UNIVERSITÀ DEGLI STUDI DI TRIESTE

First FHPC Assignment

Nicola Domenis

December 13, 2019

1 PREVIEW

In this assignment we will present the following subjects:

- the analysis of the computational power of our laptop and smartphone;
- the analysis of a strong scaling model for a simple addition problem;
- the resolution of a scalability problem for a simple parallel code that computes pi;
- the parallel implementation of the above addition model;
- the scalability of the above addition program.

2 SECTION 0

2.1 Laptop theoretical peak performance

We want to calculate the theoretical peak performance of our own portable computer by using the formula **theoretical peak performance = clock frequency x FLOPs x number of cores**. We gather that **clock frequency** = 2.90Ghz, **FLOPs** = 16 and *number of cores* = 2 for our computer architecture, an intel i7 with a Kaby Lake micro-architecture; thus we compute **theoretical peak performance** = 92.8$GFlops/s$

	Your model	CPU	Frequency	Number of Cores	Peak Performance
laptop	Asus F556U	Intel Core i7-7500	2.90 GHz	2	92.8 GFLOPs/s

2.2 Smartphone theoretical peak performance

We installed "'Mobile Linpack"' app and we run a few tests. We report here some results,even on repeated trials:

	Model	Sustained performance	Matrix size	Peak performance	Memory
Cellphone	Samsung Galaxy XCover 4	114,81 Mflops/s	250	not calculated(we didn't find the FLOPs of the cpu, which is a quad-core processor with a 1.4 Ghz clock frequency)	16,00 GB
		145.53 Mflop/s	500		
		157.5 Mflop/s	800		
		201.32 Mflop/s	800		
		155.93 Mflop/s	900		
		109.88 Mflop/s	1000		
		103.14 Mflop/s	2000		

2.3 Laptops,smartphones and the top 500

Let's check now whether our devices would have competed with the Top500 supercomputers in the past:

	Model	Performance	Top 500 year& position	number 1 HPC system
Smartphone	Samsung Galaxy XCover 4	201,32 Mflops/s	does not enter in the top500 on the first year of measurement, the 500th Supercomputer has an Rmax of 0.5 GFlops/s (equal to 2.4 times our smartphone peak performance)	Numerical Wind Tunnel,Fujitsu National Aerospace Laboratory of Japan is first in the year 1993 with a Rmax equal to 124.0 GFlops/s (equal to 616 times our cellphone's sustained peak performance)
Laptop	ASUS F556U	92.8 GFLOPs/s	3rd position at nov 1993. Remains in the top 10 until nov 1996	We have the same top 1 position with a Rpeak equal to 235.8 GFlops/s(equal to 2.5 times our laptop's theoretical peak performance)

3 SECTION 1

3.1 Model for a serial and parallel summation of n numbers

Here we discuss about modeling a simple program which consists of summing n numbers. A simple pseudocode for the serial program would be:

```
Data:array A[] of values
for i from 1 to n do
    sum = sum + A[i]
end for
return sum
```

If we choose T_{comp} as the time to compute a floating point operation we could calculate the total time of a serial computation as $T_s = N * T_{comp}$,where the code simply computes N times(the size of the problem) the sum of two values.
For the parallel program we complicate a little the execution:

```
Data:array A[] of values
```

Environment: p parallel processors
if Master process **then**
Read and Split $A[]$ into p subarrays $A_i[]$
Send $p-1$ subarrays to the other $p-1$ processors
for i from 1 to n/p **do**
$sum_0 = sum_0 + A_0[i]$
end for
Collect the resulting $p-1$ values sum_i from the processors
for i from 1 to p **do**
$sum = sum + sum_i$
end for
end if
if Slave process **then**
Receive subarrays $A_i[]$ from the Master process
for i from 1 to n/p **do**
$sum_i = sum_i + A_i[i]$
end for
Send sum_i back to the Master process
end if
return sum

If we define the times T_{read} to indicate the time needed to read a variable,and T_{comm} to indicate the time needed to communicate a variable, we can deduce the theoretical execution time of the model:

Read $A[]$
EXECUTION TIME: T_{read}

Send $p-1$ subarrays to the other $p-1$ processors
EXECUTION TIME: $T_{comm} * (p-1)$

for i from 1 to n/p **do**
$sum_i = sum_i + A_i[i]$
end for
EXECUTION TIME: $n/p * T_{comp}$
This is a parallel execution, the subarrays are added inside each processor

Send sum_i back to the Master process
EXECUTION TIME: $(p-1) * T_{comm}$

for i from 1 to p **do**
$sum = sum + sum_i$
end for
EXECUTION TIME: $(p-1) * T_{comp}$

The total sum of the execution times gives $T_p = T_{read} + (p-1+n/p) * T_{comp} + 2 * T_{comm}(p-1)$. We can calculate it with the theoretical values $T_{comp} = 2 \times 10^{-9}, T_{read} = 1 \times 10^{-4}$ and $T_{comm} = 1 \times 10^{-6}$

3.2 Scalability of the Model

Once we have the theoretical T_p and T_s we can calculate the Speedup given by the formula $Speedup(p) = T_s/T_p$. We give the following plots on the variable p:

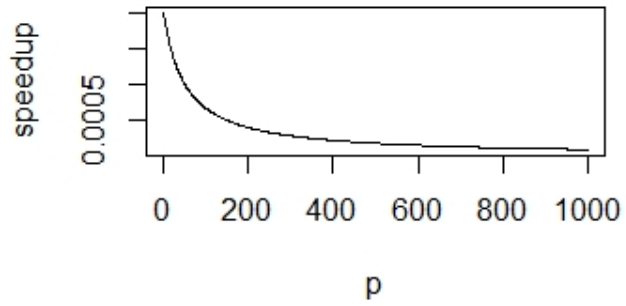

Figure 3.1: Speedup for $N = 10^2$,maximum: $speedup = 0.002$ at $p = 1$

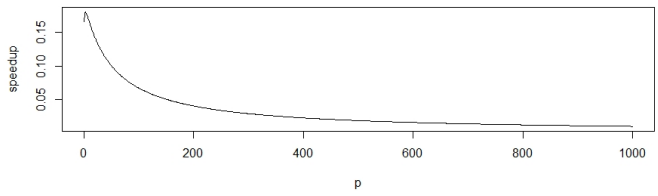

Figure 3.2: Speedup for $N = 10^4$,maximum: $speedup = 0.181$ at $p = 3$

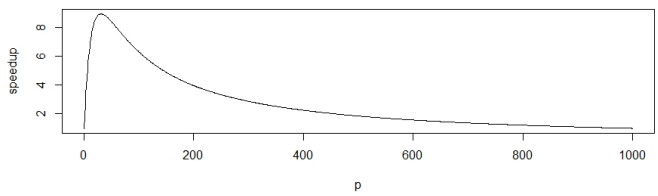

Figure 3.3: Speedup for $N = 10^6$,maximum: $speedup = 8.91$ at $p = 32$

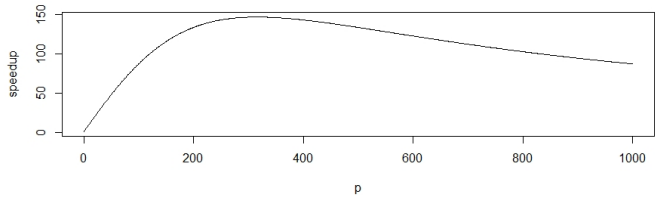

Figure 3.4: Speedup for $N = 10^8$,maximum: $speedup = 147$ at $p = 316$

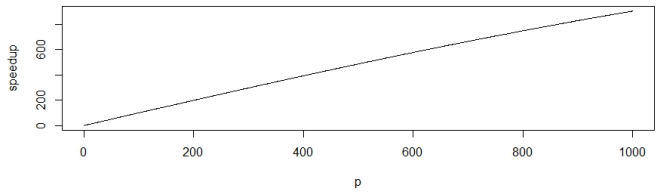

Figure 3.5: Speedup for $N = 10^{10}$,maximum: $speedup = 905$ at $p = 1000$

We notice that as we increase N we get closer to a case of perfect scaling on our number of processors. As N grows, adding a certain number of processors accelerates the calculations almost linearly. If N is low,instead, we see that the problem starts improving around $N = 10^6$, where the speedup grows up to a maximum, and then decreases:the communication time surpasses the advantage of the parallelization, thus lowering the speedup. Simply adding processors to the calculus will not speed up the process because the time it takes to the master node to assign the subarrays to the slaves and then read to read back the results grows as well. The algorithm performs well if :

- p corresponds to the maximum speedup;
- the maximum speedup is greater than 1: it makes the parallelization convenient,otherwise $T_p > T_s$ and a parallel execution would take more time than the serial one.

4 SECTION 2

4.1 *mpi_pi.c* and *pi.c* execution

We start by executing the two codes *pi.c* and *mpi_pi.c*. We have:

```
$ g++ pi.c -o pi.x
$ ./pi.x 10000000

 # of trials = 10000000 , estimate of pi is 3.141396400

 # walltime : 0.19000000
```

And the parallel file:

```
$ mpicc mpi_pi.c -o mpi_pi.x
$ mpirun -np 10 ./mpi_p.x 10000000

 # walltime on processor 1 : 0.02612305

 # walltime on processor 2 : 0.03022003

 # walltime on processor 3 : 0.02638388

 # walltime on processor 4 : 0.03122497

 # walltime on processor 5 : 0.02647901
```

```
# walltime on processor 6 : 0.02861810

# walltime on processor 7 : 0.03266811

# walltime on processor 8 : 0.02701306

# walltime on processor 9 : 0.03131413

# of trials = 10000000 , estimate of pi is 3.141720800

# walltime on master processor : 0.06575489
```

We should get the longest walltime processor time of MPI_PI.X in order to asses its speed. Let's collect various *Walltime* run times for a various number of processors.

# of processors	Maximum Walltime processor time
1	0.19799113
2	0.102010919
4	0.0.05167317
8	0.02782202
16	0.01556611
32	0.04725909
64	0.04147911

We notice that the serial execution time and the single processor parallel execution time differ because of the parallel overhead time: $T_s - T_p(1) = 0.1979911 - 0.1900000 = 7.9911ms$ .This is the parallel overhead time for a single process, calculated on the *Walltime* processor time. Those values are plotted as:

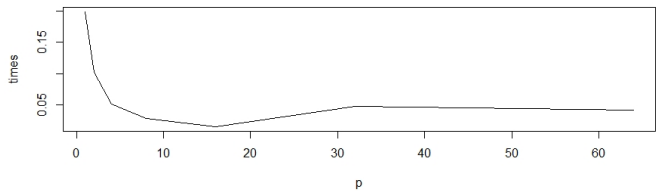

Figure 4.1: Execution time vs number of processors $N = 10^7$

The time decreases inversely to the number of processors. Now lets plot the speedup:

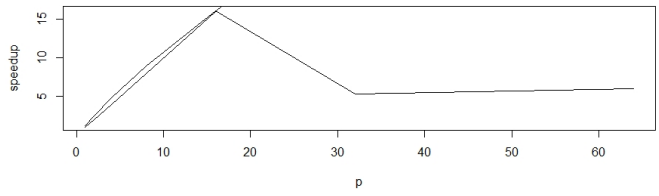

Figure 4.2: Speedup vs number of processors $N = 10^7$

We see that the speedup is not linear: the program runs faster only until around 20 processors. Let's repeat our observations by having a larger problem size. Here we have a plot that shows us the maximum *Walltime* processor time against the number of processors. Let us see the

case$N = 10^8$:

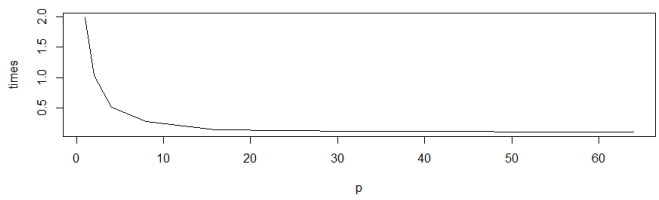

Figure 4.3: Execution time vs number of processors $N = 10^8$

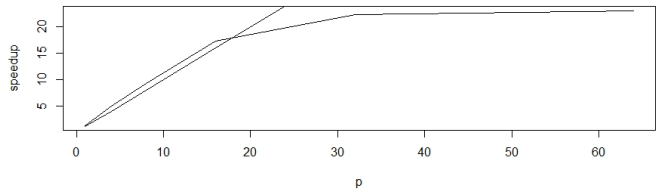

Figure 4.4: Speedup vs number of processors $N = 10^8$

We can observe that the graph is getting closer to the drawn line that represents the perfect speedup (that is speedup = p). Lets see the same graphs for $N = 10^9$:

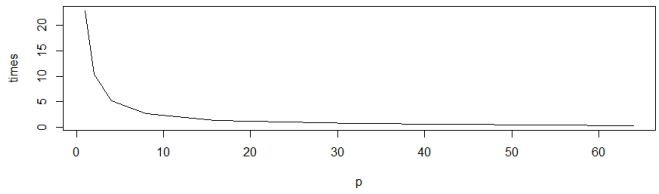

Figure 4.5: Execution time vs number of processors $N = 10^9$

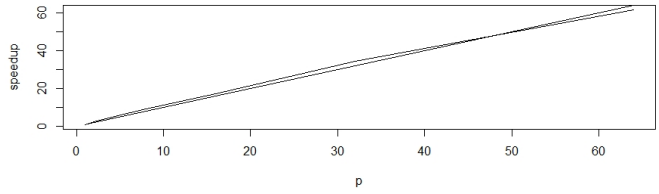

Figure 4.6: Speedup vs number of processors $N = 10^9$

Here the speedup plot is linear, for $0 < p < 64$. Adding more processors increases the execution time linearly,at least using the internal time *Walltime*.

4.2 Using Elapsed Time

If we use the *Elapsed time* from the command */usr/bin/time* we get different results:

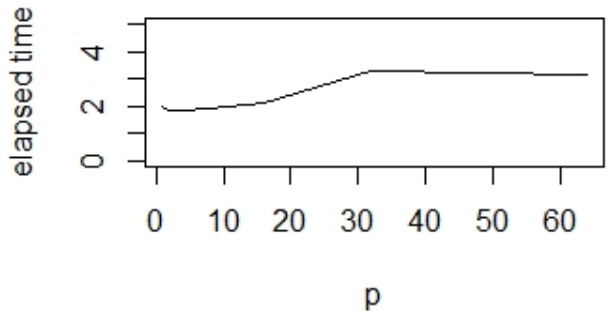

Figure 4.7: Elapsed time vs number of processors $N = 10^7$

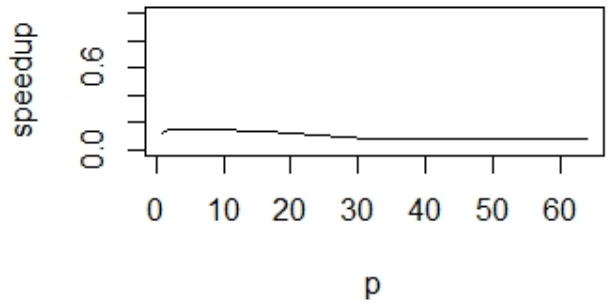

Figure 4.8: Speedup vs number of processors $N = 10^7$

As the problem size increases, we see that the problem scales better, as we see from the following two graphs, calculated with *elapsed_time* on $N = 10^9$:

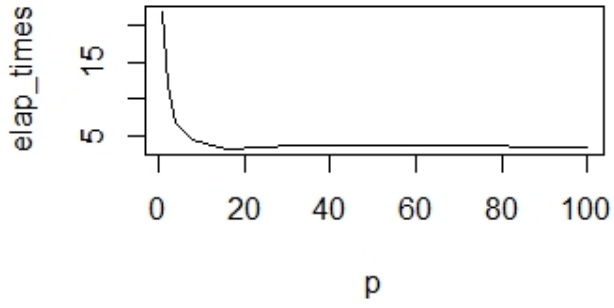

Figure 4.9: Elapsed time vs number of processors $N = 10^9$

The speedup is calculated by dividing the elapsed time for the serial code by the elapsed times of the parallel code for a different p number of processors. So $speedup = T(1)/T(p)$ as p increases.

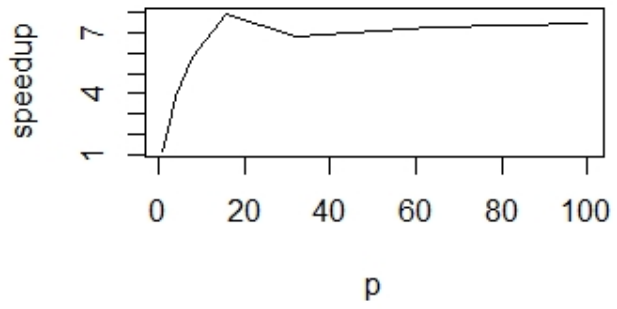

Figure 4.10: Speedup vs number of processors $N = 10^9$

We are considering *Elapsed time* and now the program doesn't scale anymore. This is normal, because the *Elapsed time* reports a more comprehensive time that includes all the overheads that are simply ignored in *Walltime*. The result is thus worse in term of scalability because the parallel time tends to grow as p grows thanks to such overheads. In the end they surpass the time saved by parallelizing the execution.

4.3 Parallel overhead

Here we present the model for deducing the overhead of our program. We study the case where $N = 10^7$ and we choose to plot the differences between the maximum processor walltime and the minimum processor walltime for a growing number of processors. We obtain the following graph

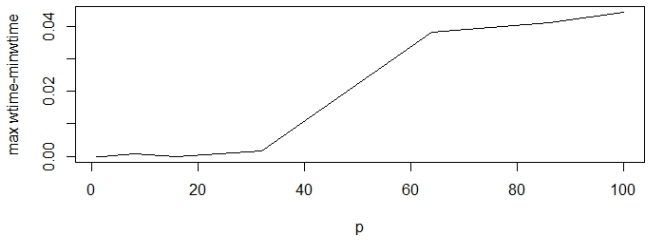

Figure 4.11: Overhead vs number of processors $N = 10^9$

We can see that our overhead model grows as p grows. This is explained by the extra computation that is adopted in order to manage more nodes. The growth of the overhead explains why measuring the scalability using *Elapsed time* does not give a scaling result, while the *Walltime* scalability does.

4.4 Weak scaling

We run a bash script to automatically collect the execution times for various proportional values of p and N. N is equal to $10^7 * p$ as p grows. We obtain the following plot using the maximum *Walltime* time for each number of processors:

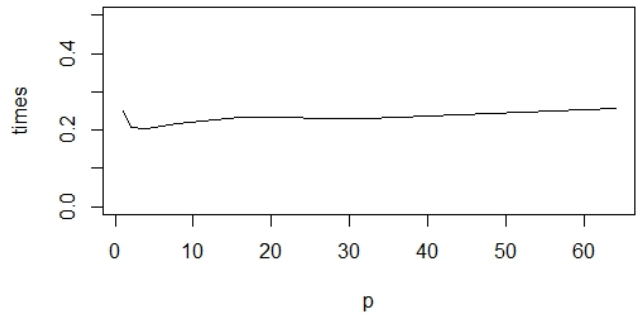

Figure 4.12: Weak scaling case: execution time vs number of processors $N = 10^7 * p$

We see that the execution time is almost constant,which is what we expect with a weakly scalable program.The weak scalability aims to enlarge the problem size and the number of processors while leaving the execution time constant. We get a different result for considering the *Elapsed time* from the */usr/bin/time* command:

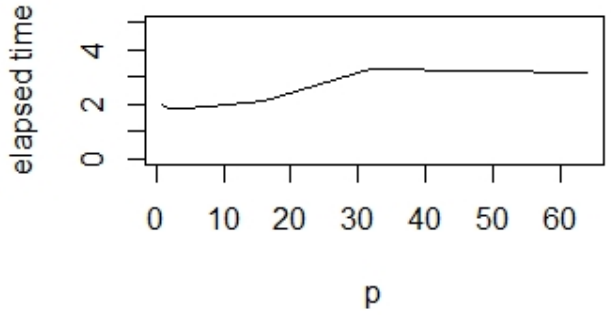

Figure 4.13: Weak scaling case: elapsed time vs number of processors $N = 10^7 * p$

We also plot the speedup we gained by calculating $speedup = T(10^7 * p, 1)/T(10^7 * p, p)$ where $T(N, p)$ is the time calculated on the problem size N and on p processors(the problem size is proportional to the number of processors).

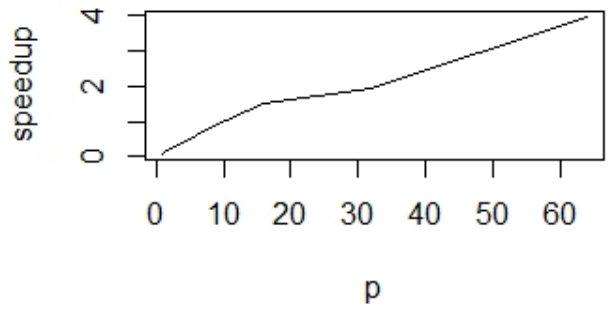

Figure 4.14: Weak scaling case: elapsed time speedup vs number of processors $N = 10^7 * p$

Here the speedup is somehow linear, showing us that the fact that the elapsed time is almost constant is an advantage in relation to the serial execution time,which instead grows:

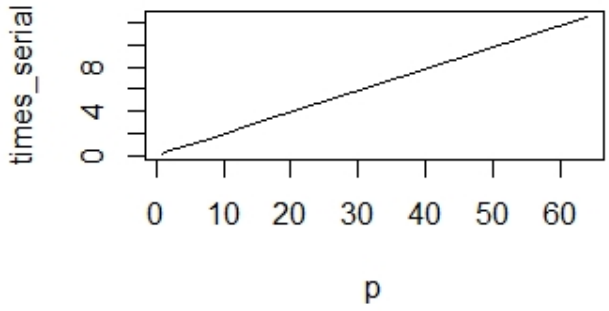

Figure 4.15: Serial program case: elapsed time vs number of processors $N = 10^7 * p$

So the resulting speedup grows because of this distinction between an almost constant elapsed execution time and a linear serial execution time. As the parallel execution time for a growing problem size stays almost the same, the serial code execution time grows,incrementing the speedup and thus making it more convenient to apply the parallelization.

5 SECTION 3

Here we have the two c++ codes *sumNumbers_mpi.cc* and *sumNumbers_coll.cc*.
The two codes implement the pseudocode we wrote above, including the variations on the assignment . We also don't use any array, but we calculate on the run the addends for the partial sum. Both codes return the sum of N consecutive integers starting from 1 to N and both codes implement the calculation using a parallel approach. Both codes also address the case in which N is not divisible by the number of processors p. In this case the master node takes care of the summation of the integers that are left out from the slave processors. The two codes don't work on one single processor, because the sum needs to be splitted at least among two processors.

The code *sumNumbers_mpi.cc* uses only *MPI_Send()* and *MPI_Recv()* while the code *sumNumbers_coll.cc* uses the collective operations *MPI_Bcast()* and *MPI_Reduce()* instead. The codes where tested using as an input a file containing the number $N = 10^9$.
We used *MPI_Walltime()* to collect a few particular subprocess times for $N = 10^9$ and $p = 10$:

$$\begin{aligned} &T_{read} = 3.38554e-05s \\ &T_{comp} * (\frac{N}{P}) = 0.355836s \\ &if N = 10^9, P = 10 \rightarrow T_{comp} = 0.355836/100000000 = 3.5836e-9s \\ &T_{comm} = 2.14577e-06s \end{aligned} \tag{5.1}$$

We see that these values are close to the theoretical values we gave before,making those plausible.

6 SECTION 4

Now its time to plot the execution time of the function we wrote. We want to test the function's strong scalability.
We consider the *Walltime* execution time for $N = 10^9$ and $N = 10^{10}$.We start by the **naive-implemented code**:

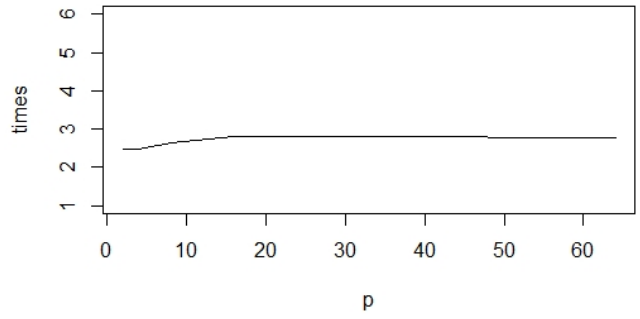

Figure 6.1: Walltime execution time vs number of processors $N = 10^9$

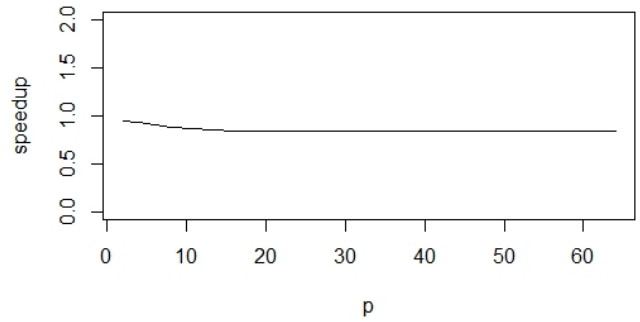

Figure 6.2: Walltime speedup vs number of processors $N = 10^9$

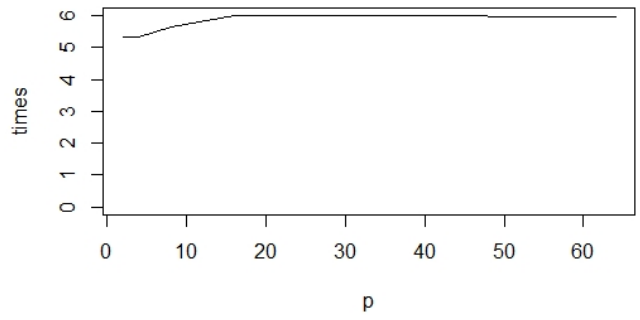

Figure 6.3: walltime execution time vs number of processors $N = 10^{10}$

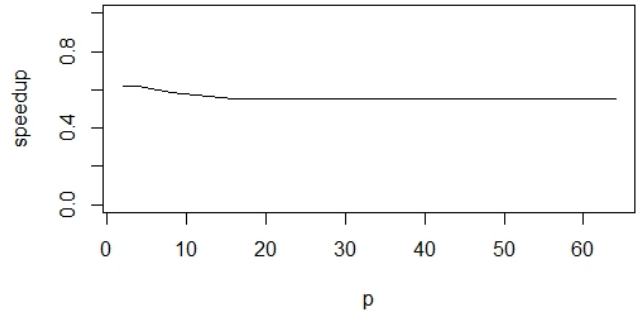

Figure 6.4: walltime speedup vs number of processors $N = 10^{10}$

We see that the program isn't scaling as we predicted in our model. Where by 10^10 the theoretical model was scaling perfectly, our model is instead not scaling. The execution time seems to be constant. The executed program was the naive-implemented one,and probably the communication time increases too much and nullifies the time saved throughout the parallelization.
We should also consider that the partial sums consists of more than one single floating point operation. So our program places more time on the computation part of the execution than our model. A better model would reconsider the formula for the parallel execution time T_p. Like for example $T_p = T_{read} + (p-1+n_{sum} * n/p + n_{serial}) * T_{comp} + 2 * T_{comm}(p-1)$ where n_{sum} represents the number of floating point operations required to calculate a single addend and n_{serial} represent the floating point operations that are used to calculate the variable *subarray_size*,which is the size of the subproblem.This could be an example on how to make a best theoretical model that, by having a larger T_p, will avoid scaling for a larger problem size N.
But here basically adding more processors does not speed up the execution time, and the program is poor at parallelizing the task at hand. Let's also see the case of *Elapsed time*,from which we draw the same conclusions:

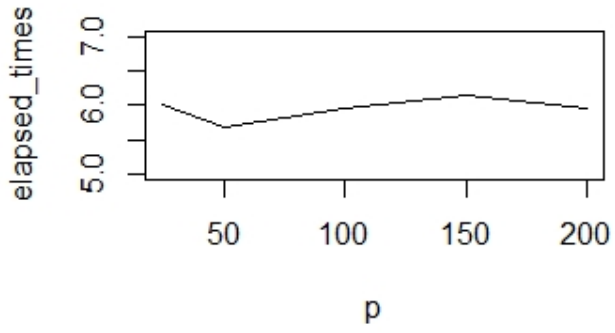

Figure 6.5: Elapsed time execution time vs number of processors $N = 10^9$

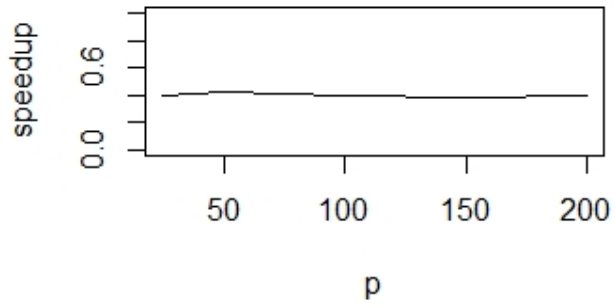

Figure 6.6: Speedup of the elapsed time vs number of processors $N = 10^9$

We get a better result if we plot the scalability for the **sumNumbers_coll.cc** code, with $N = 10^9$. The time taken is the *Walltime* given by *MPI_Walltime()*:

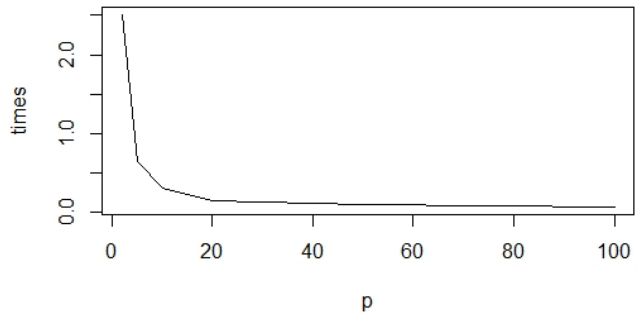

Figure 6.7: Walltime execution time vs number of processors $N = 10^9$

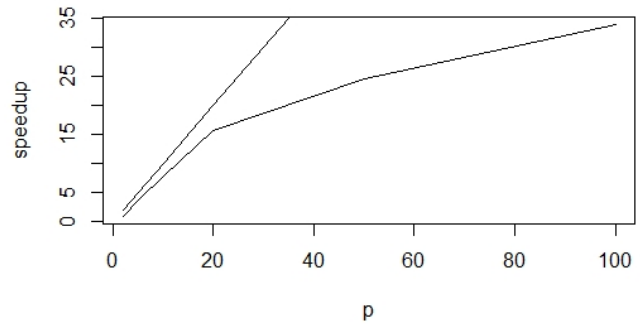

Figure 6.8: Walltime speedup vs number of processors $N = 10^9$

We see that this second code scales better than the first one. We replicated the results with $N = 10^{12}$,although there is no common data type to store the result of the sum and although we couldn't test a simple serial code of the summation with such a big number. We estimate that for $N = 10^{12}$ then $T(1) = 5s$.

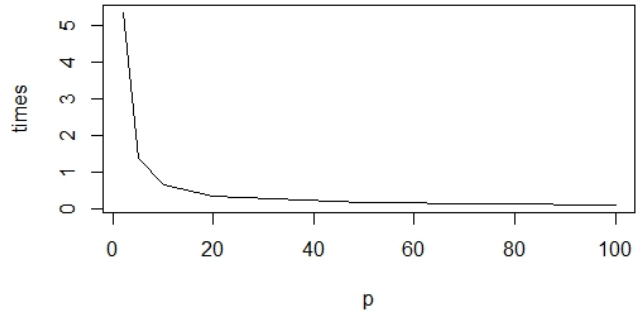

Figure 6.9: Walltime execution time vs number of processors $N = 10^{12}$

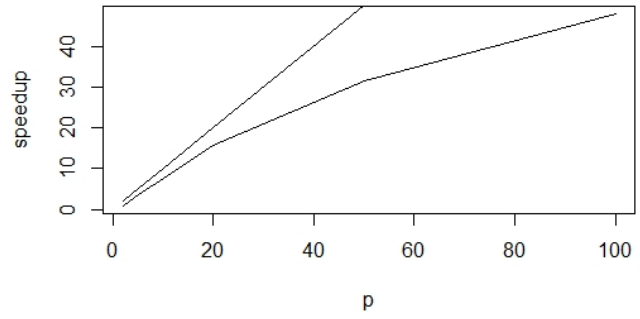

Figure 6.10: Walltime speedup vs number of processors $N = 10^{12}$

We can deduce that the code is somehow scaling as N grows.Maybe if N is large enough we

would see that the program scales.

6.1 Elapsed time case

Let's also plot the **Elapsed time** of the **sumNumbers_coll.cc** code for $N = 10^7$ to make a comparison:

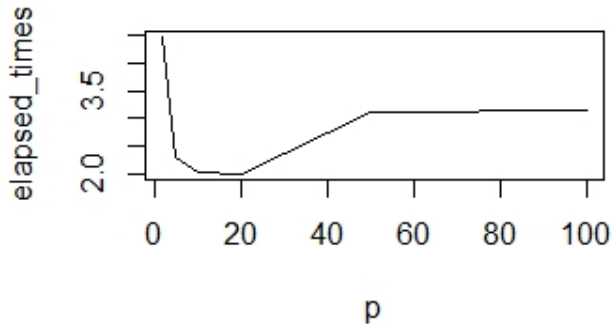

Figure 6.11: Elapsed time vs number of processors $N = 10^7$

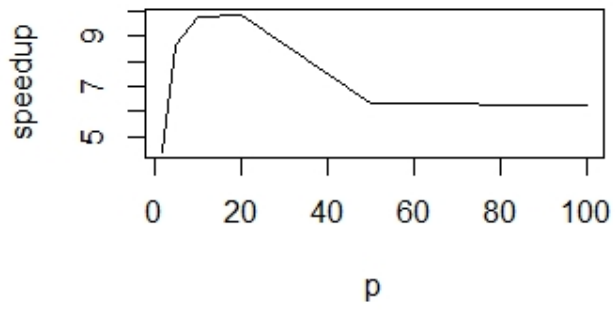

Figure 6.12: Speedup of the elapsed time vs number of processors $N = 10^7$

We see that the code doesn't scale for this problem size either. Since **Elapsed time** adresses a wider execution time, we expected a worse result. We can also see the case $N = 10^9$:

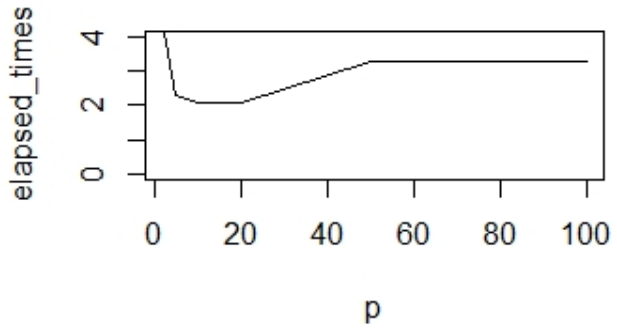

Figure 6.13: Elapsed time vs number of processors $N = 10^9$

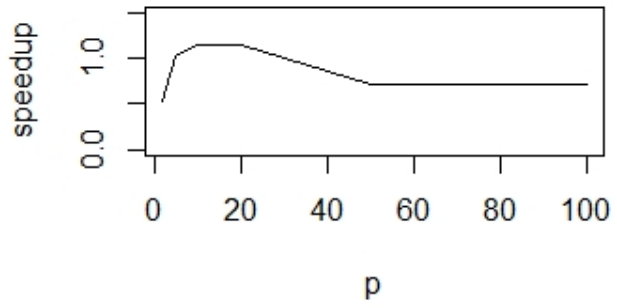

Figure 6.14: Speedup of the elapsed time vs number of processors $N = 10^9$

We are increasing the size of the problem, yet the code doesn't scale. We conclude that there must be a problem in the initialization of the code. The parallelization is not effective in reducing the execution time . Adding more processors does not speed up the execution time. This is because the *elapsed time* counts the overhead time too, while just using *walltime* returns us the execution time of the portion of the code that scales better. In the end they are different measurements that take into account different parts of the execution, and yet the elapsed time is the best one because it returns a time that measures even the overhead,which is fundamental in assessing the program's scalability, and thus its performance as a parallel problem. We must conclude that the code we wrote is not good at parallelizing the problem, although we focused on making it as close as possible to the assignment's description. The code itself works well in calculating the sum of N integers from 1 to N,but yet the code does not parallelize well. We see that there is a difference between a good-working code and an efficient one.