Project 1: Performance. Matrix Matrix multiplication.

Provided file: **Chem281P2.cpp**:

In this project you will analyze how memory access and other progamming details affect the performance of our algorithms. We will also use perf to do some rudimentary performance analysis. It is recommended that you run your program on Perlmutter. Instruction on how to use Perlmutter will be provided on another writeup. The matrix multiplication problem can be stated as computing each entry c_{ij} of an M-row, P-column matrix C as the product between an M-row, N-column matrix A and an N-row, P-column matrix B. Each entry is computed using the formula below.

$$c_{ij} = \sum_{k=0}^{N} a_{ik} * b_{kj}$$

We have to implement 3 different algorithms to perform this computation.

Problem 1:

In the first one, **matmuloop**, you will loop over the three indeces i, j, k as shown in the matmulloop code in the file Chem281P2.cpp. The sequencing of these loops may or may not be optimal. You should reorder these loops for maximum performance. After that you will parallelize this routine. Which loops can be parallelized without creating potential data-dependency errors? Also, you can parallelize one or more loops. Which option gives the best performance. Perform a scalability study using 1, 2, 4, and 8 cores (you can use Perlmutter) How does this code scale?
To invoke this routine, type

```
./Chem281P2 --loop --threads <num>
```

Note: if –threads is not specified, the number of threads used is taken from the environment variable OMP_NUM_THREADS. You can change the value of this environment variable by typing (bash shell)

```
setenv OMP_NUM_THREADS=<num>
```

For reference, my implementation of this problem runs on about 21 seconds on 4 Perlmutter cores.

Problem 2:

One way to improve the performance of this algorithm is to *tile* the matrices to improve memory locallity and better use of caches. The code in **matmultile** implements the outer loop, that is the partitioning of the matrix into tiles. You task is to implement the inner loop, that is the computation of the matrix product for each tile.
You can run matmultile by typing

```
./Chem281P2 --row_tile <num> --col_tile <num> --inner_tile <num> --threads<num>
```

You can set one, two, or three parameters here. The code uses default values for those parameters that have not been specified. If you just want to use the default values, invoke the code as

```
./Chem281P2 --tiled
```

To keep the code simple only use power of 2 values for the <num> parameter, that is 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048.

You task is to write the inner loop and parallelize this routine. Based on your experience in Problem 1, which loop/loops should you parallelize. Finally, perform a scalability study on one, two, 4, and 8 processors. How does the code scale?

For reference: my implementation runs on about 5.12 seconds of elapsed time on 4 cores.

Problem 3:

Implement the matrix vector product using the **Cblas** routine *cblas_ dgemm*. Instructions on how to use this routine are in the source file Chem21P2.cpp. Compare the execution time for this professionally written routine with the execution time of your handwritten code. You can invoke this routine as

```
./Chem281P2 --blas
```

For reference: with *cblas_ dgemm* the multiplication took 3.71 seconds of elapsed time on 1 cores.

Note: Two scripts are provided to help you compile/run your code on Perlmutter. The first script **setmsseenv.sh** has to be sourced:

```
source setmsseenv.sh
```

The second one, **buildP2** can be used to compile and link your code.

```
./buildP2 Chem281P2.cpp
```

will produce the executable file Chem281P2.

You can compile your code on the login shell. To run your code you have to get an interactive shell through Perlmutter queueing system. I would recommend to use the command:

```
salloc -A mxxxx -N 1 -q interactive -C cpu -t 30 (max 240)
```

The parameter -t specifies the number of minutes (in this example, 30). Replace mxxxxx by m<your account id>. You can get your account id by login to iris.nerc.gov.
It may take a short time to get your interactive shell. After that you can run your code using *srun*.

```
srun -n 1 ./Chem281P2 <options>
```

Problem 4: Performance Analysis
use *perf* to profile your different code versions:

```
 srun -n 1 perf stat ./Chem281P2 --loop --threads 4
 srun -n 1 perf stat ./Chem281P2 <best set of tiles> --threads 4
 srun -n 1 perf stat ./Chem281P2 --blas
```

where best set of tiles if the best set of tiled parameters that you found in Problem 2. perf will output a number of statistics for your run. Here is a summary(not all of them) of the measurements of perf stat

```
task-clock:u  #aggregated time in millisecs used/
context-switches:u
page-faults:u
cycles:u
instructions:u  # with instruction retired per cycle. Higher is better
branch-misses:u # branch mispredictions.
seconds time elapsed (running time)
```

You can use perf to examine cache missess. For that, you can use the following perf options

```
perf -e mem\_load_retired.l1\_miss,mem\_load_retired.l1\_hit <program>
perf -e mem_load_retired_l2_hit,mem_load_retired.l2_miss <program>
```

Use perf to explain the different performance behavior between your different implementations of matrix multiplication. Write a report with your findings.