AP Assignment 5: Remember That Song

By Simon Gustav Cordes

Part 1

Implementation ideas

The map-reduce program is implemented using the gen_server behaviour. This behavior is used as there is no state transition. The state of the server is just a handler process for handling the reducers and the result returned from these. This handler process is started when the start-function is called.

Mappers and reducers are created on demand when the job function is called on a started server and only run while we are actually running map –reduce on a problem.

When a map-reduce is run on a problem, the data is sent to the mappers, one at a time, which then maps the data and sends the processed data to the reducerhandler. The reducerhandler then handles distribution of the mapped data to reducers, as well as determining when all data has been mapped and reduced. The reducerhandler furthermore handles reducing the values from the reducers, as well as sending this value back to the main process.

With this model we only send messages when reducers need to be created, when data is sent to mappers, when data is sent from mappers to the reducers via the reducer handler, when data is sent from reducers to the reducer handler and when the reducer handler sends the data back to the main process. So we send messages only when needed, specifically when something needs to be done.

The only blocking calls are when reducers are created so we are sure that there is a recipient, when the reducer handler asks for the final result, and when we need to shut down mappers and reducers. All other calls are asynchronous so that mappers and reducers can continue processing information

Code Explanation

API

The API consists of the functions start, job and stop described below. The functions all call the corresponding gen_server functions, except job which calls the gen_server:call function.

```
%%% API

%% start
start() ->
    gen_server:start({local, ?MODULE}, ?MODULE, [], []).

%% job
job(Pid, NWorkers, MapFun, RedFun, Initial, Data) ->
    gen_server:call(?MODULE, {job, Pid, NWorkers, MapFun, RedFun, Initial, Data}).

%% Stops master and and the reducerhandler process
stop(Pid) ->
    gen_server:stop(Pid).
```

Callback Functions

Init

```
init([]) ->
    Pid = self(),
    Reducerhandler = spawn(fun() -> reducerhandler(Pid, [], [], init) end),
    {ok, Reducerhandler}.
```

The init function spawns a new process which runs a reducerhandler where it returns ok and the reducerhandler as the state.

Handle_call

```
handle_call({job, Pid, NWorkers, MapFun, {RedFun, Mode}, Initial, Data}, _From, Reducerhandler) ->
    %% We get the length of the data list.
    %% The idea is that the reducerhandler uses this information to determine when
    %% all the mapped data has been passed through the reducerhandler
    Remaining = length(Data),
    case Mode of
        single ->
            NMappers = NWorkers - 1,
            NReducers = 1;
        multi ->
            NMappers = round(NWorkers * (1/2)), % rougly 1/2 of the workers are mappers, rest are reducers
            NReducers = NWorkers - NMappers
    end,
    Mappers = [spawn(fun() -> mapperloop(Pid, Reducerhandler, MapFun) end) || _ <- lists:seq(1, NMappers)],
    % --------
    % We request a reply because we want to make sure that the other process is done setting
    % up it's reducers.
    % --------
    request_reply(Reducerhandler, {init, NReducers, RedFun, Remaining, Initial}),
    % The data is first sent to the mappers. Then from the mappers to the reducerhandler (see code below)
    send_data(Mappers, Data),
    receive
        {Reducerhandler, {result, Result}} ->
            lists:foreach(fun(Mapper) -> request_reply(Mapper, stop) end, Mappers),
            {reply, {ok, Result}, Reducerhandler}
    end.
```

When the job function is called and handle_call is then subsequently called, the function calculates how many mappers and reducers we need. If in multi-mode we roughly distribute them 50/50 in relation to the NWorkers parameter.

The mappers are then spawned and saved as a temporary list of process ID's.

An init-request is then sent to the reducerhandler with the number of reducers, the reduction function, the length of the dataset (as Remaining) as well as the initial value and waits for the reducerhandler to reply to the main process. When the reply is received, the data is then sent to each of the mappers through the send_data function.

When the mappers and reducers are done, the result is passed back to the main process and the mappers are killed off. The result is then returned.

Send_data

```
%% Method for sending large amounts of data to different receivers one chunk at a time.
%% In this case mappers
send_data(Mappers, Data) ->
    send_loop(Mappers, Mappers, Data).

%% The loop which implements the above send_data method
send_loop(Mappers, [Mapper|Rest], [D|Data]) ->
    async_data(Mapper, D),
    send_loop(Mappers, Rest, Data);
send_loop(Mappers, [], Data) ->
    % If the end of the queue is reached we start the queue over
    send_loop(Mappers, Mappers, Data);
send_loop(_, _, []) ->
    ok.
```

The send_data function and send_loop function distributes the data evenly over the mappers. The data is sent asynchronously since there is no reason to wait until a mapper acknowledges that it has received it. This also means that the sent data can potentially just queue up at a mapper.

Terminate

```
terminate(_Reason, Reducerhandler) ->
    % We ask the reducerhandler to clean itself up
    request_reply(Reducerhandler, stop),
    ok.
```

Terminate is essentially the opposite of init and sends a stop message to the reducerhandler which then stops the process. This is a synchronous implementation since we want to make sure that the reducerhandler is cleaned up.

Handle_cast, handle_info and code_change

```
handle_cast(_Msg, State) ->
    {noreply, State}.
```

```
handle_info(_Info, State) ->
    {noreply, State}.
```

```
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

The handle_cast, handle_info and code_change is not used and is just dummy implementations.

Synchronous communication

```
%%% Synchronous communication

request_reply(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.

reply(From, Msg) ->
    From ! {self(), Msg}.

reply_result(Pid, Result) ->
    reply(Pid, {result, Result}).
```

Standard wrapper functions from sending and replying to synchronous requests and messages.

Asynchronous communication

```
%%% Asynchronous communication

async(Pid, Msg) ->
    Pid ! Msg.

async_data(Pid, Data) ->
    async(Pid, {data, Data}).

async_reduce(Pid, Result) ->
    async(Pid, {reduce, Result}).
```

async_data is used to send data to the mappers while async_reduce is used to send data to the reducers

Mapperloop

```
%%% Mapper code

%% The loop which handles the mapper function and t
mapperloop(Pid, Reducerhandler, Fun) ->
    receive
        {Pid, stop} ->
            reply(Pid, ok);
        {data, Data} ->
            Result = Fun(Data),
            % The mapped result is sent to the redu
            async_reduce(Reducerhandler, Result),
            mapperloop(Pid, Reducerhandler, Fun)
    end.
```

Mapperloop handles the Pid of the parent process, the Pid of the reducerhandler and the map function. When a piece of data is sent to the mapper it passes it to the mapper function and sends the result along to the reducerhandler and restarts the loop to wait for a new piece of data.

When it receives a stop command from the parent it sends a reply back and ends itself.

Reducerhandler

```
%% Handler for multiple reducers
reducerhandler(Pid, Reducers, Reducerqueue, Remaining) ->
    if
        %% probably bad code practice, but it works
        % We assume that Remaining is init or a nonnegative number.
        Remaining == init ->
            receive
                {Pid, {init, NReducers, RedFun, NewRemaining, Initial}} ->
                    Me = self(),
                    NewReducers = [spawn(fun() -> reducerloop(Me, RedFun, Initial) end )
                                   || _ <- lists:seq(1, NReducers)],
                    reply(Pid, ok),
                    reducerhandler(Pid, NewReducers, NewReducers, NewRemaining);
                {Pid, stop} ->
                    reply(Pid, ok)
            end;
        Remaining > 0 ->
            receive
                {reduce, Result} ->
                    NextReducer = hd(Reducerqueue),
                    async_reduce(NextReducer, Result),
                    case tl(Reducerqueue) of
                        [] ->
                            reducerhandler(Pid, Reducers, Reducers, Remaining - 1);
                        _ ->
                            reducerhandler(Pid, Reducers, tl(Reducerqueue), Remaining -1)
                    end
            end;
        Remaining == 0 ->
            %% If remaining is 0 we start preparing for returning one value.
            %% We get the value of the first reducer.
            Acc = request_reply(hd(Reducers), return),
            %% If more reducers exists we pass the value from the previous reducers to eac
            %% to calculate the final value. See reduceresults code below
            Result = reduceresults(tl(Reducers), Acc),
            reply_result(Pid, Result),
            %% The reducerhandler is returned to the initial state.
            reducerhandler(Pid, [], [], init)
    end.
```

The reducerhandler maintains the Pid of the parent process, a list of living reducers as well as maintaining a queue of the reducers so it knows where to send the next mapped result as well as the amount of remaining data to be received.

The value of Remaining is a first set to init. We assume that it is either the atom init or a non-negative number. If it is the atom init it waits for a message of either init or stop. If the init-message is received we spawn the number of reducers specified and passes the reduction function along to the reducer as well as the initial value. The message is synchronous so we reply and call the loop again with the list of reducers and the new value for the size of remaining data.

If Remaining is greater than 0 we still have more data to process and we wait for a result from a mapper. When this is received we pass it along to the next reducer in the queue and call reducerhandler again recursively while decrementing Remaining by 1.

If Remaining is 0 we care done and we get the value from the first reducer.

If there is a single reducer we are done and return the value. If there are multiple reducers we use the property that the reduction function is associative and cumulative so we combine the results from the reducers to a final result which we return. This is done by the reduceresults function.

Since messages are treated in the order they arrive we will have no worries as the reducers are done calculating when we ask for their values.

Reduceresults

```
%% method for reducing the returned results into
reduceresults([], Acc) ->
    %% If there are no more reducers we just wan
    Acc;

reduceresults([Reducer|Rest], Acc) ->
    %% The first reducer in the queue is sent th
    %% perform RedFun(ReducerAcc, Acc) and retur
    request_reply(Reducer, {syncreduce, Acc}),
    receive
        {finalresult, NewAcc} ->
            reduceresults(Rest, NewAcc)
    end.
```

This function takes a list of reducers and an initial value as Acc. This Acc is then passed to the next reducer with calculates RedFun(X, Acc) where X is the reducers current accumulated value. This is done until we return a final value.

Reducerloop

```
reducerloop(Reducerhandler, RedFun, Acc) ->
    receive
        {reduce, Result} ->
            %% The reducer is asked to reduce a result frc
            NewAcc = RedFun(Result, Acc),
            reducerloop(Reducerhandler, RedFun, NewAcc);
        {Reducerhandler, return} ->
            %% The reducer is asked to return its accumula
            reply(Reducerhandler, Acc);
        {Reducerhandler, {syncreduce, Result}} ->
            %% syncreduce is used when the reducerhandler
            %% Used by the reduceresults branch below.
            LastAcc = RedFun(Result, Acc),
            reply(Reducerhandler, LastAcc)
    end.
```

The reducer is implemented as a loop which maintains the Pid of the reducerhandler, a reduction function and the accumulated value of all the received mapper results.

When a reduce-message is received we calculate the new accumulated value and stores it with the reducer. A return message from the reducerhandler will return the accumulated value and shut down the reducer.

If there are several reducers we can use the syncreduce message to calculate and return the final accumulated value and kill the reducer.

Part 2

The tasks of part 2 that have been solved is task 1-3.

Task 1

Task 1 has been solved the following way:

```
task1() ->
    %% We first import the data from the data set via mxm
    {_, Tracks} = read_mxm:from_file("mxm_dataset_test.txt"),
    {ok, MR} = mr:start(),
    {ok, WordCountList} = mr:job(MR,
                                 5,
                                 %% Mapper function which extracts the word counts from the
                                 fun(Track) ->
                                     {_, _, WordCounts} = read_mxm:parse_track(Track),
                                     WordCounts
                                 end,
                                 %% Reducer function which appends the word counts to eachc
                                 {fun(WordCounts, Acc) -> WordCounts ++ Acc end, single},
                                 [],
                                 Tracks),
    {ok, SumOfWords} = mr:job(MR,
                              10,
                              fun({_Words, Count}) -> Count end,
                              {fun(Count, Acc) -> Count + Acc end, multi},
                              0,
                              WordCountList),
    mr:stop(MR),
    SumOfWords.
```

As shown, the task is solved via two map-reduce jobs. The first gets the counts of all words from all tracks in the data set while the second extracts the count and sums it up. Since summing up the total number of words is an associative and cumulative function we make use of the multi-mode in the second job.

This has been run on the test data set yielding the following output:

```
71> mxm:task1().
5761183
```

This looks realistic.

Task 2

Task 2 has been solved the following way:

```
task2() ->
    {_, Tracks} = read_mxm:from_file("mxm_dataset_test.txt"),
    {ok, MR} = mr:start(),
    {ok, SongList} = mr:job(MR,
                            5,
                            fun(Track) ->
                                {_, _, WordCounts} = read_mxm:parse_track(Track),
                                WordCounts
                            end,
                            {fun(WordCounts, Acc) -> [WordCounts] ++ Acc end, single},
                            [],
                            Tracks),
    {ok, {MeanUniqueWords, MeanTotalWords, _Index}} = mr:job(MR,
                                                             3,
                                                             fun(Song) ->
                                                                 {length(Song), wordsinsong(Song, 0), 1}
                                                             end,
                                                             {fun({UniqueWords, TotalWords, 1}, {MeanUnique, MeanTotal, Index}) ->
                                                                 NewIndex = Index + 1,
                                                                 NewMeanUnique = (MeanUnique * Index)/NewIndex + UniqueWords/NewIndex,
                                                                 NewMeanTotal = (MeanTotal * Index)/NewIndex + TotalWords/NewIndex,
                                                                 {NewMeanUnique, NewMeanTotal, NewIndex}
                                                             end, single},
                                                             {0,0,0},
                                                             SongList),
    mr:stop(MR),
    {MeanUniqueWords, MeanTotalWords}.
```

```
wordsinsong([], Sum) ->
    Sum;

wordsinsong([{_Key, Value}|Rest], Sum) ->
    NewSum = Sum + Value,
    wordsinsong(Rest, NewSum).
```

,

Task two is solved by getting the tracks from the data set as a list, and for each track, calculating the number of different words via the built-in length function and the custom wordsinsong function which calculates the total number of words in a song.

The result from running the task2-function gives the following result:

```
74> mxm:task2().
{81.02969458055422,212.252993405297}
```

I have not calculated the standard deviation due to time shortage, however I think intuitively that it looks about right, considering that most songs consist of some somewhat unique verses with a couple of identical choruses.

Task 3

Task 3 has been solved the following way:

```
grep(Word) ->
    {Words, Tracks} = read_mxm:from_file("mxm_dataset_test.txt"),
    Index = get_index(Words, Word),
    {ok, MR} = mr:start(),
    {ok, SongList} = mr:job(MR,
                            5,
                            fun(Track) ->
                                {TID, _, WordCounts} = read_mxm:parse_track(Track),
                                {TID, WordCounts}
                            end,
                            {fun(Song, Acc) -> [Song] ++ Acc end, single},
                            [],
                            Tracks),
    {ok, MIDList} = mr:job(MR,
                            10,
                            fun({MID, WordCounts}) ->
                                case lists:keyfind(Index, 1, WordCounts) of
                                    {_Index, _} ->
                                        [MID];
                                    false ->
                                        []
                                end
                            end,
                            {fun(Msg, Acc) ->
                                Msg ++ Acc
                            end, single},
                            [],
                            SongList),
    mr:stop(MR),
    MIDList.
```

```
get_index(L, Word) ->
    get_index(L, Word, 1).

get_index([], _, _) -> not_found;
get_index([Word|_], Word, Index) ->
    Index;
get_index([_|Rest], Word, Index) ->
    get_index(Rest, Word, Index+1).
```

The function first gets the data as a list of words and a list of song information. We then find the index of the word in the word list via the get_index function, or the term not_found if nothing is found. We then continue to check if the index is found somewhere in the word list for the song. If it is found we pass the song ID from the mapper to the reducer which then computes a list of song ID's.

The result given from running the test is as follows:

```
76> length(mxm:grep("to")).
20694
```

```
77> mxm:grep("to").
[<<"TRAABRX12903CC4816">>,<<"TRAAEJQ128F92C484E">>,
 <<"TRAADFO128F92E1E91">>,<<"TRAAETD128F92F89A3">>,
 <<"TRAAGMC128F4292D0F">>,<<"TRAAFDU128F426E91A">>,
 <<"TRAAIMO128F92EB778">>,<<"TRAAJPW128F42311BD">>,
 <<"TRAAKMD128F1497E6C">>,<<"TRAANJZ128E078264F">>,
 <<"TRAAONF128F92FDEE9">>,<<"TRAAEOT128F14681B9">>,
 <<"TRAAFOY128F146CC17">>,<<"TRAARYR128E078AC7C">>,
 <<"TRAAJOA128F145FB71">>,<<"TRAAKVW128F426205A">>,
 <<"TRAATSW128F4263B0E">>,<<"TRAANTF128F42820D7">>,
 <<"TRAAVUT128F92F74F7">>,<<"TRABCJL128F426C3C1">>,
 <<"TRAAIQG128F425204F">>,<<"TRABGRI128F42A789F">>,
 <<"TRAAPXO128F424AB85">>,<<"TRABIMQ128F4263E0F">>,
 <<"TRAARRJ128F92CEA"...>>,<<"TRABLNQ128F9"...>>,
 <<"TRABMLM1"...>>,<<"TRAB"...>>,<<...>>|...]
```

```
78> mxm:grep("tramboline").
[]
```

As we can see, the function returns a list of the song ID's if the word is found anywhere while it returns an empty list if the word is not found. In particular we can see that the word "tramboline" is not found anywhere in a song in the data set.

Further testing

Only one test has been performed and this is the one supplied with the assignment,

```
simple_test() ->
        {ok, MR}  = mr:start(),
        {ok, Sum} = mr:job(MR,
                           3,
                           fun(X) -> X end,
                           {fun(X, Acc) -> X + Acc end, multi},
                           0,
                           lists:seq(1,10)),
        {ok, Fac} = mr:job(MR,
                           4,
                           fun(X) -> X end,
                           {fun(X, Acc) -> X * Acc end, multi},
                           1,
                           lists:seq(1,20)),
        mr:stop(MR),
        {Sum, Fac}.
```

giving the following results:

```
80> mrtests:simple_test().
{55,2432902008176640000}
```

We can see with a simple calculator that this result is indeed correct and that it is in this case working as intended.

Test discussion

The solved tasks from part 2 give a good idea of the correctness of the program. There are no problems running the code, and the program works in both single- and multi-mode. More tests could probably be done to tests the robustness of the program, however, the most basic functionality has been tested and found to be working.

Reproducing the test results

The test results can be reproduced by compiling the module mxm.erl in the Erlang shell with c(mxm). and running the functions mxm:task1(), mxm:task2(), mxm:grep("to"), length(mxm:grep("to")) and mxm:grep("tramboline") as in the given results.

The simple_test function can be run by compiling and loading the module in mrtests.erl and running mrtests:simple_test() in the Erlang shell.