Report on Assignment 1 HPC

Sebastiano Zagatti

13 November 2020

1 Theoretical Model

Performance model for a simple parallel algorithm: sum of N numbers.

- Serial Algorithm:
$$T_{\text{serial}} = T_{\text{read}} + NT_{\text{comp}}$$
- Parallel Algorithm:
$$T_{\text{parallel}} = T_{\text{comp}}(P - 1 + N/P) + T_{\text{read}} + 2(P-1)T_{\text{comm}}$$

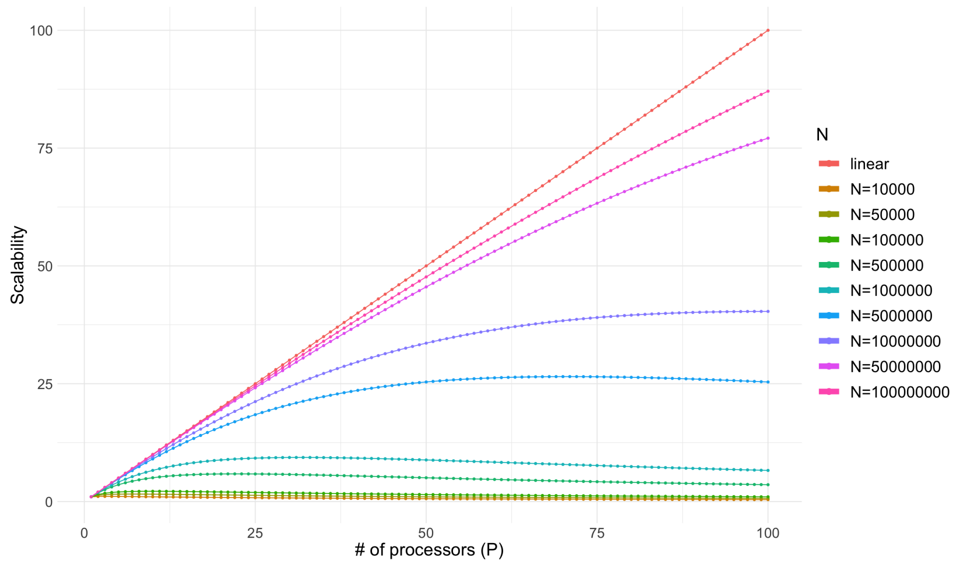
where:

- T_{comp}= time to compute a floating point operation;
- T_{read} = time master takes to read;
- T_{comm} = time each processor takes to communicate one message, i.e. latency.

We assume:

- $T_{\text{comp}} = 2 \times 10^{-9}$ s;
- $T_{\text{read}} = 1 \times 10^{-4}$ s;
- $T_{\text{comm}} = 1 \times 10^{-6}$ s.

and plot the scalability curves for various values of N:



Comments:

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

  As N increases we see that the algorithm scales better for a higher number of processors. This makes sense considering Gustafson's and Amdahl's laws: as we increase the size of the problem (so as we increase N) the serial fraction of our problem decreases and as the serial fraction decreases we have better scalability for a higher number of processors. In particular we can see that the best results are obtained for values of N grater than 5×10^7.

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

  The number of processors for which the algorithm produces the best results depends on the size of the problem. We can observe that for values of N lower than 10^7 we get a scalability curve that has a maximum value and then decreases: the value of P corresponding to the maximum value is the number of processors that produce the best results. For values of N greater than 10^7 we consider the best value to be $P = 100$, but if we would increase the number of available processors, we would obtain a behaviour similar to the one previously described.
  For the values displayed in the previous graph we get:

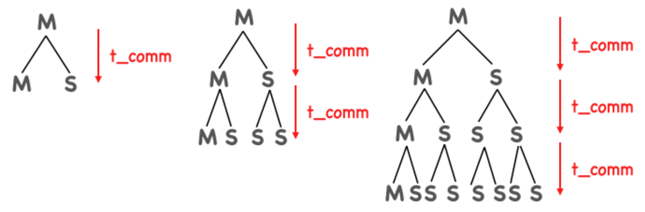
N	Best P
10000	3
50000	7
100000	10

N	Best P
500000	22
1000000	32
5000000	71

N	Best P
10000000	100
50000000	100
100000000	100

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

  We can modify the communication model of the algorithm obtaining a logarithmic dependency with the number of processors: instead of using one t_{comm} to communicate with each of the slave processors, the master will use the first t_{comm} to communicate with one processor (the first slave), then for the second t_{comm} both the master and the first slave will communicate with one different processor each.



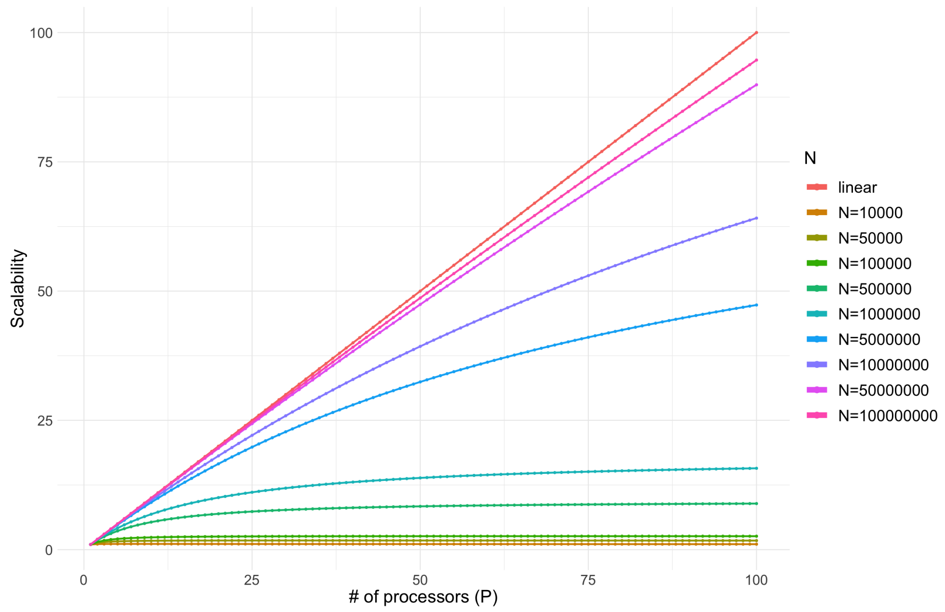
In this way, we have reached four processors in $2t_{\text{comm}}$, while we would have reached only three processors with the previous algorithm in the same time spawn.
We obtain that the total time needed to reach P processors is given by:

$$t_{\text{comm}} \log_2(P)$$

The same idea can be applied from bottom to top when the different slave processors communicate the results to the master one, so we can conclude that the new model will be:

$$T_{\text{parallel}} = T_{\text{comp}}(P - 1 + N/P) + T_{\text{read}} + 2T_{\text{comm}} \log_2(P)$$

Using this new model we obtain the following scalability curves:



When compared to the previous one, this model provides better results. Further proof of this is given by the fact that the best values of P are higher than the previous ones:

N	Best P
10000	7
50000	34
100000	66

N	Best P
500000	100
1000000	100
5000000	100

N	Best P
10000000	100
50000000	100
100000000	100

2 Play with MPI Program

2.1 Compute strong scalability of a mpi_pi.c program

- *Determine the CPU time required to calculate PI with the serial calculation using 10000000 (100 millions) iterations (stone throws).*
  Make sure that this is the actual run time and does not include any system time.

  The time required by the serial program to calculate PI with 100 million iteration obtained is:

  $$t_{walltime} = 2.57\,s$$

  We can compare this value with the elapsed time obtained using /usr/bin/time:

  $$t_{elapsed} = 2.57\,s$$

  thus, we can conclude that the value obtained is the actual run time and does not include any system time.

- *Get the MPI code running for the same number of iterations (i.e. moves) on one processor.*
  Comparison of this to the serial calculation gives you some idea of the overhead associated with MPI. Report and discuss if there are any differences in performance between the two codes on one single core.

  The MPI code has been run considering 10^8 iterations on one processor and the result obtained is:

  $$t_{\text{parallel (1 core)}} = 2,84\,s$$

  we compare this result with the one obtained in the previous paragraph:

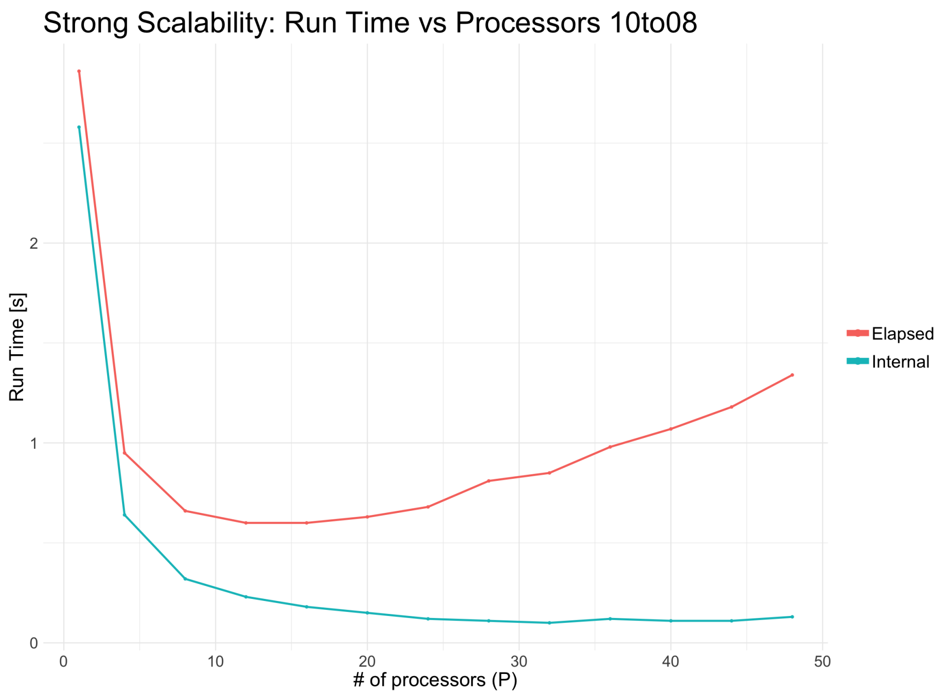
  $$t_{\text{serial (1 core)}} = 2,57\,s$$

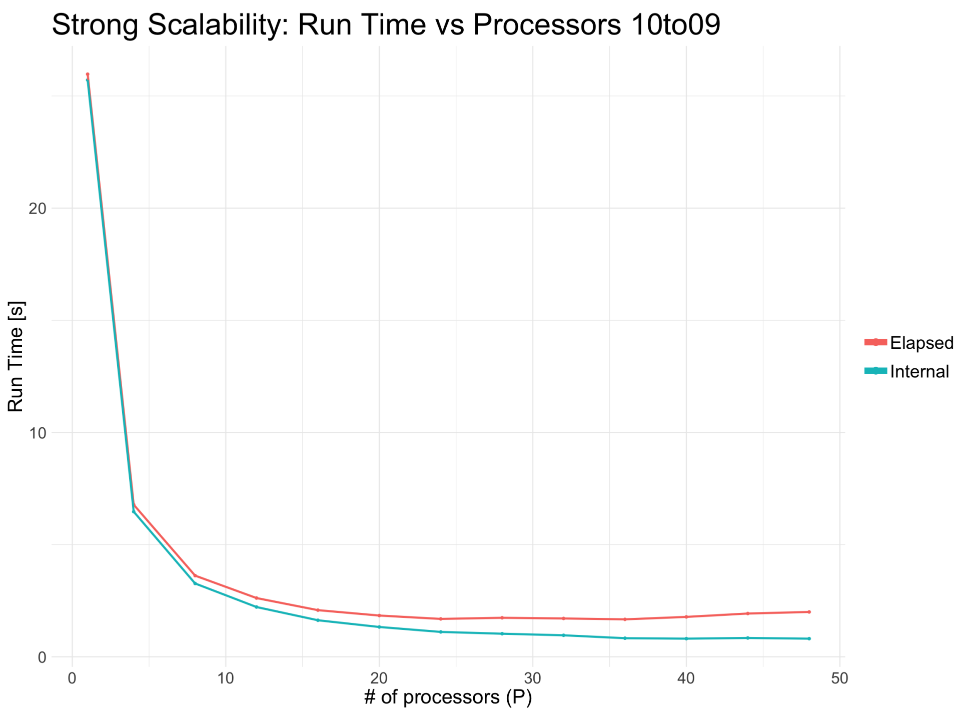
  We observe that the difference between the two values is $0,27\,s$, which corresponds to a 11% relative error, therefore we can't consider this difference as a simple statistical fluctuation and we can conclude that it represents the overhead associated to MPI and the parallelization.
  In order to compare the performance of the two codes on one single core, different numbers of iterations have been considered (namely the ones needed for the following points), the results, with the corresponding relative errors are displayed in the following table:

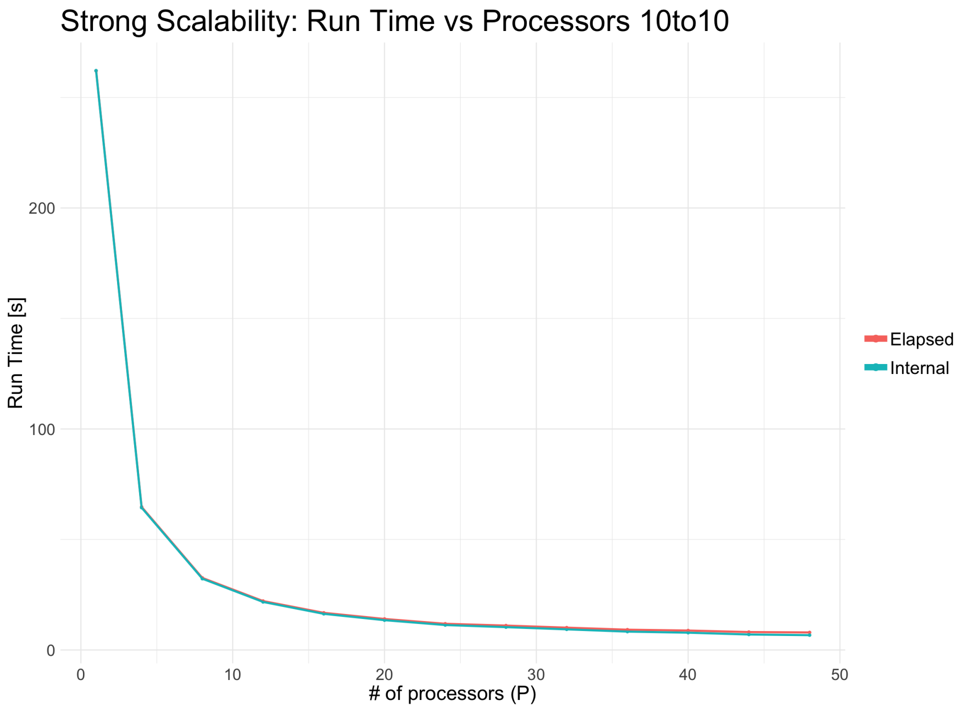
Number of Iterations	t_{serial}	t_{parallel}	Relative Error
10^8	2.57	2.84	11%
10^9	25.70	26.02	1.2%
10^{10}	256.90	257.86	0.3%
10^{11}	2570.95	2610.10	1.5%

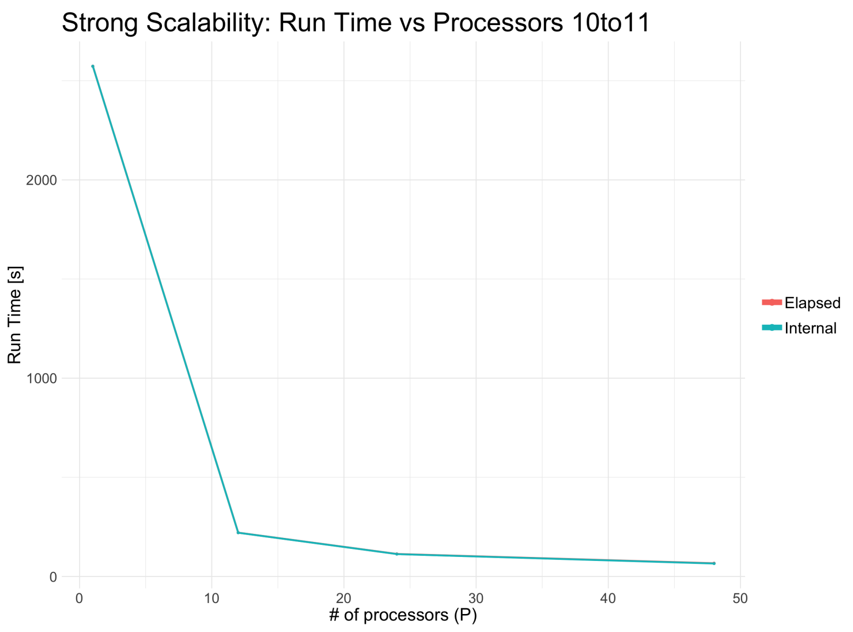
From these results we can conclude that the only case in which the MPI overhead is relevant, in comparison with the statistical fluctuations, is the one with the smaller number of iteration.
Some considerations need to be made regarding the calculation of scalability: are we allowed to use $t_{\text{parallel (1 core)}}$ as the numerator in the scalability formula or should we use $t_{\text{serial (1 core)}}$?
As said in the previous paragraph, the relative error should be $\sim 1\%$ in order to consider the difference as a statistical fluctuation and, since the overhead obtained with 10^8 iterations does not satisfy this request, the serial time $t_{\text{serial (1 core)}}$ will be used in the computation of the scalability with 10^8 iterations; in order to maintain coherence the same will be done for other numbers of iterations.

- *Strong Scaling Test: Make a plot of run time versus number of nodes from the data you have collected. Consider both elapsed time collected by* usr/bin/time *command and the internal time (take the highest time among all the processors.*

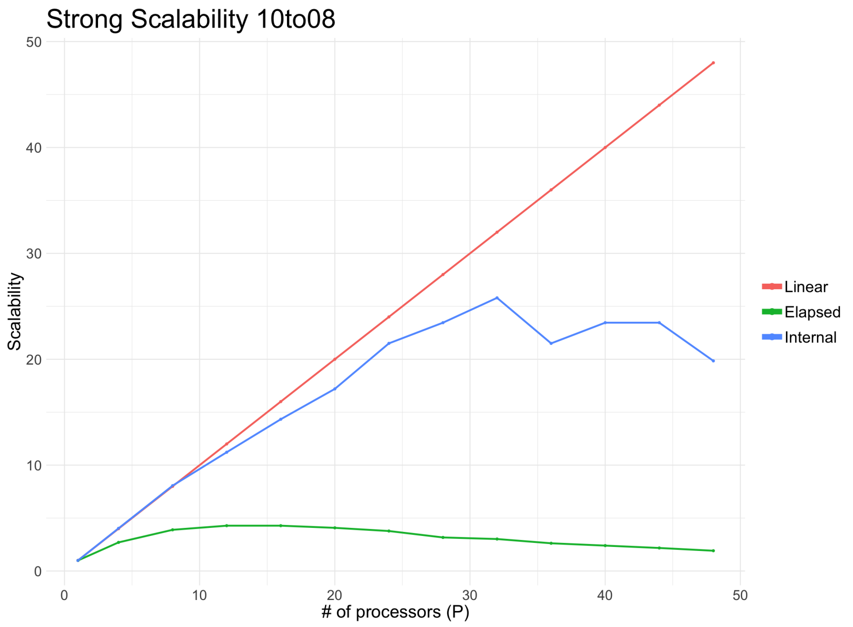


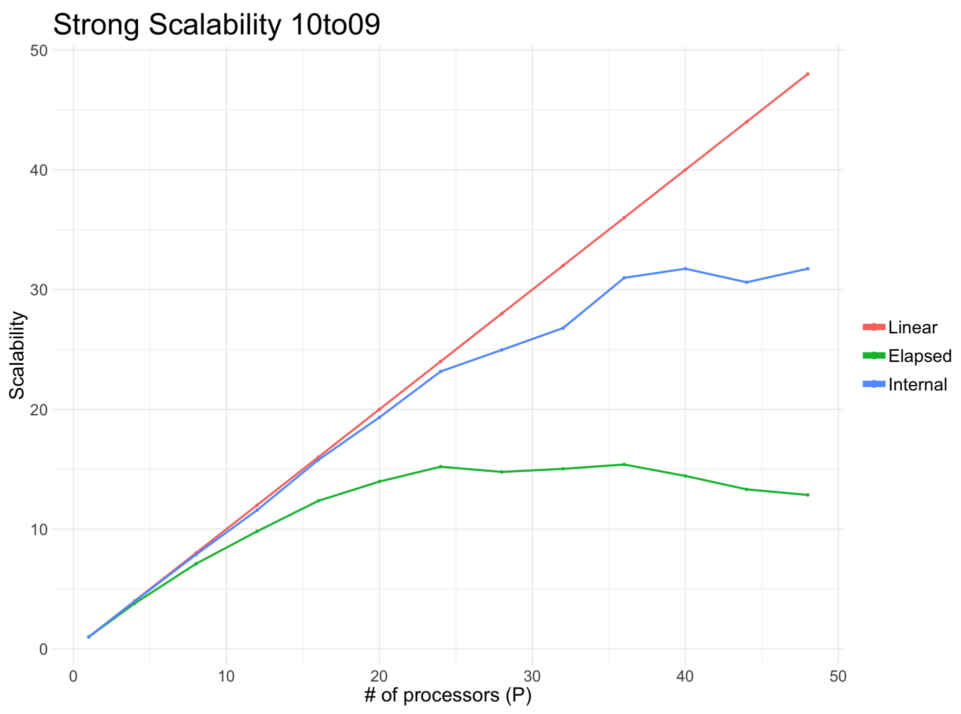

Strong Scalability: Run Time vs Processors 10to09
Run Time [s]
# of processors (P)
0
10
20
30
40
50
Elapsed
Internal

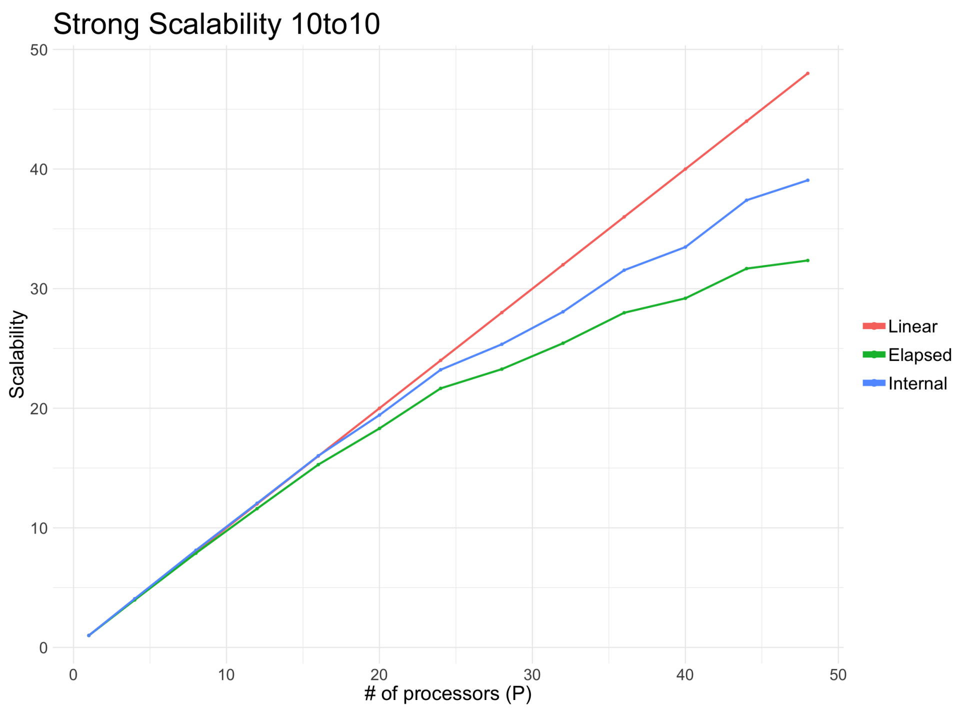

Strong Scalability: Run Time vs Processors 10to10
Run Time [s]
# of processors (P)
0
100
200
10
20
30
40
50
Elapsed
Internal

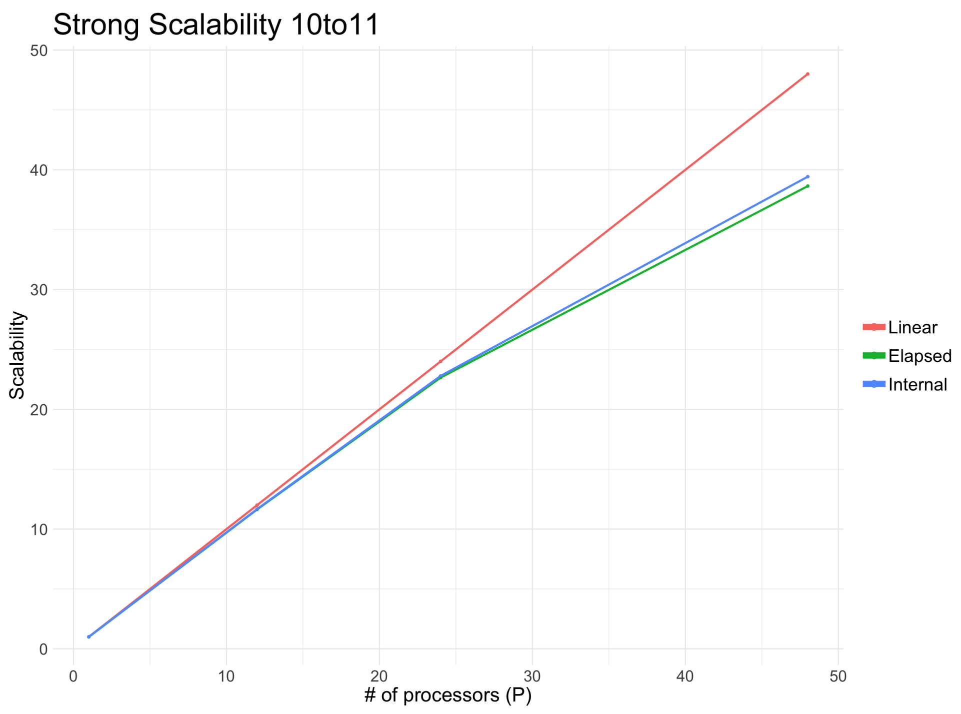


Provide a plot where you compare the scalability curves obtained using these two different timing modality. Perfect scalability here would yield a straight line graph. Comment on your results. Repeat the work playing with larger datasets.




Strong Scalability 10to09
Scalability
# of processors (P)
0
10
20
30
40
50
Linear
Elapsed
Internal

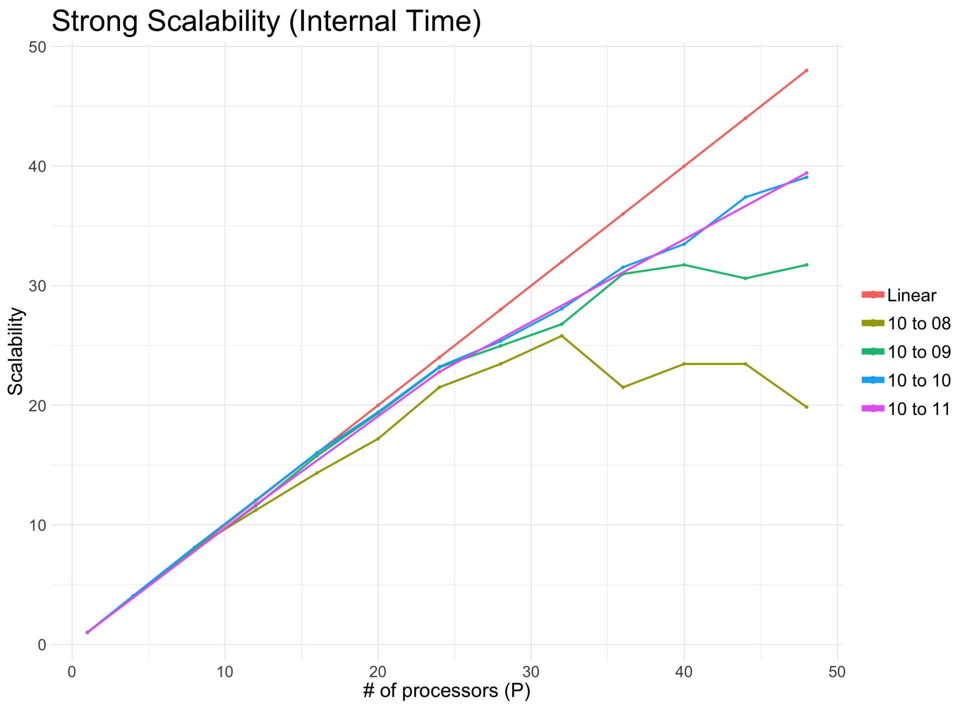

Strong Scalability 10to10
Scalability
# of processors (P)
0
10
20
30
40
50
Linear
Elapsed
Internal



In all four cases (different number of iterations N) we can observe that the results obtained with the internal time and the ones obtained with the elapsed time are different. This difference derives from the kind of processes that are involved in each time calculation: the internal time keeps track of all the work done by each processor, including the communication time related to the master-slave communication protocol, while the elapsed time keeps track of the whole execution of the program; therefore, we can conclude that the difference between elapsed and internal time are mainly due to two factors: the overhead related to MPI and other code operations, like variable definition and I/O.
We observe that, as the number of iterations increases, the difference between the two scalability measures decreases; this can be explained considering that code operations, which are not involved in the internal time, are independent from N, while the same can not be said for the overall execution time of the code, which grows as N. In addition, as discussed in the previous paragraph, the overhead related to MPI and parallelization becomes negligible when the number of iterations increases, so we can conclude that as N increases both the time needed for the code operations and the MPI overhead will become negligible when compared with the overall code execution time, thus explaining the behaviour of the scalability curves for different N values.

Another observation that can be made involves the comparison with the perfect scalability straight line: in all the cases, both the elapsed time scalability curve and the internal time scalability curve display a behaviour that can be justified by Gustafson's and Amdahl's laws: as the problem size increases, the serial fraction of the problem decreases, resulting in a better behaviour of the scalability.

- *Provide a table as explained above for each measurement final plot with at least 3 different sizes and for each of the them please report and comment your final results; to produce this final graph just use one of the two times at your choice. Comment again the results and your choice.*



The internal time includes only the computation and communication done by the processors and does not involve other code instruction or any overhead, therefore it's a better choice if we want to compare our results with the theoretical predictions, since they only take into account computation and communication as well.
As said in the previous paragraph, we can observe the typical behaviour of the scalability curves when we increase the number of iterations; we can notice a similar behaviour for 10^{10} and 10^{9}, in order to appreciate the difference in the performance between the two curves, we would need to compute the scalability considering more than 48 processors, as explained in section 1 of this report.

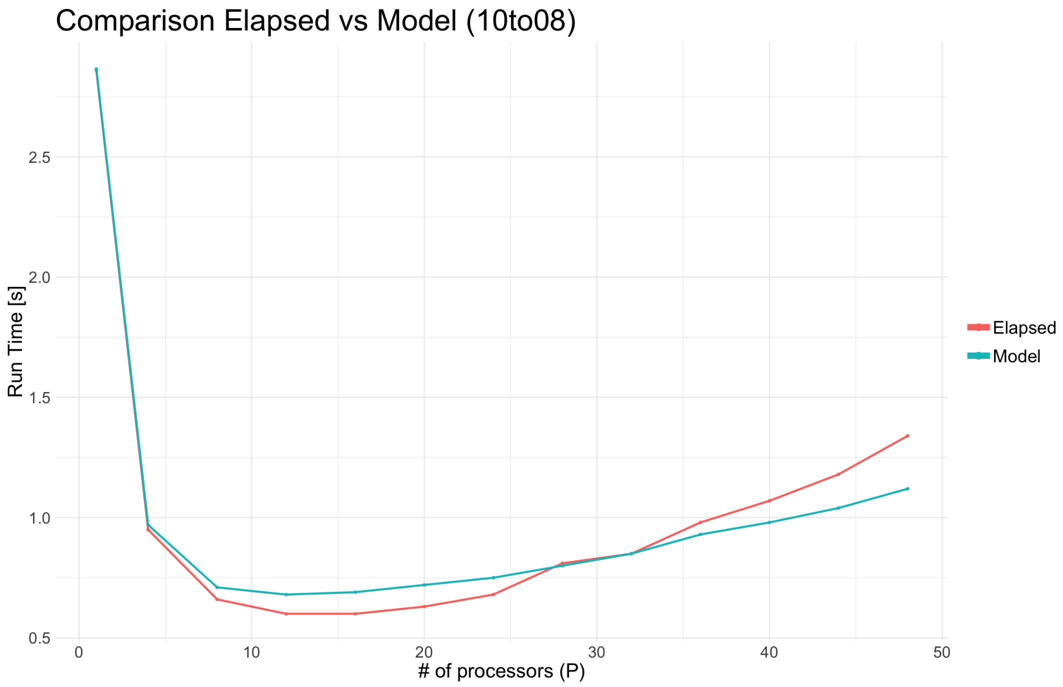
2.2 Identify a Model for the Parallel Overhead

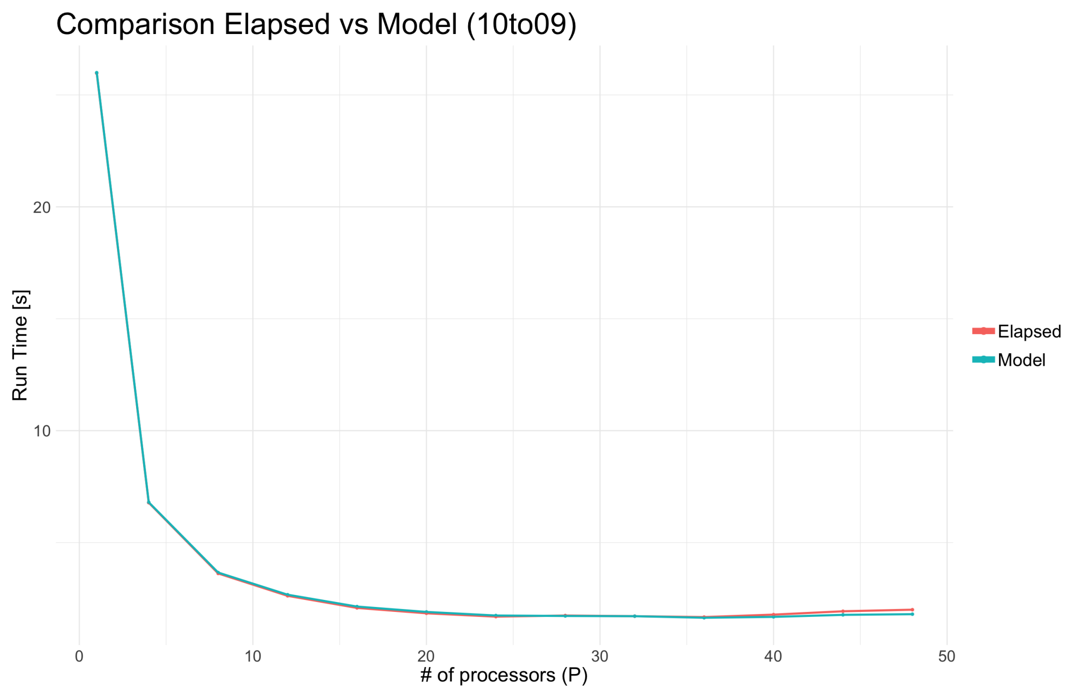
In this section you are asked to identify a simple model that estimate the parallel overhead of our simple program as a function of the number of processor. We ask here to extend the simple model discussed previously and adapt/modify to the present program. To do this please review carefully all the data collected so far and identify which is the indirect method to measure the parallel overhead from the output of the program.

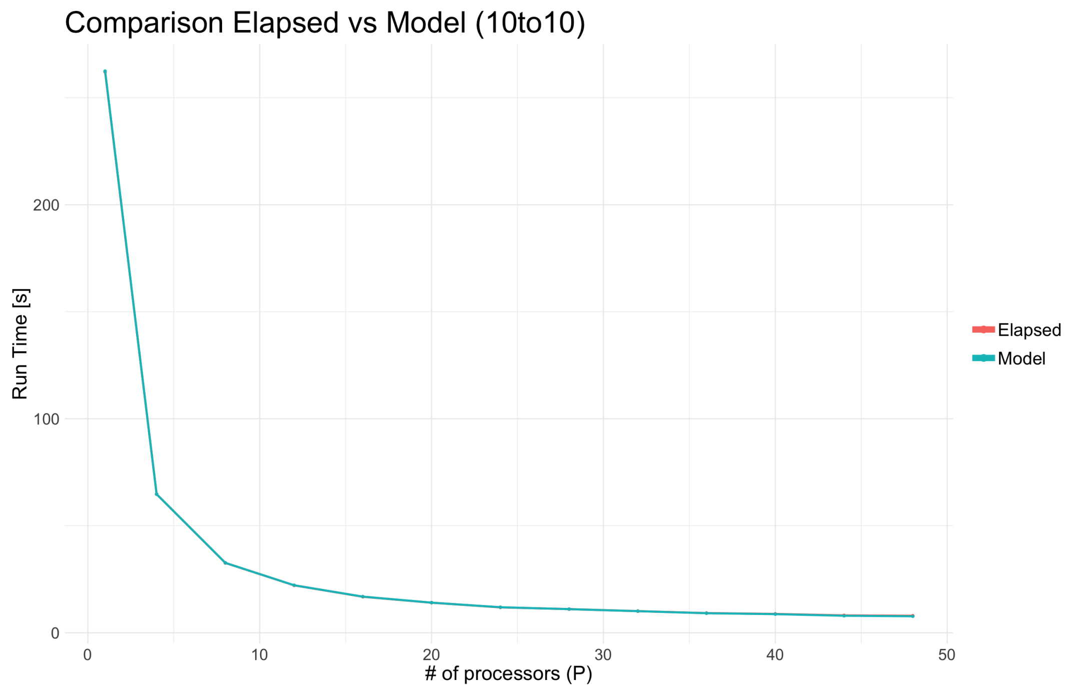
In order to be able to model the parallel overhead, first of all we need to take into account the difference between the elapsed time and the internal time, trying to model this difference as a function of the number of processors P.
As we have seen in the second point of subsection 2.1, the only relevant overhead between internal and elapsed time is the one on one single core, so the proposed model is given by:

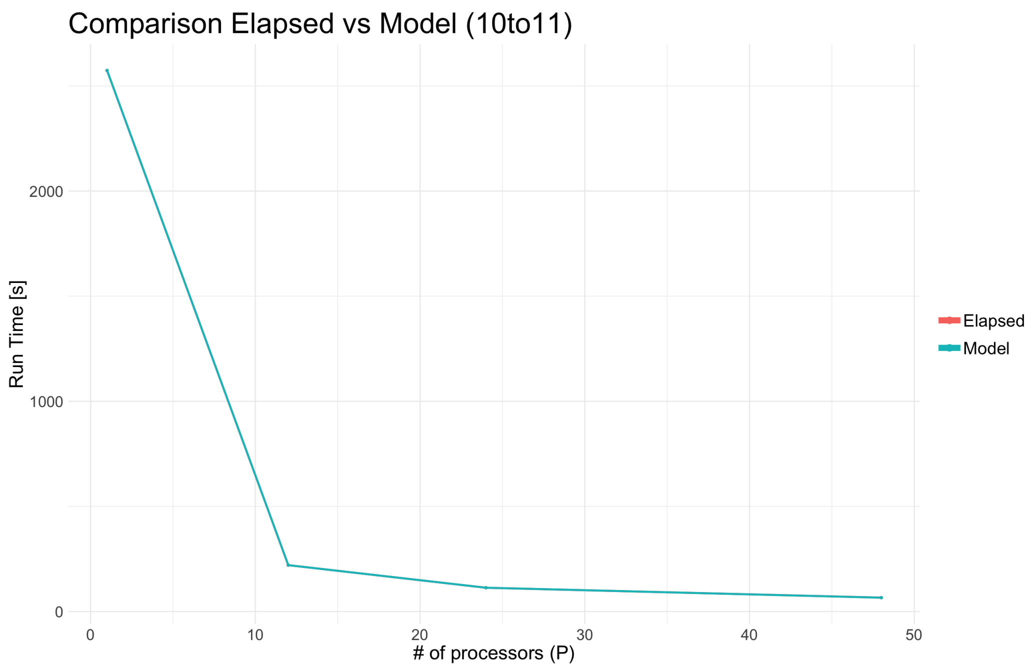
$$t_{\text{elapsed}}(P) = t_{\text{internal}}(P) + 0.27 + k\,P$$

The obtained value for the constant is $k = 0.015$ and the results for the run time are displayed in the following graphs:

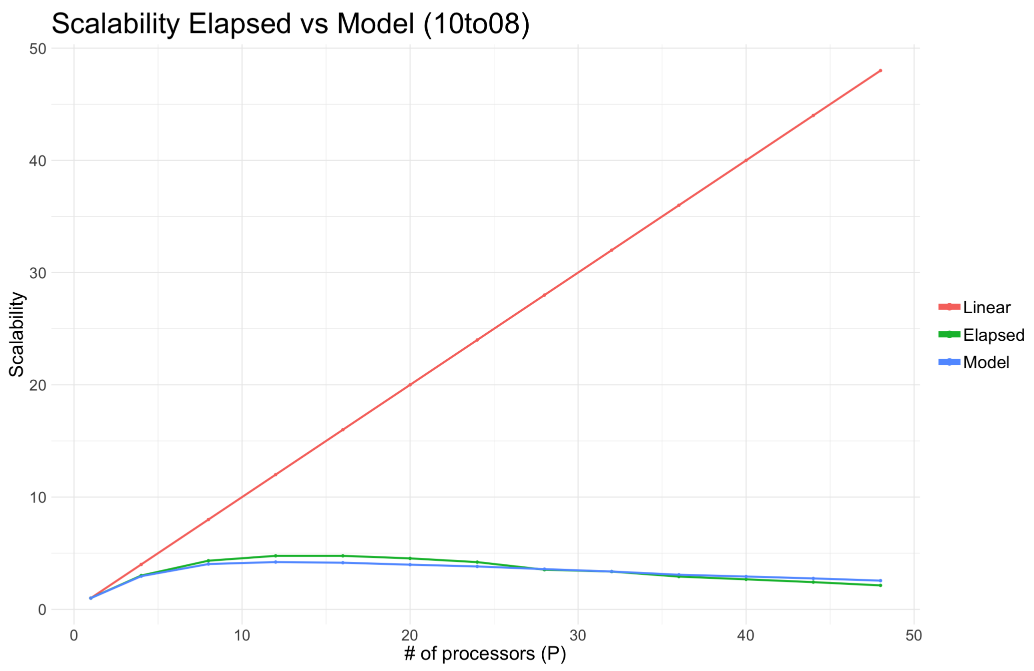


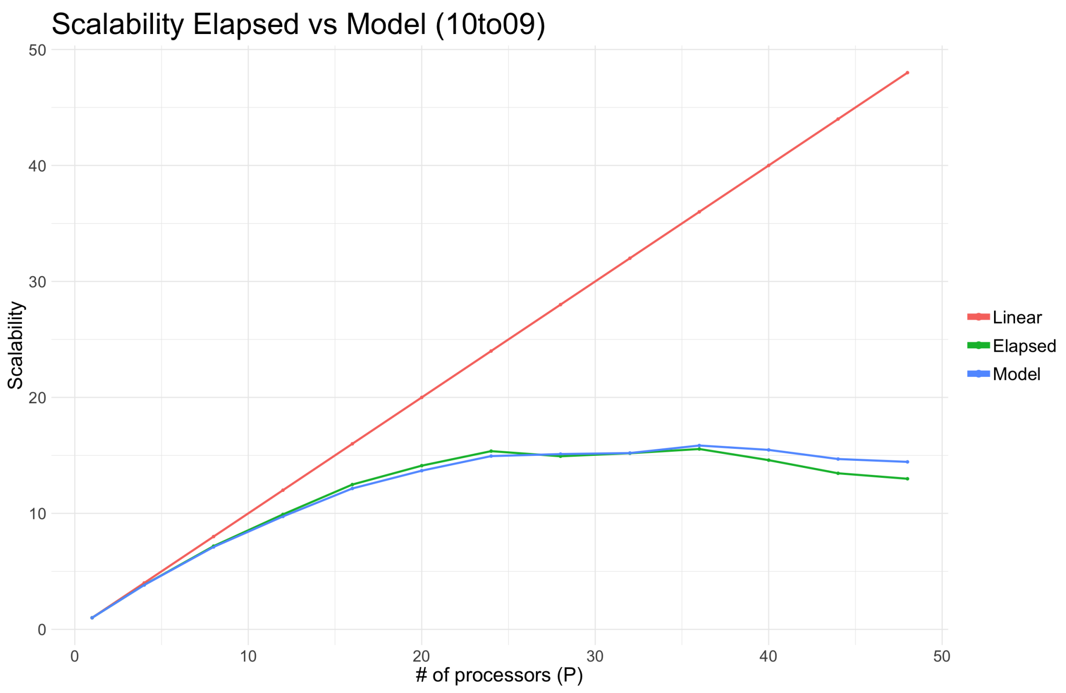

Comparison Elapsed vs Model (10to09)
Run Time [s]
20
10
0
10
20
30
40
50
# of processors (P)
Elapsed
Model

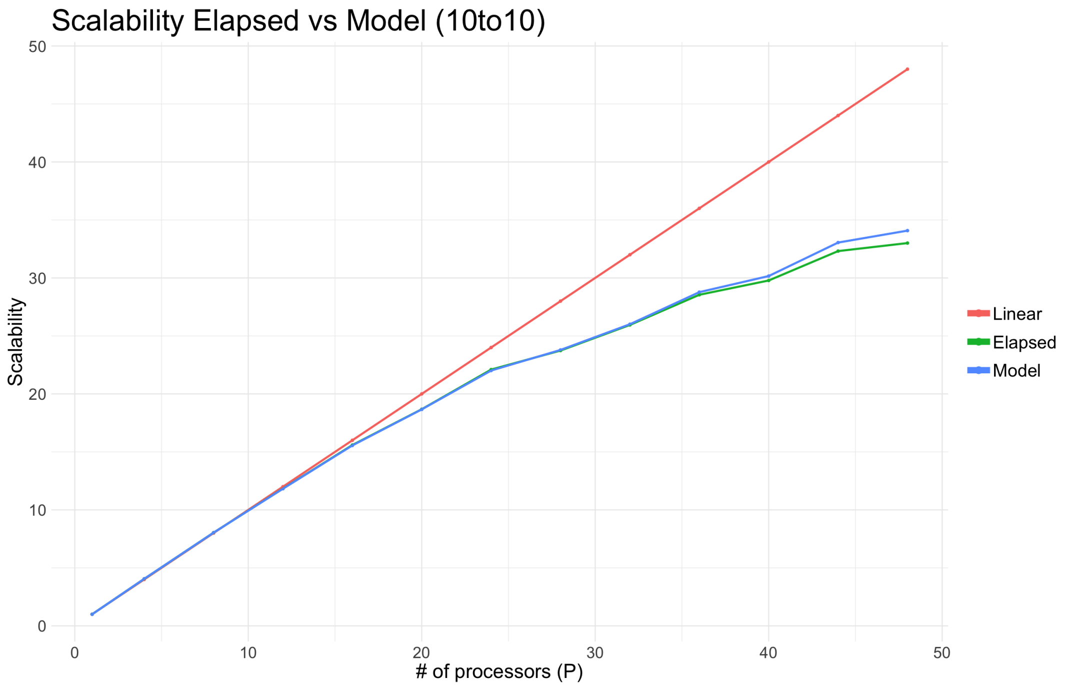

Comparison Elapsed vs Model (10to10)
Run Time [s]
200
100
0
0
10
20
30
40
50
# of processors (P)
Elapsed
Model

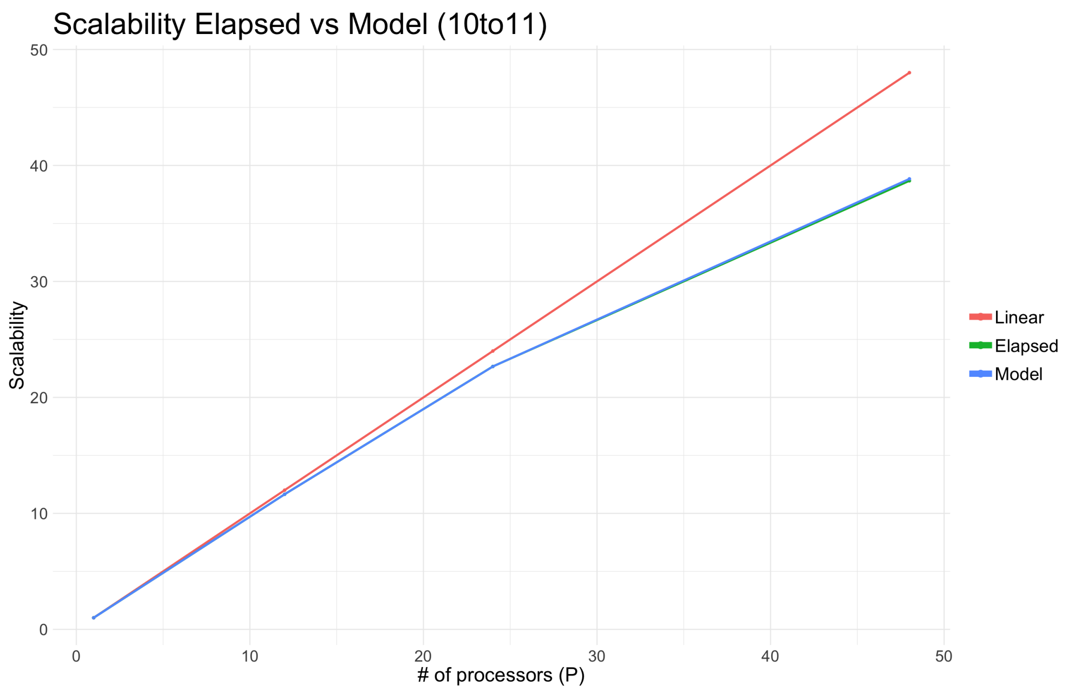


The corresponding result for the scalability are presented in the following graphs:




Scalability Elapsed vs Model (10to09)
Scalability
# of processors (P)
0
10
20
30
40
50
Linear
Elapsed
Model


Scalability Elapsed vs Model (10to10)
Scalability
# of processors (P)
0
10
20
30
40
50
Linear
Elapsed
Model



The following step is to compare the value of t_{internal} with the theoretical value of t_{parallel}, that we consider to be:

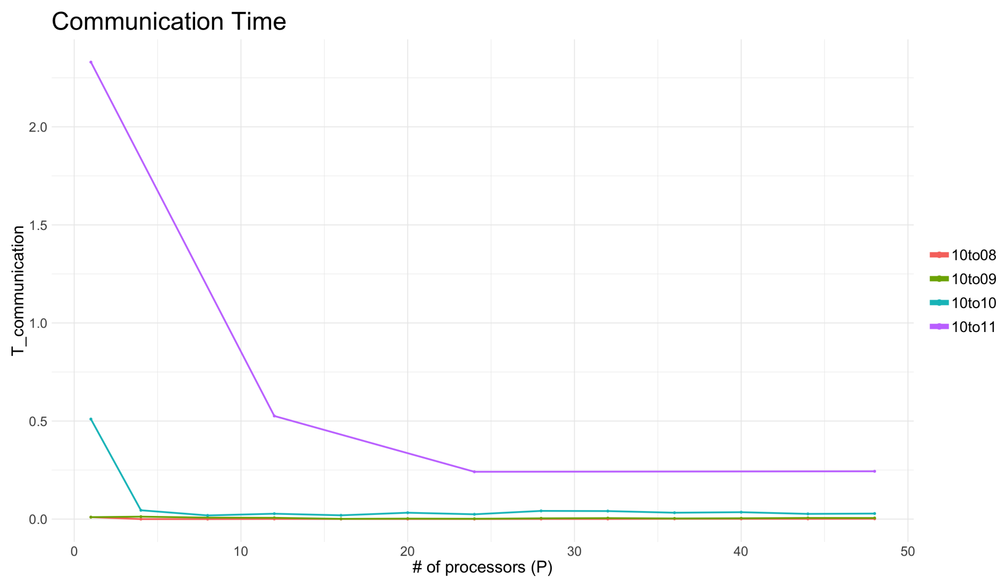
$$t_{\text{parallel}} = \frac{t_{\text{serial}}}{P}$$

The difference between t_{internal} and t_{parallel} is given by the communication time that we model as a linear dependence with the number of processors P, the constant of this dependency is the communication time:

$$t_{\text{internal}} = \frac{t_{\text{serial}}}{P} + P\, t_{\text{communication}}$$

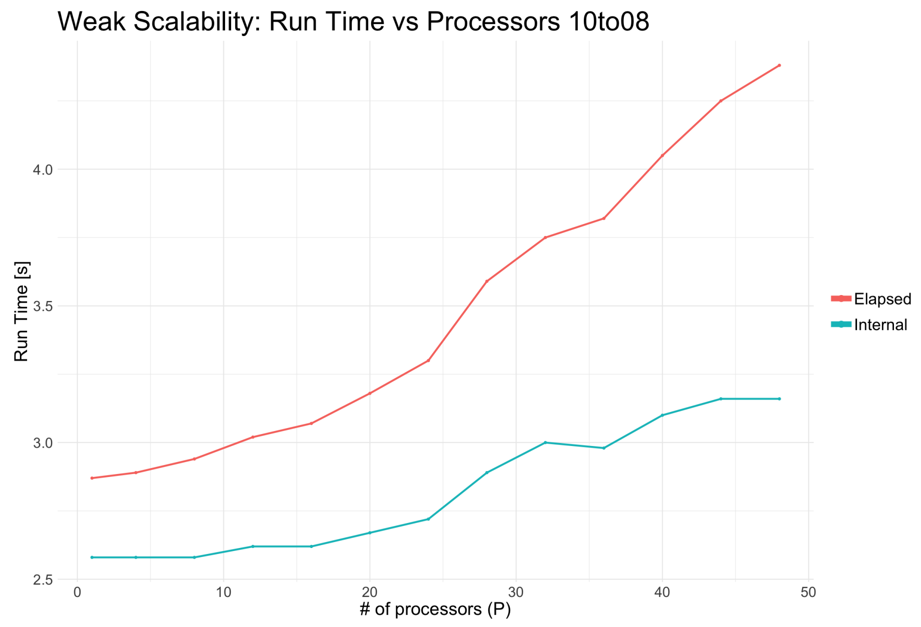
Therefore, by plotting $(t_{\text{internal}} - \frac{t_{\text{serial}}}{P})/P$ we expect to obtain a constant value. As displayed by the following graph we could verify this assumption only for the cases of $N = 10^8$ and $N = 10^9$, but not for the other two cases.
A possible improvement to this result could be obtained by increasing the number of measures we take in to account, in order to obtain better statistical evidence of our results; another possibility is to consider a linear dependence of $t_{\text{communication}}$ with the number of iterations N.
To conclude, we have that the general model for the parallel overhead we propose is:

$$t_{\text{elapsed}} = t_{\text{parallel}} + 0.27 + P\,(t_{\text{communication}} + k)$$

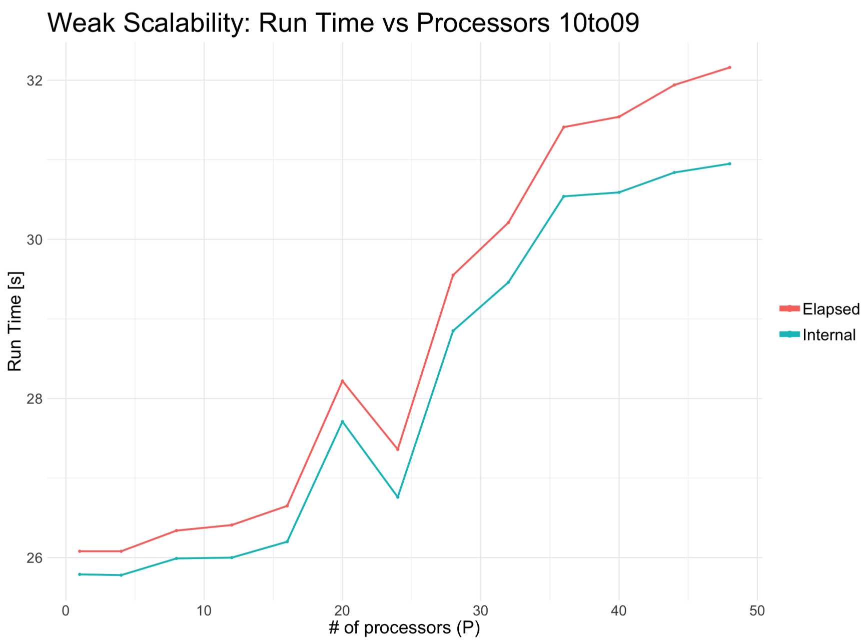


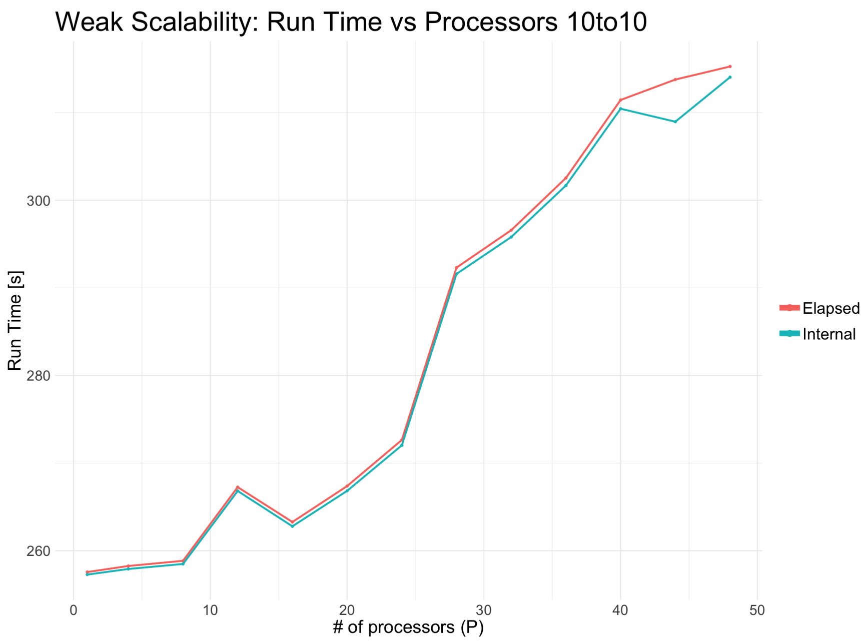
2.3 Weak Scaling

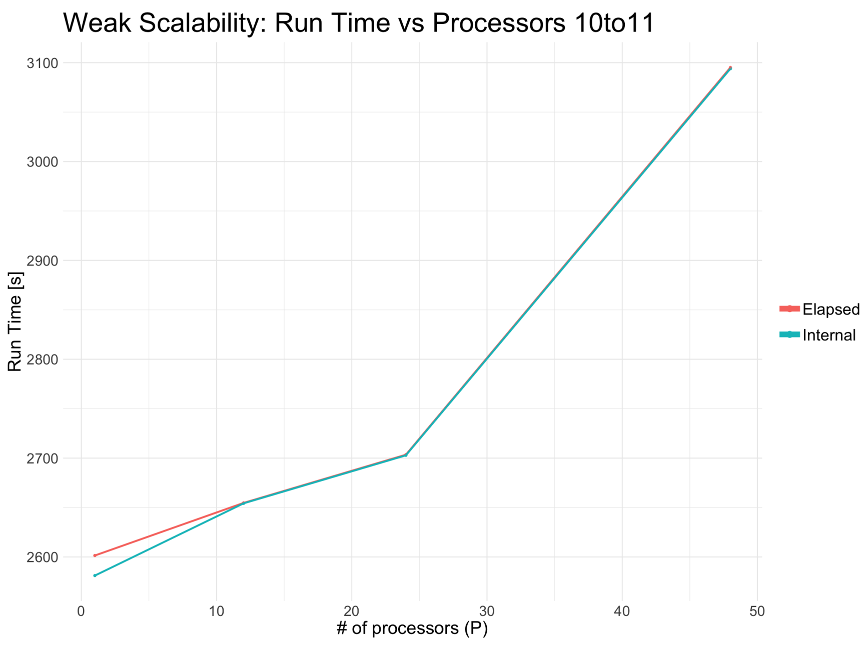
- *Weak Scaling Test: Record the run time for each number of nodes and make a plot of the run time versus number of computing nodes.*



- *Weak scaling would imply that the runtime remains constant as the problem size (i.e. number of moves, aka iterations) 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.*







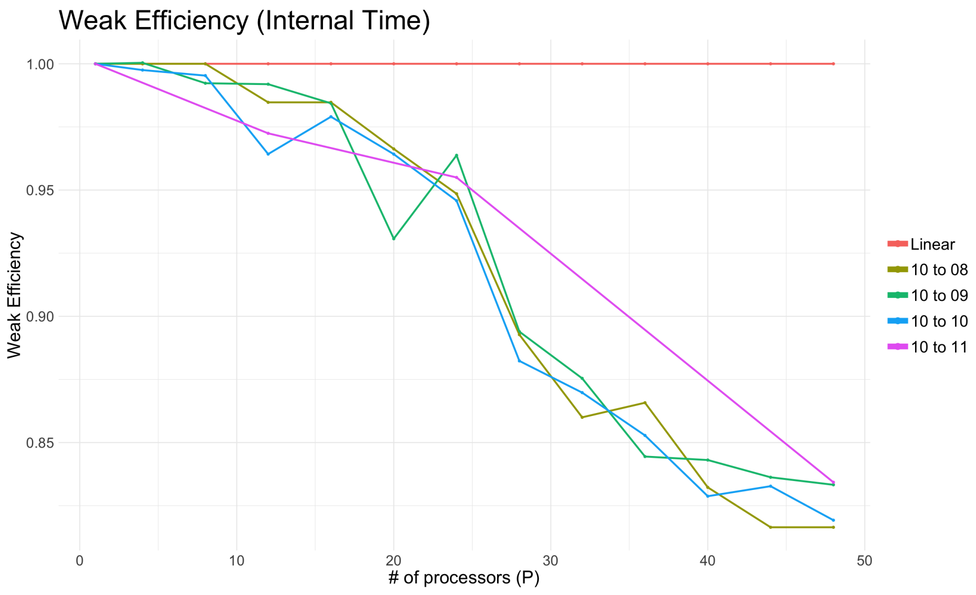
The results we have obtained show that the run time is not constant as the problem size and the number of nodes increase.

- *Plot on the same graph the efficiency $(T(1)/T(p))$ of weak scalability for the different number of iterations and comment the results obtained.*

Since we expect the run time to be constant as the problem size and the number of nodes increase, we have that the theoretical weak efficiency should be 1 so we compare our results with the straight line:

$$\text{Weak Efficiency} = 1$$

We can observe that the weak efficiency that we have measured decreases as P grows, but we notice that this kind of behaviour seems to be independent from the number of iterations we consider, therefore we can conclude that this kind of behaviour is caused by the parallel overhead, which depends linearly on the number of processors.
Once again we have considered the internal time for the same reasons explained for the strong scalability case.


Weak Efficiency (Internal Time)
Weak Efficiency
# of processors (P)
1.00
0.95
0.90
0.85
0
10
20
30
40
50
Linear
10 to 08
10 to 09
10 to 10
10 to 11