Assignment 1
Introduction to multiprocessor architecture

Simon Maulini - 248115
Arthur Vernet - 245828

October 8, 2018

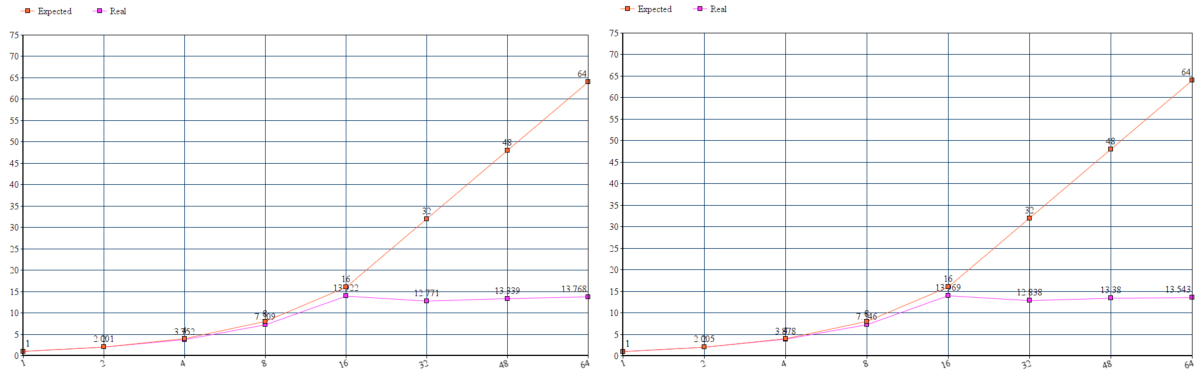
Part 1: *pi.c*

Using Monte-Carlo Integration, the computation of π can be done by randomly generating points inside the unit square and using the following approximation: $\pi = 4 * \frac{I}{N}$, where I is the number of points inside the quarter-circle, and N the total number of points generated. The (x, y) coordinates of the points were therefore comprised between $[0, 1]$. A point is considered being inside the quarter circle if $x^2 + y^2 \leq 1$.
Points are generated and their position checked in parallel inside a *OpenMP* for loop, which automatically split the number of iterations between a previously defined number of different threads. Using the *OpenMP reduction* operator, a previously initialized global variable is incremented in parallel every time a point falls inside the quarter-circle. In reality, each thread will create its own copy of the variable and increment it locally, to then combine them together once the threads join, i.e. at the end of the for loop.[1] Note that, before entering the for loop, each thread instances its own random number generator. This way, they don't fight over the use of a single global random number generator, which decreases the program efficiency by a lot.
The branch condition - determining whether a point is comprised in the quarter-circle - inside the for loop is the operation that dominates the execution time of the parallel phase.
The number of points to generate affects the number of iterations of the for loop, and therefore the number of time the branch condition is checked. With N, the number of points to generate, and assuming the branch condition is a $\mathcal{O}(1)$ operation, the program runs in $\mathcal{O}(N)$.

Part 2: *integral.c*

Using Monte-Carlo Integration, a numerical estimation of the integration of an arbitrary function can be done by generating values inside its integration domain and compute: $\frac{1}{N} * \sum_{i=1}^{N} f(x) * (b - a)$, where x is a randomly generated value inside the integration domain, f the arbitrary function, $[a, b]$ the integration domain, and N the total number of points.
The program structure is similar to the previous exercise, i.e. a *OpenMP* for loop coupled with the *OpenMP reduction* operator. Values are randomly generated in parallel to then increment by $f(x) * (b - a)$ a global variable.
The previous multiplication is the operation that dominates the execution time in the parallel phase.
Again, the program runs in $\mathcal{O}(N)$ for similar reasons as before.

[1] http://pages.tacc.utexas.edu/ eijkhout/pcse/html/omp-reduction.html



(a) *pi.c* (b) *integral.c*

Figure 1: Speedup comparisons for both programs in function of the number of threads. In orange, the expectations. In purple, the reality.

Runtime analysis

When ignoring hardware limitations, one would expect a linear speedup proportional to the number of threads for both programs as they're similar.

When using the SCITAS cluster and setting the argument *#SBATCH –cpus-per-task* to 1, there is no speedup in the execution time of the programs even when increasing the number of threads through the program arguments. However, when doing it locally, there is a significant speedup as expected. Therefore, *#SBATCH –cpus-per-task* had to be set to 16 (unfortunately the *sbatch* command doesn't accept higher numbers) to get the desired speedup.

pi.c - 1000000000 iterations								
Threads	1	2	4	8	16	32	48	64
Runtime	14.98s	7.486s	3.992s	2.078s	1.076s	1.173s	1.123s	1.088s
Expected speedup	1	2	4	8	16	32	48	64
Real speedup	1	2.001	3.752	7.209	13.922	12.771	13.339	13.768
integral.c - 1000000000 iterations with $f(x) = x$ and $[a, b] = [5, 9]$								
Threads	1	2	4	8	16	32	48	64
Runtime	8.844s	4.431s	2.291s	1.226s	0.636s	0.692s	0.664s	0.656s
Expected speedup	1	2	4	8	16	32	48	64
Real speedup	1	2.005	3.878	7.246	13.969	12.838	13.380	13.543

Table 1: Speedup comparisons for both programs when ran on the cluster.

As it can be seen, the estimation made was quite right until the programs use 16 threads or more. After this point, the speedup stays stuck around 13.5. As it is the cluster seems to only use 16 different cores, therefore it is hard to get increased performance with more than 16 threads. It actually seems normal and logical given the impossibility to set *#SBATCH –cpus-per-task* to values bigger than 16.