Exercise 1 - Strong scalability VS Weak scalability

Two different C scripts were given implementing a Monte Carlo approximation of pi:

- **`pi.c`**, serial implementation
- **`mpi_pi.c`**, parallel implementation using OpenMPI

It's required to show Amdahl's law in two of its aspects:

- strong scalability, in which the size of the problem is kept constant while the parallelization increases
- weak scalability, in which the size of the problem increases accordingly to the increase in parallelization

In both cases, it's interesting to look at the speedup for each level of parallelization within the experiment:

Given *p* number of processors employed to run the experiment,

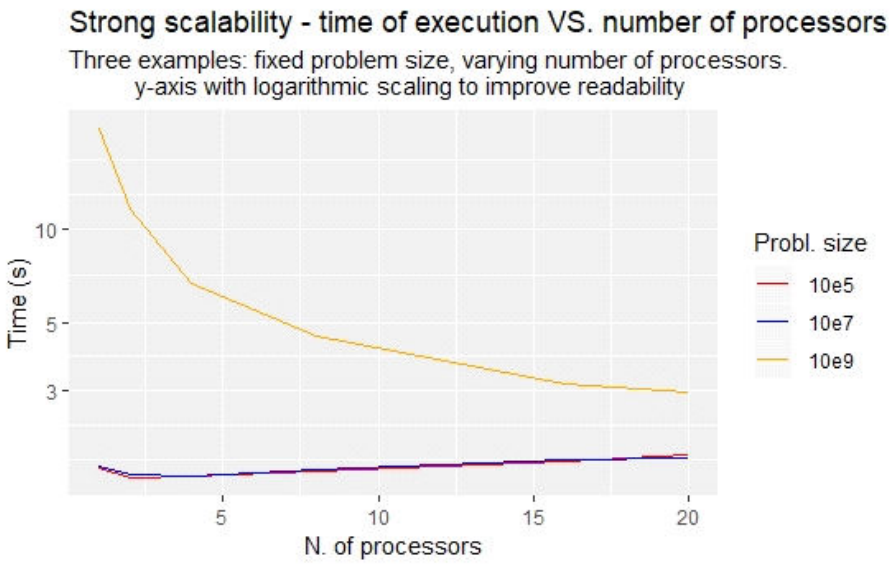
$$\textbf{\textit{Speedup}}_p = \textbf{\textit{Time}}_1 / \textbf{\textit{Time}}_p$$

Since the difference in timing between the serial implementation and the parallel implementation ran on 1 process only is large (~100x larger), I decided to keep the MPI version as benchmark for the $\textbf{\textit{Time}}_1$ part of the formula.

STRONG SCALABILITY TEST

To test strong scalability on the problem, I compiled & ran the program `mpi_pi.c` on a varying number of processors while keeping the problem size, i.e. the number of simulations to compute the approximation of pi, constant.

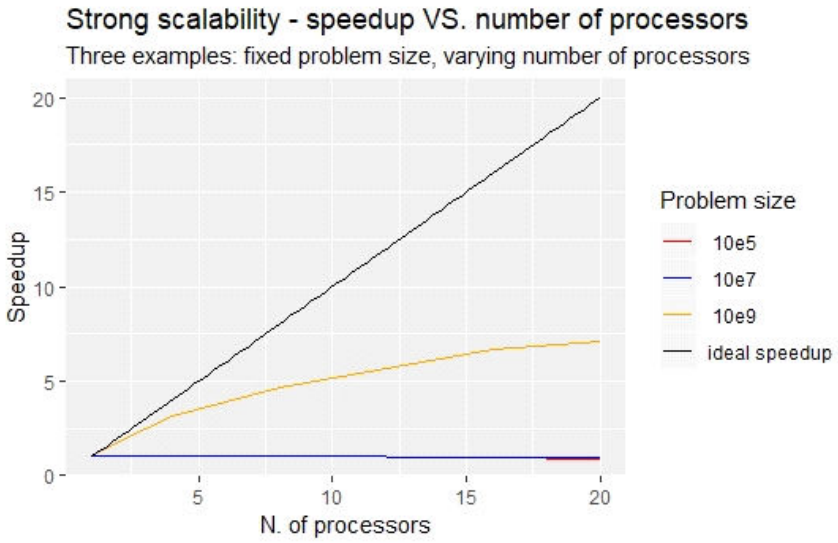
The experiment was repeated for three different values of problem size (100.000, 10.000.000, 1.000.000.000 iterations) while recording time of execution; these are the results I got:



The first two problem sizes yielded almost the same timings, with the time for 1 processors almost equal for both cases (~1.7 s) slightly decreasing to ~1.6 s with 2 processors, slowly increasing up to ~1.8 s when we employ 20 processors.

For the problem size of 1G simulations, instead, we get a very large execution time of ~20 s for the serial execution, dramatically decreasing to ~6 s at 4 processors, eventually taking ~3 s to run for 20 processors.

I calculated the speedup for the same experiment:



In black it's the ideal speedup one would expect if the whole program could be ran all in parallel, i.e. the program should ran in $\boldsymbol{Time_1/p}$; unfortunately, our program can't be fully ran in parallel: there's a (small) part of serial code, plus there's a huge overhead associated with the *message passing* between processe, which introduces *barriers* in the code in which one or more processors must stop and wait other processors to complete given tasks. This overhead increases as the number of processors

increases, and this can be noted in the chart above:

- in all problem sizes, the deviation between the ideal speedup and the actual speedup observed increases as we increase the number of processors: this means that the parallelization overhead gets larger
- moreover, the problem sizes 10e5 and 10e7 yield almost overlapping curves, recording very low values of speedup; on the other hand, the 10e9 speedup curve (orange) shows much better results: this means that, for the former two problem sizes, the amount of parallel work is very low w.r.t. the serial work and the message passing; instead, by increasing the problem size we achieve much better results, with small difference from the ideal speedup even with 4-8 processors. The same observation can be done on the timings chart, where the 10e9 problem size is the only one in which timings actually improve (i.e. get lower) each time we increase the number of processors: eventually, if we had enough processors and time at disposal, what we'll have is that the parallelization overhead would prevail, and we'll record an increasing time of execution as we increase the number of processors.

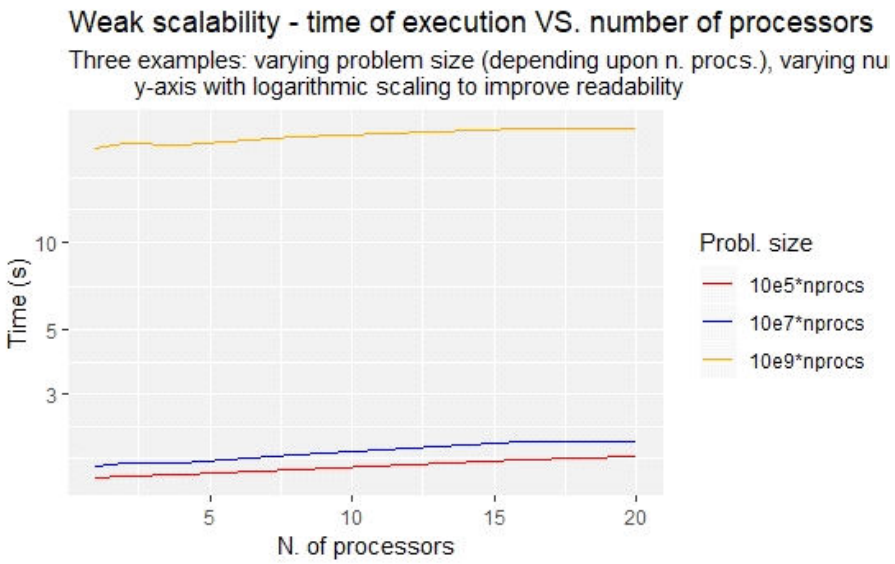
Compilation & run (minimal code)

```
#compile

mpicc parallel_pi

#run (w/ 1G iter)

niter_tot=1000000000

for procs in 1 2 4 8 16 20; do

     #since each processor does niter simulations, niter is the total problem size

     #divided by the number of processors

     time mpirun -np $procs ./a.out $((niter_tot/procs))

done
```

WEAK SCALABILITY TEST

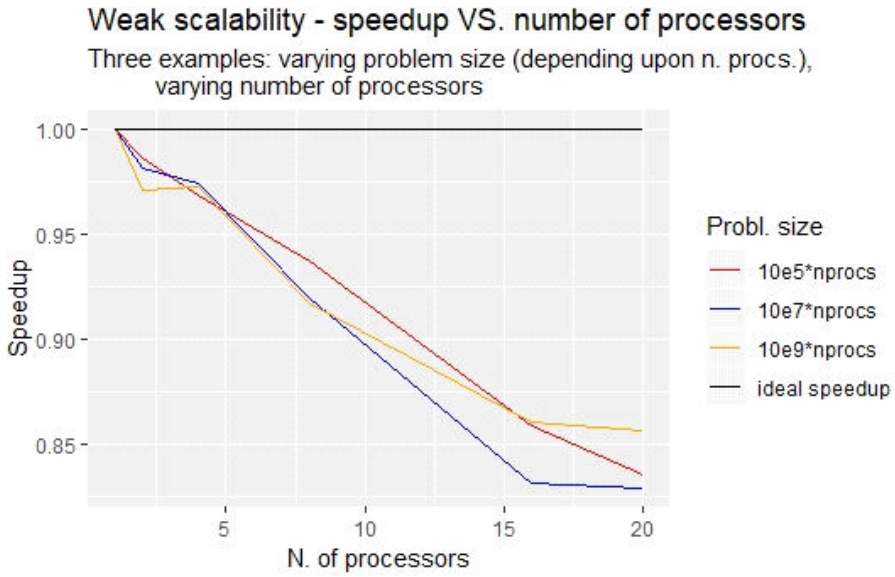
To test weak scalability, I compiled & ran the program `mpi_pi.c` on a varying number of processors while varying the problem size w.r.t. the increase in processors.

The experiment was repeated for three different values of problem size (100.000, 10.000.000, 1.000.000.000 iterations) while recording time of execution. These are the results I got:



Now we see that the three curves are almost a horizontal line, and the trends are very similar for all three problem sizes, while the timings for problem size 10e9 is much larger than the other two.

Let's check the speedup:



Again, in black is the ideal speedup: if the program would run all in parallel (without message passing) I would expect to have the program run in the exact same time, without deviations: that would yield a horizontal line as speedup: instead, what I see is that all three programs ran with an (similar) increasing time as processors number increased, thus resulting in a lower speedup (around 85% at 16 and 20 processors). Again, this shows that, as the number of processors increases, the overhead associated with parallelization is higher, while remaining constant w.r.t. the problem size: as we increase the number of processors, and adjust the problem size accordingly, the overhead due to serial code + message passing increases and is the same no matter what was the initial problem size.

Compilation & run (minimal code)

```
#compile
mpicc parallel_pi
#run (w/ 1G iter)
niter_initial=1000000000
for procs in 1 2 4 8 16 20; do
      #niter for the current problem size is niter_initial*procs
      #Hence, the number of simulations to assing to each processor is niter_initial
      time mpirun -np $procs ./a.out niter_initial
done
```

Is this problem more appropriate for weak or strong scaling?

As this program is very good for **explaining** both weak and strong scaling, if we consider speedup, the program is better at weak scaling since it achieves approximately the same speedup for any given initial problem size (85% for 20 processors), while, on the strong scalability, speedup for small problem size is abysmal (always remaning in the neighbourhood of 1 for a problem size of 10e5-10e7), and it needs to be pumped up pretty much (starting from 10e9) to obtain a fair result.