Computation of π and scaling with OpenMP

William Beordo

May 18, 2022

Here I present a `c++` code in which the value of π is computed in two different ways:

- Through the numerical quadrature of the following integral

$$\int_0^1 \frac{4}{1+x^2} = \pi \ . \tag{0.1}$$

  Unlike the algorithm used in class and practical sessions, in which the Midpoint rule is adopted to approximate the integral, here I used the Trapezoidal rule.

- Through a Monte Carlo method in which random sampling is performed to compute a two-dimensional integration of a unit circle. The number of points that fall inside the circle gives an estimate of the value of π.

In our simple problem the latter method is much less precise than the first one, but it is particularly useful for higher-dimensional integrals when other approaches become impracticable. Moreover the Monte Carlo approach is clearly non-deterministic, therefore each realization provides a different outcome.

1 Parallelization with OpenMP

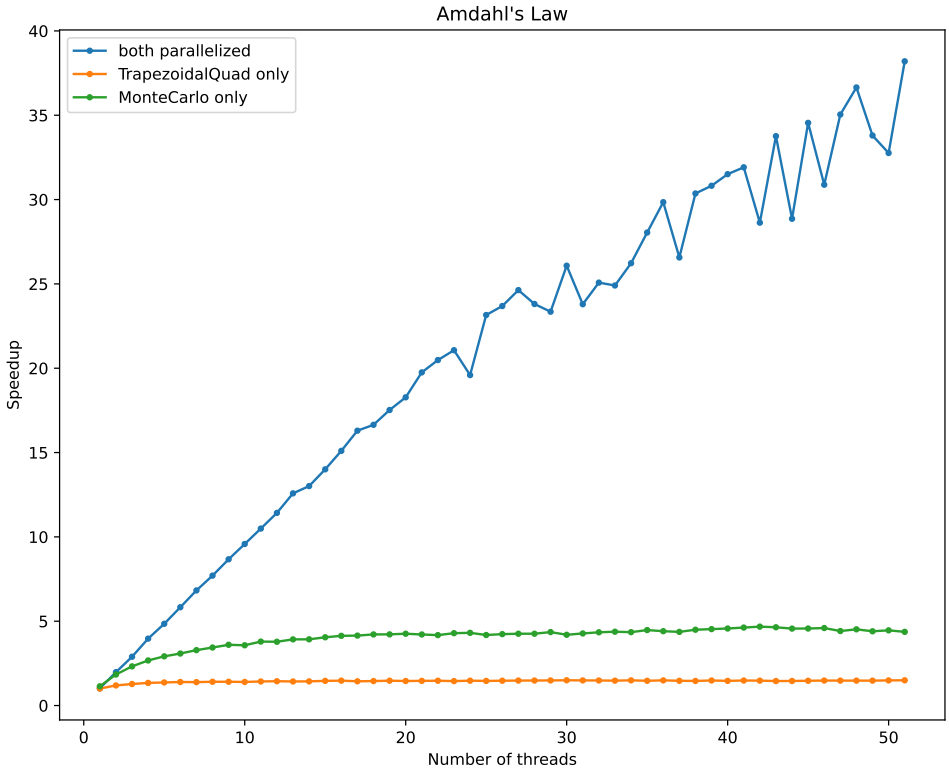
Both methods are called sequentially in the main code, so that we can observe how computing time changes when only one of them or both are parallelized.

For the numerical quadrature, the `#pragma omp parallel for` directive automatically splits the integration intervals among the spawned threads. Also, the `reduction(+:sum)` is required in order to avoid race conditions for the shared variable that computes the value of π.

For the Monte Carlo method, the `#pragma omp parallel` construct creates the thread pool. Then a different seed for the random number generation is assigned to each thread, in order to ensure that each thread has its own random number sequence and that the latter is different from that of other threads. Finally, as for the quadrature method, the `#pragma omp for reduction(+:count)` directive handles the splitting of the pairs of random numbers among the generated threads, protecting the shared variable that calculates the number of points dropped within the circle.

In Figure 1, we compare the speed-up factor (i.e. the gain in execution time between the fully sequential code and its parallelized version) with respect to the number of threads used. The speed-up is quite negligible (about a factor of 2) if we parallelize only the Trapezoidal rule for the numerical integration. It becomes more relevant, albeit still quite low, if only the Monte Carlo

Figure 1: Amdahl's law for the computation of π.



algorithm is parallelized: this is indeed the most time consuming part of the code. With these two cases we can observe the Amdahl's law: even with dozen of cores the speed-up cannot increase further, with the sequential part being the limiting factor. A significant gain in execution time is achieved for the fully parallel code. Here the speed-up scales linearly with the number of threads up to more than 20 cores; then the slope decreases, but with 51 cores the plateau still doesn't show up. This means that one could still achieve an additional gain.