Assignment 2b: "Histogram w/ prime factors"

In this part of the assignment (2b), you are given a C++ program that counts the number of prime factors for each integer between 2 and 50000. The occurrences of each number of prime factors are then recorded in a histogram with 10 bins, numbered from 0 to 9. Let X represent the number of prime factors of an integer, if X is smaller or equal to 9, then the program must increment the histogram bin corresponding to X. If X is greater than 9, which is the largest bin of the histogram, the program will update the histogram bin corresponding to 9 (see **Example**).

Example: Assume the program is counting the number of prime factors of 20. The prime factorization of 20 results in 2 * 2 * 5, which means that the number of prime factors of 20 is equal to 3. Therefore, the program must increment the histogram bin that corresponds to 3.

Now, assume that the program is counting the number of prime factors of 1024. The prime factorization of 1024 results in 2^10, which means that the number of prime factors of 1024 is equal to 10. Since 10 is greater than the index of the largest bin available (which is 9), the program will increment the bin number 9.

Requirements summary:

- Use **OpenMP** to implement 2 parallel versions of the histogram program (**histogram-v2.cpp**).
- The first parallel version (**histogram-v2-naive.cpp**) should use the data-scoping and synchronization mechanisms provided by OpenMP to ensure the correctness of your parallel solution.
- The second version (**histogram-v2-best.cpp**) should be your best-effort implementation, which must overcome performance issues (**related to OpenMP**) to achieve the desired speedup (**see below**).
- The output of your parallel solutions **must match** the output produced by the sequential program that was provided. If this is not observed, your solution is considered wrong.
- Your best-effort implementation should be correct and achieve a speedup of at least **12x** (for **32 threads on ALMA**) compared to the sequential version of the code (**histogram-v2.cpp**).
- The desired speedup can be achieved by only including OpenMP directives. Therefore, avoid unnecessary algorithmic optimizations.
- Keep your code flexible by avoiding OpenMP run-time routines and hard-coded clauses. Resort to environment variables whenever possible. E.g., use the OMP_NUM_THREADS environment variable in place of the `num_threads` clause or the `omp_set_num_threads` routine.

Detailed Description

1. Naive Histogram **[Implement in histogram-v2-naive.cpp]**

At this stage, your goal is to parallelize the populate method of the histogram structure. The populate method already returns the number of prime factors corresponding to the current iteration number and increments the bins of the histogram accordingly (in a sequential manner). To parallelize this method, distribute the workload among a team of OpenMP threads, and use the synchronization and data-scoping mechanisms provided by OpenMP to avoid data-races and ensure the correction of your solution.

2. Best Effort Histogram **[Implement in histogram-v2-best.cpp]**

Implement the same functionality as above, and overcome potential performance issues related to OpenMP to achieve the desired speedup. This version should achieve a speedup of at least **12x** on a node of the ALMA cluster (for **32 threads**), while still producing the correct result.

Hint: the function that counts the number of prime factors is the most likely culprit. If you were not able to identify the performance issue that is present in your naive implementation, try measuring the execution time of each thread.

Challenge (optional): Try to implement your best-effort version by including a single (yet long) OpenMP compiler directive. This challenge is entirely optional.

3. Compiling and Running your code

To compile at home use:

```
g++ -o <executable_name> --std=c++20 -fopenmp -O2 <cpp_file_name>.cpp
```

To compile on ALMA:

```
/opt/global/gcc-11.2.0/bin/g++ -o <executable_name> --std=c++20 -fopenmp -O2
<cpp_file_name>.cpp
```

To run on ALMA use (example for 32 threads):

- `OMP_NUM_THREADS=32 srun --nodes=1 ./<executable_name>`

The expected output is the following:

```
Bins: 10, sample ceiling: 50000
0:0
1:5133
2:12110
3:12794
4:9192
5:5356
6:2823
7:1381
8:663
9:546
total: 49998

time elapsed: 1.07358 seconds.
```

4. Submission Guidelines

The program has to be submitted before the deadline on the online platform after it has passed all checks. You also need to run your code on ALMA to measure the speedup achieved.

The required speedup on ALMA is at least **12x** for 32 threads, and you should measure the speedup of your best effort solution for **2**, **4**, **8**, **16**, and **32 threads** (not on the front-end, but by using the **srun** command as shown above). The sequential execution time on ALMA is around **1.07** seconds. Once you have the results, enter them on the online platform. Ensure that both the code and speedup graph are available on the online platform. The online platform should not be used for speedup measurements.

Deadline: June 22, 2025 until 23:59 on the online platform.