High Performance Programming and Systems 2024

Exam Problem

Synopsis: Implementing histograms and processing file formats with irregular structure.

Preamble

This document consists of 17 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 7.

Make sure to read the entire text before starting your work. There are useful hints at the end. You should also carefully read the code handout, including the header files. They contain normative instructions on what un-implemented functions are supposed to do.

The following rules apply to the exam.

- The exam is *strictly individual*. You are not allowed to communicate with others about the exam in any way.
- Do not share or discuss your solution with anyone else until the exam is finished. Note that some students have received a timeline extension. Do not discuss the exam until an announcement has been made on Absalon that the exam is over.
- Posting your solution publicly, including in public Git repositories, is in violation of university rules on plagiarism.
- If you believe there is an error or inconsistency in the exam text, *contact one of the teachers*. If you are right, we will announce a correction.

- The specification may have some intentional ambiguities in order for you to demonstrate your problem solving skills. These are not errors. Your report should state how you resolved them.
- Your solution to the exam problem is evaluated holistically. You are graded based on how well you demonstrate your mastery of the learning goals stated in the course description, including your ability to write correct C programs with appropriate error detection. You are also evaluated based on the elegance and style of your solution, including compiler warnings, inconsistent indentation, whether you include unnecessary code or files in your submission, and so on.

1 Introduction

Histograms are an important building block in data analysis. They are commonly used in the form of a visual charts, but the underlying computational idea is to partition and count data, possibly following some form of discretisation. In this sense, a histogram with k bins can be seen as an array of k numbers. You will be constructing such histogram arrays in this exam. To keep the scope of the exam reasonable, you will be working with somewhat contrived data, and you will not be using histograms to solve any actual domain problem.

The exam is partitioned into several tasks, listed in section 5. Some of the tasks have dependencies on each other, which we will note explicitly. If you find yourself struggling with a task, consider attempting another (non-dependent) task. Before we can specify the exact tasks, we will need to define a number of concepts, formats, and utility programs.

2 Histograms

2.1 Counting histogram

A *counting histogram* counts the number of occurrences of each unique element in some collection. Formally, A k-bin counting histogram of a set of n samples X is a k-element array H where

$$H[j] = \#\{x_i \mid x_i \in X \wedge x_i = j\} \tag{1}$$

and $\#S$ produces the number of elements in set S. Note that elements of X that are outside the bounds of H are ignored.

2.2 All-pairs distances histogram

An *all-pairs distances counting histogram* (henceforth just *distance histogram* computes a counting histogram of all the distances between different points in a collection of points. For our purposes, the points will be in two-dimensional space, and the distance will be Euclidean rounded down to the nearest integer. Formally, a k-bin distance histogram of a set of n points P is a k-element array H where

$$H[j] = \#\{(a, b) \mid p_a, p_b \in P \wedge p_a \neq p_b \wedge j = \lfloor d(p_a, p_b) \rfloor\} \tag{2}$$

and $d(a, b)$ is the Euclidean distance function given by

$$d(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2} \tag{3}$$

Again, note that distances that are equal to or greater than k are ignored.

3 File formats

Apart from computing histograms, you will also be reading and writing various file formats. These are documented below.

3.1 The *ints* file format

An *ints* file stores a sequence of integers and is represented as follows.

1. A little-endian 32-bit integer n.
2. A sequence of n little-endian 32-bit integers.

An *ints* file takes up $4(n + 1)$ bytes. An *ints* file storing k integers is used to represent a k-bin histogram.

3.2 The *points* file format

A *points* file stores a sequence of points in two-dimensional space and is represented as follows.

1. A little-endian 32-bit integer n.
2. A sequence of $2n$ double-precision numbers, with each pair representing a point in two-dimensional space, such that the x coordinate is the first number of each pair.

Thus, n denotes the number of points in the *points* file. A *points* file takes up $16n + 4$ bytes.

3.3 The *mapping* file format

A *mapping* file contains associations between ASCII strings[1] and integers. The idea is that each string is assigned a unique integer code that is used while performing data processing, and at the end the integer can be converted back into the original string. A *mapping* file is represented as follows.

1. A sequence of any number of *associations*, where an association is represented as follows:
   a) A little-endian 32-bit integer n.
   b) A sequence of n bytes s, called the *string*.
   c) A little-endian 32-bit integer x, called a *code*.

Each association identifies a string s with a (supposedly) unique code x. A mapping file is illegal if multiple strings are mapped to the same code, of if the same string occurs in multiple associations.

In contrast to *ints* and *points* files, a *mapping* file does not have a header denoting the number of associations - the file simply ends when there are no more.

3.4 The *lines* file format

A *lines* file is a file comprising zero or more lines, each of which is terminated by a newline character (`'\n'`, ASCII 10). That is, a *lines* file is a standard (Unix) text file.

4 Utility programs

You are given various utility programs for running the functions you develop, as well as reading and writing data files. A few of these programs will work immediately, but most will require you to solve one or more tasks before they become operational. When you need to modify

[1] This is strictly not required by the file format—any byte sequence is allowed as a string. However, you may assume that the strings are always human-readable ASCII strings.

the source code for these programs, this will be specified explicitly in a task.

4.1 `gen-ints`

The command

`./gen-ints` *n k s f*

produces an *ints* file *f* containing *n* non-negative integers $0, s, 2s, \ldots$, modulo *k*. This is useful for producing input to counting histograms. The `gen-ints` program is operational in the code handout.

4.2 `ints2text`

The command

`./ints2text` *f*

reads the *ints* file *f* and prints the integers to stdout, one per line. The `ints2text` program is operational in the code handout.

4.3 `gen-points`

The command

`./gen-points` *n s f*

produces a *points* file containing *n* points roughly organised in a square grid, with each point separated from its nearest neighbours by the distance *s* (which may be a decimal number). The `gen-points` program becomes operational after Task C has been solved.

4.4 `points2text`

The command

`./points2text` *f*

reads the *points* file *f* and prints the points to stdout, one per line. The `points2text` program becomes operational after Task C has been solved.

4.5 run-histogram

The command

`./run-histogram` *r a k samplesf histf*

produces a *k*-bin counting histogram of the *ints* file *samplesf* and stores the resulting histogram as an *ints* file *histf*. The histogram is computed *r* times redundantly, and the average runtime is printed to stdout. The *a* option indicates which *algorithm* to use for computing the histogram. In the code handout, no algorithms are yet operational. You will implement algorithm `sequential` in Task A and algorithms `parallel_bins` and `parallel_samples` in Task B.

4.6 run-dist-histogram

The command

`./run-dist-histogram` *r a k pointsf histf*

produces a *k*-bin distance histogram of the *points* file *pointsf* and stores the resulting histogram as an *ints* file *histf*. The histogram is computed *r* times redundantly, and the average runtime is printed to stdout. The *a* option indicates which *algorithm* to use for computing the histogram. In the code handout, no algorithms are yet operational. You will implement algorithm `sequential` in Task D and `parallel` in task E.

4.7 strings2mapping

The command

`./strings2mapping` *stringsf outf*

reads the *lines* file *stringsf* line-by-line and produces a *mapping* file named *outf*, which must contain an association for each distinct line in *stringsf*. Each association *must not* contain a newline character, and *must* have a distinct code. The codes must start at 0 and be consecutive. You will implement this program in Task G.

4.8 `strings2ints`

The command

`./strings2ints` *mappingf stringsf intsf*

uses the *mappings* file *mappingf* to convert each line in the *lines* file *stringsf* to its corresponding code and writes an *ints* file named *intsf* containing those codes. The number of integers stored in *intsf* is therefore equal to the number of lines in *stringsf*. If a line in *stringsf* has no association in *mappingf*, `strings2ints` must terminate with an error message and a nonzero exit code. You will implement this program in Task H.

5 Tasks

Read the comments in the code handout for details on how to implement the functions specified in the tasks below.

5.1 Task A: Sequential counting histograms

Implement the function

`histogram_sequential()`

in `histogram.c`. This task depends on no other tasks.

5.2 Task B: Parallel counting histograms

Implement the functions

`histogram_parallel_bins()`

and

`histogram_parallel_samples()`

in `histogram.c`. This task depends on no other tasks.

5.3 Task C: Reading and writing *points* files

Implement the functions

`read_points()`

and

`write_points()`

in `util.c`. This task depends on no other tasks.

5.4 Task D: Sequential distance histograms

Implement the function

`histogram_sequential()`

in `dist-histogram.c`. This task depends on task C.

5.5 Task E: Parallel distance histograms

Implement the functions

`histogram_parallel()`

in `dist-histogram.c`. This task depends on task C.

5.6 Task F: Reading *lines* files

Implement the function

`read_lines()`

in `util.c`. This task depends on no other tasks.

5.7 Task G: `strings2mapping`

Finish the implementation of `strings2mapping.c` to behave as specified in section 4.7. This task depends on task F.

5.8 Task H: `strings2ints`

Finish the implementation of `strings2ints.c` to behave as specified in section 4.8. This task depends on task F.

6 Code handout

You are allowed to modify the following files in the code handout. Do not modify any files not listed below. You are allowed to add new files, although it is not necessary.

- `util.c`
- `histogram.c`
- `dist-histogram.c`
- `strings2mapping.c`
- `strings2ints.c`
- `Makefile` (modifying this file is not required to solve any of the tasks).

7 Your Report

Your report should be no more than ten pages in length and must be structured exactly as follows:

Introduction: Briefly mention very general concerns, any ambiguities in the exam text and how you resolved them, and your own estimation of the quality of your solution. Briefly mention whether your solution is functional, which test cases cover its functionality, which test cases it fails for (if any), and what you think might be wrong. Do not include any information also covered by the specific questions below.

A section answering the following numbered questions:

1. The following questions pertain to `util.c`.
   a) In `read_lines()`, why is it challenging that lines can be of any length, and that any number of lines can be present in a file, and how do you handle it?
   b) Is any file a valid *lines* file?
   c) How can `read_lines()` fail?
2. The following questions pertain to `histogram.c`.
   a) Do the memory accessed performed by `histogram_sequential()` exhibit good spatial or temporal locality? Why or why not?
   b) Measure and show the weak and strong scaling of `histogram_parallel_bins()` and `histogram_parallel_samples()` as you vary the number of threads. Explain how you chose the datasets and why one function may or may not scale better than the other.
   c) Is it plausible that two inputs, both of size n and for some fixed k, but containing different values, can differ in how long it takes to compute their histograms, and if so, what might such inputs look like, and what is the reason one is slower than the other?
3. The following questions pertain to `dist-histogram.c`.

a) Do distance histograms exhibit worse or better locality than counting histograms?
b) How did you parallelise `histogram_parallel()`, and why?
c) Measure and show the weak and strong scaling of `histogram_parallel()` as you vary the number of threads. Explain how you chose the datasets.

4. The following questions pertain to `strings2mapping.c` and `strings2ints.c`.
   a) Why might it be more difficult to write C programs that use mapping files, if we did not assume that strings are human-readable ASCII strings, but could contain arbitrary bytes?

Your answers must be as precise and concrete as possible, with reference to specific programming techniques and/or code snippets when applicable. All else being equal, **a short report is a good report**.

A Examples

The following examples demonstrate file formats and the results of running utility programs, *assuming* the relevant tasks have been implemented correctly.

1 gen-ints

The command

```
$ ./gen-ints 10 5 2 n10_k5_s2.ints
```

produces a file `n10_k5_s2.ints` that contains the following bytes:

```
0a 00 00 00 00 00 00 00 02 00 00 00 04 00 00 00
01 00 00 00 03 00 00 00 00 00 00 00 02 00 00 00
04 00 00 00 01 00 00 00 03 00 00 00
```

2 ints2text

Suppose `n10_k5_s2.ints` is as above. We can show its contents in human-readable form as follows:

```
$ ./ints2text n10_k5_s2.ints
0
2
4
1
3
0
2
4
1
3
```

3 run-histogram

Suppose `n10_k5_s2.ints` is as above. The command

```
$ ./run-histogram 1 sequential 5 n10_k5_s2.ints n10_k5_s2.H
```

produces a file `n10_k5_s2.H` that contains the following bytes:

```
05 00 00 00 02 00 00 00 02 00 00 00 02 00 00 00
02 00 00 00 02 00 00 00
```

4 gen-points

The command

```
$ ./gen-points 10 1.4 n10_s1.4.points
```

produces a file `n10_s1.4.points` that contains the following bytes:

```
0a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 66 66 66 66
66 66 f6 3f 00 00 00 00 00 00 00 00 66 66 66 66
66 66 06 40 00 00 00 00 00 00 00 00 cc cc cc cc
cc cc 10 40 66 66 66 66 66 66 f6 3f 00 00 00 00
00 00 00 00 66 66 66 66 66 66 f6 3f 66 66 66 66
66 66 f6 3f 66 66 66 66 66 66 f6 3f 66 66 66 66
66 66 06 40 66 66 66 66 66 66 f6 3f cc cc cc cc
cc cc 10 40 66 66 66 66 66 66 06 40 00 00 00 00
00 00 00 00 66 66 66 66 66 66 06 40 66 66 66 66
66 66 f6 3f
```

5 points2text

Suppose `n10_s1.4.points` is as above. We can show its contents in human-readable form as follows:

```
$ ./points2text n10_s1.4.points
(0.000000,0.000000)
(0.000000,1.400000)
(0.000000,2.800000)
(0.000000,4.200000)
(1.400000,0.000000)
(1.400000,1.400000)
```

```
(1.400000,2.800000)
(1.400000,4.200000)
(2.800000,0.000000)
(2.800000,1.400000)
```

6 run-dist-histogram

Suppose `n10_s1.4.points` is as above. Then

```
$ ./run-dist-histogram 1 sequential 5 n10_s1.4.points n10_s1.4_k5.H
```

produces a file `n10_s1.4_k5.H` with the following contents:

```
05 00 00 00 00 00 00 00 2c 00 00 00 0c 00 00 00
16 00 00 00 0a 00 00 00
```

7 strings2mapping

Suppose `lines.txt` contains

```
foo
bar
baz
foo
foo
baz
```

then the command

```
$ ./strings2mapping lines.txt lines.mapping
```

produces a file `lines.mapping` that contains the following bytes:

```
03 00 00 00 66 6f 6f 00 00 00 00 03 00 00 00 62
61 72 01 00 00 00 03 00 00 00 62 61 7a 02 00 00
00
```

8 strings2ints

Suppose `lines.txt` and `lines.mapping` are as above. If we run

```
$ ./strings2ints lines.mapping lines.txt lines.ints
```

we produce a file `lines.ints` that contains the following bytes:

```
06 00 00 00 00 00 00 00 01 00 00 00 02 00 00 00
00 00 00 00 00 00 00 00 02 00 00 00
```