DSSC - Scalability

Presta Alberto - Grimaldi Marco Alberto

November 2018

Introduction to the exercise

We want to estabilish the scalability of a HPC application. In this particular exercise we will use the "Monte-Carlo algorithm" in order to give a good approximation of the number π. As soon as we have already both a serial implementation of the algorithm (pi.c) and a parallel MPI implementation (MPIpi.c), our aim is to see how well this application scales up to the total number of cores.

0.1 Serial Code

In this firs step we want to determine the CPU time required to calculate PI with the serial calculation using different iterations (from 10^7 to 10^{10}). In order to compile the serial code we have to write the following command:

$$time\ gcc\ pi.c\ -o\ serial\_pi$$

$$serial\_pi\ \ 1000000$$

We repeat the last command also for 10^8, 10^9 and 10^{10} and we get the following results (obviously the results are in second):

	10^7	10^8	10^9	10^{10}
real	0.259	2.542	25.353	229.80
user	0.254	2.534	25.367	229.66
sys	0.002	0.002	0.001	0.00
%	98	99	100	100

We note that, using the command *time*, we have tree different values:

1. *Real time*: the real execution time (from the starting until the end of the program
2. *User time*: The amount of time spent by CPU executing your code's instructions (time spent on processor)
3. *System time* : system CPU time is the time spent running code in the operating system kernel on behalf of your program.

So we have some system time and this could represents a problem because we are mostly interested in time spent on the CPU (User time): However, we notice that the system time is almost irrelevant. Using the command usr/bin/time we can have also the percentage of time spent inside the CPU.[1].

[1] technically we could calculate by ourselves with the following formula: $\% = \frac{100*(user+sys)}{real}$

0.2 Parallel code

Now we want to get MPI code running for different iterations and we want to reach the time needed to calculate PI, in order to have a measure of scalability (both strong and weak scalability). In order to do so, we have written two different bash script (one for the strong scaling and one for weak scaling) where we compute the Walltime changing both the number of processors (from 1 to 20) and the number of iterations (from 10^7 to 10^{10}).

Strong scalability

For the strong scalability, we have that:

$$N * P = const$$

Where N stands for the number of iterations and P stands for the number of processor. Basically we don't change the total number of iterations when we increase the number of procs: in that way the time should decrease in a inverserly proportional way. Results which we have obtained running MPI programm are shown in the table 6 at the end of the report.

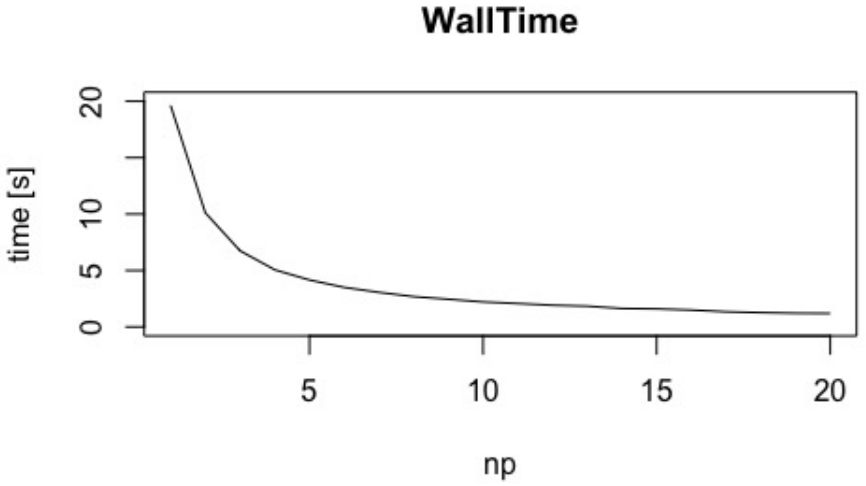

Figure 1: this graph shows how time decreases while the number of procs increases with 10^7 iterations

We want to say that until now we have used the Walltime,which means that we are considering basically only the time spent on the CPU (this is also due to the fact that the code has been written in order to leave out time related to system or other devices). Analyzing strong scalability we will see that with many processors and with a low number of iterations (for example 10^7) system time is not negligible at all. Let's see some useful graph in order to understand how Walltime decreases with the number of processors. We could investigate if, with the increase of the number of processors, the rate of decreases of walltime would speed up (let's see Figure 1).

Speed up Strong scalability

In an ideal situation, there would be *perfect speed up*:

$$Speedup(p) = p$$

What kind of trend has the real speed up changing number of iterations? let's see figure 2: top.
We can notice that for all the different number of iteration, the speed up's rate is very similar to the perfect speed up, which means that we have got a very efficient and fast code to calculate PI! However until now we haven't considered time spent on the system and for network communication between processors. can these things affect speedup? in certain situation we suppose yes; indeed when we deal with a relatively low number of iterations (10^7 for example) and an high number of processors (20 for example) we have the Amdahl's law: Even if our parallel part of the code works perfecty, speedup will always be limited by the "serial part" (Figure 2: bottom.

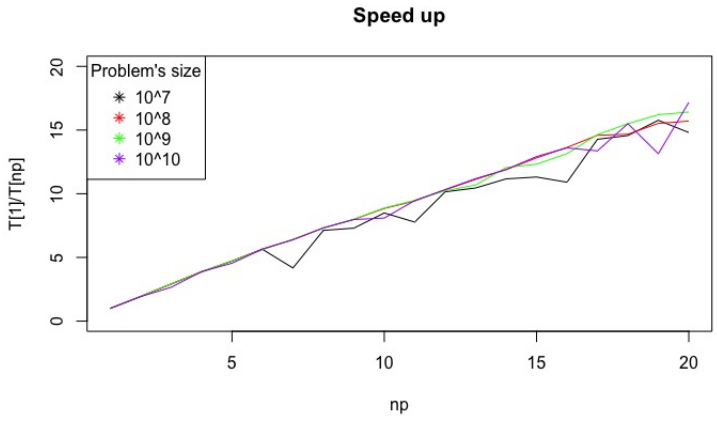

Figure 2: Speedup considering only User time, almost perfect.

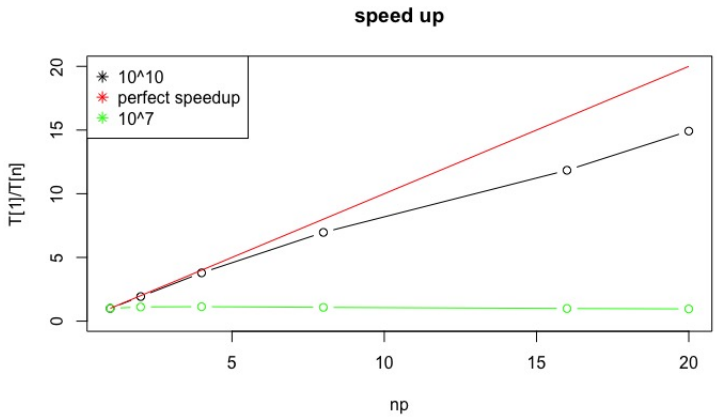

Figure 3: Speedup considering also System time, both the number of iterations and the number of threads influences the pattern of the strong speedup

How can we resolve this very bad situation? If we consider the weak scalability (and we will do in the following section) we have the Gustafson's Law, which says that the serial part of the code normally decreases when the size of the problem increases.

Weak Scalability

For the Weak Scalability, we have the following situation:

$$\frac{N}{P} = const$$

What does it mean? It means that the total number of iterations $N_{tot} = N \cdot P$ grows up linearly with the number of processors P. Our goal is to verify if the time remains constant with the increase of N and P; in the Figure 4 we can see how "time-rate" with 10^9 iterations and the how time increases in percentage with respect to the number of iterations.[2]

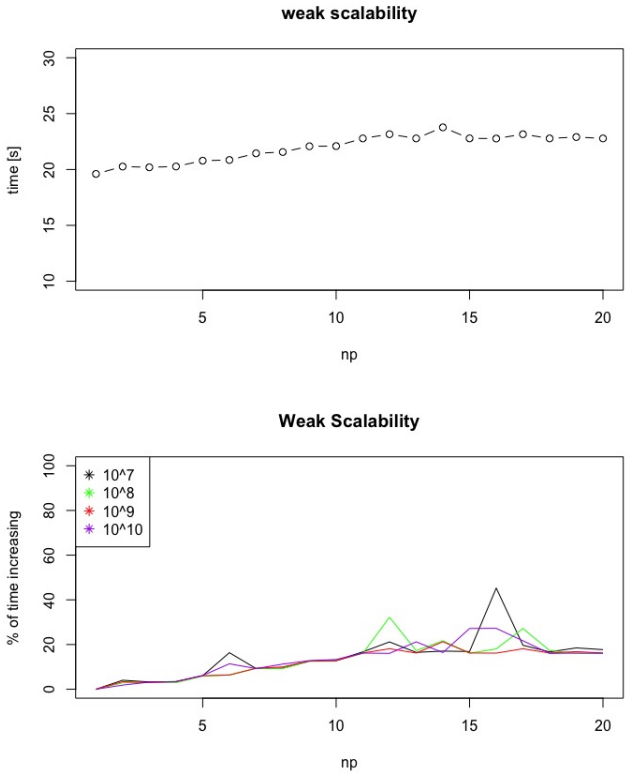

Figure 4: In the first figure we have the variation of time with 10^9 iterations, while in the second one we see how time increases in percentage: for example with 5 processors we have an increase of about 10 %.

One more things that we could analyze is the fact that, as well as in the case of strong scalability, we didn't use the command *time*. Neglecting then the system time (which always remains very low), we can compare user time with the elapsed time: in the following figure we do this in the case of 10^9 iterations.

[2]The table 7 with all results is linked at the end of the report.

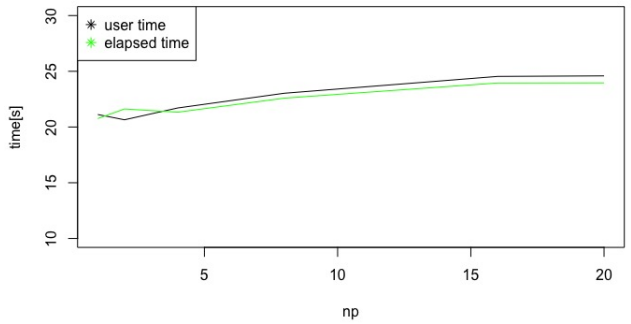

Figure 5: Comparison between elapsed time and user time.

We can see that, in the case of 10^9 iterations (which is a pretty high number) there is no such a great difference between user time and the total time for the computation.

Conclusion

During the report we have mentioned the *Amdahl's law*, which could suggest that weak Scalability would be better for analyzing problems where the number of iterations increases a lot; nonetheless our results show to us that if we consider only CPU time we reach a very good speed up (almost the perfect speed up) as we can see in figure 2. So probably because our algorithm is written in a very good way in order to be parallelized, we have we have collected very comforting data both as regards weak scalability and strong scalability.

Table of times for weak and strong scalability

	time7	time8	time9	time10
1	0.19577098	1.9618061	19.627402	196.04875
2	0.10100579	1.0116172	10.119152	102.02549
3	0.06732678	0.6727781	6.747136	73.78640
4	0.05067301	0.5050640	5.060762	50.49573
5	0.04152608	0.4159730	4.157619	43.12313
6	0.03473783	0.3466170	3.501692	34.66348
7	0.04695106	0.3067231	3.058460	30.67298
8	0.02748704	0.2681892	2.682488	26.80094
9	0.02682400	0.2457340	2.452429	24.58109
10	0.02305698	0.2218249	2.210516	24.22951
11	0.02517891	0.2080400	2.070440	20.71548
12	0.01929021	0.1903758	1.918621	18.98852
13	0.01873088	0.1763201	1.836618	17.52834
14	0.01753306	0.1650031	1.627994	16.47152
15	0.01730704	0.1531072	1.593572	15.17965
16	0.01795506	0.1438770	1.493139	14.41294
17	0.01371694	0.1344790	1.340825	14.67460
18	0.01343608	0.1337140	1.266550	12.64794
19	0.01241612	0.1263919	1.211437	14.92291
20	0.01322293	0.1249311	1.195802	11.41878

Figure 6: in the row we have the number of processors, while in the columns we change the numbers of iteration (we consider 10^7, 10^8, 10^9, 10^{10});Obviously time is calculated in second.

	Wtime7	Wtime8	Wtime9	Wtime10
1	0.1959369	1.962668	19.60434	196.1529
2	0.2038429	2.019265	20.26465	199.7836
3	0.2023508	2.020403	20.19541	202.4198
4	0.2025209	2.021056	20.27140	202.9759
5	0.2078171	2.077466	20.78910	208.2283
6	0.2279670	2.087997	20.85109	218.4851
7	0.2140670	2.145762	21.44657	214.3092
8	0.2144260	2.145906	21.56400	218.2241
9	0.2206340	2.207058	22.07857	221.2619
10	0.2211969	2.214650	22.09377	222.2451
11	0.2287030	2.277572	22.78196	227.8942
12	0.2374179	2.594809	23.15778	227.6389
13	0.2282331	2.301839	22.78143	237.7194
14	0.2293849	2.388020	23.77280	228.3499
15	0.2289948	2.278916	22.78638	249.4456
16	0.2846539	2.316967	22.77123	249.5954
17	0.2344949	2.495831	23.15673	238.6626
18	0.2286520	2.302985	22.78296	227.5997
19	0.2321749	2.277158	22.90456	227.8935
20	0.2306609	2.279065	22.78207	227.7198

Figure 7: in the row we have the number of processors, while in the columns we change the numbers of iteration (we consider 10^7, 10^8, 10^9, 10^{10});Obviously time is calculated in second.