

FOUNDATIONS OF HPC

ASSIGNMENT 1

Section 1: Performance model

In this section, the aim is to evaluate, from a theoretical point of view, how the performance of a program changes when you vary the number of processors.

To do that, you can develop a theoretical program to solve a certain problem.

Here, for example, a simple program to perform the sum of N numbers has been developed. Both the serial and the parallel versions are available:

$$\text{Serial time} = T_{read} + N * T_{comp}$$

$$\text{Parallel time} = T_{read} + \left(P - 1 + \frac{N}{P}\right) * T_{comp} + 2 * (P - 1) * T_{comm}$$

where:

- T_{read} is the time required by the master processor to read the data (set equal to 10^{-6}),
- T_{comp} is the time needed to perform a floating-point operation (set equal to $2 * 10^{-9}$),
- T_{comm} is the time each processor takes to communicate a message (set equal to 10^{-6}),
- P is the number of processors.

Figure 1.1 (below) contains some graphs obtained by implementing the previous algorithm with different N values, plotting the number of processors (P) on the x-axis and the ratio between serial and parallel time, as a measure of the scalability of the program, on the y-axis.

If the function increases by the number of processors, it means that the overall performance of the implementation is increasing as well, while a decrease denotes a loss in performance as result of an increase in the number of processors which, of course, has to be avoided. So, the number of processors at which the function reaches its maximum has to be considered as the threshold after which you should not use more processors.

The different trends can be explained completely through the differences in N of the various implementations.

When N is small, after an initial increase in scalability you can observe a constant decrease due to the fact that the communication time keeps growing with the number of processors P , while the computational time enhancement is very small.

When N is large, instead, the computational time decreases at a faster rate than the increase in communication time and so the parallel time decreases and the scalability keeps growing; only when P is very large the increase in communication time exceeds the reduction in computational time and the parallel time increases, making the scalability decrease.

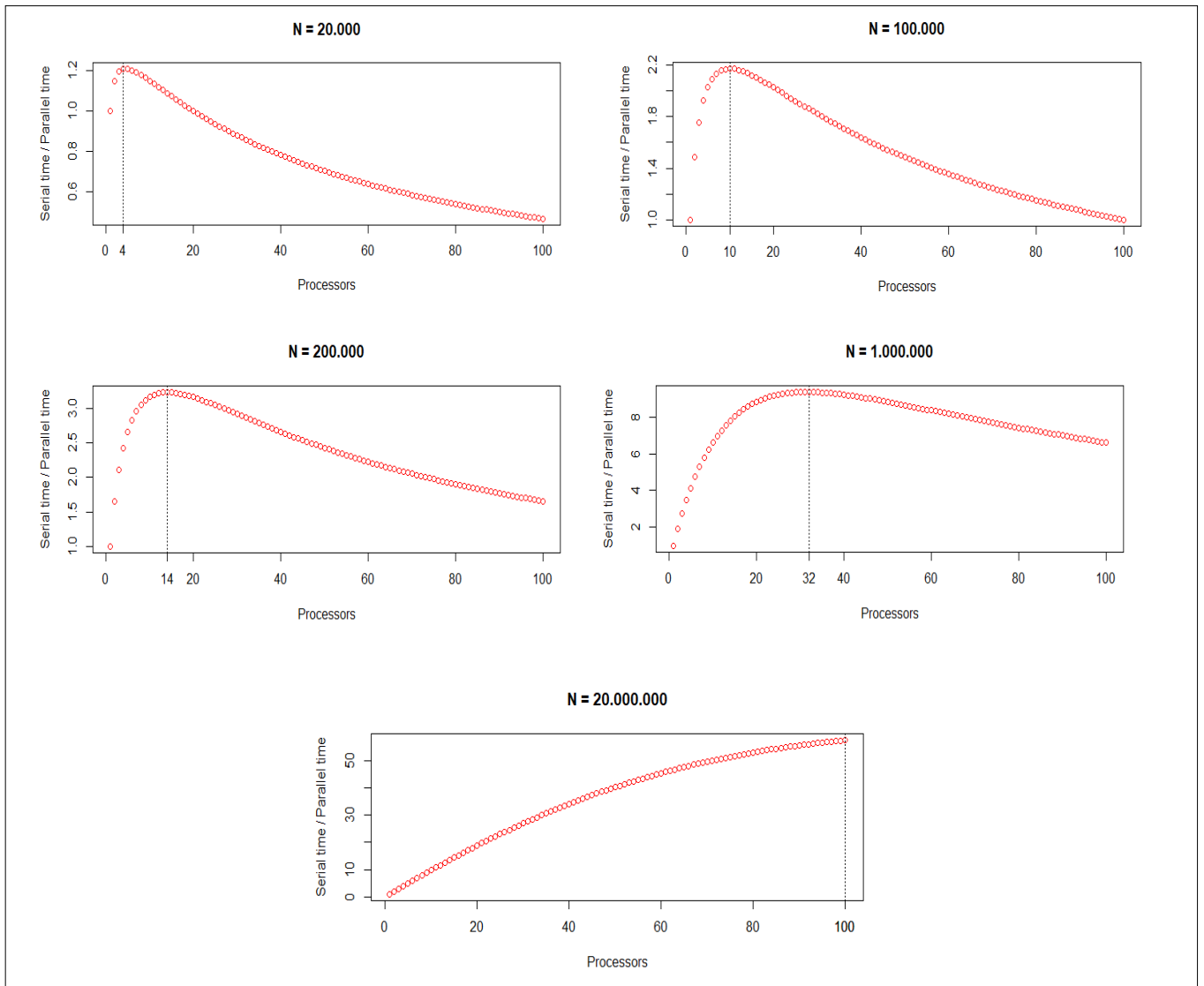


Figure 1.1: the scalability of the algorithm due to the choice of “ N ” parameter (the quantity of numbers to be added up).

• Enhanced model

In the previous paragraph a very simple implementation of the algorithm for the sum of N numbers as been used, in which the master processor reads the data, splits them in equal parts depending on the number of the slave processors and distributes these subsets of N to each of them.

In this way, there are $2 * (P - 1)$ communications among the processors: $P - 1$ when the data are distributed and other $P - 1$ communications when the slave processors give back to the master the results of their calculations. Moreover, at the end of all the communications, the master processor has to perform a final computation which includes all the partial results.

Thus, it could be interesting to develop an “enhanced” version of the algorithm to improve its efficiency, and so its performance.

An idea to reduce the communication times could be to have an initial split of the data in two halves, one of whom is then delivered from the master processor to one of the slaves; then, both the master and the slave processors distribute a half of their current amount of data to other two slaves, and this process continues until all the processors have obtained a part of the original data. The same schema is followed also after the slaves have completed their calculations and give their results back.

At this point you could realize that, given P processors, the number of communications (K) is given by:

$$K = 2 * \log_2 P$$

or, given K communications, the numbers of processors involved in the computation is:

$$P = 2^{K/2}$$

In this way, the $2 * (P - 1) * T_{comm}$ time for communication becomes: $2 * (\log_2 P) * T_{comm}$.

For what concerns the computational time, a possible way to reduce it is to make it possible, for the processors which gather the results of the lower-level slaves, at each step, to sum these results to their own ones before transmitting, in turn, the outcomes to the upper-level processors. In this way, the master processor will just perform a single addition instead of $P - 1$.

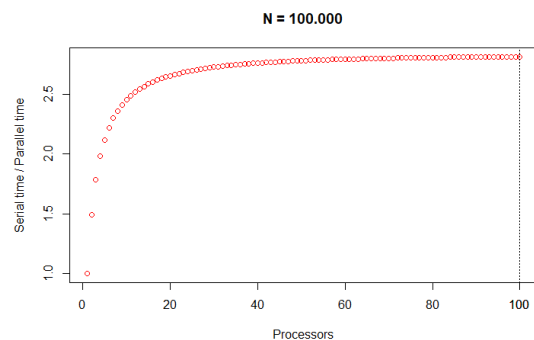
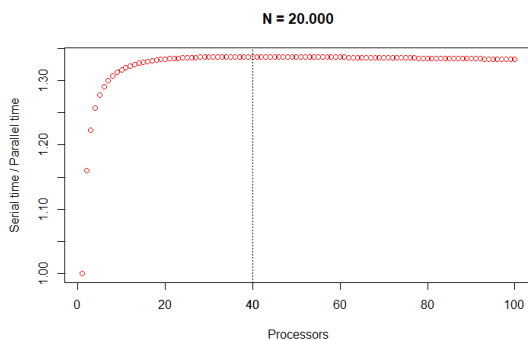
By doing this, the $(P - 1 + \frac{N}{P}) * T_{comp}$ time for computation becomes: $(\frac{N}{P} + \log_2 P + 1) * T_{comp}$.

At this point, the final algorithm for the parallel time could be expressed as:

$$Parallel\ time = T_{read} + \left(\frac{N}{P} + \log_2 P + 1\right) * T_{comp} + 2 * (\log_2 P) * T_{comm}$$

The results of the application of the new algorithm are shown in Figure 1.2 (below).

As you can see, the increase in the scalability of this program with respect to the previous one is huge for all the executions, with the maximum reached always at $P = 100$, and the curves never decreasing. The last plot is the most impressive, because the function appears to be roughly as a straight line with a quite high slope in all the considered range of P values.



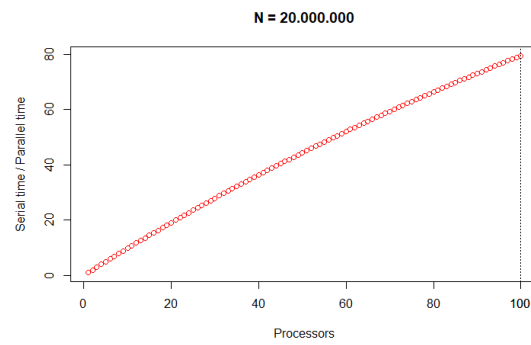
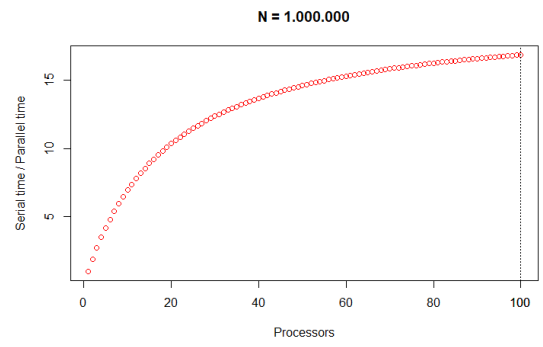
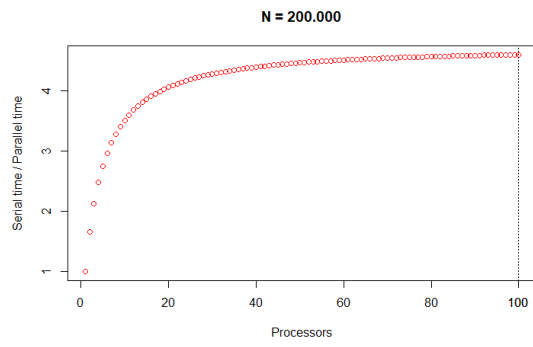


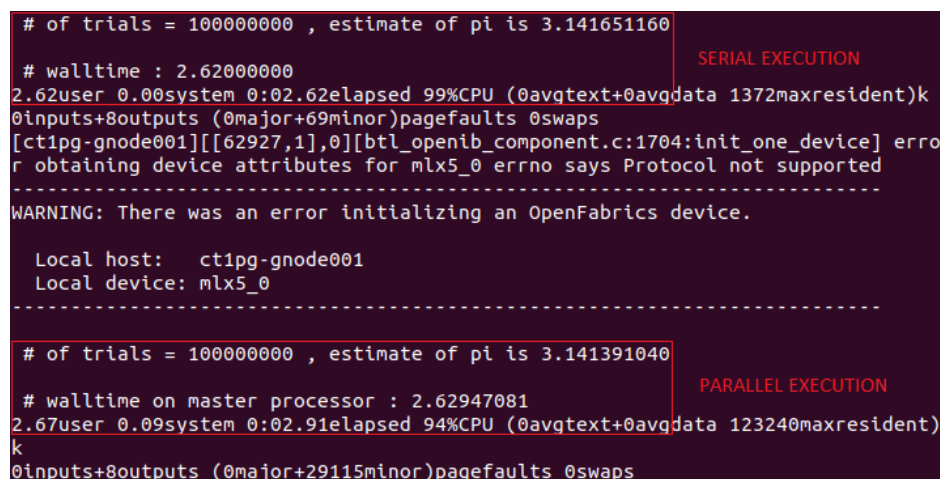
Figure 1.2: the scalability of the “enhanced” algorithm.

Section 2: Strong scalability, parallel overhead and weak scalability

For the simulations of this section, a toy MPI program has been provided.

It performs a Monte-Carlo integration to compute π (pi, the mathematical constant). The idea is to have a circle inscribed inside a square of unit length: in this case, the ratio between the area of the circle ($\frac{\pi}{4}$) and the area of the square (1) is $\frac{\pi}{4}$. Therefore, if you randomly choose N points inside the square, on average, only $M = N * \frac{\pi}{4}$ points will belong to the circle. From the last relation you can obtain an estimate for π .

First of all, you could compile and execute both a serial and a parallel version of the program on just one processor, to get an idea of the overhead associated with MPI.



```
# of trials = 100000000 , estimate of pi is 3.141651160
# walltime : 2.62000000
2.62user 0.00system 0:02.62elapsed 99%CPU (0avgtext+0avgdata 1372maxresident)k
0inputs+8outputs (0major+69minor)pagefaults 0swaps
[ct1pg-gnode001][[62927,1],0][btl_openib_component.c:1704:init_one_device] error
r obtaining device attributes for mlx5_0 errno says Protocol not supported
-----
WARNING: There was an error initializing an OpenFabrics device.

Local host:  ct1pg-gnode001
Local device: mlx5_0
-----

# of trials = 100000000 , estimate of pi is 3.141391040
# walltime on master processor : 2.62947081
2.67user 0.09system 0:02.91elapsed 94%CPU (0avgtext+0avgdata 123240maxresident)k
0inputs+8outputs (0major+29115minor)pagefaults 0swaps
```

The image shows a terminal window with two sections of output. The top section, labeled 'SERIAL EXECUTION', shows the results of a serial execution of a Monte-Carlo pi estimation program. It reports 100,000,000 trials, an estimate of pi as 3.141651160, a walltime of 2.62 seconds, and 99% CPU usage. The bottom section, labeled 'PARALLEL EXECUTION', shows the results of a parallel execution on the same processor. It reports the same number of trials and a slightly different estimate of pi (3.141391040). The walltime on the master processor is 2.62947081 seconds, and the CPU usage is 94%. Both sections show a warning about an error initializing an OpenFabrics device.

Figure 2.1: comparison between the serial and the parallel execution of the program on one processor, with $N = 100$ millions.

You can notice that, for the serial version, the walltime (that is the execution time of the program according to the machine's internal clock) coincides exactly with the elapsed-time (that is the amount of time from the beginning to the end of the execution, computed with the `/usr/bin/time` function). Moreover the percentage rate of CPU usage (%CPU) is very high, close to the maximal value of 100%.

Thus, in this implementation of the program there is no waste of time in operations which are not related to actual computations, and it could be used as a benchmark to determine the performance loss in the parallel version. As you can see, the output of execution time for the latter is quite different: while the walltime is slightly higher than the serial one, the elapsed-time is now much larger and the %CPU has decreased of about 5%. In addition, now, the system-time has a value different from zero. You can interpret these results as a proof of the delays in the execution due to the presence of MPI instructions (which are not even used in the case of one processor, but are compiled anyway).

Of course, if you should decide which is the best implementation for one single processor, the choice should be the serial version: it avoids to include in the program some useless instructions which, besides, slow down the final execution.

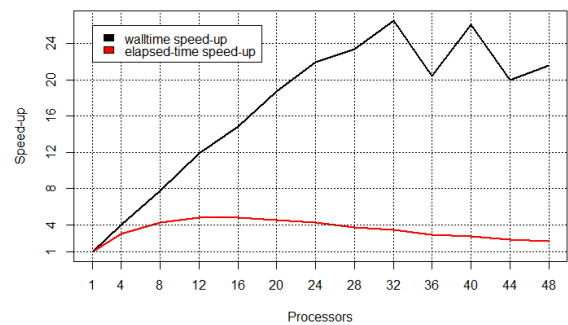
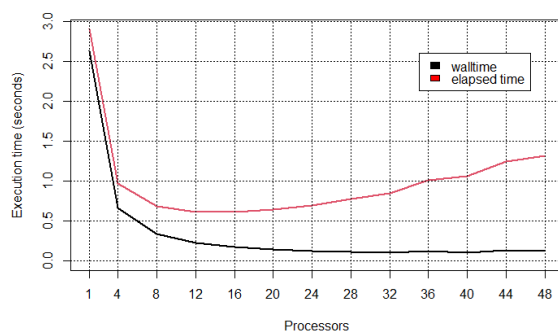
Now, focusing on the MPI program, you could interrogate on its performance on more processors. This is a good point because, as said before, the program does not perform well on a single processor: it has been designed exactly to exploit a greater number of processors.

You could carry on this investigation in two directions: it is possible to evaluate both the strong scalability and the weak scalability of the program.

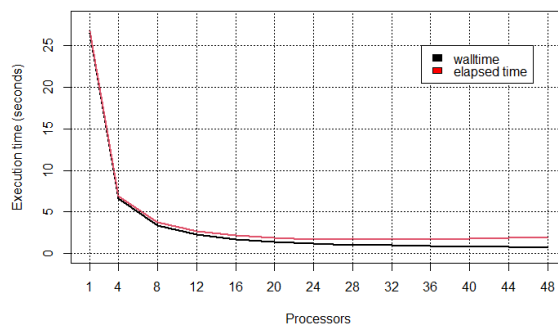
- Strong scalability

In the strong scalability test, you keep the data size constant and verify if increasing the number of processors brings to a decrease in execution time.

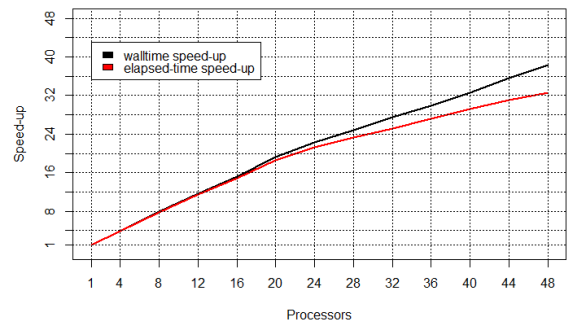
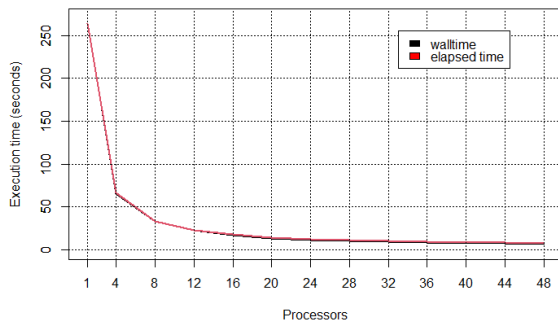
Here, four different scalability tests are reported, each one with a different data size (N).



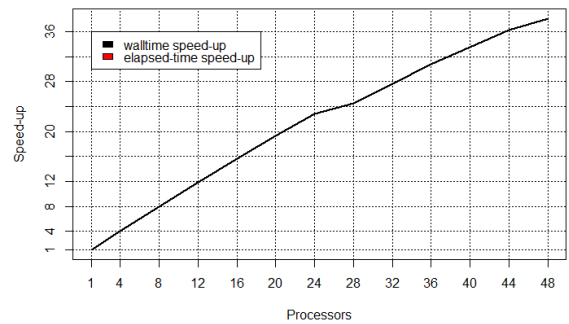
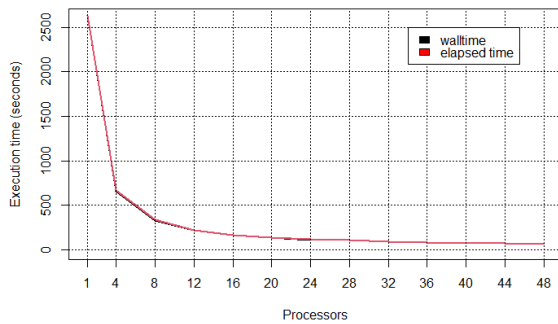
(a)



(b)



(c)



(d)

Figure 2.2: plots of execution walltime and elapsed-time (left) and speed-up (right) of the program against the number of processors, for different N values: (a) $N = 100$ millions, (b) $N = 1$ billion, (c) $N = 10$ billions, (d) $N = 100$ billions.

By means of the previous plots, you can make some interesting observations about the performance differences as consequence of the variations in the number of used processors.

Looking at the left-side of Figure 2.2, you could compare the walltime and elapsed-time trends for different data sizes. First of all, based on the implicit assumption that the elapsed-time includes also some non-computational time (thus some amount of time used for other operations, like communication between the processors), the fact that elapsed-time is (at least slightly) greater than walltime is correct.

So the dimension of the difference between the two times, when the data size is relatively “small” (such as 100 millions or 1 billion), shows an increasing divergence when the number of processors is increasing as well. For higher values of N , the difference looks to be negligible for the only reason that the scale is different: when data size is small, the computational time is small too, and so the time dedicated to other tasks assumes, in proportion, a large value; conversely, by using an elevated number of data, the time for computational operations increases and, thus, the relative weight of non-computational operations is reduced, making the absolute difference between elapsed-time and walltime approximately constant across the plots.

For what concerns the right-side of Figure 2.2, the concept of speed-up has been used to provide a measure of the strong scalability across the processors.

The speed-up is given by:

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

where $T(1)$ is the execution time for one processor and $T(P)$ is the execution time for P processors.

It is trivial to observe that, if $S(P) = P$, it means that $T(P)$ is exactly $T(1)$ divided for the number of processors P , and so you can define this as a case of “perfect speed-up” (or “linear scaling”).

In real cases, however, this is a very rare condition because of the overhead, that occurs and increases when you try to use more processors in your execution, as well as other problems like latency (that is the time required to transfer the information from a part of the system to another one).

For example, in this application, you see that the curves describing the speed-up are such that every time you increase the number of processors (on x-axis) the effective computational power does not increase in a proportional way, and sometimes it reaches a maximum and then starts decreasing.

- Model for the parallel overhead

As shown in the previous paragraph, in this application there is not a very good scalability when you increase the number of processors with a problem of fixed data size.

It might be useful to try to develop a model to indirectly estimate the parallel overhead occurring during the execution, as it is certainly one of the main causes of bad scalability.

To do this, a possible idea is to compute the difference between the elapsed-time (which, as said before, probably contains also some time spent in operations different from the computations) and the walltime computed by the system.

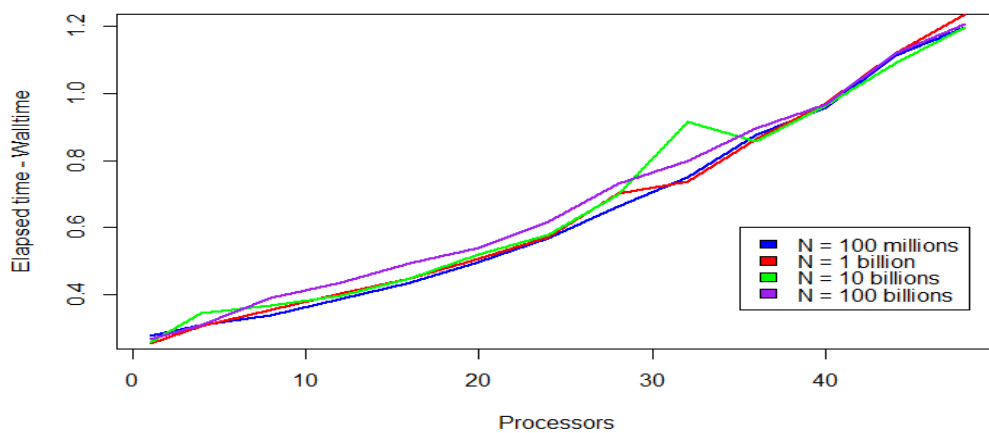


Figure 2.3: plot with the estimate of the parallel overhead against the number of processors, for the four implementations with different values for N .

As you can see by looking at the graph above, the parallel overhead increases when P grows, in all the considered implementation and with similar trends and values.

This is exactly what you would expect, given the previous paragraph's conclusions about the scalability: by increasing the number of processors you cannot obtain a proportional enhancement of performance because, even if the computational time probably keeps decreasing, there is an increase in the time devoted to other tasks, like the communication among the processors, which depend essentially on the number of processors and not on the data size.

- Weak scalability

In the weak scalability test, you adapt the data size to the number of used processors and verify if the execution time remains constant.

Here, you can see the results of weak scalability test on the same MPI program, with four different data sizes as before (in this case, only the walltime has been considered).

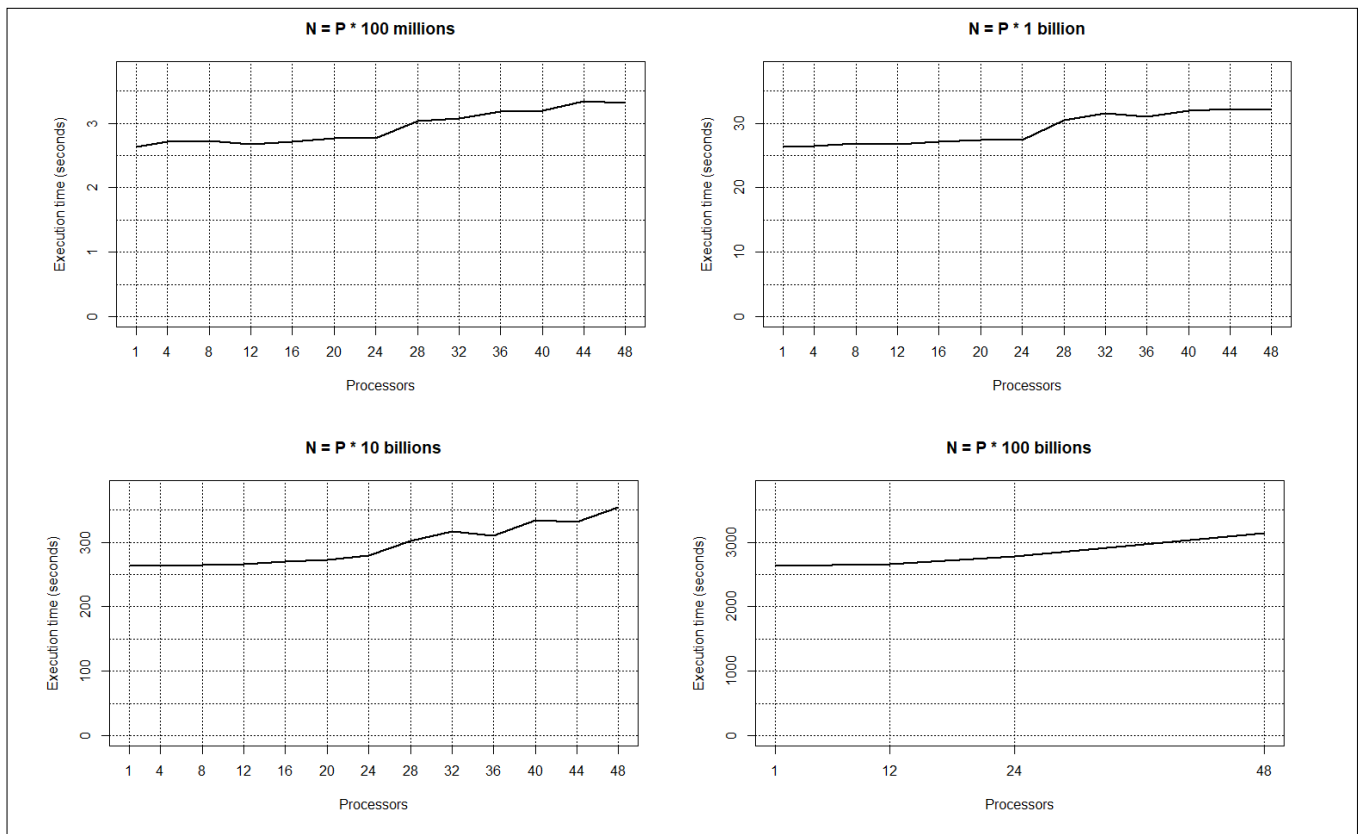


Figure 2.4: plots of execution time of the program against the number of processors (P), with N values growing proportionally to P (in the graphs, N values are referred to one-processor executions).

In the case of weak scalability test, what you expect to see is a steady trend, because when you increase the number of processors, the data size grows proportionally (so, for example, if you use $N = 100 \text{ millions}$ on one processors, you will use $N = 200 \text{ millions}$ on two processors, and so on).

Actually, in this application, you can notice such a tendency approximately, but in general the execution time increases by the number of processors, denoting that there is not a perfect (weak) scalability, probably (again) due to problems like overhead and latency.

In order to give a syntetic measure of weak scalability and to make it possible to compare the executions of the program with different data sizes, also in this case you can compute the ratio $T(1)/T(P)$, that you can define as a sort of “weak efficiency”.

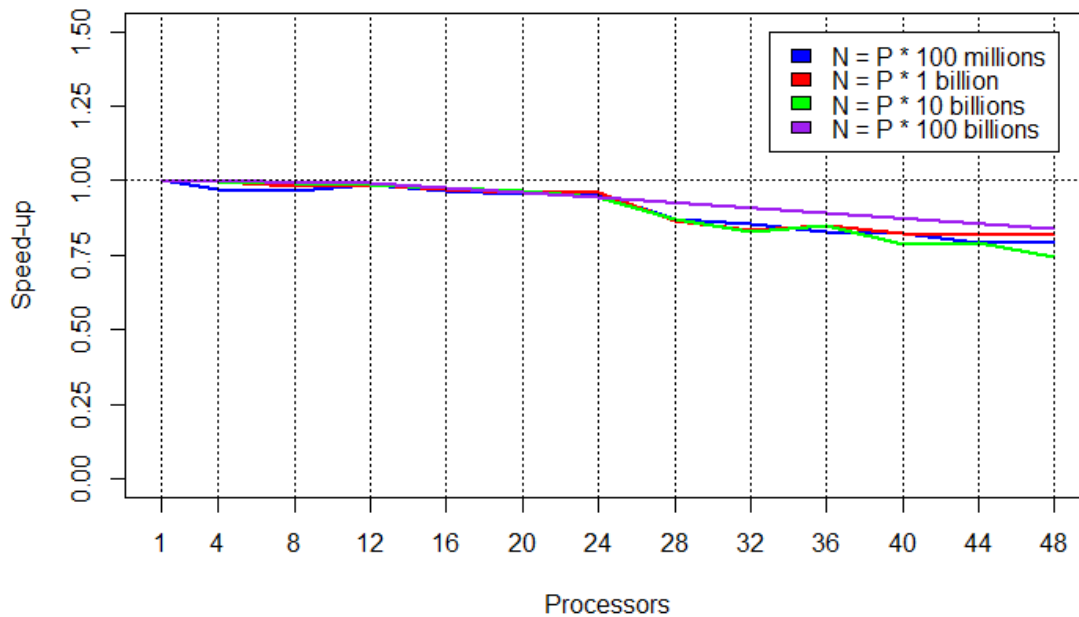


Figure 2.5: plot of “weak efficiency” against the number of processors.

By looking at the graph above, it looks like the different executions have roughly the same trend, with a $T(1)/T(P)$ ratio which is slightly smaller than the benchmark value of 1 (which denotes linear scalability) for small numbers of processors; however, when the number of processors is greater than 24, the ratio starts decreasing in all the cases, until it reaches values between 0.75 and 0.85 for 48 processors, and this constitutes a further evidence to support the idea that parallel programs’ problems, like overhead and latency, play an important role in the overall performance.