Project 1: Multi-thread Programming

This project uses a (oversimplified) machine learning inference system to allow you to put some basic OS concepts into practice and learn how system-level optimizations help improve performance.

Suppose you are hired by a company to improve the efficiency of the optimization process of a recommender system. This recommender system represents the users and items (i.e. goods offered by the E-commerce platform) as vectors, and uses these embedding vectors to calculate the semantic similarities between user-user, user-item and item-item pairs. Since you are a system expert, the company hopes you can improve the efficiency of the embedding maintenance of the recommender system.

Introduction

The AI algorithm engineers of the company explain the basics of the setup to you, as follows.

Embedding Vectors

Embedding vectors represent the entities (either a user or an item). These vectors are stored as rows in an embedding matrix. These embedding vectors have a property that the product of two embedding vectors (maybe after some processing) indicates the similarity of the two corresponding entities.

```
...
# The following line illustrates the i-th row of the embedding matrix
-0.6365,-0.1280,-0.7883,0.9397,0.2103,0.3158,0.9385,-0.0953,-0.4800,-0.7988
...
```

Operations on Embedding Vectors

Three important operations on the embedding vectors are

1. Initializing (`init_embedding`) the vectors on cold start (`cold_start`) ;
2. Updating the vectors after observing user activities (`update`).
3. Recommend an item to a user from a list of items.

The embedding initialization is difficult, as there is few information about a new entity. For a new user, a simple method is to randomly initialize the embedding and run some cold start procedure (e.g. recommending some popular item) to probe the user's interest. For a new item, we may expose them to highly active users.

After observing some user activities, we gather information about the users' interest and items' popularity. To encode this information into the embeddings, we need to update the embedding vectors with some optimization method (e.g. gradient descent) where we pull in the distance between similar entities and push away those dissimilar entities.

Embedding Holder

In this project, we utilize `EmbeddingHolder` (a collection of embeddings) as the database to store and update the company's data. Initially, we read data from `*.in` to create two basic `EmbeddingHolder` for **users** and **items** respectively; our task is to update this database according to the incoming `Instruction`s.

Codebase

The project directory looks like this:

```
.
├── benchmark.cc
├── BUILD
├── format.cc
├── lib
│   ├── BUILD
│   ├── embedding.cc
│   ├── embedding.h
│   ├── embedding_test.cc
│   ├── instruction.cc
│   ├── instruction.h
│   ├── model.cc
│   ├── model.h
│   ├── model_test.cc
│   ├── utils.cc
│   ├── utils.h
│   └── util_test.cc
├── q0.cc
├── q1.cc
├── q2.cc
├── q3.cc
├── README.md
└── WORKSPACE

```

The source files in `lib/` implements the basic interfaces of the system. The `lib/embedding.h` contains the main interface of the embedding matrix data holder. The mentioned operations on the embedding matrix are implemented in `lib/model.h`. `Instruction` is in `lib/instruction.h`. The utilities are in `lib/utils.h`. You find these files belongs to the AI team and you should not modify them, and do your optimization all from the system level.

The entrance of the program is in `q*.cc`. You can modify everything in this project, except those tests in `lib`. You should ensure that after your modification, it still passes the tests in `lib`.

The project builds with the bazel build system, as we discussed in the discussion session. You should install bazel following the online documents and read the BUILD and WORKSPACE files carefully to understand the dependencies and code structure.

Unit testing

Writing unit tests is an essential way to build and optimize system programs and thus we would require that you include unit tests for all major functions you write for your code.

We have provided test cases in the libraries for you as examples. Please do not modify these unit tests. You can modify library code, and add your own test cases to the library test, but make sure that after your modification, our provided test cases still pass.

Format Checking

In this project we test the standard output stream for final grading, so make sure that you output the correct answer as you expect. We provide a format checking script in `format.cc`. You can modify it to test all output results of your submission. Run the format checking script after building it like this:

```
bazel-bin/q0 |grep [OUTPUT] > format.out && bazel-bin/format
```

Instruction Files

The workload of the company comes as `Instruction`s with `order` and `payload` fields. Both fields are integers. In this project, the `Instruction.order` is either 0, 1 or 2, indicating the task of "init", "update" or "recommend". The workflow of different task types are different as described bellow.

- For **"init"**, you should first create a new embedding. The payload is a list of existing embedding indices in the input matrix (`EmbeddingHolder`) that are used for cold start (downstream applications may use them for interest probing). You should add this new embedding into the `EmbeddingHolder` using `append`. Then you need to call `cold_start` for each of the embedding indices. Note that after `cold_start`, **only user embeddings are updated** (see `q0.cc` for example).
- For **"update"**, the payload is `[user_idx, item_idx, label, (iter_idx)]`. You should invoke the `calc_gradient` on these two embedding vectors from `lib/model.h` and then call `update_embedding` from `lib/embedding.h` **on both the user and the items** (see `project/embedding.cc:run_q0` for example). The `iter_idx` indicates the epoch number of current update, see Task-3 for information.
- For **"recommend"**, the the payload is `[user_idx, iter_idx, item_idx1, item_idx2, item_idx3, item_idx4, ...]`. You should invoke the `recommend` on these embedding vectors from `lib/model.h` and then output the result.

Note that NO provided functions are thread safe on their own.

Other coding instructions

For the following tasks, you should minimize duplicate code. If there are code sharing among the following tasks, you should extract as much shared code as possible into separate `.h` and `.cc` library files, but not put everything into a single `.cc` file. You should create your own `_test.cc` file for your library code. 20% of this project grade are reserved for good coding style (including the completeness of your own test cases).

We recommend you to use the standard C++11 Threading to implement multi-threading. However, please feel free to use any other C++ threading library (such as `pthread`, or Intel's TBB to do the following task). If you need external libraries, please include them in the `WORKSPACE` file and make them to install and build automatically. The TAs are not obliged to manually install any libraries on the grading machine.

Submission instructions

You should create a `.diff` file of your latest commit from the latest commit of the main project on learn.tsinghua, as follows:

```
git diff YOUR_COMMIT_SHA1 MAIN_REPO_COMMIT_SHA1 >
${STUDENT_ID_1}_${STUDENT_ID_2}.diff
```

To test whether your `.diff` works, clone a new repo and call `git apply ${STUDENT_ID_1}_${STUDENT_ID_2}.diff` and see if your code still works.

Task-0: Play with bazel build, the code and sanity check

In this task, you need to understand the project structure, read the `q0.cc` code and the `project/BUILD` files, install bazel, build and run the provided sample program and tests. This task is not included for grading. Just provided as an example for you to read. If you need more information on the Bazel build system, please refer to

> ***NOTE:*** You can take this task's code as an example and benchmark for following tasks, but do not copy and paste this unoptimized code! Otherwise you will get penalized. Task-0 is not included in the final grading.

> ***NOTE:*** Your code is not required to output exactly the same answer as `q0.cc`. You just need to guarantee your code is thread-safe.

Task-1: Supporting concurrent users

At the beginning, the recommender system is new and has very few items or users. Fortunately, the company attracts more and more users. Thus, we need to modify the system so that when multiple users come in at the same time, and send `Instruction`s concurrently the system could behave correctly and efficiently.

ToDo:

In this task, your job is to process an `Instruction` set of "init" and "update". Considering that `cal_gradient` is time-consuming (10s of seconds running time, mostly waiting for I/O), you should think of handling multiple `Instruction`s concurrently. Design a synchronization mechanism using locks to guarantee that your codes are thread-safe. Output your final `EmbeddingHolder` of **1) users and 2) items** using the provided `EmbeddingHolder::write_to_stdout()`.

Again, note that NO provided functions are thread safe on their own. Feel free to modify these functions in the `lib/` directory, but do not modify existing test cases there (you can add your own test cases).

Grading:

You will be graded by both the correctness and efficiency of your calculation under heavy and arbitrary mixtures of incoming `Instruction`s.

> ***NOTE:*** In this task, a single `Instruction` only runs in a single thread (i.e. no internal parallelism within an `Instruction`).

> ***NOTE:*** The correctness means thread-safety. We allow any order of updates, as long as it is thread-safe.

Task-2: Accelerate "Init" task through concurrency

Now the engineers find that it is still too slow to conduct each "init" task, as each of the task need to read multiple embeddings to perform the init. They hope to read and use these embeddings concurrently so shorten the time required to call a single "init". We can further speed up the process by building an internally-concurrent "init" function.

ToDo:

In this task, your job is to process an `Instruction` set of "init" and "update". You can start with your codes in Task-1. Try to conduct multiple `cold_start` in parallel and update the newly initialized embedding collectively without violating **thread-safety**. Output your final `EmbeddingHolder` of **1) users and 2) items** using the provided `EmbeddingHolder::write_to_stdout()` function. Note that you still need to support multiple concurrent users.

Grading:

You will be graded by the correctness and efficiency of your calculation.

Task-3: Supporting incremental embedding updates

Now the recommender system has many users and items. Everyday, the system observes user activities and use them to update the embedding matrix for better recommendation performance. In this task, you will need to implement the updating process of the embedding matrix. The algorithm engineers of the company come up with a "genius" optimization algorithm. This algorithm requires the optimizer to update the embeddings iteratively, and they call each iteration an *epoch*. This innovative epoch-based optimization differs from existing methods in that the calculation of later epochs depends on the results of earlier epochs. In this task, your job is to support this epoch-based update.

ToDo:

In this task, you should deal with the data dependency among `Instruction` s. The input `Instruction` set contains both "init" and "update" tasks. The `iter_idx`s are in ascending order, and takes nonnegative integer values starting from 0. One `update` instruction should be executed only after all `update` instructions with smaller `iter_idx`s are completed to guarantee the dependency correctness (using data from the last iteration as input). You can start with your codes in Task-2 and try to make all update execution parallel and thread-safety. Output your final `EmbeddingHolder` of **1) users and 2) items** using `EmbeddingHolder::write_to_stdout()`.

Grading:

You will be graded by the correctness and efficiency of your calculation.

Task-4: Doing recommendation while updating the embedding

Except the maintenance of the recommender system, the company also needs to generate recommendations for each user using the existing database. The recommendation should depend on the current version of data and not disturb the maintenance of the recommender system. Note that when an embedding update is going on, you can not at the same time read the embedding, as the embedding update is not an atomic operation. As the embedding update can take a long time, and can be quite frequent, you need to figure out a way to allow recommendation to execute without getting blocked by the updates, and at the same time allowing the update to happen, and being able to use some quite recent updated embedding values for recommendation.

ToDo:

In this task, the input `Instruction` set contains all three types of tasks: "init", "update", and "recommend". Note that the "recommend" instruction contains an `iter_idx`, and you should use the embeddings after the updates with index `iter_idx` finish. A recommend instruction with `iter_idx=-1` can be scheduled before any updates. You can start with your codes in Task-3. You should output the recommend result as soon as you get it by calling the provided `Embedding::write_to_stdout()` (we accept all possible order of correct outputs). The delay of recommender response will impact your final score. There is no need to output the final `EmbeddingHolder` in this tasks.

> ***NOTE:*** You should output your recommend results in a thread-safe manner, too.

> ***NOTE:*** In this task, update instructions still have `iter_idx` constraints. Recommend instructions can be executed after update instructions with larger `iter_idx`.

Grading:

You will be graded by the correctness and delay of your recommendation (from the programs' start to recommend result output), as well as being able to read the updated embedding after a relatively short period of time.