Name: Floriane Gabriel Sciper: 248112

Julien von Felten 234865

1: a) For both program, we decide to split the work equally for each thread. Using OpenMP, we have all threads running in parallel. We use also a private counter for each thread to sum (sumThread) so we don't need to access all the time a global variable that have to be atomically accessed (meaning sequential access). The random variables are also generated in each thread. If the random generator was global, we could have some problems with the concurrency (using twice the same random number or have to wait the other thread to generate a random number). After the for loop, we add the sumThread to the general sum atomically to avoid having concurrent problem. At the end, we compute what the program needs to compute (pi or the integral computation). This means that we could parallelize most of the program, the only thing we cannot parallelize is the initialization at the beginning and the computation at the end, which is impossible to parallelize.

Since we only have threads in parallel and two "phases" in serial.

b)The two phases in serial are only initialization of variables and the computation of the sum and PI or the integral. The second phases, the operation that takes the most time is the division. But the biggest phase is the parallel one which takes the biggest time. The operations that dominate the execution time for the parallel part is in the for loop since it is where it spends the most time. Since we have to generate a new random number and we have some multiplication, this can be one of them. We don't know how much it costs to generate the next random number in comparison with the multiplication.

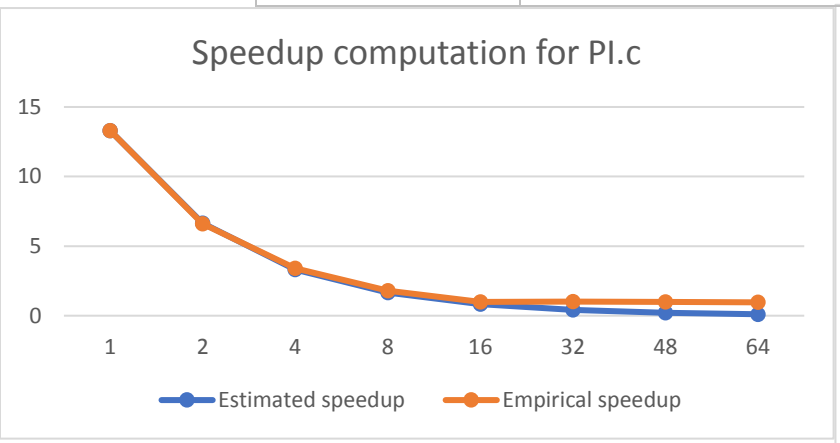
c) The argument that affects the number of performance is the samples in our method. Then it is the splitSamples that divides the job among the threads. The for-loop must be done $\frac{samples}{num_{threads}}$ times. So, the more samples we have, the more job the threads will have to do. We would say that it is linear since each thread can work on their own and it depends only on the computation $\frac{samples}{num_{threads}}$, where samples is a constant. So, the big O would be : $O\left(\frac{C}{n}\right) = O\left(\frac{1}{n}\right)$.

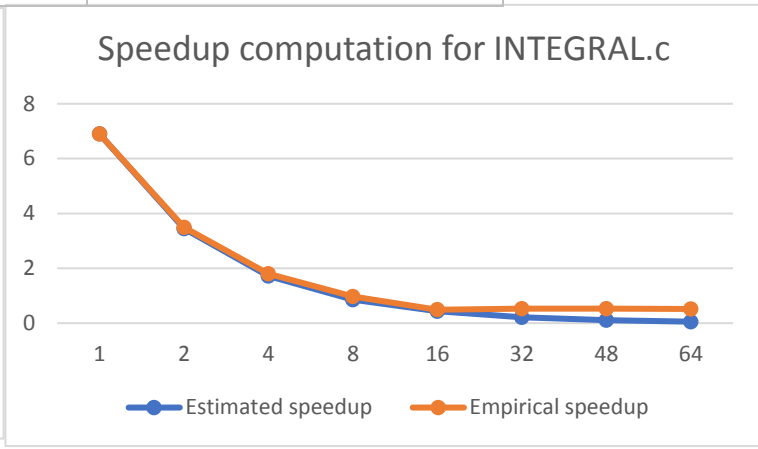
d) For the speed up, we can say that we could parallelized the whole program, so the computation of the speedup using Amdahl's Law is the number of threads if we compare with a sequential run. This computation ignores the overhead and the atomic operation. We can expect the speed up to be close to the number of threads for a small number but when we have a bigger number of threads, the gain will be cancelled out by the overhead. We should have a maximum speed up value possible. You can see the graphics on the second page.

2: The speed up is computed as follow: $Speedup(p\ threads) = \frac{Time(1\ thread)}{Time(p\ threads)}$, where p is the number of threads. We computed it with $5 * 10^9$ points for both pi.c and integral.c.

| PI.c : | | |
|---|---|---|
| Number of threads | Executed time | Speed up |
| 1 | 13.3 | 1.0 |
| 2 | 6.6 | 2.015 |
| 4 | 3.4 | 3.912 |
| 8 | 1.8 | 7.39 |
| 16 | 1.001 | 13.287 |
| 32 | 1.021 | 13.026 |
| 48 | 0.98 | 13.571 |
| 64 | 0.96 | 13.85 |

| INTEGRAL.c : | | |
|---|---|---|
| 1 | 6.9 | 1.0 |
| 2 | 3.5 | 1.97 |
| 4 | 1.8 | 3.83 |
| 8 | 0.97 | 7.11 |
| 16 | 0.49 | 14.08 |
| 32 | 0.53 | 13.02 |
| 48 | 0.53 | 13.02 |
| 64 | 0.52 | 13.27 |




Thoses values have been calculated on the SCITAS cluster with 16 cores.

For the computation of integral, the bounces were 5 and 9, with the function $f(x) = x$.

3: We can see that the computation is similar to what we expected in part 1, with the time converging to a value which is around one second for the PI.c and around 0.5 second for INTEGRAL.c. We can also see that if we double the number of threads, we nearly double the performance (speedup is around 1.97 or 2.015, then 3.912 or 3.83). This is what we expected: for a small number of threads, we have have a speedup that is close to the number of threads. But when we have too many threads, the speedup isn't close to the number of threads anymore (16 threads, 13 or 14 speedup). It even reaches a value and there is no improvement for the speedup. This happens because SCITAS allows us to use 16 cores maximum.

Conclusion:

With this homework, we have realized that we can have a real speedup (up to 13.8 times faster) for a specific task using OpenMP. To do so, it is not possible to write only "#pragma omp parallel" but we need to change a bit the computation and how to divide the work for each thread.