UNIX TEXT UTILITIES HOWTOS FOR THE H5 CORPUS ANALYSIS GROUP

IRA WOODHEAD

29 JULY 2008

The Robot Army is a command-oriented general purpose distributed processing system, allowing transform and aggregation operations over any size data set. The corpus format consists of ordered sets of untyped, variable length records of any form. Command orientation, as contrasted with code orientation, provides significant benefits. The user can leverage skills by invoking largely the same commands as on a single machine. One need not even write new programs at all if they already exist and fit the mapreduce model. For example, many standard unix utilities can be used as is. In addition, writing new programs is made simple by the fact that the system is by nature language agnostic. There is no library API to learn, only STDIN, STDOUT, and STDERR. This allows the programmer to develop and test on a single machine with local sample data, then run on multiple machines with no customization necessary.

The Robot Army is written in Perl in a functional style for brevity. Small in lines of code (first release less than 1k loc), it is easy to understand and modify, and easy to install.

An introductory comment can go here...

Preliminaries

About STDIN, STDOUT, *streams*, *pipes* and *redirects*. Skip ahead if you know what these mean.

WHATEVER FLOWS through the STDOUT of

The RobotArmy: map and mapreduce operations on distributed corpora.

SYNOPSIS

```
On the commandline:
# tt map inputcorpus map-command outputcorpus
# tt mapreduce inputcorpus map-command reduce-command outputcorpus
```

DESCRIPTION

RobotArmy is a generalized distributed data processing system for use with a CT (Corpus Tool) corpus.

CT is part of RobotArmy. You should have already read the perldoc for **ct** before getting here.

You are also strongly encouraged to read the google paper (http://labs.google.com/papers/mapreduce.html) before proceeding further, as it will make everything much clearer.

There is at least one open implementation of mapreduce. The purpose of RobotArmy is to be

1. Language agnostic, using Perl as the infrastructure and leaving choice of language for mapreduce programs up to the user, and
2. as simple as possible while still being useful.

Some people have a particular thing for simplicity with good reason. The simpler your system is, the easier it is to make it do more without breaking.

This is a simplified, easy to use implementation. Based on shared ssh keys and remote calls, and using a lightweight RESTish state tracking mechanism, it is currently suitable for clusters of any size.

To use it, you just write two programs, a map program and a reduce program. The map program reads in a stream, defines input records, and creates a list of key/value pairs from it. The reduce program takes a transformed version of the map output consisiting of aggregated keys with their lists of values. It returns a (possibly different) key and another list (possibly a single value).

Strange as it may seem, an astonishingly large class of data processing tasks can be formulated as these pairs of programs.

HOW TO PROCESS ANY SIZE DATASET

Example: Wordcount

Consider the problem of counting all the occurrences of every unique word in a corpus of documents. We wish to print out records consisting of word/frequency pairs.

Here is a working program to do this:

```
## file wc.pl
my %words;
while(<>){
  $words{$_}++ for split /\W+/;
}
print "$_\t$words{$_}\n" for keys %words;
```

This can be used by saying:

```
# cat corpus.txt | wc.pl > result.txt
```

This program will work with small datasets. Larger ones will break it by eating up the memory. The way around this is to output

intermediate data, sort it, and then aggregate it. For this we need two separate programs and a sort in between.

First we modify the above to output intermediate data per input record. This is done by moving the structure to the scope of one record instead of the scope of the whole corpus:

(In this case records are lines of text)

```
## file: wc.m
while(<>){
  my %words;
  $words{$_}++ for split /\W+/;
  print "$_\t$words{$_}\n" for keys %words;
}
```

NOTE: Notice the simple difference between wc.pl and wc.m

Then we write a program to aggregate - or reduce - the resulting data once it has been sorted:

```
## file: wc.r
my ($key, $sum) = ('', 0);
while(<>){
  chomp;
  my($newkey, $val) = split /\t/;
  if($key eq $newkey){
    $sum += $val;
    next;
  }
  print "$key\t$sum\n";
  $key = $newkey;
  $sum = $val;
}
print "$key\t$sum\n" if $key;
```

This could be used directly by saying:

```
# cat corpus.txt | wc.m | sort | wc.r > result.txt
```

The RobotArmy package essentially does this, only across multiple machines. In fact, those very program listings of **wc.m** and **wc.r** work within the system already.

We can use the **tt mapreduce** program with a distributed corpus called **odyssey**, like so:

```
# tt mapreduce odyssey wc.m wc.r odyssey_wc
```

NOTE: Notice the similarity between this invocation and the last example.

There will appear a new distributed corpus called odyssey_wc, containing records of the form "word\tsum\n".

Using RobotArmy

You can use the **tt mapreduce** or **tt map** commands to launch map and mapreduce operations, or the **ct** program to execute any command on every host in the pool. Type "ct help" for a list of all the options.

After all the setup has been done, there should be sufficient information in the config file and command line for RobotArmy to know what to do. This info includes

the cluster of machines the input corpus is on (cluster)

the location of the corpus files on those machines (path)

the name of the input corpus

the name(s) of your program(s) comprising the operation

the name of the result corpus

Normally only the last three are specified on the command line, while the rest are in the config files. Hence the typical call looks like this:

```
# tt mapreduce odyssey wc.m wc.r odyssey_wc
```

If something goes wrong and you need to clean up, this will get rid of the distributed output corpus:

```
# ct delete odyssey_wc
```

The **cmd** option is quite useful in practice, but be careful, especially with **rm**! If your master machine is also a pool minion, you might delete every copy of something.

Some more useful built-in commands are:

ct list - gives a list of available corpora on the cluster.

ct reseat corpus localcorpus -cluster=master - places a corpus in a local dir on the master. Use only on a result corpus that can actually fit on the master.

Requirements

Unix

Ssh using shared keys.

WWW::Resource

RobotArmy vs Other Implementations

There are at least two other implementations of this processing model: Google's original, and a Lucene-related project called **Hadoop**. Google wrote theirs in C++, while the Lucene authors used Java.

Compared to the other, much more involved versions, this is more or less a toy.

A quite significant difference is the absense of a distributed file system. Google uses "GFS" to make input files available to map tasks, while this implementation simply stores distributed files across the cluster so that each host is responsible for the files on its local storage. This requires a preliminary "distribution step" which partitions a set of records across all hosts. This is quite simple since it can be implemented as a MapReduce operation! See 'partition.m' and 'remove_keys.r' in the bin directory for examples. This turns out to be more convenient than it looks, because often you will want to perform another mapreduce operation on the results of the last, and in this way the files are already distributed.

I haven't looked at Hadoop very much. Only to note that it appears much more "enterprise", which is to say it is more complicated. It has four xml configuration files (RobotArmy has only one, on one machine), and a lot ... a **lot** ... of code. It's no doubt much more stable, and also harder to get started. Not to mention there are no Perl bindings.

Just a bit more comment on relative complexity. The current Hadoop build has 313 java files with 61,810 lines of code. This doesn't include non-java files. There are 14 jsp files, 13 sh files, and 586 html files, plus. You could not check out the source code and understand everything in half a day. I'm not disparaging the Hadoop effort at all. It's probably very high quality. However the basic concept is very simple, and it might be possible to have a simple yet stable implementation.

Even though this version is probably not as stable as the others, it comprises just over 800 lines of code in three executable files. It has one configuration file, can be used almost immediately with very minimal effort, and is rather noninvasive, requiring nothing but distributing shared ssh keys on pool machines.

In short, perfect for exploring, understanding, and profitably using the MapReduce concepts with near-trivial time investment.

It's my goal to have a very sound system with comparable performance and stability to Google or Hadoop, using so little code that it can be read and understood in less than a day by a normal programmer.

Load Balancing and Fault Tolerance

Google and Hadoop deal with these issues separately, using a distributed file system for load balancing and centralized tracking of tasks for fault tolerance. I don't particularly like those because they're complicated and would require a lot more code than I feel is really necessary. I'm trying the idea of "replicated responsibility" to handle both load balancing and fault tolerance, and so far it works rather well and takes little code.

In RobotArmy, the input data are distributed evenly across hosts with a tunable **replication factor** which garuantees that F-1 hosts can go down at any time and the operation will still complete. The number of hosts that can **actually** go down is larger, and is related to the total amount of input data, but I have yet to do an analysis of what that is. The concept of **shared responsibility** provides load balancing, in that each machine is responsible for making sure all files residing locally are processed, and will step up to do the processing of nonlocal files themselves if given the opportunity (ie if finishing previous files faster than its brothers). In practice this works well. The spread between the first machine finishing and the last machine finishing is bound by the amount of time it takes to process one corpus file (usually a matter of a couple-few minutes, depending on your task).

TO DO

I would like to improve RobotArmy as much as possible while still conforming to the goals of simplicity, stability and performance. If you have suggestions for elegant improvements, get in touch with me.

SEE ALSO

http://labs.google.com/papers/mapreduce.html
Linux Server Hacks, Rob Flickenger

AUTHOR

Ira Woodhead, <ira at sweetpota dot to>

References