Lab 11

MapReduce

MapReduce is a software framework for processing (large[1]) data sets in a distributed fashion over several machines. The core idea behind MapReduce is *mapping* your data set into a collection of <key, value> pairs, and then *reducing* over all pairs with the same key. The overall concept is simple, but is actually quite expressive when you consider that:

1. Almost all data can be mapped into <key, value> pairs somehow, and

2. Your keys and values may be of any type: strings, integers, dummy types... and, of course, <key, value> pairs themselves.

The canonical MapReduce use case is counting word frequencies in a large text, but some other examples of what you can do in the MapReduce framework include distributed sort, distributed search, web-link graph traversal and machine learning.

A MapReduce Workflow

When we write a MapReduce workflow, we'll need 2 scripts: the map script, and the reduce script. Their proper execution will be handled by the MapReduce framework.

When we start a map/reduce workflow, the framework will *split* the input into segments, passing each segment to a different machine. Each machine then runs the *map script* on the portion of data attributed to it.

The *map script* takes some input data, and maps it to <key, value> pairs according to your specifications. For example, if we wanted to count word frequencies in a text, we'd have <word, count> be our <key, value> pairs. Our map script, then, would emit a <word, 1> pair for each word in the input stream. Note that the map script does no *aggregation* (i.e. actual counting) - this is what the reduce script it for. The purpose of the map script is to model the data into <key, value> pairs for the reducer to aggregate.

Emitted <key, value> pairs are then "shuffled" (to use the terminology in the diagram below), which basically means that pairs with the same key are grouped and passed to a single machine, which will then run the *reduce script* over them[2].

The *reduce script* (which you also write) takes a collection of <key, value> pairs and "reduces" them according to the user-specified reduce script. In our word count example, we want to count the number of word occurrences so that we can get frequencies. Thus, we'd want our reduce script to simply sum the *values* of the collection of <key, value> pairs which have the same key[3].

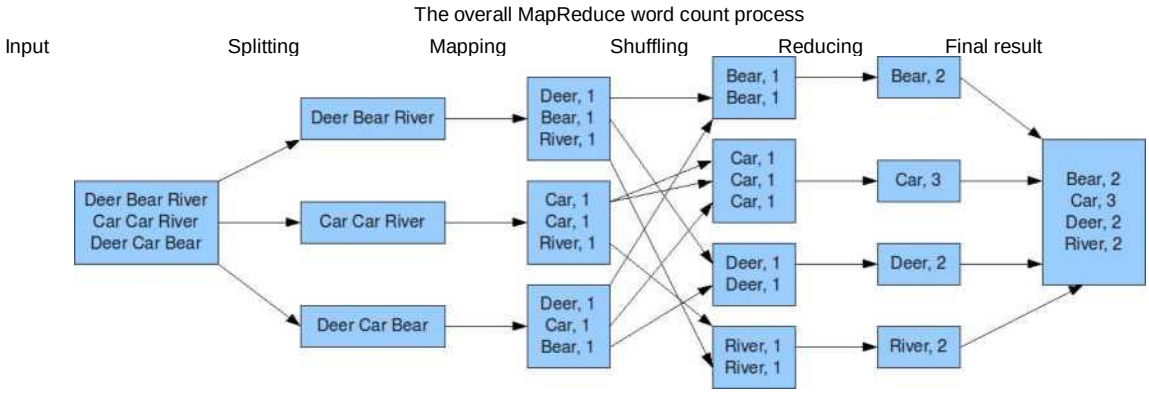
---

[1] The data doesn't **have** to be large, but it is almost always much faster to process small data sets locally than on a MapReduce framework.

[2] Note: this is over-simplified, as the workflow may utilize multiple reduce stages, but it accurately imparts the general idea of what's going on.

[3] Note: do not assume that all <key, value> pairs passed to a reduce script will have the

The diagram below illustrates the described scenario nicely[4].



Want to Read More?

The Google MapReduce paper gives the nitty-gritty details[5]

Java's ForkJoin Framework

A major new feature in Java 5 was the ExecutorService, an abstraction for executing tasks asynchronously. The ThreadPoolExecutor implements this concept using an internally administered thread pool, the worker threads of which work their way through the tasks arising. The ThreadPoolExecutor has a central inbound queue for new tasks (Runnables or Callables), which is shared by all worker threads.

One disadvantage of the ThreadPoolExecutor is that a competition situation can arise when threads try to access the shared inbound queue and the synchronization overhead can waste valuable task processing time. There is also no support for having multiple threads collaborate to compute tasks.

Since Java 7, an alternative implementation of the ExecutorService has been available in the form of the ForkJoinPool. This uses additional structures to compensate for the disadvantages of the ThreadPoolExecutor and potentially permits more efficient task processing.

For learning about basic parallel operations, there are only 2-4 classes you need to know about:

1. **`ForkJoinPool`**: An instance of this class is used to run all your fork-join tasks in the whole program.
2. **`RecursiveTask<V>`**: You run a subclass of this in a pool and have it return a result; see the examples below.

---

same key. If you take a look at the provided reduce.py script, you'll notice that we maintain a *dictionary* of keys. The "shuffling" phase will try to pass pairs with the same key to the same reducer, but due to different rates of computation (from different mappers) and the distributed nature of the workflow (we can just use multiple reduce stages) this will not often be the case.

[4] http://blog.trifork.com/wp-content/uploads/2009/08/MapReduceWordCountOverview.png

[5] http://labs.google.com/papers/mapreduce.html

3. **`RecursiveAction`**: just like `RecursiveTask` except it does not return a result
4. **`ForkJoinTask<V>`**: superclass of `RecursiveTask<V>` and `RecursiveAction`. `fork` and `join` are methods defined in this class. You won't use this class directly, but it is the class with most of the useful javadoc documentation, in case you want to learn about additional methods.

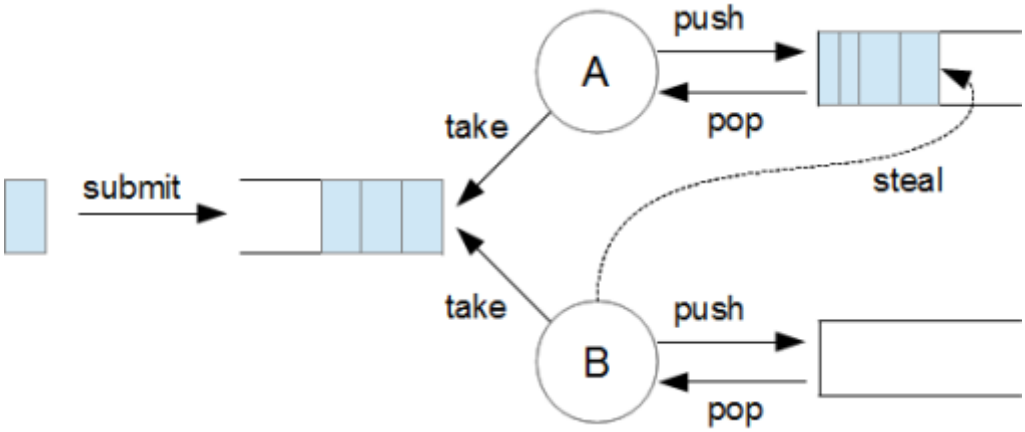
All the classes are in the package `java.util.concurrent`, so it is natural to have import statements like this:

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
```

To use the library, you need a `ForkJoinPool` object. In Java 8, get the object you want by calling the static method `ForkJoinPool.commonPool()`. It is the job of the pool to take all the tasks that can be done in parallel and actually use the available processors effectively.

How it works and structure

Just like the `ThreadPoolExecutor`, the `ForkJoinPool` uses a central inbound queue and an internal thread pool. With the `ForkJoinPool`, however, each worker thread has its own task queue, to which it can add new tasks (see figure 1). As long as its own task queue still contains tasks, each thread sticks to processing only these tasks. When their task queues are empty, threads employ a work-stealing algorithm, in that they search through task queues belonging to other worker threads for available tasks and process them if found. If they fail to find one, they access the shared inbound queue.



A `ForkJoinPool` with two worker threads, A and B. In addition to the inbound queue, each thread also has its own internal task queue. Each thread starts by processing its own queue, before moving on to other queues (figure 1).
This procedure has two potential benefits. Firstly, where possible, each thread works only on its own queue for an extended period of time without having to interact with other threads. Secondly, threads have the ability to take work off each other.

To minimize overhead from the additional queues and from work stealing, the `ForkJoinPool` implementation incorporates major performance enhancements. Two factors are worth stressing:

The worker thread task queues use a *deque* (double ended queue) data structure which supports the efficient addition and removal of elements at both ends. The worker thread works at only one end of the deque, as with a stack. It both places new tasks on (pushes) and fetches tasks for processing from (pops) the top of the stack. When stealing work, the other threads, by contrast, always access the other end of the deque. Consequently, threads usually only face competition when they attempt to steal work from the same queue – and even if that happens, the thread "owning" the queue is rarely affected by that competition.

A potential problem is that, should a thread attempts to steal work fail to find an available task, it could burn CPU time by continuously trawling through the same empty queues. To prevent this, "unemployed" worker threads switch into a rest state in accordance with a fixed set of rules. The framework then addresses them specifically when tasks become available once more.

It is important to note that local task queues and work stealing are only utilized (and therefore only produce benefits) when worker threads actually schedule new tasks in their own queues. If this doesn't occur, the `ForkJoinPool` is just a `ThreadPoolExecutor` with an extra overhead.

One scenario in which tasks schedule further tasks is based on the principle of splitting – if the amount of computing required for a task is very large, it can be recursively split into several small subtasks. Work stealing means that other threads will also contribute to processing these tasks (and will split their own subtasks further where possible). Only if a subtask is deemed to be small enough (according to a defined set of rules), and splitting it further is not worthwhile, will it be computed directly. What makes this approach particularly tempting is that very large subtasks that other threads can then immediately steal are created at the start of the process of recursively splitting tasks. This favors even load balancing between threads.

In another, often underestimated, scenario, tasks schedule further tasks without any splitting. A task then represents a self-contained action, but nonetheless triggers follow-up actions represented by additional tasks. In principle this can be used to model any type of event-driven process. In the extreme case of such a scenario, each worker thread could continually provide itself with further tasks. All threads would be busy but there would still be no competition situations.

An Example

The key for this example, which is hinted at nicely by the name `RecursiveTask`, is that your `compute` method can create other `RecursiveTask` objects and have the pool run them in parallel. First you create another object. Then you call its `fork` method. That actually starts parallel computation — `fork` itself returns quickly, but more computation is now going on. When you need the answer, you call the `join` method on the object you called `fork` on. The `join` method will get you the answer from `compute()` that was figured out by `fork`. If it is not ready yet, then `join` will *block* (i.e., not return) until it is ready. So the point is to call `fork` “early” and call `join` “late”, doing other useful work in-between.

Those are the “rules” of how `fork`, `join`, and `compute` work, but in practice a lot of the parallel algorithms you write in this framework have a very similar form, best seen with an example. What this example does is sum all the elements of an array, using parallelism to potentially process different 5000-element segments in parallel.

```
import java.util.concurrent.ForkJoinPool;
```

```
import java.util.concurrent.RecursiveTask;

class Sum extends RecursiveTask<Long> {
    static final int SEQUENTIAL_THRESHOLD = 5000;

    int low;
    int high;
    int[] array;

    Sum(int[] arr, int lo, int hi) {
        array = arr;
        low   = lo;
        high  = hi;
    }

    protected Long compute() {
        if(high - low <= SEQUENTIAL_THRESHOLD) {
            long sum = 0;
            for(int i=low; i < high; ++i)
                sum += array[i];
            return sum;
         } else {
            int mid = low + (high - low) / 2;
            Sum left  = new Sum(array, low, mid);
            Sum right = new Sum(array, mid, high);
            left.fork();
            long rightAns = right.compute();
// does not block the thread, instead goes on to doing something else
(working on its own queue of tasks or stealing)
            long leftAns  = left.join();
            return leftAns + rightAns;
         }
     }

     static long sumArray(int[] array) {
         return ForkJoinPool.commonPool().invoke(new
Sum(array,0,array.length));
     }
}
```

How does this code work? A `Sum` object is given an array and a range of that array. The `compute` method sums the elements in that range. If the range has fewer than `SEQUENTIAL_THRESHOLD` elements, it uses a simple for-loop like you learned in introductory programming. Otherwise, it creates two `Sum` objects for problems of half the size. It uses `fork` to compute the left half in parallel with computing the right half, which this object does itself by calling `right.compute()`. To get the answer for the left, it calls `left.join()`.

Why do we have a `SEQUENTIAL_THRESHOLD`? It would be correct instead to keep recurring until `high==low+1` and then return `array[low]`. But this creates a lot more `Sum` objects and calls to `fork`, so it will end up being much less efficient despite the same asymptotic complexity.

Why do we create more `Sum` objects than we are likely to have processors? Because it is the framework's job to make a reasonable number of parallel tasks execute efficiently and to schedule them in a good way. By having lots of fairly small parallel tasks it can do a better job, especially if the number of processors available to your program changes during execution (e.g., because the operating system is also running other programs) or the tasks end up taking different amounts of time.

So setting `SEQUENTIAL_THRESHOLD` to a good-in-practice value is a trade-off. The

documentation for the ForkJoin framework suggests creating parallel subtasks until the number of basic computation steps is somewhere over 100 and less than 10,000. The exact number is not crucial provided you avoid extremes.

Gotchas

There are a few “gotchas” when using the library that you might need to be aware of:

1. It might seem more natural to call `fork` twice for the two subproblems and then call `join` twice. This is naturally a little less efficient than just calling `compute` for no benefit since you are creating more parallel tasks than is helpful. But it turns out to be a *lot* less efficient, for reasons that are specific to the current implementation of the library and related to the overhead of creating tasks that do very little work themselves.
2. Remember that calling `join` blocks until the answer is ready. So if you look at the code:

   ```
   left.fork();
   long rightAns = right.compute();
   long leftAns  = left.join();
   return leftAns + rightAns;
   ```

   you'll see that the order is crucial. If we had written:

   ```
   left.fork();
   long leftAns  = left.join();
   long rightAns = right.compute();
   return leftAns + rightAns;
   ```

   our entire array-summing algorithm would have no parallelism since each step would completely compute the left before starting to compute the right. Similarly, this version is non-parallel because it computes the right before starting to compute the left:

   ```
   long rightAns = right.compute();
   left.fork();
   long leftAns  = left.join();
   return leftAns + rightAns;
   ```

3. You should *not* use the `invoke` method of a `ForkJoinPool` from within a `RecursiveTask` or `RecursiveAction`. Instead you should always call `compute` or `fork` directly even if the object is a different subclass of`RecursiveTask` or `RecursiveAction`. You may be *conceptually* doing a “different” parallel computation, but it is still part of the same parallel task. Only sequential code should call `invoke` to begin parallelism. (More recent versions of the library may have made this less of an issue, but if you are having problems, this may be the reason.)
4. When debugging an uncaught exception, it is common to examine the “stack trace” in the debugger: the methods on the call stack when the exception occurred. With a fork-join computation, this is not as simple since the call to `compute` occurs in a different thread than the *conceptual* caller (the code that called `fork`). The library and debugger try to give helpful information including stack information for the thread runnin g`compute` and the thread

that called `fork`, but it can be hard to read and it includes a number of calls related to the library implementation that you should ignore. You may find it easier to debug by catching the exception inside the call to `compute` and just printing that stack trace.

5. In terms of performance, there are many reasons a fork-join computation might run slower than you expect, even slower than a sequential version of the algorithm.

Credits:
https://homes.cs.washington.edu/~djg/teachingMaterials/grossmanSPAC_forkJoinFramework.html
http://www.h-online.com/developer/features/The-fork-join-framework-in-Java-7-1762357.html
http://www.cse.buffalo.edu/~bina/cse487/fall2011/TomWhiteMR.doc
http://hci.stanford.edu/courses/cs448g/a2/files/map_reduce_tutorial.pdf