K-mer counting assignment

Introduction

DNA sequencing is a fundamental technique in bioinformatics. It is used to determine the order of nucleotides in a DNA molecule. The DNA sequence is composed of four nucleotides: adenine (A), cytosine (C), guanine (G), and thymine (T). The sequence is represented as a string of characters, where each character is one of the four nucleotides.

K-mer counting is a common operation in bioinformatics. A K-mer is a substring of length K in the DNA sequence. For example, if the DNA sequence is "ACGTACGTACGT" and K=3, then the K-mers are "ACG", "CGT", "GTA", "TAC", "ACG", "CGT", "GTA", "TAC", "ACG", and "CGT". The K-mer counting operation counts the number of occurrences of each K-mer in the DNA sequence.

In this assignment, you will implement a **parallel** K-mer counting program. The program will read DNA sequences from a file, count the number of occurrences of each K-mer in the sequences, and output the results. The program should be implemented using the C++ programming language and the OpenMP/MPI parallel programming framework.

Starter Code

A simple serial implementation of a K-mer counting program is provided. You can use this code as a starting point for your parallel implementation. The code consists of the following files:

- `main.cpp` is the entry of the program.
- `logger.cpp` and `timer.cpp` provide utility functions for logging and timing.
- `dnabuffer.cpp`, `dnaseq.cpp`, and `fastaindex.cpp` provide utility classes for reading DNA sequences from a file and extracting K-mers.
- `hashfuncs.cpp` provides utility functions for hashing K-mers, which may be useful for your parallel implementation.
- `kmerops.cpp` contains the implementation of the K-mer counting operation. This is the file you will need to modify to parallelize the K-mer counting operation.

How to compile

To compile, use `make` in the project directory. The length of K-mer is specified as a compile parameter. For example, you can use

```
make K=31 -j8
```

indicating that the length of K-mer is 31. The `-j8` option specifies that the compilation should use 8 threads.

Note that K is a changable compile-time parameter. Although we most likely will use K=31 for grading, you should ensure that your parallel implementation works for any odd value of K.

How to run

To run on Perlmutter, the default environment should be enough. You can use `srun`, for example

```
srun -C cpu -N 1 -n 1 -c 128 --cpu_bind=cores ./kcount $PATH_TO_DATASET
```

or a sbatch script to run the kmer counter application. Please refer to NERSC Script Generator and Documentation for explanation to slurm parameters. Our binary application is only taking in one parameter at runtime, which is the path to the fasta dataset.

After each run, the elapsed time will be printed to stdout. A histogram of the K-mer frequency will also be printed. The histogram shows the number of K-mers that occur with a certain frequency. It serves as a simple way to verify the correctness of the K-mer counting operation.

Indexing a dataset

The current implementation requires the fasta dataset to be indexed in advance. The provided datasets are already indexed and come with a corresponding index file. However, if you want to run the kmer counter with your own dataset, you can use samtools to index it. To do this on Perlmutter, execute the following commands:

```
module load spack
spack install samtools
spack load samtools
samtools faidx $PATH_TO_YOUR_DATASET
```

Your task

Your task is to parallelize the K-mer counting operation in the `kmerops.cpp` file.

Task 1: Shared Memory Parallelization

In this task, you should use the **OpenMP** parallel programming framework to parallelize the k-mer counting code.

You are only allowed to modify the `kmerops.cpp` and `kmerops.h` file. Currently, two serial implementations, based on hash tables and sorting algorithms, are provided as starter code. You can parallelize the code in any way you see fit, but you should aim to generate **accurate** results (no estimation) and achieve good performance on CPU nodes of Perlmutter.

You should provide a `count_kmer_omp` function, which takes in a `DnaBuffer` object wrapped in a unique_ptr and returns a `KmerList` object. `DnaBuffer` consists of multiple DNA sequences, and `KmerList` is a vector of (K-mer, frequency) tuples. The function signature should be identical to the provided `count_kmer` function, i.e.

```
std::unique_ptr<KmerList>
count_kmer(const DnaBuffer& myreads)
```

Please refer to the provided code for the definition of `DnaBuffer` and `KmerList`, and related functions.

Additionally, you should provide a `run_omp.sbatch` , preferably adapted from the template generated by NERSC Script Generator , as the luanching script. You can adjust the number of threads, core binding, and other parameters to optimize the performance of your parallel implementation. The dataset path in the script should be substitued by environment var `$KCOUNT_DATASET_PATH` .

Task 2: Distributed Memory Parallelization

In task 2, you should try process-level parallelization for the parallel k-mer counter. You should use the **MPI programming model** for distributed memory parallelization. You should not modify any files other than the sbatch script and `kmerops.cpp/.h`. Initialization of MPI is already done by the framework.

You need to provide a `count_kmer_mpi` function, with the same function signature as `count_kmer`. Since we have multiple processes, the input DnaBuffer will be partitioned into multiple (roughly equal-sized) segments, and each process will be given a non-overlapping part of the DnaBuffer. This is done by the framework automatically, however please remember there is no guarantee on how the DNA sequences inputs (and K-mers) are split across processes.

In addition, since each process will return one KmerList object, you should guarantee that **every valid K-mer appears in only one of the KmerList objects**. There are no further requirements on which process should return which K-mers. The print histogram function will merge the histograms from all processes and print the final histogram.

Your MPI solution will be evaluated on **4** physical nodes. Please also provide a `run_mpi.sbatch`, which launches the kcount application with your desired parameters for MPI solution on no more than 4 nodes.

Bonus Task: Hybrid Parallelism

You are encourage to try hybrid parallelization (OpenMP + MPI) as a bonus task. The requirements are similar to task 1 and task 2. Please provide a `count_kmer_hybrid` function and a `run_hybrid.sbatch` launch script.

Hint

1. K-mer counting is not a embarrassingly parallel problem, as different DNA sequences may share identical K-mers. You should be careful when parallelizing the code. One possible solution is to use locks / atomic operations (on hash tables) to avoid race conditions. Another solution is to firstly divide the input data into several independent parts, process the parts parallely, and then merge the results from each part (if necessary). The provided `GetKmerOwner` function may be useful for this purpose.
2. Many HPC problems are more memory-bound than compute-bound. K-mer counting can be one of them, so you may want to optimize the memory access pattern to improve the performance of your parallel implementation. More compute threads may not always lead to better performance.
3. When optimizing the performance of you parallel K-mer counter, it's important to take into account the architecture of the supercomputer. Go through the NERSC's documentation on the architecture of Perlmutter, and pay special attention to inter-node connect, inter-socket configuration, and NUMA domain settings before tuning your parallel implementation.

Grading

Grade will be based on the following criteria (The timing and speedup calculation only takes into account the time spent on `count_kmer` functions):

1. Implementation: (40%+40%).
    - 3 datasets will be used to evaluate the correctness and performance of your OpenMP implementation. You will get 5 points for counting each dataset correctly and 5 points for each dataset that runs at least 3x faster than the (fastest) serial implementation. An extra 10 points will be given if all datasets are counted correctly, and at least one dataset meets the performance requirement.
    - 3 datasets will be used to evaluate the correctness and performance of your MPI implementation. You will get 5 points for counting each dataset correctly and 5 points for each dataset that runs at least 50x faster than the (fastest) serial implementation. An extra 10 points will be given if all datasets are counted correctly, and at least one dataset meets the performance requirement.
2. Report (20%). You should write a report in PDF format that describes your parallel implementation. Be sure to include the following content:
    - description of your parallel implementations
    - discussion of the parallelization strategy and optimization techniques you used or considered
    - performance evaluation of your parallel implementation, including the final speedup, the weak scaling results, and performance of your counters under different parallelization parameters
    - performance comparison between serial, OpenMP, and MPI implementations on one node
    - analysis of the performance results above
3. Bonus Task.

Submission

The code should be submitted as a zip file containing the modified `kmerops.cpp` and `kmerops.h` files, the `sbatch` launch script and the report in PDF format. Please remove all outputs to stdout/stderr in your code in case they interfere with the autograder.

Please make sure that your code can be compiled and run on Perlmutter with the provided framework.

Credit: Some of the code is taken from the ELBA application.