Remember that song?

Course: Advanced Programming, Block 1, 2012

Deadline: 16:00, October 25, 2012

1 Objective

The objective of this assignment is to gain hands-on programming experience with Erlang and to get an understanding of how to implement map-reduce algorithms.

The goal is to implement a simple map-reduce framework, and then use this framework to process the musiXmatch dataset (mxm), the official collection of lyrics from the Million Song Dataset.

The assignment consists of two parts, and an optional third part:

- Part 1: Implementing a Map-Reduce framework, that is independent of mxm.
- Part 2: Use your framework to implement some simple algorithms processing the mxm dataset.
- Part 3: Suggestions for various extensions. This part is **optional**, it will have no influence whatsoever on your grading.

1.1 What to hand in?

You should hand in two things:

1. Your code.
2. A short report explaining the code, and an assessment of the quality of code including what this assessment is based on.

1.2 Scope

The following topics are *not* within the scope (of Part 1 and 2) of this assignment:

- Error handling
- Absolute performance

2 Part 1: Map-Reduce Framework

As entry-point for the framework we only want to communicate with a single process, which in turn will take care of starting a number of worker processes for the mapping and reducing phases.

Thus, in the framework there are three kinds of processors:

1. A single *coordinator* that serves as the entry-point for the framework.
2. A fixed size pool of *mappers* that takes care of the mapping phase.
3. A single *reducer* that takes care of the reducing phase.

2.1 API for the Coordinator

The API for the coordinator should be exposed from at module called mr. The module should export three functions:

- A function `start(N)` for starting a coordinator with N mappers and a single reducer. The coordinator should start these processes and return `{ok,Pid}` if it succeeds, where Pid is the process ID of the coordinator.
- A function `job(Pid, MapFun, RedFun, Initial, Data)` for starting a new Map-Reduce job, where `Pid` is process id of a coordinator, `MapFun` is the function that the mappers should perform, `RedFun` is the function that the reducer should perform, `Initial` is the initial value for the coordinator, and `Data` the data that the mappers should be mapped over. The function should return `{ok, Result}` if it succeeds.

  The arguments should have the following types (here using Haskell type-syntax):

  ```
  MapFun  :: a -> b
  RedFun  :: b -> res -> res

  Initial :: res
  Data    :: [a]
  ```

- A function `stop(Pid)` for stopping the coordinator and all worker processes for that coordinator.

2.2 Example

The following example shows how to use the framework for adding and factorial of the numbers 1 to 10.

```
test_sum() ->
    {ok, MR}  = mr:start(3),
    {ok, Sum} = mr:job(MR,
                       fun(X) -> X end,
                       fun(X,Acc) -> X+Acc end,
                       0,
                       lists:seq(1,10)),
```

```
{ok, Fac} = mr:job(MR,
                   fun(X) -> X end,
                   fun(X,Acc) -> X*Acc end,
                   1,
                   lists:seq(1,10)),
mr:stop(MR),
{Sum, Fac}.
```

2.3 Implementation

We encourage (but don't require) you to base your solution on the provided skeleton implementation in the file `mr_skel.erl`.

Use the following rough sketch for the implementation of job:

- The coordinator sends the relevant functions and initialisation data to the mappers and reducer.
- The coordinator sends the data asynchronously to the mappers, for instance by using the send_data function in the skeleton file. Then the coordinator waits for a result from the reducer, that it can pass on to the client.
- When a mapper receive some data it processes the data and asynchronously sends the result to the reducer. One piece of data should result in exactly *one* message sent to the reducer.
- When the reducer receive some data from a mapper it computes a new intermediate result, and then wait for more data. Once it has received all expected data from the mappers it sends the final result to the coordinator.

3 Part 2: MusicXMatch Dataset

The *musiXmatch dataset* (mxm) is the official collection of lyrics from the Million Song Dataset (MSD). The dataset comes in two text files, describing training and test sets in bag-of-words format: each track is described as the word-counts for a dictionary of the top 5,000 words, and a third text file providing matches by mxm based on artist names and song titles from MSD.

Getting the data

- The cut down test version of the bag-of-words:
  http://labrosa.ee.columbia.edu/millionsong/sites/default/files/AdditionalFiles/mxm_dataset_test.txt.zip
- The full version of the bag-of-words:
  http://labrosa.ee.columbia.edu/millionsong/sites/default/files/AdditionalFiles/mxm_dataset_train.txt.zip
- The matching with MSD:
  http://labrosa.ee.columbia.edu/millionsong/sites/default/files/AdditionalFiles/mxm_779k_matches.txt.zip

If you have wget on your machine you can use the commands:

```
wget http://labrosa.ee.columbia.edu/millionsong/sites/default/files/AdditionalFiles/mxm_dataset_test.txt.zip
wget http://labrosa.ee.columbia.edu/millionsong/sites/default/files/AdditionalFiles/mxm_dataset_train.txt.zip
wget http://labrosa.ee.columbia.edu/millionsong/sites/default/files/AdditionalFiles/mxm_779k_matches.txt.zip
```

You can use the provided `read_mxm` module to read in the bag-of-words datasets.

3.1 Tasks

In all the following tasks you should use the mr framework from Part 1, and the bag-of-words dataset (you decide whether to use the test or the train version, just remember to specify which one you have used).

1. Compute the total number of words in all songs together.
2. Compute the average number of different words in a song *and* the average total number of words in a song.
3. Make function grep that for a given word can find the MSD track IDs for all songs with that word. (For your testing, remember that the words in the dataset are stemmed.)
4. Compute a reverse index, that is a mapping (as a dict) from words to songs where they occur.
5. Discuss (shortly) advantages and disadvantages of using a reverse index over the grep function for queries.

4 Assessment

Your hand-in should contain a short section giving an overview of your solution, including **an assessment** of how good you think your solution is, and how you have tested you solution. It is also important to document all *relevant* design decisions you have made. Likewise, if you have failed to complete your implementation, describe in detail the implementation strategy and tests you had planned.

5 Hints

- What kind of data (state) should each process keep track of?
- Start by thinking about which messages the different kinds of processes should send to each other and what information these messages should contain. Then design a representation of these messages.
- If you want to start with Part 2 before making Part 1, it's relatively straightforward to implement a sequential version of the mr module, using the map and foldl functions form the lists module.
- Make yourself familiar with the manual page for lists and the manual page for dict for information about functions in the lists and dict modules. It will come handy.

- While developing the program I've found it helpful to use verb+io:format+ to print various information to the console. For example, my `coordinator_loop` had the following as the first line for a while:

  ```
  io:format("~p is coordinator with the mappers ~p ~n", [self(), Mappers]),
  ```

  (Yes, it's a side-effect but it's useful.) See the manual page for the io module for more information about control sequences available in the format string (like ~p in the example).

- If you want to time a function, the timer:tc function comes handy.

6 Part 3: Extensions

Here are some suggestions for extensions to the assignment, in no particular order.

- **Contest:** Suggest interesting queries/computations you can perform over the dataset. The best suggestion(s) will be awarded a price (queries with accompanying correct implementations will be ranked higher). The size of the price is determined by the interestingness of the query (judged by Ken/Michael). TAs can participate in the contest, but are held to higher standards.

- The framework only have a single reducer, extend the framework so that there is a fixed pool of reducers. Start by thinking about how the types/API for start and job should be changed.

- Currently the initial reading in and splitting of data is done sequentially. Change that so that the splitting is done concurrently with the processing. (You'll need to get intimate with the binary module.)

- No error handling is required for Part 1 and 2. Use the techniques (to be) taught in the course for making your code more robust. For example, set up supervisor(s) for the mappers and reducers, so that (i) if one of worker processes fails they should all fail, or (ii) if one of the mappers fails the reducer is told about the error and can decide how to handle it, or (iii) something else that you think makes sense.

- Change the API so that the coordinator also takes a list of Erlang nodes as argument, and then starts the mappers evenly distributed on these nodes. Thus enabling the work to be distributed on several machines.