Report

Assignment 1

1- Explain how you solved each problem (of Parts 1 & 2) in a parallel fashion, what parts of the program remain serialized, and how will performance improve with parallelization in the context of Amdahl's Law. Feel free to roughly estimate the speedup you expect, and you don't need to provide real run-times for this task. The purpose of the exercise is to get you to reason about the speedup of the program based on how you parallelized it.

A. : Both problems have been solved using the same method : we used the « for » directive with the « reduction » clause. We didn't use the combined « parallel - for » directive as we wanted to get an unique « rand_gen » value for each thread with the « init_rand() » function before calling the « for » loop. Reduction at the end will sum every computations of each thread, these ones having run on a different set of points from the given sample. To get the different values for the generated points to calculate PI, we simply used the « next_rand » function because we want random values between 0 and 1, as this function does. For the generated points used to calculate the integral, to get values inside the given interval, we still used the « next_rand » function, but inside a tricky formula which is « a + next_rand * (b - a) », so we can get random values inside the interval [a;b].

The only parts of the programs that remain serialized are the main execution block and the auxiliary functions that were already implemented. Concerning the functions we had to implement, the only serialized parts are when we simply declare our variables just before the parallelization clause, and just after the parallelization block end, when we calculate our final variable (PI, or the integral) using the known formula for each of them, with the calculated sum of areas (for the integral), or the incremented counter of inside-points (for PI). Then both of these two functions are mostly parallelized.

In the context of Amdahl's law, and according to its formula for the theoretical speedup, the more the proportion of the whole task, occupied by the part of the task benefiting from improved resources, is large, the more the theoretical speedup of the execution of the whole task increases.

In our code, for each program, the great majority of the execution time resides in the « calculate_pi » and the « integrate » functions, which benefits from improved resources (parallelization). We then can roughly consider that the theoretical speedup will be proportional to the speedup of the part of the task that benefits from improved system resources, which, we would think, tends to increase if we provide more threads to be running in parallel during the execution.

2- In a table, present the execution times, and speedups over the single threaded version, you measured for the following set of thread counts {1,2,4,8,16,32,48,64}. Run all of your results on the SCITAS cluster using the job submission system explained in the exercise session, as well as the link on the course webpage.

A. : Table of results for pi.c program with a sample of 500000000 points :

Number of threads	Execution time (in seconds)	Speedup over single threaded version
1	6.906	1
2	3.461	1.995
4	1.804	3.828
8	0.9606	7.189
16	0.4926	14.019
32	0.4309	16.026
48	0.3844	17.965
64	0.3602	19.172

Table of results for integral.c program with the function « x » on the interval [5;9] and a sample of 2000000 points:

Number of threads	Execution time (in seconds)	Speedup over single threaded version
1	0.01781	1
2	0.008996	1.979
4	0.004662	3.820
8	0.002666	6.680
16	0.00172	10.354
32	0.003337	5.337
48	0.003728	4.777
64	0.003825	4.656

3- Compare the speedups you measured on the SCITAS machines, to what you predicted in Task 1. If they are different, why?

A. : For the pi.c program, our expectations were right : the more the number of threads running in parallel is high, the more the theoretical speedup of the execution of the whole task increases. Amdahl's law still states that the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement. Because the program is obviously not totally parallelized, the upper bound of the theoretical speedup is not the infinite. That's why the speedup is increasing slower when the number of threads becomes very high, and then tends to reach a maximum possible value.

But for the integral.c program, we have a contradiction in our expectations. When the number of treads becomes high, the speedup over the single threaded version stops increasing and start to continuously decrease. We then can admit that there still exists a value for the number of threads that lets the speedup reach a peek performance, but then, we see that a performance increase in one part of the system can negatively impact its overall performance in direct contradiction to the way Amdahl’s Law is instructed.

This is due by the fact that a thread needs resources such as CPU resources, RAM resources, Network and/or hard drive resources. All of these resources are finite and any one can limit the performance of a system. If a thread is consuming too much of these limited resources, or if too much threads are consuming them at the same time (in our case), then this can cause bottlenecks, and situations where threads are fighting for resources.