Introduction to Parallel Programming and MapReduce

Audience and Pre-Requisites

This tutorial covers the basics of parallel programming and the MapReduce programming model. The pre-requisites are significant programming experience with a language such as C++ or Java, and data structures & algorithms.

Serial vs. Parallel Programming

In the early days of computing, programs were *serial*, that is, a program consisted of a sequence of instructions, where each instruction executed one after the other. It ran from start to finish on a single processor.

Parallel programming developed as a means of improving performance and efficiency. In a parallel program, the processing is broken up into parts, each of which can be executed concurrently. The instructions from each part run simultaneously on different CPUs. These CPUs can exist on a single machine, or they can be CPUs in a set of computers connected via a network.

Not only are parallel programs faster, they can also be used to solve problems on large datasets using non-local resources. When you have a set of computers connected on a network, you have a vast pool of CPUs, and you often have the ability to read and write very large files (assuming a distributed file system is also in place).

The Basics

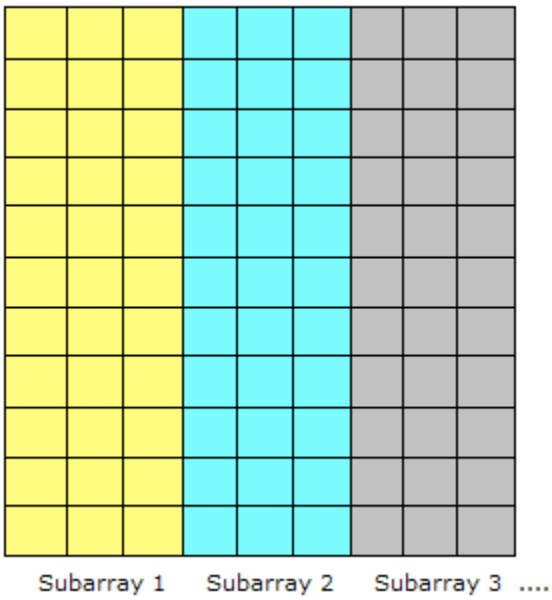
The first step in building a parallel program is identifying sets of tasks that can run concurrently and/or paritions of data that can be processed concurrently. Sometimes it's just not possible. Consider a Fibonacci function:

$$F_{k+2} = F_k + F_{k+1}$$

A function to compute this based on the form above, cannot be "parallelized" because each computed value is dependent on previously computed values.

A common situation is having a large amount of consistent data which must be processed. If the data can be decomposed into equal-size partitions, we can devise a

parallel solution. Consider a huge array which can be broken up into sub-arrays.



If the same processing is required for each array element, with no dependencies in the computations, and no communication required between tasks, we have an ideal parallel computing opportunity. Here is a common implementation technique called *master/worker*.

The MASTER:

- initializes the array and splits it up according to the number of available WORKERS
- sends each WORKER its subarray
- receives the results from each WORKER

The WORKER:

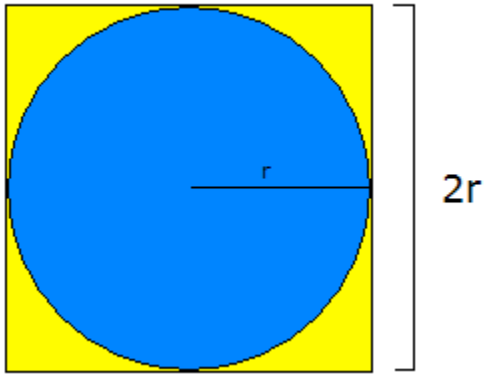
- receives the subarray from the MASTER
- performs processing on the subarray
- returns results to MASTER

This model implements *static load balancing* which is commonly used if all tasks are performing the same amount of work on identical machines. In general, *load balancing* refers to techniques which try to spread tasks among the processors in a parallel system to avoid some processors being idle while others have tasks queueing up for execution.

A static load balancer allocates processes to processors at run time while taking no account of current network load. Dynamic algorithms are more flexible, though more

computationally expensive, and give some consideration to the network load before allocating the new process to a processor.

As an example of the MASTER/WORKER technique, consider one of the methods for approximating pi. The first step is to inscribe a circle inside a square:



The area of the square, denoted As = $(2r)^2$ or $4r^2$. The area of the circle, denoted Ac, is pi * r^2. So:

```
pi = Ac / r^2
As = 4r^2
r^2 = As / 4
pi = 4 * Ac / As
```

The reason we are doing all these algebraic manipulation is we can parallelize this method in the following way.

1. Randomly generate points in the square
2. Count the number of generated points that are both in the circle and in the square
3. r = the number of points in the circle divided by the number of points in the square
4. PI = 4 * r

And here is how we parallelize it:

```
NUMPOINTS = 100000; // some large number - the bigger, the closer the
approximation

p = number of WORKERS;
numPerWorker = NUMPOINTS / p;
countCircle = 0;   // one of these for each WORKER

// each WORKER does the following:
for (i = 0; i < numPerWorker; i++) {
  generate 2 random numbers that lie inside the square;
  xcoord = first random number;
```

```
  ycoord = second random number;
  if (xcoord, ycoord) lies inside the circle
  countCircle++;
}

MASTER:
  receives from WORKERS their countCircle values
  computes PI from these values: PI = 4.0 * countCircle / NUMPOINTS;
```

What is MapReduce?

Now that we have seen some basic examples of parallel programming, we can look at the MapReduce programming model. This model derives from the `map` and `reduce` combinators from a functional language like Lisp.

In Lisp, a `map` takes as input a function and a sequence of values. It then applies the function to each value in the sequence. A `reduce` combines all the elements of a sequence using a binary operation. For example, it can use "+" to add up all the elements in the sequence.

MapReduce is inspired by these concepts. It developed within Google as a mechanism for processing large amounts of raw data, for example, crawled documents or web request logs. This data is so large, it must be distributed across thousands of machines in order to be processed in a reasonable time. This distribution implies parallel computing since the same computations are performed on each CPU, but with a different dataset. MapReduce is an abstraction that allows Google engineers to perform simple computations while hiding the details of parallelization, data distribution, load balancing and fault tolerance.

Map, written by a user of the MapReduce library, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key *I* and passes them to the reduce function.

The reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. [1]

Consider the problem of counting the number of occurrences of each word in a large collection of documents:

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
  EmitIntermediate(w, "1");

reduce(String key, Iterator values):
// key: a word
// values: a list of counts
```

```
int result = 0;
for each v in values:
  result += ParseInt(v);
Emit(AsString(result));     [1]
```

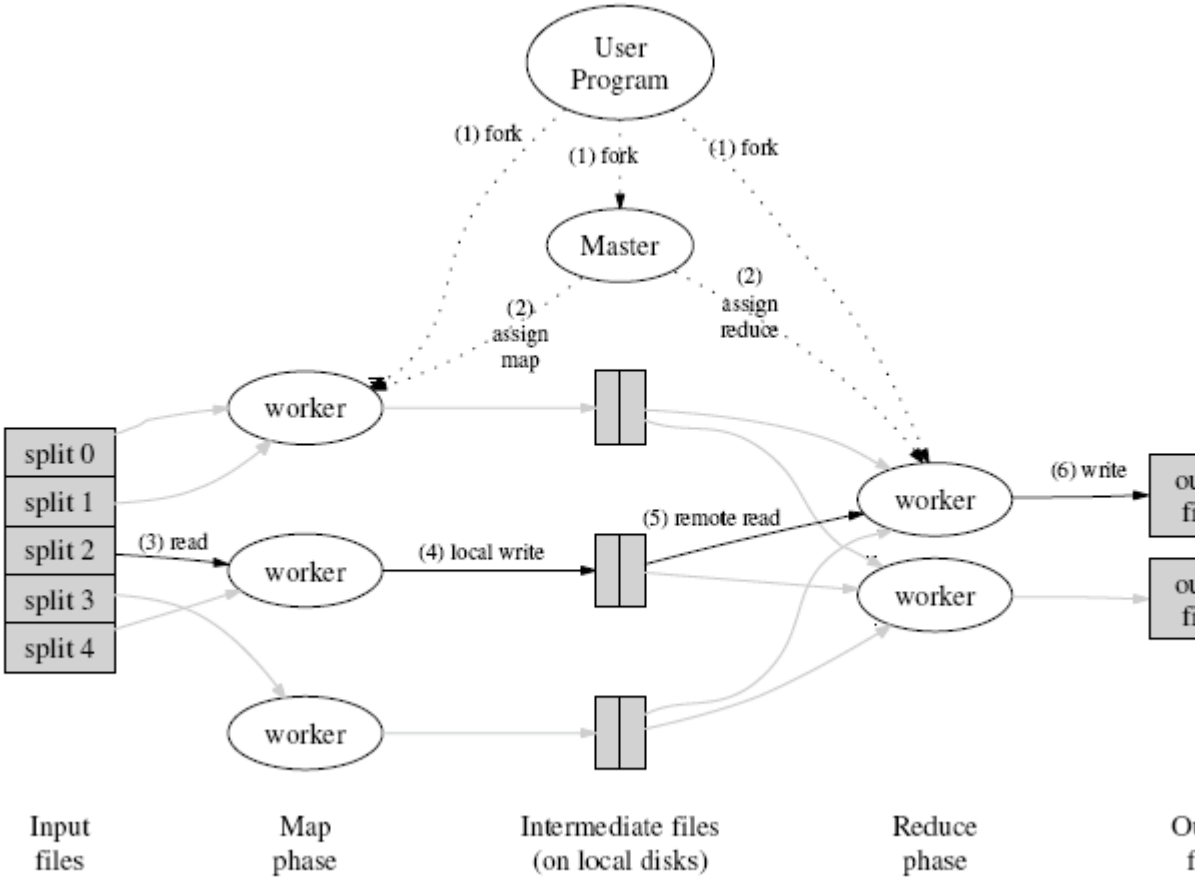
The map function emits each word plus an associated count of occurrences ("1" in this example). The reduce function sums together all the counts emitted for a particular word.

MapReduce Execution Overview

The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits or *shards*. The input shards can be processed in parallel on different machines.

Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., hash(key) mod R). The number of partitions (R) and the partitioning function are specifed by the user.

The illustration below shows the overall fow of a MapReduce operation. When the user program calls the MapReduce function, the following sequence of actions occurs (the numbered labels in the illustration correspond to the numbers in the list below).



1. The MapReduce library in the user program first shards the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece. It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is special: the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the corresponding input shard. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. If

the amount of intermediate data is too large to fit in memory, an external sort is used.
6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

After successful completion, the output of the MapReduce execution is available in the R output files. [1]

To detect failure, the master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial idle state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling.

Completed map tasks are re-executed when failure occurs because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global fille system.

MapReduce Examples

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computations.

Distributed Grep: The map function emits a line if it matches a given pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency: The map function processes logs of web page requests and outputs <URL, 1>. The reduce function adds together all values for the same URL and emits a <URL, total count> pair.

Reverse Web-Link Graph: The map function outputs <target, source> pairs for each link to a target URL found in a page named "source". The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: <target, list(source)>.

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of <word, frequency> pairs. The map function emits a <hostname, term vector> pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all

per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final <hostname, term vector> pair.

Inverted Index: The map function parses each document, and emits a sequence of <word, document ID> pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list(document ID)> pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions. [1]

References

[1] Dean, Jeff and Ghemawat, Sanjay. **MapReduce: Simplified Data Processing on Large Clusters** http://labs.google.com/papers/mapreduce-osdi04.pdf

[2] Lammal, Ralf. **Google's MapReduce Programming Model Revisited**. http://www.cs.vu.nl/~ralf/MapReduce/paper.pdf

[3] Open Source MapReduce: http://lucene.apache.org/hadoop/