Reexam in HPPS

April 17–21, 2023

Preamble

This document consists of 12 pages including this preamble; make sure you have them all. Your solution is expected to consist of a *short* report in PDF format, as well as a `.zip` or `.tar.gz` archive containing your source code, such that it can be compiled immediately with `make` (i.e. include the non-modified parts of the handout as well).

You are expected to upload *two* files in total for the entire exam. The report must have the specific structure outlined in section 3.

- The exam is *strictly individual*. You are not allowed to communicate with others about the exam in any way.
- If you believe there is an ambiguity or error in the exam text, *contact one of the teachers*. If you are right, we will announce a correction. Extra time will not be granted in the event of errors, so make sure to continue working on other sections in the meantime.
- Your final grade is based on an overall evaluation of your solution, but *as a guiding principle* each task and question is weighted equally.

Make sure to read the entire text before starting your work. There are useful hints at the end.

1 Library design

The overall theme of this exam problem is to implement a small C library for calculating a histogram of a given array object. All of the necessary functions for this have been implemented sequentially. Your task is to alter them so that they can opperate in parallel, and then investigate their performance.

Concretely, in `histogram.c` you will implement a collection of C functions and datatypes declared in `histogram.h`. Each of the subtasks involves parallelising several of the functions in `histogram.c`, and then carrying out appropriate benchmarking.

These general principles apply:

- You must not modify `histogram.h`.
- The functions within `histogram.c` can be interpreted as pseudo-code for your eventual solution. By this it is meant that they implement a sequential solution, however some changes may need to be applied to make them suitable for parallelisation.
- See `histogram.h` and `array.h` for the specific types of the underlying structs.
- Test functions have been provided to help debug your solutions. These are barebones checkers, and it will be possible to pass these checks and still have an incorrect solution. They are a good *indication* of correctness however.
- It may be that you do not correctly implement parallel solutions, or that you solution is slow. You can still answer the questions in section 3 with an imperfect solution.
- **Make sure to compile your code with optimisations enabled when benchmarking.** See also section 5.1.

The specific functions you must parallelise are listed below, but first we will describe the problem of histograms that they solve.

1.1 Histograms

A histogram is a description of a distribution of values. This is done through the use of *bins*, which contain a count of how many values are contained in a given range. Although we can take a histogram of many mathematical objects, for this assignment we will only be considering one-dimensional arrays of doubles, with each element in the array being a positive floating point numbers between zero and an arbitrary whole number upper limit.

Histogram bins can also be of arbitrary range, though for this assignment we assign each number to a bin by truncating it to an integer. For example, consider the following array of 10 elements between 0 and 3:

$$[2.8, 0.6, 2.2, 1.3, 0.3, 2.6, 2.5, 2.0, 0.6, 2.0]$$

According to the histogram definition we will be using in this assignment, the histogram of the above array would have 3 bins, corresponding to the intervals $[0, 1), [1, 2), [2, 3)$, and would count the elements to produce:

$$[3, 1, 6]$$

1.1.1 `histogram_simple`

This function computes a histogram in a simple, sequential manner. It traverses the input array one element at a time, determining which bin should be updated and then updating it.

Do not alter this function. Use it as a baseline measure it in your benchmarks.

1.1.2 `histogram_parallel`

This function must be a parallel version of `histogram_simple`. Your task here is to use OpenMP to parallelise the presented code. Note that part of this task is to assess the possibility of race conditions and deadlocks within your code, and make whatever alterations are necessary to prevent them.

1.1.3 `histogram_blocked`

This function will divide the input array into blocks, with each block having its own histogram to compute. These individual blocks and histograms can then be computed in parallel with each of their results being combined to

form the final result. Your task here is to use OpenMP to parallelise the presented code. Note that as-handed-out, this code is entirely sequential and will will need adaption so that individual threads create their own histograms. Note that part of this task is to assess the possibility of race conditions and deadlocks within your code, and make whatever alterations are necessary to prevent them.

1.1.4 `histogram_multipass`

This function will make several loops through the input array (refered to as a pass), with each pass only counting certain elements. For instance the first pass through the array may only count the numbers 0-9 and so only update bins 0-9. The second pass may then only count 10-19 and update those bins, and so on until all bins are filled. Each pass can then be computed in parallel by different threads. Your task here is to use OpenMP to parallelise the presented code. Note that part of this task is to assess the possibility of race conditions and deadlocks within your code, and make whatever alterations are necessary to prevent them.

1.1.5 `histogram_blocked_multipass`

This function will combine the algorithms of the blocked and multi-pass functions. The input array will be seperated into blocks, with each block histogram being calculated via the multi-pass algorithm used in the above function. Your task here is to use OpenMP to parallelise the presented code. Note that part of this task is to assess the possibility of race conditions and deadlocks within your code, and make whatever alterations are necessary to prevent them.

1.2 Helper programs

The code handout contains a number of small programs that make use of `histogram.c`. Compile them by running `make`. All functions should work as handed-out, though some will depend on the data generated by running `make datafiles` first. You can use them for testing and benchmarking. Their behaviour is summarised below. For full details, see their source code.

The following programs are used to generate data for histogram functions.

`./random-array ELEMS MAX FILE`
: Generate a random array of `ELEMS` different elements, each between 0 and `MAX`. The resulting array is written to `FILE`.

`./print-array FILE`
: Print the array in the provided file `FILE`.

The following functions are used to test the *functional correctness* of your implementations. As handed-out, these should all pass.

`./test-simple-histogram FILE BINS RUNS`
: Using the array in file `FILE`, creates its own histogram with `BINS` bins, then calculates another using `histogram_simple` from `histogram.c`. The two arrays are then compared and any differences reported. The test is repeated `RUNS` times.

`./test-parallel-histogram FILE BINS RUNS`
: Identical to `./test-simple-histogram`, except it uses `histogram_parallel` from `histogram.c` to generate the second histogram.

`./test-blocked-histogram FILE BLOCKS BINS RUNS`
: Identical to `./test-simple-histogram`, except it uses `histogram_blocked` from `histogram.c` to generate the second histogram using `BLOCKS` different blocks.

`./test-multipass-histogram FILE PASS BINS RUNS`
: Identical to `./test-simple-histogram`, except it uses `histogram_multipass` from `histogram.c` to generate the second histogram, with each pass counting `PASS` many bins.

`./test-blocked-multi-pass-histogram FILE BINS BLOCKS PASS RUNS`

: Identical to `./test-simple-histogram`, expect it uses `histogram_blocked_multi_pass` from `histogram.c` to generate the second histogram, with each pass counting `PASS` many bins across `BLOCKS` different blocks.

The following functions are used to benchmark the different histogram functions. In these, `FILE`, `BINS`, `BLOCKS`, `PASS` and `REPEATS` are the same as in their corresponding test function. Note that these benchmarks merely call the appropriate functions, they do not necesarily have appropriate values for your investigations, and it is up to you to determine what values to use.

`./benchmark-simple-histogram FILE BINS RUNS`
: Compute mean runtime of `histogram_simple` from `histogram.c`.

`./benchmark-parallel-histogram FILE BINS RUNS`
: Compute mean runtime of `histogram_parallel` from `histogram.c`.

`./benchmark-blocked-histogram FILE BINS RUNS`
: Compute mean runtime of `histogram_blocked` from `histogram.c`.

`./benchmark-multipass-histogram FILE BINS RUNS`
: Compute mean runtime of `histogram_multipass` from `histogram.c`.

`./benchmark-blocked-multipass-histogram FILE BINS RUNS`
: Compute mean runtime of `histogram_blocked_multipass` from `histogram.c`.

You are free to modify any of the programs if you wish. This is especially true of the benchmarking programs, which you may wish to modifiy or write new ones based on them.

2 Your task

You are given sequential implementations for various histogram algorithms. Your task is to convert these to parallel implementations and then investigate their performance under differing conditions. Use parallelism (with OpenMP) as appropriate. The precise functions to parallelise are listed below, though do note you are free to call additional functions from within them if you so wish. After your implementation is done, you will be performing performance analysis (see section 3).

2.1 Task: implementation

Implement the following functions in `histogram.c`:

- `histogram_parallel(arr, bins)`, which computes a histogram of the input array, `arr`, and counts all entries between 0 and `bins`.
- `histogram_blocked(arr, bins, blocks)`, which computes a histogram of the input array, `arr`, and counts all entries between 0 and `bins`. The array is divided into `blocks` many blocks.
- `histogram_multipass(arr, bins, pass)`, which computes a histogram of the input array, `arr`, and counts all entries between 0 and `bins`. The histogram is computed in a number of passes, with each pass counting `pass` many bins.
- `histogram_blocked_multipass(arr, bins, blocks, pass)`, which computes a histogram of the input array, `arr`, and counts all entries between 0 and `bins`. The array is divided into `blocks` many blocks, and each histogram is computed in a number of passes, with each pass counting `pass` many bins.

Feel free to diverge from the provided sequential code as you wish, but your solutions will be expected to implement the provided algorithm. For example, you may wish to add additional data structures or function calls within one of the given functions but the blocked function should still ultimately be dividing the input array across several smaller workloads.

3 The structure of your report

Do not put your name in your report. The exam is supposed to be anonymous. Your report must be structured exactly as follows.

Introduction:

Briefly mention general concerns, your own estimation of the quality of your solution, whether it is fully functional, which programs you have written or modified in order to answer the questions, and possibly how to run your tests. Make sure to report the computer you are benchmarking on (in particular the core count). You should also

mention if your code passes the correctness tests from section 1.2, and if not why not.

Sections answering the following specific questions:

1. a) Does `histogram_parallel()` exhibit good temporal and/or spatial locality? Why or why not? Does it depend on `bins`?
   b) How did you parallelise `histogram_parallel()`? Mention any race conditions or deadlocks you identified and what steps you took to address these problems.
   c) Measure and show the performance of the function as you vary the size of the input data, and the amount of bins in `histogram_parallel()`. Explain briefly under what conditions this algorithm performs best.
   d) Measure and show the weak and strong scalability of `histogram_parallel()` as you vary the number of threads. Explain how you chose the datasets.
2. a) Does `histogram_blocked()` exhibit good temporal and/or spatial locality? Why or why not? Does it depend on `bins` or `blocks`?
   b) How did you parallelise `histogram_blocked()`? Mention any race conditions or deadlocks you identified and what steps you took to address these problems.
   c) Measure and show the performance of the function as you vary the size of the input data, the amount of bins, and the number of blocks in `histogram_blocked()`. Explain briefly under what conditions this algorithm performs best.
   d) Measure and show the weak and strong scalability of `histogram_blocked()` as you vary the number of threads. Explain how you chose the datasets.
3. a) Does `histogram_multipass()` exhibit good temporal and/or spatial locality? Why or why not? Does it depend on `bins` or `per_pass`?
   b) How did you parallelise `histogram_multipass()`? Mention any race conditions or deadlocks you identified and what steps you took to address these problems.

c) Measure and show the performance of the function as you vary the size of the input data, the amount of bins, and the amount of counted elements per pass in `histogram_multipass()`. Explain briefly under what conditions this algorithm performs best.

d) Measure and show the weak and strong scalability of `histogram_multipass()` as you vary the number of threads. Explain how you chose the datasets.

4. a) Does `histogram_blocked_multipass()` exhibit good temporal and/or spatial locality? Why or why not? Does it depend on `bins`, `blocks`, or `per_pass`?

b) How did you parallelise `histogram_blocked_multipass()`? Mention any race conditions or deadlocks you identified and what steps you took to address these problems.

c) Measure and show the performance of the function as you vary the size of the input data, the amount of bins, the number of blocks, and the amount of counted elements per pass in `histogram_blocked_multipass()`. Explain briefly under what conditions this algorithm performs best.

d) Measure and show the weak and strong scalability of `histogram_blocked_multipass()` as you vary the number of threads. Explain how you chose the datasets.

4 The code handout

The code handout contains the following.

- `timing.h`: Helper function for recording the passage of time. **Do not modify this file.**
- `histogram.h`: Function prototypes. **Do not modify this file.**
- `histogram.c`: Sequential function definitions that you will have to parallelise and measure.
- `generate_data.sh`: Generates a range of different arrays and write them to files for use in benchmarking. You will need to select useful values for this data.

- `benchmark-local.sh`: Calls benchmarking functions for all the histogram functions. Expects that input data already exists. You will need to select a range of useful values for these benchmarks.
- `benchmark-slurm.sh` and a variety of scripts starting `slurm_...`: Identical to `benchmark-local.sh`, but can be used on MODI to schedule benchmarks using slurm.
- `Makefile`: You may want to modify the `CFLAGS` for debugging and benchmarking. You are also free to add targets for new programs if you wish, or to modify the existing ones.
- The helper programs listed in section 1.2.

You can run `make datafiles` to produce a variety of random data files. These are not sufficient to answer the questions in section 3, but show how data generation works.

You can afterwards run `make benchmark` to run the included benchmark programs. Again, these are not sufficient to answer the questions in section 3, but can be used for inspiration.

Feel free to add more programs for benchmarking if you wish, or to modify the existing ones.

5 Additional Help

5.1 General advice

- Make sure that the code you hand in compiles. If some of your code is unfinished and does not compile, *comment it out* or remove it. The code in the handout is sequential as does compile. Do not hand in something that is less functional than what you have been given.
- If may be a good idea to switch between implementation and report writing. If you find some part of the implementation difficult, write report sections for the implementation parts you have already done. A good report on incomplete work is *probably* better than a incomplete report on good work.

- It is up to you to decide on proper workloads (i.e. input sizes) that can provide good answers to the questions asked in section 3. Make sure to report the workloads you use. As a hint, each loop of the sequential histogram is very quick, so you are unlikely to see parallel speedups unless you have very many elements, or very many bins (potenially even millions of each).
- You don't *have* to show your speedup results as graphs, although it is preferable. If you are short on time, tables with numbers will do.
- Use `OMP_NUM_THREADS` to change how many threads are used when running OpenMP programs.
- The Makefile contains two sets of `CFLAGS`: one that is good for debugging, and one that is good for benchmarking. You can uncomment the ones you wish to use. Remember to recompile everything afterwards (`make -B`).
- Scripts have been provided to get your code running on MODI's slurm system. Note that this is not essential, but it would give you access to a 32 core machine that may make for more interesting results.
- All else being equal, **a short report is a good report**.

5.2 OpenMP clauses

You are expected to use OpenMP as it was taught during the lectures for this course. However, due to the open ended nature of the programming tasks, some additional clauses have been highlighted here that you *may* wish to use. These are not essential, and it is perfectly possible to achieve a high grade without using them. They are however handy shortcuts for certain parallel designs.

5.2.1 `atomic`

The `atmoic` clause is used within an omp parallel loop to tell the compiler that the following line cannot be interupted, in the same manner as if a shared lock/mutex was placed around the line. It is used like so:

```
int count = 0;
#pragma omp parallel for
for (int i=0; i<N; i++) {
  #pragma omp atomic
  count++;
}
```

Do note that the above example is a correct, but inefficient implementation as we have defined many parallel loops, but then only run the content sequentially. You can read more at:

https://www.openmp.org/spec-html/5.0/openmpsu95.html

5.2.2 `critical`

The `critical` clause is very similar to `atomic`, except it can be used to surround a scope of arbitrary length. It is used like so:

```
int count = 0;
#pragma omp parallel for
for (int i=0; i<N; i++) {
  #pragma omp critical
  {
    count++;
  }
}
```

The above code is essentially the same as the `atomic` example, except that many different lines could be included within the critical section if required, whilst `atomic` will only ever lock a single line. You can read more at:

https://www.openmp.org/spec-html/5.0/openmpsu89.html