HPPS 2024

Assignment Four

Locality optimisations

Due: 2024-12-24

Synopsis: Perform locality optimisations and analysis of cache behaviour.

1 Introduction

This assignment is meant to test three competences:

1. Writing modular C programs comprising multiple files.
2. Handling dynamic memory and pointers, particularly avoiding memory leaks and invalid memory accesses.
3. Analysing cache behaviour of code and performing cache optimisations.

Keep this in mind when working on the code, and particularly when choosing what to focus on for your report. Make sure to read the entire text before starting. In particular, read section 6 to understand the code handout.

2 Purpose

You will be writing a series of programs to perform *queries* on a real-world dataset. The dataset in question is OpenStreetMap's dataset of "place names" for the whole planet. The full dataset is available at `https://osmnames.org/download/`, and contains 21 million records. Each record contains many fields, representing information about a place somewhere on Earth. For this assignment, the important fields are:

`osm_id`: a distinct number ("ID") identifying a place (e.g. "3546485").

`lon`, `lat`: the longtitude and latitude of the place (e.g. "12.5604108", "55.7026665").

`name`: the canonical human-readable name of the place (e.g. "Universitetsparken").

Decompression with `gunzip` yields a file `planet-latest_geonames.tsv` that contains an initial line labeling the record fields, and then one line per record, with tab-seperated fields (`'\t'`). The handout contains functions for reading the dataset into memory in the `record.h` and `record.c` files.

The full dataset is large and somewhat unwieldy. For testing, it can be useful to extract a smaller subset. For example, we can extract the first 20000 records by running the following Unix command[1]:

```
$ head -n 20000 planet-latest_geonames.tsv
  > 20000records.tsv
```

You will be writing various programs for performing two kinds of queries against the dataset:

1. Given an integer, find the place with the corresponding `osm_id`, if any.

2. Given longtitude and latitude, find the closest place.

Each program will be *interactive*: after loading the dataset (given by a filename), the program will run a loop that continuously reads and processes queries from the user. This situation is common in practice, and has the interesting property that it can be advantageus to perform costly *preprocessing* to produce an *index* of the data in order to respond more efficiently to queries. This is because the data will only be loaded once at startup, from which we might service millions of queries.

The two kinds of queries are covered by two subtasks. Apart from your code, you are also expected to write a short report (no more than 5 pages).

For some tasks, the code handout will contain files you can modify. For others, you will need to create new files from scratch. Feel free to base them on the handed out ones. Remember to also update the Makefile!

[1] Actually this extracts only 19999 records, as the first line is metadata.

3 Querying by ID

In the following subtasks you will be writing programs that given an ID, prints the name of the place with that ID, if any.

3.1 `id_query_naive.c`: Brute-force querying

The first part of this subtask is finishing the program `id_query_naive.c`, which makes use of the library code in `id_query.h`/`id_query.c`. Read this code *carefully* (particularly the header file) to understand how to use it. The `id_query_naive.c` program should operate by performing a linear search through all records for the desired ID. Usage example:

```
$ ./id_query_naive planet-latest_geonames.tsv
Reading records: 32070ms
Building index: 0ms
45
45: Douglas Road -1.820557 52.554324
Query time: 72697us
1337
1337: not found
Query time: 118261us
```

The next two subtasks involve creating faster versions of this program. Do *not* optimise `id_query_naive.c` itself—keep it around as a "known correct" implementation so you can see whether your faster versions are still correct, and so you can easily measure the impact of your changes.

3.2 `id_query_indexed.c`: Querying an index

Write a program `id_query_indexed.c` that, instead of searching an array of **`struct`** `record` values, searches an array of the following:

```
struct index_record {
  int64_t osm_id;
  const struct record *record;
};
```

E.g. we might define the following structs and functions:

```
struct indexed_data {
```

```
  struct index_record *irs;
  int n;
};

struct indexed_data*
mk_indexed(struct record* rs, int n);
void
free_indexed(struct indexed_data* data);
const struct record*
lookup_indexed(struct indexed_data *data, int64_t needle);
```

3.3 `id_query_binsort.c`: Querying a sorted index

Sort the array of records by their `osm_id` (use `qsort()`), and use binary search when looking up records. You may use the standard `bsearch()` function if you wish, but it is not required.

3.4 Bonus (optional)

Feel free to write more programs if you wish. Maybe you can think of optimisations I didn't. One fun idea is to use an Eytzinger layout for the sorted records, as discussed at `https://algorithmica.org/en/eytzinger`.

4 Querying by coordinate

In the following subtasks you will be writing programs that given a longtitude/latitude coordinate, prints the name of the closest place in the dataset.

For the purpose of this assignment, we consider the distance between two coordinates to be given by their Euclidean distance

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

although this is actually nonsense for longtitude/latitude coordinates since they occur on a sphere. You'll have plenty of other details to worry about, so we'll pretend the Earth is a rectangle for now.

4.1 `coord_query_naive.c`: Naive brute-force querying

The first part of this subtask is finishing the program `coord_query_naive.c`, which makes use of the library code in `coord_query.h`/`coord_query.c`. Read this code *carefully* (particularly the header file) to understand how to use it. The `coord_query_naive.c` program should operate by performing a linear search through all records and pick the record whose location is closest to the query coordinates. Usage example:

```
$ ./coord_query_naive planet-latest_geonames.tsv
Reading records: 30823ms
Building index: 0ms
10 10
(10.000000,10.000000): Mun-Munsal (9.962352,10.011060)
Query time: 178534us
45 45
(45.000000,45.000000): Levokumsky District (45.085875,45.022457)
Query time: 172391us
```

4.2 Bonus (optional)

Use the *k*-d-tree you developed for A3 to write a program that uses a *k*-d-tree to accelerate the point lookup.

5 Your Report

You are expected to comment on the *interesting* details of your implementation. You are *not* expected to give a line-by-line walkthrough of your code. Most importantly, you are expected to reflect on the *quality* of your code:

- Do you think it is functionally correct? Why or why not?
- Is there some improvement you'd have liked to make, but didn't have the time?

It is more important to be aware of the strengths or shortcomings of your solution, than it is to have a complete solution.

5.1 The structure of your report

Your report should be structured exactly as follows:

Introduction: Briefly mention very general concerns, your own estimation of the quality of your solution, and possibly how to run your tests.

A section answering the following specific questions:

1. Evaluate the temporal and spatial locality of the four programs you have implemented.
2. Do your programs corrupt memory? Do they leak memory? How do you know?
3. How confident are you that your programs are correct?
4. Benchmark your programs on various workloads and explain the differences. How did you pick the workloads?

All else being equal, **a short report is a good report**.

6 Handout/Submission

Alongside this PDF file there is a tarball containing a framework for the assignment.

```
$ tar xvf a4-handout.tar.gz
```

You will now find a `a4-handout` directory containing the following:

`.gitignore`

A suitable `.gitignore` file, should you choose to use Git.

`record.h`, `record.c`

Library code for reading the OpenStreetMap dataset. You *must* read and understand the header file, but reading the implementation file is optional.

id_query_naive.c

A skeleton file for the first subtask.

Makefile

The file that configures your `make` program. As a special treat,

```
$ make planet-latest-geonames.tsv
```

will download the dataset for you.

random_ids.c

A program for generating random valid IDs corresponding to some data file. Compile with `make random_ids` and run as e.g.:

```
./random_ids planet-latest_geonames.tsv \
  | head -n 1000000 > 1M_ids
```

This is useful for generating test data for the first task.

id_query.h, id_query.c

A reusable implementation of the loop that reads ID queries from the user and looks them up in the index.

coord_query.h, coord_query.c

A reusable implementation of the loop that reads coordinate queries from the user and looks them up in the index.

7 Deliverables for This Assignment

You must submit the following items:

- A single PDF file, A4 size, no more than 5 pages, describing each item from report section above.
- A single zip/tar.gz file with all code relevant to the implementation, including at least all the files from the handout. For this assignment it is not necessary to add additional files.

Remember to follow the general assignment rules listed on the course homepage.