Data Structures and Functional Programming	**Problem Set 5**
CS 3110, Fall 2013	**Due at 11:59 PM, Thursday, November 14**
Version: 1	**Last Modified: November 2, 2013**

In this project, we will implement a version of Google's MapReduce. Before starting this assignment, please read the first three sections of the original Google research paper (available here) to familiarize yourself with the MapReduce framework.

Instructions

Compile Errors

All code you submit must compile. **Programs that do not compile will be heavily penalized.** If your submission does not compile, we will notify you immediately. You will have 48 hours after the submission date to supply us with a patch. If you do not submit a patch, or if your patched code does not compile, you will receive an automatic zero.

Naming

We will be using an automatic grading script, so it is **crucial** that you name your functions and order their arguments according to the problem set instructions, and that you place the functions in the correct files. Incorrectly named functions are **treated as compile errors** and you will have to submit a patch.

Code Style

Finally, please pay attention to style. Refer to the CS 3110 style guide and lecture notes. Ugly code that is functionally correct may still lose points. Take the extra time to think out the problems and find the most elegant solutions before coding them up. Good programming style is important for all assignments throughout the semester.

Late Assignments

Please carefully review the course website's policy on late assignments, as **all** assignments handed in after the deadline will be considered late. Verify on CMS that you have submitted the correct version, **before** the deadline. Submitting the incorrect version before the deadline and realizing that you have done so after the deadline will be counted as a late submission.

Part 1: Map Reduce (60 points)

Modern applications rely on the ability to manipulate massive data sets in an efficient manner. One technique for handling large data sets is to distribute storage and computation across many computers. Google's MapReduce is a computational framework that applies functional programming techniques to parallelize applications.

In this problem set, you will implement a simplified version of the MapReduce framework. The main part of this assignment will require you to implement a coordinator that will distribute work to independent workers and collect the results of the computation. You will implement implement various applications that make use of your framework.

Understanding MapReduce

MapReduce was spawned from the observation that a wide variety of applications can be structured into a **map phase**, which transforms independent data points, and a **reduce phase** which combines the transformed data in a meaningful way. This is a very natural generalization of the `List.fold` function with which you are intimately familiar.

MapReduce applications provide the `map` and `reduce` functions that are applied in these two phases. These functions must meet the following specifications:

```
type key   = string
val map    : (key * 'a)       -> (key * 'b) list
val reduce : (key * 'b list) -> 'c list
```

Our implementation of MapReduce requires that all keys and values are strings, but we have written the values here as polymorphic types to emphasize the structure of the map and reduce functions[1]:

- The `map` function takes a single `key * value` pair and transforms the value. This function is called once for each data point in the input, and output from these calls is computed in parallel.
- The `map` results are then serially aggregated in a `combine` phase. Values associated with duplicate keys are merged here into a single list.
- Finally, the `reduce` function takes a `key * value list` pair and outputs a list of transformed values. `reduce` is called once for each independent key. As with `map`, these calls are computed in parallel.

This is the high-level structure of MapReduce. Applications plug in to this generalized framework which parallelizes the execution of the application-provided functions on the application-provided input.

[1] More general types can be encoded as strings using the marshalling and unmarshalling facilities described later in the writeup.

Code Structure

Here we summarize the contents of the release files provided via CMS.

controller

- `main.ml`
  This is the main module that is called to start up an application. It parses the command-line arguments to determine which application to start, then calls the main method of that application, passing it the command-line arguments.

- `map_reduce.ml`
  This module implements the map, combine, and reduce operations. It exports the general `map_reduce` function.

- `worker_manager.ml`
  This module handles communication with workers.

  It includes functions to initialize and kill mappers and reducers, select inactive workers, assign a map/reduce task to a worker, and return workers that have completed their task to the pool of inactive workers. Detailed descriptions of each function in the module are in `worker_manager.mli`.

worker_server

- `program.ml`
  This module provides the ability to build and run mappers/reducers. It includes the function `get_input` and `set_output`, which application code uses to interface with the framework at large.

- `worker_server.ml`
  This module is responsible for initializing the server and spawning threads to handle client connection requests. These threads simply invoke `Worker.handle_request`.

- `worker.ml`
  This module is responsible handling requests from the MapReduce controller. A more thorough description is contained in the "Your Tasks" section.

Communication Protocol

The protocol that the controller application and the workers use to communicate is stored in `shared/protocol.ml`. This section explains how you should respond to the types of requests you can receive in `worker.ml`. These requests are encoded as type constructors in the `worker_request` type. The responses that the worker returns to the controller are similarly described by the `worker_response` type.

Requests

- `InitMapper (code : string list)`
  Initialize a mapper which will later execute the provided code.
- `InitReducer (code : string list)`
  Initialize a reducer.
- `MapRequest (id : worker_id, key : string, value : string)`
  Execute mapper with id `id` with the provided key and value as input.
- `ReduceRequest (id : worker_id, key : string, values : string list)`
  Execute reducer with id `id` with `(key, values)` as input.

Responses

- `Mapper (id_opt : worker_id option, error : string)`
  `Mapper (Some id, _)` indicates the requested mapper was successfully built, and has the returned id. `Mapper (None, error)` indicates compilation of the mapper failed with the returned error.
- `Reducer (id_opt: worker_id option, error : string)`
  Same as above, except for a reducer.
- `InvalidWorker (id : worker_id)`
  Indicates that the map or reduce request was made using an invalid id.
- `RuntimeError (id : worker_id, error : string)`
  Indicates that a worker failed to call `Program.set_output`.
- `MapResults (id : worker_id, result : (string * string) list)`
  Successful termination of a map request.
- `ReduceResults (id : worker_id, result : string list)`
  Successful termination of a reduce request.

Marshalling

We noted above that our map and reduce functions operate on strings and not arbitrary data types. We have imposed this restriction because only string data can be communicated between agents. Values can be converted to and from strings explicitly using `Util.marshal` and `Util.unmarshal`, which call the OCaml built-in `Marshal.to_string` and `Marshal.from_string` functions, respectively. You can send strings via communication channels without marshalling, but other values need to be marshalled. Your mapper or reducer must also convert the input it receives from strings back to the appropriate type it can operate on, and once it has finished, it needs to convert its output into strings to communicate it back.

Note that marshalling is not type safe. OCaml cannot detect misuse of marshalled data during compilation. If you unmarshal a string and treat it as a value of any type other than the type it was marshalled as, your program will compile, run, and crash. You should therefore take particular care that the types match when marshalling/unmarshalling. Make sure that the type of marshalled input sent to your mapper matches the type that the mapper expects, and that the type of the marshalled results that the mapper sends back matches the type expected. The same is true for reducers using the reducer messages. Explicit type annotations can be helpful in this situation:

```
let x : int = Util.unmarshal my_data
```

Hashtbl

For this assignment, there are some places where you will be expected to use a hash table. See the Ocaml Hashtbl module. A hash table is a data structure that maps keys to values, and has O(1) insert, delete, and search time complexity.

A hash table is often a good choice for very large data sets with random access. Because map reduce is based upon extremely large data sets, we expect that your controller (in `map_reduce.ml`) will use a hash table to keep track of data.

OCaml's hash tables are mutable: the table is modified in place. You should exercise caution when modifying mutable data structures from multiple threads simultaneously. Familiarize yourself with the Ocaml Mutex module.

Your Tasks

All code you submit must adhere to the specifications defined in the respective .mli files. As always, do not change the .mli files.

You must implement functions in the following files:

- `controller/map_reduce.ml`
  The code in this module is responsible for performing the actual map, combine, and reduce operations by initializing the mappers and reducers, sending them input that hasn't been mapped/reduced yet, and then returning the final list of results.

The map and reduce functions must be implemented according to the specifications in `controller/map_reduce.mli`. The functionality should mirror the above example execution description. Both functions must make use of available workers simultaneously and be able to handle worker failure. However, you don't need to handle the case when all workers fail when there is no coding error in your worker code (i.e. complete network failure). Additionally, there should only be one active request per worker at a time, and if a worker fails, it should not be returned to the `Worker_manager` using the `push_worker` function.

Both map and reduce share the same basic structure. First, the workers are initialized through a call to the appropriate `Worker_manager` function. Once the workers have been initialized, the input list should be iterated over, sending each available element to a free worker, which can be accessed using the `pop_worker` function of `Worker_manager`. A mapper is invoked using `Worker_manager.map` providing the mapper's id and the (key, value) pair. Each mapper (and therefore invocation of `Worker_manager.map`) receives an individual (key, value) pair, and outputs a new (key, value) list, which is simply added onto the list of all previous map results.

A reducer is invoked similarly using `Worker_manager.reduce`. These functions block until a response is received, so it is important that they are invoked using a thread from the included `Thread_pool` module. Additionally, this requires that these spawned threads store their results in a shared data structure, which must be accessed in a thread-safe manner.

Once all of the input has been iterated over, it is important that any input that remains unfinished, either due to a slow or failed worker, is re-submitted to available workers until all input has been processed. Finally, close connections to the workers with a call to `Worker_manager.clean_up_workers` and return the results.

Note: It is acceptable to use `Thread.delay` with a short sleep period (0.1 seconds or less) when looping over unfinished data to send to available workers in order to prevent a flooding of workers with unnecessary work.

The combine function must be implemented according to the specification, but does not need to make use of any workers. It should combine the provided (key, value) pair list into a list of (key, value list) pairs in linear time such that each key in the provided list occurs exactly once in the returned list, and each value list for a given key in the returned list contains all the values that key was mapped to in the provided list.

- `worker_server/worker.ml`
  The code in this module is responsible for handling communication between the clients that request work and the mapper/reducers that perform the work.

  The `handle_request` function must be implemented according to the .mli specification and the above description, and must be thread-safe. This function receives a client connection as input and must retrieve the `worker_request` from that connection.

  If the request is for a mapper or reducer initialization, then the code must be built using `Program.build`, which returns `(Some id, "")` where id is the worker id if the compilation was successful, or `(None, error)` if the compilation was not successful. If the build

succeeds, the worker id should be stored in the appropriate mapper or reducer set (depending on the initialization request), and the id should be sent back to the client. If the build fails, then the error message should be sent back to the client. Use `send_response` to return these messages. If the request is to perform a map or reduce, then the worker id must be verified to be of the correct type by looking it up in either the mapper or reducer set before the mapper or reducer is invoked using `Program.run`. The return value is `Some v`, where `v` is the output of the program, or None if the program failed to provide any output or generated an error.

Note that the mapper and reducer sets are shared between all request handler threads spawned by the `Worker_server`, and therefore access to them must be thread-safe.

Note: Code in `shared/util.ml` is accessible to all workers by default.

Execution Example

In this section, we will step through a word count MapReduce program.

Client Execution Example

The input to the `word_count` job in MapReduce is a set of (document id, body) pairs. In the Map phase, each word that occurs in the body of a file is mapped to the pair (word, "1"), indicating that the word has been seen once. In the Combine phase, all pairs with the same first component are collected to form the pair (word, ["1"; "1"; ...; "1"]) with one such pair for each word. Then in the Reduce phase, for each word, the list of ones is summed to determine the number of occurrences of that word.

For simplicity, our framework accepts only a single data file containing all the input documents. Each document is represented in the input file as a (id, document name, body) triple. See `data/reuters.txt` or `data/word_count_test.txt` for examples. `Map_reduce.map_reduce` prepares a document file by first calling `Util.load_documents` and then formatting the documents into (key, value) pairs for the mapper.

We have included a full implementation of this application in `apps/word_count`. Here is a detailed explanation of the sequence of events.

1. The controller application is started using the command:

   ```
   ./controller.exe word_count <filename>
   ```

   It immediately calls the main method of `apps/word_count/word_count.ml`, passing it the argument list. The main method calls `controller/Map_reduce.map_reduce`, which is common controller code for performing a simple one-phase MapReduce on documents. Other more involved applications have their own controller code.

2. The documents in *filename* are read in and parsed using `Util.load_documents` which splits the collection of documents into {id; title; body} triples. These triples are converted into (id, body) pairs.

3. The controller calls `Map_reduce.map kv_pairs "apps/word_count/mapper.ml"`.

4. `Map_reduce.map` initializes a mapper worker manager using:

   ```
   Worker_manager.initialize_mappers "apps/word_count/mapper.ml"
   ```

   The `Worker_manager` loads in the list of available workers from the file named addresses. Each line of this file contains a worker address of the form `ip_address:port_number`. Each of these workers is sent the mapper code. The worker creates a mapper with that code and sends back the id of the resulting mapper. This id is combined with the address of the worker to uniquely identify that mapper. If certain workers are unavailable, this function will report this fact, but will continue to run successfully.

5. `Map_reduce.map` then sends individual unmapped (id, body) pairs to available mappers until it has received results for all pairs. Free mappers are obtained using:

   ```
   Worker_manager.pop_worker()
   ```

   Mappers should be released once their results have been received using:

   ```
   Worker_manager.push_worker
   ```

   Read `controller/worker_manager.mli` for more complete documentation. Once all (id, body) pairs have been mapped, the new list of (word, "1") pairs is returned.

6. `Map_reduce.map_reduce` receives the results of the mapping. `Util.print_map_results` can be called here to display the results of the Map phase for debugging purposes.

7. The list of (word, "1") pairs is then combined into (word, ["1"; ...; "1"]) pairs for each word by calling `Map_reduce.combine`. `Util.print_combine_results` prints the output.

8. `Map_reduce.reduce` is then called with the results of `Map_reduce.combine` and the name of the reducer file `apps/word_count/reducer.ml`.

9. `Map_reduce.reduce` initializes the reducer worker manager by calling the appropriate `Worker_manager` function, which retrieves worker addresses in the same manner as for mappers.

10. `Map_reduce.reduce` then sends the unreduced (word, count list) pairs to available reducers until it has received results for all input. This is performed in essentially the same manner as the map phase. When all pairs have been reduced, the new list of (word, count) tuples is returned. In this application, the key doesn't change, so `Worker_manager.reduce` and the reduce workers only calculate and return the new value (in this case, count), instead of returning the key (in this case, word) and count.

11. The results are returned to the main method of word_count.ml, which displays them using `Util.print_reduce_results`.

Worker Execution Example

1. Multiple workers can be run from the same directory as long as they listen on different ports. A worker server is started using `worker_server.exe <port_number>`, where `<port_number>` is the port the worker listens on.

2. Worker_server receives a connection and spawns a thread, which calls `Worker.handle_request` to handle it.

3. `Worker.handle_request` determines the request type. If it is an initialization request, then the new mapper or reducer is built using Program.build, which returns either the new worker id or the compilation error. See worker_server/program.ml for more complete documentation. If it is a map or reduce request, the worker id is verified as referencing a valid mapper or reducer, respectively. If the id is valid, then `Program.run` is called with that id and the provided input. This runs the relevant worker, which receives its input by calling `Program.get_input()`. Once the worker terminates, having set its output using `Program.set_output`, these results are returned by Program.run. If the request is invalid, then the appropriate error message, as defined in shared/protocol.ml, is prepared.

4. Once the results of either the build or map/reduce are completed, then `Worker.send_response` is called, which is responsible for sending the result back to the client. If the response is sent successfully, then `Worker.handle_request` simply recurses, otherwise it returns unit.

Part 2: Using MapReduce (30 points)

Median Grade

To get you started with using your map reduce, we will start with a simple app. You are provided with these functions in Util for general I/O and string manipulation:

```
(*parses file into student types*)
load_grades (filename: string) : student list

(*splits a string of classes into a list of class strings*)
split_to_class_lst string -> string list

(*splits a string by spaces*)
split_spaces string -> string list

(*prints the kvs list given*)
print_reduced_courses (kvs_list : (string * string list) list) -> ()
```

Data will come in as a student object that holds a student id and a string list of courses. It is your job to give us the median grade for each course represented.

Transaction Track

This application is based extremely loosely on the Bitcoin protocol and asks you to track the ownership of bitcoins. You do not need to know anything about Bitcoin that isn't in this writeup. Bitcoin is a crypto-currency which uses one way hashes to ensure the security of transactions.

Bitcoin data are stored in a public data structure called the blockchain in a defined order. The elements of the blockchain are called "blocks" and the id of a given block is called the blockid. Each block has a hash which is dependent on all previous blocks for security. A block consists of generating a new coin and any number of transactions. A transaction consists of a list of inputs, private keys to decrypt the inputs, and a list of outputs and public keys to restrict ownership of the outputs. We have pre-parsed the block chain and removed all unnessary data for you. A transaction has the following format:

incoincount outcoincount incoinid1 incoinid2... outcoinid1 outcoinamount1 outcoinid2 outcoinamount2...

When users attempt transactions, they typically have slightly more in coins than out coins. This difference is known as a transaction fee. Note that when a coin is used as an input, the entirety of its **present** value is consumed.

A block has the following format:

txcount tx1 tx2 tx3 ...

The first transaction in a block will usually be a mining transaction. Coins are mined by solving mathematical problems (SHA-256). The first transaction typically consists of claiming the block reward and all of the transaction fees from the transactions within this block (thus providing the incentive for accepting transactions). Block rewards began at 50 bitcoins and halve periodically until reaching zero. Coinids are strings and coin amounts are integers. They range from 1, representing .00000001 bitcoins (1 satoshi) to 2.1 quadrillion (representing one coin containing all of the bitcoins that could ever potentially exist).

Due to the progressive weakening of encryption with use, coinids are rarely reused. As such, the vast majority of coinids have zero value. For this problem, we would like you to write a MapReduce application which processes the entire blockchain and returns a list of coinids and their present value. Our given code will then filter this list and remove zeros.

We have provided you with skeleton code; it is only necessary for you to complete `mapper.ml` and `reducer.ml`. As `transaction_track.ml` shows, the mapper should take in a block id, block data tuple. Use `Util.split_words` to turn the block data string into a string list. The mapper should output key, value pairs with coinids as keys. The mapper should accept these pairs and output the final value for each coinid as a single element string list to allow reuse of our util functions.

Karma Problem : N-body simulation

An `n-body simulation` models the movement of objects in space due to the gravitational forces acting between them over time. Given a collection of n bodies possessing a mass, location, and velocity, we compute new positions and velocities for each body based on the gravitational forces acting on each. These vectors are then applied to the bodies for a small period

of time and then the process repeats, creating a new vector. Tracking the positions of the bodies over time yields a series of frames which, shown in succession, model the bodies' movements across a plane. The module `shared/plane.ml` defines representations for scalar values, two-dimensional points, vectors, and common functions such as Euclidean distance. Using Plane, we can define a type that represents the mass, position, and velocity of a body (all found in `util.ml`):

```
type mass = Plane.scalar
type location = Plane.point
type velocity = Plane.vector
type body = mass * location * velocity
```

We can also define a function `acceleration` that calculates the acceleration of one body due to another:

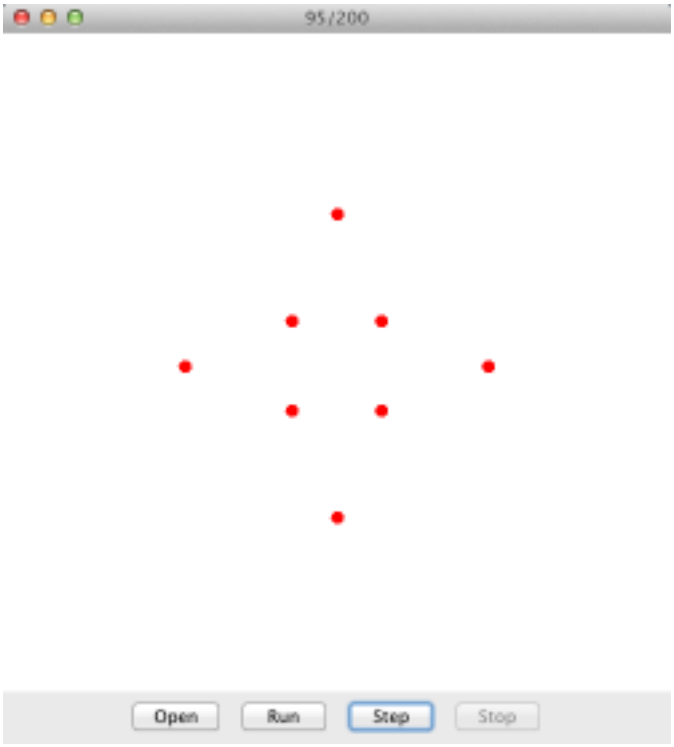
```
val acceleration : body -> body -> Plane.vector
```

To understand how the acceleration function works, we need to review a few basic facts from physics. Recall that force is equal to mass times acceleration ($F = m \times a$) and the gravitational force between objects with masses m_1 and m_2 separated by distance d is given by $\frac{G \times m_1 \times m_2}{d^2}$ where G is the gravitational constant (found in `util.ml` as `cBIG_G`). Putting these two equations together, and solving for a, we have that the magnitude of the acceleration vector (due to gravity) for the object with mass m_2 is $\frac{G \times m_1}{d^2}$. The direction of the acceleration vector is the same as the direction of the unit vector between from the subject body to the gravitational body; in this case, the acceleration direction is from m_2 to m_1. Note that this calculation assumes that the objects do not collide.

Given accelerations for each body, we move the simulation forward one time step, updating the position `p` and velocity `v` of each body to `p + v + a/2` and `v + a` respectively, where `a` is the `Plane.vector` in the sequence returned by `acceleration`.

This algorithm fits nicely into the MapReduce framework. Accelerations for each body can be computed and applied in parallel: map across the bodies to get the accelerations on each due to every other body, then apply each acceleration vector to get a new position and velocity for the body. Note that many useful methods are provided in module `Plane`. Types and some debugging methods are declared in `util.ml`. Read them carefully before you start.

We have provided implementations of `Nbody.main` and the IO helper `Util.string_of_bodies`. Your task is to create the `nbody/mapper.ml` and `nbody/reducer.ml` for this app, and use them to implement `Nbody.make_transcript`, which will run a simulation for a given number of iterations and generate a textual representation of the bodies over time. This output file can be opened using the supplied viewer `bouncy.jar`, which displays the simulation:



Recall the command to run jar files:

```
java -jar bouncy.jar
```

Specifically, `make_transcript` should take a list of `(string * body)` pairs, where the string uniquely identifies the dynamic bodies, and an integer `steps` and update the bodies for `steps` iterations using the `acceleration` function described above.

`nbody/mapper.ml` and `nbody/reducer.ml` should modify the bodies at each step while maintaining the identifier strings. You should document bodies' positions after each update using `Util.string_of_bodies` and return a complete string once steps updates have occurred. We have provided sample bodies in `shared/simulations.ml`. You can use these as models to write your own simulations, which you may **optionally** submit as `Simulations.zardoz`. Particularly creative submissions may receive **additional** karma. Have fun!

Part 3: Dependent Types (15 points)

In the file `mapSpecs.ml`, you should give dependent type specifications for some of the functions in the OCaml `Map` module. To get you started, we have given you the specifications for `bindings`, `mem`, and `add`. You should fill in the specifications for `empty`, `find`, `remove` and `equal`.

Your dependent types should be written in comments using the OCaml/SL syntax, but your types should refer to functions written in ordinary OCaml. See the provided specifications for examples.

Your types should be sufficient to guarantee that a function never fails. That is, if the OCaml specification allows a function to fail, your type for that function should prevent it from being

called on an input that would cause it to fail.

We strongly recommend that both partners work on this part, since these make good exam questions.

Part 4: Version Control (3 points)

You are required to use a source control system like Git or SVN. Submit the log file that describes your activity. You may use Cornell SourceForge to create a SVN repository, or you may use a private repository from a provider like xp-dev, bitbucket, or github. Do not post your code in a public repository. That would be an academic integrity violation.

For information on how to create you subversion repository on SourceForge, read Create a Subversion Repository.

If you use Windows, and are unfamiliar with the command line or Cygwin, there is a GUI based SVN client called Tortoise SVN that provides convenient context menu access to your repository.

For Debian/Ubuntu, a GUI-based solution to explore would be Rapid SVN. To install:

```
apt-get install rapidsvn
```

Mac users looking into a front-end can try SC Plugin, which claims to provide similar functionality to Tortoise. Rapid SVN might work as well. There is also a plug-in for Eclipse called Subclipse that integrates everything for you nicely. Note that all of these options are simply graphical mimics of the extremely powerful terminal commands, and are by no means necessary to successfully utilize version control.

Part 5: Design Review Meeting (7 points)

We will be holding 15-minute design review meetings. Your group is expected to meet with one of the course staff during their office hours to discuss this assignment. A sign-up schedule is available on CMS.

Be prepared to give a short presentation on how you and your partner plan to implement MapReduce, including how you will divide the work, and bring any questions you may have concerning the assignment. Staff members will ask questions to gauge your understanding of varying aspects of the assignment, ranging from the high-level outline to specific design pitfalls. This is not a quiz; you are not expected to be familiar with the intricacies of the project during your design review. Rather, you are expected to have outlined your design and prepared thoughtful questions. Your design review should focus on your approach toward implementing the MapReduce framework, specifically controller/map_reduce.ml, but you may spend a few minutes at the end discussing the MapReduce applications.

Design reviews are for your benefit. Meet with your partner prior to the review and spend time outlining and implementing MapReduce. This is a large assignment. The review is an opportunity to ensure your group understands the tasks involved and has a feasible design early on.