UNIVERSITÀ DEGLI STUDI DI TRIESTE

First FHPC Assignment

Nicola Domenis

November 2, 2019

1 PREVIEW

In this assignment we will present the following subjects:

- the production of a parallel program code
- the graphs of the theoretical and real speedup of the code
- anything else

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* = $2.90 Ghz$, $FLOPs = 16$ and $number of cores = 2$ for our computer architecture,an intel i7 with a Kaby Lake microarchitecture; 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 test. 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	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 technologies 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 of 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 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 $T_s = N * T_{comp}$,whereas 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 and Split $A[]$ into p subarrays $A_i[]$
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:

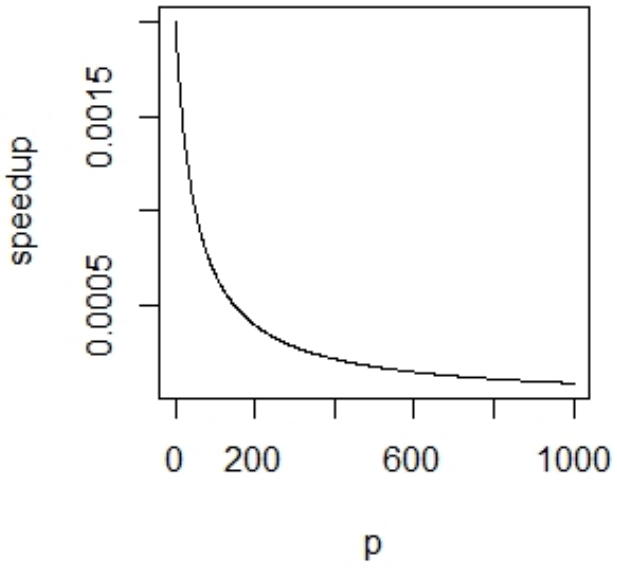

Figure 3.1: Speedup for $N = 10^2$,maximum: $speedup = 0.00199$ at $p = 1$

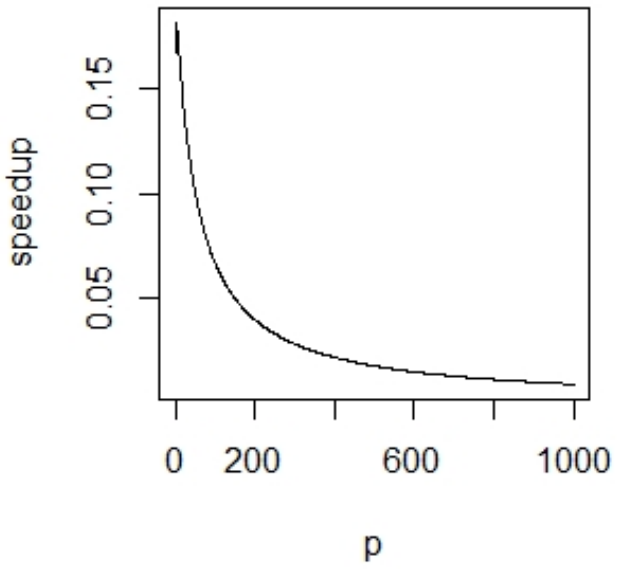

Figure 3.2: Speedup for $N = 10^4$,maximum: $speedup = 0.180$ at $p = 3$

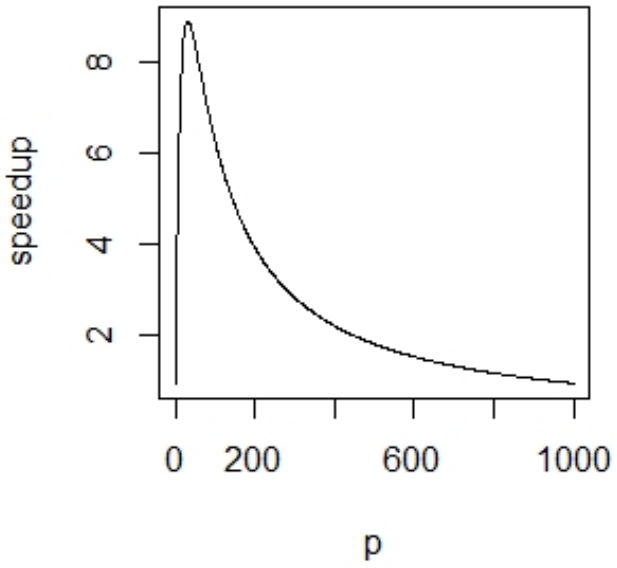

Figure 3.3: Speedup for $N = 10^6$,maximum: $speedup = 8.90$ at $p = 32$

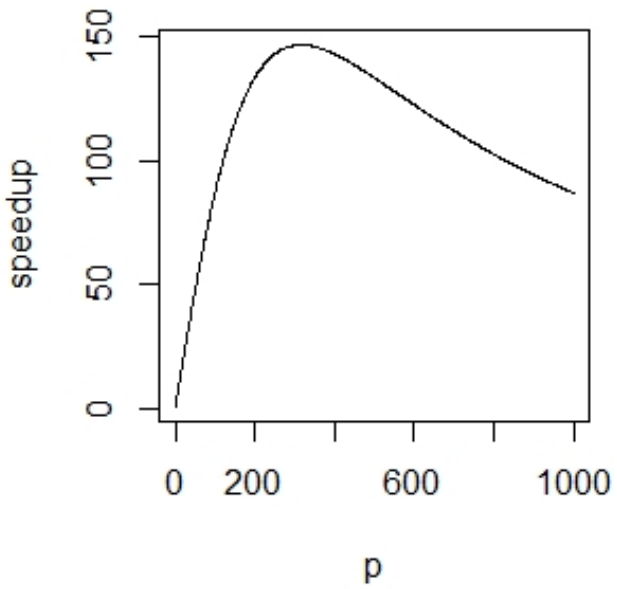

Figure 3.4: Speedup for $N = 10^8$,maximum: $speedup = 146.67$ at $p = 316$

We notice that this is a case of strong scaling, where adding a certain number of processors accelerate the calculations of a same-size problem, but only to a certain point. After the maxi-

mum, the communication time overpowers 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 subarray to the slaves will become too high to be easily computed. This is different when we have a high problem size, starting from around $N = 10^4$ we see that the model starts scaling from p=1 to the maximum of the plot.

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
$ time ./pi.x 10000000

 # of trials = 10000000 , estimate of pi is 3.141396400

 # walltime : 0.19000000

real    0m0.275s
user    0m0.271s
sys     0m0.001s
```

And the parallel file:

```
$ mpicc mpi_pi.c -o mpi_pi.x
$ time 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

real    0m1.890s
```

```
user    0m11.881s
sys     0m0.630s
```

We should get the longest time of all the parallel execution times of mpi_pi.x in order to asses its speed.
Let's collect various run times for a different number of processors.

# of processors	Master processor speed
1	0.19828200
2	0.10205293
4	0.05147886
8	0.04555607
16	0.01546288
32	0.00758505

We notice that the serial run time above and the 1-processor parallel run time on this table differ because of the parallel overhead time: $T_p(1) - T_s = 0.19828 - 0.190000 = 8.28ms$ Those values are plotted as:

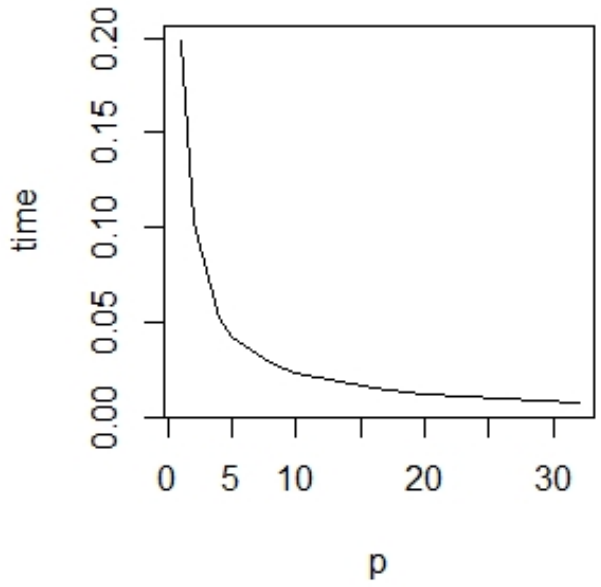

Figure 4.1: Speed vs number of processors graph

The time decreases with an inverse proprotionality to the time. Now lets plot the speedup:

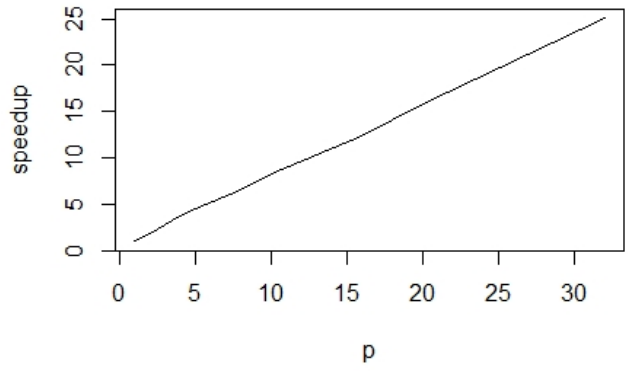

Figure 4.2: Speedup vs number of processors graph

We see that it is linear,thus we have strong scalability: as we increase the number of processors we obtain a faster parallel program.
Let's repeat our observations by having a larger number of parallel processors. Here we have a plot that shows us the master execution time according to the number of processors. The times are not strictly decreasing because the processor is less scalable as p grows.

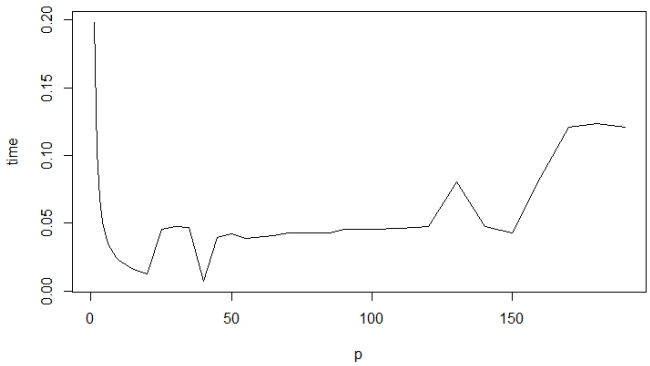

Figure 4.3: N= 10000000Execution time vs number of processors

We can see that the speedup decreases for a large number of processors

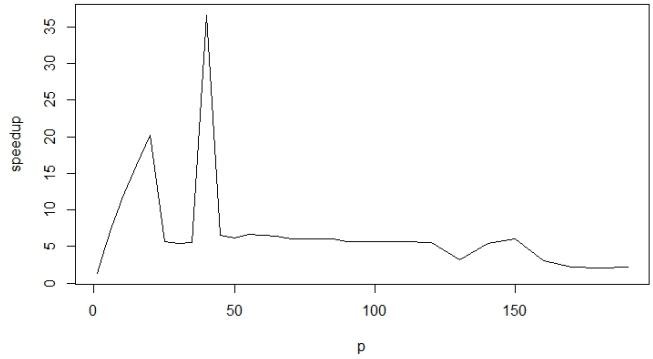

Figure 4.4: N=10000000Speedup vs number of processors

We can observe the first linear growth that we plotted earlier. The speedup then decreases. Lets see the same graphs for N=100000000:

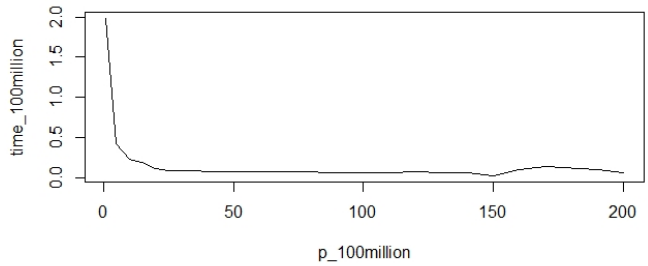

Figure 4.5: N= 100000000 execution time vs number of processors

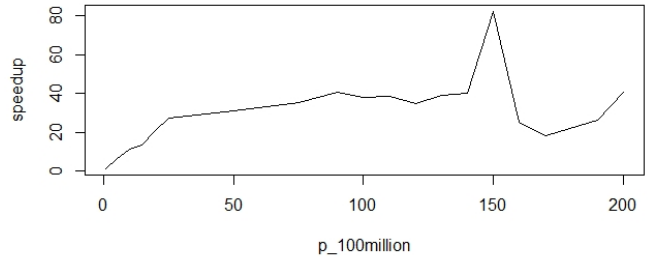

Figure 4.6: N= 100000000 speedup vs number of processors

Now lets see the results for N=1000000000:

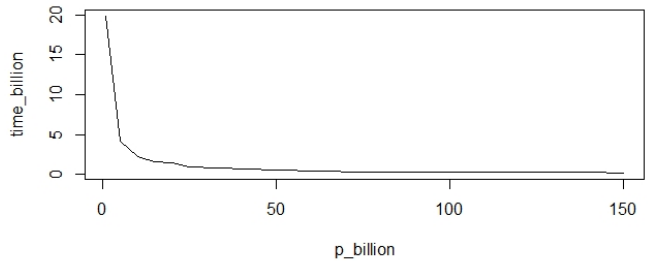

Figure 4.7: N= 1000000000 execution time vs number of processors

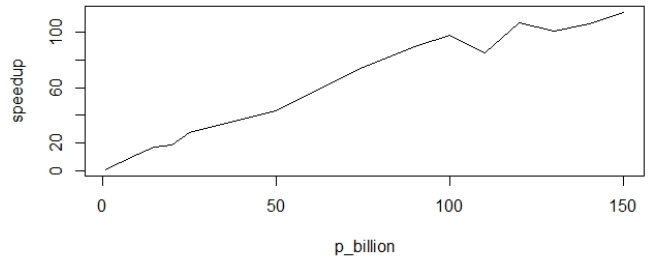

Figure 4.8: N= 1000000000 speedup vs number of processors

We have modeled the strong scalability of the program

5 SECTION 3

5.1 The written code

Now we will focus on the two codes that represent an implementation of the theoretical model for the summation of the first n integers. We have the serial implementation in the code SUM_OF_N.C and the parallel implementation of the code SUMNUMBERS_MPI.C. Let's try the code for $n = 1000$:

```
$ g++ sum_of_n.c -o sum_of_n.x
$ time ./sum_of_n.x < n.txt
total sum: 500500
real    0m0.005s
user    0m0.000s
sys     0m0.002s

$ module load openmpi
$ mpicc mpi_sum_of_n.c -o mpi_sum_of_n.x -std=c11
time mpirun -np 10 ./mpi_sum_of_n.x < n.txt
time spent on process 1 is 0.000041 seconds
time spent on process 2 is 0.004197 seconds
time spent on process 3 is 0.000022 seconds
time spent on process 4 is 0.009296 seconds
```

```
time spent on process 5 is 0.004883 seconds
time spent on process 6 is 0.000022 seconds
time spent on process 7 is 0.008823 seconds
total sum: 500500
time spent on process 0 is 0.023464 seconds
time spent on process 8 is 0.000022 seconds
time spent on process 9 is 0.004411 seconds

real    0m1.853s
user    0m11.561s
sys     0m0.635s
```

Let's try again with N=1000000000 and p = 10. We collect a few particular times of the execution:

$$\begin{aligned} T_{read} &= 3.38554e-05 seconds \\ T_comp(N/P) &= 0.355836 \\ \rightarrow T_{comp} = 0.355836/100000000 &= 3.5836e-9 T_comm = 2.14577e-06 \end{aligned} \tag{5.1}$$

We see that those values are close to the theoretical values we gave before