**First Assignment Report**

**Section 0**

Theoretical Peak performance for laptop:

* CPU: Intel(R) Core(TM) i7-7500U
* Frequency: 2.70 GHz
* 2 Cores
* Floating point operations for cycle: 16 (Intel Kaby Lake architecture, https://en.wikipedia.org/wiki/FLOPS)

- Write here the final result obtained:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Your model | CPU | Frequency | Cores | Peak performance |
| Laptop | i7-7500U | 1 | 2.70 GHz | 2 | 86.4 GFLOPS |

- Compute sustained Peak performance for your cell-phone

-Identify the CPU and calculate the Peak performance of the CPU: Qualcomm Snapdragon 625, Octa core, 2 GHz, 2 Flops per cycle.

| | Your model | sustained performance | size of matrix | Peak performance | memory |

| ---------- | ----------------------- | --------------------- | -------------- | ---------------- | ------ |

| SmartPhone | Qualcomm Snapdragon 625 | 1209 MFLOPS | 2500 | 32 GFLOPS | 4 GB |

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

- Fill in the following table

| | Your model | performance | Top500 year | number 1 HPC system | number of processors (TOP500) |

| ---------- | ----------------------- | ----------- | --------------- | ------------------------------------------- | --------------------------------- |

| SmartPhone | Qualcomm Snapdragon 625 | 1209 MFLOPS | Until 11/1994 | Until 1985 (Cray-2 1.9 GFLOPS) | 4 (Cray 2) |

| Laptop | i7-7500U | 86.4 GFLOPS | Until 11/2001 | Until 11/1993 (Numerical Wind Tunnel Japan) | 140 (Numerical Wind Tunnel Japan) |

​

## Section 1: theoretical model

- devise a performance model for a simple parallel algorithm: sum of N numbers

- Serial Algorithm : n-1 operations

$T\_{serial}= N\*T\_{comp}$

$T\_{comp}$= \*time to compute a floating point operation\*

- Parallel Algorithm : master-slave

- read N for each processor ===> $P T\_{read}$

$T\_{comm}$ = \*time each processor takes to communicate one message, i.e. latency..\*

$T\_{read}$ = \*time it takes to read\*

- N/P sum over each processors (including master) ===> $T\_{comp} N/P$

- Slaves send partial sum ===> $(P-1)T\_{comm}$

- Master performs one final sum ===> $(P-1) \times $T\_{comp}$

the final model: $T\_p= T\_{comp}\times (P -1 + n/P) +P T\_{read} + (P-1)\times T\_{comm} $

- compute scalability curves for such algorithm and make some plots

- assumptions:

- $T\_{comp} =2 \times 10^{-9}$

- $T\_{read}= 1 \times 10^{-4}$

- $T\_{comm}= 1 \times 10^{-6}$

Play with some value of N and plot against P (with P ranging from 1 to 1000)

- Comment on them:

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

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

- Can you try to modify the algorithm sketched above to increase its scalability ?

(hints: try to think a better communication algorithm)

## Section 2 : play with MPI program

### 2.1: compute strong scalability of a mpi\_pi.c program

The application we are using here is a toy application: a Monte-Carlo integration to compute PI; The idea is to have a circle inscribed inside a square of unit lenght. The ratio between the area of the square (1) and the circle (pi/4) is $\pi/4$. Therefore, if we randomly choose N points inside the square, on average, only `M=N\*pi/4` points will belong to the circle. From the last relation we can estimate pi. We provide a basic serial implementation of the algorithm ( program pi.c ) and we also give a parallel MPI implementation ( mpi\_pi.c ) that computes PI by the same algorithm using MPI to compute the algorithm in parallel. Your exercise is to see how well this application scales up to the total number of cores of one node. You can modify the codes we have given, or you can write your own.

Steps to do:

- Compile the serial and parallel version.

- 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.

Hint: use /usr/bin/time command to time all the applications

- 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 ?

- 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 calculationgives you some idea of the overhead associated with MPI. Again what time do you consider here ?

- Keeping Niter = 10 millions, run the MPI code for 2, 4, 8 and 16 and 20 MPI processs.

In principle, all that needs to be done is to run the program multiple times, changing the -np argument to mpirun each time. Such a study could be as simple as the following bash for loop:

```

for procs in 1 2 4 8 16 32 64 ; do

time mpirun -np ${procs} my-program the-input

done

```

A real study is of course a litte bit more complicated because we neeed to get the performance of each of those runs. Furthermore we run on Ulysses (or any other HPC resources) and then we need to submit jobs.

In any case it is important to script your job executions, so that you can quickly measure scalability. So in this part of the lab you need to implement a strong scaling script. Script will be implemented by means of bash scripting, or if you prefer you can use python as well.

- Make a plot of run time versus number of nodes from the data you have collected.

- Strong scalability here would yield a straight line graph. Comment on your results. Repeat the work playing with a large dataset.

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

### 2.2: identify a model for the parallel overhead

In this section you are asked to identify a simple model that estimate the parallel overhead of our simple program as a function of the number of processor. We ask here to extend the simple model discussed previously and adapt/modify to the present program.

To do this please review carefully all the data collected so far and identify which is the indirect method to measure the parallel overhead from the output of the program.

### 2.3: weak scaling

- Now let us do some running that constitutes a weak scaling test.

This means increasing the problem size simultaneously with the number of nodes being used. In the present case, increasing the number of iterations, Niter.

- 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 differrent number of moves.

- Plot on the same graph the efficiency (T(1)/T(p)) of weak scalability for different number of moves and comment the results obtained.

## Section 3: implement a parallel program using MPI

We request here to implement the sum of N number using MPI paradigm. Programming language can be any among python C/ C++ and Fortran90.

The program should:

- read from an input file a number N arbitrarly large (at least 1Billion number)

- distribute to each processor the amount of work to be done

- the array of integer numbers should contain integer numbers ( from 1 to N)

- deals with cases where N (size of the array) is not exactly divisible by P (number of Processors)

- collect the partial sums on one single processor and print it down.

- measure the reading time, the communication times (initial distribution phase and final collectin phase) and the computation time (hints: use MPI\_walltime)

- write as final output the time taken by each processor

- implement the algorithm with naive communication algorithms and THEN using collective operations

## Section 4 (run and compare)

In this final section you are ruquested to run and measure strong scalability of the program written in section 3.

- Run the implemented program for different large enough sizes of the array

- Plot as in section 3 scalability of the program

- compare performance results obtained against the performance model elaborated in section 2

- comment on the assumption made in section 1 about times: are they correct ? do you observe something different ?