Assignment #2

Michał Przybylski, Francesco Ragaini

February 2025

Task 1

In order to build an algorithm that is working for the master-worker problem we need to figure out the two functions worker and master. They are builted in that way

Algorithm 1 Master Algorithm version 0

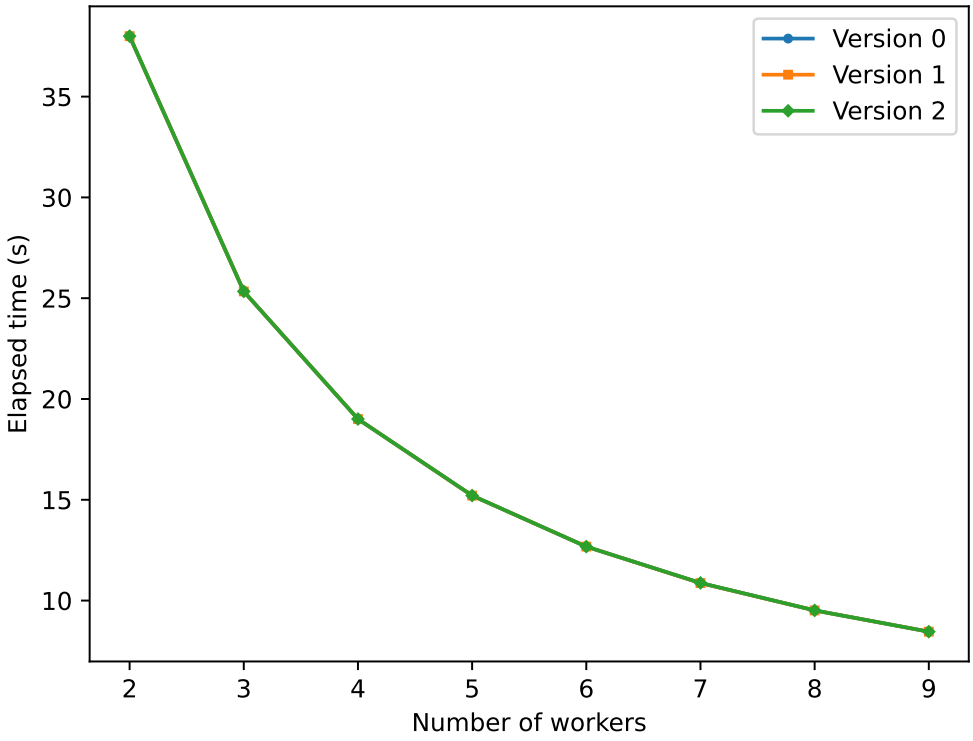
```
1: Initialize all the needed variables: sent ← 0, done ← 0, workdone ← 0
2: for each worker do
3:     Send a task to the worker
4:     sent ← sent + 1
5: end for
6: while sent < total_task do
7:     receive a task done (blocking)
8:     send a new task (task_to_do[sent]) to the same worker (blocking)
9:     result[done] ← worker
10:    done++
11:    sent++
12: end while
13: for each worker do
14:    receive a task done (blocking)
15:    sent the sleeping message to the same worker (blocking)
16:    result[done] ← worker
17:    done++
18: end for
```

Algorithm 2 Worker Algorithm version 0

```
1: while true do
2:     receive a task from the master (blocking)
3:     if task is sleeping message then
4:         break
5:     end if
6:     perform the task
7:     send the result to the master (blocking)
8: end while
```

After this algorithm, the master is able to perform statistics on the time needed for the program. After that, we tried to improve the performance of the program, leading to version 1, which introduces some non-blocking commands. In particular, the first sending is non-blocking, the send inside the while loop is also non-blocking (the check for it is after the operations in the while loop), and the send inside the last for loop is non-blocking (and they are checked at the end of the master function).
We now tried to improve the algorithm once more, and the changes are: the first task is sent before the others (the idea is to make the first worker finish its tasks at different times than the others in order to reduce the queue at the master's door). The send inside the while loop is not checked (in fact, this program is really fast and reliable, so it is not really necessary to check all the sends). It is not possible to make the receive inside the while loop non-blocking because we need the master to wait for the workers to complete the tasks.
(IT MIGHT BE A GOOD IDEA TO SHOW THE UPDATED ALGORITHM HERE, BUT I DON'T THINK THERE'S ENOUGH SPACE).
We can now check the elapsed times to see if our improvements are making the code faster, this is done in figure **??**

As is possible to see our changing are not really improving the times, but is also possible that those changing are significant just for high number of workers (in fact all the changing are ment to reduce the time that the worker need to wait at the master door and since they are just a few this time is already small.) but since this first task was just ment to set a benchmark we can bring this question with us for the next one.



Task 2

We can finally mouve to the second question that is really more interesting in term of physical applications, in fact here the tasks are ment to be usefull. We can now expose the algortihm for both master and worker that are similar to the version 0 of the first question.

Algorithm 3 Master Algorithm version 0

```
1: Initialize all the needed variables: done ← 0, sent ← nworker
2: settings_sorted ← vector of size n_settings
3: for each worker i from 1 to nworker do
4:     Send settings[i-1] to worker i
5: end for
6: while sent < n_settings do
7:     Receive worker from any source
8:     Receive accu from worker
9:     Receive momentaneus_setting from worker
10:    Send settings[sent] to worker
11:    accuracy[done] ← accu
12:    settings_sorted[done] ← momentaneus_setting
13:    done++
14:    sent++
15: end while
16: for each worker do
17:    Receive worker from any source
18:    Receive accu from worker
19:    Receive momentaneus_setting from worker
20:    settings_sorted[done] ← momentaneus_setting
21:    accuracy[done] ← accu
22:    Send rest_signal to worker
23:    done++
24: end for
```

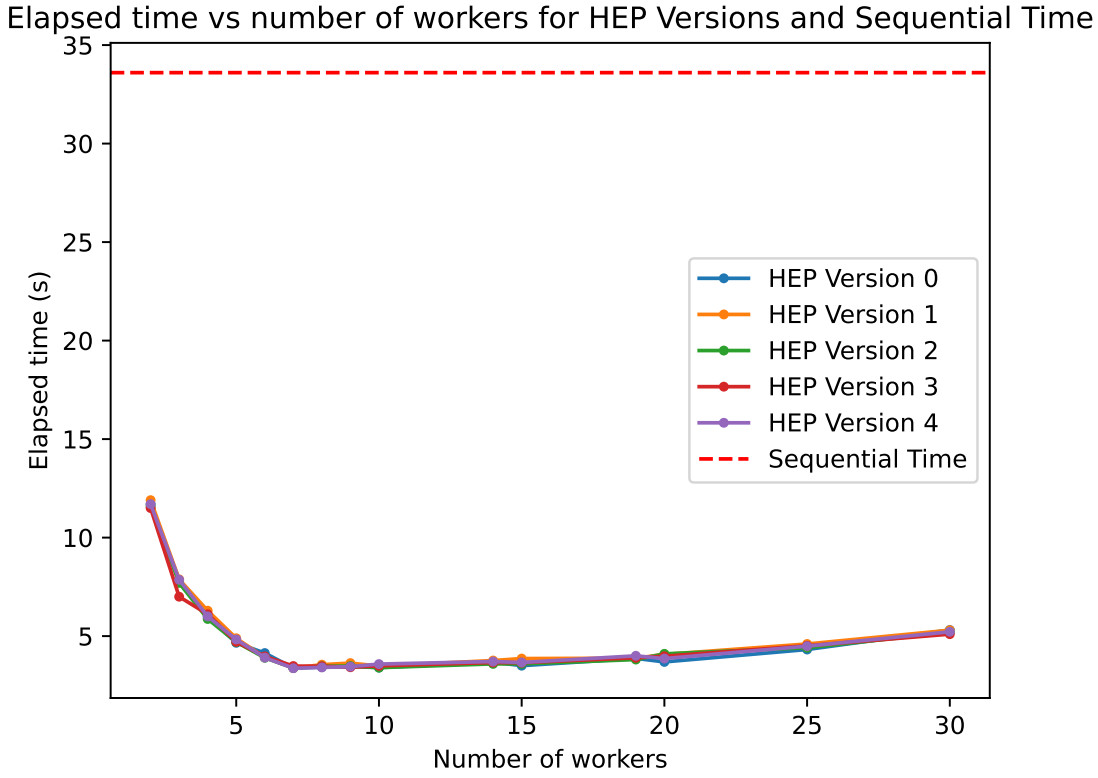
Algorithm 4 Worker Algorithm version 0

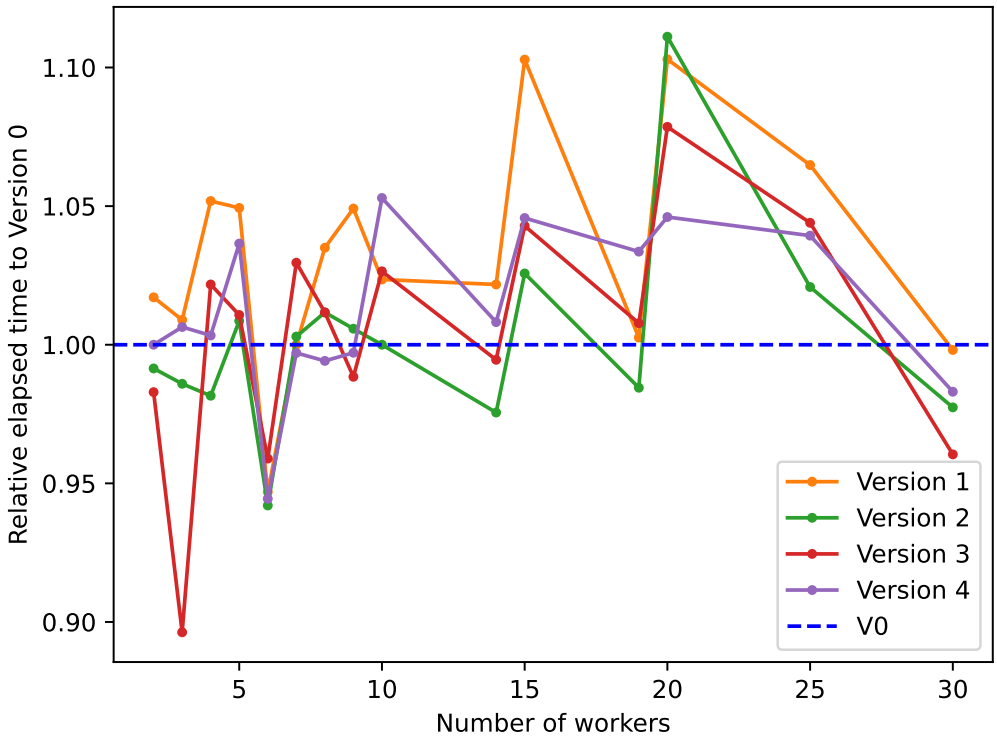
```
1: while true do
2:     receive setting from the master (blocking)
3:     if setting[0] is -999 then
4:         break
5:     end if
6:     perform the task and compute acc
7:     send rank to the master (blocking)
8:     send acc to the master (blocking)
9:     send setting to the master (blocking)
10: end while
```

In this case as been necessary to add 3 send condition for the worker, in fact it needs to comunicate the informations about his ranking, the results and the settings that is using (this last part is really important, in fact in order to find the best settings is crucial to know which resolution is connected to which settings).
Is now time to try to improuve the code, the first two versions (version 1 and version 2) are made following the same ideas as before. The fact that we need to use 3 sent seems to be the main problem of the code, so we try to condens all the informations, this is done by passing one vector (called result with len = 10, 8 for the settings, 1 for the rank and 1 for the result), in this way the number of comunication is reduced and the master can work on it, to order the result, after it sent the new task to the worker that in this way is not loosing time. After a small conversionation with the TA we find out a better way to do it, in fact is possible to use MPI_Status, ranking and tag to pass all the information in one command, the code inside the while is now so

```
while sent < n_settings do
    Receive result from any source (blocking)
    worker ← result[0]
    Send settings[sent] to worker (non-blocking)
    settings_sorted[done] ← result[2:]
    accuracy[done] ← result[1]
    done++
    sent++
end while
```

(ALSO IN THIS CAS WE CAN THING TO SHOW THE ALL ALGORITHM) We can now see the difference between the different versions of the code for the number of workers (**??**) and the sequential code To see better the behavior of it we can also look at the ratio between the version and the first version (see Figure **??**). It is also important to underline that all the given results


Elapsed time vs number of workers for HEP Versions and Sequential Time
Elapsed time (s)
Number of workers
HEP Version 0
HEP Version 1
HEP Version 2
HEP Version 3
HEP Version 4
Sequential Time


Relative elapsed time to Version 0
Number of workers
Version 1
Version 2
Version 3
Version 4
V0

are checked to ensure that the final result (the best settings and its accuracy) is the same as the sequential one.
As is really clear from the plots, the different results are not reporting such a big difference, but we are sure that the last one is better for a higher number of workers, its behavior will be shown in Task 3. It is also important to notice that for some reason the time needed for a higher number of workers is increasing, this might be because the communication between the nodes is not as effective as we expected it to be. This will also be explored in Task 3. On the other hand the parallel code is working really faster that the sequential code.