# Exercise 1: π approximation

Student: **Laura Balasso**

Academic year: 2018/19

The aim this exercise is to approximate the value of π with the following integral:

$$\pi = 4\int_0^1 \frac{1}{1+x^2}\,dx \approx 4\sum_{i=0}^{N-1} hf(x_{i+\frac{1}{2}})$$

Where N = $10^8$ is the number of rectangles used for the approximation, h = 1/N and f(x) = $\frac{1}{1+x^2}$
I first implemented a serial program, then I parallelized the code with OpenMP. Into a parallel region, created with `pragma omp parallel`, I divided the workload equally between the threads: I established the potion of the domain over which each thread works computing the above sum of N/num_threads rectangles. The final result is obtained by collecting the local results and summing them. This final step must be serialized to avoid race conditions. Since this can be performed in different ways, three versions of the code are provided: the first using an **atomic** section, the second using a **critical** section and the third using the **reduction**.

The aim is observing the scalability of the code, with respect to the number of threads used.
For this purpose, I measured the execution time of the parallel region with `omp_get_wtime()` for each of the three versions.
In the following table are reported the execution time in seconds for different numbers of threads of each version of the code. Each execution time is obtained as an average of five identical runs.

| N. threads: | 1 | 2 | 4 | 8 | 16 | 20 |
|---|---|---|---|---|---|---|
| **atomic** | 3.907936 | 1.99256 | 1.053063 | 0.567495 | 0.387807 | 0.402761 |
| **critical** | 3.930306 | 1.991822 | 1.018949 | 0.54066 | 0.304913 | 0.309338 |
| **reduction** | 3.898611 | 1.985442 | 1.029573 | 0.539467 | 0.315499 | 0.317304 |

*Table 1. Runtime in seconds of the three versions of the code, obtained as average of five runs.*

In the following plot we can see the graphical representation of the data above. We can see that the three curves are almost identical so there is no evident difference in terms of runtime between the three versions.

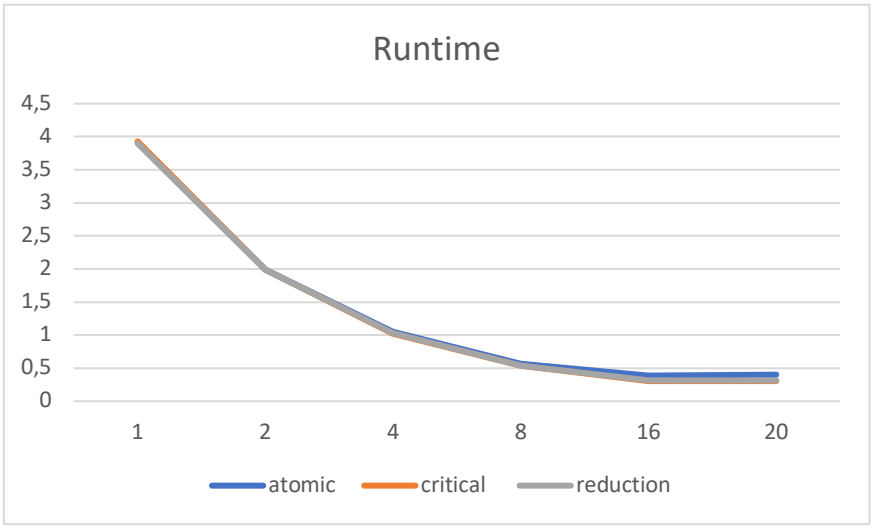


*Figure 1. Runtime against number of threads.*

The program scales up almost perfectly until 16 threads: each time the number of threads is doubled, the runtime is half of the previous. From 16 to 20 threads instead, the runtime decreases very slightly. This probably happens because the workload is not enough to have a benefit in splitting it among more than sixteen threads. When we try to do it, the threads are no more working well together, so the cost of synchronization exceeds the benefit of having a smaller workload per thread.
This behaviour can be seen more clearly in the following graph, in which is plotted the speedup, obtained as Speedup(n) = Runtime(1)/runtime(n).

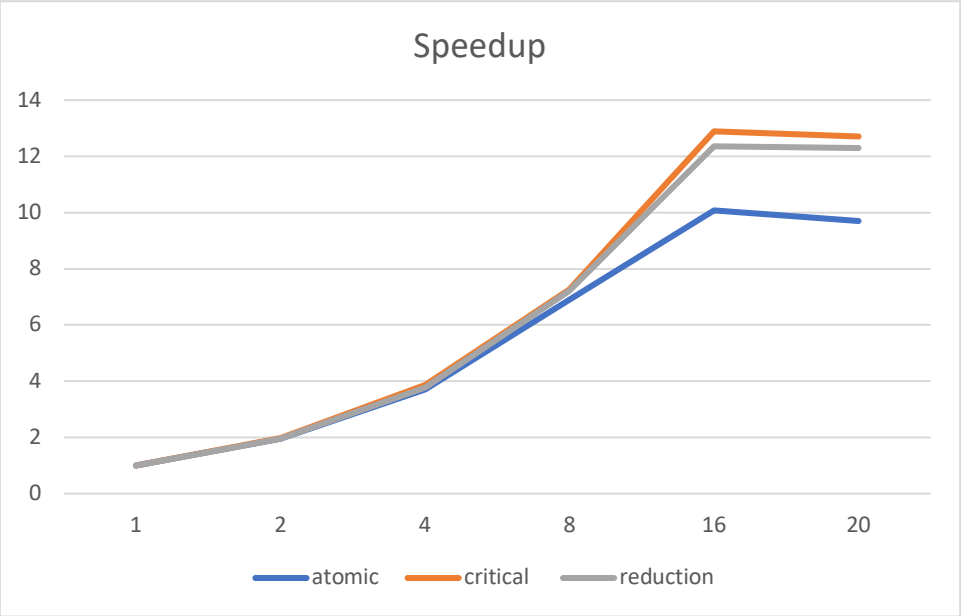


*Figure 2. Speedup against number of threads*

In this graph also emerges that the version of the code that uses the `atomic` region performs worse than the others.