COMP 322: Fundamentals of Parallel Programming

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Lecture 24: Map Reduce

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

Acknowledgments for Today's Lecture

- Lecture 24 handout
- Slides from MapReduce lecture in Stanford CS 345A course
  - http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt
- Slides from COMP 422 lecture on MapReduce
  - http://www.clear.rice.edu/comp422

Announcements

- HW5 submission deadline postponed to 5pm on Monday, March 21st

HW5: Review of Table 1 from Lecture 19

`j.u.c.atomic` Class and Constructors	`j.u.c.atomic` Methods	Equivalent HJ `isolated` statements
AtomicInteger	int j = v.**get**();	int j; `isolated` j = v.val;
	v.**set**(newVal);	`isolated` v.val = newVal;
AtomicInteger()	int j = v.**getAndSet**(newVal);	int j; `isolated` { j = v.val; v.val = newVal; }
// init = 0	int j = v.**addAndGet**(delta);	`isolated` { v.val += delta; j = v.val; }
	int j = v.**getAndAdd**(delta);	`isolated` { j = v.val; v.val += delta; }
AtomicInteger(init)	boolean b = v.**compareAndSet**(expect,update);	boolean b; `isolated` if (v.val==expect) {v.val=update; b=true;} else b = false;
AtomicIntegerArray	int j = v.**get**(i);	int j; `isolated` j = v.arr[i];
	v.**set**(i,newVal);	`isolated` v.arr[i] = newVal;
AtomicIntegerArray	int j = v.**getAndSet**(i,newVal);	int j; `isolated` { j = v.arr[i]; v.arr[i] = newVal; }
(length) // init = 0	int j = v.**addAndGet**(i,delta);	`isolated` { v.arr[i] += delta; j = v.arr[i]; }
	int j = v.**getAndAdd**(i,delta);	`isolated` { j = v.arr[i]; v.arr[i] += delta; }
AtomicIntegerArray	boolean b = v.**compareAndSet**(i,expect,update);	boolean b; `isolated` if (v.arr[i]==expect) {v.arr[i]=update; b=true;} else b = false;
(arr)		

HW5 Clarifications

- Clarification 1: You can ignore the possibility of queue overflow in class IQueue.
- Clarification 2: Remember to take AtomicInteger.get() operations into account along with compareAndSet() operations when considering serialization edges in Problem 3.
- Clarification 3: A do-while loop in Java executes the loop body at least once, and only exits the loop when the while condition is false.
- Clarification 4: Problem 1) asks for an expansion of the compareAndSet() calls in accordance with Table 1 of the Lecture 19 handout. The isolated statement should only enclose the compareAndSet computation and nothing more.

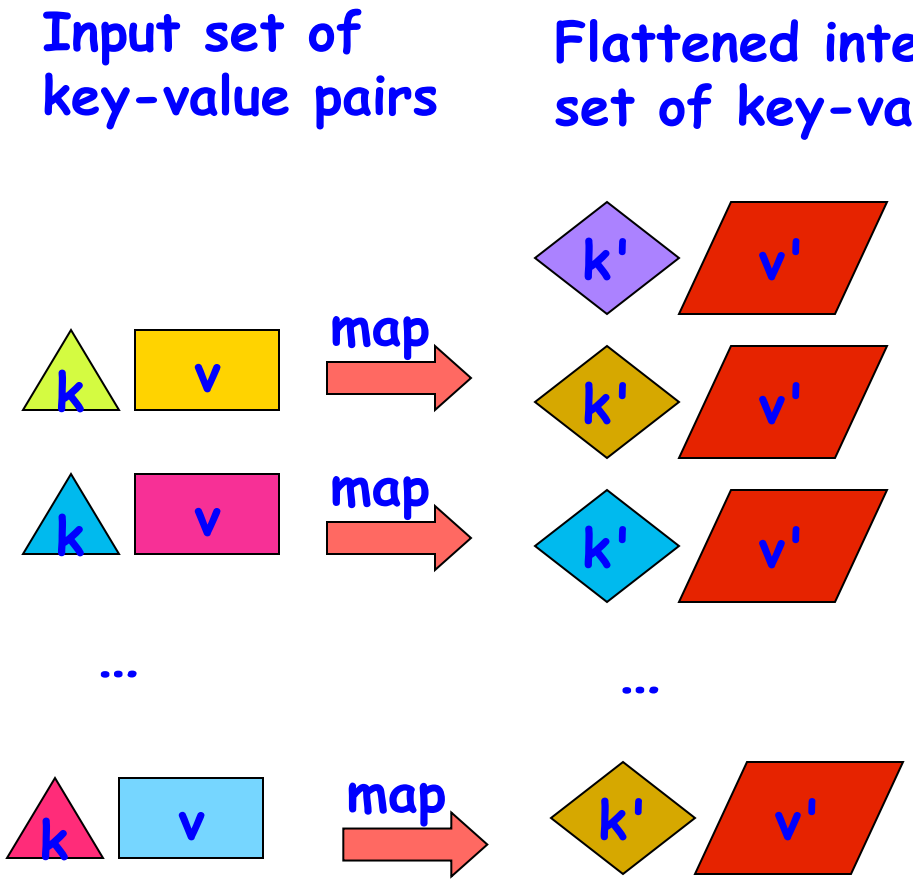
Recap: map and reduce (fold) functions in Scheme

- (map f (list x1 ...xn)) = (list (f x1)...(f xn))
  - (map f L) takes two parameters as inputs, a unary function, f, and a list, L, and returns a new list obtained by applying f to each element in L.
  - All applications of function f can be performed in parallel. If each application of f takes O(1) constant time, then WORK = O(n) and CPL = O(1).
- (foldr g base (list x1 ...xn)) = (g x1 ...(g xn base))
  - (foldr g base L) takes three parameters as inputs, a binary function, g, a base (init) value, and a list, L. It returns a right-associative reduced value obtained by applying g on elements of L.
  - If we don't know anything about function g, then we have to assume that it must be applied sequentially as shown above.
  - If g is associative, it can be computed using parallel reduction algorithms with WORK = O(n) and CPL = O(log n).
  - For today's lecture, we will assume that all functions used for reduce operations are both associative and commutative.

Sets of Key-Value Pairs

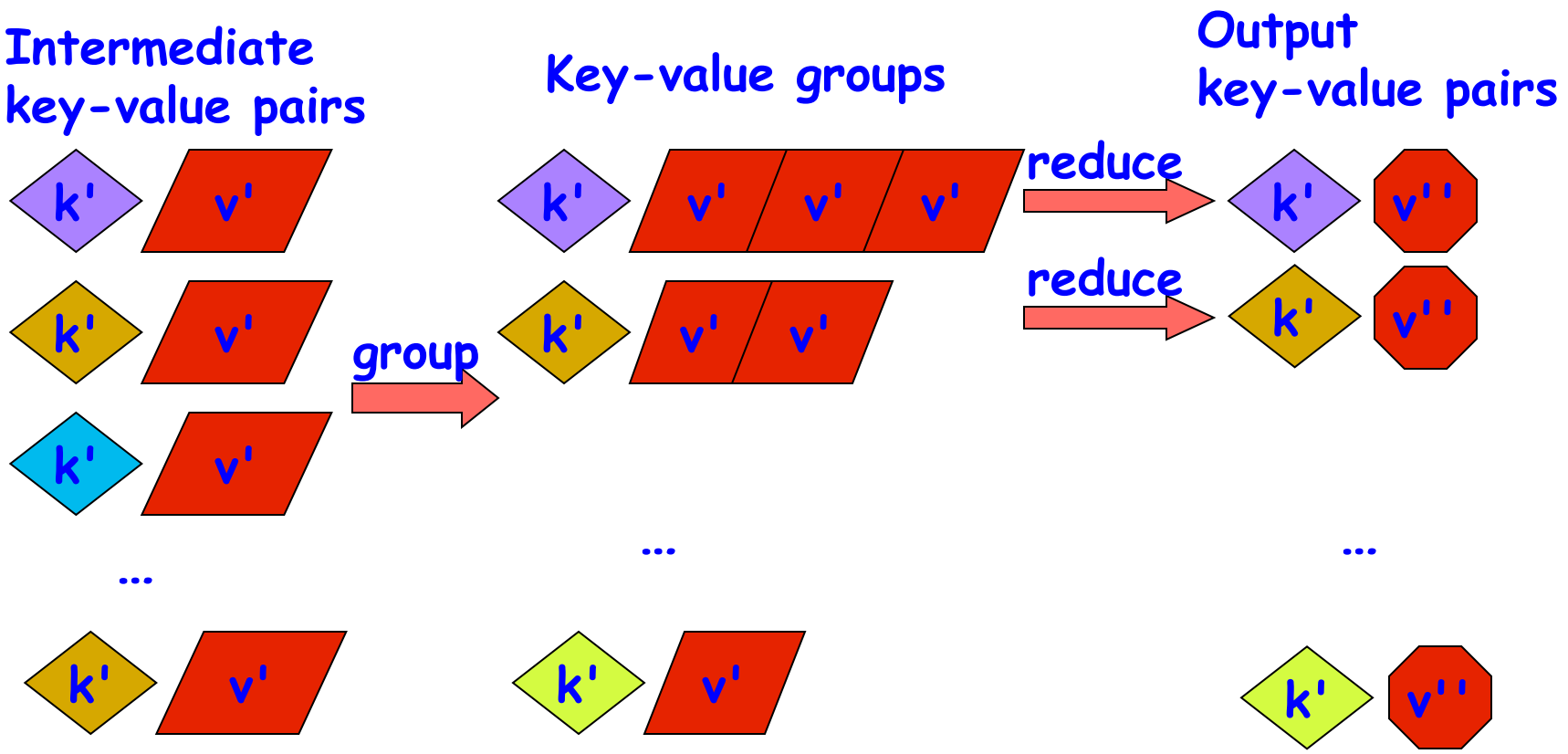
- *Input set* is of the form {(k1, v1), . . . (kn, vn)}, where (ki, vi) consists of a key, ki, and a value, vi.
  - Assume that the key and value objects are immutable, and that equality comparison is well defined on all key objects.
- Map function f generates sets of *intermediate key-value pairs*, f(ki,vi) = {(k1' ,v1'),...(km',vm')}. The kj' keys can be different from ki key in the input of the map function.
- Assume that a *flatten* operation is performed as a post-pass after the map operations, so as to avoid dealing with a set of sets.
- *Reduce* operation groups together intermediate key-value pairs, {(k', vj')} with the same k', and generates a reduced key-value pair, (k',v''), for each such k', using reduce function g

MapReduce: The Map Step



Source: http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt

MapReduce: The Reduce Step



Source: http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt

WordCount example (Listing 1)

1. Input: set of words
2. Output: set of (word,count) pairs
3. Algorithm:
4. a) For each input word W, emit (W, 1) as a key-value pair (map step).
5. b) Group together all key-value pairs with the same key (reduce step).
6. c) Perform a sum reduction on all values with the same key (reduce step).

- All map operations in step a) (line 4) can execute in parallel with only local data accesses
- Step b) (line 5) can involve a major reshuffle of data as all key-value pairs with the same key are grouped together.
- Step c) (line 6) performs a standard reduction algorithm for all values with the same key, and in parallel for different keys.

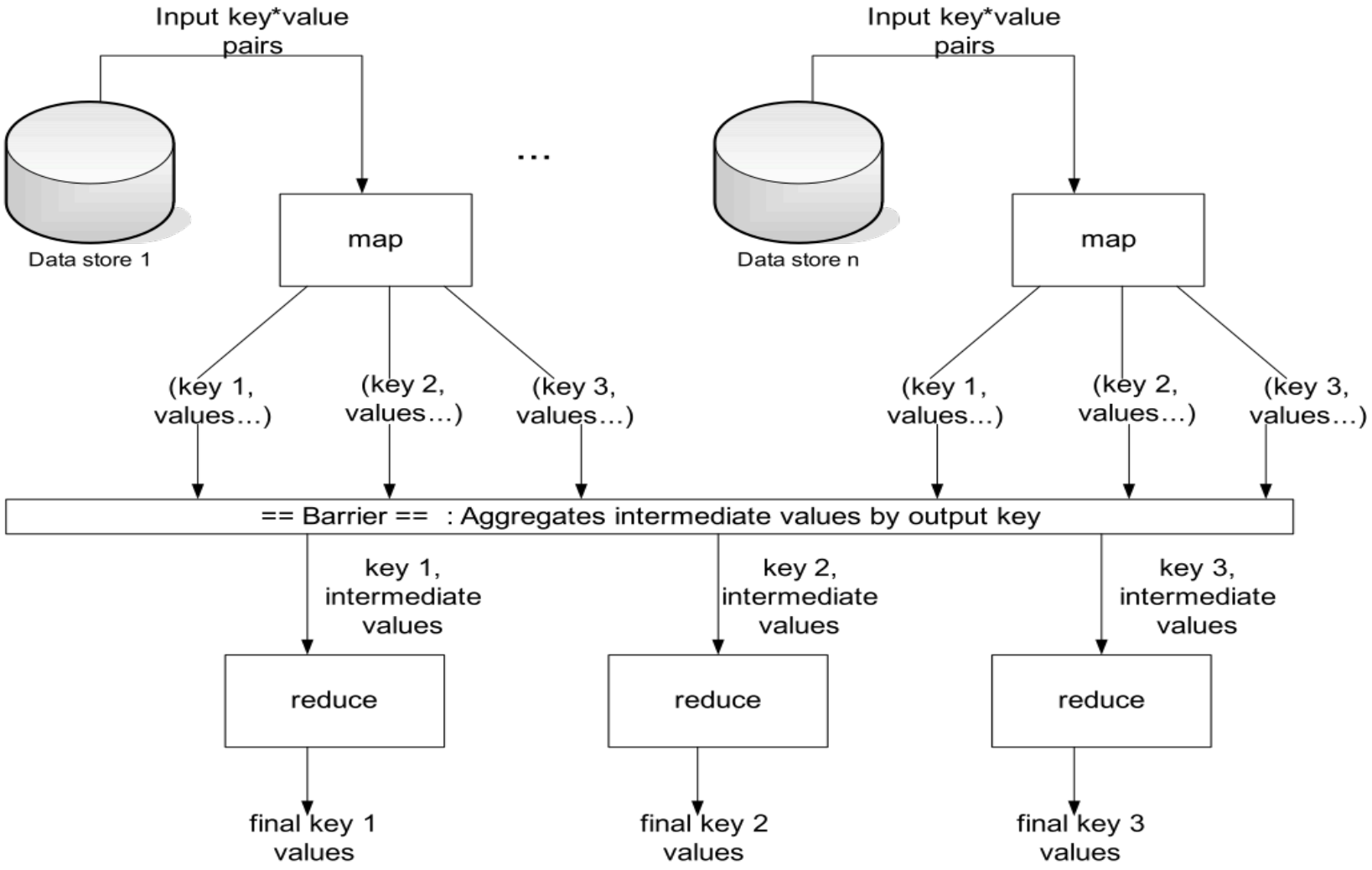
Motivation: Large Scale Data Processing

- Want to process terabytes of raw data
  - documents found by a web crawl
  - web request logs
- Produce various kinds of derived data
  - inverted indices
    - e.g. mapping from words to locations in documents
  - various representations of graph structure of documents
  - summaries of number of pages crawled per host
  - most frequent queries in a given day
  - ...
- Input data is large
- Need to parallelize computation so it takes reasonable time
  - need hundreds/thousands of CPUs
- Need for fault tolerance

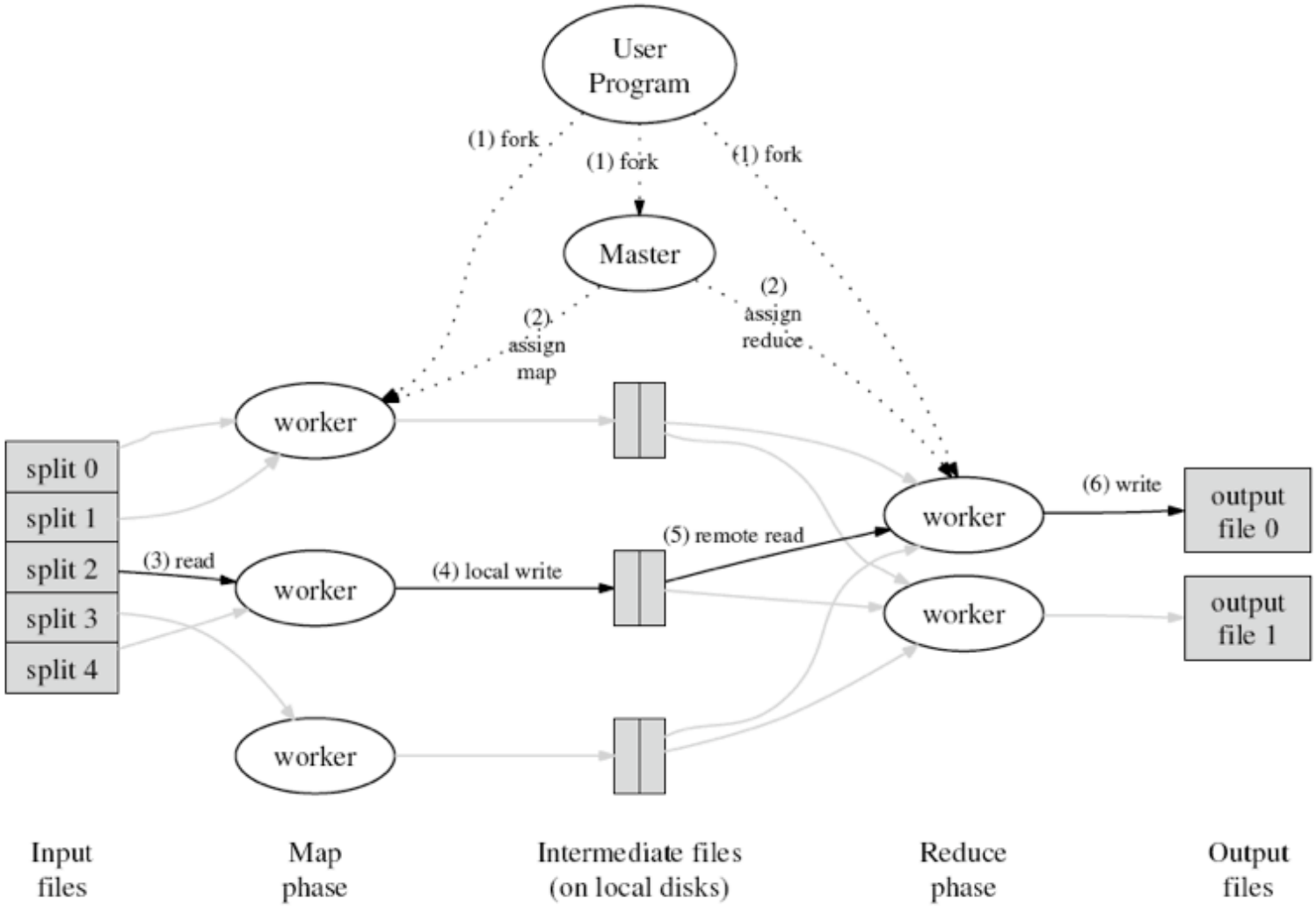
Example applications of MapReduce in Data Center Clusters (Table 1)

Application	Map function	Reduce function
Distributed grep	emit line if it matches pattern	no-op (copy intermediate data to output data)
URL access frequencies	emit (URL,1) pairs from web logs	add counts for same URL, emit (URL,total-count) pairs
Reverse web-link graph	emit (target,source) pairs for each target link found in source page	concatenate source URLs for same target, emit (target,source-list) pairs
Distributed sort	emit (key,record) pairs	no-op (records will be grouped by key)

Overall schematic for MapReduce framework on a data center cluster



Execution Overview



Execution Overview Details

1. Initiation — the MapReduce library the splits input files into M pieces (typically 16-64 MB per piece), and starts up program on a cluster with 1 master and W workers
2. Master assignment — the Master node assigns M map tasks and R reduce tasks to the workers. Typical values are M = 200,000 and R = 5,000 for W = 2,000.
   - The master attempts to assign tasks to workers that are located close to desired input data (locality management).
3. Map task — a worker assigned a map task parses key-value pairs from input data, invokes the map function on each pair, and produces intermediate key-value pairs.
4. Partition — the intermediate key-value pairs are partitioned into R regions for R reduce tasks.
5. Group — each worker uses Remote Procedure Calls (RPC) to read intermediate data from remote disks, after which it sorts its set of pairs by key.
6. Reduce — the worker iterates over sorted intermediate data, calls reduce, and appends output to final output file
7. Completion — when all is complete, user program is notified

Full "Word Count" Example: Main Program

```
#include "mapreduce/mapreduce.h"

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);
  MapReduceSpecification spec;

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }
  // Specify the output files:
  // /gfs/test/freq-00000-of-00100
  // /gfs/test/freq-00001-of-00100
  // ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");

  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.

  return 0;
}
```


Full "Word Count" Example: Map

```
#include "mapreduce/mapreduce.h"

class WordCounter : public Mapper {
public:
   virtual void Map(const MapInput& input) {
      const string& text = input.value();
      const int n = text.size();
      for (int i = 0; i < n; ) {
         // Skip past leading whitespace
         while ((i < n) && isspace(text[i])) i++;
        // Find word end
         int start = i;
         while ((i < n) && !isspace(text[i])) i++;
         if (start < i) Emit(text.substr(start,i-
start),"1");
      }
   }
};
REGISTER_MAPPER(WordCounter);
```


Full "Word Count" Example: Reduce

```
#include "mapreduce/mapreduce.h"

class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (! input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }
    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);
```