Part 1 - Pi

1. Code description

a. For this program we used the MCI technique, which approximates the area underneath a curve. We separated the program into three parts, two sequentials and one parallel. We parallelized the initialization of the random generator (l.51 + l.64), and the for-loop (l.54 - l.62). The reason we chose to parallelize `init_rand()` is that if we didn't, then all threads would have the same generator and therefore the precision of the computation would not be as good. We parallelized the for-loop since the computation of each loop is independent from one another. The only dependence between the loop instances is `countIn`, which we compute in a critical way in the end of the parallel part, to avoid having read and write conflicts. The two sequentials parts are the initialisation and settings of variables (l.40 - l.46), and the final computation (l.70 - l.72). There is no interest and no viable way to make those sections parallel.

b. In the first sequential part, the `omp_set_num_threads` function is the most time consuming one. In the parallel for-loop, after multiple tests it seems to be the generation of random numbers that takes the most time, but the multiplication is not far behind. Computing `pi` is obviously the longest operation of the second and last sequential section.

c. In both sequentials parts, each instruction is only executed once, and thus both sections are executed in constant time O(1). The parallel part will be executed `num_threads` times, with the for-loop within repeated `chunk` times, which means the execution time of the parallel part is O(`chunk`) = O(`samples`/`num_threads`). As we chose `samples` to be equal to 10^8, and `num_threads` to be between 1 and 64, we estimate the parallelized fraction to be 99% of the total program.

d. Using Amdahl's law : $$Speedup = \frac{1}{\frac{Fraction_{enhanced}}{Speedup_{enhanced}} + (1 - Fraction_{enhanced})}$$

We assume that $Speedup_{enhanced}$ increases linearly with the number of threads, which is a reasonable assumption since each thread runs independently from the others. We estimated in 1.1.c. that $Fraction_{enhanced}$ = 0.99. We can then estimate the speedup for the whole program, which can be seen in Figure 1.1. The predicted speed up is optimistic since we didn't take everything into account, for example the cost of parallelisation, or the potential waiting time of a thread because of the critical instruction. Moreover, the workload grows with the number of threads.

2. Execution time and observed speedup

Using Scitas, we measured the running time of pi.c 8 times, with the numbers of threads changing, `samples` always being equal to 10^8 and with 28 CPUs per task. We plotted the results along with the computations made in 1.1.d. in the Figure 1.1, Figure 1.2 and Figure 1.3.

PI

Number of threads	Measured Time	Measured Speed up	Expected Time	Expected Speed up
1	2,137	1,000	2,137	1,000
2	1,071	1,995	1,079	1,980
4	0,536	3,987	0,550	3,883
8	0,268	7,974	0,286	7,477
16	0,138	15,486	0,154	13,913
32	0,135	15,830	0,087	24,427
48	0,104	20,548	0,065	32,653
64	0,107	19,972	0,054	39,264

Figure 1.1

Figure 1.2

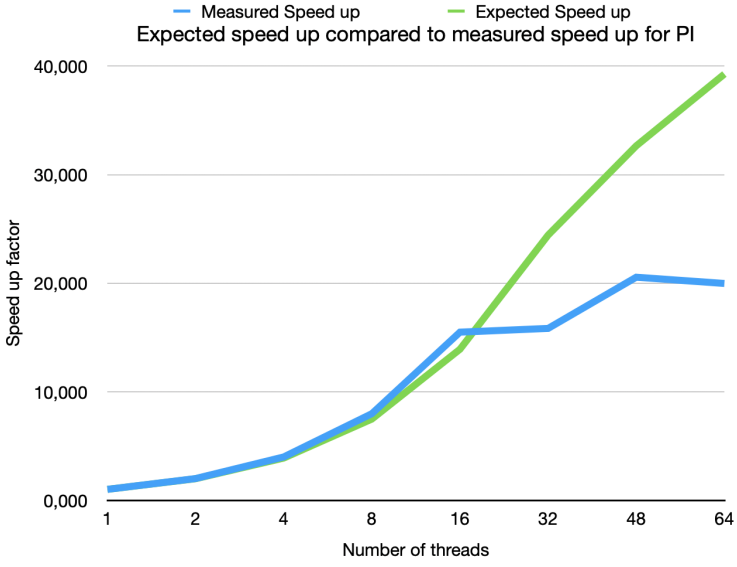
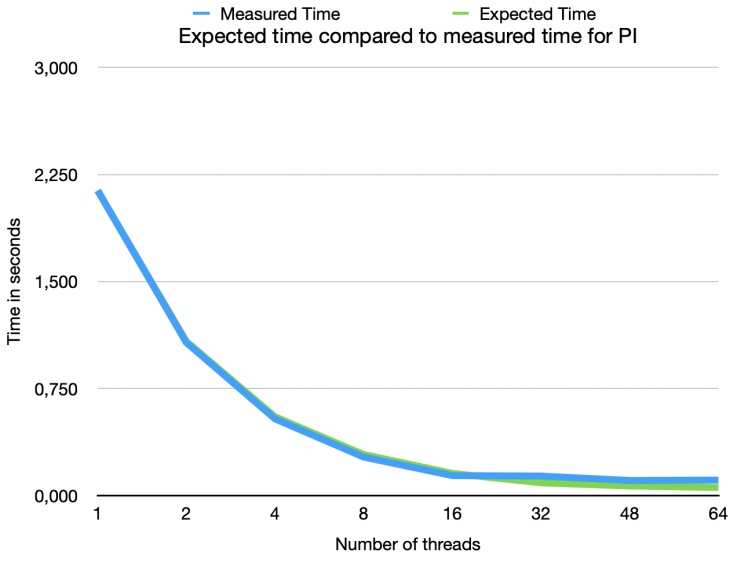


Figure 1.3



3. Comparisons

We can observe that up until 16 threads, the estimations and the measures are really close, but beyond that, the estimation keeps growing while the measure pretty much stalls. The reason behind that is mainly due to hardware limitations. Indeed, when computing the speedup with Amdahl's law, one does assume they have an unlimited number of cores at their disposal, which is far from true in real life. Since the SCITAS cluster we used to measure our runtimes had up to 28 cores, we couldn't obtain a speedup as good as the expectations once the number of threads went beyond 28.

Part 2 - Integral

1. Code description

a. We separated the program into three parts, two sequentials and one parallel. We parallelized the initialization of the random generator, and the for-loop for the same reasons as in part 1. We parallelized the for-loop since the computation of each loop is independent from one another. The only dependence between the loop instances is `surface`, which we compute in a critical way in the end of the parallel part, to avoid having read and write conflicts. The two sequentials parts are the initialisation and settings of variables, and the final computation. There is no interest and no viable way to make those sections parallel.

b. In the first sequential part, the `omp_set_num_threads` function is the most time consuming one. In the parallel for-loop, it depends on the time spent by the estimated function. Computing `integral` is the only and thus longest operation of the second and last sequential section.

c. Similar to part 1, we have that the execution time is O(`chunk`) = O(`samples/num_threads`).

d. Similar to part 1. As we chose `samples` to be equal to 10^9, and `num_threads` to be between 1 and 64, we estimate the parallelized fraction to be 99% of the total program. Results can be seen in Figure 2.1.

2. Execution time and observed speedup

Using Scitas, we measured the running time of integral.c 8 times, with the numbers of threads changing, samples always being equal to 10^9 and using the identity function as f(x) over the interval [0, 10]. We plotted the results along with the computations made in 2.1.d. in Figure 2.1, Figure 2.2 and Figure 2.3.

Figure 2.1

Integral

Number of threads	Measured Time	Measured Speed up	Expected Time	Expected Speed up
1	9,610	1,000	9,610	1,000
2	4,817	1,995	4,853	1,980
4	2,409	3,989	2,475	3,883
8	1,205	7,975	1,285	7,477
16	0,702	13,691	0,691	13,913
32	0,478	20,105	0,393	24,427
48	0,392	24,509	0,294	32,653
64	0,420	22,881	0,245	39,264

Figure 2.2

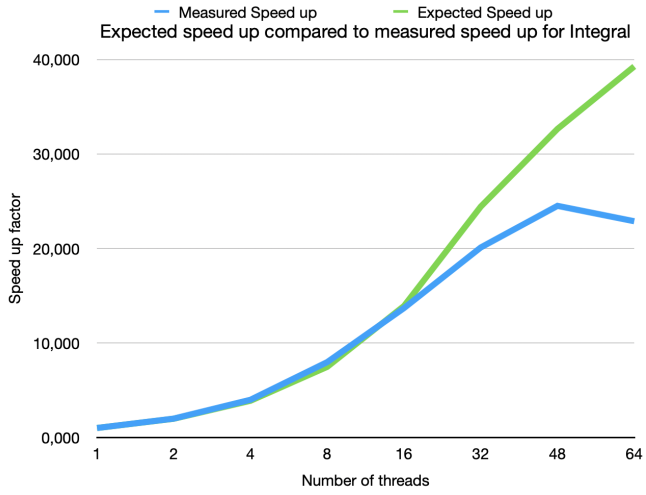
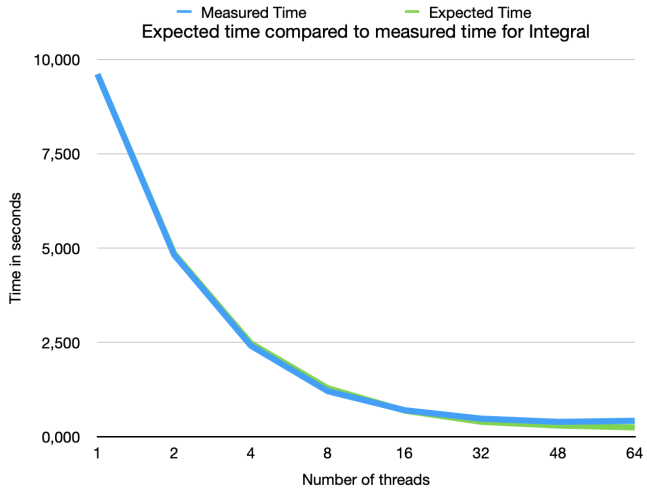


Figure 2.3



3. Comparisons

As in part 1.3, we can observe that up until 16 threads, the estimations and the measures are really close, but beyond that, the estimation keeps growing while the measure slows down. The reasons behind that are the same as in part 1, namely they are due to hardware limitations of the SCITAS cluster we used, we couldn't obtain a speedup as good as the expectations once the number of threads went beyond the number of cores of the cluster.