Università degli Studi di Trieste

dipartimento di matematica e geoscienze

Corso di Laurea Magistrale in Data Science and Scientific Computing

High Performance Computing: Assignment 1

Student:
Pietro Morichetti - SM3500414

Anno Accademico 2019-2020

Section 0: About performance

Question 0 - Compute Theoretical Peak performance for your laptop/desktop.

In the following table there are some information about the feature of my personal laptop.

	The model	CPU	Frequency
My Laptop	Acer, Aspire E15, E5-571G-597D	Intel Core i5-5200U	2.20 GHz
	Number of cores	**Number of floating point**	**Peak Performance**
	2	8	70.4 GFlops/s

In particular, the number of floating point is calculated by using:

- The number of operation: fma and avx2, two basic operation for each of them;
- The number of circuits for cores (ncc): the circuits that performes the fma and avx2 operations, in that case we have 2 circuits for each cores.

In conclusion the number of floating point is determined as $fma \cdot avx2 \cdot ncc$ equals to eight, while the peak performance is given by the formula in the HPC slides.

Question 1 - Compute sustained Peak performance for your cellphone.

In the following table there are some information about the feature of my personal cell-phone given by the Mobile Limpak, an application set upfor android devices.

	The model	Size of matrix	CPU	Real Performance
My SmartPhone	Xiamo Redmi Note 2	2000	Mediatek MT6795	233.65 Mflops/s
	Memory	**Number of Cores**	**Number of floating point**	**Peak Performance**
	16 Gb	8	24	48 GFlops/s

In particular, the number of floating point is the result of the product between: number of cores, frequency of 2 GHz and floating point (4 cores are ARM Cortex-A32 with 2 operations per cycle and the last 4 cores are ARM Cortex-A57 with 4 operations per cycle).

Question 2 - Find out in which year your cell phone/laptop could have been in top1% of Top500.

There is none reference for the laptop.

	The model	Performance	Top 500 year
My Laptop	Acer, Aspire E15, E5-517G597D	70.4 GFlops/s	among 4° and 5° position in 1994
My SmartPhone	Xiamo Redmi Note 2	48 GFlops/s	VPP500/30 in 1994

	Number 1 HPC system	Number of processor (TOP500)
My Laptop	Numerical Wind Tunnel	30
My SmartPhone	Numerical Wind Tunnel	30

Section 1: Theoretical model

Device a performance model for a algorithm of the sum of N numbers and P processor, with P ranging from 1 to 1000.

- Serial Algorithm (N-1 operations): $T_{serial} = N \cdot T_{comm}$;
- Parallel Algorithm (Master-Slaves): $T_{parallel} = T_{comp} \cdot \left(P - 1 + \frac{N}{P}\right) + T_{read} + 2 \cdot (P-1) \cdot T_{comm}$.

Where: $T_{comm} = 10^{-6}$, $T_{read} = 10^{-4}$ and $T_{comp} = 2 \cdot 10^{-9}$ seconds.

The scalability curves for this algorithm is result of the following image (figure 1), in which the scalability respect of different values of N is considered: 10^8 for the blue one, 10^9 for the orange one, 10^{10} for the green one and 10^{15} last one.

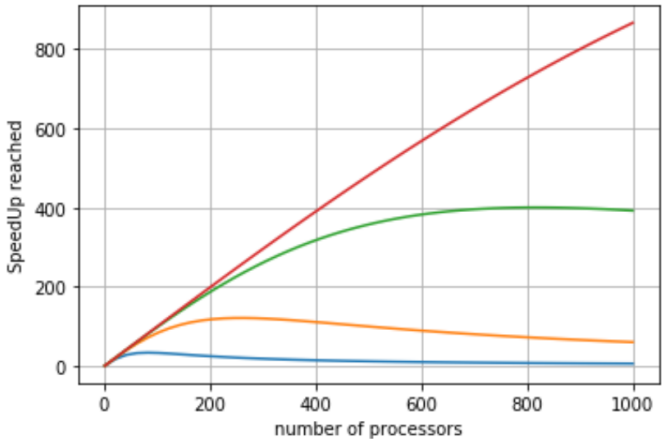

Figure 1: Scalability curve

Question 0 - For which values of N do you see the algorithm scaling ?

In order to answer this (and the next) question, it may be necessary to use other kind of plots instead of the previous one (figure 2). Where for a fixed number of processors (i.e. two processors) the plot shows the scalability function; in detail the algorithm is scaling when $N = 10^5$.

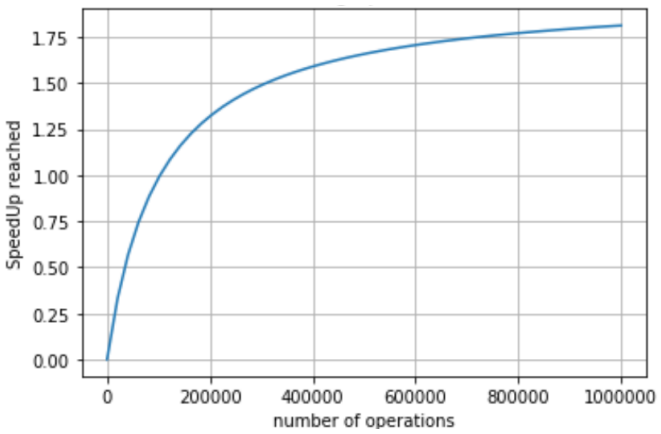

Figure 2: Algorithm scaling

Question 1 - For which values of P does the algorithm produce the best results ?

In that case, it is necessary to consider the number of the operations fixed at 10^5, then the graphic below will be obtained (figure 3).

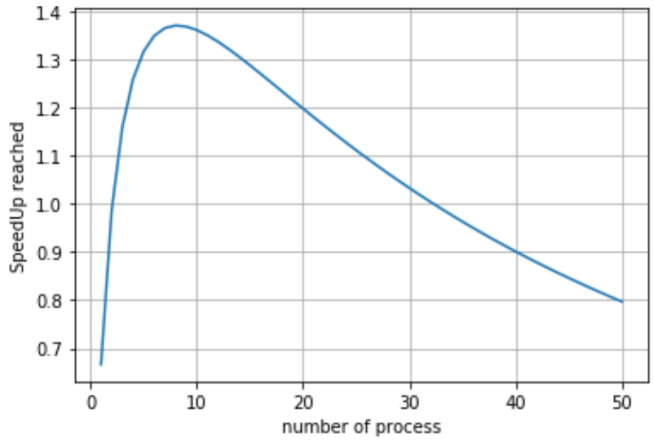

Figure 3: Best scalability

The blue line represent the SpeedUp in accordance with the number of processors; in particular, the curve reaches the maximum value with eight processes, than it decreases.

Question 2 - Can you try to modify the algorithm sketched above to increase its scalability ?

First of all, it is necessary to define: N($= K \cdot P$) the number of the operation, P the number of processes and to start from the first part of this algorithm:

the sending procedure.

In order to increase the scalability you have to work on the communication time as the following procedure. First step is given by shipping the dimension of sub-problem K to the first slave (called S_1). Then, these two processes send to other two slaves that don't know the K value, and you go on in iterative way which every processes knows about the problem. In particular, you can consider this kind of communication as a "binary step communication", such that in every iteration there are 2^i processes that knows about K. In the end you have $2^i = P$, that is $i = log_2(P)$ and it's a top-down estimation; the image below (figure 4) explains the send procedure.

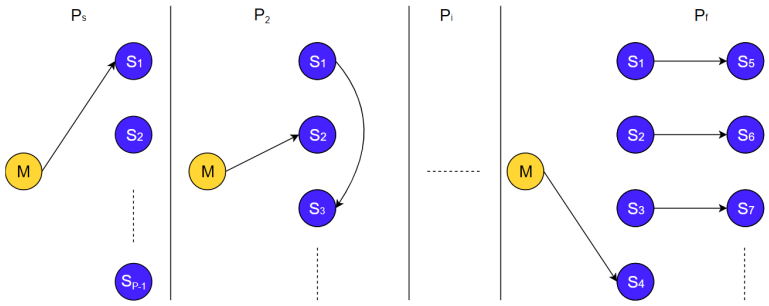

Figure 4: Idea of possible algorithm improvement - part one

Now, will be shown the second part of this algorithm: the receiving procedure.

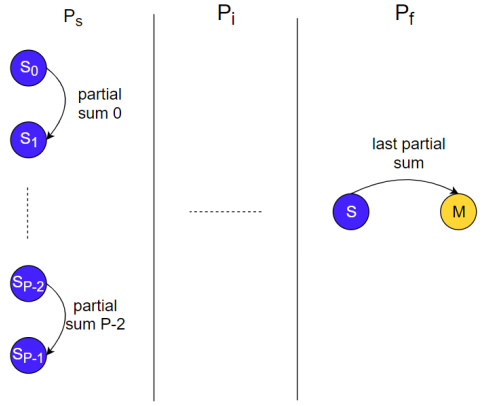

Figure 5: Idea of possible algorithm improvement - part two

From the sending procedure all the processes knows about the sub-problem. Now, Consider the picture 5 during the time P_i everyone make the sum operation obtaining one single value. Look that, during this first phase the communication time is T_{comm} the computation time is equal to $K \cdot T_{comp}$, and the master node makes the same operation of the other nodes

(as a slave).

In the phase P_1, half processes send to the other half its own results (called *partial sum <processID>*) and who receives the value makes the sum operation with the own value; during this phase the communication time is T_{comm}, the computation time is equal to $2 \cdot T_{comp}$ and only $\frac{P}{2}$ proccesses adding up.

You have to iterate this least phase until the computational time is equal to $1 \cdot T_{comp}$, i.e. during the P_f phase where there are only one slave sendind its own results to the master, who makes the last operation. Summarizing all the procedure you obtain the table above.

Phase	**Active nodes**	**Problem size**	**Communication**	**Computation**
start	P	N/2	T_{Comm}	$K \cdot T_{Comp}$
1	P/2	N/4	T_{Comm}	$2 \cdot T_{Comp}$
i	$P/2^i$	$N/2^i$	T_{Comm}	$2 \cdot T_{Comp}$
final	2	0	T_{Comm}	$1 \cdot T_{Comp}$

The total number of the interation in the P_i phase is equal to $log_2(P)$ because $\frac{P}{2^i} = 1$, when $i = log_2(P)$. Then using this alghoritm the parallel time is:

$$T_p = \frac{N}{P} \cdot T_{Comp} + 2 \cdot \left[\sum_{i=1}^{log_2(P)} (T_{Comp} + T_{Comm}) \right] + T_{Read}$$

Now, this result and the parallel time gaves at the beginning of this section differ by a value that you must compare:

$$log_2(P) >\overset{?}{=}< P - 1$$

For $P = 2$ they are the same, while for $P > 2$ or $P = 1$ the first element is minor than the second element (the $P \leq 0$ case is not considered). In conclusion the algorithm explained above is more efficient, in every case.

Note: if the number of processes is an odd number, then you have to consider the lower whole part of $log_2(P)$.

Section 2.1: Play with MPI program

The application we are using here is a toy application: a Monte-Carlo integration to compute PI. We provide a basic serial implementation of the algorithm (program pi.c) and we also give a parallel MPI implementation (mpi_pi.c).

Question 0 - Determine the CPU time required to calculate PI with the serial calculation using 1000000 (10 millions) iterations (stone throws). Make sure that this is the actual run time and does not include any system time.

In order to answer the question, you have to run the serial version code by using the */usr/bin/time* command to time all the applications.

```
[pmoriche@login2 code]$ /usr/bin/time ./pi.x 1000000

 # of trials = 1000000 , estimate of pi is 3.141224000

 # walltime : 0.02000000
0.02user 0.00system 0:00.02elapsed 92%CPU (0avgtext+0avgdata 1936maxresident)k
0inputs+0outputs (0major+143minor)pagefaults 0swaps
```

Figure 6: Serial times

The figure 6 performs several kind of times, but in this case the *usertime*, the *walltime* and the *elapsedtime* are more interesting:

- usertime: only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.
- walltime: the actual amount of time taken to perform a job and can be effected by anything else that the system happens to be doing at the time.
- elapsedtime: the actual time taken from the start of a computer program to the end. In other words, it is the difference between the time at which a task finishes and the time at which the task started.

For all of three them is exactly 0.02 seconds, because they are the same thing (in this case).

Question 1 - The parallel code writes walltime for all the processor involved. Which of these times do you want to consider to estimate the parallel time ?

Getting the MPI code running for the same number of iterations, you can observe the above figure 7 in which is rappresented each walltime for all the processor involved.

In this case the last one process which complete the job is the slave number 9, but it might not be the same if you run it again. In fact it dependes on the default behaviour decided in the MPI library, but in general: if the size of the message is small than the message is sends to the system buffer which will manage the submit; otherwise the process have to wait until the recipient has recevied the email.

In any case it is the right way to consider the least process, among slaves and master, as the reference point for the estimation of the parallel time.

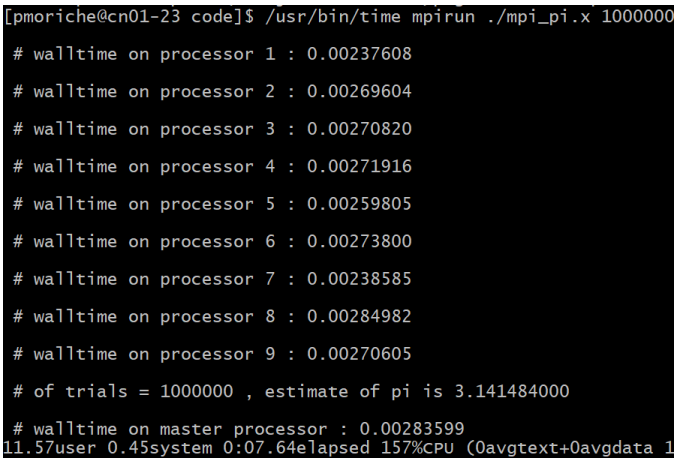

Figure 7: Walltimes for mpi_pi program

Question 2 - First let us do some running that constitutes a strong scaling test. A comparison of this to the serial calculation gives you some idea of the overhead associated with MPI. Again what time do you consider here ?

In this case you have to run the MPI code with only one process and the serial code too; the image 8 gaves you different times.

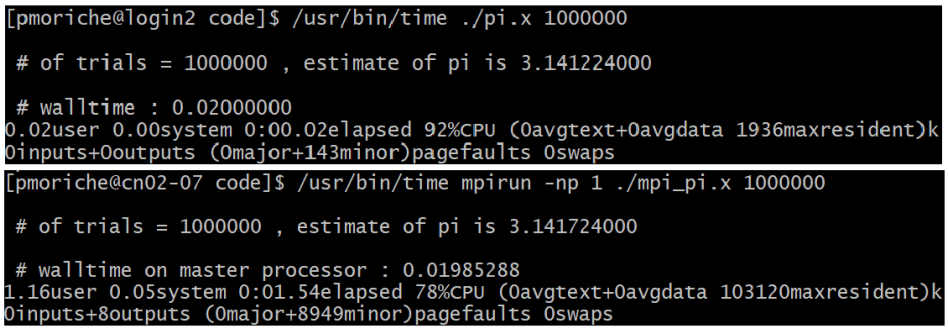

Figure 8: Comparison of the serial and parallel times

In this case you can see that the overhead associated with MPI is given by:

$$overhead = elapsed\_time_{mpi} - \frac{user\_time_{mpi} + sys\_time_{mpi}}{number\_processes}$$

Which is equal to 0.40 seconds, then exist an overhead not be entirely negligible, while using the same formula in the serial case you will obtain an overhead equal to zero, which is correct.

Question 3 - Keeping Niter = 10 millions, run the MPI code for 2, 4, 8 and 16 and 20 MPI processs.

In principle, all that needs to be done is to run the program multiple times, changing the -np argument to mpirun each time; therefore, you have to implement a strong scaling script as below.

```
typeset -i N=${1}
typeset -i tmp=${2}

for procs in 1 2 4 8 16
do
 /usr/bin/time mpirun -np ${procs} ./mpi_pi_modified.x ${N}
           1> ./a.txt 2> ./b.txt
 cat ./b.txt | grep elapsed | cut -d" " -f3 |
           cut -d"e" -f1 | cut -d":" -f2 >> section_2_1_elapsed-${tmp}.txt
 sort a.txt -r | head -1 >> section_2_1_maxwalltime-${tmp}.txt
 rm a.txt
 rm b.txt
done
```

First of all, the mpi_pi is modified to print only the walltime values for each process (it means that only the printf instruction are modified, the rest remain the same of the original one).

So, the bash script imposes the exectution of the mpi_pi_modified program for a set of procesess, and for each iteration, pull out the elapsed time using a sequence of pipelined instruction (note that in input it receives the dimension of the problem and an integer value to discern from other files with different problem's size). Therefore, the same script creates another one file which saves the maximum walltime for each number of processes considered (may be useful).

Question 4 - Make a plot of run time versus number of nodes from the data you have collected

Thanks to the script in the previous question, you can collect an elapsed time and walltime dataset, which in $N = 10^7$ as in figure 9. Here, after few processes the scaling doesn't work anymore and the time increase very quickly, better still the elapsed time growing as a straight line graph; therefore, is suitable don't increase the number of processes accross a cert threshold. Viceversa, the walltime decreases as processes increase which proves how the computational weight is divided by all the processes. In particular, this feature is well rappresented by the straight line graph; however using the walltime you don't perform the overhead such that knock over the efficiency, as well shown by the SpeedUp graph realized using the elapsed time.

Note that the elapsed-processes graph and the SpeedUp-processes graph (using the elapsed time) are one mirrored by the other.

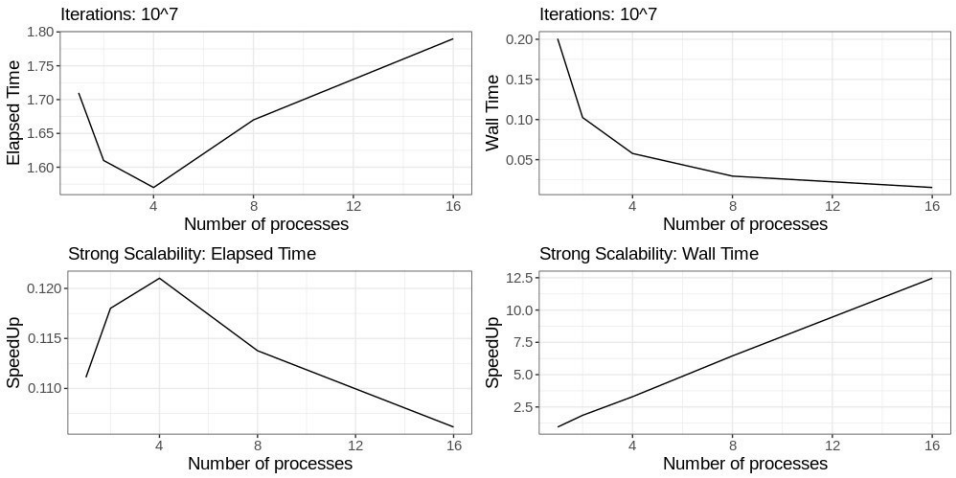

Figure 9: Run time versus number of nodes and strong scalability using the elapsed time (in the left side), and using the walltime (in the right side)

	Subset of Processes				
	1	2	4	8	16
Size	**Elapsed Time (s)**				
10^7	1.8	1.73	1.71	1.78	1.91
10^8	3.47	2.57	2.06	1.89	1.91
10^9	21.3	11.74	6.68	4.3	3.17
10^{10}	202.24	103.83	54.18	29.95	16.35

Table 1: Strong scalability for different size

Question 5 - Provide a final plot with at least 3 different size and for each of the size report and comment your final results

Consider as the three different size of the problem 10^7, 10^8 and 10^9; so you will provide the elapsed time in the following table 1.

That will gives the figure below (figure 10).

In particular these plots are a «zoom in» version of the first one plot, in other words the rise of the problem improves the scalability. In fact, the first left graph shows which just after eight processes the algorithm doesn't scale anymore, actually starts to appear the overhead which decrease the efficiency. While incresing the size of the problem the algorithm scaling still if the number of processes increased and there'is an inprovement in efficiency.

Section 2.2: Identify a model for the parallel overhead

In order to estimate the parallel overhead used the following parameters:

- S: serial section (unknown);

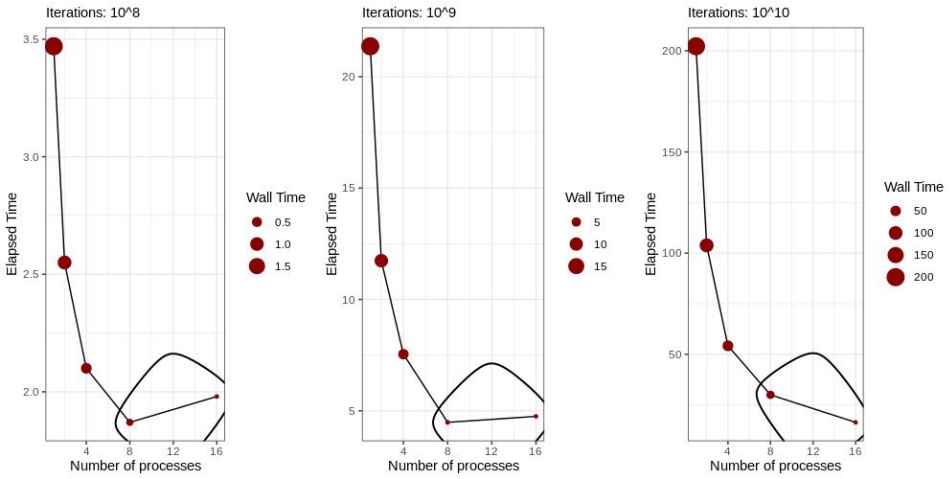

Figure 10: Strong scalability for three different size of the problem

- s: serial section added from communication instructions (unknown);
- pp: potentially parallelizable, as instance the computational section unknown);
- P_i: i processes used in the same time (controllable);
- T_{P_i}: the real time supplied using i proccesses (determinable).

So, the model for the present program is given by:

$$T_{P_i} = S + s \cdot P_i + \frac{pp}{P_i}$$

In order to determine the three unknown parameters you have to run the program for three different number of processes (as for example 1, 8, 16) and solve the system equation showed below.

$$\begin{cases} T_{P_1} = S + s \cdot P_1 + \frac{pp}{P_1} = \\ T_{P_8} = S + s \cdot P_8 + \frac{pp}{P_8} = \\ T_{P_{16}} = S + s \cdot P_{16} + \frac{pp}{P_{16}} = \end{cases} \Rightarrow \begin{cases} S = 0.894 \; s = 0.085 \\ pp = 0.73 \end{cases}$$

In particular, the overhead is given by the parameters s for each processes that involved in the program. Therefore, the picture above (figure 11) gives you an idea about the real time trend in function of processes, for different number of processes.

In first section the curve decrease when P_i increase, because the $\frac{pp}{P_i}$ is significant yet; while in the second section (after the minimum point) prevail the overhead and the elapsed time grows up quickly.

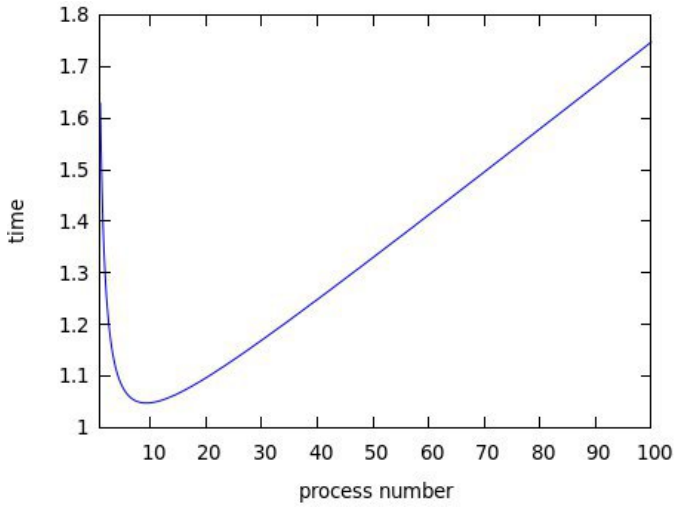

Figure 11: Parallel overhead

Section 2.3: Weak scaling

Question 0 - Record the run time for each number of nodes and make a plot of the run time versus number of computing nodes

The following plot (figure 12) shows not only the relationship between run time and number of computing nodes, but also the weak scalability for the Question 2 on the same section.

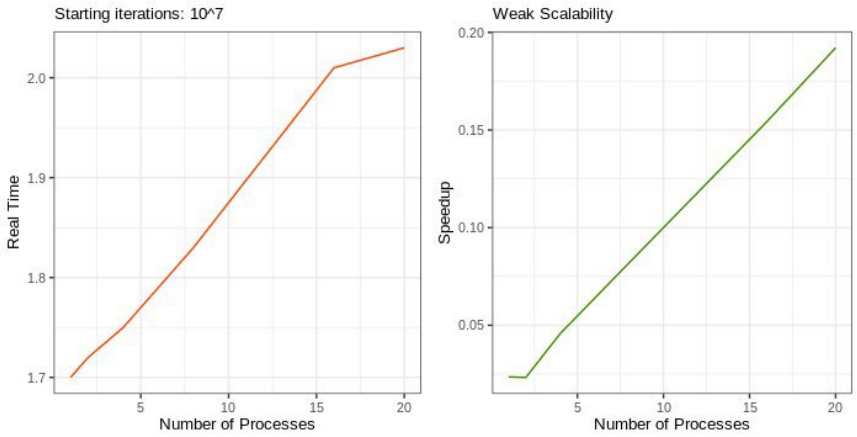

Figure 12: Run time vs number of processes on the right side, weak scalability on the left side

Question 1 - Weak scaling would imply that the runtime remains constant as the problem size and the number of compute nodes increase in proportion. Modify your scripts to rapidly collect numbers for the weak scalability tests for differrent number of moves.

Consider the script bash of the previous sub-section, in this case you have to modify it as below.

```
typeset -i N=${1}
typeset -i tmp=${2}

for procs in 1 2 4 8 16
do
 /usr/bin/time mpirun -np ${procs} ./mpi_pi.x ${N} 2> ./a.txt
 cat ./a.txt | grep elapsed | cut -d" " -f3 | cut -d"e" -f1 |
               cut -d":" -f2 >> ./section2_3-${tmp}.txt
 rm a.txt
 N=$(($N*2))
done
```

Now, the script starts with a given fixed value of N and, in every iteration, increases the problem size and the number of processes in the same way, i.e. $N_{new} = N \cdot \frac{P_{current}+1}{P_{current}}$. Therefore, using as input 10^7, you will collect the data showed in the following table.

Size	Number of Processes	Elapsed Time (s)
$1 \cdot 10^7$	1	1.7
$2 \cdot 10^7$	2	1.72
$4 \cdot 10^7$	4	1.75
$8 \cdot 10^7$	8	1.83
$16 \cdot 10^7$	16	2.01

Even if the number of processes and the problem size are increasing proportionally, the run time doesn't remain constant inasmuch it would like remains constant but it's not possible a cause the present of the overhead.

Question 2 - Plot on the same graph the efficiency (T(1)/T(p)) of weak scalability for different number of moves and comment the results obtained

The figure 13 shows: the real time, the SpeedUp and the efficiency changing the problem size, respectively. In particular, the weak scalability graph is very similar to the theoretical weak curve and it means the ratio between problem size and number of processor remains nearly constant proportionally. However, has been registered a light increment of the walltime because of the problem size increment. Therefore, the SpeedUp graph shows how the ratio between the SpeedUp and the number of processes remains constant and equals to 0.010. Accordingly, the efficiency have to decrease very

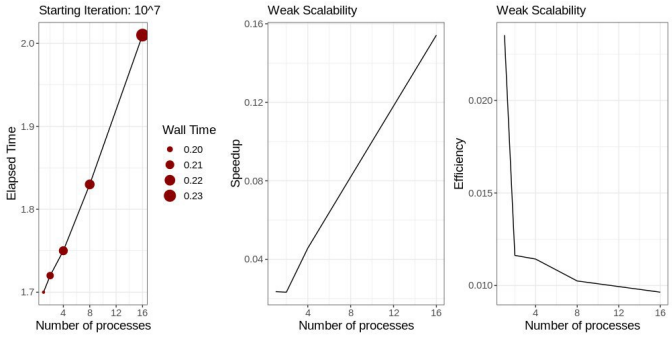

Figure 13: Weak Efficiency

quickly as number processes increase, then remains constant around that value; finally, it means that just after two processes is not suitable increase the number of processes because the algorithm dosn't scale very well.

Section 3: Implement a parallel program using MPI

This section shows a brief explanation of the mpi_sum program that you will found in the codes folder. The task establishs the master processes reads the dimension of the problem from a file .txt (called *number_ operation*), and sends to all the slaves this value, which will call *dim*.

Hereafter, everyone (included the master) divide dim by the number of all processes that involved in the code in order to determine N, size of the sub-problem and dimension of an array. This array has to collect a specific sequence of numbers which the particular process have to sum up like a *partial sum* of the total problem. However, exist the possibility that some processes have created an array with dimension N and some with N+1, this different accurs because you have to consider dim might not be a multiple of procesess number (which means exist a rest R); so, those how have the length equals to N+1 have to sum up an additional number.

Note that those have to consider the rest are always the processes with rank minor of the remainder, moreover everyone use its own rank to understand from what value to wich value have to fill up the array (code is more clarified in this sense).

When it's done, every slave send its own result to the master process, which have to collect these value into another array and add up them all up to compute the final score.

Note that in order to measure the computation time of the master process, the communication time has been subtracted from the walltime. However, if the problem size is too low (as instance $N = 10000$ or more) the master process takes more time to communicate than to perform the computation; the result is a negative computation time. In any case, this thing is

not necessary wrong since it may be understood as a sign the size is too small, or that there are too many active processes (and you can act accordingly).

The mpi_collective_sum program does the same things with the collective mpi function.

Section 4: Run and compare

Question - Plot as in section 3 scalability of the program and compare performance results obtained against the performance model elaborated in section 2

In order to answer the question you have to determine the SpeedUp of the mpi_sum program and to do that you have to achieve the same algorithm in the serial way (see the codes folder). Having executed the serial_sum and having taken the elapsed time you can calculate the strong scalability for the naive mpi sum and collective mpi sum.

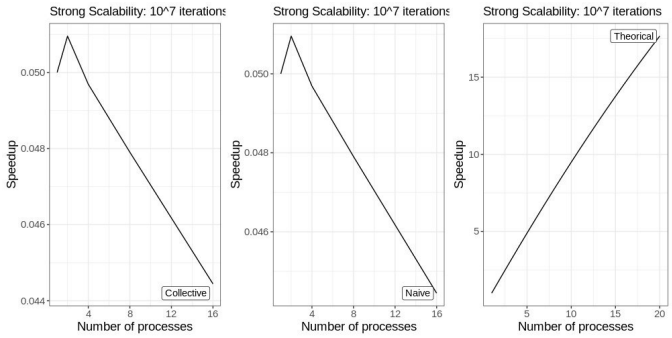

Figure 14: Strong scalability of the mpi_sum program with $N = 10^7$

At this point you can observe, in the figure 14, how the naive and collective version are very similar and it means that the collective communication does not improve anything. However, the strong scalability graphs shows you the programs don't scale at all, or at least they only scale until they reach the second process.

While using a 10^8 sized problem, the figure 15 registers a light improvement in term of SpeedUp for both of naive and collective program, even if they are not commensurate with the theoretical strong scalability.

Question 2.1 - Comment on the assumption made in section 1 about times: are they correct ? do you observe something different ?

Recording the assumpion in the section 1: $T_{comm} = 10^{-6}$, $T_{read} = 10^{-4}$ and $T_{comp} = 2 \cdot 10^{-9}$ seconds, and in order to answer the quesions you have to

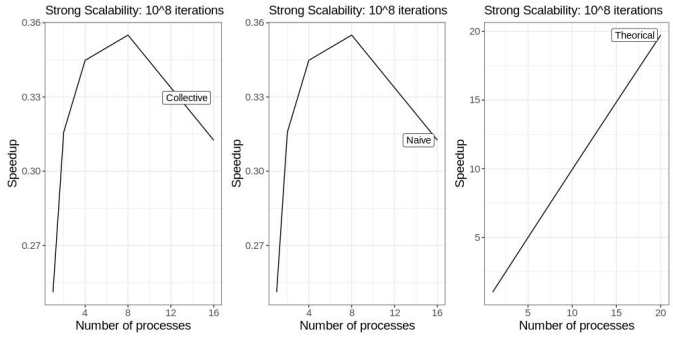

Figure 15: Strong scalability of the mpi_sum program with $N = 10^8$

collect all the times for a different number of processes for a fixed $N = 10^7$ size of the problem. In particular, these data are correct in the theoretical point of view, because: the T_read remains constant as the constant serial section, the T_comp decreases as processes increase and T_comm increases as processes increases.

So, by making the weighted average for each kind of times you will obtain the following results: $T_{comm} = X \cdot 10^{-2}$, $T_{read} = x \cdot 10^{-4}$ and $T_{comp} = x \cdot 10^{-3}$ seconds; where 'x' means a generic value which is not important (more interests is the degree of the time). In conclusion, the T_read is equal to the T_read of the model, while the other two are greater than those of the model, i.e. three times larger.