Click to show/hide PDF Layer

Lab 1

Pablo Arias[*], **Daniel Santos-Oliván**[†], **Álvaro Moure**[‡]

Instructions

In this lab, we will learn how to use OpenMP to parallelize code for shared-memory computers. We will explore the basics of OpenMP through its two main paradigms: *threads* and *tasks*. This session consists of three mathematical problems, each with several questions.

You must compile your answers and explanations into a report, detailing how you solved each problem and addressing all the questions. This lab assignment is worth **20% of the total course grade**, and the deadline for submission is **April 22 at 12:00h**.

Each group must submit a compressed file named `lab1_TxGyy.zip`, where `TxGyy` is your group identifier. A `.tar` or a `.tgz` file is also accepted (e.g., `lab1_T1G1.zip` or `lab1_T2G21.zip`). The compressed file **must** contains three folders (`1_cholesky`, `2_histogram`, and `3_argmax`) and **all** the requested files with the following structure. Additional files will not be considered and will not be penalized.

```
lab1_TxGyy.zip ─┬─ 1_cholesky ──┬─ cholesky.c
                │               ├─ cholesky.h
                │               ├─ main.c
                │               ├─ Makefile
                │               ├─ job.sh
                │               └─ output_files (.out .err)
                ├─ 2_histogram ─┬─ histogram.c
                │               ├─ random.c
                │               ├─ random.h
                │               ├─ Makefile
                │               ├─ job.sh
                │               └─ output_files (.out .err)
                ├─ 3_argmax ────┬─ argmax.c
                │               ├─ Makefile
                │               ├─ job.sh
                │               └─ output_files (.out .err)
                └─ report.pdf
```

A sample file named `lab1_TxGyy.zip` containing the reference codes has been published in Aula Global. You need to create your job scripts to perform your tests according to what is asked. The Makefiles should not be modified unless the exercise says so. Focus on the code and the work requested for each exercise.

If you have any questions, please post them on the lab class forum in Aula Global. However, do not post your code in the Forum. If you have other questions regarding the assignment that you consider cannot be posted in the Forum (e.g., personal matters or code), please contact the lab responsible person.

[*] pablo.arias@upf.edu.
[†] daniel.santos@upf.edu.
[‡] alvaro.moure@upf.edu.

Criteria

The codes will be tested and evaluated on the same cluster where you work. The maximum grade on each part will only be given to these exercises that solve in the most specific way and tackle all the functionalities and work requested. All the following criteria will be applied while reviewing your labs in the cluster.

Exercises that will not be evaluated:

- A code that does not compile.
- A code giving wrong results.
- A code that does not adhere to all the input/output requests.
- `lab1_TxGyy.zip` delivered files not structured or named as described previously.

Exercises with penalty: A code with warnings in the compilation.

1. Cholesky factorization

The Cholesky factorization (or Cholesky decomposition) is a factorization of a symmetric, positive definite matrix A into two matrices: the lower matrix L and the upper matrix U, where these names refer to the lower and upper part of the diagonal of the matrix. In the Cholesky factorization, the generated matrices are the transpose of the other, $L = U^T$, and vice versa.

In this exercise, we will have to finish the sequential version of Cholesky in `cholesky()` and then parallelize it in `cholesky_openmp()`. We will calculate U from A and then calculate L by doing the transpose of matrix U. **The matrix transpose operation has to be implemented to optimize the efficiency of L1 cache memory usage.** The formulas used to calculate the elements of U are the following, depending if they are diagonal elements or non-diagonal elements:

$$u_{ij} = \frac{a_{ji} - \sum_{k=0}^{i-1} u_{kj} u_{ki}}{u_{ii}}$$

off-diagonal elements

$$u_{ii} = \sqrt{a_{ii} - \sum_{k=0}^{i-1} u_{ki}^2}$$

diagonal elements

It is important to notice that we only need to compute the corresponding elements of the diagonal and the lower elements (for L) and upper elements (for U). For the sake of simplicity, L and U may be square matrices, and have the rest of the elements be zero.

Once calculated the Cholesky factorization we want to check that it has been calculated correctly. This is done by performing $B = LU$ and later verifying that $B = A$. Multiply matrices LU and calculate B. Matrix B will have the same size as A. After generating B, compare the elements of B and A and check that the difference between elements is less than 0.001% as follows:

$$Error[\%] = \left|\frac{B_{ij} - A_{ij}}{A_{ij}}\right| \times 100$$

In the file `cholesky.c` there are the two C functions that we will work with. We will need to first finish the sequential implementation and later parallelize all 5 steps for the OpenMP version. The steps of our functions are the following.

1. Matrix initialization for A, L, U and B (already done)
2. Compute Cholesky factorization for U
3. Calculate L from U^T: iterate only over the non-zero elements.
4. Calculate $B = LU$: matrix multiplication. Iterate only over non-zero elements of L.
5. Check if all elements of A and B have a difference smaller than 0.001%.

Here is a sample output of the code (you might have different results). Notice than `argv[1]` is the size n of the square matrix A.

```
Sequential Cholesky
Initialization: 0.160174
Cholesky: 52.790788
L=U.T: 0.051511
B=LU: 88.952372
Matrices are equal
A==B?: 0.015213

OpenMP Cholesky
Initialization: 0.161941
Cholesky : 3.670381
L=U.T: 0.003441
B=LU: 5.572723
Matrices are equal
A==B?: 0.001657
```

Report Questions 1 **Cholesky (40%)**

1. Expose your parallelization strategy to divide the work in the Cholesky algorithm and in the matrix multiplication. Justify the selection of the scheduler and chunk size and compare different schedulers with different chunk sizes and show the results.

2. Make two plots: one for the speedup of the Cholesky factorization and another for the matrix multiplication for $n = 3000$. Use 1, 2, 4, 8, and 16 cores for a strong scaling test. Plot the ideal speedup in the figures and use a logarithmic scale to print the results. Discuss the results.

2. Histogram

In this exercise, we provide a program that will fill an array with pseudo-random values, build a histogram of that array, and then compute statistics. This can be used as a simple test of the quality of a random number generator.

The code is sequential. The goal of this exercise is to parallelize it using three different methods: using critical pragma, locks, and a reduction.

The code requires no arguments.

The output must be like this. It must execute the 4 variants one after the other. This execution has been performed on a laptop. Thus, your times can be different. Remember to reinitialize the histogram after each variant.

```
4 threads
Sequential
```

```
histogram for 50 buckets of 1000000 values
ave = 20000.000000, std dev = 394.372925
in 0.003438 seconds
par with critical
histogram for 50 buckets of 1000000 values
ave = 20000.000000, std dev = 394.372925
in 0.077274 seconds
par with locks histogram for 50 buckets of 1000000 values
ave = 20000.000000, std dev = 394.372925
in 0.058147 seconds
par with reduction
histogram for 50 buckets of 1000000 values
ave = 20000.000000, std dev = 394.372925
in 0.000601 seconds
```

Report Questions 2 — Histogram (30%)

1. Explain how have you solved each of the parallelizations.
2. Explain the time differences between different parallel methods if there are any.
3. Make a speedup plot for the different parallelization methods for 1, 2, 4, 8, and 16 cores. Discuss the results.

3. Argmax

The goal in this exercise is to write a function that traverses a vector v of doubles and computes the maximum value m and the index idx_m there that maximum is locate (i.e. the argmax).

You have to write your code in a file `argmax.c`. In addition to the main function your code needs to contain the following functions:

```c
void initialize(double *v, int N) {
  for (int i = 0; i < N; i++) {
    v[i] = (1 - pow(0.5 - (double)i/(double)N, 2)) * cos(2*M_PI*100* (i - 0.5)/N);
  }
}

// computes the argmax sequentially with a for loop
void argmax_seq(double *v, int N, double *m, int *idx_m) {
}

// computes the argmax in parallel with a for loop
void argmax_par(double *v, int N, double *m, int *idx_m) {
}

// computes the argmax recursively and sequentially
void argmax_recursive(double *v, int N, double *m, int *idx_m) {
}

// computes the argmax recursively and in parallel using tasks
void argmax_recursive_tasks(double *v, int N, double *m, int *idx_m) {
}
```

The recursive function `argmax_recursive` should work as follows. For input vectors with more than K elements, the function should divide the vector in two halves, compute the max and the argmax on each half by calling itself recursively, and combine the results of each half to compute the max and argmax for the whole input vector. For input vectors with less than K elements, the function should compute the max and argmax sequentially with a loop.

The version with tasks, should package each call of the recursive function in a task.

The main function needs to do the following steps:

1. receive as command line parameters the size N of the vector v
2. allocate memory for v and initialize it using the provided function
3. call the four functions above measuring their running time, storing the outputs in variables

`seq_m`	`par_m`	`rec_m`	`task_m`
`seq_idx_m`	`par_idx_m`	`rec_idx_m`	`task_idx_m`

4. print for each of the functions a string such as this one:

```
    sequential for       argmax: seq_m=10.34, seq_idx_m=50023, time=0.2143s
    parallel   for       argmax: seq_m=10.34, seq_idx_m=50023, time=0.2143s
    sequential recursive argmax: seq_m=10.34, seq_idx_m=50023, time=0.2143s
    parallel   recursive argmax: seq_m=10.34, seq_idx_m=50023, time=0.2143s

```

Report Questions 3 — Argmax (20%)

1. Explain the recursive implementation of the argmax, and how you parallelized using tasks.
2. Run the code with $N = 1, 2, 4, 8$ processors for a vector of size $N = 1000000$ and plot the strong speed-up try with $K = 16$ and $K = 512$. Comment on the obtained results.