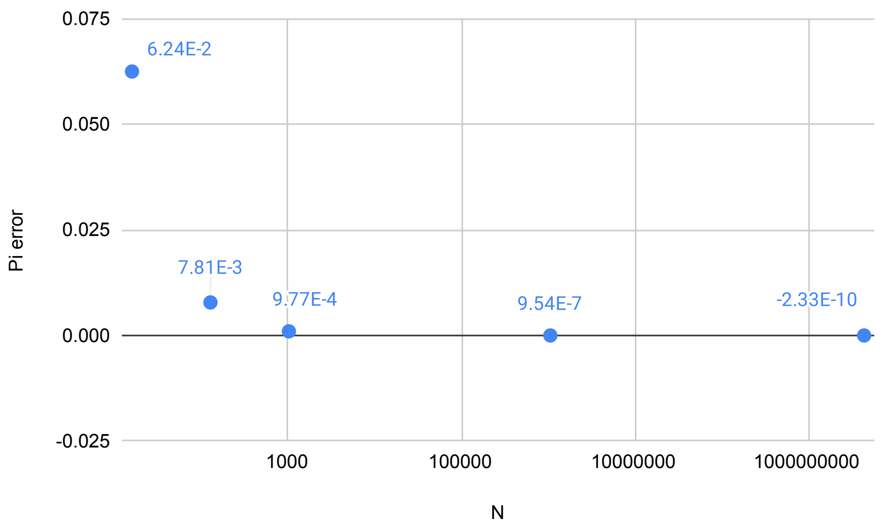

Figure 2: Error of π depending on the number of steps.

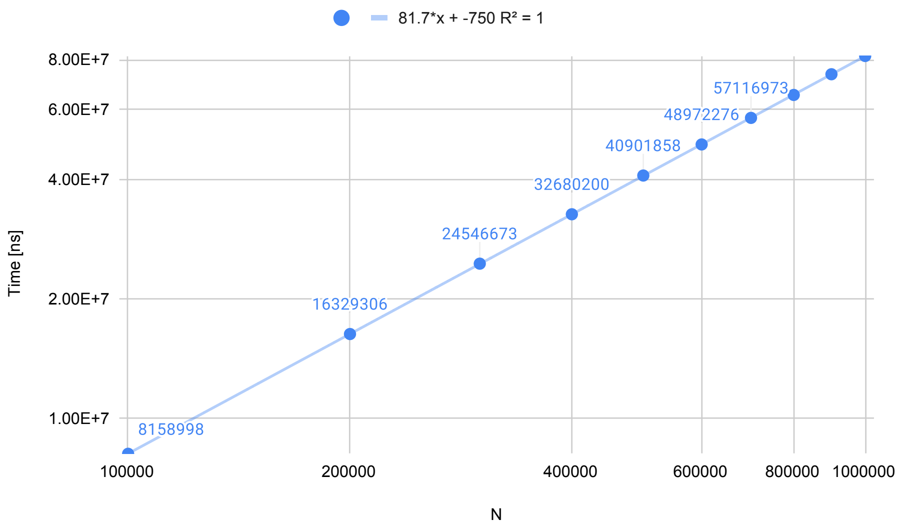

Figure 3: Additional execution time measures for different number of steps in order to ensure the linear evolution of the program.

metric	value	rate
context-switches	0	0.000 K/sec
cycles	948892883067	2.599 GHz
instructions	1760707418158	1.86 insn per cycle

Table 1: Number of context-switches, cycles and instructions of the sequential program with 4294967295 steps

Number of threads	Mean	Std. dev.	Coeff. Variance
1	364940648990	20974417.62	0.00005747350336
2	183938563171	266542344.7	0.001449083542
4	92720060189	247904972.7	0.002673692965
8	46532888371	10420600.67	0.0002239405511
16	23281641494	30808437.3	0.001323293175

Table 2: Metrics observed in the parallel execution of the program with 5 repetitions for each number of threads.

algorithm	π value	error
goal	3.141592653822623882170723120825823	-
standard parallel	3.14159679412841796875	$-4.14053862473029... \times 10^{-}6$
Kahan parallel	3.1415927410125732421875	$-8.74227800037249... \times 10^{-}8$

Table 3: π approximations comparison.

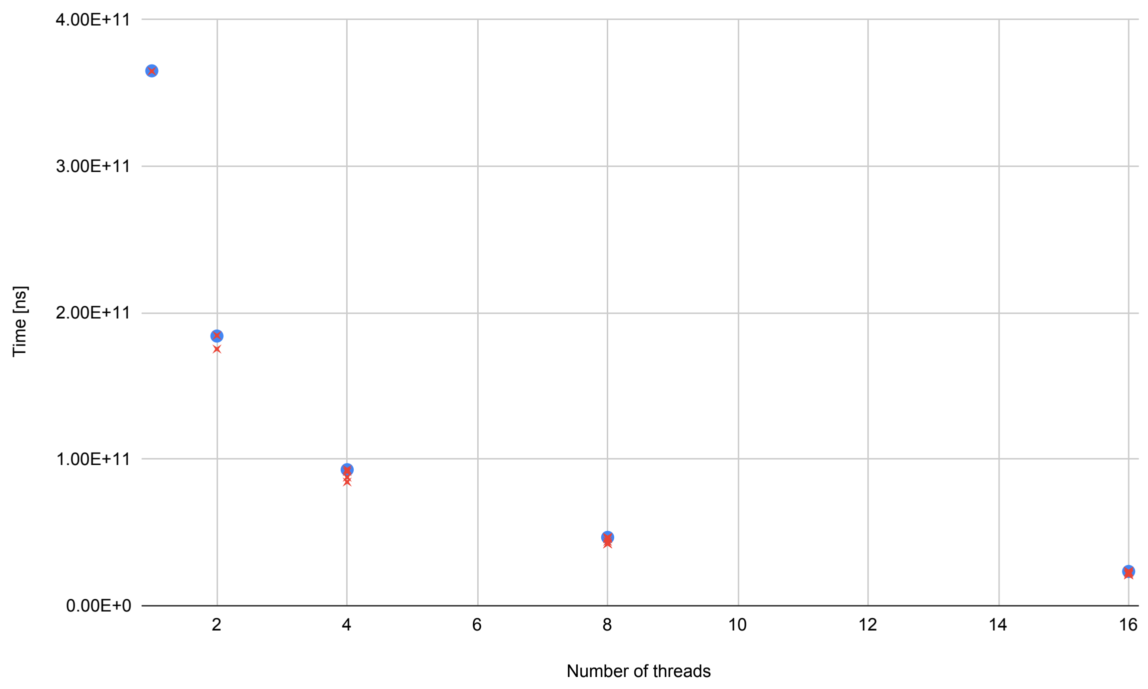

Figure 4: Execution time for parallel versions of the program for a different number of threads. The dots are the average value. The crosses are the specific times for the different threads in one of the experiments.

metric	value	rate
context-switches	0	0.000 K/sec
cycles	924461383826	2.599 GHz
instructions	1664070694298	1.80 insn per cycle

Table 4: Number of context-switches, cycles and instructions of the parallel program with 4294967295 steps and 4 threads

metric	value	rate
context-switches	0	0.000 K/sec
cycles	920017403076	2.599 GHz
instructions	1647964591190	1.79 insn per cycle

Table 5: Number of context-switches, cycles and instructions of the parallel program with 4294967295 steps and 8 threads

thread id	duration ns	start s	start µs	end s	end µs
0	2759478	1605831251	272137	1605831251	274898
1	2754479	1605831251	272167	1605831251	274922
2	1703501	1605831251	274986	1605831251	276690
3	2543449	1605831251	272581	1605831251	275125

Table 6: Thread beginning, end and execution times for 4 threads

thread id	duration ns	start s	start µs	end s	end µs
0	1007938	1605829900	134218	1605829900	135228
1	1010695	1605829900	134246	1605829900	135257
2	558970	1605829900	145630	1605829900	146190
3	983639	1605829900	135448	1605829900	136433
4	1046711	1605829900	136497	1605829900	137545
5	979718	1605829900	135421	1605829900	136401
6	1047623	1605829900	136453	1605829900	137501
7	4735693	1605829900	137682	1605829900	142419
8	914441	1605829900	136693	1605829900	137610
9	1009167	1605829900	137603	1605829900	138613
10	1002552	1605829900	138664	1605829900	139667
11	2921410	1605829900	139735	1605829900	142657
12	918140	1605829900	140580	1605829900	141498
13	887529	1605829900	138261	1605829900	139149
14	957395	1605829900	139212	1605829900	140170
15	885505	1605829900	140232	1605829900	141119
16	868846	1605829900	141175	1605829900	142045
17	918537	1605829900	141549	1605829900	142468
18	919792	1605829900	141683	1605829900	142603
19	920484	1605829900	142649	1605829900	143570
20	892724	1605829900	144261	1605829900	145154
21	919656	1605829900	143616	1605829900	144536
22	861202	1605829900	142470	1605829900	143331
23	805426	1605829900	143380	1605829900	144186
24	931418	1605829900	144622	1605829900	145554
25	920334	1605829900	142702	1605829900	143623
26	973111	1605829900	139661	1605829900	140635
27	969072	1605829900	137570	1605829900	138540
28	919451	1605829900	144650	1605829900	145570
29	927216	1605829900	143671	1605829900	144598
30	916924	1605829900	140685	1605829900	141603
31	994946	1605829900	138611	1605829900	139606

Table 7: Thread beginning, end and execution times for 32 threads