# First Assignment FHPC

Gianluca Guglielmo

October 30, 2020

## 1 Theoretical Model

The first section of the assignment requires to study a parallel algorithm for the sum of N numbers by computing the system's *scalability* and eventually improve it. Scalability is defined as:

$$S(P) = \frac{T(1)}{T(P)} \tag{1}$$

where $T(1)$ is equal to the time needed to execute the workload on one worker and $T(P)$ is the time needed to execute the workload on P workers.

Machine times are fixed:

$$t_{comp} = 2 \times 10^{-9}$$
$$t_{comm} = 10^{-6}$$
$$t_{read} = 10^{-4}$$

Time's unit of measure is seconds. $t_{comp}$ is the *computational* time for a single floating point operation, $t_{comm}$ is the *communication* time needed to move a message between two nodes in the network and $t_{read}$ is the time the master needs to read and store the input in its memory.

The serial time's algorithm for the sum of $N$ numbers is:

```
t_serial = N * t_comp + t_read
```

I've added `t_read` in order to have the serial time equal to the parallel time with `P = 1`. Furthermore, $N$ should be $N - 1$, but the computational time for a single sum is negligible.

The parallel time's algorithm for the sum of $N$ numbers with $P$ CPUs is:

```
t_parallel = (P - 1 + N/P) * t_comp + t_read + 2 * (P - 1) * t_comm
```

The latter is an approximation as well. In order to simplify the execution of the problem we assumed that $N/P$ are the numbers summed by each

processor even if the result of the division is not an integer. To solve this problem and have a more precise estimation of $t_{parallel}$ we could have written an algorithm where the CPUs are filled evenly until the last one, which would have a minor amount of numbers to sum, and $N/P$ could have been replaced by the amount of numbers in the single "full" CPUs. This fix has negligible effects on the $t_{parallel}(P)$ study and it's ignored.

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

I have computed `t_serial` and `t_parallel` using a Python script that implements the algorithms shown in the last section. `t_serial` for `N` is stored in a vector `ser[N]`, while `t_parallel` for `N` and `P` is stored in a 2D numpy array `par[N,P]`.

I've run the algorithm for values of `N` in the interval $[2, 10^{10})$ with steps of $10^4$, and for every value of `P` in $[2, 100]$. I've assumed `par[N,1] = ser[N]`.

I've searched for the first value of `N` for which `par[N,2]<ser[N]`, meaning that the usage of more than one CPU in parallel becomes more profitable then just the serial algorithm on one CPU. I've found this to be around `N_1` = $2\times10^3$.

I've also searched for the first value of `N` for which `par[N,100] < par[N,i]` $\forall$ `i in [1,99]`. This value `N_2` = $9.9\times10^6$ indicates that $\forall$ `N > N_2` the parallel algorithm gives best results with `P` $\geq$ `100`.

This two values `N_1` and `N_2` divide the possible values of `N` in 3 subdomains, which are interesting for the study of the scalability.

In `(0,N_1]` the serial algorithm is faster than the parallel one, thus adding cores translates into slowing down the sum more and more. Scalability `S(P)` is monotonic decreasing (figure 1).

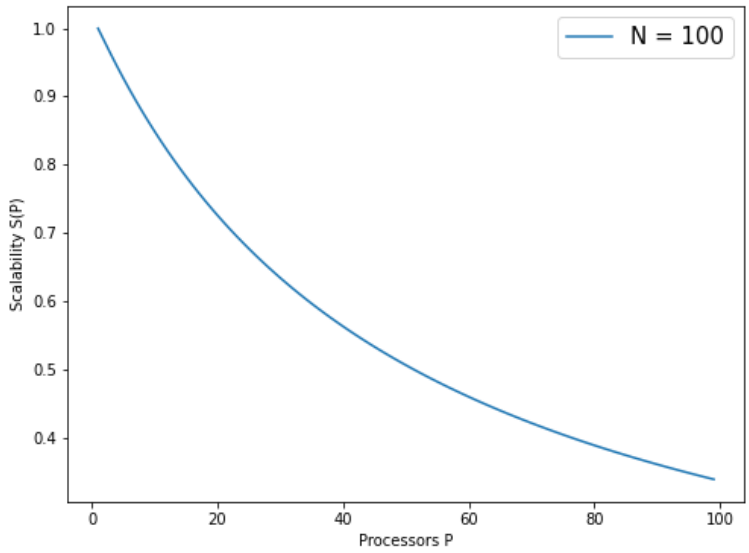



Figure 1: `N = 10^2, max S(P) at P = 1`

In `(N_1,N_2]` some value of P in `[2,99]` gives the best results, thus scalability `S(P)` has a maximum in `[2,99]` (figure 2).

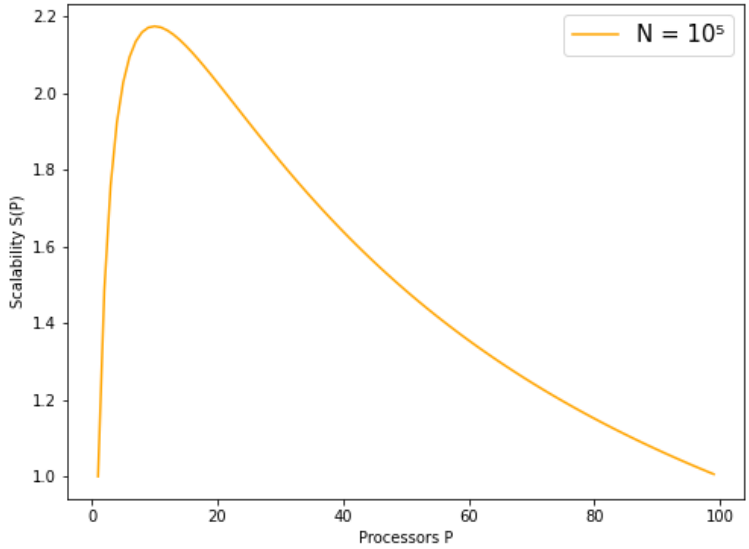



Figure 2: `N = 10^5, max S(P) at P = 10`

In `(N_2, +∞)` the best value of `P` is either `P = 100` or above the maximum number of cores considered. Scalability `S(P)` is monotonic increasing (figure

3).

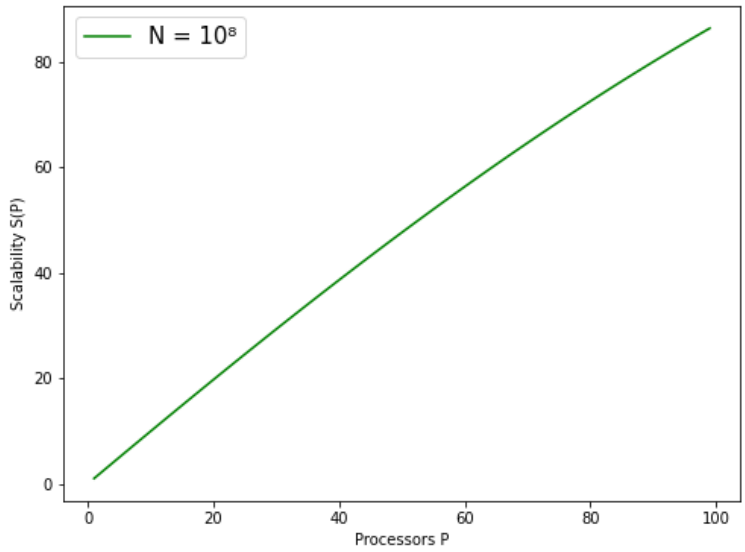
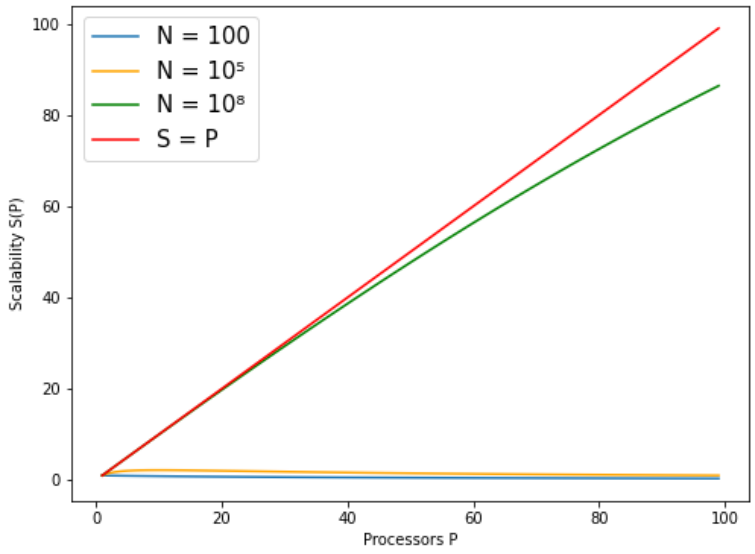



Figure 3: `N = 10^8, max S(P) at P = 100`

All three functions are sublinear (figure 4).




In short, scalability can be seen for `N >` $2{\times}10^3$.

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

The best `P` for each `N` can be found analitically. Setting

$$a = t_{comp} \qquad b = t_{read} \qquad c = t_{comm} \qquad d = N$$

the parallel time as a function of P is

$$t_{parallel}(P) = a(P - 1 + d/P) + b + 2(P - 1)c$$

deriving it we get

$$t'_{parallel}(P) = a - \frac{ad}{P^2} + 2c$$

and setting it equal to 0 we get

$$P(N) = \sqrt{\frac{ad}{2c + a}} = \sqrt{\frac{t_{comp}N}{2t_{comm} + t_{comp}}} = k\sqrt{N}$$

with $k \approx 3.16 \times 10^{-2}$. $P(N)$ must be approximated to the nearest integer.

I've run some tests to see if this formula outputs the correct `P`. To check this I've built a simple Python subscript that, for each `N`, outputs the `P` which gives the smallest `t_parallel`:

```
for n in range (0, 10**6):
  k += 10**4
  ser[n] = serial(k)
  par[n,0] = ser[n]
  scal[n,0] = 1
  minim = ser[n]
  index[n] = 1
  for p in range (1, 100):
    par[n,p] = parallel(k, p + 1)
    scal[n,p] = ser[n] / par[n,p]
    if par[n,p] < minim:
      minim = par[n,p]
      index[n] = p + 1
```

The entry `index[N]` corresponds to the best `P` for `N`$\times 10^4$. The formula appears to work, here are some values of `N`:

```
N = 1000
index[N] = 1, P(N) = 0.999

N = 100000
index[N] = 10, P(N) = 9.99

N = 1000000
index[N] = 32, P(N) = 31.6
```

### 1.3 Can you try to modify the algorithm to increase its scalability?

The parallel time's algorithm can be improved by acting on the communication mechanism since there is a bottleneck that forms when all the slaves send back their partial sums to the master at once. This can be bypassed if we divide the cores into a group that sends its results and a group that receives them. The cores in the second group then add their results to the one received. This process is continued recursively until only the master is left.

If `P = 2^k` with `k` $\in \mathbb{N}$ then the steps needed to go from P cores to the master are

$$h = log_2 P = k$$

and this can be easily seen by thinking of a perfect binary tree with `P` leaves: `h` is the height of the tree, meaning the steps needed to reach the root.

If, on the other hand, `P` $\neq 2^k$ with `k` $\in \mathbb{N}$, then

$$h = k'$$

where $2^{k'}$ is the smallest power of 2 greater than P. To figure out why this works we can think of dividing the cores into two groups: the first group has $2^{k'-1}$ cores, while the second has the rest of them. We can carry out the previously explained process easily in the first group with $h = k' - 1$. The second group has less cores than the first one and can be thought of as a binary tree of the same height with some leaves missing, meaning that it has $h \leq k' - 1$. The second group has to wait for the first to finish its operations, then there is one last message with the last partial sum sent to the master where the final result is calculated.

The goal of this improvement is to act on the communication times since `t_read` is not modifiable and `t_comm` is three orders of magnitude greater than `t_comp`. Still, this fix also improves the computational time since the `P-1` term is also substituted by `k`. The final algorithm is:

```
def parallel_en(n,p):
  for k in range (1,8):
    if p >= 2**(k-1) and p <= 2**k:
      break
  t_parallel = t_comp * (k + n/p) + t_read + (p - 1 + k) * t_comm
  return t_parallel
```

# 2 MPI Program

## 2.1 Strong Scaling

The second section of the assignment requires to run a program for the estimation of $\pi$ using the Montecarlo technique, where the user passes a number of steps as input. Two versions of the program are given, with the first one being a serial version (`pi.x`), while the second one implementing an MPI protocol (`mpi_pi.x`).

Running the two versions on a single core with `N_iter` = $10^8$ we observe a run time of:

```
t_pi = 2.64
t_mpi_pi = 2.86
```

The serial version clearly outperforms the parallel one because of the MPI overhead, that is the time needed for communication and synchronization by the parallel algorithm. The `elapsed \usr\bin\time` was used in order to compare the actual run times of the programs.

We then ran a strong scaling test with `mpi_pi.x` using `1, 4, 8, ..., 48` processors and recorded both the elapsed `\usr\bin\time` and the `walltime` for the master processor. Figure 4 shows a comparison of the two run times. The same tasks were then repeated for `N_iter` = $10^9$, $10^{10}$, $10^{11}$. The subsequent figures show the system's scalability as defined earlier.

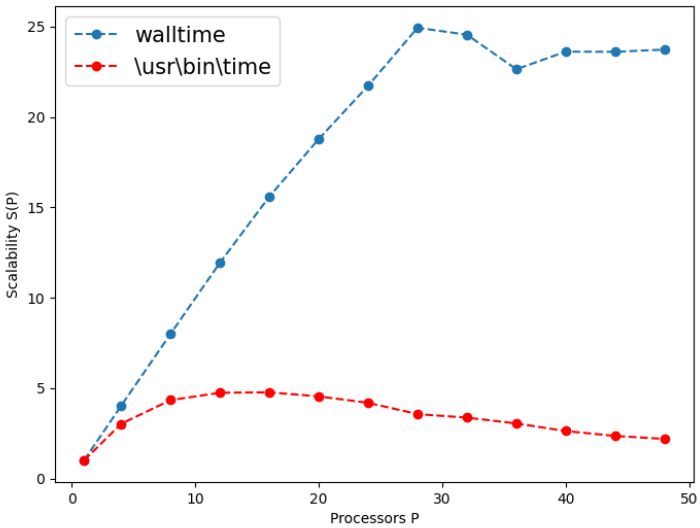



Figure 4: `N = \usr\bin\time vs walltime, N_iter` = $10^8$

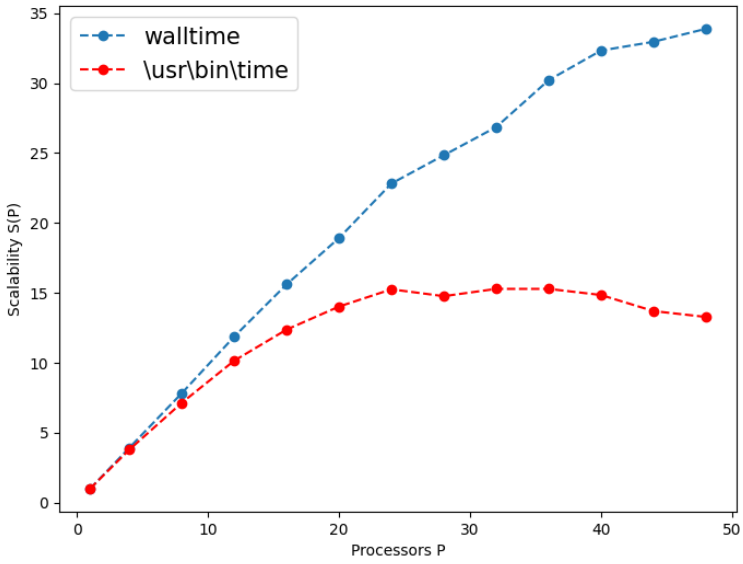



Figure 5: `N = \usr\bin\time vs walltime, N_iter =` $10^9$

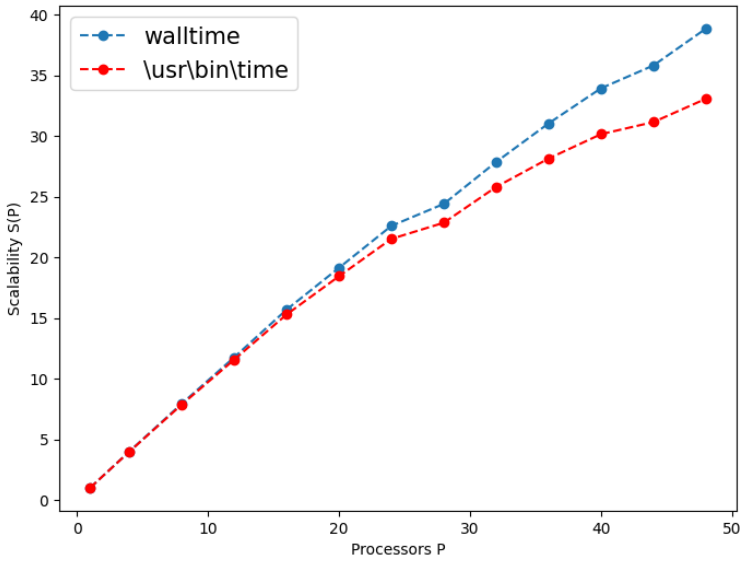



Figure 6: `N = \usr\bin\time vs walltime, N_iter =` $10^{10}$

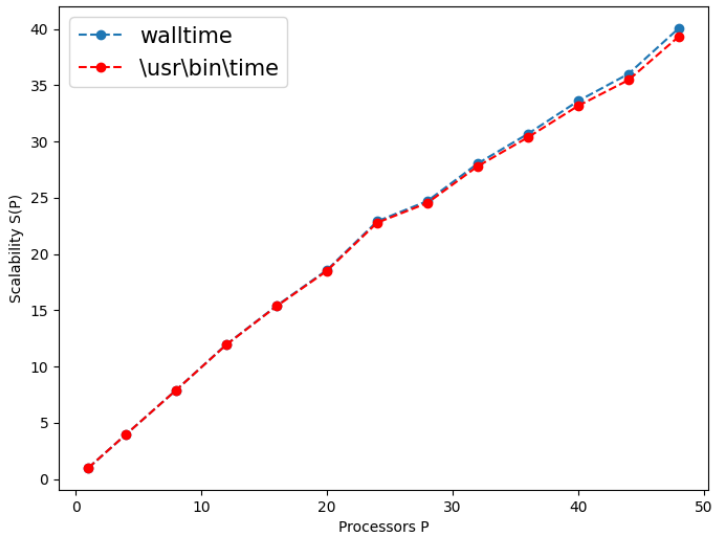



Figure 7: `N = \usr\bin\time vs walltime, N_iter =` $10^{11}$

What the previous figures show is the difference in scalability between the performances recorded by the master processor's `walltime` and the `\usr\bin\time`. The thinning relative difference between the two lines as `N_iter` increases shows that more moves make the `system time` associated to the `mpirun` progressively more negligible and bring a fuller CPU usage which implies better scalability.

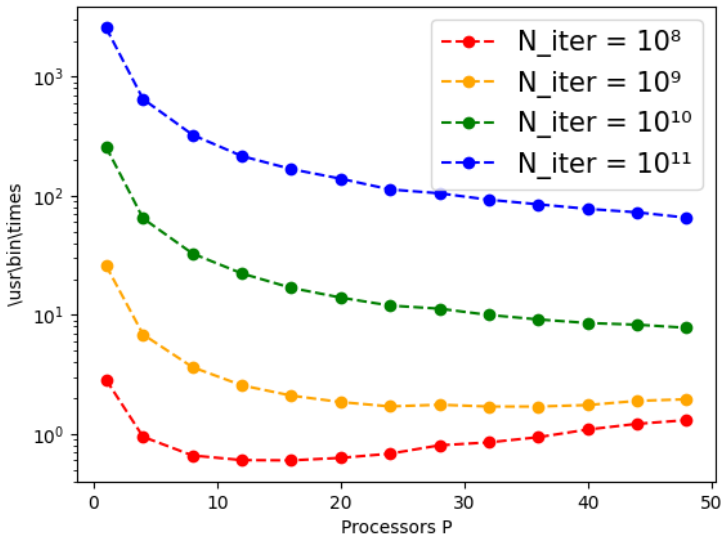



Figure 8: `N = \usr\bin\time of three N_iter`

Figure 8 shows a comprehensive plot of the four runs of the program with `\usr\bin\time` as a function of `P`. The log scale was chosen since without it the run with `N_iter` = $10^{11}$ couldn't be plotted due to its huge initial times that made it difficult to see the other plots. A great time reduction can be seen, showing the benefits of the parallelization.

### 2.2 Parallel Overhead

In order to find a suitable model for the parallel overhead I used the simple formula:

```
overhead = t_parallel * p - t_serial
```

Elaborating the previous formula for every `P` in each run of the program we get the graph in Figure 9.

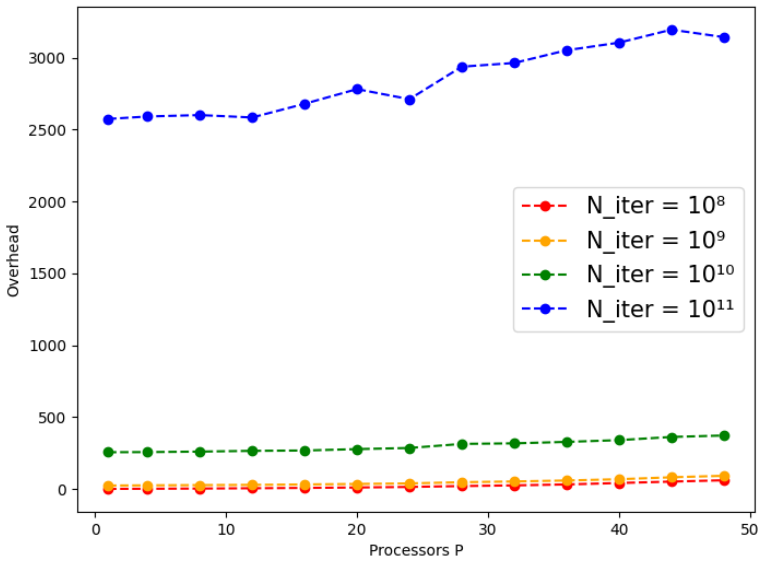



Figure 9: `Overhead for the four N_iter`

The best fit for each of the graphs is a polynomial fit of `degree = 2`

```
f(x) = A * x**2 + B * x + C
```

with coefficients

| N_iter | A | B | C |
|---|---|---|---|
| $10^8$ | 0.0286 | -0.1598 | 1.4468 |
| $10^9$ | 0.0321 | -0.1736 | 24.8573 |
| $10^{10}$ | 0.0405 | 0.6224 | 252.0428 |
| $10^{11}$ | 0.1127 | 9.2055 | 2527.1202 |

As it can be seen visually (and by looking at the table), the best model is practically the same for each `N_iter` and the only remarkable difference seems to be in the coefficient C. Taking into consideration that every run has 10 times more steps than the previous one, the almost ten-fold increase in C for each case seems to justify the previous assertion on the dependence of the model on `N_iter`.

### 2.3 Weak Scaling

The last part of the assignment regards weak scaling. The size of the problem was increased with proportionality to the number of processors used. We expect the time to be constant as `P` and the problem size increase.

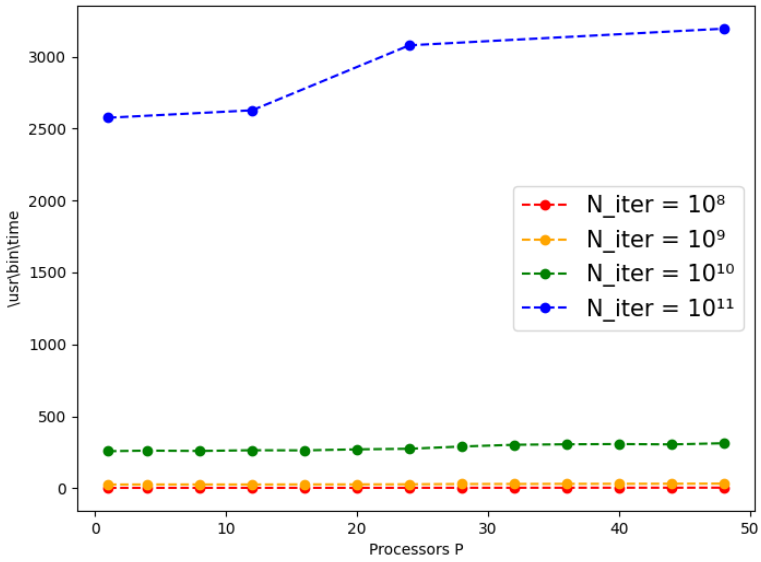



Figure 10: `N = \usr\bin\time of four N_iter`

The size of the problems for the runs in Figure 10 is

```
size = N_iter * p
```

For small `N_iter` the system can approximately achieve a constant time throughout the executions, while as the number of executions per core increases we start to see some important overhead, as demonstrated by `N_iter` = $10^{11}$.

Defining a weak efficiency as $T(1)/T(p)$, we are also interested in plotting the weak scalability for the different number of iterations, which can be seen in Figure 11.

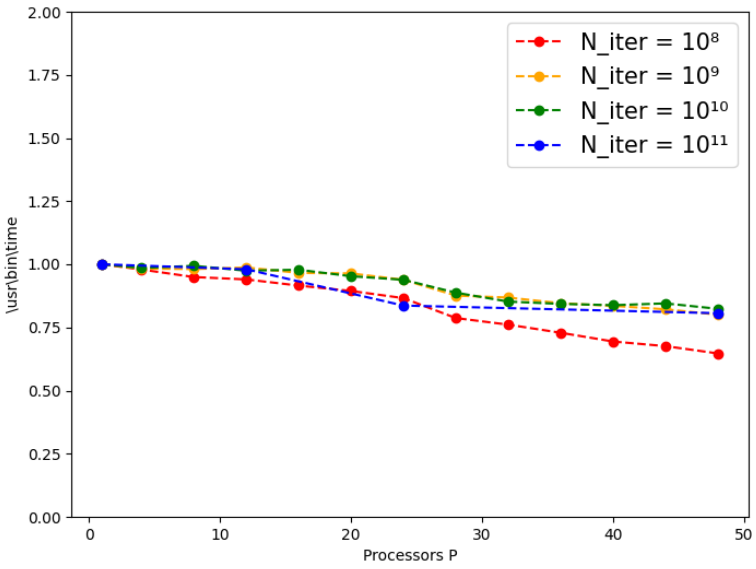



Figure 11: `Weak Scalability`

As expected, the scalability stays almost constant for all the runs, and as the number of processors increases we start to see a decrease given by the parallel overhead.