Monte Carlo Integration with OpenMP

Raffaele Ancarola, Louis Jaugey
raffaele.ancarola@epfl.ch, louis.jaugey@epfl.ch

October 6, 2020

1 Introduction

In this paper, we will use Monte-Carlo method to first calculate an approximation of π and then, to develop a general integrator for one dimensional functions. The computation will be executed in parallel on the CPU, using OpenMP API. The performance changes resulting from the number of threads/cores used will be studied and discussed.

2 Method

As explained above, the Monte-Carlo method is used to approximate the expected result. The main idea behind this numerical method is to generate many random sample numbers which are then used for the approximation.

Approximation of π To calculate π, one can generate random floating point values x and y between 0 and 1. If the Euclidean distance $d = \|x^2 + y^2\|$ is smaller than one, then the point is inside the circle. Let C be the number of point in the circle and T the total of points. Then,

$$\pi \approx \frac{4C}{T} \tag{1}$$

where the factor 4 comes from the fact that we approximate only the area of a quarter of a circle.

Approximation of a 1D integral The integral of a general function can be (badly) approximated by

$$\int_b^a f(x)\mathrm{d}x \approx (b-a)f(x) \tag{2}$$

Though this is a bad approximation, the average of these values over randomly generate x's can be good and this what will be used for the final approximation.

Parallelisation For both of these computations, the random floating point variables used to compute one step of the approximation are generated in for loops. The result of the step is added to a general variable that we will call C. In order to take advantage of the multiple threads, the for loop is split in n_{tr} sub-loops, each executed in one of the n_{tr} threads. The sub-loops' iterator is initialised to the thread id (between 0 and $n_{tr}-1$) and is incremented by n_{tr} at each iteration. If C is kept as a shared variable, it will encounter race conditions between threads. A possible solution would be to increment C atomically, however this slows the execution down. The solution used here is to have a C_p which is private to each thread and a shared C_s in which the C_p's are combined, atomically.

Performance The operations dominating the execution time are clearly the parallelised for loop block (that we call `for_loop` phase) and the computation of the sum of each thread contribution (`final_count` phase). Both phases are present in `pi.c` and `integral.c`.
The `final_count` phase takes non-negligible execution time because, although it's performed at the end of every separate thread, it involves synchronization, meaning that it is executed only once all threads are done.
In `for_loop`, a set of constant time operations are executed N times. Dividing the task execution in n_{tr} independent threads reduces the time complexity to $\Theta(N/n_{tr})$.
The initialization phase, for large N, can be neglected.

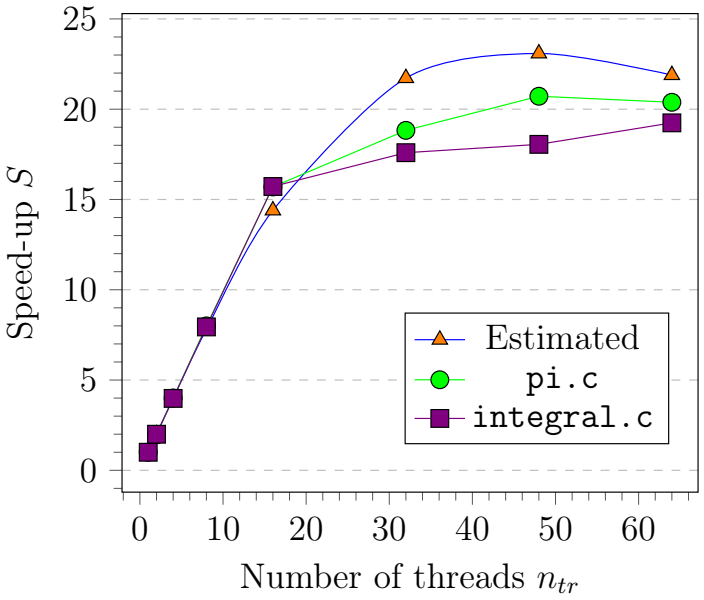

Figure 1: Speed-up as function of the number of running threads N_{tr} evaluated for the theoretical case, the program `pi.c` and the program `integral.c`. Both are computed on the `scitas EPFL cluster` [1] using an `sbatch` reservation task.

Parallelisation speedup The only part of the program which can be sped up is `for_loop`. Let p be the fraction of the total execution time spent on the `for_loop` phase. The total execution time depends on the number of samples N and an additional synchronisation time of `final_count`. This phase is supposed to be linear with respect to n_{th} and is proportional to $\beta \cdot (n_{th}-1)$, where β is the proportionality coefficient, called the *waiting factor*. Hence, p can be estimated by:

$$p(n_{tr}, N) = \frac{N}{\beta \cdot (n_{tr} - 1) + N} \tag{3}$$

Using then the Amdahl's law and considering that `for_loop` can be speed-up of n_{tr} times, we are able to retreive the total speed-up:

$$S = \frac{1}{1 - p + \frac{p}{n_{tr}}} \tag{4}$$

The graph in figure 1 and the table 1 show that the speed-up obtained running the programs on the `scitas EPFL cluster`, is well described by the Amdahl's law and the estimation of the theoretical case is well chosen. Furthermore, it comes out that $\beta \approx 50000$.

	`pi.c`		`integral.c`	
N_{tr}	T [s]	S	T [s]	S
1	2.14 ± 0.02	1	1.06 ± 0.01	1
2	1.09 ± 0.05	1.96918	0.532 ± 0.001	1.99839
4	0.534 ± 0.010	4.00415	0.267 ± 0.002	3.98269
8	0.267 ± 0.001	7.99782	0.134 ± 0.001	7.94212
16	0.136 ± 0.005	15.6962	0.0676 ± 0.0005	15.7183
32	0.114 ± 0.005	18.8279	0.0604 ± 0.0051	17.5911
48	0.103 ± 0.009	20.7175	0.0589 ± 0.0062	18.0524
64	0.105 ± 0.009	20.3829	0.0552 ± 0.0026	19.2437

Table 1: Results of elapsed time T and speed-up S as function of the number of running threads n_{th} evaluated on the `scitas EPFL cluster`. Times were taken evaluating the mean over 10 `slurm` outputs. NB: also S is subject to standard deviation but it's so small that it can be negletted. Sources: python script `graphs/gengraphs.py` and slurms raw data in `graphs/slurms/` (see [2]).

Documentation and sources

[1] https://scitas-data.epfl.ch/confluence/exportword?pageId=17564177

[2] https://c4science.ch/source/multiproc/browse/master/A1/