1 Theoretical model

The goal of this section is to analyze the performance model of the master-slave version of the parallel program which computes the sum of n numbers. Time for the serial algorithm ($n-1$ operations):

$$T_{serial} = nT_{comp} \tag{1}$$

Time for the parallel algorithm (master-slave):

$$T_p = T_{comp} \times (P - 1 + \frac{n}{P}) + T_{read} + 2(P-1) \times T_{comm} \tag{2}$$

where $T_{comp} = 2 \times 10^{-9}$ is the time to compute a floating point operation, $T_{read} = 1 \times 10^{-4}$ is the time the master takes to read and $T_{comm} = 1 \times 10^{-6}$ is the latency.

For small values of n, the time it takes to compute the sum of n numbers increases as the number of processors increases, probably because the time associated with communication when using a bigger number of processors dominates the actual computational time. For larger n, the time initially decreases when the number of processors is small, however as the number of processors increase, the time starts to increase, as shown in figure 1; this suggests that given n, it exists an optimal number of processors with which the program is most efficient.

Instead, for large values of n, the time decreases as the number of processors increases, as expected. In figure 2 it is shown the time with respect to the number of processors, the parallel speedup and the efficiency; the parallel speedup is defined as $S(p) = T(1)/T(P)$, where $T(1)$ is the serial time for the sum of n numbers, while the efficiency is $E(p) = S(p)/p$. For a perfect speedup ($S(p) = p$) the efficiency is $E(p) = 1$. From the graphs we can assume that the problem with large numbers is almost scaling perfectly, because the efficiency starts at 1, but as the number of processors increases it slowly decreases towards 0.9. Therefore, when n is large, we can say that the problem has a good scalability until 60 processors, from which it starts to decrease more significantly.

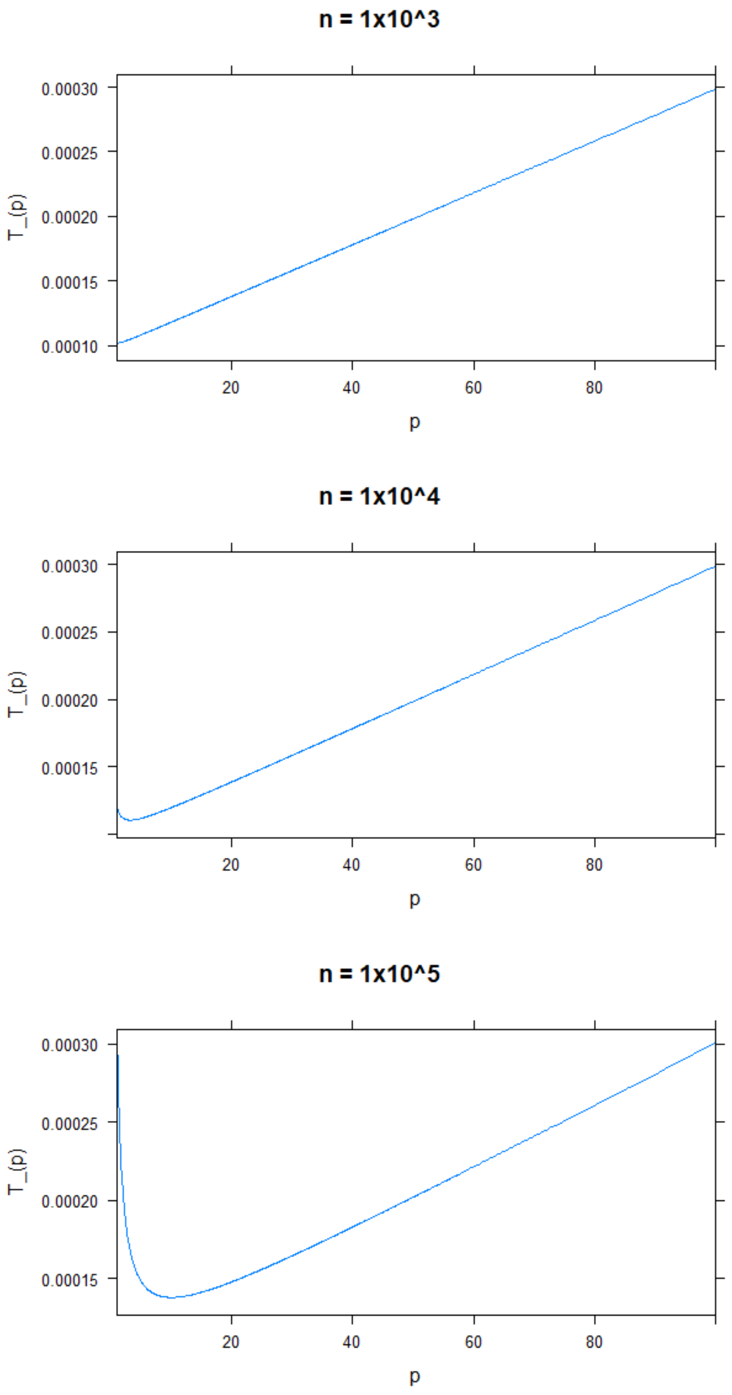
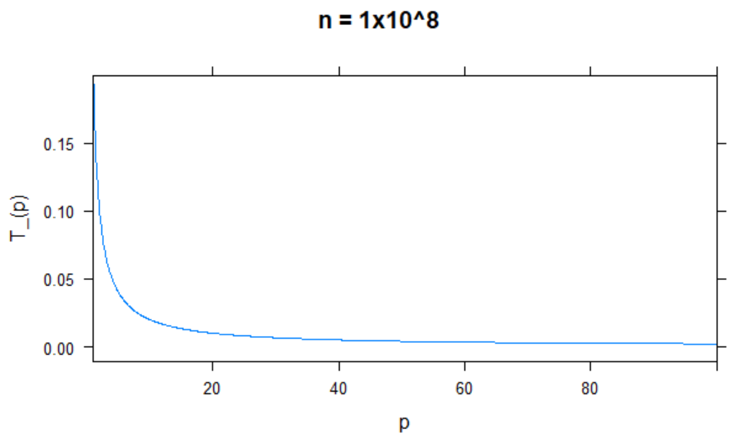
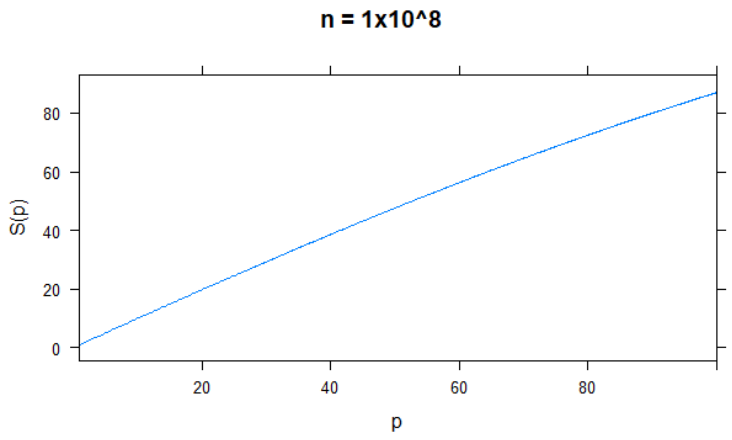


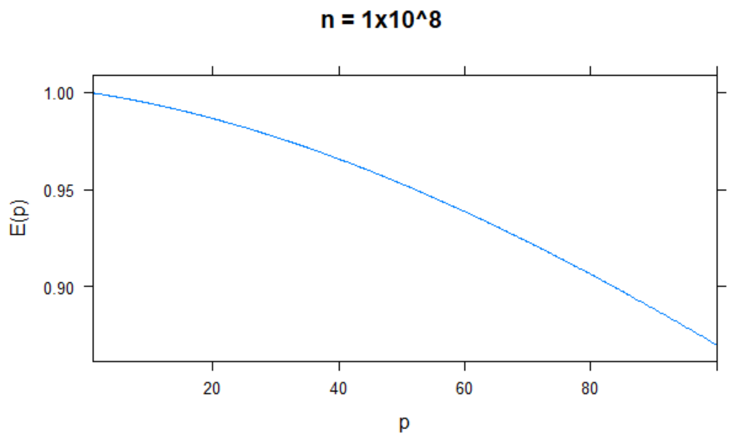
Figure 1: Time with respect of the number of processors p for the sum of n numbers



(a) Time with respect to the number of processors



(b) Parallel speedup



(c) Efficiency

Figure 2: Time, scalability and efficiency for $n = 10^8$

2 Strong and weak scalability

The goal of this section is to analyze how well an application scales up to the total number of cores in one node. This application has a serial implementation (*pi.c* program) and a parallel one using MPI (*mpi-pi.c*). These programs compute π using a Monte-Carlo integration: given a circle inscribed inside a square of unit lenght, the ratio between the area of the square and the circle is $\pi/4$. By generating n random points with a Monte-Carlo technique, on average, only $M = N \times \pi/4$ points will belong to the circle; from the last relation we can then estimate π. After analyzing the behavior of *pi.c* and *mpi-pi.c* on one processor, a strong and weak scalability tests are carried out.

2.1 1 processor

I compared the time for executing the *pi.c* program and the parallel version *mpi-pi.c* with 100 millions iterations on one processor with the bash script *1proc.sh*: it simply launches the two programs ten times and computes the /usr/bin/time; I then computed the mean. For *pi.c*, the mean is 2.984 s, while for *mpi-pi.c* is 3.235 s. This result suggest that there is an overhead associated with the MPI program, because of the time threads spend communicating with each other and waiting for each other to finish.

2.2 Strong scalability

To carry out the strong scalability test the size of the problem must remain constant (i.e. the number of moves for the Monte-Carlo integration), while increasing the number of cores utilized; the runtime represent the measure used to analyze the scalability.

In figure 3 is reported the time of execution for the parallel program versus the number of processors p, using different number of moves n; in blue it is shown the time calculated internally by the program, while in orange is reported the elapsed time computed with the command /usr/bin/time. For $n = 10^8$, the /usr/bin/time, while initially decreasing, starts to increase as the number of processors increase, suggesting that the communication time between processors has a greater impact; whereas the time calculated internally by the program approaches zero. A similar behavior can be seen when $n = 10^9$. However, for greater n the difference between the two methods for calculating the time becomes negligible, because the scale of times involved is much higher, as shown in the graphs with $n = 10^{10}$ and $n = 10^{11}$, where the orange and blue lines overlap.

In figure 4 is reported the speedup ($S(p) = T(1)/T(P)$, where $T(1)$ is the time of execution on one processor) versus p for different n, to analyze the strong scalability; the color legend is the same as for figure 3. In both cases

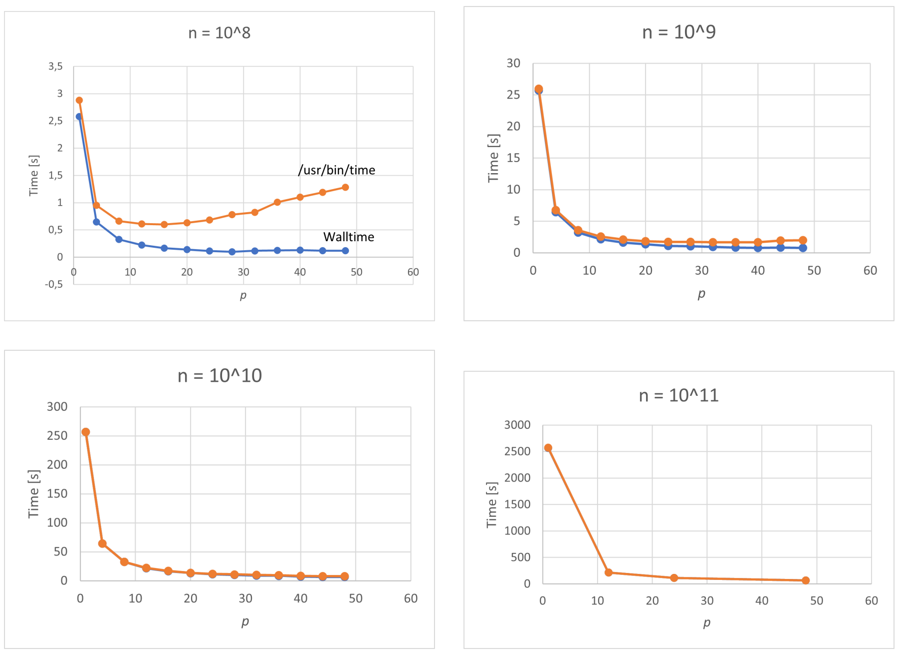


Figure 3: Time with respect of the number of processors p

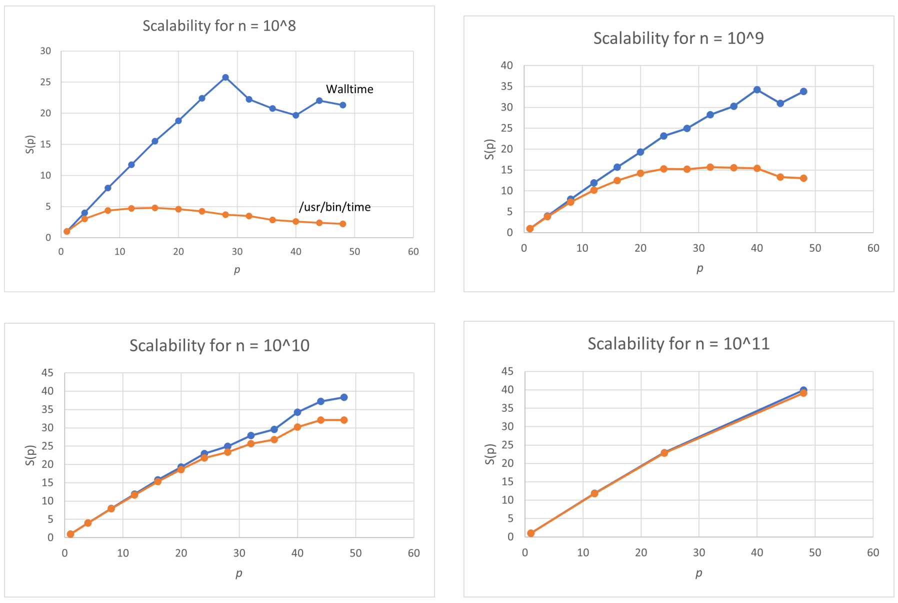
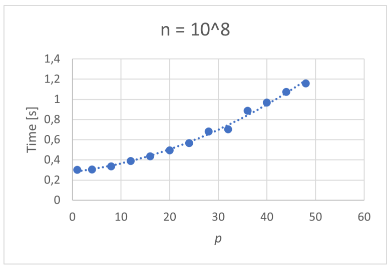


Figure 4: Speedup with respect of the number of processors p



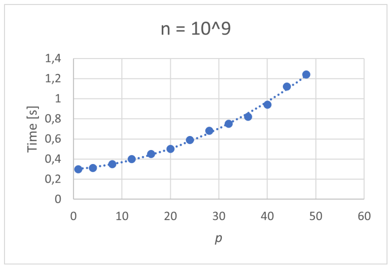


Figure 5: Difference between /usr/bin/time and walltime of mpi-pi.c with respect of the number of processors p

the scalability is not linear, and the speedup calculated with /usr/bin/time is much lower as p increases than the speedup computed with the walltime of the program, when the number of iterations is lower; nevertheless, for bigger n the difference between the two disappears as expected from the results in figure 3. For small n, the scalability starts to decrease for a smaller number of processors than for bigger n; this is probably due to the fact that when the number of iterations is small, the time associated with communication between the cores has a bigger impact than the actual computation time. When n is big, the scalability is almost linear until 20-30 processors, where it starts to decrease.

2.3 Model for parallel overhead

The graphs in figure 3 resemble the same functional form of the ones I reported in figure 1 for the theoretical model of the time for the sum on n numbers; for instance, for $n = 10^8$ and $n = 10^9$, it can be seen that the overhead increases as the number of processors increase. From Ahmdal's law, the mathematical formalization of the overhead associated with communication can be summarized in the formula

$$T(P) = s + \frac{p}{P} + c(P) \tag{3}$$

where c(P) can assume different functional forms depending on several factors such as network topology, size of the messages and communication pattern. By plotting the difference between the /usr/bin/time and the internal walltime, as shown in figure 5, it seems that this difference increases as the number of processors increase, suggesting that the communication overhead adds to the total time with a non linear contribution. However, I was not able to identify the exact mathematical dependency of the overhead associated with communication.

2.4 Weak scalability

To run a weak scalability test, the problem size must be increased simultaneously with the number of cores being used. In our case, for instance, if we compute π with the mpi-pi.c program with 10^8 moves on one core, then on 4 cores we must use 4×10^8 moves, and so on as the number of cores increases. Weak scaling imply that the runtime remains constant as the problem size and the number of processors increase proportionally. In figure 6 is reported the time of execution versus the number of processors; the blue line represents the walltime computed internally by the program while the orange one is the time computed with /usr/bin/time. The graphs show that the runtime slightly increases as the number of processors increases, probably due to the overhead associated with communication between more processors.

To measure weak scalability we define a sort of "weak efficiency" as the ratio $\frac{T(1)}{T(p)}$ between the runtime on 1 processor T(1) and the one on p processors T(p); for perfect weak scalability we expect this ratio to remain constant at 1. In figure 7 it is reported the weak efficiency for the different sizes of the problem. In every case, the ratio clearly decreases as the number of processors increases, meaning that there is not a perfect weak scalability, and the efficiency of the program decreases with the number of processors used.

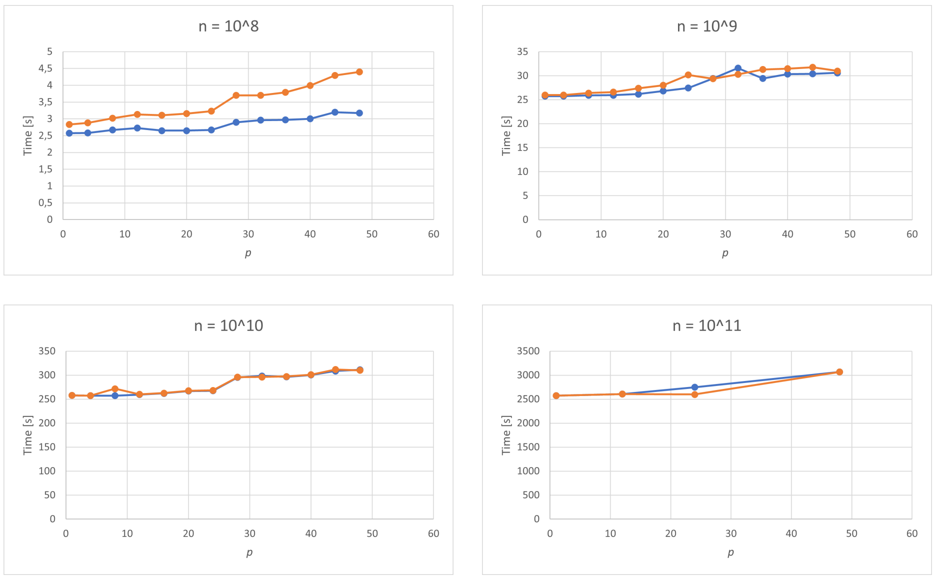


Figure 6: Time versus the number of processors p

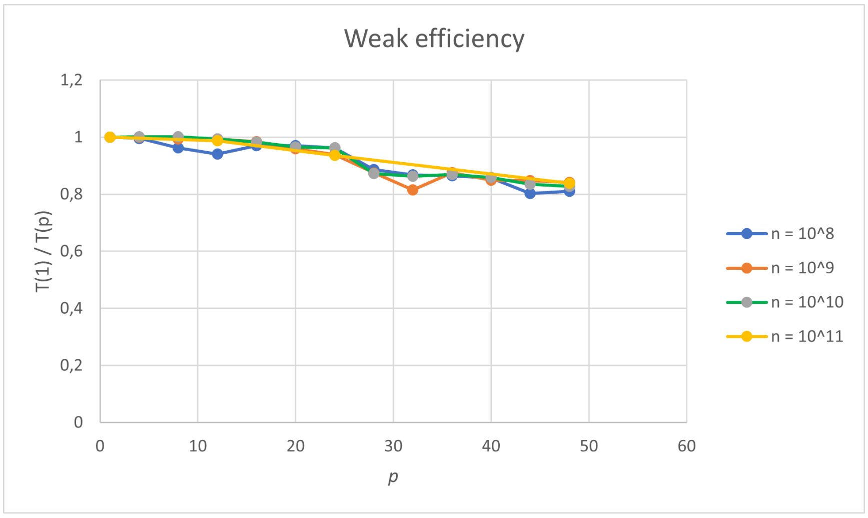


Figure 7: Weak efficiency versus the number of processors p