Report

Assignment 1 - HPC

Davide Roznowicz

2/11/2020

Contents

Section 1: theoretical model

Analysis and visual comparisons between naive and enhanced parallel algorithms

In order not to repeat similar plots a lot of times, I decided to insert the enhanced version together with the naive one (in the same plot) from the very beginning. However, the explanation concerning how the enhanced algorithm was thought and built will be provided only after these visual representations.

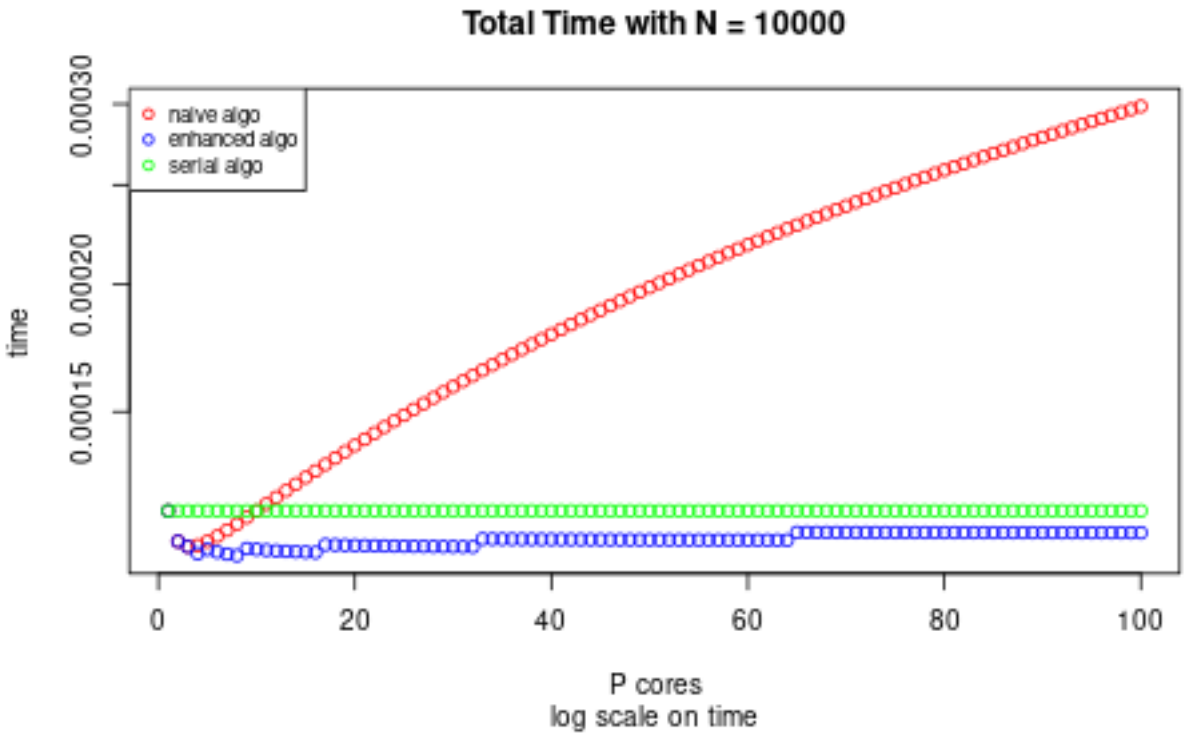
The first series of plots shows the total time the algorithms (serial, naive and enhanced) need to perform the assigned task, varying the number of cores P from 0 to 100 on the x-axis. N is fixed: the first value given to to it is 10^4, then 10^5, 10^6, 10^7. The time is calculated in the following way:

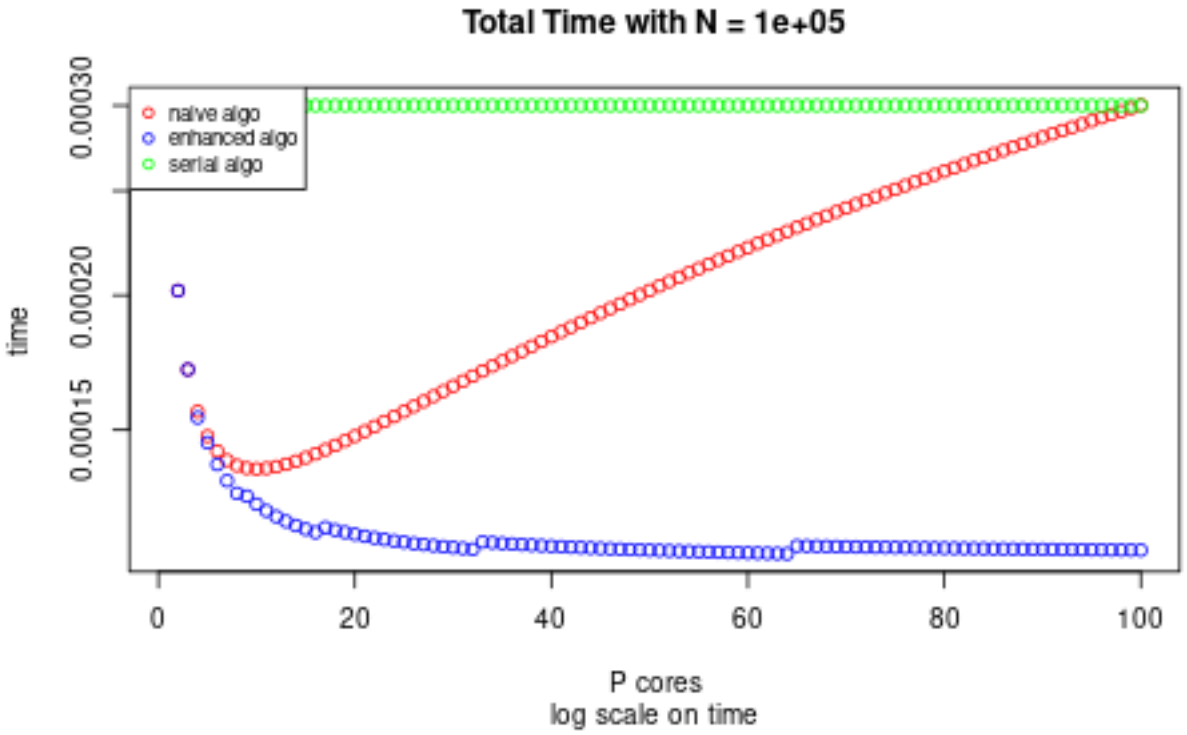
$$T_{serial} = T_{read} + N * T_{comp}$$

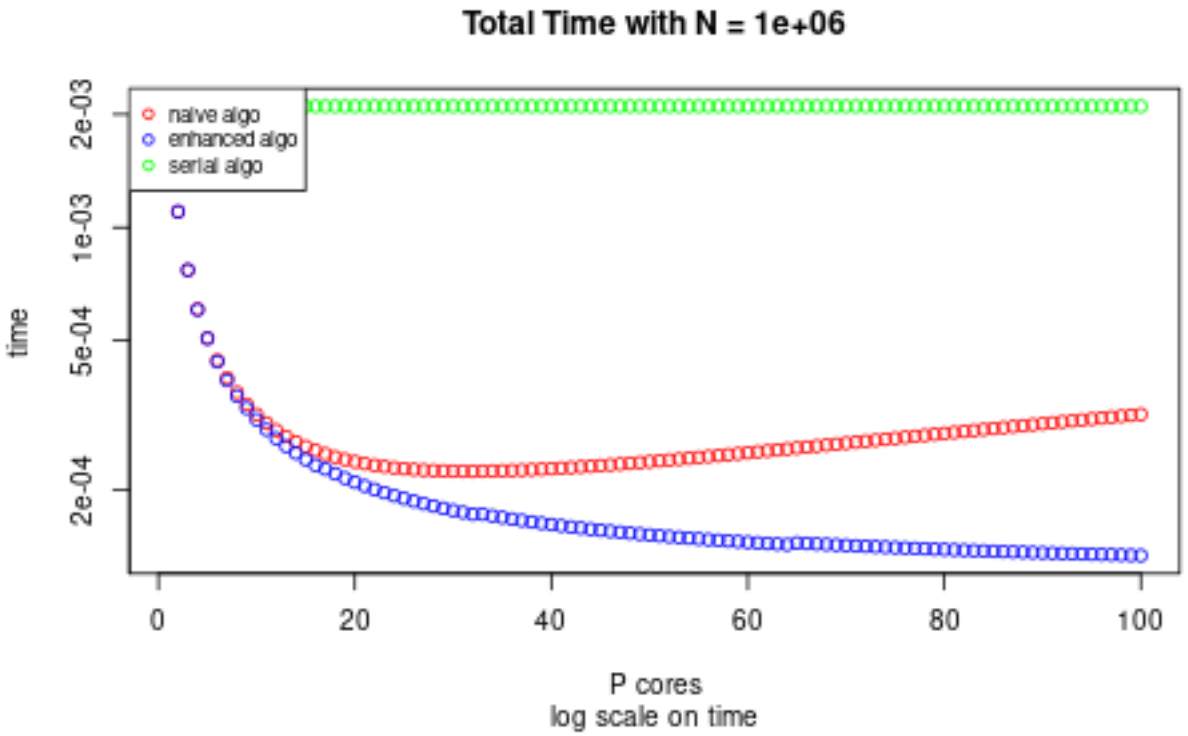
$$T_{naive}(P) = T_{comp} \times (P - 1 + \frac{N}{P}) + T_{read} + 2(P-1) \times T_{comm}$$

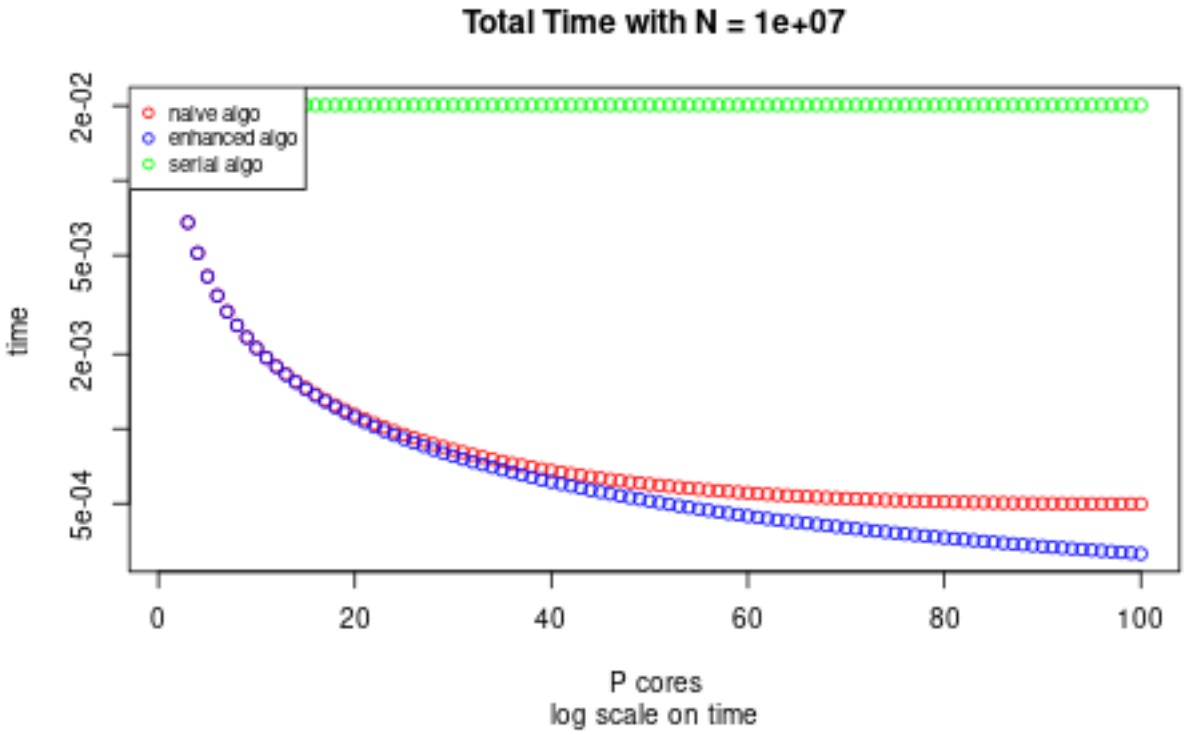
$$T_{enhanced}(P) = 2 \times T_{comm} \times \lceil log_2(P) \rceil) + T_{read} + (\frac{N}{P} + \lceil log2(P) \rceil) \times T_{comp}$$

Notice that, despite the fact that the exercise does not show T_{read} in T_{serial} equation, I believe it should be taken into consideration anyway: that's way it has been inserted into the equation.

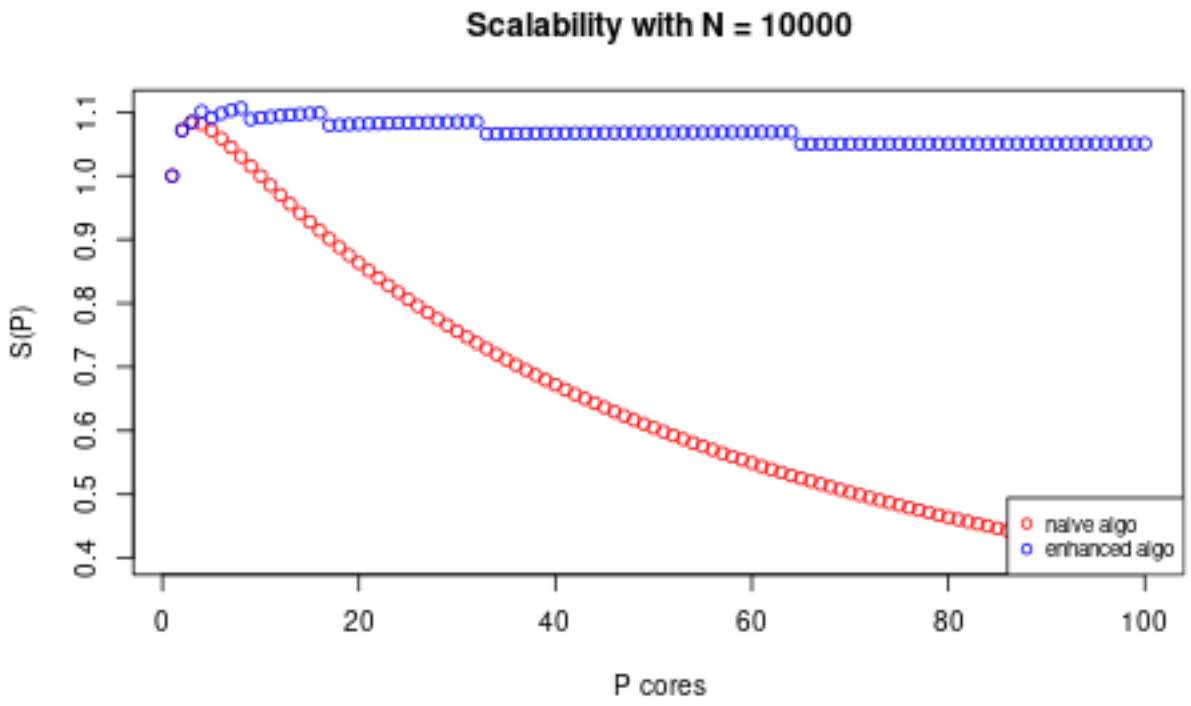

Total Time with N = 10000
naive algo
enhanced algo
serial algo
time
0.00015
0.00020
0.00030
0
20
40
60
80
100
P cores
log scale on time

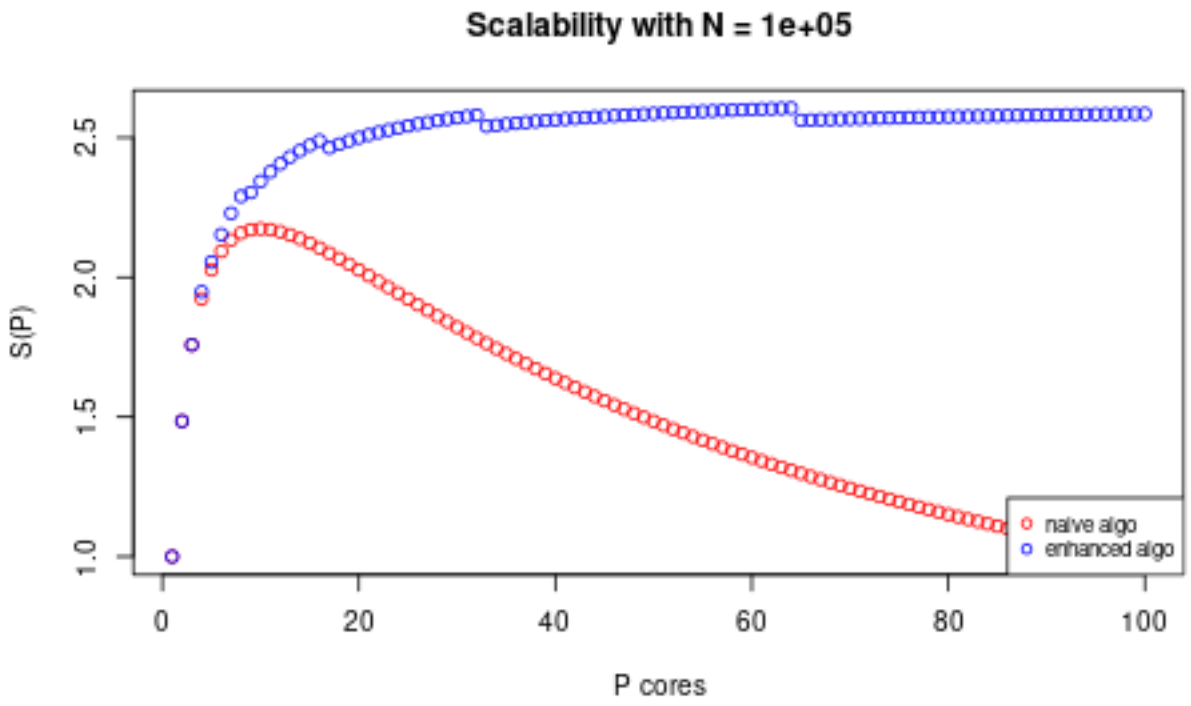

Total Time with N = 1e+05
naive algo
enhanced algo
serial algo
time
0.00015
0.00020
0.00030
0
20
40
60
80
100
P cores
log scale on time

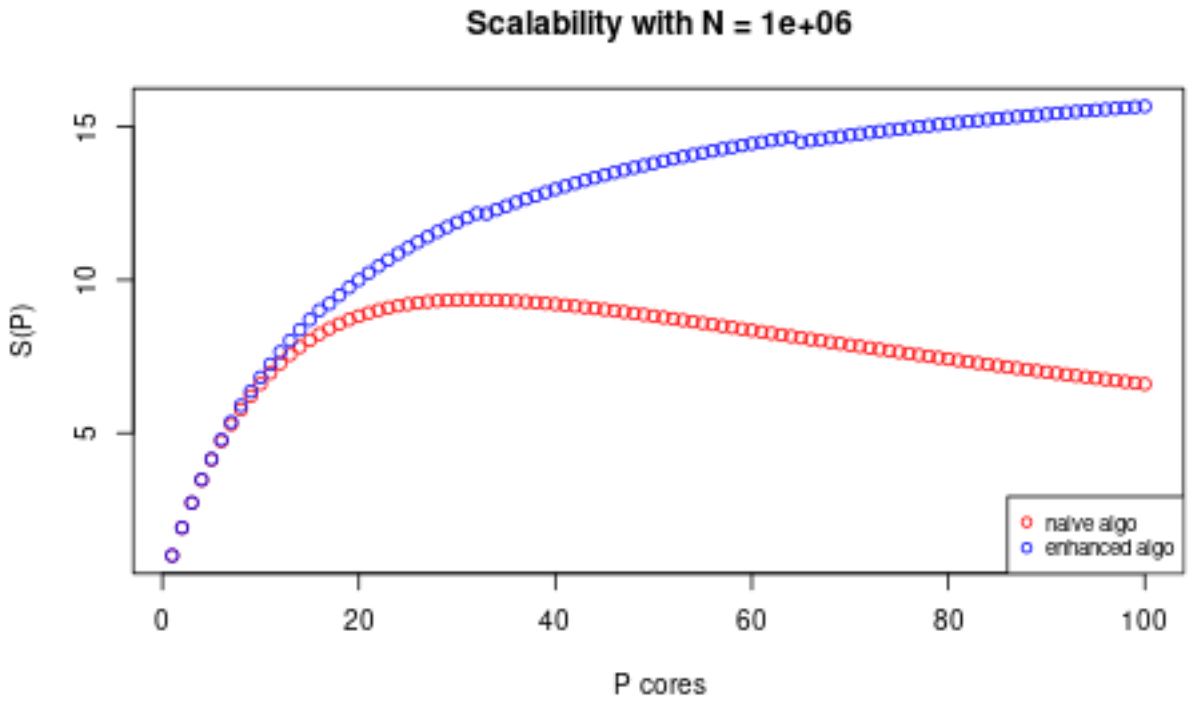


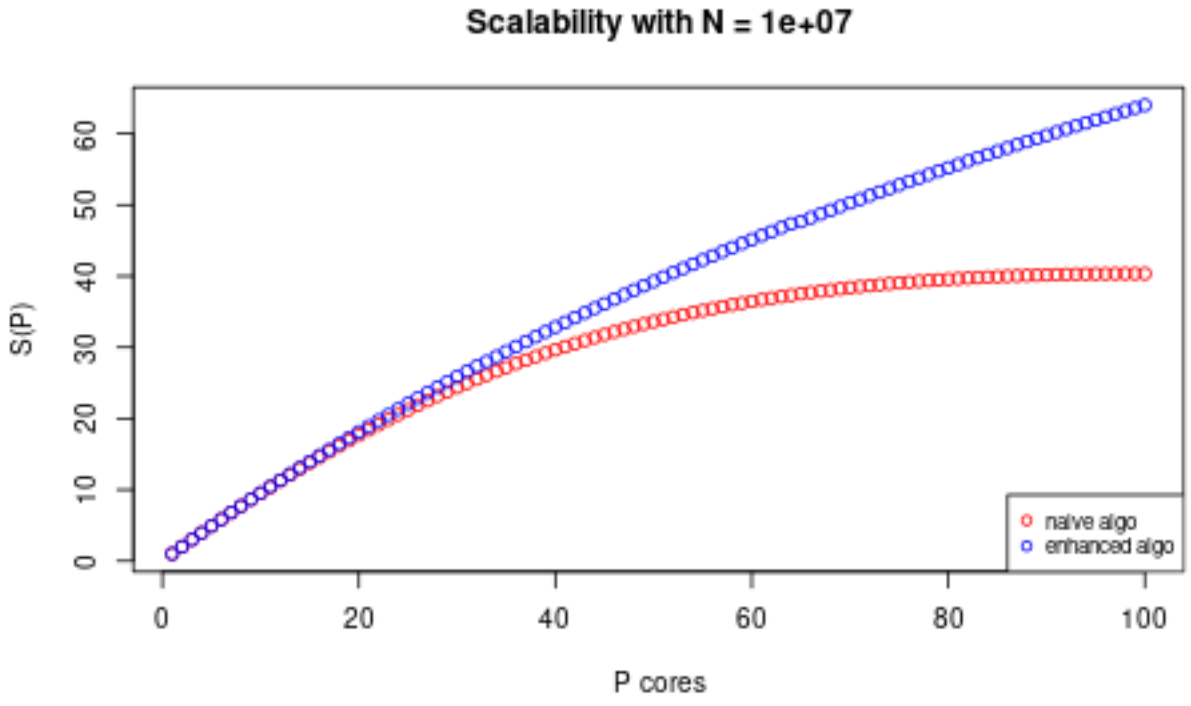


The second series of plots shows the scalability of the algorithms (serial, naive and enhanced), varying the number of cores P on the x-axis. N is fixed again to the previously mentioned values.


Scalability with N = 10000
S(P)
P cores
naive algo
enhanced algo
0.4
0.5
0.6
0.7
0.8
0.9
1.0
1.1
0
20
40
60
80
100


Scalability with N = 1e+05
S(P)
P cores
naive algo
enhanced algo
1.0
1.5
2.0
2.5
0
20
40
60
80
100


Scalability with N = 1e+06
S(P)
5
10
15
0
20
40
60
80
100
P cores
naive algo
enhanced algo


Scalability with N = 1e+07
S(P)
0
10
20
30
40
50
60
0
20
40
60
80
100
P cores
naive algo
enhanced algo

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

We easily notice that the higher N is, the better the parallel algorithms scale. Except for N and P very small, case in which T_{comm} represents a large and unnecessary overhead, the enhanced algorithm tends to be far more scalable than the naive version.

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

Notice: the considered range for P is [1,100], as stated in the exercise requests.

```
## [1] "if N = 10000 , the best P for naive algo is : 3"
## [1] "if N = 10000 , the best P for enhanced algo is : 8"
## [1] "if N = 1e+05 , the best P for naive algo is : 10"
## [1] "if N = 1e+05 , the best P for enhanced algo is : 64"
## [1] "if N = 1e+06 , the best P for naive algo is : 32"
## [1] "if N = 1e+06 , the best P for enhanced algo is : 100"
## [1] "if N = 1e+07 , the best P for naive algo is : 100"
## [1] "if N = 1e+07 , the best P for enhanced algo is : 100"
```

Therefore it is clear that if N grows, so does the best P both for naive and enhanced algorithms. As pointed before, the communication time is a huge overhead for the naive algorithm: in fact, unless N is very large (and also $\frac{N}{P}$ is large), its scalability curve tends to flatten almost immediately and then go down. This happens because the increasing number of cores cannot sustain easily the rapid growth of T_{comm}, that is by the way three orders of magnitude more influential than T_{comp}. Instead, if N is large enough, the scalabily curve of the enhanced algorithm is much more "resilient" than the one belonging to the naive version, thus the best P tends to be much higher: this is mainly associated with the improvements concerning the designed communication system, such that it is not linearly dependent on P, but instead logarithmically dependent.

Can you try to modify the algorithm sketched above to increase its scalability ? (hints: try to think of a better communication algorithm)

The enhanced version of the naive algorithm is designed thinking that there is no particular sense in compelling the master core to send all the "messagges" sequentially. Instead it seems much more convenient to let other cores "help" the master in spreading the assigned task. Thus, the optimal choice appears to be the following:

1. After reading the task, the *master* should send half of the numbers to $core_1$ and keep the other half for itself;
2. the *master* and $core_1$ send half of their remaining numbers to $core_2$ and $core_3$ respectively;
3. the *master*, $core_1$, $core_2$, $core_3$ send half of their remaining numbers to $core_4$, $core_5$, $core_6$, $core_7$ respectively;
4. ... so on up to reaching $core_{P-1}$;
5. the sums are crunched by the P cores (each core has the same amount of numbers);
6. the computed P-1 sums are sent back to master following the route provided in the first four phases but in reverse order; at each step the partial sums are computed in the slave cores and then sent to another core as described before; this is repeated until only the *master* and $core_1$ have the partial sums (*master* has the sum of half of the amount of the original N numbers, while $core_1$ has the other half);
7. $core_1$ gives to the *master* its partial sum and the *master* computes the final sum;

This way many of the "steps" for fully spreading an equal amount of numbers to each core are done in parallel and not just sequentially like in the naive algorithm.

Therefore:

- $\sum_{k=0}^{steps-1} 2^k \geq P - 1$
- $\frac{1-2^{steps}}{1-2} \geq P - 1$
- $2^{steps} - 1 \geq P - 1$
- $2^{steps} \geq P$

- $steps \geq \frac{\log(P)}{\log(2)}$
- $steps \geq \log_2 P$

However, due to the discrete context, $\lceil \log_2 P \rceil$ should be considered instead of $\log_2 P$. Thus, the minimum is $steps = \lceil \log_2 P \rceil$.

- Read N and distribute N to P-1 slaves ===> $T_{read} + \lceil \log_2 P \rceil \times T_{comm}$
- $\frac{N}{P}$ sum over each processors (including master) ===> $T_{comp} \times \frac{N}{P}$
- Slaves send partial sum ===> $\lceil \log_2 P \rceil \times T_{comm}$
- Slaves perform partial sums while sending back the numbers to the *master* ===> $(\lceil \log_2 P \rceil - 1) \times T_{comp}$
- Master performs one final sum ===> T_{comp}

The final model:

$$T_{enhanced}(P) = 2 \times T_{comm} \times \lceil log_2(P) \rceil) + T_{read} + (\frac{N}{P} + \lceil log2(P) \rceil) \times T_{comp}$$

This enhanced algorithm not only improves the communication time because of a faster way to send the equal amount of numbers to each core (and then in the reverse process of getting them back), but also manages to crunch the partial sums at the slaves level, in order to process everything in parallel as much as possible.

performance-model.csv

The considered range for P is [1,100], as stated in the exercise requests. In the .csv the following results have been written down:

```
## [1] "if N = 20000 , the best P for naive algo is : 4"
## [1] "if N = 20000 , the best P for enhanced algo is : 16"
## [1] "if N = 1e+05 , the best P for naive algo is : 10"
## [1] "if N = 1e+05 , the best P for enhanced algo is : 64"
## [1] "if N = 2e+05 , the best P for naive algo is : 14"
## [1] "if N = 2e+05 , the best P for enhanced algo is : 100"
## [1] "if N = 1e+06 , the best P for naive algo is : 32"
## [1] "if N = 1e+06 , the best P for enhanced algo is : 100"
## [1] "if N = 2e+07 , the best P for naive algo is : 100"
## [1] "if N = 2e+07 , the best P for enhanced algo is : 100"
```

Section 2 : play with MPI program

2.1: compute strong scalability of a mpi_pi.c program

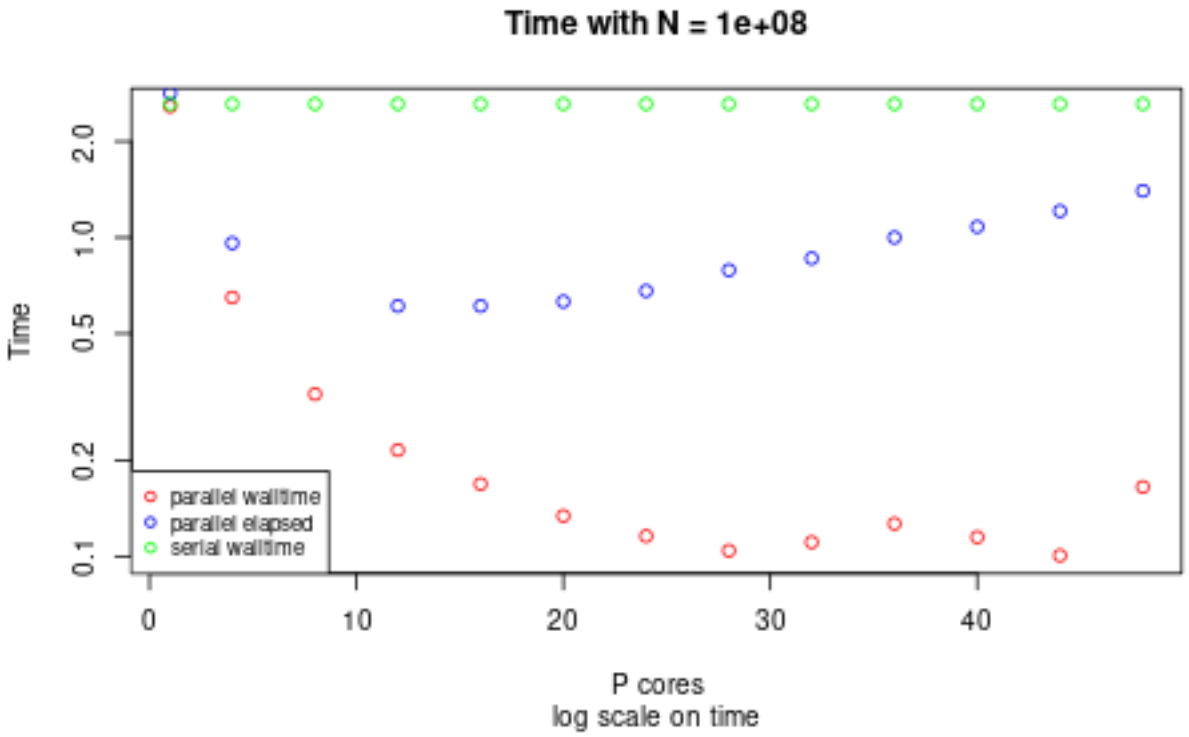
Single core serial versus single core parallel

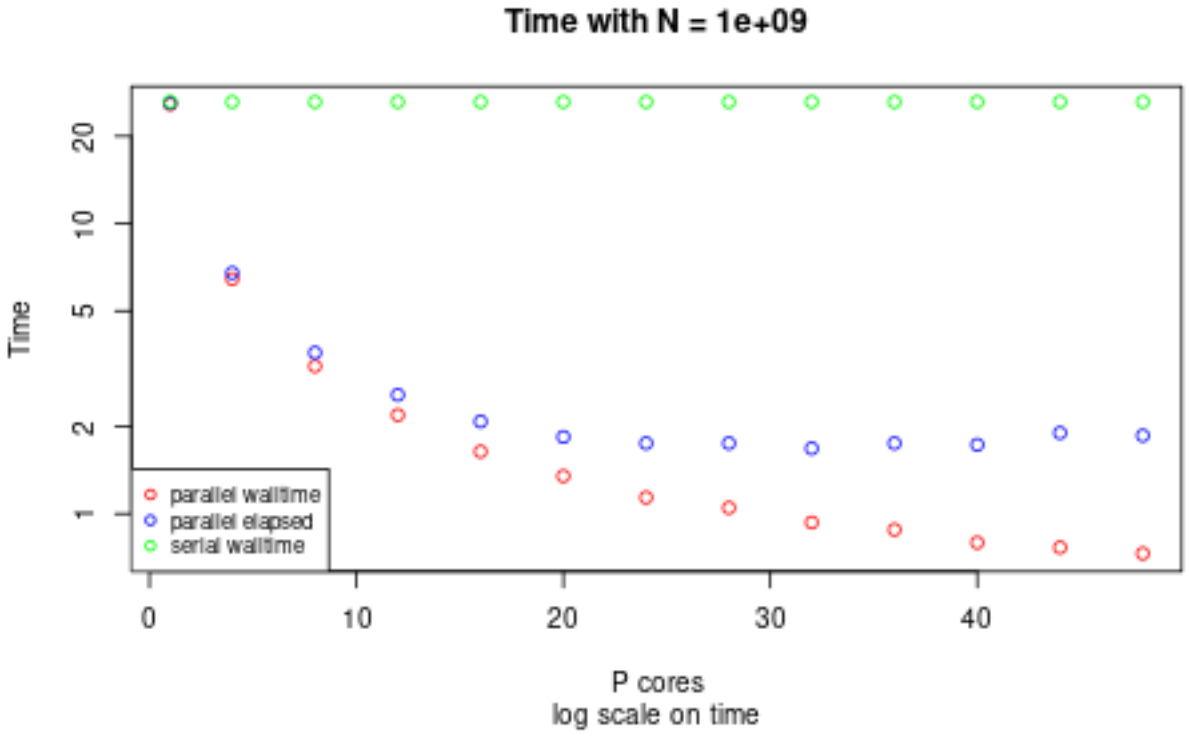
The serial calculation of pi with 10^8 iterations takes ~2.620 seconds, both for internally measured walltime and externally computed elapsed time (by /usr/bin/time). Instead the parallel calculation of pi with 10^8 iterations takes ~2.575 as walltime that is quite close to the serial version. However the total elapsed time is higher ~2.830 seconds. This is because, despite the fact that there are no communication times among the cores (there is a single core), we should take into account the overhead associated with starting the parallel version of the program (system calls, starting machinaries...) : system time is not negligible (~0.13 seconds) as it was before in the serial context. Moreover, the elapsed time is larger than the walltime also because it includes printing the outcome of the simulation. Of course, as a consequence, $\%CPU = \frac{user}{elapsed}$ is lower in the parallel version.

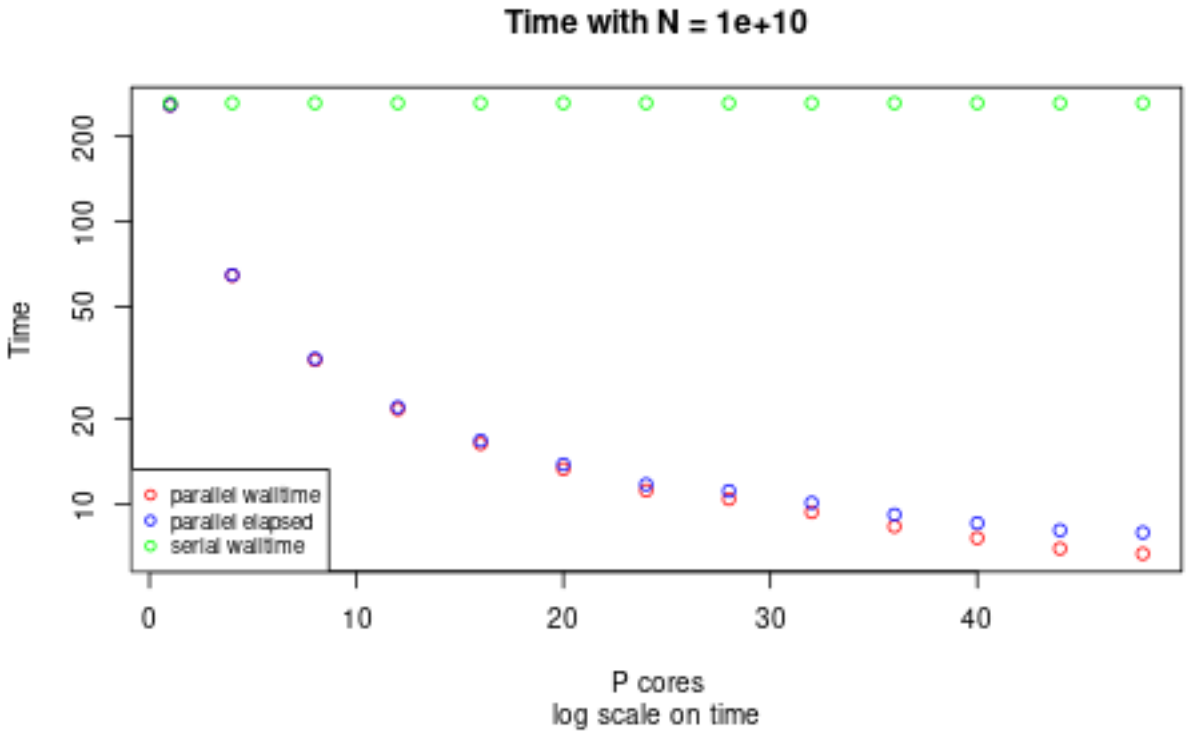
Strong scalability

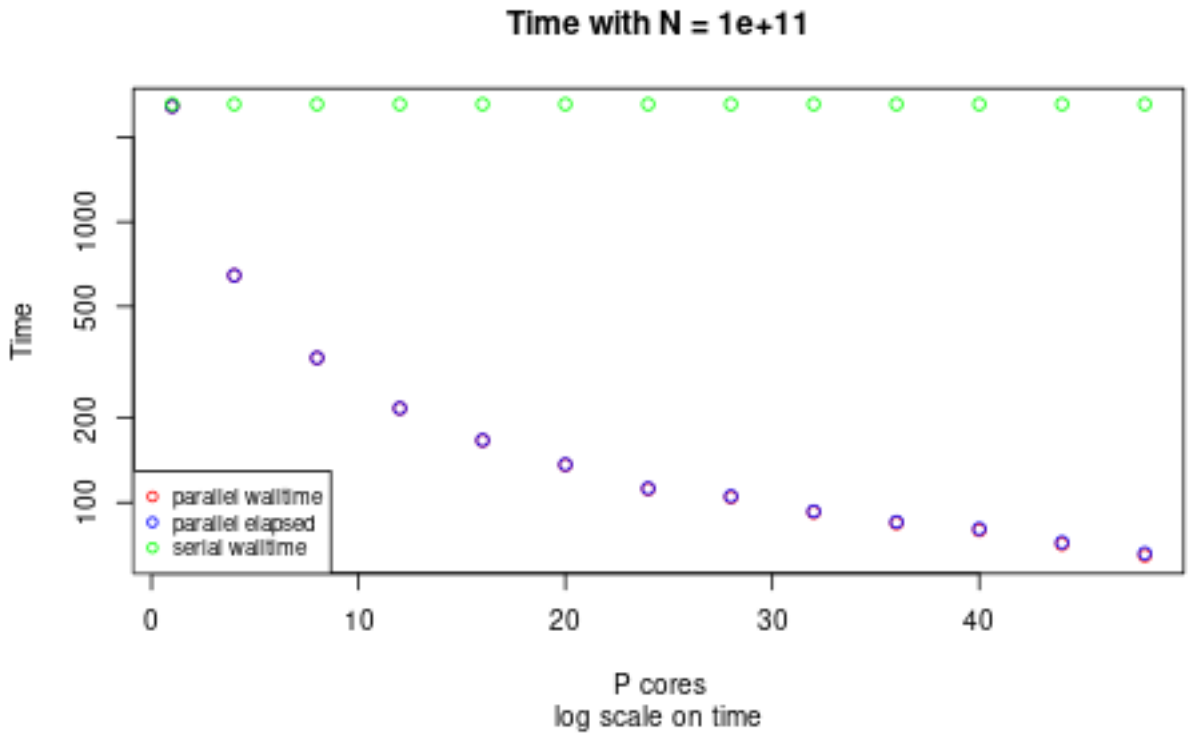
The walltime related to the master core will be used to fill in the .csv and perform the requested analysis: in fact, as we are interested in the highest time among the cores, the master seems to be a good choice.

Let's look at some plots comparing the walltime (average among the three runs) and elapsed time against the number of cores. The serial walltime is also added for further comparison.



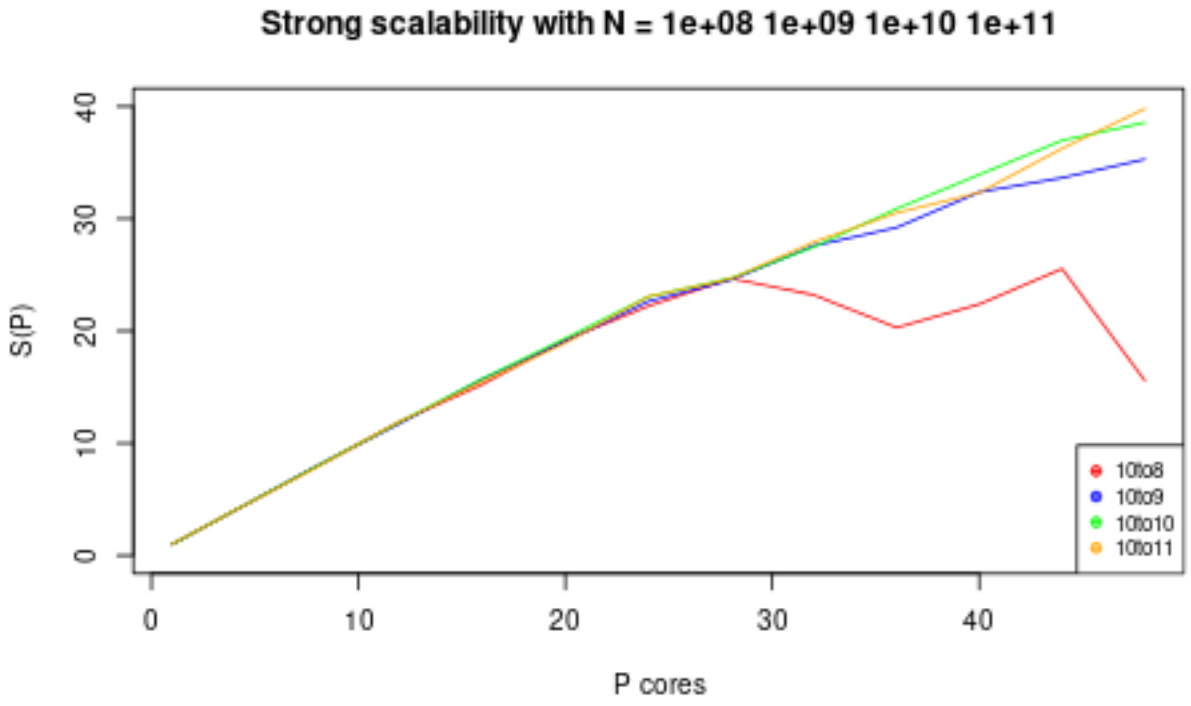

Time with N = 1e+09
Time
1
2
5
10
20
0
10
20
30
40
P cores
log scale on time
parallel walltime
parallel elapsed
serial walltime


Time with N = 1e+10
Time
10
20
50
100
200
0
10
20
30
40
P cores
log scale on time
parallel walltime
parallel elapsed
serial walltime



As previously stated, the walltime dotted line for the parallel version is always below the elapsed time, given the fact that it doesn't take into consideration neither printing time nor additional system overhead. The latter is particularly consistent when dealing with small sizes of N; when N grows the relative weight of this factor for the simulation time gets less visible. Notice that in the serial version, even though N changes, the system time remains ~0.00 while walltime and elapsed time are almost the same.

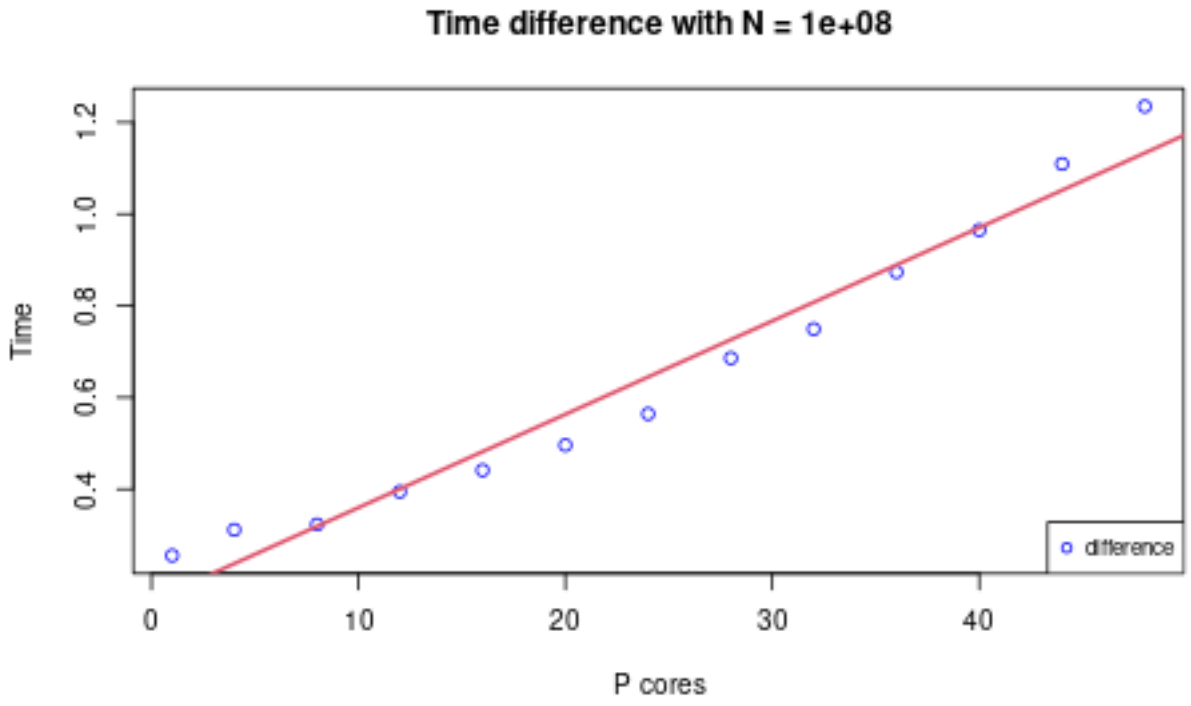
Now let's look at scalability plots: 4 scalability curves for each size of N are drawn in the same chart. As a matter of comparison, the walltime is chosen to compute the scalability funcion: otherwise there would be some components such as printing or system that might skew the outcome as they also include some factors that are constant (but difficult to see). Choosing the walltime allows us to focus on scalability by spotting how communication times among cores (included in walltime) affect the curves. As one-core-benchmark for computing scalability we opt for the parallel version since the time is very close to the serial one. Lines connecting the dots are drawn because points would be too confusing for visualization.



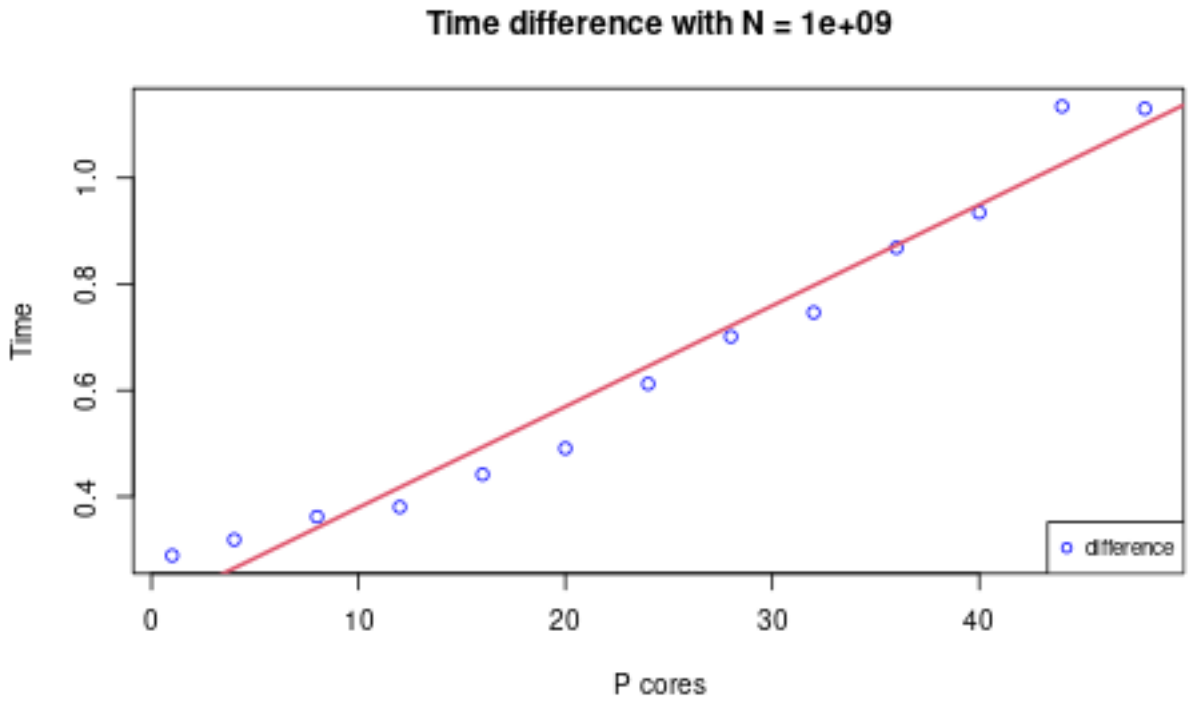
We easily notice that the higher N is, the better the parallel algorithm scales and approximates a straight line. Up to 24 cores there's almost perfect speed-up for all the involved sizes of N; later some break-points appear according to the size of N. When N is small, communication times heavily affect the outcome, thus the curves flatten. Some swings are evident in the final parts of the curves: these are due to some noise particularly evident when the time to perform the simulation is in the order of a couple of seconds for one core (e.g. $N = 10^8$).

2.2: identify a model for the parallel overhead

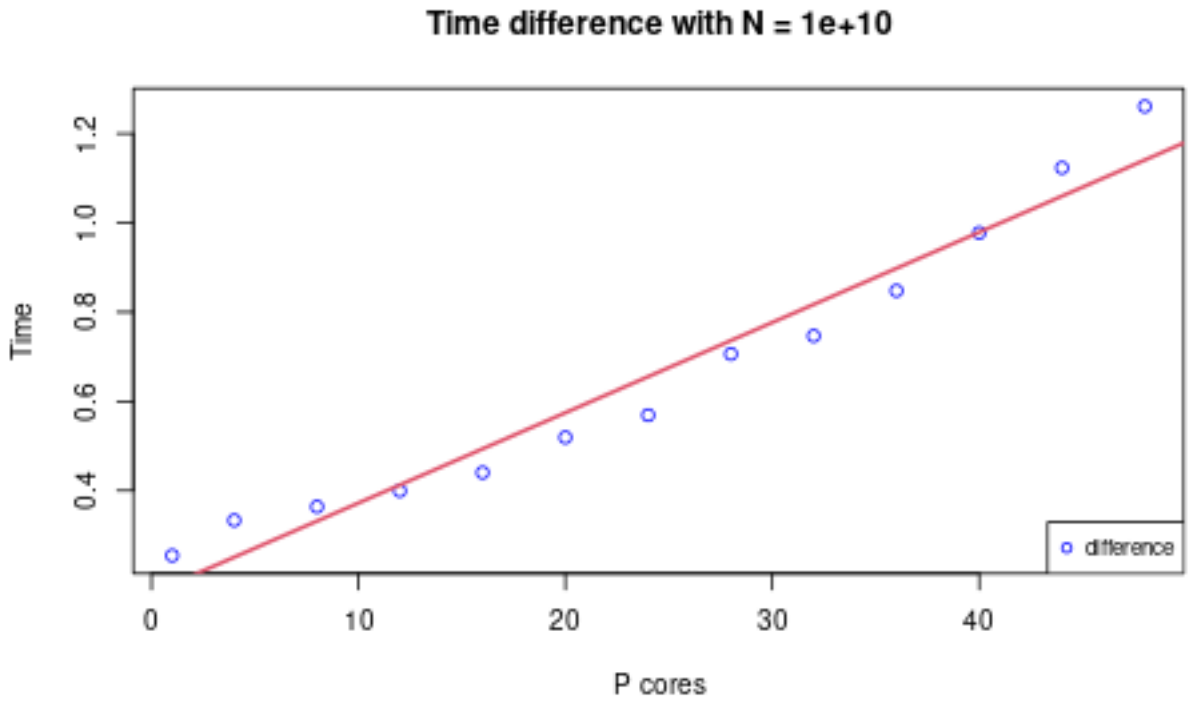
Looking back at the time charts for strong scalability, it seems that there is a linear dependence on P: in fact, after fixing N, we have that as P grows so does the absolute difference among walltime and total elapsed time. The main time components included in "elapsed" but not in "walltime" are the system and the printing. Despite the fact there might be constant terms, the number of printed sentences seems increasing proportionally to the number of cores; similarly, the time required to spawn and synchronize parallel tasks gives the impression to be somehow expanding with P. Let's try to visualize the plots of this absolute difference; then let's fit a regression model:



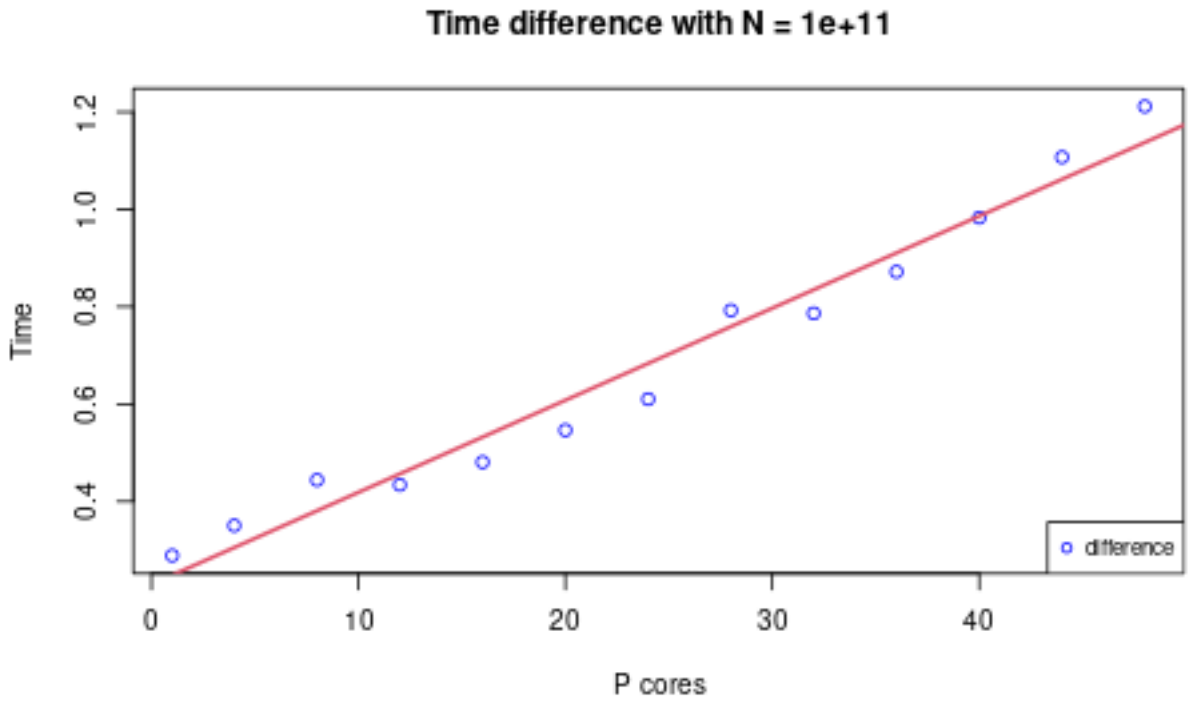

```
##
## Call:
## lm(formula = diffr ~ P_seq)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.080643 -0.040750 -0.006134  0.057404  0.101460
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.156191   0.033160    4.71 0.000639 ***
## P_seq       0.020355   0.001172   17.36 2.42e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.06275 on 11 degrees of freedom
## Multiple R-squared:  0.9648, Adjusted R-squared:  0.9616
## F-statistic: 301.5 on 1 and 11 DF,  p-value: 2.422e-09
```




```
##
## Call:
## lm(formula = diffr2 ~ P_seq)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.07873 -0.03712 -0.01500  0.02873  0.10904
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.189402   0.030705   6.168 7.02e-05 ***
## P_seq       0.019003   0.001086  17.505 2.22e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0581 on 11 degrees of freedom
## Multiple R-squared:  0.9653, Adjusted R-squared:  0.9622
## F-statistic: 306.4 on 1 and 11 DF,  p-value: 2.221e-09
```




```
##
## Call:
## lm(formula = diffr3 ~ P_seq)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.08644 -0.05338 -0.01337  0.06315  0.11994
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.170186   0.036450   4.669 0.000684 ***
## P_seq       0.020222   0.001289  15.693 7.08e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.06897 on 11 degrees of freedom
## Multiple R-squared:  0.9572, Adjusted R-squared:  0.9534
## F-statistic: 246.3 on 1 and 11 DF,  p-value: 7.079e-09
```




```
##
## Call:
## lm(formula = diffr4 ~ P_seq)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.073559 -0.048548 -0.003333  0.045057  0.073909
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.229713   0.028725   7.997 6.56e-06 ***
## P_seq       0.018914   0.001016  18.624 1.15e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.05436 on 11 degrees of freedom
## Multiple R-squared:  0.9693, Adjusted R-squared:  0.9665
## F-statistic: 346.9 on 1 and 11 DF,  p-value: 1.146e-09
```

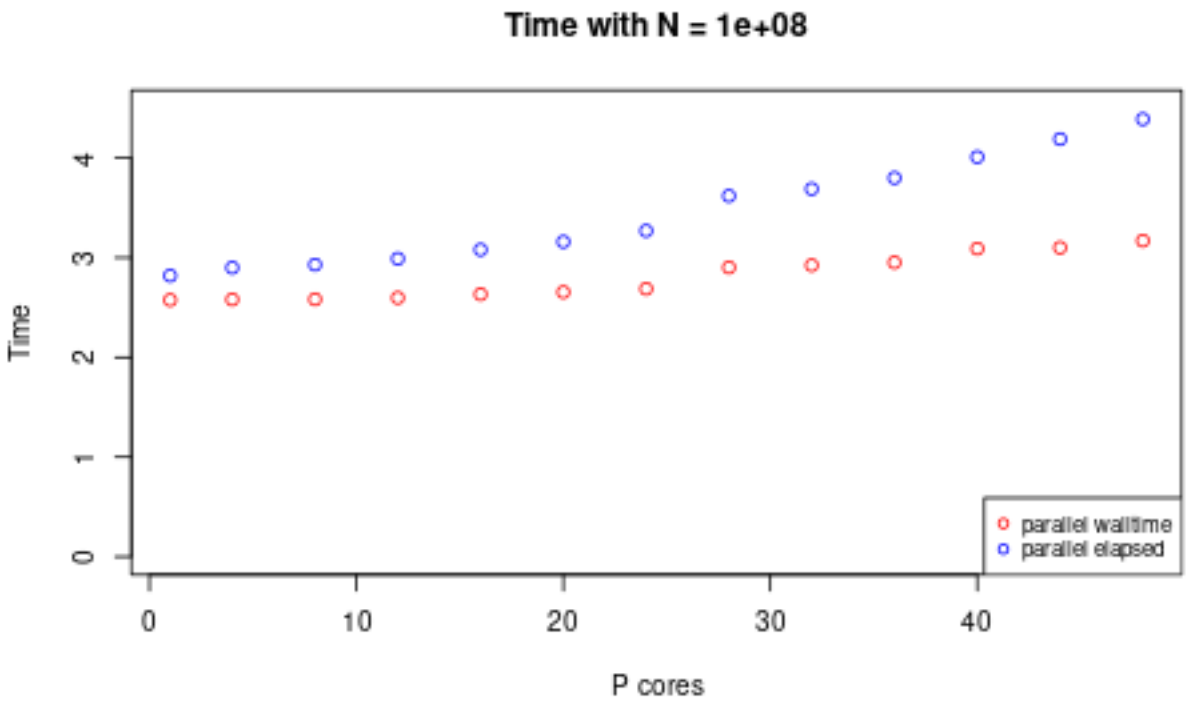
At first glance it would look like a regression line works perfectly. This is confirmed by the stats of the regression showing very similar slope coefficients (values between 0.018804 and 0.020355) for each computed regression: therefore it seems N doesn't have meaningful impact in the overhead function. Moreover we discover that the p-values associated with the slope coefficients are very small (order of 10^{-9}): thus we reject the hypothesis of coeff= 0. Also the intercept coefficients are not far from each other (between 0.156191 and 0.234349). We can conclude that there is a strong relationship of linear dependence on P and precisely:

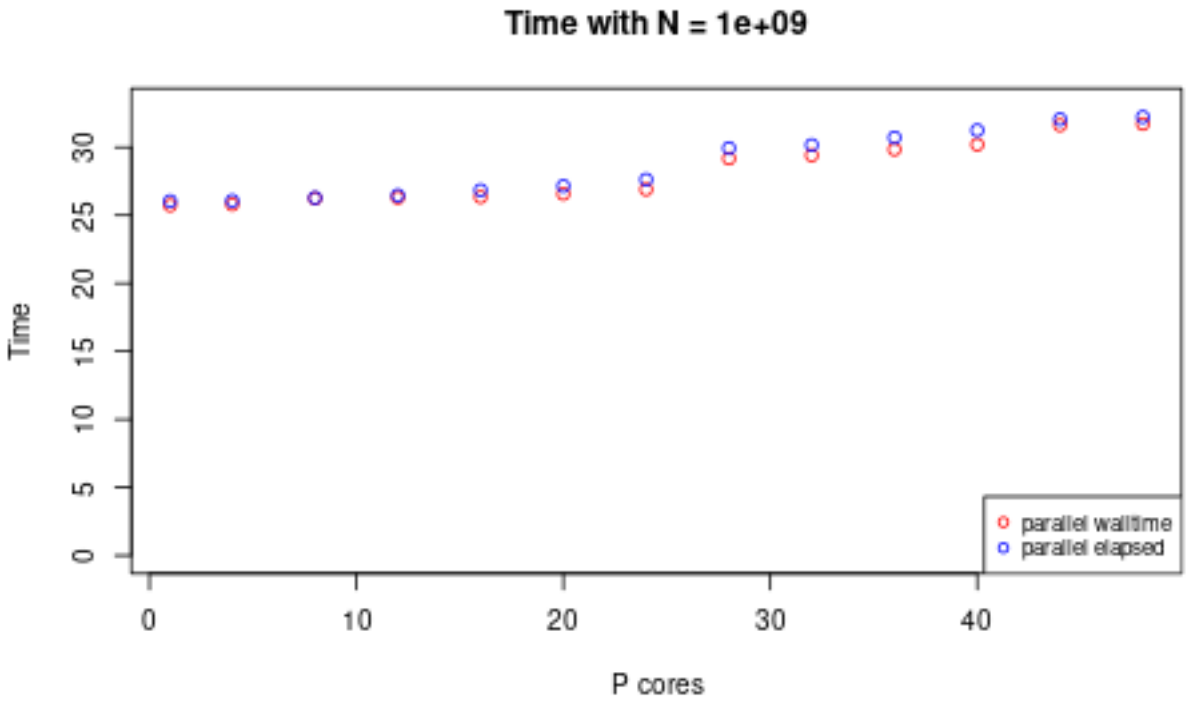
$$T_{overhead}(P) \approx c + k \times P$$

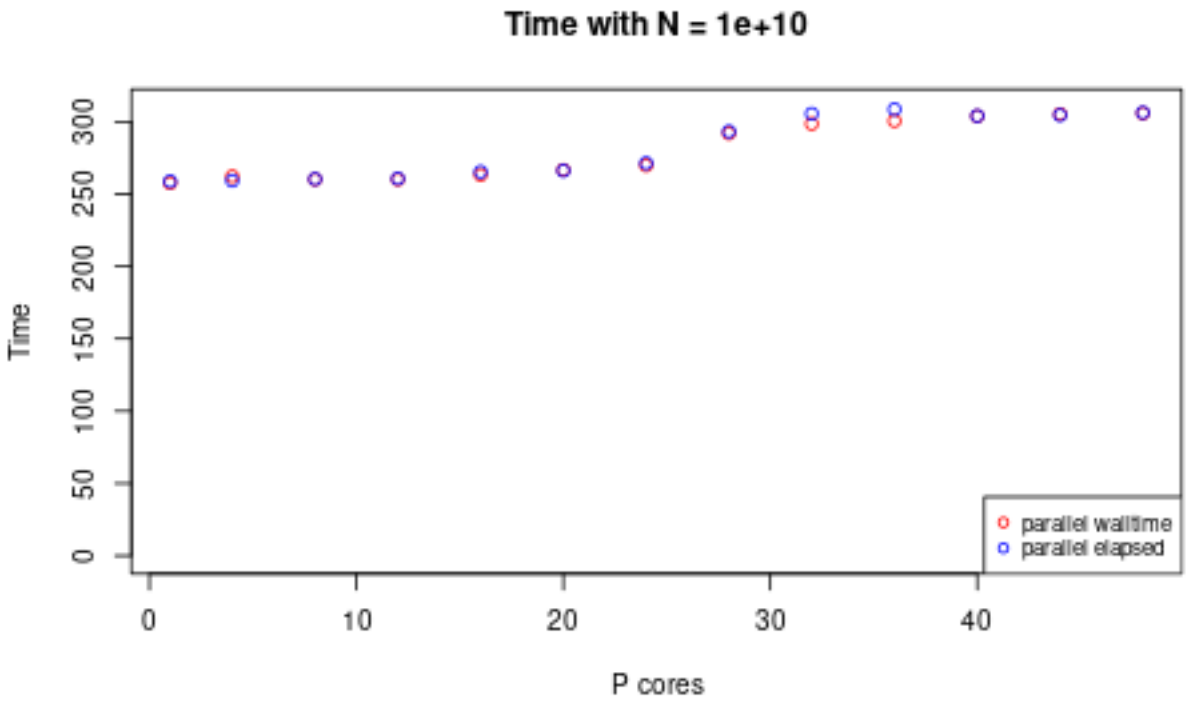
where the intercept c is likely to be in the 95% confidence interval 0.1863731 ± 0.03194011; while the slope coefficient k should be included in $0.01962321 \pm 0.0007708309$.

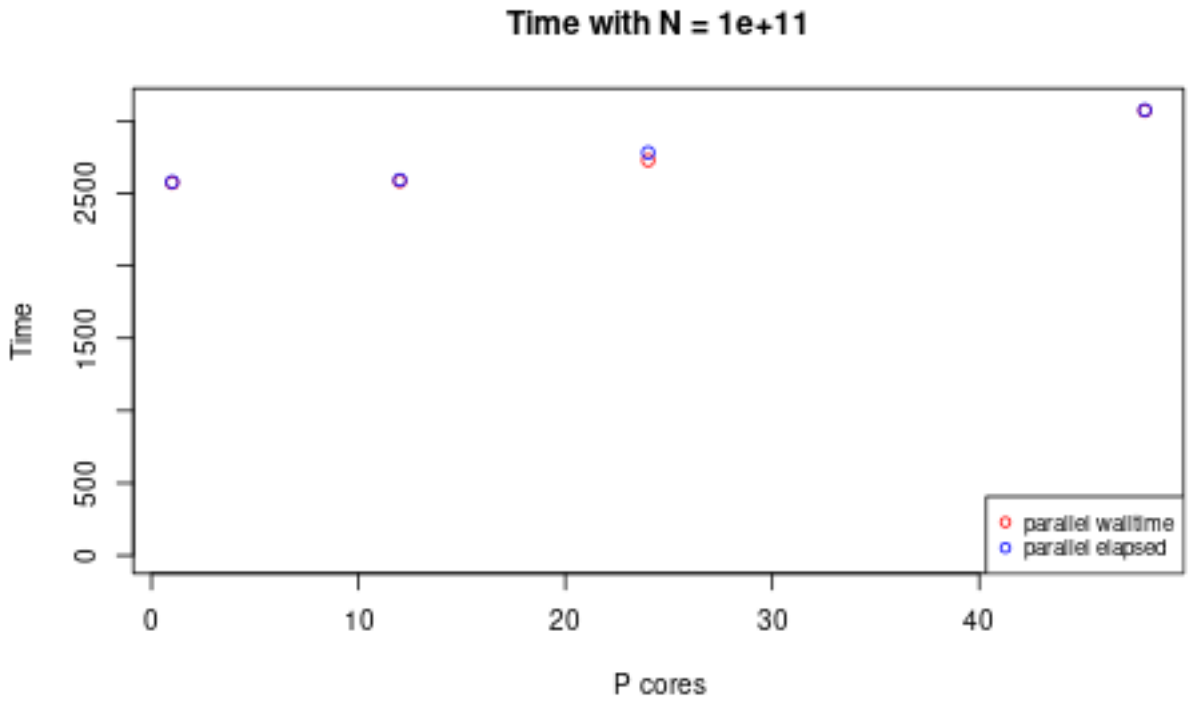
2.3: weak scaling

Let's look at the plots for weak scalability.



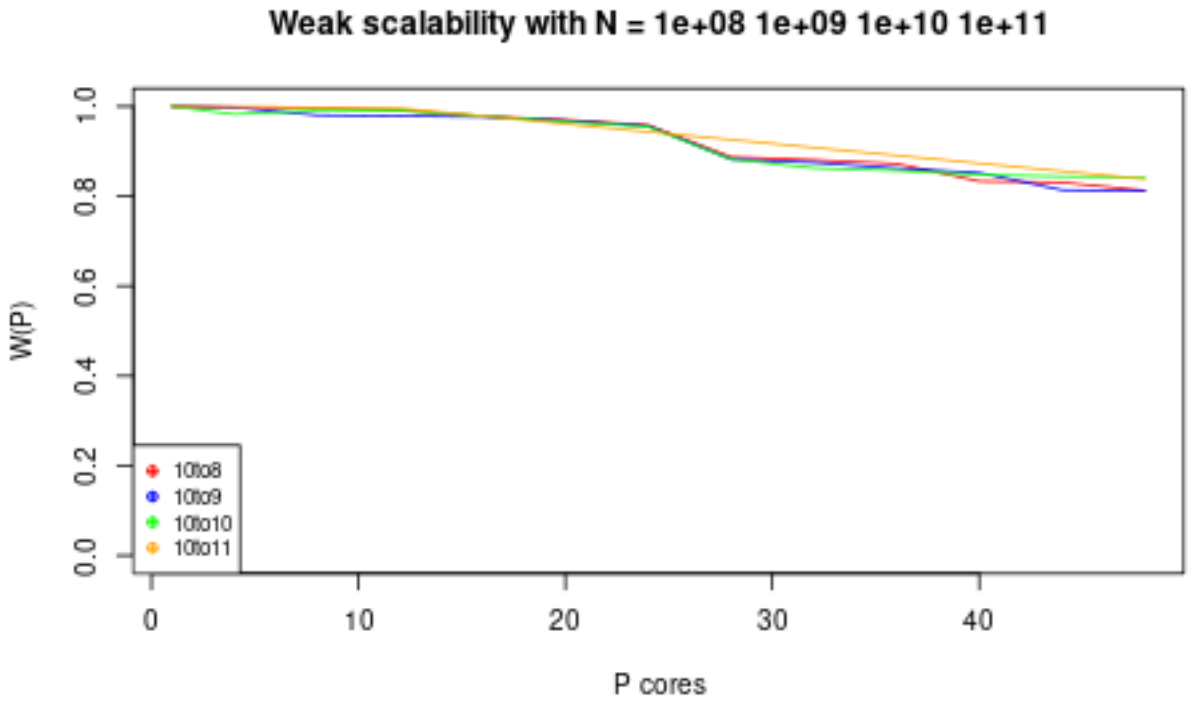

Time with N = 1e+09
Time
0
5
10
15
20
25
30
0
10
20
30
40
P cores
parallel walltime
parallel elapsed


Time with N = 1e+10
Time
0
50
100
150
200
250
300
0
10
20
30
40
P cores
parallel walltime
parallel elapsed



According to perfect weak scalability, time should remain constant; actually in the plots we see that there are no perfect horizontal lines: that's because walltime includes communication time among the cores lifting the overall time as P increases (logarithmically according to the enhanced model described in the first section).

Let's look at the plots for weak scalability (weak efficiency function) using walltime as before.



Again, theoretically speaking $\frac{T(1)}{T(P)}$ should stay constant. However as expected, we see curves slightly skewed towards the bottom of the chart. The line referred to weak efficiency for $N = 10^{11}$ is not so accurate because of the very few data points available. Despite this fact, the curves tend to share a very similar and close path: it is probably due to the fact that we were required to increse the number of iterations linearly with the number of cores, thus approaching the straight line limit (here it would be a horizonatal line) as in strong scalability (when the number N was big).