Sergi Aliaga

Northeastern University

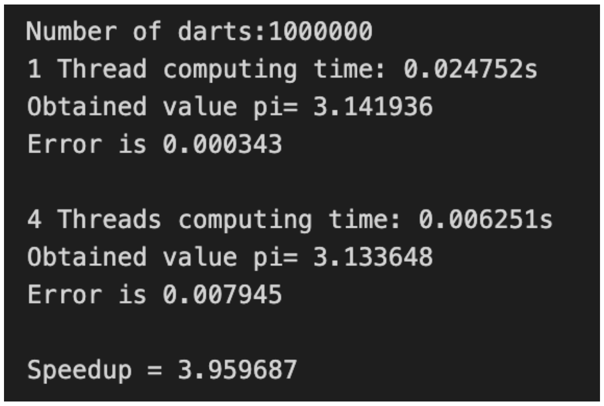
Homework 2

Question 2

In this problem you will develop two different implementations of the computation of pi numerically using pthreads and OpenMP. Then you will compare them in terms of scalability.

*a. Evaluate the speedup that you achieve by using pthreads and multiple cores. You are free to use as many threads as you like. The program should take two input parameters, the number of threads and the number of "darts" thrown. Your program should print out the time required to compute pi and the final value of **pi**. Make sure to document the system you are running on and the number of hardware threads available.*

The implemented version has been tested with 1e6 darts and 4 threads when multi-threading. The obtained output is reported in Fig. 1. The CPU utilized is an Apple M1 Max, 10 cores (8 performance at 3.22GHz and 2 efficiency at 2.064GHz), 64Gb of unified memory and running macOS Monterey version 12.1.

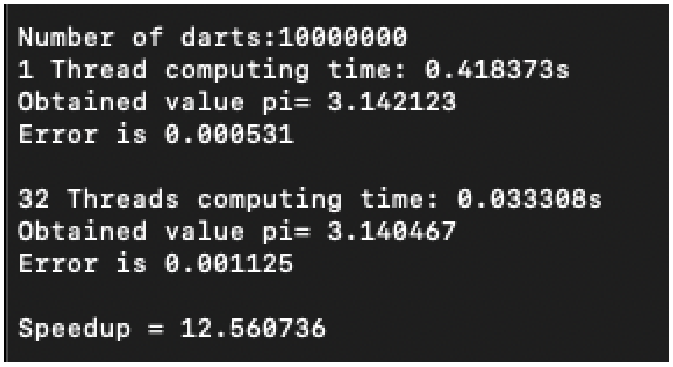


```
Number of darts:1000000
1 Thread computing time: 0.024752s
Obtained value pi= 3.141936
Error is 0.000343

4 Threads computing time: 0.006251s
Obtained value pi= 3.133648
Error is 0.007945

Speedup = 3.959687
```

Figure 1: Monte Carlo simulation results

From the reported results we obtain a considerable speedup when multithreading. However, there seems to be a larger error when parallel computing is considered.

For the comparison with the OpenMP implementation in the next section, the code has been executed at the COE system node with 24 CPUS with 2 threads per core (hyperthreading), resulting in a total of 48 hardware threads available. Each CPU has a clock rate of 2.67GHz. Since the system is much more powerful, for this part we increased the number of "darts" to **1e7** and the number of threads to **32**. The results are reported in Fig. 2. As expected, the parallel performance is much superior to the serial one, but still and order of magnitude less precise for the same number of throws.



```
Number of darts:10000000
1 Thread computing time: 0.418373s
Obtained value pi= 3.142123
Error is 0.000531

32 Threads computing time: 0.033308s
Obtained value pi= 3.140467
Error is 0.001125

Speedup = 12.560736
```

Figure 2: Monte Carlo simulation results in the Discovery Cluster

It is important to remark that considerable effort has been put into understanding the behavior of the standard random generator for C. After the first tests of the implementation, we observed a worse performance with the parallel implementation than with serial, which was not expected. The problem was related to the fact that the random number generation was being implemented with the *"rand()"* function, which is not thread-safe since it uses a hidden state variable that is shared and modified on each call. This caused the parallel implementation to become serial due to the modification of this shared state. The solution has been to implement the random number generation with *"rand_r(unsigned int *seedp)"* , which has a similar behavior but uses the variable pointed by *seedp* to store the state of the generator. This way, creating a private variable *seedp* we were able to obtain the expected improve in performance.

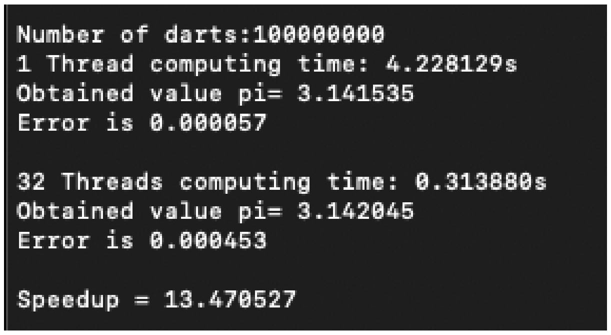
b. Now develop the same program using OpenMP. Repeat all of the steps requested in part a.).

For this section, an implementation of the Monte Carlo simulation with OpenMP has been written and included in file "Q2b.c". The main changes are in the "multi_thread()" function, where a pragma for parallel loops has been specified, with the same amount of threads as in the previous section:

"#pragma omp parallel for reduction(+: circle) private(x,y,distance, seed, i) num_threads(NTHREADS)"

The loop immediately after the pragma has been identified as a reduction operation since we add the results of all the threads to variable *"inc_circle"*.

The results reported are very similar to the previous section, even slightly better, which makes sense given that OpenMP is built up on pthreads. The main advantage however is a considerable simplification of the code.



```
Number of darts:100000000
1 Thread computing time: 4.228129s
Obtained value pi= 3.141535
Error is 0.000057

32 Threads computing time: 0.313880s
Obtained value pi= 3.142045
Error is 0.000453

Speedup = 13.470527
```

Figure 3: Results of the OpenMP implementation

c. *Now compare the two implementations in terms of strong and weak scaling, where the number of Monte Carlo simulations (i.e., "darts" thrown) is used to assess weak scaling.*

For the scalability analysis of the implementations, we fixed 1e7 darts and 1 thread as the baseline. From there, we evaluated strong scaling by increasing the number of threads and keeping the number of "darts" constant. On the other hand, for the weak scaling analysis, we incremented the number of "darts" according to the increment in the number of threads.

The results obtained are reported in Fig. 4 and Fig. 5. As a first observation, we report little to none differences between the Pthread and OpneMP implementations, except for the strong scaling with 32 threads.

For the strong analysis, we report a linear increase in performance in lower number of threads. However, after 8 threads, performance is reduced, and stabilize around a speed up value of 12. This might be the consequence of Ahmdal's Law, where we are reaching the limit in parallelization of the algorithm and no further speedup is allowed due to a minimum sequential part in the code.

In the weak scaling analysis we also see Gustafson's Law in action up to 8 threads, where speedup is kept constant to 1 while the input size (number of darts) is scaled accordingly (2e7, 4e7, 8e7,… for 2, 4, 8,… threads respectively). However, we see a reduction in performance when 16 and 32 threads are considered. This might be due to a nonlinear scalability, that is, up from 8 threads, for a doubling of processing power there is less than a double increment in input data size available for achieving the same performance.

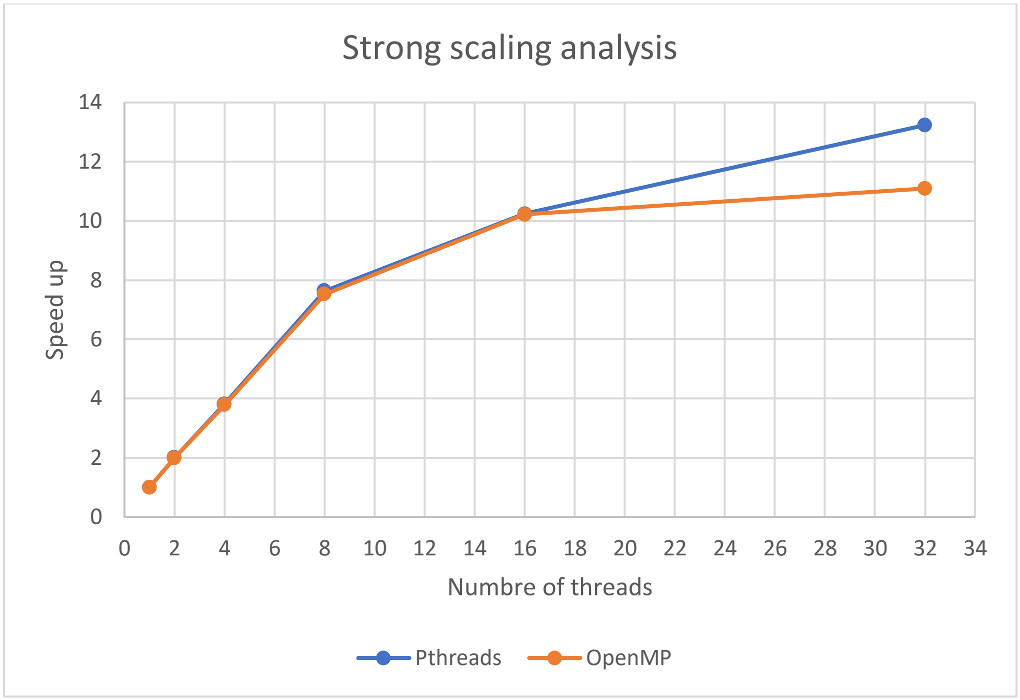


Figure 4: Strong scaling analysis

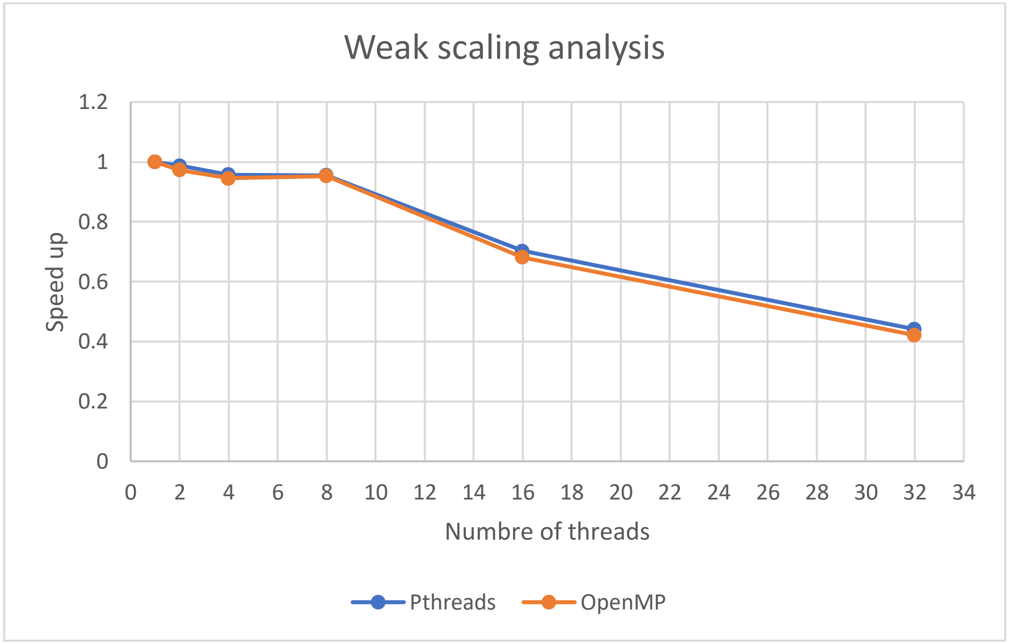


Figure 5: Weak scaling analysis