Aristotle University of Thessaloniki
Physics Department
MSc Computational Physics

SET 2:

Estimating π with *Monte Carlo* simulations
An `OpenMP` application

Subject: Computational Tools - Part 3
Professor: Nikolaos Stergioulas
Implemented by: Ioannis Stergakis
AEM: 4439

October 2024

Contents

List of Figures

1 Introduction

Parallel programming is quite a useful tool, especially when it comes to performing heavy computational tasks. There are several ways to implement parallel programming, but in this assignment we will use the simplest one: `Shared-Memory Programming`. In particular, we will use the `OpenMP` (Open Multiprocessing) environment, that is supported by the `C/C++` and `FORTRAN` languages.

Our goal is to perform a sufficient number of **Monte Carlo** simulations to estimate the irrational number π. Between simulations, we will change the total number of threads or the total number of points used by the algorithm that executes the Monte Carlo method. In this way, we will be able to study both the **speedup** resulting from the parallelization of the simulation process and the **rate of convergence** of the estimates to the actual value of π.

2 Methodology

2.1 Implementing the *Monte Carlo* method with `OpenMP` in a `C` script

As an initial step, we wrote the `pi_estimation_MC.c` script (included in the same directory with this report). First and foremost, we insert the necessary `C` libraries (headers `.h`): the basic `<stdio.h>` and `<stdlib.h>`, as well as the `<math.h>` header (for mathematical functions such as `pow`) and the `<time.h>` header (for time measurement). Notice, that the `OpenMP` tools and functions are also included via the header `<omp.h>`.

Moving on to the body of the `main` function, we have to make sure that the information about the total points and the threads to be used for the simulation, can be given arbitrarily by the user, during the typing of the execution command of the produced `.exe` file (we will discuss about this command later). To do so, we define two arguments for the `main` function. The first one is an **int** argument named `argc`, the value of which is the number of elements separated by space in the execution command. The second argument is the pointer `*argv` of a **char** list, containing as elements the elements of the typed execution command in a char format.

To be more specific, to compile the `pi_estimation_MC.c` script, we type the follwing command: '`gcc pi_estimation_MC.c -o pi_estimation_MC -fopenmp`', in a `Windows` terminal, or '`g++ pi_estimation_MC.c -o pi_estimation_MC.exe -fopenmp`', in a `Linux` terminal, to get the `pi_estimation_MC.exe` executable file. Notice that the `-fopenmp` command ensures that the executable is `OpenMP` ready. To run the executable we have to type the following command: '`pi_estimation_MC {number of points} {number of threads}`', in a `Windows` terminal, or '`./pi_estimation_MC {number of points} {number of threads}`', in a `Linux` terminal, ensuring prior the run that we are in the directory of the executable file. For example, if we want a simulation with 10^3 points running on 4 threads we have to type: `pi_estimation_MC 1000 4`, in a `Windows` terminal.

Thus, the element `argv[0]` will contain the name of the executable, i.e pi_estimation_MC,

while the element `argv[1]` will contain the number of points as '1000' and he element `argv[2]` the number of threads as '4'. To obtain, the actual values of these last two elements as float numbers, we use the command `atof` and then we typecast them to **long long int** values and store them to their respective variables (`total_points` and `num_threads`). The algorithm, also, gets the maximum number of physical threads, with the function: `omp_get_max_threads()` and sets the selected number of threads for the parallel task with the command: `omp_set_num_threads(num_threads)`. In the case, where the user forgets to type the number of points and/or the number of threads, the algorithm sets as default the number of points to 10^9 and/or the number of threads to the maximum physical possible, respectively.

Let's discuss, now, about the *Monte Carlo* method for the estimation of π. The logic behind this method is quite simple. We generate random points in the square 2D-plane formed by the boundaries [-1,1] in both the x and the y axis, i.e. points with random x and y coordinates in the interval [-1,1]. Then, we calculate the square of the radius of these points with the Pythagorean formula:

$$r_{sq} = r^2 = x^2 + y^2 \tag{1}$$

and count the points that lie within the circle $x^2 + y^2 = 1$, that is points with radius:

$$r \leq 1 \;\; or \;\; r_{sq} \leq 1 \tag{2}$$

The ratio of the points inside the circle to the total generated points, multiplied by 4 gives us an estimation of π:

$$\pi_{est} = 4 * \frac{points\ inside\ circle}{total\ points} \tag{3}$$

So, it is obvious we have to define some **double** variables, for the x and y coordinates (`x_points,y_points`) and the square of the radius (`radius_sq`), as well as a variable for the estimation π (`pi_estimation`) and the error of the estimation (`pi_error`). The counter of the points that will lie inside the circle $x^2 + y^2 = 1$ during the simulation should, also, be initialized to 0 (`in_circle_points`$= 0$), along with the initialization of the seed for the random number generator. For the last one, we use the `srand()` function and provide as seed the current time, to ensure the seed is different every time the executable is running. Finally, we define some variables (`start_time,end_time` and `cpu_time`) for the measurement of the execution time of the *Monte Carlo* simulation.

The simulation starts with the assignment of the current time to the variable `start_time`. The definition of the parallel region comes next, using the command `#pragma omp parallel`. Notice, that we don't use a default data scope (`default(none)`), that is we have to declare explicitly the scope of every variable that takes part in the parallel region. Thus, the variables `pi_estimation`, `pi_error` and `total_points` are declared as **shared** values (all threads have access to them), since their values are calculated prior or after (and they do not change during) the parallel region. On the other hand, the values of the variables `x_points`, `y_points` and `radius_sq` are generated randomly inside the parallel for loop and change after every iteration. So, each thread needs to have its own copy of the last 3

variables, in order to not interfere with the load of other threads, and these variables must be declared as **private** variables.

Regarding the parallel for loop, it is declared as a serial for loop, with an initial value of 0 and a final value of `total_points` for the loop index, but with the addition of the `#pragma omp for` command, the algorithm automatically allocates the work to the selected number of threads. Now, at the end of each iteration, the algorithm increases the number of points in the loop by 1 if the condition (2) is satisfied. Therefore, the value of the variable `in_circle_points` is essentially a sum, calculated separately from each thread, and whose final value will be obtained by adding up these separate values at the end of the parallel region. This is done, with the declaration of the `in_circle_points` as a **reduction+:** variable. For more about the variables scope and the work-sharing inside a parallel region see [1].

After, the parallel region we calculate the value of the π estimation (according to (3)), along with its error and terminate the measurement of the simulation's execution time. The algorithm ends with the print of the simulation results and the `return` 0 of the `main` function.

2.2 Parallel speedup investigation

For the investigation of the parallel speedup's behavior, we need the execution time data that are printed as output during the run of the executable `pi_estimation_MC.exe`. In the `Jupyter` notebook `Monte_Carlo_pi_analysis_1.ipynb` we offer 2 options in obtaining these data (see Section 1 in the notebook). The user can either take data from a live *Monte Carlo* session (that runs inside the notebook) using the `pi_estimation_parallel_1()` function or can insert data from a file using the `pi_estimation_insert()` function.

To be more specific, the function `pi_estimation_parallel_1()` takes as inputs the number of total a points and a list with different number of threads to be used. Then, utilizing the `subprocess` module, it compiles and runs the `pi_estimation_MC.c` script, as many times as the length of the threads list are. That is, we can perform as many simulation as many as the different numbers of threads are, calling the function only once. To obtain the output, the function uses the regular expression module (`re`) and searches for the execution time value (as measured in the `C` script), along with the value of pi estimation. The function, also, calculates the execution time (measured using the `Python's time` module) and returns a nested list, containing the list of the π estimations, a list of the `C` measured execution times and a list with of the `Python` measured execution.

On the other hand, the function `pi_estimation_insert()` allows the user to read `.csv` files and returns them as `pandas` matrices. This way, one can record the data from a parallel session in a `.csv` file and insert them in the notebook whenever its needed.

Now, for the analysis part, we defined two functions for the execution times (see Section 2 in the notebook). The function `exe_time_analysis_1()` calculates and shows the average difference between the `C` measured and the `Python` measured execution times and plots the **Times vs Number of Threads** diagram of a single session. On the contrary, the

function `exe_time_analysis_2()` offers the opportunity to compare the execution times of the **Exhaust** and the **Separate** sessions, by calculating again some average differences between the execution times of these different sessions and plotting the respective Times vs Number of Threads diagram.

As Exhaust, we mean the session were the user has selected a list of numbers for the threads, picking each number only once, that is the session is continuous and needs only one run of the `exe_time_analysis_1()` function. On the other hand, in the Separate session, the user gives the same numbers for the threads, but only one at a time. So, after one simulation the session stops and the user has to give the next number of threads and rerun the function `exe_time_analysis_1()`. This allows the processor to relax a bit between the simulations and might have an impact on its performance.

Moving on, to the analysis of the parallel speedup (see Section 3 in the notebook), we have again two functions: the `par_speed_analysis_1()` and the `par_speed_analysis_2()` functions, providing a comparison between `C` and `Python` measured speedup and an overall comparison between two sessions (Exhaust and Separate), respectively. During the operation of both functions, the experimental (resulted from the simulations) speedup in the latency, that occurs from the parallelization of the *Monte Carlo* simulation, is calculated, using the formula:

$$S_{exper.}(N_{threads}) = \frac{E.T.(N_{least\ threads})}{E.T.(N_{threads})} \tag{4}$$

where $E.T.(N_{least\ threads})$ is the execution time of the simulation, when the least number of threads (of a session) is used and $E.T.(N_{threads})$ the execution time of the simulation, when a $N_{threads}$ number of threads is used. For example, for a session with a list $[1, 2, 4, 8]$ for the number of threads to be used, the resulted experimental speedups will be: $S_{exper.}(2) = \frac{E.T.(1)}{E.T.(2)}$, $S_{exper.}(4) = \frac{E.T.(1)}{E.T.(4)}$ and $S_{exper.}(8) = \frac{E.T.(1)}{E.T.(8)}$.

However, *Gene Amdahl* proposed a theoretical formula, that calculates the speedup in the latency of the execution of a task with a certain workload, that can be expected of a system whose resources are improved. This formula is known as Amdahl's law and is shown below:

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{5}$$

where $S_{latency}$ is the theoretical speedup of the execution of the whole task, s is the speedup of the part of the task that benefits from improved system resources (in our study we use the number of threads as the value of s) and p is the proportion of execution time that the part benefiting from improved resources originally occupied (in our study the original resources are the least used threads of a session).

In both the `par_speed_analysis_1()` function and the `par_speed_analysis_2()` function, the experimental speedup values, obtained using (4), are plotted against the number of threads. Furthermore, a fitting on the experimental points is performed, by utilizing the `curve_fit` of the `SciPy` module and putting in as fitting function, the Amdahl's law formula of (5). This way, the parameter p is calculated and we are in the position to make

predictions about the improvements in the latency, when a much larger amount of threads is used, for instance 10000 threads.

2.3 Convergence rate investigation

The convergence rate investigation, requires a different approach for the fitting. In the second `Jupyter` notebook we created, the `Monte_Carlo_pi_analysis_2.ipynb`, we define the function `pi_estimation_parallel_2()`. Its operation is similar to the operation of the `pi_estimation_parallel_1()` function, but with a twist. This time only one number of threads can be used, while we can perform *Monte Carlo* simulations for different total random generated points, in a single run of `pi_estimation_parallel_2()`. Moreover, since we are interested in the errors of π's estimations, after every simulation we obtain the estimated value of π, from the output of the executable file (using again the regular expression module) and calculate the true percent relative error e_t of the estimation. The formula for the last one is:

$$e_t = 100 * \left| \frac{\pi_{true} - \pi_{est}}{\pi_{true}} \right| \% \tag{6}$$

where π_{true} is the actual value of π (in our study we use the value given by the `Numpy` module: `np.pi`) and π_{est} is the estimation of π. Of course, we can insert the errors' data from a recorded session, by opening a `.csv` file, using the `pi_estimation_insert()` function, that we defined in this notebook too.

For the analysis part, we defined three functions, in order to investigate the existence of some kind law of power in the convergence rate (see Section 2 in the notebook). Using the `convergence_analysis_1()` the user can plot the values of the error e_t against the total number of random points generated, in a 'log-log' scale diagram and experiment with different exponents for the fitting curve: $y = x^{exponent}$. This way, one can find the fitting curve that best matches with the slope of the experimental data, since in the 'log-log' scale the expression of the fitting curve becomes a linear expression: $\log y = exponent * \log x$.

Since the optimal exponent is found, the determination of the incident of the linear expression comes next. Starting from the general expression of the law of power: $y = a * x^b$, in the 'log-log' scale we get the fitting curve: $\log(y) = \log(a) + b * \log(x)$. Therefore, by changing the value of a, one can adjust the overall y-position of the fitting curve, as the incident $\log a$ changes. We aim at a curve, that has experimental data on both sides (higher and lower) to ensure the minimum distance from them. This correction of the incident of the fitting curve, is implemented with the operation of the `convergence_analysis_2()` function, where one can try different values of the a factor.

Finally, to get a more accurate and justified answer for the best combination of the a and b parameters, we have to make the diagnostic graph of the fitting curve (or curves). This graph includes the normalized residuals in prediction vs the predicted values plot, as obtained from a single fit, along with the bounds of the 95% confidence interval of the standard normal distribution $N(0, 1)$ (-1.96 and 1.96).

At this point, we have to be more specific about the calculation of these predictions and

their residuals. Since, we assume the error in π estimation data follow the non-linear $y = a * x^b$ behavior, we can not perform a direct fit on them, rather we use the transformation: $Y = log(y) = log(errors)$ and $X = log(x) = log(points)$ and perform a fit on the new X and Y data. This is, because the new data are expected to have linear behavior: $Y = \log(a) + b * X$. Thus, in order to get the predicted values of the fit, we can simply calculate the values of the errors, as predicted using the $y = a * x^b$ function and then take their decimal (or natural) logarithm. As for the residuals in the predictions, we simply calculate the difference between the logarithm of the predicted error values and the logarithm of the actual error values (the ones from the *Monte Carlo* simulations). Last, to normalize the residuals, we divide them by their standard deviation. All the aforementioned, are included in the operation of the `convergence_analysis_3()` function.

3 Results and Discussion

We ran the *Monte Carlo* simulations in a `Windows` laptop with the `Intel Core Ultra 185H` processor. This processor has 6 performance cores (i.e. 2 threads per core) and 10 efficient cores (i.e. 1 thread per core) resulting in a total of 22 physical threads. So, it is of particular interest to see its performance in the `OpenMP` environment implementation.

3.1 Parallel speedup results

For the analysis of the execution times (E.T.) and the parallel speedup (P.S.), we used the `pi_estimation_parallel_1()` function with 10^10 random generated points and the following list of the number of threads: $[1, 2, 4, 8, 16, 32, 64]$. We performed both an Exhaust and a Separate session and saved the data in the `MC_Sim1_exhaust.csv` and `MC_Sim1_sep.csv`, respectively. As we can see in Fig.1, the difference between the `C` and `Python` measured E.T. is negligible, at an average of 0.1 second, for both sessions. On the contrary, there is a difference in the E.T. between the Exhaust and Separate sessions, with the Separate session being faster at an average of approximately 14 seconds. However, the main information, one can get from Fig.1, is the significant reduce of the E.T. as the number of threads used for the parallel task is increasing.

Indeed, we start from over 300 seconds E.T. when only 1 thread is used and end with E.T. around 50 or less seconds, when more than 10 threads are used. This acceleration in the E.T. can be more clear, by looking at Fig.2. The speedup of the Separate session is bigger, resulting to approximately 90.35% as the parallel portion that benefits from the increase in the number of threads, while the Exhaust session speedup results to 86.7. The difference is not much, but must be taken into account. As for the theoretical speedup limit when 10000 threads (cores) are used, the Separate session analysis gives approximately 10.3 times faster execution, while the Exhaust session analysis gives approximately 7.5 times faster execution. For more details see the Section 4.1 in the `Monte_Carlo_pi_analysis_1.ipynb` notebook. In Section 4.2, one can perform a live Exhaust session and see the respective results.

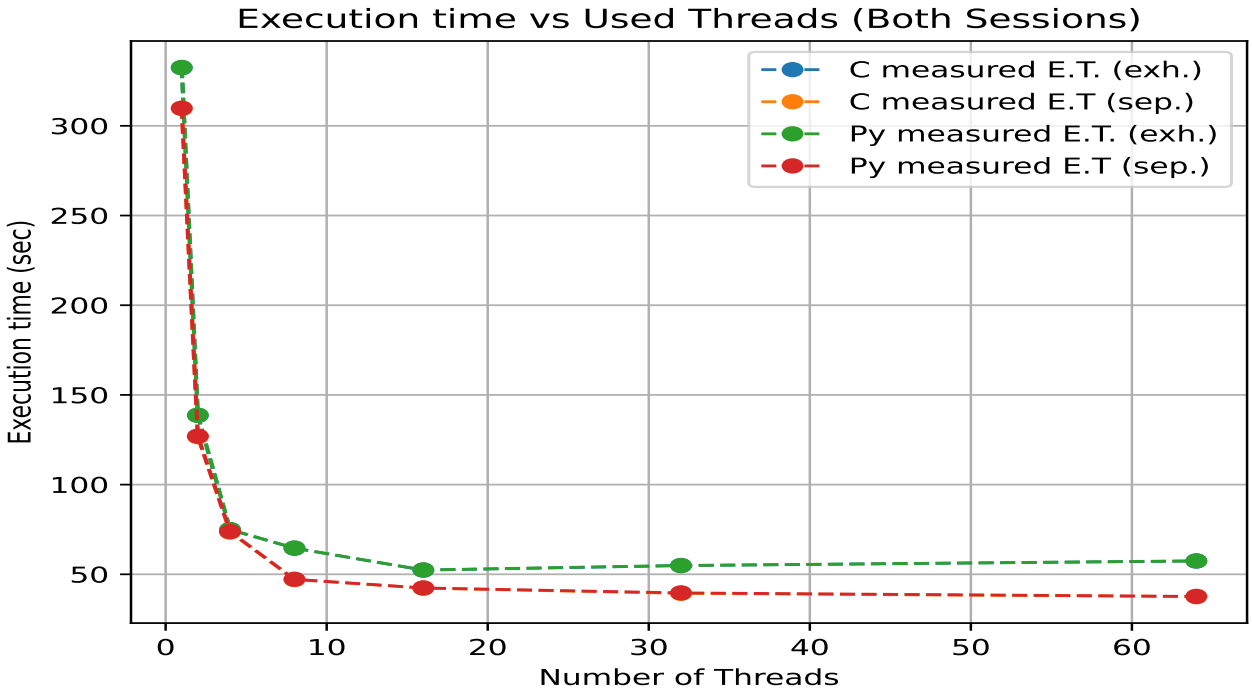

Figure 1: Execution time (E.T.) vs number of threads plots, a) C measured E.T. from Exhaust session (blue), b) C measured E.T. from Separate session (orange), c) Python measured E.T. from Exhaust session (green), d) Python measured E.T. from Separate session (red)

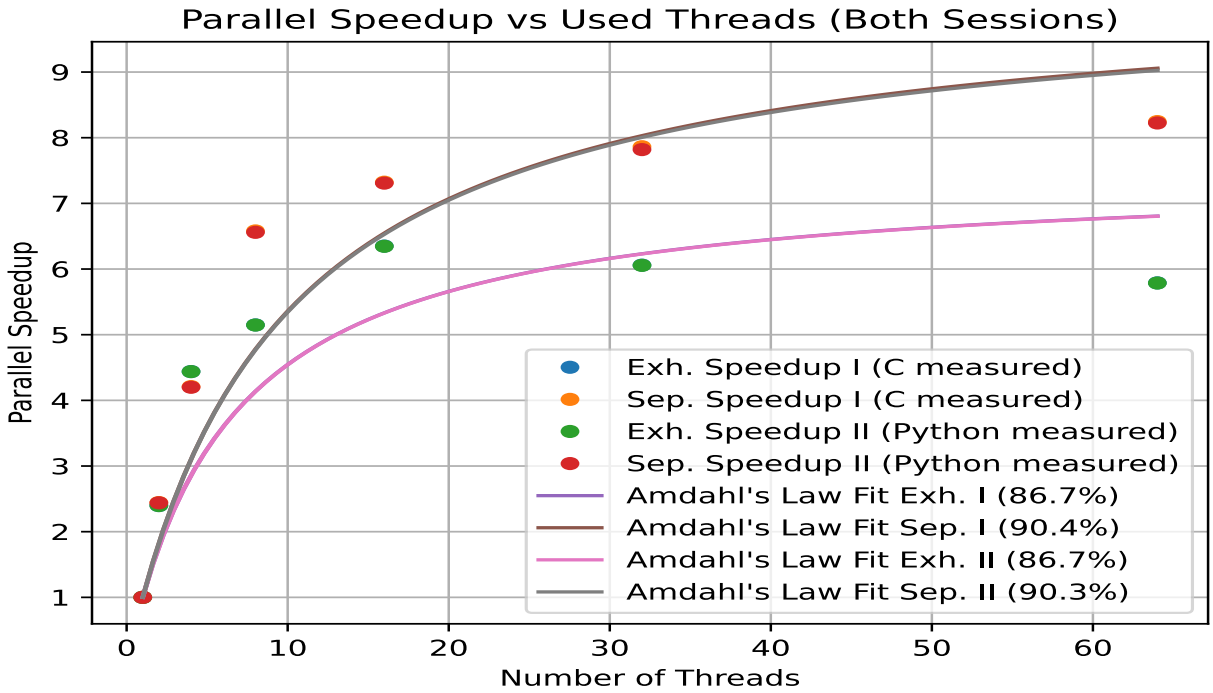

Figure 2: Parallel speedup (P.S.) vs number of threads plots and their fit using Amdahl's law, a) C measured P.S. from Exhaust session (blue), b) C measured P.S. from Separate session (orange), c) Python measured P.S. from Exhaust session (green), d) Python measured P.S. from Separate session (red), e) Fit on the C measured P.S. of Exhaust session (purple), f) Fit on the C measured P.S. of Separate session (brown), g) Fit on the Python measured P.S. of Exhaust session (pink), h) Fit on the Python measured P.S. of Separate session (grey). The percentages in the parentheses are the proportion of the E.T. when 1 thread was used

3.2 Convergence rate results

References

[1] ΝΙΚΟΣ ΤΡΥΦΩΝΙΔΗΣ. ΕΡΓΑΛΕΙΑ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ: ΕΙΣΑΓΩΓΗ ΣΤΟΝ ΠΑΡΑΛΛΗΛΟ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ ΜΕ OpenMP.