Final assigments FHPC course 2022/2023

This is the final assignment for the 2022/2023 FHPC course. It consists of several exercises: two of them mandatory and a third one to be choosen among several options.

version 1.0 Please note: this document can be modified several times in the next few days in order to improve the clearness of the information to provide a better understanding of what we are asking.

Rules:

Materials (code/scripts/pictures and final report) should be prepared on a github repository, starting by this one and sharing it with the teachers. The final report (in pdf format) should be sent by e-mail to the teachers. In the mail you should provide also the github repo to allow us to check, if needed, the materials.

deadlines

You should send us the e-mail at least one week before the exam. For the first two scheduled "appelli" this means:

- exam scheduled at 07.02.2023 ***deadline 01.02.2023 at midnight***
- exam scheduled at 23.02.2023 ***deadline 16.02.2023 at midnight***

Important note: the assigments is valid till ***01 OCTOBER 2023***

If you plan to take the exam later that this date please contact us by email

Structure of this github directory:

```
README.md
report.pdf
exercise1/README.md
exercise1/all the files for exercise 1
exercise2/README.md
exercise2/all the files for exercise 2
exercise3/README.md
exercise3/all the files for exercise 3
```

where:

- report.pdf will contain a detailed report of all the exercises in this documents.
- README.md a file in markdown language that describes briefly what you did.
- exercise1/README.MD should explain all the files present in the directory and give detailed information on
  - which software stack we should use to compile the codes and run all the programs you prepared.
  - a short description of scripts you write and use to complete the exercise

    - a short description of the datasets available (in .csv file are commma separated values) to produce any kind of figures you provided
- exercise2/README.MD should explain all the files present in the exercise 2 directory
- exercise3/README.MD should explain all the files present in the exercise 3 directory

exercise 1: parallel programming

see pdf in the `exercise1/` folder.

exercise 2 : Comparing MKL, OpenBLAS and BLIS on matrix-matrix multiplication

The goal of this exercise is to compare performance of two math libraries available on HPC: MKL, OpenBLAS and BLIS. The BLIS library must be downloaded and compiled by the students as optional phase.

You will perform the comparison focusing on the level 3 BLAS function called *gemm*. Such function comes in different flavours, for double precision (dgemm), single precision (sgemm).

The code provided `gemm.c` is a standard *gemm* code, where 3 matrices A,B,C are allocated, A and B are filled and the BLAS routine calculates the matrix-matrix product C=A*B

The relevant call is

```
  GEMMCPU(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k, alpha, A, m,
B, k, beta, C, m);
```

where `GEMMCPU` is a macro to easily switch from single precision (`cblas_sgemm`) to double precision (`cblas_dgemm`).

The standard cblas interfaces are

```
  void cblas_dgemm(const enum CBLAS_ORDER Order, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_TRANSPOSE TransB, const int M,
const int N, const int K, const double alpha, const double *A, const int
lda, const double *B, const int ldb, const double beta, double *C, const
int ldc)

  void cblas_sgemm(const enum CBLAS_ORDER Order, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_TRANSPOSE TransB, const int M,
const int N, const int K, const float alpha, const float *A, const int
lda, const float *B, const int ldb, const float beta, float *C, const int
ldc)
```

The first argument `Order` specifies wheter we are using column major storage or row major storage. `TransA` and `TransB` tell that the matrices should be taken as they are, so not transposed. The rest are the standard GEMM arguments, to perform the operation

```
    C(M,N) = alpha*A(M,K)*B(K,N) + beta*C(K,N)
```

The parameters `lda`, `lbd` and `ldc` are the leading dimensions of the matrices, which, since we are using colmajor order, should be the number of rows (lda=M, ldb=K, ldc=M)

Compile and run

To compile and run the code, first submit an interactive job to the queue system and request a shell on the target machine.

On ORFEO:

```
salloc -n 128 -N1 -p FHPC --time=1:0:0
```

Load the needed module

```
  module load architecture/AMD
  module load mkl
  module load openBLAS/0.3.21-omp
```

In `Makefile` modify variable OPENBLASROOT with your installation path to OpenBLAS (if you compiled by yourself the library) or use the enviromental variable `OPENBLAS_ROOT` provided by the module `openBLAS`:

```
OPENBLASROOT=${OPENBLAS_ROOT}
```

and set the desired compilation flag (`-DUSE_FLOAT` / `-DUSE_DOUBLE`).

Type

```
srun -n1 make cpu
```

Compilation will make use of the Intel MKL implementation, which is multithreaded. By default this variable has been set by the queue system to the number of cores requested at submission time, but can be changed at runtime. by means of the environment variable `OMP_NUM_THREADS`.

To run the code simply issue

```
srun -n1 --cpus-per-task=128  ./gemm_mkl.x
srun -n1 --cpus-per-task=128  ./gemm_oblas.x
```

NOTE: The argument `--cpus-per-task` notify to the scheduler that you can use 128 cores to run the application. If you use *uncorrectly* `-n128` you will run 128 application instances. We aim to start only one instances and allow it to use 128 cores.

With no argument it will calculate a matrix multiplication with M=2000 K=200 and N=1000 with OMP_NUM_THREADS set to number of processor you choose when you submit the the job (24 in this case)

You can now use positional argument to specify the size

```
srun -n 1 --cpus-per-task=128 ./gemm_oblas.x 40000 40000 20000
```

You can now play with the code varying the size of the matrix and also the number of threads and affinity.

Mandatory Steps to do:

choose one architecture and then:

- measure scalability over the size of the matrix at fixed number of cores (12 for thin or 64 for epyc)
  - increase the size of matrices size from 2000x2000 to 20000x20000 (single precision) and analyse the scaling of the GEMM calculation for at least MKL and openblas.
  - repeat the above point using double precision
  - compare with the peak performance of the processor.
  - play with different threads allocation policy and discuss the results.
- measure scalability over the number of cores at fixed size. Choose an intermediate size and then:
  - increase the number of cores and analyse the scaling of the GEMM calculation for at least MKL and openblas.
  - repeat the above point for double precision

Optional steps to do:

- repeat the above using BLIS library as well
- repeat the above using both Intel and Epyc node and compare performance

exercise 3 (optional)

These exercises are just sketched; an updated and more detailed version will be provided before the end of the year.

You may choose among the following:

exercise 3.1: perform a comparision between epyc and fat nodes for level 1 genomic analysis multithreaded software (bcl2fastq)

You will be asked to compare the time it takes to run the bcl2fastq software on fat and epyc nodes and find out the overall scalability on Epyc nodes.

exercise 3.2: install and run MLperf benchmark on GPU nodes

You will be asked to download/compile and run on ORFEO gpu nodes and perform a benchmark analyis on both V100 and A100 nodes

exercise 3.3: compare different broadcast algorithms of MPI algorithm within openMPI library

You will be asked to evaluate different algorithms implemented with openMPI and discuss their choice.