**Intelligence Artificielle**

**Assignment 1**

**I) Introduction:**

In this assignment we wrote two programs and parallelized them. First, we had to calculate by approximation PI and the second task was to approximate an integral between two points of a function. For the two programs we used the same code structure and thus the parallelization is the same for both, the only changes are the calculations made in the threads. Thus, we will describe both programs the same way.

**II) Algorithms description:**

The algorithm executes a for-loop on each sample and then adds the newly computed value in the loop to a global variable which will hold the sum of all the computation done in the for-loop.

The parts of the program we parallelized is the for loop so that different threads can execute different iterations at the same time. Each thread holds its own variable which will then, at the end of the threads for loop, be added to the global variable. To ensure that only one thread at the time access this global variable we used the Open mp command: pragma omp critical allows only one thread access a program block at a time. The beginning and the end of the program are sequential. The end is special because we process the remaining samples that we couldn’t parallelize (as there were less remaining samples than threads).

The beginning hasn’t an operation that dominates the execution time as we only initialize the parameters for the parallel part. The parallel for-loop has its execution time dominated by the generation of the random values. The last part which is sequential has its execution time dominated by the computation of the remaining samples and by the calculation made on the global variable which at this time holds all the computation (multiplication) made by the parallel for-loop.

We also thought about the implementation of the random number generator, either use one per thread or use a common one for all threads. We used the first method because the second one was good for few threads but the more they were, the more the single generator object became a bottleneck to the execution time. We concluded that the one generator was overwhelmed with requests from the different threads and couldn’t handle them as fast as they came in.

**III) Asymptotic execution time and speed up estimation:**

For the first sequential phase the execution time is constant regardless of the number of samples and threads that will run in the program, thus Phase 1 = O(cst1)

The parallelized part depends on the number of samples but also on the number of thread because each thread must create his own generator which takes some time, but this time is negligible compared to the number of samples which increases linearly the number of computations. In this part both programs (Pi and Integral) are slightly different. In the first one each thread must generate two random numbers for the x and y coordinates. In the second one there is only one number generated per loop. Thus, the execution time of both are not the same (as we will show below). We can represent this by multiplying the complexity by the number of times we have to generate a random number in one iteration. We also need to consider the fact that we parallelized the program, therefor Phase 2 has an asymptotic execution time of Number\_of\_generator\_calls \* O (#samples / Number\_of\_threads).

As the first sequential part, the last part has an asymptotic execution time. Even if we compute the remaining samples it’s only a fraction of the total computation. Also, we only computed between 0 and Number\_of\_threads -1 samples. Therefor the execution time is constant O(cst2)

Total asymptotic execution time = Number\_of\_generator\_calls \* O (#samples / Number\_of\_threads ) + O(cst1)

+ O(cst2)

= Number\_of\_generator\_calls \* O (#samples / Number\_of\_threads) + Cst

We expect a speed up that is equal to the number of threads we use (compared to using a single thread). For example, if we use 2 threads, we expect a speed up by 2 and so on for every other amount of thread. We can ignore the constant running time factor which has a much tinier impact than the linear one.

**IV) Algorithm testing:**