Assignment 3 - Direct Simulation Monte Carlo and Parallelization

Christian Wiskott

University of Vienna — November 24, 2020

Task A: Calculate π using DSMC

Fig. 1 presents the results of the numerical computation of π using a Monte-Carlo simulation. The instructions of the assignment sheet were followed and resulted in the code provided with this report.

The following code-snippet shows the relevant part of the function to calculate π. The coordinates are randomly chosen (without replacement) from a given array x, which constitutes the chosen 2-D domain. Then the norm of the particle-vectors is computed to calculate the number of particles inside the circle.

With this result the numerical value of π is then computed, as well as the difference to the true value.

Sub-function of taskA.py to calculate π for number of particles n

```
particles_in_the_circle = 0
x_coord = np.random.choice(x, n, replace=False)
y_coord = np.random.choice(x, n, replace=False)
for i in range(n): # Iterates over all n particles
   if np.linalg.norm([x_coord[i], y_coord[i]]) <= radius:
   """ Checks if particle is in the circle """
     particles_in_the_circle += 1

pi_approx = particles_in_the_circle / n * 4
difference = pi_approx - np.pi
```

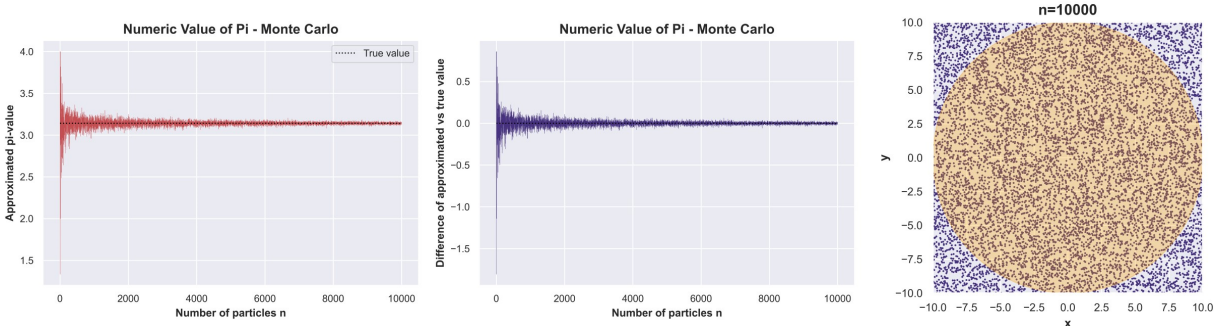

Figure 1: The left figure shows the convergence of the numerical value to the true value of π for increasing number of particles n up to M=10k. The middle plot shows the difference of the numerical value to the true value, which converges to 0. The right figure shows the particles inside- and outside of the circle for an intermediate step of the computation with n=10k.

Task B: Parallelize your code from Task A

In order to parallelize the code, it had to be adapted for this purpose and was implemented using the multiprocessing-modul Pool. The following code-snippet shows the implementation.

taskB.py

```
for proc in range(1, procs+1): # Uses 1,2,3,... cores
        start = datetime.datetime.now()
        p = Pool(proc) # Initializes processes
        pi = p.map(MC, range(1, M+1)) # Calc for all n
        # Finishes the process
        p.close()
        p.join()
        runtime = (datetime.datetime.now() - start)
```

Here the computation is carried out for ascending number of cores. The resulting values for π, as well as the execution times are recorded and displayed at the end of the calculation.

Table 1 shows the effect of the parallelization on the execution times and the respective speed-ups for different number of cores used during the calculation. This task is well suited for parallelization, as can be seen by the 2.99 - times speed-up using 4 cores.

To verify the convergence to the same result as in task A, fig. 2 shows the same plots again for the parallelized task for M=5k. Here it can be seen, that the convergence to the true value of π is comparable to that of task A.

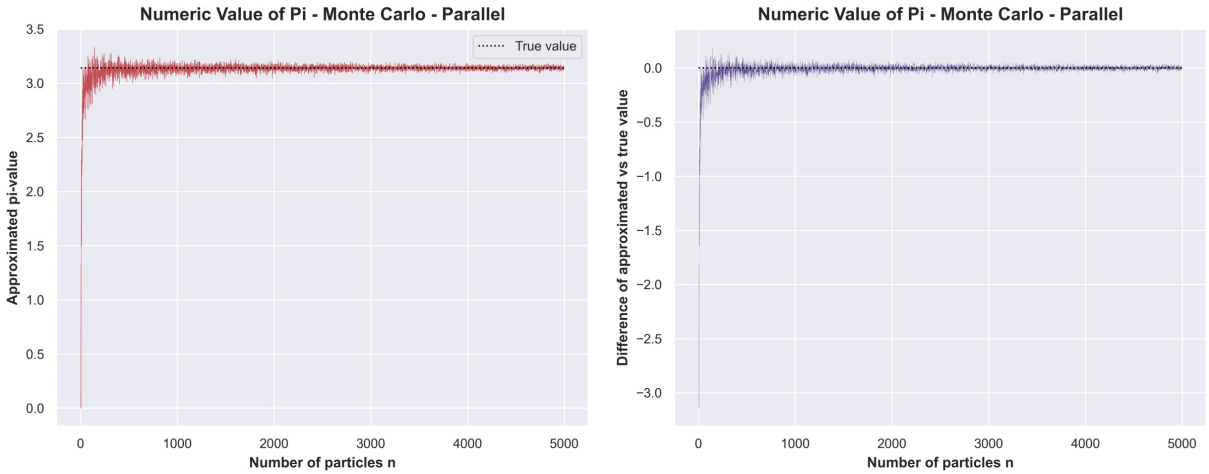

Figure 2: Shows the same plots as in task A, but the results of the parallel computation. Shows the same convergence behaviour with increasing number of particles.

# of cores	1	2	3	4
Execution Time [s]	518.04	286.92	193.36	173.13
Speed-Up	1	1.81	2.68	2.99

Table 1: Results of the parallelization using different numbers of cores with M=20k. The speed-up was calculated as the fraction of the execution time using only one core and the execution time using multiple cores. Using four cores resulted in the best performance with a speed-up of 2.99.