Jesse Goncalves

Dr. Kim

PHYS 3910

18 March 2019

Notes: Does Parallelization make Monte Carlo methods

more efficient in time than fourth-order Runge-Kutta?

In this project, I sought to experimentally investigate the question posed in the title: does the “embarrassingly” parallel nature of Monte Carlo (MC) methods mean that they can actually become more efficient in time for solving Initial Value Problems (IVPs) than the more popular fourth-order Runge-Kutta (RK4) method on large enough parallel and distributed hardware structures? I used python’s multiprocessing library to compare the performance of these two methods for solving a simple IVP on my four-core MacBook Air, and extrapolate based on these results to their performance on computer architectures with more computational power.

To solve an IVP using MC methods, we discretize the domain over which we hope to obtain a solution and generate random numbers from a uniform distribution over each discrete interval to approximate the value of the integral over that interval. This process is “embarrassingly parallel” because the value of these integrals can be approximated independently of each other, so it can easily be distributed to separate cores, and we can even further distribute the work of approximating these integrals and then combine results after their computation. The primary barrier to efficient parallelization is communication between processes, as this requires that they be performed in lock-step, one after another, and the communication itself adds to the computation time. Due to the independent nature of MC methods, we avoid these barriers, making it a perfect candidate for parallelization.

RK4 methods for solving IVPs, on the other hand, are not so easily parallelizable. We again discretize the domain, but the integral approximation we obtain over each interval generally depends on the approximation in the preceding interval. This means that these computations cannot be performed asynchronously, but must occur sequentially, making the method a poor candidate for parallelization. Thus, my hypothesis for this project was that MC methods could be made to perform better than RK4 methods to a comparable order of error on large enough parallel and distributed hardware structures.

To test my hypothesis experimentally, I wrote MC and RK4 python programs to solve a simple IVP using python’s multiprocessing library. This library provides limited functionality for parallelization, and I quickly discovered that its functions in the Pool class far outperformed its functions in the Process class for both MC and RK4. Thus, using the Pool class, I compared the performance of MC and RK4 methods for different numbers of processes. In order to compare these different methods, I used the same discretization for both and set the number of random points generated for each interval in the MC method in order to achieve the same order of error as RK4 (h4). I then ran both code 10 times at each number of processes (from one to 10) and averaged the results.

The results were unsurprising in that RK4 far outperformed MC in terms of raw computation times, but MC scaled far more efficiently than RK4 as the number of processes was increased. In fact, RK4 ran most efficiently with a single process. The performance of MC flattened at around four processes, corresponding to the number of cores on my computer—its performance should continue to scale on larger parallel and distributed structures. Thus, based on these results alone, it seems that MC would become more efficient than the serial implementation of RK4 on a large enough hardware structure. However, the generalizability of my results to further classes of IVPs, and to libraries or languages with greater parallelization functionality, is not entirely clear. Further optimization of my parallel MC and RK4 methods, translation to other libraries and languages, extension to other classes of IVPs, and further theoretical analysis of the question would undoubtedly provide a clearer answer to the question.