C++ Mapreduce

Document Number: ISO/IEC JTC1 SC22 WG21 N3563 - 2013-03-15
Supersedes: ISO/IEC JTC1 SC22 WG21 N3446=12-0136 - 2012-09-21

Authors:
Chris Mysen, mysen@google.com, ccmysen@gmail.com
Lawrence Crowl, crowl@google.com, Lawrence@Crowl.org
Adam Berkan, aberkan@google.com
Reply-to:
Chris Mysen <mysen@google.com>

Introduction

The difficulties in writing race-free and efficient threaded and distributed code often lies in the difficulty in coordinating and dividing the processing of large data problems. There are several classes of data-parallel problems where there are large numbers of homogenous operations on blocks of data, which can can be approached by mapping input onto a key-value space then combining values at the key level. An example of this is counting words in a set of source files, where the input space is a set of files, and the output space is word->count aggregated across all inputs. This simple concept can then be mapped onto a 3 stage parallel process with highly parallel mappers each reading an input file and generating (word, 1) pairs, a shuffle mechanism which acts as a barrier that reorganizes data by word, then a highly parallel reduce which takes the vector of 1s for each word, and sums them to produce the total count (with no synchronization required).

This mechanism has been put to great use in data-heavy distributed processing problems due to the high level of parallelism achievable and the convenience of being able to re-organize data through the shuffle operation, but has also found good use in the implementation of many threaded algorithms which can be implemented as stages of independent (map and reduce) tasks which require high amounts of data reorganization between stages to achieve parallelism.

The word-counting problem will be used as a simple motivating example throughout this paper. A traditional concurrent solution to this would be to read the files in different threads and update a global map of word->count as words are read from input files (with heavy contention on the output map). As seen, map-reduce solves this by keeping the map-structure but instead of having all threads contend on global structures at a granular level, data is grouped to allow for highly parallel, unsynchronized operations. The key piece of efficiency comes from the shuffle which re-organizes the data between the map and reduce stages (we will see some additional efficiency improving operations later).

Some other examples of problems which can be organized in this way are: building reverse indexes (converting files into word->file pointer maps), matrix multiplication (operating on matrix blocks in the mapper), clustering (distance calculation, then recalculating cluster centroids), regression (partial statistics calculation, then aggregation across all data).

Overview

For large scale distributed problems, the map-reduce framework has proven to be a highly effective way at creating highly parallel workflows working over distributed filesystems on petabyte scale operations and has been used for analysis, machine learning, and implementation of many distributed computation problems at Google.

It is widely popular with several implementations, including some open source, because of the simplicity of the programming model and the high amount of parallelism possible. It’s also a convenient programming model because it simplifies operations into a problem of mapping data into a new keyspace, and combining data within that keyspace, with the task of reorganizing the data between stages and coordinating this process into a highly efficient shuffle operation which minimizes the time threads spend coordinating their work. You can think of map-reduce as a different way of programming parallelism (as a data-parallel operation with high data interdependence) than working with threads (which is inherently more focused on task-parallelism).

The whole flow of the proposed works as follows:

- Input is defined as any iterable structure of values.
- Each of a set of mappers each operates on a single input value and emits one (or more) key+value pairs (logically acting like a key->value multimap)
    - An optional combiner class which acts as a pre-reducer on a per-mapper basis (reducing the storage footprint of the mapper output)
- The shuffle stage (implemented by the controller) reorganizes the key-value pairs by key and shard (basically per reducer thread)
- Each of a set of reducers receives (key, list of values) from the mappers and reducing the set values to a key+output pair
- The map-reduce controller handles the work of organizing these operations, it splits up inputs for the mappers, executes mappers until all input is consumed, blocks to coalesce and sort key/value pairs from the mapper outputs (and optionally runs the combiner phase), and a reduces the key+value lists to a single output key/output pair.

Map-reduce translates well into threaded applications and can be designed in such a way as to make highly data parallel operations simple to write and optimize without requiring particularly deep understanding of threaded programming or data synchronization (in fact, the programmer can be completely unaware of synchronization and still write efficient parallel map-reduces) as the programmer generally need only write stateless map() and reduce() functions and all the coordination of work is handled by the framework. The framework also provides controls which allows for more advanced use cases, like optimizations about how data is processed, stored, and shuffled.

This proposal outlines the details of a version of mapreduce which is logically simple but very extensible, based heavily off of both distributed and threaded versions of mapreduce, allowing for implementations which work as distributed, threaded, or both. There are some simplifications in this implementation which are detailed later, but much of the flexibility of many of the implemented map-reduces is retained.

Terminology

Before discussing the interface in detail, there are a few terms which are helpful to understand:

- **Mapper** - process which translates arbitrarily divided input up into key-value pairs (an example is reading a file chunk and generating word-count for that file chunk)
- **Shuffler** - process which restructures the unordered key-value outputs of the mapper

into a map of key->set of values required by the reducer (in the word counting example, taking all separate word-counts from the input files and re-organizing by word-count)

- **Reducer** - process which takes the key-set of values and creates a single key-value output (sum up the counts for each word and output a single word-count for all files)
- **Combiner** - reducer process which can pre-aggregate data coming from a single mapper used to optimize the footprint of data before entering the shuffle phase (for word counting, the mapper would output word-1, and the combiner pre-aggregates by word per mapper, creating word-count)
- **Shard** - a chunk of data, organized by key, which is a logical concept used to improve data locality and/or load balancing
- **MapReducer** - parent controller which creates threads and coordinates stages of the pipeline

Basic Use

Usage of the mapreduce library includes the following:

1. define a concrete mapper class which implements the mapper interface functions. The mapper class defined a set of pre-defined types and receives a mapper_tag classification (the default is input_unaware) as well as the following member functions.
    a. `(optional) start()`
    b. `(optional) flush()`
    c. `(optional) abort()`
    d. `map()`
2. define a concrete reducer class which subclasses the reducer class. The functions are effectively the same, except clearly the reduce() function instead of map()
3. declare a set of map_reduce_options which contain a set of parameters for the map_reduce class (number of workers, thread pool class, etc).
4. declare a concrete instance of the map_reduce class, templatized on the input/output iterator classes used to process values, and on the mapper/reducer classes previously defined.
5. call the run() function of the map_reduce object, which then consumes all inputs of the iterator passed in and passes all output to the output iterator (provided to the map_reduce object via the map_reduce_options). Functionally, the map_reduce works in 3 stages.
    a. In the first stage, the mapper threads are created and provided a pointer to the input iterator, they each continue consuming inputs until the input values are fully consumed and outputing key-value pairs to an output buffer (thread local)
    b. In the second stage is the synchronization stage, where the map_reduce class detects the completion of the mapper jobs (based on all mappers stopping), then the output of the mappers is passed to the shuffler which collects the output buffers from the mappers, sorts them, then passes the values to an input buffer for each assigned reducer (default is a hash of the key, so random assignment)

c. In the last stage, the reducer threads are created and process all values for a given key, outputting the final output of the map-reduce pipeline (with the output target unspecified but normally this will be to some map structure as well).

Additional Optimizations Provided

These optimizations are intended to handle some of the cases which come up when optimizing map-reduce operations, primarily focused on improving data locality and storage efficiency.

- **Combiner** - a commutative/associative reducer which allows you to pre-aggregate values before they are shuffled for use by the final reducer. This mechanism only works when the output_type == value_type, but can provide a large impact on storage footprint (and potentially performance due to data locality and the cost of reading/writing to storage). For the library implementer, this would sit on top of the output of the map stage and would combine the output of the mapper with the previously stored map output and store the new combined output. For example, a sum combiner would take the old sum and add the new value on top.
- **Input Splitter** - this is a function running with the mapper thread, which takes an iterator value and converts to 0 or more input type values. This can easily be used to chunk or store data in more efficient/natural ways (for example, a splitter could be defined to read words from a file in the example below). This can also be used to create work units which are larger than a single input value (for example NxN blocks of a source input image). The identity_splitter is provided as a default, which passes the values untouched to the mapper, but replacement splitters can also be specified.
- **Shard Function** - function which defines which keys are sent to which reducers. This capability allows for better control of object locality. The default shard function assumes keys can be hashed.

Example Code

This example illustrates a simple map-reduce pipeline implementing word counting on a set of input files as mentioned in the introduction (showing the minimal implementation required for this problem). It takes as input a set of files and outputs word->count for all words in the file.

```
class WordMapper : public mapper_traits<string, string, int> {
 public:
  void start() {}
  void flush() {}
  bool abort() { return true; }

  template <typename OutIter>
  void map(const string& filename, OutIter output) {
        ifstream infile(filename.c_str());
        while (infile.good()) {
            string line;
            getline(infile, line);
            regex word_re("(\\w+)");
            sregex_iterator iter(line.begin(), line.end(), word_re);
            sregex_iterator end;
            for (; iter != end; ++iter) {
                string word = iter->str();
                output = std::pair<string, int>(word, 1);
            }
        }
        infile.close();
  }
};

class SumReducer : public reducer_traits<string, int, int> {
 public:
  void start(unsigned int shard_id, bool is_combiner) {}
  void flush() {}
  bool abort() { return true; }

  template <typename InIter, typename OutIter>
  void reduce(
      const string& key,
      InIter* value_start, InIter* value_end,
      OutIter output) {
    unsigned long long value_sum = 0ULL;
    for (InIter* value_iter = value_start;
         (*value_iter) != (*value_end);
         ++(*value_iter)) {
      value_sum += *(*value_iter);
    }
    output = std::pair<string, int>(key, value_sum);
  }
};

// Set up input and output for the MR operation
blocking_output_map out_map;
output_iter out_iter(&out_map);

buffer_queue<string> input_queue(1000, file_names.begin(), file_names.end());
queue_wrapper< buffer_queue<string> > input_wrap(input_queue);

// Create the map-reduce operation and hook up the input.
map_reduce_options<WordMapper, SumReducer, output_iter> opts
  { 3, /* num mapper threads */
    19, /* num reduce shards */
    5, /* num reducer threads */
    new simple_thread_pool, /* executor */
    out_iter /* iterator receiving output calls */
  };

map_reduce<input_iter, output_iter, WordMapper,  SumReducer> mr(opts);

// Block on the map-reduce completion.
if (mr.run(input_wrap.front().begin(), input_wrap.front().end())) {
  cout << "Success" << endl;
}
```

Proposed Interface

Mapper Class Definition

The mapper traits class is just a set of typedefs which declare the types of values which the mapreduce operates on, declaring the types expected of any concrete mapper class. The actual functionality is in the mapper class below.

```
template <typename I, typename K, typename V,
          typename Args=void,
          input_tag tag=input_unaware>
class mapper_traits {
  typedef I input_type;
```
type of input to the map function

```
  typedef K key_type;
```
type of the key output of the map function

```
  typedef V value_type;
```
type of the value output of the map function

```
  typedef Args arg_type;
```
(optional parameter) type of args passed to the mapper constructor

```
  input_tag mapper_tag = tag;
```
(optional parameter) tag defining the type of mapper constructed

```
};
```

The core interface that any implementer will need to define (NOTE: this is simply a prototype example and not an abstract base class).

```
class MyMapper : public mapper_traits<I, K, V> {
  // Function which is called at the start of processing of some unit of work.
  // When an input splitter is used, this is called on the boundary of each
  // work unit to allow for communication of data at the granularity of
  // the source work unit. Calling this should happen at least once per
  // mapper, but may be called in a cycle which looks like:
```

```
  // start() -> map(), map(), map() -> flush()
  //
  // This is primarily to allow for setting up data shared between mapper
  // inputs but which needs to be flushed or stored as part of a commit of
  // some data (e.g. statistics counters, or some additional output)
  void start();

  // main worker function of the mapper class.
  // input - represents the input value being passed to the mapper by the
  //   calling library
  // OutIter - output iterator which receives the mapper output. expected that
  //   OutIter takes pair<key_type, value_type> as input.
  template <typename OutIter>
  void map(const I& input, OutIter output);

  // optional behavior which allows the calling library to abort the
  // mapper in case it is no longer needed. this will cause the map
  // function to stop executing its current input and become ready for
  // execution again. the caller may call flush if abort() returns true.
  // false implies the mapper cannot abort. may block the caller until
  // abort is successful.
  // Returns - whether abort was successful or not
  bool abort();

  // Finalizer function which is called once per work unit (per input provided
  // to the map reduce library. Used to flush internal state accumulated
  // per work unit.
  void flush();
};
```

Reducer Class Definition

Traits base class used to define a set of traits required of any reducer class (mainly typedefs). This class is not required by implementers but is convenient to use.

```
template <typename K, typename V, typename O,
          typename Args=void,
          input_tag tag=input_unaware>
class reducer_traits {
  typedef K key_type;
```

key type of the input to the reducer (also the key type of the output)

```
  typedef V value_type;
```

value type of the input to the reducer

```
  typedef O output_type;
```

value type of the output of the reducer

```
  typedef Args arg_type;
```

(optional parameter) type of constructor args to the reducer

```
  input_tag reducer_tag = tag;
```
(optional parameter) tag defining the type of reducer constructed
```
};
```

The core interface that any implementer will need to define.
```
class MyReducer : public reducer_traits<K, V, O> {
  // this is called once per unit of work before reduce is called for
  // any values in that work unit.
  // shard_id - represents an id which indicates which worker this is
  //   to allow for optimizations based on which worker is active.
  //   shards are assumed to be a zero-indexed list of workers from
  //   0..num reducer workers.
  // is_combiner - a special bit which indicates that the reducer is being
  //   used as a combiner and to potentially behave differently as a result
  //   (combiners require that the reduce function be associative and
  //   commutative, as combiners may only see a subset of data and will be
  //   asked to reduce other pre-reduced values).
  void start(size_t shard_id, bool is_combiner);

  // reduce function which takes a set of input values and processes
  // them as a unit. this receives input as an iteration over a set of
  // values and is expected to call the output = with a
  // pair<key, output_value> for each output value.
  template <typename InIter, typename OutIter>
  void reduce(
      const K& key,
      InIter* value_start, InIter* value_end,
      OutIter output);

  // this is the same as the abort function on the mapper class and should
  // return true if abort succeeded on the current reducer input.
  bool abort();

  // like the mapper flush function, this is called at least once in the
  // lifetime of the reducer, but should generally be called once per
  // work unit. the work unit is defined at the level of a shard of
  // data, which is an implementation specific unit of work optimization.
  void flush();
};
```

Map-reduce options class

```
      template <typename Mapper, typename Reducer, typename OutIter>
      struct map_reduce_options {
        // Args to be passed into the mapper for construction
        Mapper::arg_type* mapper_args;
        // Args to be passed into the reducer for construction
        Reducer::arg_type* reducer_args;
      };
```

Configuration class which contains implementation specific configuration for the map-reducer library. Implementations may include things like
* number of mapper tasks
* number of reducer tasks
* thread pool to execute from
* mapper args
* reducer args
* configuration files

map_reduce class is templatized on the mapper/reducer classes as well as some helpers which define how sharding is handled and if input needs to be split before processing. Uses a thread-pool to create mapper tasks. all the configuration of the map-reduce is done via the template parameters, including the input/intermediate/output types.

Must override shard_fn if the key type does not have a std::hash specialization defined or you want to use a non-hash based sharding function.

```
template <typename Iter,
          typename OutIter,
          typename Mapper,
          typename Reducer,
          typename Combiner = identity_reducer<typename Reducer::key_type,
                                               typename Reducer::value_type>,
          typename shard_fn = default_shard<typename Mapper::key_type>,
          typename input_splitter =
              identity_splitter<typename Iter::value_type,
                                typename Mapper::input_type> >
class map_reduce {
 public:
  map_reduce(
      const map_reduce_options<Mapper, Reducer, OutIter>& opts)

  bool run(Iter iter, Iter end, OutIter out_iter);
```

Blocking interface to execute the map-reduce library. this function does all the work of setting up the mapper and reducer tasks, defining work for the reducers, shuffling data, and waiting for all data to be processed. The contract is that when this function completes, all values between iter and end will have been fully processed by the entire map reduce pipeline and all outputs will have been generated for all input.
Returns true if the map-reduce completed successfully.

```
  void run_async(Iter iter, Iter end);
```

Same as the run interface, but returns immediately after the map-reduce has started and will not block waiting for the mapreduce to complete

```
  bool wait();
```

Blocks until run_async() completes. acts as a noop if run() has been called.
Does nothing if the map-reduce operation has completed.
Can be used for synchronization on completion of the map-reduce.
Returns true if the map-reduce completed successfully.

```
  bool done();
```

Returns true if the mapreduce is completed (can be used for to check for completion of the map-reduce as part of coordination or synchronization)

```
};
```

Template Params

Iter - iterator type over input
OutIter - output iterator where reduced values will go
Mapper - mapper class which supports start/map/flush functions
Reducer - reducer class which supports start/reduce/flush functions

Combiner (optional) - reducer used to combine values prior to sending to the shuffler
shard_fn (optional) - sharding function which defines which values go to which reducers. defaults to using a hash-based sharding function and assumes that std::hash is defined for the key type
input_splitter (optional) - special function which can take the value type of the input iterator type and generates values which can be consumed by the map function. this defaults to an identity_splitter which does nothing to the input and passes untouched to the mapper function.

Input Aware Mapper and Reducer (optional)

There are cases where the mapper/reducer care about the raw input which they are processing (needing to know information about the files or other information encoded directly in the input). To handle this case, there is a special input-awareness tag which indicates whether or not the mapper/reducer care about the raw input (e.g. the filename rather than the contents of the file).

Input Tagging enum

```
enum input_tag {
  // capability which indicates the mapper/reducer wants a reference to the
  // input passed to map-reduce as well as to the actual mapper input,
  // providing some additional information about the source data (e.g.
  // filename, work task identifier). The data passed is implementation
  // specific.
  input_aware,
  // this indicates that the mapper/reducer does not care about the source
  // data, but may/may not care about the start of the work unit,
  // so the start/flush calls are still made by the framework, but the raw
  // data is not passed to the map() and reduce() functions.
  input_unaware,
};
```

There is an alternative version of the mapper and reducer classes which provides more visibility into the raw data and input metadata. For distributed mapreduce, a number of optimizations can be made when the mapper and reducer have some metadata, including things like shard id, filenames, directories, or reader objects. The google distributed mapreduce provides access to this information directly, though they complicate the interface significantly and are very implementation specific. The difficulty about making this generic is to provide access to such implementation specific metadata. Google's implementation places this metadata in a mapper-input object, but this API modifies this approach a bit by providing access to the raw work unit passed into the map reduce framework. The contents of this will vary. For a file based input, this may be as simple as the source filename, or for a fully distributed implementation, may contain filename, file offset, temp directory, shard id, etc.

This variation on the mapper and reducer is the same as the input_unaware version, but with a single extra parameter.

```
class MyMapper : public mapper_traits<I, K, V> {
 public:
  void start();
  bool abort();
  void flush();

  // main worker function of the mapper class.
  // raw_input - reference to the input passed into the mapreduce framework
  //   with the same type as Iter::value_type for the Iter type passed into
  //   the map_reduce class.
  // input - represents the input value being passed to the mapper by the
  //   calling library
  // OutIter - output iterator which receives the mapper output. expected that
  //   OutIter takes pair<key_type, value_type> as input.
  template <typename RawInputType, typename OutIter>
  void map(const RawInputType& raw_input, const I& input, OutIter output);

};

class MyReducer : public reducer_traits<K, V, O> {
 public:
  void start(size_t shard_id, bool is_combiner);
  bool abort();
  void flush();

  // raw_input - raw input work unit passed into the reducer, which contains
  //   data required to generate value_start and value_end (e.g. the set of
  //   keys to be processed by this reducer and the map of all data, or the
  //   filename/offset being processed by the reducer)
  // key - next key to be processed by the reducer
  // value_start - iterator pointing to the first value for key
  // value_end - ending value iterator
  // output - output iterator to which output is sent from the reducer
  template <typename RawInputType, typename InIter, typename OutIter>
  void reduce(
      const RawInputType& raw_input,
      const K& key, InIter* value_start, InIter* value_end,
      OutIter output);
};
```

Construction

The construction of mapper and reducer object is generally assumed to use the default constructor since mappers and reducers are generally stateless. But, there is a facility which allows an arg type to be defined by the mapper and reducer (default is a void argument). These values can then be added to the map_reduce_options object which can pass arguments to the mappers/reducers at construction time.

Note that this type can be left as void and no constructor provided and the library should be able to detect this case and not call the constructor if a constructor is not available.

Distributed MapReduce

Much of the discussion about the proposed map_reduce framework talks about the ability to provide a distributed implementation without changing the programmer implementation significantly.

The proposed interface supports distributed map-reduces fairly easily as well as a number of distributed systems optimizations and fault tolerance mechanisms.

To implement the distributed map-reduce using the word counting example above, would be a fairly straight-forward extension of the threaded version. The code below shows how mapreduce would be initialized assuming a slightly modified map_reduce controller class to better handle the change in input and output formats.

Note, the major changes here are the addition of a set of serialization functions and the removal of the input and output iterators. These are necessitated by the fact that I/O is external to the class (largely through files or distributed files) and the values are byte or text strings instead of the raw types (this interfaces with the internal C++ types through the serialization functions).

```
// default_serializer relies on a deserialize<T>() or serialize<T>()
// functions exist for all the template types (PODs should be predefined).
template <typename input_type,
          typename key_type,
          typename value_type,
          typename output_type>
class default_serializer {
  input_type deserialize_input(istream& input);
  key_type deserialize_key(istream& input);
  value_type deserialize_value(istream& input);

  string serialize_key(ostream& input);
  string serialize_value(ostream& input);
  string serialize_value(ostream& input);
};

template <typename Mapper,
          typename Reducer,
          typename Combiner = identity_reducer<typename Reducer::key_type,
                                               typename Reducer::value_type>,
          typename Serializer =
              default_serializer<Mapper::input_type,
                                 Mapper::key_type,
                                 Mapper::value_type,
                                 Reducer::output_type>
          typename shard_fn = default_shard<typename Mapper::key_type>,
          typename input_splitter =
              identity_splitter<string,
                                typename Mapper::input_type> >
class map_reduce {
 public:
  map_reduce(const map_reduce_options& opts)

  bool run();

  void run_async();
  bool wait();
  bool done();
};


int main() {
  // Read configuration from some standard location - this can be via file, or
  // via configuration params, but will contain information about the
  // input/output files, including file formats, filenames, and file chunks
  // (if the file is too large to be processed by one mapper). This should
  // also contain a location about where to place the sharded shuffler
  // output files.
  map_reduce_options opts;
  opts.load_from_file(<options_file>);
  map_reduce_job_options job_opts;
  job_opts.load_from_file(<job_options_file>);

  // Start the task based on the config generated for it.
  // For the controller job, this will read configurations and launch all
  // the worker tasks with a modified version of the config which identifies
  // the task type and run separate code.
  // NOTE: this also uses SumReducer as a combiner to reduce the communication
  // from the mappers.
  map_reduce<WordMapper, SumReducer, SumReducer> mr(opts, job_opts);
  mr.run();
}
```

Staging Jobs

The key to making this system work on a distributed system is that the binary used for all machines is the same. The special piece which makes this work is the special job_opts configuration which tells the map-reducer what mode to run the binary in (a mode variable defines what job type is being run, PARENT, CONTROLLER, MAPPER, MAPPER/REDUCER, REDUCER, SHUFFLER). This mode means that the run() command actually can execute as a controller, mapper/combiner, shuffler, reducer, depending on what task the job is given.

This bootstrap mechanism is why the map-reduce job is configured with the templates and run in the manner above, since it creates a binary which can easily be replicated and used for all operations without significant overhead. Doing it this way has some disadvantage when writing a map-reduce pipeline requiring running several jobs in a sequence, but this case can be handled with only some additional work by introducing a parameter defining which map-reduce configuration to use (and the pipeline declares itself as PARENT to force it to also stage CONTROLLER jobs).

It is the task of the controller to create the right number of jobs of the other types and create job_opts for each of them which places each job in the right running mode to complete their allocated work (and report back to the controller). When all work for a given stage (mapper/combiners, shufflers, reducers, output sort) is complete, the controller tells the relevant jobs to shut down. When all stages are complete and output has finished, the controller job completes and shuts itself down.

A simple run-through of how this works is as follows:

- PARENT creates input and stages a CONTROLLER job with some set of job_options (parent and controller can be the same)
- CONTROLLER stages MAPPER jobs and passes job-specific parameters to each.
- CONTROLLER stages SHUFFLE jobs
- MAPPER jobs ask for tasks from CONTROLLER and execute each work task serially, writing output to temp files, failed jobs or work is re-executed
- When all MAPPER jobs complete, any duplicate jobs are cleaned up, CONTROLLER stages the SHUFFLE jobs and provides pointers to the outputs of the MAPPER jobs, SHUFFLE output is written as sorted (and sharded) maps to a new set of temp files
- When all SHUFFLE jobs complete, they are shut down by the CONTROLLER and CONTROLLER stages REDUCE jobs
- REDUCE jobs ask for input shards from CONTROLLER, REDUCE jobs process key-value lists, and output to final output files (any failed REDUCE shards are restarted on another reducer)
- When REDUCE jobs complete, CONTROLLER shuts down REDUCE jobs and shuts itself down, and returns if everything completed successfuly.

A few features which are hidden from the caller which are generally necessary in a distributed context are:

- Fault tolerance support at the work-unit level, allowing the framework to drop output from failed workers. The framework would handle this by watching the workers and re-enqueueing work which failed to complete.
- Duplicate work processing/deduplication per work-unit with worker abort, this would mean allowing some work units to be run by multiple workers. The framework would do this by watching for slow workers and enqueueing duplicate tasks and dropping the mapper/reducer outputs for duplicate work units.
- **Specialized interface to source work-unit information via capabilities model (e.g. mapper which receives work-unit metadata in addition to the input value), for worker and input specific optimizations.

Differences from the Google Implementation

The above interface is focused on an in-memory version of the map-reduce library with concessions for the distributed form of it. There are a number of interface changes which were intended to simplify the interface and abstract out a number of google-specific changes, including the following:

- Keys/Values in this library are not restricted to string types (this generalizes the concept such that the underlying serilization scheme is separated from the functional work of the mappers/reducers)
- The Google implementation allows registration of mappers and reducers, so a single binary can link countless mappers and reducers, which allows runtime configuration.

Pre-compiled template versions are less flexible, but knowing the set of possible mapper/reducer combinations are compile time is not highly restrictive and removes the indirection and virtual methods required by registration (improving performance for highly performance sensitive applications).

- An async version of the mapreduce has been added, though this has less applicability in a distributed context, it enables an application to launch multiple simultaneous map-reduces and to join on these operations at a later stage in the application.
- There is a set of features where mappers/reducers can get metadata about themselves and the data which they are processing (filenames, shard id, file readers/writers, output directories). This is all used to provide mechanisms to optimize the logic, but makes the code very implementation specific, so it has been dropped and included in the mapper_tag and reducer_tag traits which allow external data to be passed into the mapper optionally. This still requires that the implementer know the data format and details about the implementation, but is abstracted out from the core API.
- There are a couple of specialized functions which have some utility but are hard to map into this framework, including:
    - output_to_every_reducer - this provides the ability to send a particular key-value pair to all work units sent to the reducer. doing so allows for special tracking and counting mechanisms to be built, but doesn't fit well with the output-iterator abstraction. this can be emulated by adding sentinel keys which provide metadata, or by creating an out-of-band communication mechanism keyed on the shard_id.
    - secondary_key - this allows there to be a value which isn't in the key but participates in how the data is sorted for processing. This is useful for having a particular ordering of data sent to the reducer phase, but doesn't map well into the simple Key-Value mapping function. There is no way to emulate this without making the keyspace significantly more complicated and incompatible with some C++ containers, so implementing this is left as a special feature of the implementation (e.g. exposing the value_comp in the multimap container)

Code Location

A working threaded implementation of the mapreduce library can be seen as part of the google-concurrency-library at:

https://code.google.com/p/google-concurrency-library/source/browse/include/map_reduce.h

With some simple tests/examples at:
https://code.google.com/p/google-concurrency-library/source/browse/testing/map_reduce_test.cc

The coded version is still under ongoing development and may not strictly match the written definition here, but provides at least a working understanding of how the system can be implemented.

Standard Library Interoperability

Because of the design of the current proposal (which uses iterators on input and output), the framework is compatible with a number of standard library mechanisms. Note though that the mappers require that their input iterator be thread-safe, so it assumes that there will be better thread-safe data structures for this purpose (e.g. the queue interface proposed in N3533 and others). The same is true of the output stage, which requires a thread safe output map (in the example this is a blocking map, though specialized output maps would be more efficient here).

The expectation is that this will use a currently as yet un-standardized executor interface, which would be passed in via the mapreduce options class. The framework could create its own threads as well, but having an executor/thread pool interface provides greater control and the current implementation assumes one.

For a distributed version of this interface, the library will also have to manage launching tasks on remote machines, but that is out of scope of this proposal.

Conceptual Variations

These variations are possible forms of the mapreduce class which have been discussed, primarily as simplifications of the interface, though most of these come at the cost of making this library no longer re-usable in distributed processing.

Object Passing

The proposed interface above defaults to mapper and reducer classes have a default constructor, but can initialize a single-parameter arg pointer by specifying a non-void arg_type in the template (or using some sort of variadic template args for construction). This allows the library to construct mapper/reducer objects on the fly. Moreover this type of approach is compatible with distributed map-reduce which would need to construct mapper and reducer objects on each remote machine using serialized objects.

An alternative to this is to have the mapper and reducer classes by copy-constructable and a prototype object be passed into the framework at initialization, allowing the objects to be more easily constructed and initialized. Having a copy-constructable mapper, would, for example allow for shared state to be easily created in the mapper constructor. It also enables mappers which could receive lambdas which would simplify the framework further to allow callers to be able to write map-reduces without writing a mapper or reducer class.

For example, this potentially enables a lambda-based map-reduce, something like the following (note this would actually require the ability to have template-based lambdas to work properly, as the input and output iterators are not fixed types):

```
map_reduce<queue::iterator, queue::iterator, lambda_mapper<int, int, float>,
lambda_reducer<int, float, float>> mr(opts,
 lambda_mapper<int, int, float>::create([] (int i) { return pair<int,
float>(i, sin(i))),
 lambda_reducer<int, float, float>::create([](int k, ...) {} ));
```

In fact, such an approach would allow for a functional variant of the mapreduce class which could perform the mapreduce operation without any notion of threading behavior (basically an algorithm which operates on iterators and outputs to an iterator using lambda mapper and reducers, similar to how Python can map/reduce over any iterable structure).

The big disadvantage of the copy-construction approach is that it makes a distributed implementation of the framework infeasible as the class instances would need to be serialized somehow.

Combiner and input_tag implicit

There is some complexity added to the interface because of the input_tag enumeration which allows for multiple implementations of the mapper class. There has been some discussion about making these concepts implicit such that the the presence of an input-aware map() function can be called using an enable_if style block (this could also be used to automatically define a combiner when a reducer declares itself as combinable). This was left out but is worth additional discussion.

Input Value Iteration

The proposed implementation provides two dimensions of iteration, the framework receives a begin and end iterator when run, but also supports an input_splitter concept which allows each input value to be divided into multiple inputs to the map-function. This is done to allow the caller to enqueue larger work units (e.g. a file chunk, or a range of values) for a number of reasons (particularly for performance reasons).

An alternative interface doesn’t expose the input_splitter to the template interface of the map_reduce class and instead asks the input iterator to wrap the source input in an iterator-over-iterators. Doing so hides the source unit of work to the map reduce framework, effectively making the input appear to be an iterator over the input-type of the mapper, rather than another unit.

An example of this is the source providing an iterator over filenames (files containing lines of text), and mappers receiving a single line of text from each filename. With the input-splitter in the template, the map-reduce framework would read a filename, then the splitter would read the file one line at a time, passing the values into the mapper one line at a time. For the iterator-of-iterators approach, the framework would receive an iterator over lines of text and the iterator would be responsible for internally reading source values and converting the files into an iterator of map input values.

The difference between these approaches seems nominal except the iterator-over-iterators is logically simpler for the framework, but it also hides the source work unit from the map-reduce framework, which imposes a number of problems for optimization. Particularly, the ability to track and change behavior at a work-unit level becomes impossible. The distributed versions of map-reduce, for example, use the chunking of inputs to implement fault-tolerance on a per-work-unit level. Threaded map-reduce can use the work-unit concept to keep statistics, implement more performance friendly work progress tracking, and to allow for flushing and reset operations at a natural unit of granularity. As such, the flexibility of the proposed interface outweighs the slight increase in interface complexity.

Changes from N3446

- The implementation and justification sections have been simplified and clarified to explain more of the background behind mapreduce, as well as with some more examples
- More detail of how a distributed system version of this has been added
- The semantics of the staging operations have been further clarified
- A more distributable example has been used and is used throughout the document
- Added a pointer to the code