Assignment I

Foundations of High Performance Computing

November 6, 2019

Eros Fabrici

0 Warm up

- Compute Theoretical Peak performance for your laptop

	Model	CPU	Frequency	Number of cores	Peak Performance
Laptop	Acer Aspire E 15	Intel Core i5-4210U	1.7Ghz	4	108,8 GFlop/s

Table 1: Acer Aspire E 15

- Compute sustained Peak performance for your cell-phone

	Model	Sustained performance	Size of the matrix	Peak Performance	Memory
Smartphone	Xiaomi MI 8 Lite	1090.83 MFlop/s	2000	32 GFlop/s	6 GB

Table 2: Xiaomi Mi 8 Lite

The CPU is Snapdragon 660 Qualcomm SDM660 and its frequency is 1.8Ghz for 4 cores, 2.2Ghz for the other 4. For the sake of simplicity, the frequency used in the formula was the average, namely 2Ghz. It is capable of executing 2 floating point operations per cycle.

- Find out in which year your cell phone/laptop could have been in top1 of Top500.

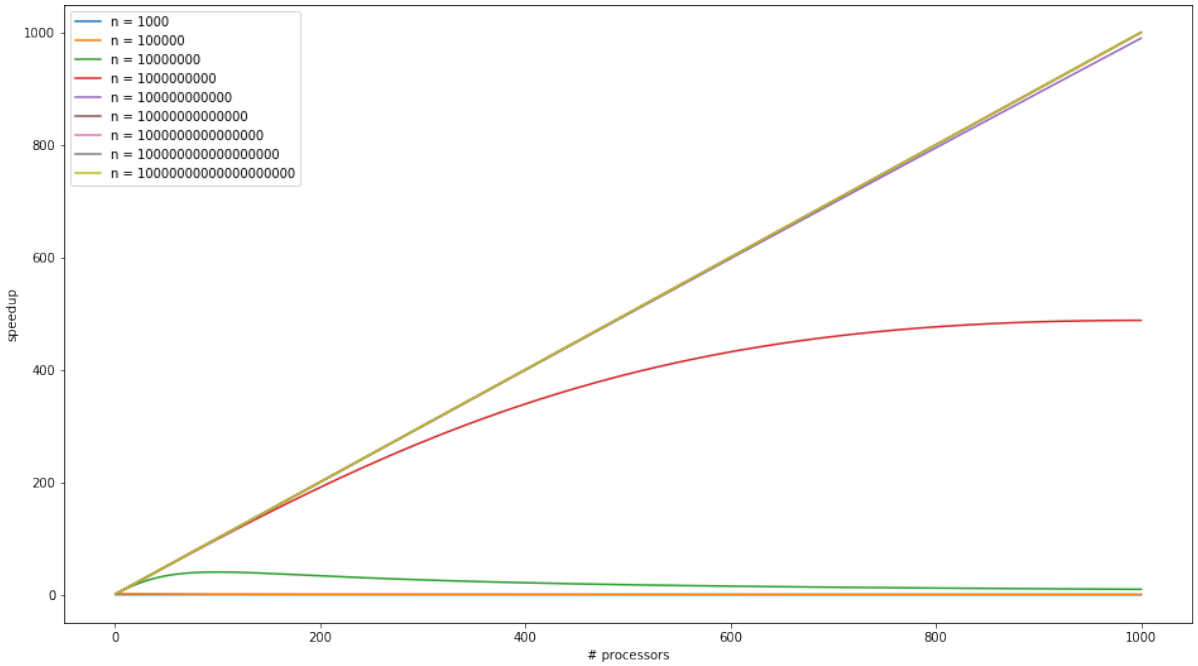
  – The oldest list is dated June 1993 and none of my devices would have been first in the past according to those lists (the first in 06/1993 has 131Gflop/s of theoretical peak performance), or at least not after June 1993.

1 Theoretical Model

Following the model described during the lectures, below the graph and the code for generating it.

```
import numpy as np
import matplotlib.pyplot as plt
T_comp = 2*10**(-9)
T_read = 10**(-4)
T_comm = 10**(-6)

def compute_Tp(n, p):
    """
    n: input size
    p: number of processors
    """
    return T_comp * ((p-1)+(n/p)) + T_read + 2*(p-1)*T_comm

def compute_T1(n):
    """
    n: input size
    """
    return n*T_comp


N = [10**e for e in range(3,21,2)]
p = np.linspace(1,1000,1000)
```

```
23
24 for n in N:
25     speedup = (compute_T1(n)/compute_Tp(n, p))
26     plt.plot(p, speedup, label="n = " + str(n))
27     plt.legend()
28     plt.xlabel("# processors")
29     plt.ylabel("speedup")
30     plt.rcParams["figure.figsize"] = [16,9]
```



- For which values of N do you see the algorithm scaling?
  - The graph shows clearly that, for small values of n, the algorithm does not scale, while, the more we increase n, the more the algorithm scales. In my example, it is possible to notice that best result is observed for 10^11 on wards.
- For which values of P does the algorithm produce the best results?
  - It depends. For large values of n, the best results are obtained with high values of P e.g. 1000. While for low values of n, for instance $n = 10^7$, the best result is obtained when $P \approx 80$, then it decreases.
- Can you try to modify the algorithm sketched above to increase its scalability?
  - The idea in based on the model of a Binary Search Tree, where the nodes are the processes. The master will keep its N/P part and sends the remaining two halves to two children respectively and computes its partial sum and waits for the partial sums from its two children. The two children do the same thing recursively. By using this technique, at each level of the tree the communications are done in parallel, namely a series of 2 communications is done at the same time, therefore the communication cost is the same as searching in BST, namely $O(log_2(n))$ with n the number of the nodes in the tree, but in our case it will be $4log(p-1)T_comm$ Following the final model.

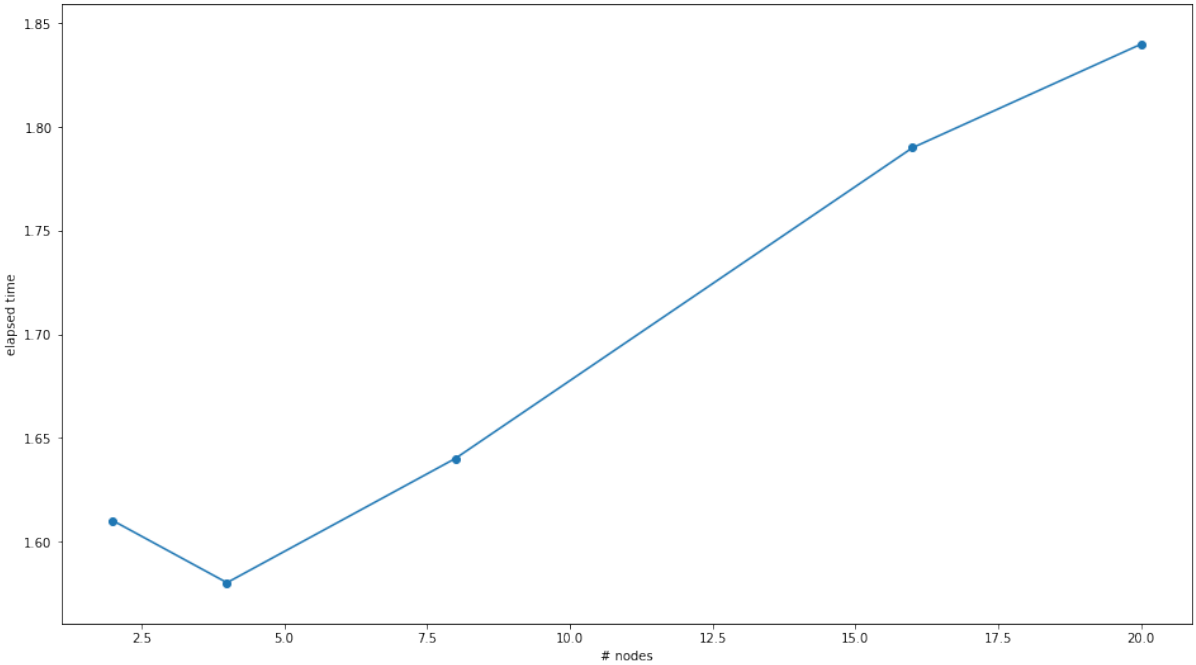
$$T(N,P) = T_{comp}(P - 1 + N/P) + T_{read} + 4log_2(P-1)T_{comm}$$

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 1000000 (10 millions) iterations (stone throws). Make sure that this is the actual run time and does not include any system time.
  - The time for executing `pi.c` with $n = 1000000$ is 0.02 (elapsed time).
- Get the MPI code running for the same number of iterations. The parallel code writes walltime for all the processor involved. Which of these times do you want to consider to estimate the parallel time?
  - It should be cosidered the highest time, however, using `/usr/bin/time` is more precise, as it computes the elapsed time from invocation till termination. Therefore I will consider usr/bin/time output.
- First let us do some running that constitutes a strong scaling test. This means keeping the problem size constant, or in other words, keeping Niter = 10 millions. Start by running the MPI code with only one processor doing the numerical computation. A comparison of this to the serial calculation gives you some idea of the overhead associated with MPI. Again what time do you consider here?
  - The best choice is always to consider the elapsed time returned by the command `usr/bin/time` which shows clearly the difference between executing the serial program (e. time $0.19s$) and the mpi program with one process (e. time $1.70s$).
- Keeping Niter = 10 millions, run the MPI code for 2, 4, 8 and 16 and 20 MPI processs. Make a plot of run time versus number of nodes from the data you have collected.

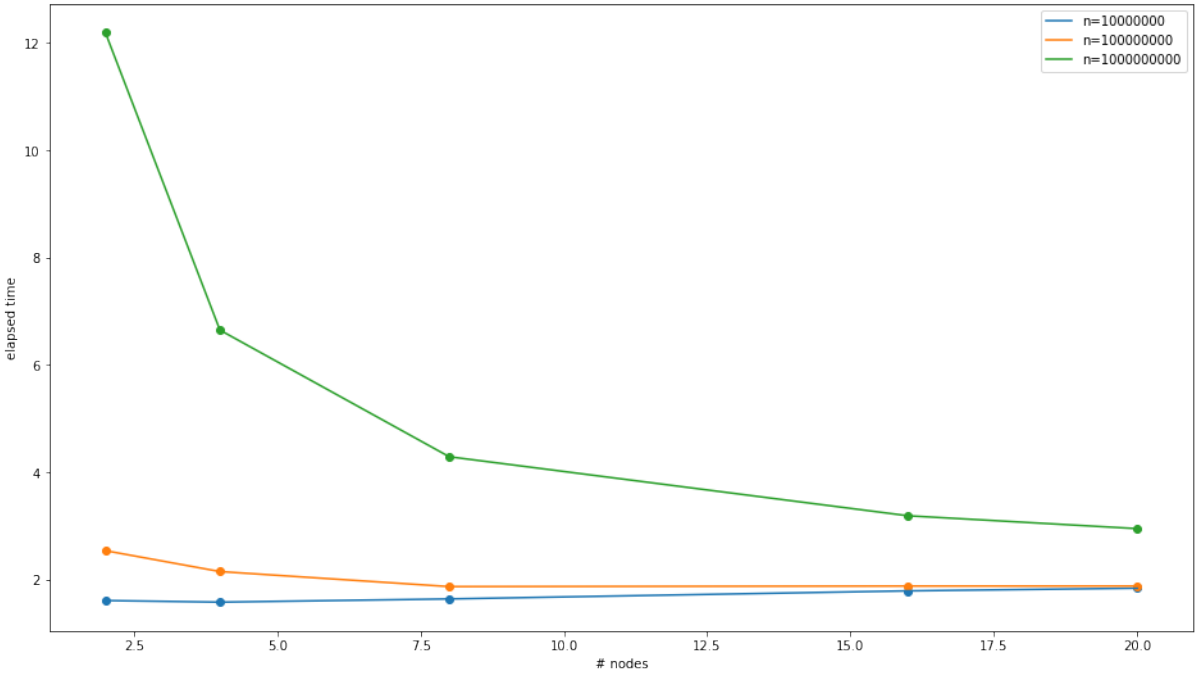
```
# niter= 10000000
# (#_of_processes, elapsed_time)
result = [(2,1.61), (4,1.58), (8,1.64), (16,1.79), (20,1.84)]
plt.scatter(*zip(*result))
plt.xlabel("# nodes")
plt.ylabel("elapsed time")
plt.plot(*zip(*result))
```



  - The image above shows clearly that with $n = 10^7$ the elapsed time benefits only when we increase the processes from 2 to 4, for the other values it increases.

- Provide a final plot with at least 3 different size and for each of the size report and comment your final results.

```
import numpy as np
import matplotlib.pyplot as plt
#input sizes tested (Niter size)
#10000000
#100000000
#1000000000

results = [[(2,1.61), (4,1.58), (8,1.64), (16,1.79), (20,1.84)],
           [(2,2.54), (4,2.15), (8,1.87), (16,1.88), (20,1.88)],
           [(2,12.20), (4,6.65), (8,4.29), (16,3.19), (20,2.95)]]

input_size = 10000000
i = 1
for n in results:
    plt.scatter(*zip(*n))
    plt.plot(*zip(*n), label="n=" + str(input_size*i))
    plt.legend()
    plt.xlabel("# nodes")
    plt.ylabel("elapsed time")
    plt.rcParams["figure.figsize"] = [16,9]
    i = i * 10
```



– This graph observes what already seen in the section 0: the algorithm scales good with high values of n, while with small values the is no throughput in increasing the number of processes.

2.2 Identify a Model for the Parallel Overhead

The model was derived according to the data used for plotting the first graph in section 2.1. The idea is that the parallel overhead is influenced by the time to execute a serial part, a time for spawning processes and the time for executing the parallel part.
Let S the time for executing the serial part, p the number of processes, C_{spawn} the cost of spawning a new process plus the cost of message passing operations and P the time to execute the parallel part in serial, we have

$$T(p) = S + p * C_{spawn} + P/p$$

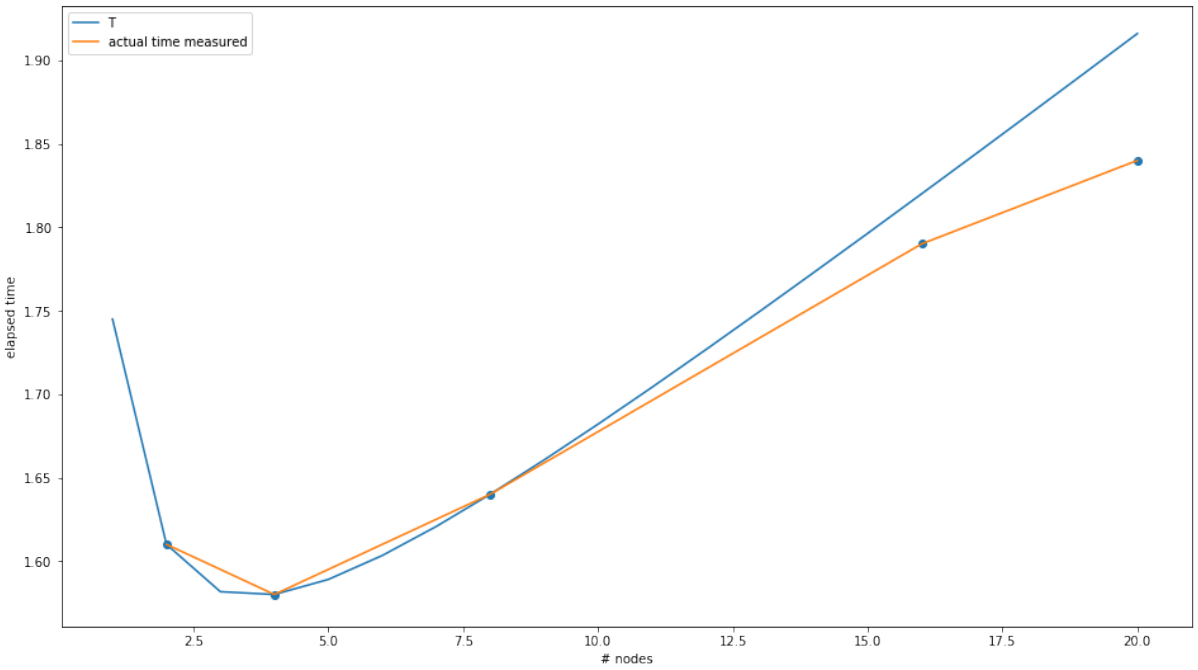
By solving the following system

$$\begin{cases} 1.61 = S + 2 * C_{spawn} + P/2 \\ 1.58 = S + 4 * C_{spawn} + P/4 \\ 1.64 = S + 8 * C_{spawn} + P/8 \end{cases}$$

where 1.61, 1.58 and 1.64 are the times recorded that are displayed in the first graph in section number 2.1, we obtain that $S = 7/5$, $C_{spawn} = 1/40$ and $P = 8/25$. Following the plot showing the results compared to the plot aforementioned.

```
S=7/5
C_spawn=1/40
P = 8/25
def T(numprocs):
    return S + C_spawn*numprocs + P/numprocs
p = np.linspace(1,20,20)
plt.plot(p, T(p), label="T")
result = [(2,1.61), (4,1.58), (8,1.64), (16,1.79), (20,1.84)]
plt.scatter(*zip(*result))
plt.xlabel("# nodes")
```

```
plt.ylabel("elapsed time")
plt.plot(*zip(*result), label="actual time measured")
plt.legend()
plt.rcParams["figure.figsize"] = [16,9]
```



The graph above shows that T following the same curve plotted before, which is the time elapsed for the entire execution. Therefore the model for the parallel overhead can be represented by $p * C_{spawn}$ which grows with the increase of p.

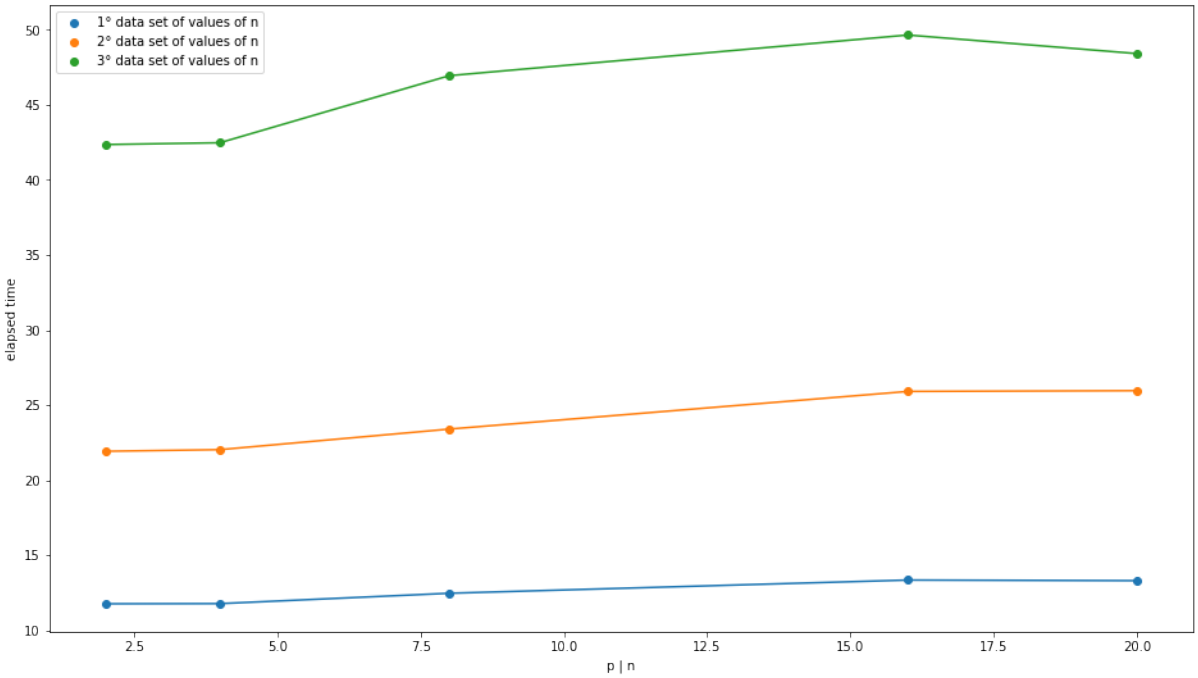
2.3 Weak Scaling

- 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 and the number of compute nodes increase in proportion. Modify your scripts to rapidly collect numbers for the weak scalability tests for different number of moves.

```
n=(1000000000 2000000000 4000000000 8000000000 10000000000)

for j in 1 2 4 ; do
   i=0
   for procs in 2 4 8 16 20 ; do
      /usr/bin/time mpirun -np ${procs} mpi_pi.x $((${n[$i]} * $j))
      let "i = i + 1"
   done
done
```

```
#weak scalability - elapsed time
# n varies in three ranges which are
# 1000000000, 2000000000, 4000000000, 8000000000, 10000000000
# 2000000000, 4000000000, 8000000000, 16000000000, 20000000000
```

```
# 4000000000, 8000000000, 16000000000, 32000000000, 40000000000
# p varies in 2, 4, 8, 16, 20

T_p = [[(2,11.76), (4,11.77),(8,12.46),(16,13.34),(20,13.30)],
       [(2,21.92),(4,22.03),(8,23.40),(16,25.91),(20,25.96)],
       [(2,42.36),(4,42.48),(8,46.95),(16,49.66),(20,48.42)]]

for i in range(0,3):
    plt.scatter(*zip(*T_p[i]), label=str(i+1) + "° data set of values of n")
    plt.legend()
    plt.xlabel("p | n")
    plt.ylabel("elapsed time")
    plt.plot(*zip(*T_p[i]))
```



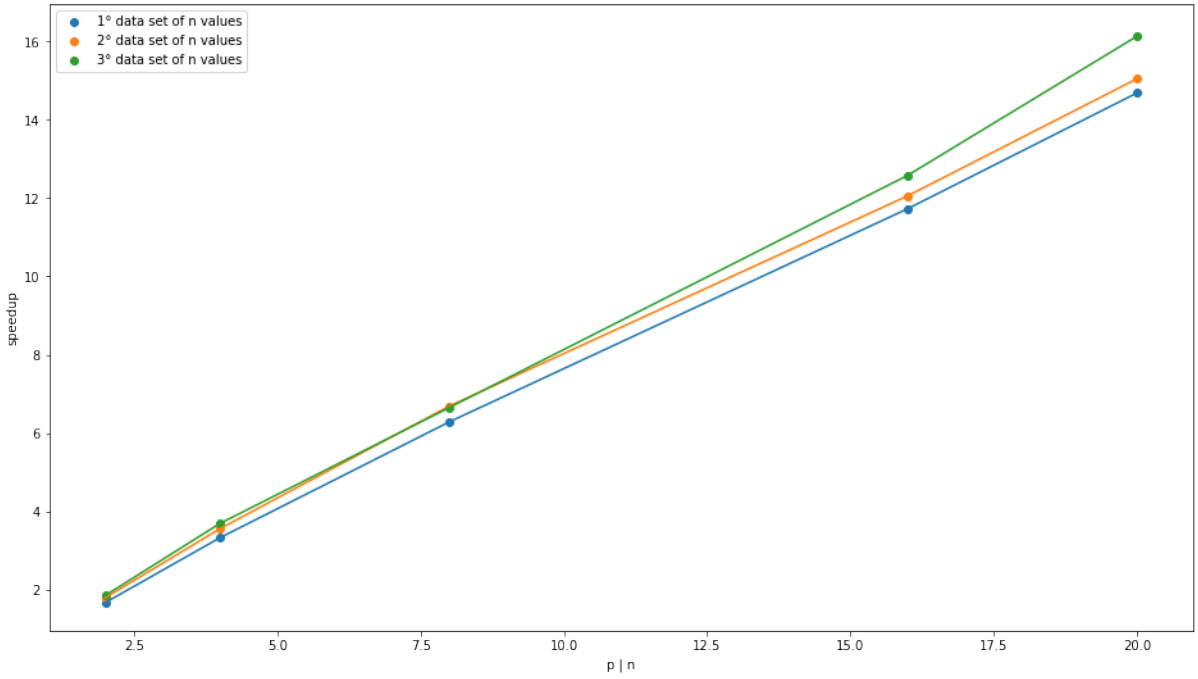
  - According to the above plot, it is possible to deduce that the elapsed time remains constant, even for other data sets with higher values of n.

- Plot on the same graph the efficiency (T(1)/T(p)) of weak scalability for different number of moves and comment the results obtained. **NOTE: it is asked to plot on the same graph used for plotting the elapsed time versus number of nodes, but I preferred to plot the results in a different one as in the following regards the speedup, not the elapsed time.**

```
#Bash script for running serial program on the three data
#set of values of $n$; for the parallel one, I used the
#values extracted from the scripted presented in previous
#question
for i in 1 2 4 ; do
   for j in 0 1 2 3 4 ; do
      /usr/bin/time ./pi.x $((${n[j]}* $i))
```

```
    done
done
```

```
##weak scalability - speedup
# n varies in three ranges which are
# 1000000000, 2000000000, 4000000000, 8000000000, 10000000000
# 2000000000, 4000000000, 8000000000, 16000000000, 20000000000
# 4000000000, 8000000000, 16000000000, 32000000000, 40000000000
#
# note that n increases with the same increase rate of p

#times measured with n and p increasing
T_p = [[11.76, 11.77,12.46,13.34,13.30],
       [21.92,22.03,23.40,25.91,25.96],
       [42.36,42.48,46.95,49.66,48.42]]

#times measured with the serial program with n increasing
T_1 = [[19.54,39.07,78.26,156.52,195.42],
       [39.13,78.20,156.41,312.53,390.87],
       [78.13,156.52,312.10,625.07,781.68]]

speedup = []
res_ws = []
for j in range(0,3):
    for i in range(0,5):
        speedup.append(T_1[j][i]/T_p[j][i])
        if i != 4:
            res_ws.append((2**(i+1),speedup[i]))
        else:
            res_ws.append((20,speedup[i]))
    plt.scatter(*zip(*res_ws), label=str(j+1) +"° data set of values of n")
    plt.legend()
    plt.xlabel("p | n")
    plt.ylabel("speedup")
    plt.plot(*zip(*res_ws))
    speedup = []
    res_ws = []
```



– The speedup behaves as expected according to the weak scaling definition: by increasing p and n with the same rate, the speedup increases linearly. As written in the comments above, $p \in \{2, 4, 8, 16, 20\}$ and I used three data set of values of n, which grows with the same rate of p.

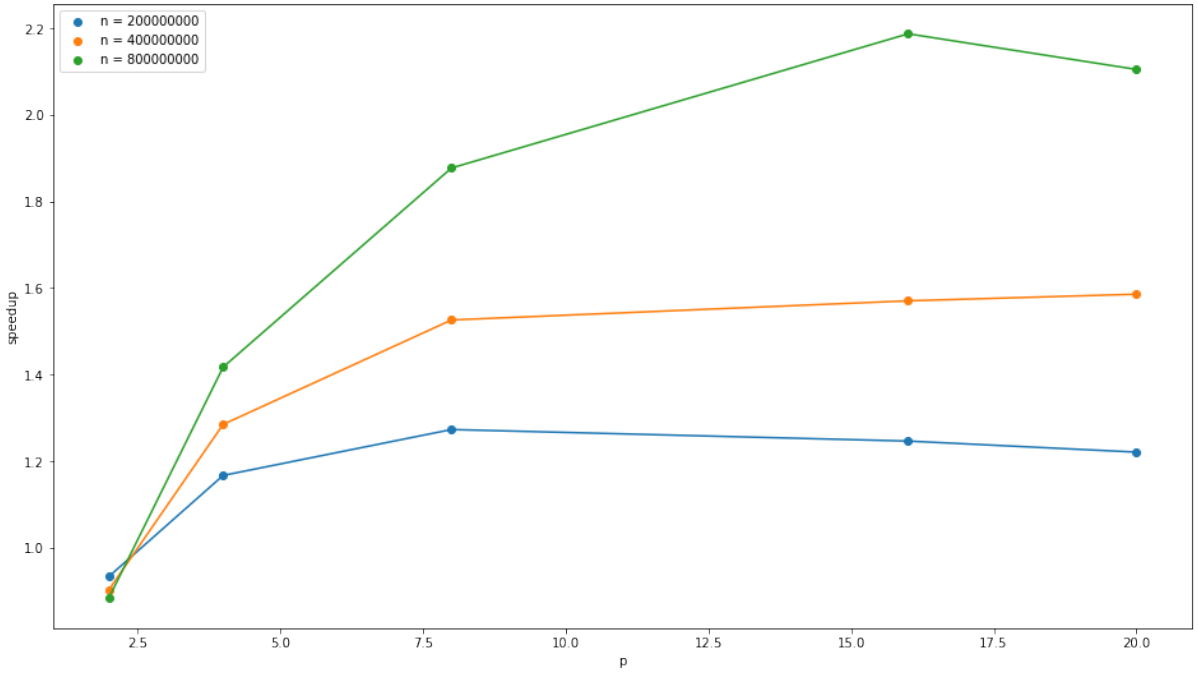
3 Implement a Parallel Program Using MPI

The parallel program is written in C and, for sake of simplicity, it is based on the naive model.

4 Run and Compare

```
#n is
#200'000'000
#400'000'000
#800'000'000
T_p = [[2.55, 2.04, 1.87, 1.91, 1.95],
       [3.60, 2.53, 2.13, 2.07, 2.05],
       [5.69, 3.55, 2.68, 2.30, 2.39]
      ]

T_1 = [2.38, 3.25, 5.03]
res = []
speedup = []
for i in range(0,3):
    for j in range(0,5):
        speedup.append(T_1[i]/T_p[i][j])
        if j != 4:
            res.append((2**(j+1),speedup[j]))
        else:
```

```
            res.append((20, speedup[j]))
    plt.scatter(*zip(*res), label="n = " + str(200000000*(2**i)))
    plt.legend()
    plt.xlabel("p")
    plt.ylabel("speedup")
    plt.plot(*zip(*res))
    speedup = []
    res = []
```



It is possible to observe that the program does not scale good for the first two values of n, while it scales good up to 16 cores for the biggest n.

- Comment on the assumption made in section 1 about times: are they correct? do you observe something different?
  - The times assumed does not really correspond to the ones registered during the execution. To be more precise, T_comp is the highest according to the model, while in practice it is the lowest. The highest seems to be the communication time, while reading time is the second.