First Assigment FHPC course

Section 0: Warm-up

Compute Theoretical Peak performance for your laptop

Theoretical peak performance is given by the equation:

$FLOPS = cores \times clock \times FLOPS/cycle$

Where FLOPS/cycle is obtained looking at:

- https://en.wikipedia.org/wiki/FLOPS

While all the other information are gathered thanks to the bash command:

```
cat /proc/cpuinfo
```

	Your model	CPU	Frequency	Number of core	Peak performance
Laptop	HP - 15-ba026nl	AMD Quad-Core A10-9600P	3,3 GHz	4	52.8 GFlop/s

Compute theoretical and sustained performance on your smartphone

To find the sustained performance, the following app was used:

- https://play.google.com/store/apps/details?id=com.sqi.linpackbenchmark

	Your model	CPU	Theoretical Peak performance	Size of Matrix	Sustained performance
SmartPhone	Motorola Moto X Force	Qualcomm Snapdragon 845	16.6 GFlop/s	1500	500,17 MFlop/s

In order to find the maximal value, the size of the matrix was gradually increased from 500 to 3000, reaching a peak at 1500, while keeping an average performance of 450 MFlop/s.

The informations concerning the Top500 where found at the link:

- https://www.top500.org/list/1993/06/?page=5

	Top500	Position	Performance (Rmax)	Performance (Rpeak)	Number of processors (TOP500)
SmartPhone	June 1993	412	0.5 GFlop/s	0.6 GFlop/s	5
Laptop	June 1997	2	30.4 GFlop/s	69.6 GFlop/s	544

Section 1: Theoretical model

Theoretical model

The speedup of a parallel application is defined as

$SU(P) = T(1)/T(P)$

where $T(1)$ denotes the execution time of the serial algorithm, while $T(P)$ the execution time of a parallel application on P processor. It is possible to evaluate this two times in the following way:

- Serial Algorithm :

$T(1) = T_{serial} = N * T_{comp}$

T_{comp}= *time to compute a floating point operation*

- Parallel Algorithm (master-slave):

1. Read N and distribute N to P-1 slaves:

   $T_{read} + (P-1) \times T_{comm}$

T_{comm} = *time each processor takes to communicate one message*
T_{read} = *time master takes to read*

2. N/P sum over each processors (including master) :

   T_{comp}/N

3. Slaves send partial sum :

   $(P-1)T_{comm}$

4. Master performs one final sum:

   $(P-1) \times T_{comp}$

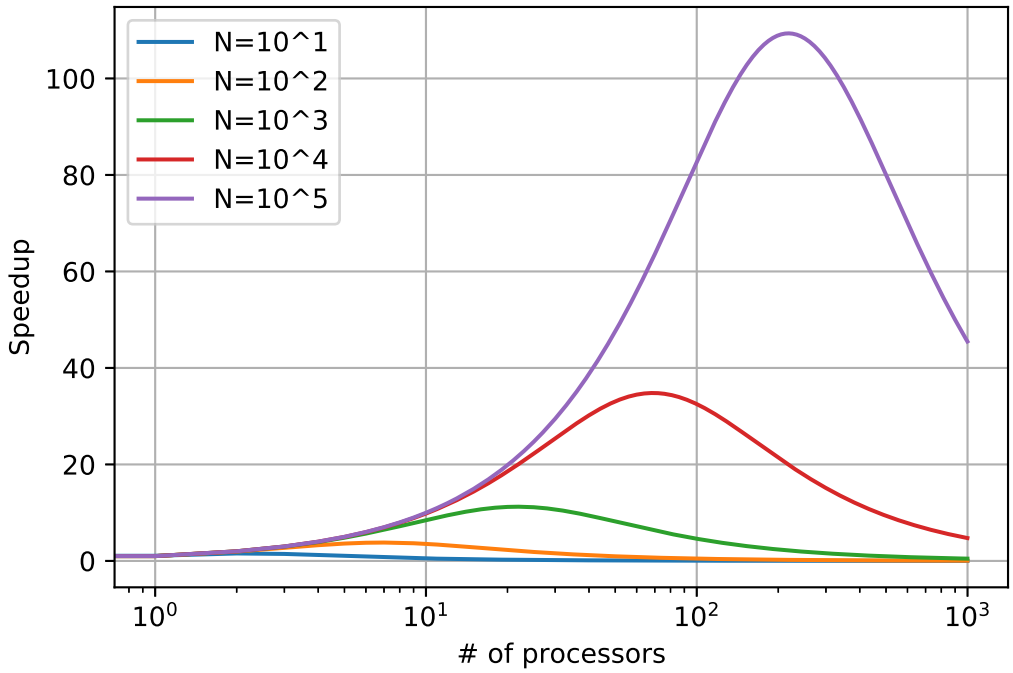
5. The final model: $T(P) = T_{comp} \times (P - 2 + n/P) + T_{read} + 2(P-1) \times T_{comm}$

Scalability curves could be plotted using rule of thumb estimastons for the parameters

	Estimation
T_{comp}	2×10^{-9}
T_{read}	1×10^{-4}
T_{comm}	1×10^{-6}

Graphs

Graph where made using Matplotlib library in pyton

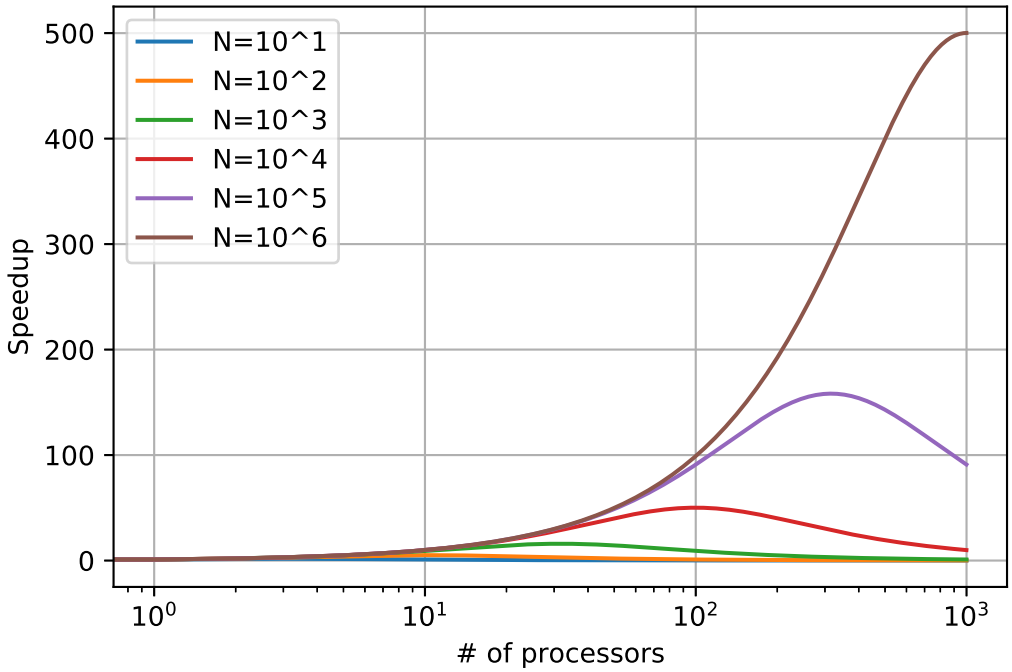


- The algorithm start to substantially scale for N of the order of 10^3. Before that the speedup is of the order of unity,
- For every assigned value of N, each curve show a specific maximum of speedup, before communication time starts becoming preponderant,

The algorithm could be improved through the use of collective operations. In that way all the $P-1$ communications are reduced to only one T_{comm}

The $T(P)$ expression is now:

$$T(P) = T_{comp} \times (P - 2 + n/P) + T_{read} + 2 \times T_{comm}$$



The plots shows the same behavior as before, but It is possible to notice that the maximum is higher and shifted to the right.

Section 2 : Play with MPI program

Compute strong scalability of a mpi_pi.c program

Time required by each algorithm were measured using:

- The greatest time recorded among each core by the function walltime, used in the script to cut out any system operation (ideally it should be the one of the master node)
- The bash command

```
/user/bin/time
```

Who returns three values:

Time	Description
elapsed	Time spent from start to finish of the call
user	Amount of CPU time spent in user mode
system	Amount of CPU time spent in kernel mode

In the following table is reported time measured using walltime for the serial program, and for the MPI program using just 1 processor:

	Time
Serial	0.190s
Parallel	0.204

A rough estimation of the the overhead can be obtained with:

$$T_{parallel} - T_{serial} = 0.204s - 0.190s = 1.4 \times 10^{-2}$$

For the sake of collecting data to compute strong scalability, the following bash script were implemented to collect walltime (fixing N=10 millions):

```
for procs in 1 2 4 8 16 20 ; do
mpirun -np ${procs} mpi_pi 10000000
done
```

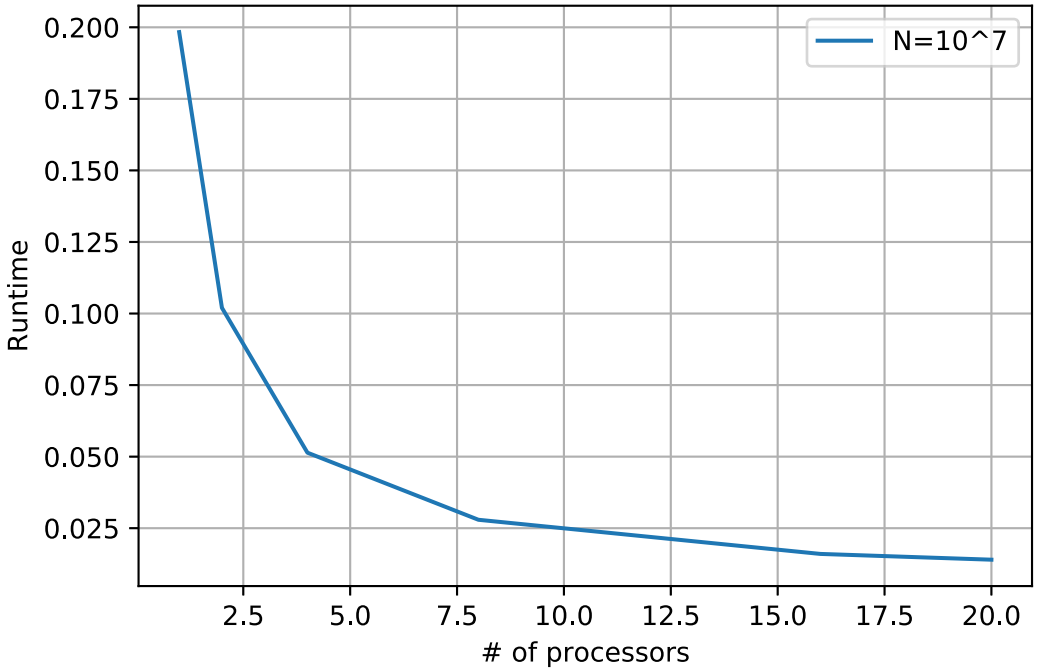
While using $/user/bin/time$:

```
for procs in 1 2 4 8 16 20 ; do
time mpirun -np ${procs} mpi_pi 10000000
done
```

Here are reported times obtained with walltime:

Processors	Walltime
1	0.198
2	0.102
4	0.051
8	0.028
16	0.016
20	0.014

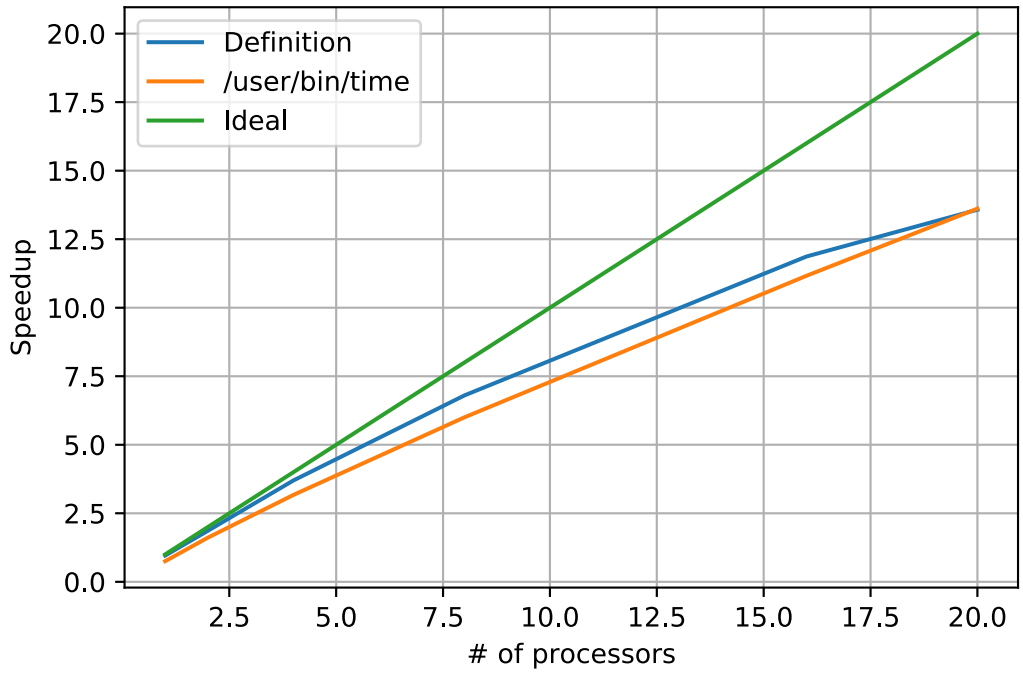
The following plot show the correlation between runtime and number of processors: as the latter increase, the former decrease



Knowing the runtime, it is now possible to evaluate the speedup. The values obtained by both the definition, and a rough estimation using /user/bin/time ($Speedup = (user + system)/elapsed$) are reported in the following plot and table:

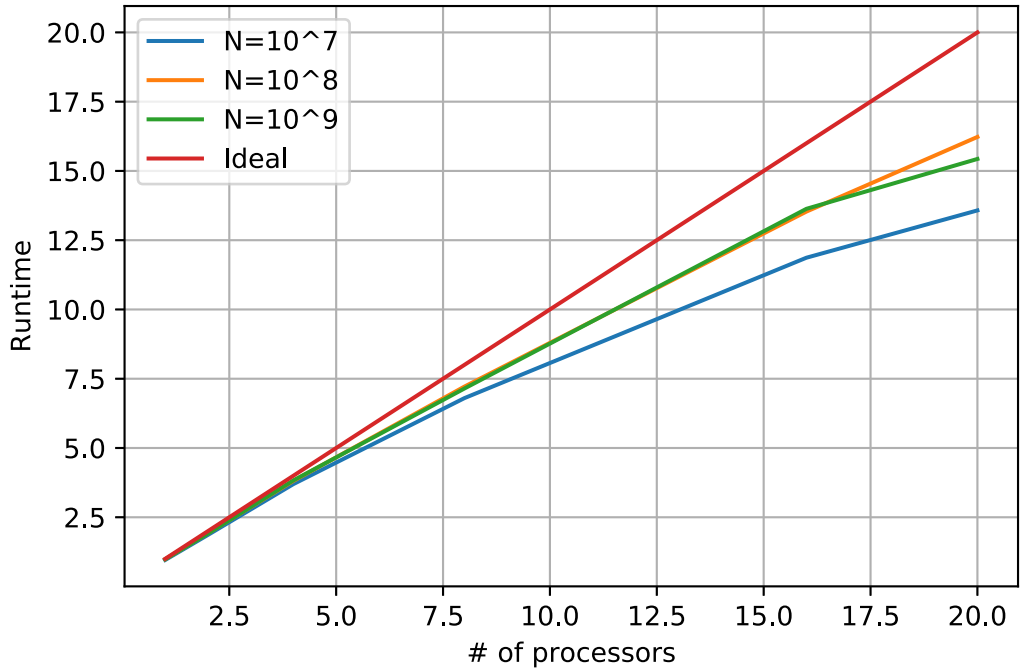
Processor	Definition	/user/bin/time
1	0.958	0.752
2	1.863	1.612
4	3.698	3.170
8	6.800	6. 000
16	11.867	11.163
20	13.575	13.615

The graph shows that both plots have qualitatively the same behavior, and both of them tend to differentiate themselves from the ideal linear speedup.



In the following table are displayed for increasing values of N. Times are collected using only walltimes, as they avoid measuring operations such as initializations and prints.

Processors	Time (10^8)	Time (10^9)
1	1.985	20.007
2	1.020	10.284
4	0.514	5.128
8	0.272	2.750
16	0.145	1.441
20	0.121	1.273



2.2: Identify a model for the parallel overhead

First of all, besides the constant already defined in section 1, the following variables are defined as follow:

	Purpose
T_{par}	Required by the parallel part of the code
T_{ser}	Required by the serial part of the code
T_{comm}	Required by the communication between the processors
T_{sync}	Required by the synchronization of the processors
T_{rand}	Required to generate a random number (assumed equal to a floating point operation)

The total runtime of the parallel program can be expressed as

$$T(P) = T_{par} + T_{comm} + T_{sync}$$

T_{par} can be found as the sum of the following operation:

- Loop over n point: $2 \times n \times T_{comp}$
- Generation of n random points: $2 \times n \times T_{rand}$
- Checking if the points are inside the circle: $4 \times n \times T_{comp}$
- Count suitable points: $\pi \times n/a \times T_{comp}$
- Sum all points in master node: $(P-1)n \times T_{comp}$
- Compute π: $3 \times T_{comp}$

In which n are the operation per processor (N/P, with N=10^7 for us)

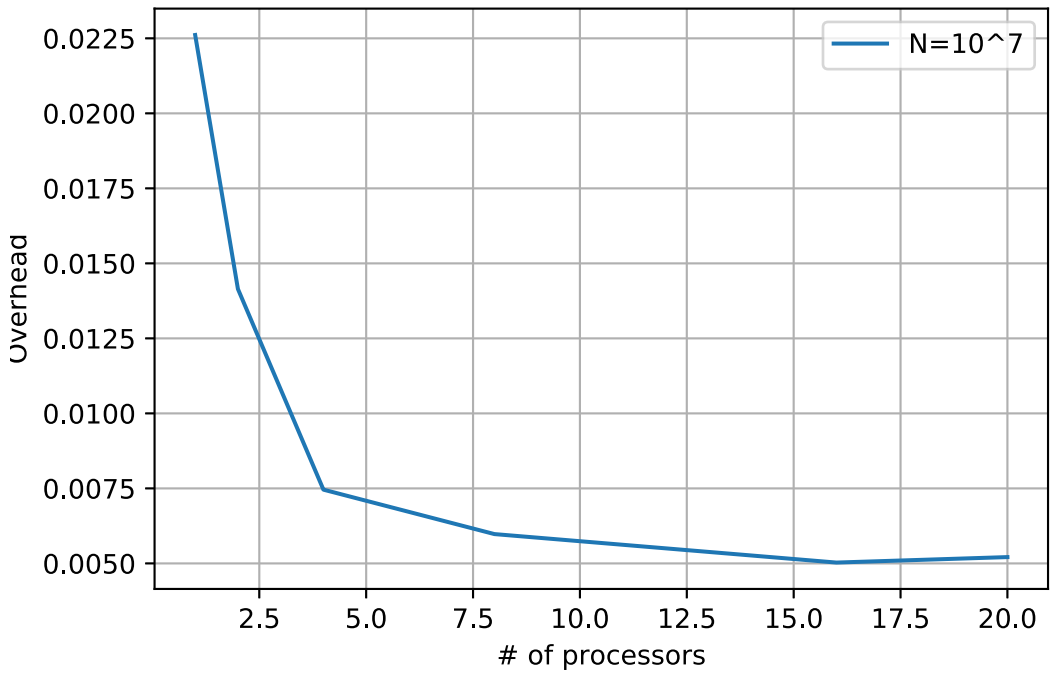
T_{comm} can be found as:

- Sending of the partial sum to the master: $(P-1) \times T_{single\_comm}$

So we can deduce the overhead and the T_{sync} from the formula:

$$T_{ser} = T(P) - T_{par} = T(P) - 2 \times n \times T_{rand} - (6 \times n + P + 2 + \pi \times n/4) \times T_{comp}$$
$$T_{ser} = (P-1) \times T_{comm} + T_{sync}$$

Processors	Overhead	Synchronization
1	2.261 $\times 10^{-2}$	2.261 $\times 10^{-2}$
2	1.414 $\times 10^{-2}$	1.414 $\times 10^{-2}$
4	7.458$\times 10^{-3}$	7.455$\times 10^{-3}$
8	5.977$\times 10^{-3}$	5.970$\times 10^{-3}$
16	5.028$\times 10^{-3}$	5.013$\times 10^{-3}$
20	5.210$\times 10^{-3}$	5.191$\times 10^{-3}$



The following table show the percentage of the run time spent in overhead:

Processors	Overhead fraction
1	11.4%
2	13.9%
4	14.5%
8	21.4%
16	31.4%
20	37.2%

As expected, fixed the number N, the overhead fraction is increasing with the number of processor. That is caused by the fact that while the number of processes carried on by every processor decrease, time spent in communication increases.

2.3: Weak scaling

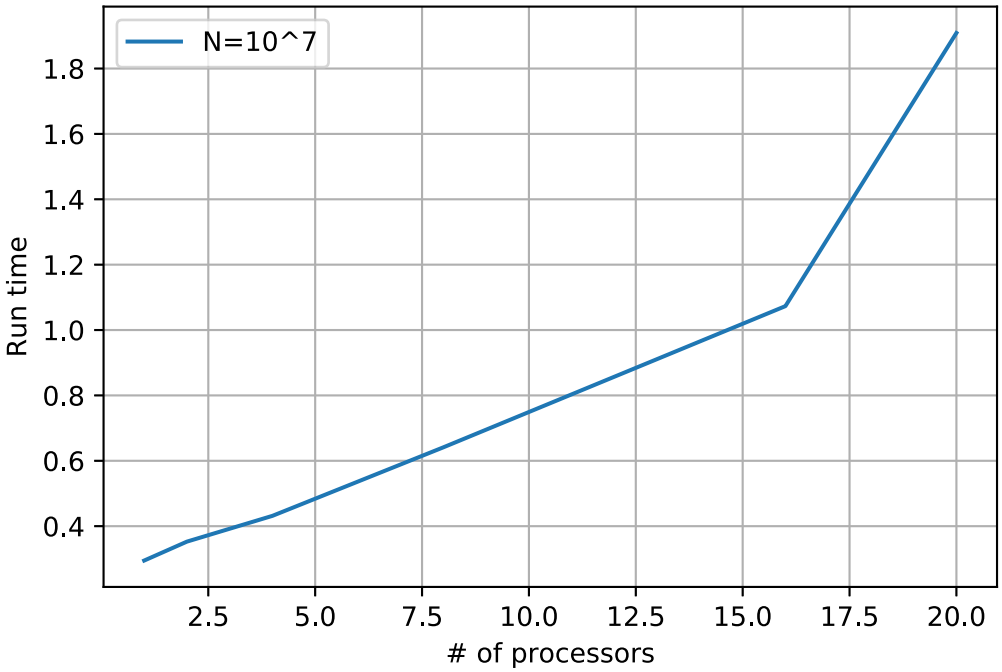
A weak scalability study of the program was obtained keeping the ratio of work per processor (N/P) constant.

In order to to so, the following bash script was implemented:

```
for procs in 1 2 4 8 16 20 ; do
mpirun -np ${procs} mpi_pi $((${procs}*10000000))
done
```

The following table shows times obtained using as N/P ratios 10^7, 10^8 and 10^9

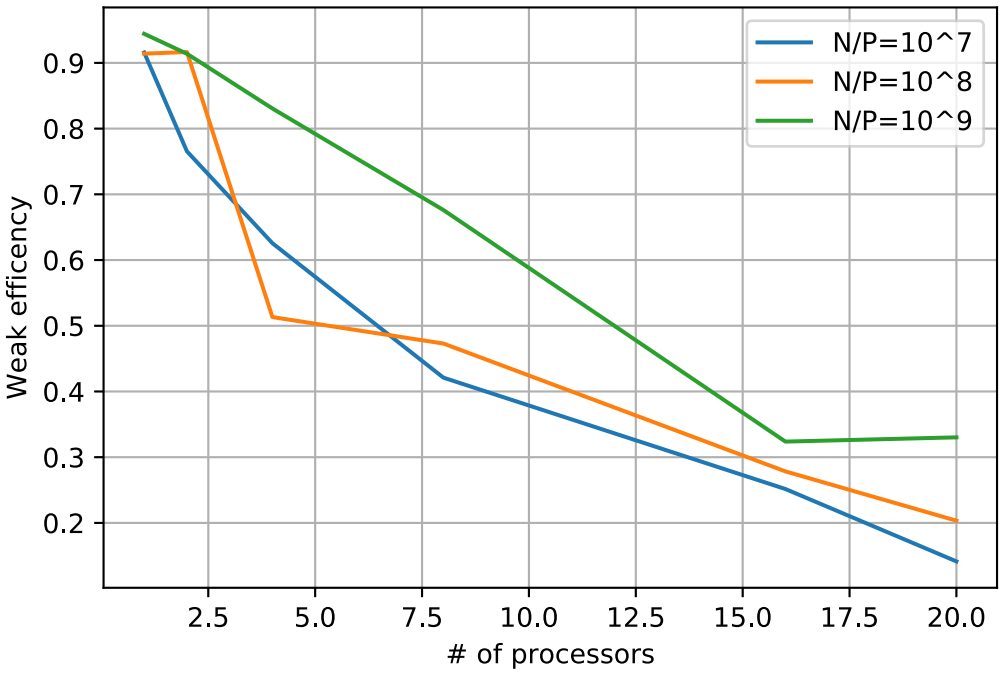
Processors	Time 10^7	Time 10^8	Time 10^9
1	0.295	2.801	26.900
2	0.353	2.793	27.793
4	0.432	4.988	30.579
8	0.641	5.413	37.567
16	1.073	9.195	78.460
20	1.909	12.584	76.924



It is possible to compare the weak scalability of the program defining a sort of weak efficiency as:

$$ES(P) = T(1)/T(P)$$

The results for the ratios reported above are plotted in the following graph:



A perfect weak scaling program would show a straight line equal to 1, but in the case studied the increasing time required for communication and synchronization lead to a decreasing trend.

Section 3: Implement a parallel program using MPI

The two attached C programs compute the sum of the first N integers, given N in the bash line. The first one use MPI_Send and MPI_Recv, while the second use collective operations. In both cases the result has been checked through the comparison with Gauss' formula:

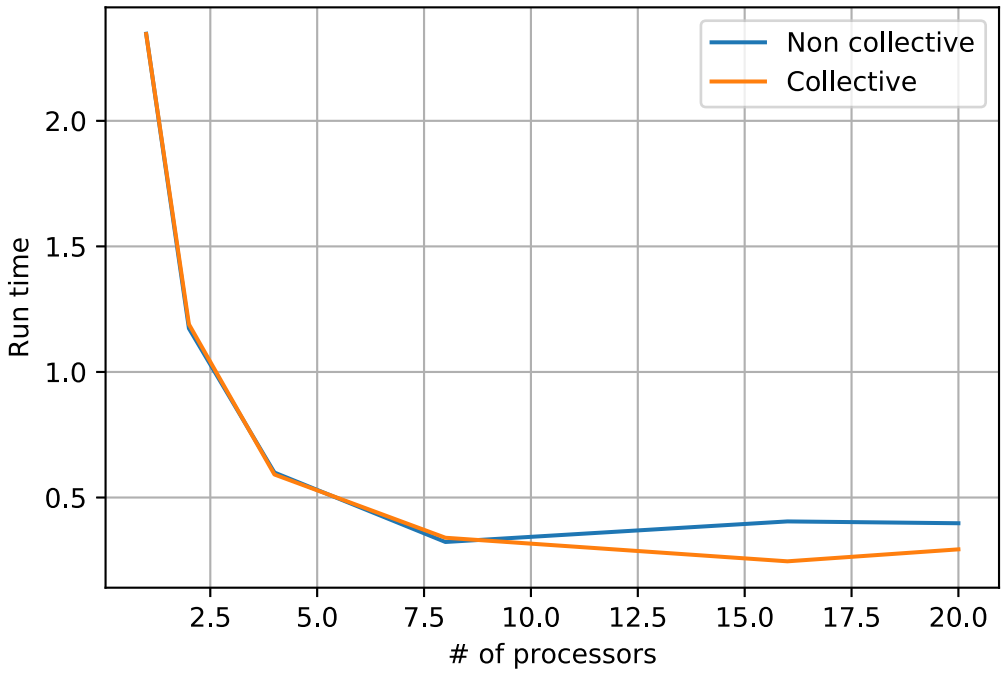
$$Sum(N) = (N+1) \times N/2$$

Section 4 : Run and compare

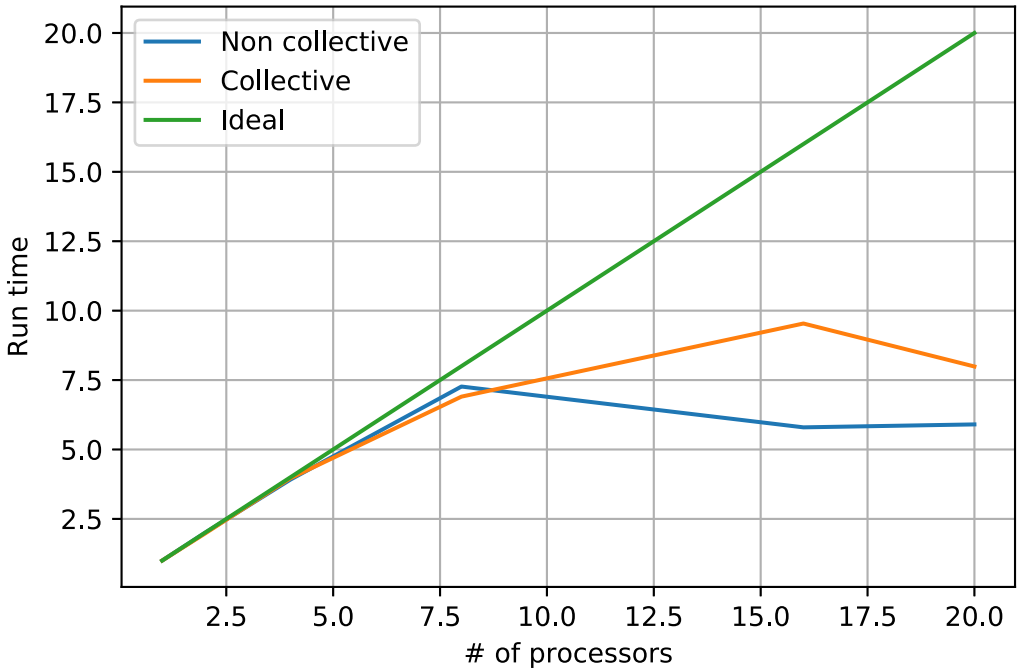
Run times for the collective and non collective program are reported in the following table (N=10^8):

Processors	Non collective	Collective
1	2.345	2.345
2	1.173	1.190
4	0.601	0.592
8	0.323	0.340
16	0.405	0.246
20	0.398	0.294

The differences between the two became visible already at 16 processors.



The following plot show the speedup for increasing value of N :



As expected, the collective program show a better scalability than the non collective one (which start to not scale from 16 processors on). This is cause by the fact that the time required to carry on the calculation differs heavy for each processor, increasing the synchronization time required.

The distance with the ideal model increase with P.

In the program implemented, every P read simultaneously the value of N, so the communication time is $T_{comm} = (P-1) \times T_{single\_comm}$

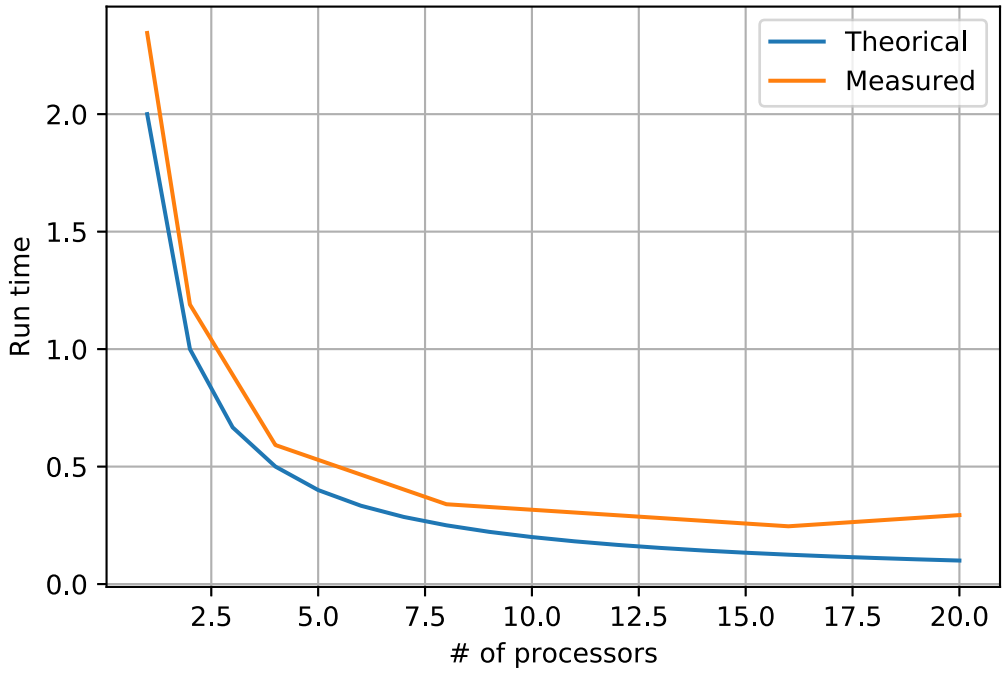
In the collective program the communication time is instead just T_{single_comm}

Given the notably better performance, I will just consider the collective program from now on.

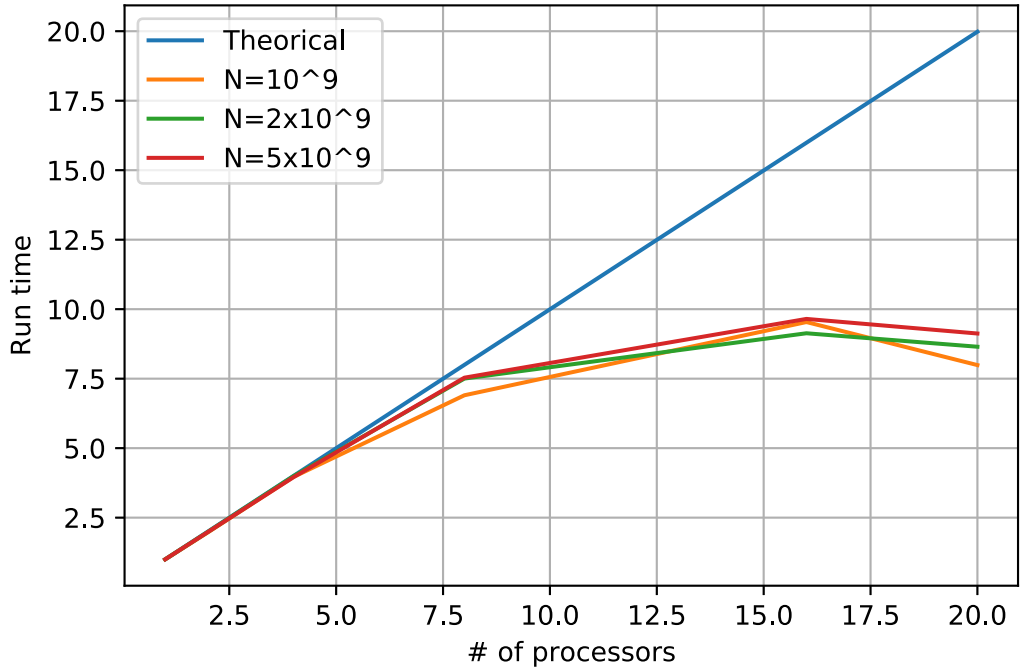
Considering what has been already said in section 1, the theoretical model for the runtime and for the speedup are:

$$T(P) = T_{comp} \times (P - 2 + n/P) + T_{read} + 2 \times T_{singel\_comm}$$
$$SU(P) = [T_{comp} \times (N-1) + T_{read}] / [T_{comp} \times (P - 2 + n/P) + T_{read} + 2 \times T_{singel\_comm}]$$

Non considering the synchronization time, the real run time (for $N = 10^9$)compared to the one theorized is the following:



The speedup, plotted for $N = 10^9$, $N = 2 \times 10^9$, $N = 5 \times 10^9$ is:



As said before, the synchronization time infers heavily on the scalability of the program.