

Part 1:

I scripted this with a series of four main shell scripts run on my development system (PartA.sh, PartB.sh, PartC.sh, and PartD.sh), a few scripts which were provisioned into the vms and run by the main scripts over ssy, and a few ruby scripts for handling some of the more complex string manipulation/choice functions (such as generating the fpo file, nodefile, etc.).

Part 2:

Preprocessing

I attempted to put as much work as reasonably possible into initializing and building optimized data structures, to keep the mumbler fast and simple at execution time. See the figure below on the next page. I did not use an established map-reduce framework, although aspects of what I did were fundamentally similar to mapping and reducing.

First, I pulled over the google zip files in parallel and unzipped them. This took approximately 1 minute per file, so took about 35 min. The .csv's are all in the GPFS name space, but have data distributed according to the vm which did the copy/unzip.

The files are already at least partially sorted, so I next do mapping (to eliminate the extra fields, just leaving the bigrams and the count), and do a combining of adjacent line counts of the same bigram. This greatly reduces the filesizes down significantly. I also remove all lines which don't meet my criteria for words- they must consist only of letters and apostrophe. (And the apostrophe may not be a leading apostrophe.) Note this definition of a valid word is fairly restrictive. The process could generalize. (In fact, for the "initial letter hash" files later, I do include a "capital X pseudo-letter which could capture these non-alpha characters; but that would be an extension of the current project.)

Code comment- the map/combine was taking about 1.5 minute, so this part I re-implemented in c, which got it down to about 10 seconds per file. (But these are UTF-8 files; so my first attempt at parsing as characters using strtok was fast but wrong! I had to re-implement it with sscanf to handle UTF-8.)

Next, on each vm, each of the vm-local .csv files are split by first character into 27 files- one for each lower case letter, and one "catch-all" for every other initial character. The ultimate goal is to have a fast look-up of candidate words, with the first lookup being a simple identity hash on the first character (i.e. from the first character determine which file to open, then within the file do a search (linear/grep/whatever) for the whole word. In the final version I didn't grep through the files at run time, but I still kept this "alphabetic" deal out as a first stage hashing.

Next a shuffle sort is performed. This operation does have significant inter-vm network traffic, but it is only done one time as part of pre-processing. At the end of it, files for about 1/3 of the letters are put local to each vm. This is where most of the "inter-vm" internet traffic occurs. At the end of the paper I show screen shots of nmon while the system is idle, while shuffle sort is running, and while another of the pre-processing steps (which has minimal inter-vm traffic) is running.

Finally, each of the letter files are (and again, in parallel on all the machines), reduced into records for key value stores. The key is the first word of the bigrams. The Value is a list of two items. The first part of the value is the total count of bigrams seen, the second is a list of candidate “following words,” paired with their respective counts. There is NO precalculation of probabilities as floating points for comparison later, neither do the candidate “following words” need to be sorted. At runtime, the mumbler will just generate a random integer between zero and the total bigram count, and “walk the list” until a the (randomly created) desired cumulative total is reached. This is the generated “Following word”.

Record format: LEAD_WORD => [total, [TAIL1, count], [TAIL2, count]...]. These are all Marshaled into strings and saved using Ruby’s wrapper around the DBM key value system. There’s one further optimization I haven’t done- The “count” fields in the record format could be changed to cumulative count fields to give a small increase in speed in the loop walking the list (i.e. n integer additions where n is the length of the list), but given the nature of the problem this is probably a negligible improvement.

Runtime

Runtime of mumbler is simple. An outer “mumbler” program (written in ruby; only about 9 lines) repeatedly calls a “pick_next_word.rb” program (also ruby, about 24 lines).

There’s fairly low inter-vm network traffic when this runs. About 1/3 of the time, a word will be in a dbm key/value store which is saved locally; 2/3 of the time the word will require access through another vm, but only for a single word lookup in a dbm database. There are no linear scans taking place across the intranet, just lookups of individual word records.

It doesn’t escape my attention that inter-vm traffic could be further reduced. The solution presented does let a locally running instance of dbm access remote blocks in other vms. To further reduce the network traffic, one could put a small REST server local to each VM, then mumbler, instead of accessing the key value store directly on the other vms (and doing whatever block transfers dbm chooses to do), could make REST calls to the servers on the other vms. Those servers would then access the (local to them) instances of dbm, and return just the “next word” (not the entire block set that dbm would fetch.) I haven’t implemented this final optimization, as it seems it may be beyond the intended scope of the exercise.

Preprocessing prior to running mumblor: 3 vms

Google

Curl over and unzip
(pullover.rb script)



Map Lines (remove
year), do "best effort"
combine, as inputs are
at least partially sorted
(also in pullover.rb script.
Uses map_combiner.c for
performance critical part)



Alphabetic Deal out lines
from (now sorted) N files
on each VM into (still sorted)
files) separate files for each
initial letter (X for digits, etc)
(alphabeticDealOut.rb script)

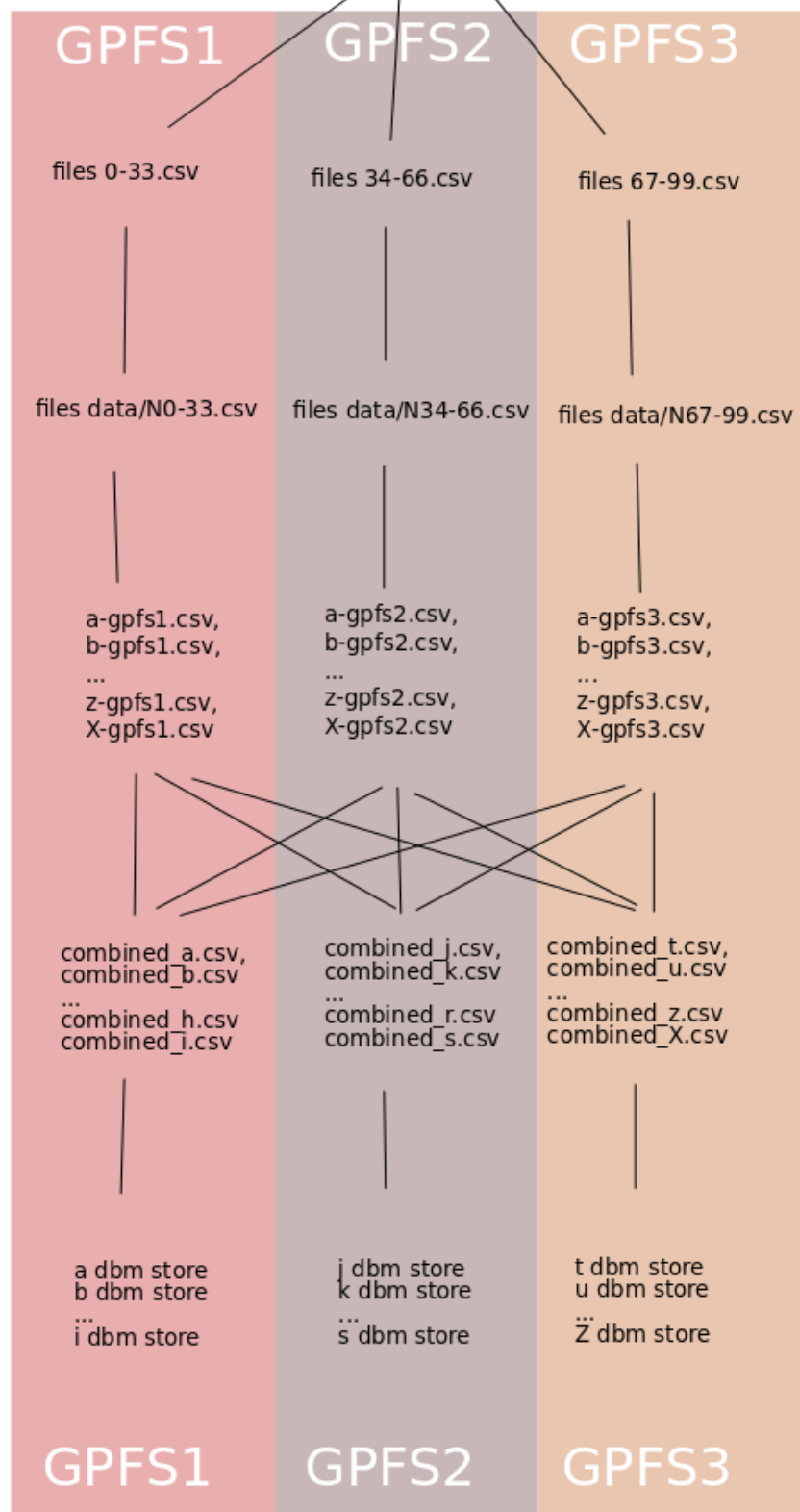


Shuffle sort: merge a subset of
letters into files on each vm.
This step involves significant
internal network traffic
(shuffleSort.rb script)

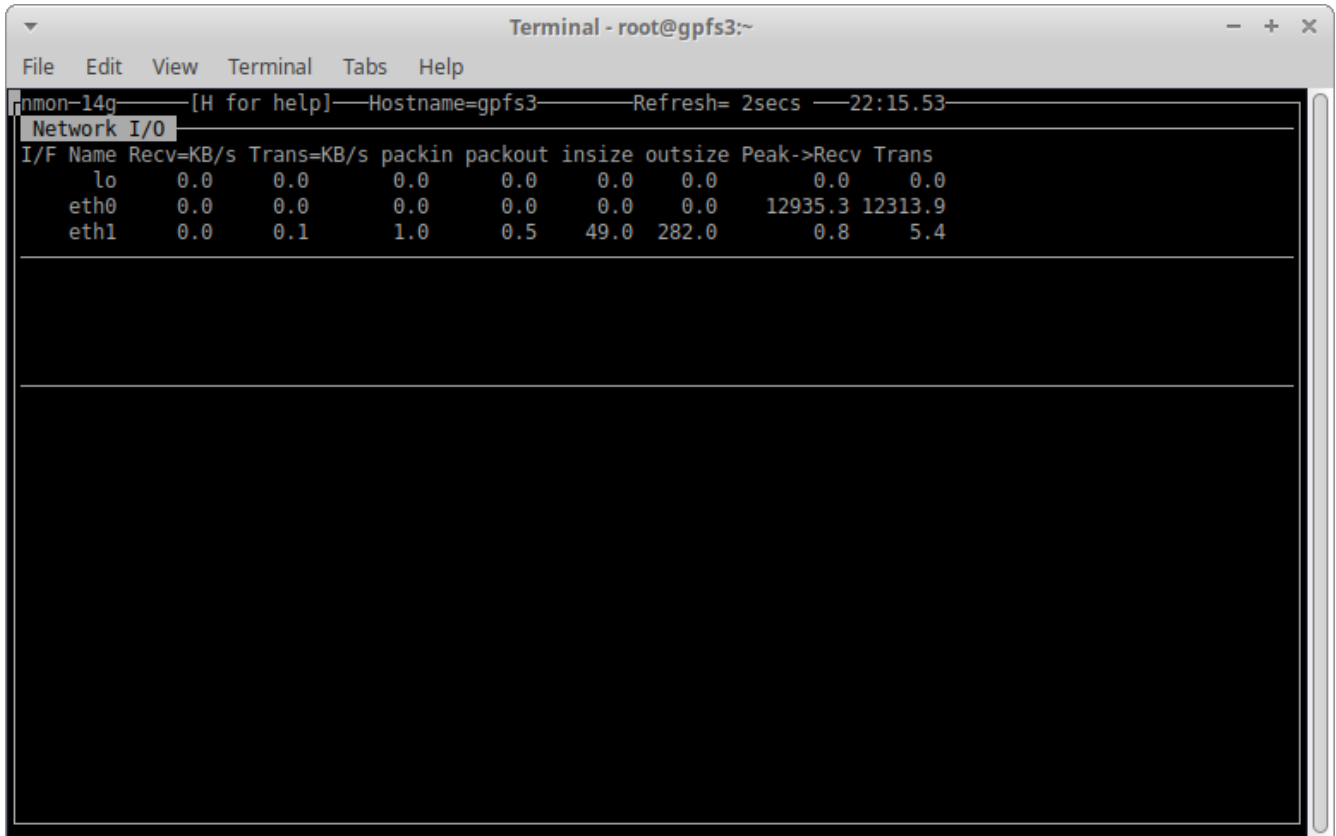


Reduce to final form of
dbm key-value stores,
distributed over VMs.
Keyed by bigram leadword,
gives distribution of tailwords
(reduceToKeyValueStore.rb
script)

This is the end of pre-processing.
Have 27 dbm key/value stores
distributed across our VMs
(could have used more VMs if
desired). All dbm file are in
the same gpfs file system, but
blocks are localized as shown



NMON with system idle



Terminal - root@gpfs3:~

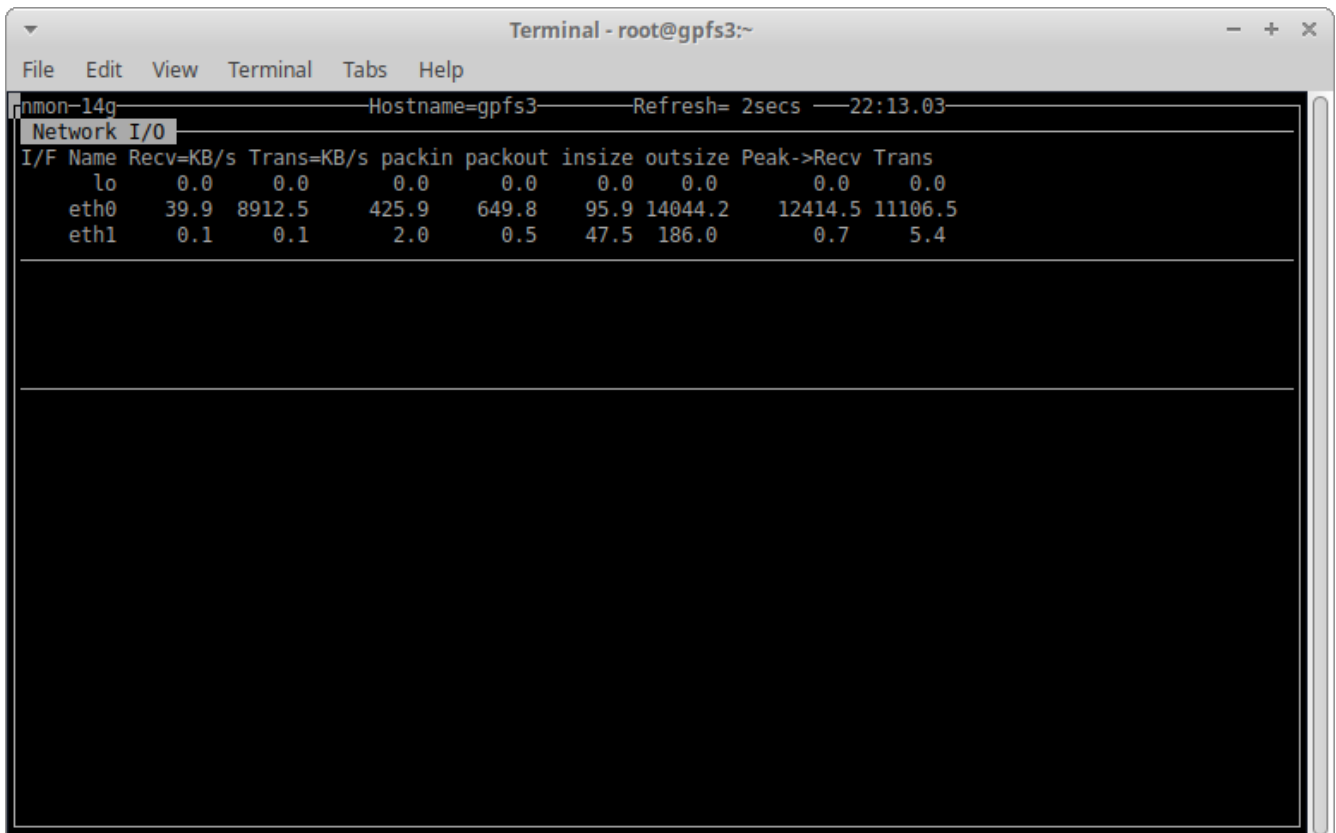
File Edit View Terminal Tabs Help

nmon-14g [H for help] Hostname=gpfs3 Refresh= 2secs 22:15.53

Network I/O

I/F	Name	Recv=KB/s	Trans=KB/s	packin	packout	insize	outsize	Peak->Recv	Trans
lo		0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
eth0		0.0	0.0	0.0	0.0	0.0	0.0	12935.3	12313.9
eth1		0.0	0.1	1.0	0.5	49.0	282.0	0.8	5.4

NMON preprocessing running most inter-vm traffic intensive operation (“shufflesort” operation)



Terminal - root@gpfs3:~

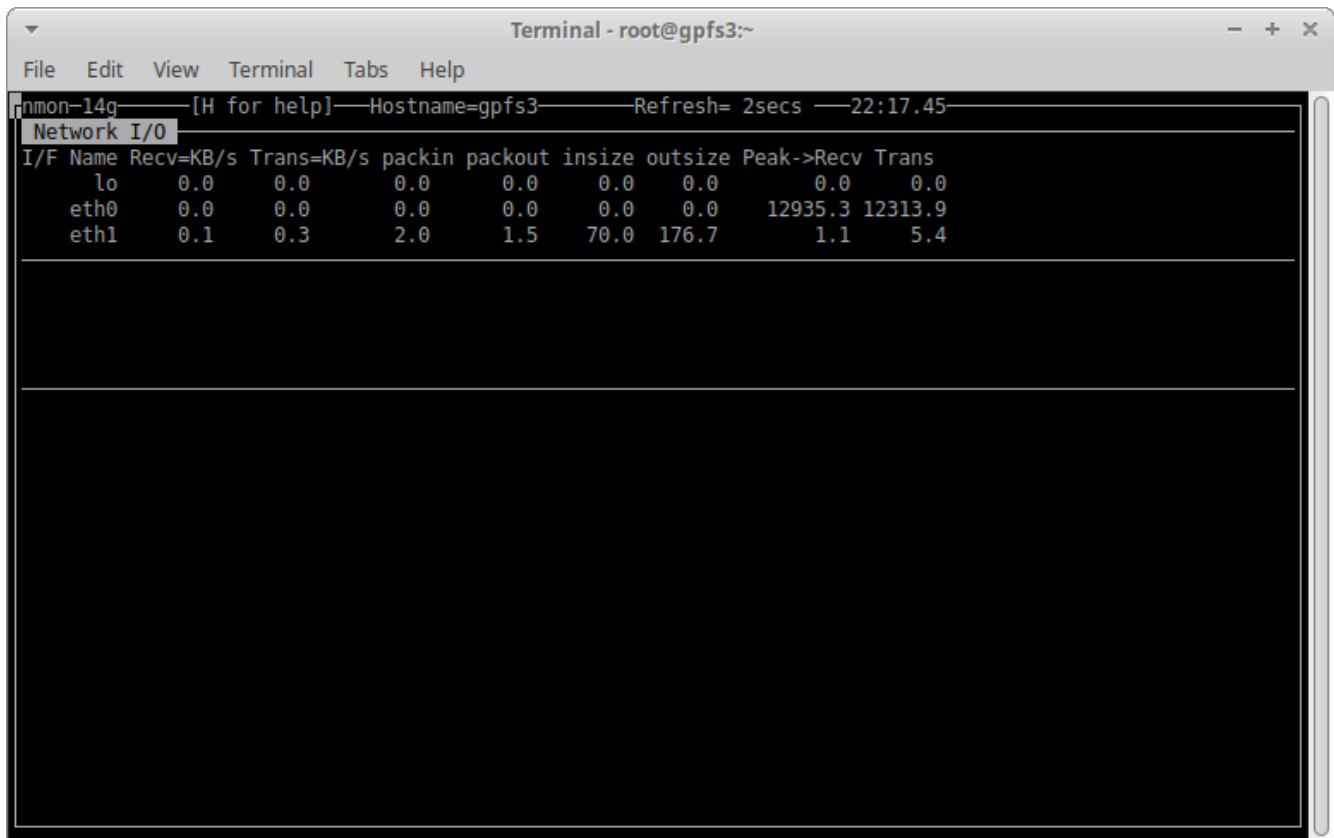
File Edit View Terminal Tabs Help

nmon-14g Hostname=gpfs3 Refresh= 2secs 22:13.03

Network I/O

I/F	Name	Recv=KB/s	Trans=KB/s	packin	packout	insize	outsize	Peak->Recv	Trans
lo		0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
eth0		39.9	8912.5	425.9	649.8	95.9	14044.2	12414.5	11106.5
eth1		0.1	0.1	2.0	0.5	47.5	186.0	0.7	5.4

NMON with preprocessing running non-network-intensive task (“reduceToKeyValueStore”)



A terminal window titled "Terminal - root@gpfs3:~" displays the output of the nmon-14g command. The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The terminal output shows the nmon-14g header with "[H for help]", "Hostname=gpfs3", "Refresh= 2secs", and "22:17.45". Below the header, the "Network I/O" section is highlighted. A table follows, showing network statistics for interfaces lo, eth0, and eth1. The table has columns for I/F Name, Recv=KB/s, Trans=KB/s, packin, packout, insize, outsize, Peak->Recv, and Trans. The data for eth0 shows high receive and transmit rates, while lo and eth1 show much lower activity.

I/F Name	Recv=KB/s	Trans=KB/s	packin	packout	insize	outsize	Peak->Recv	Trans
lo	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
eth0	0.0	0.0	0.0	0.0	0.0	0.0	12935.3	12313.9
eth1	0.1	0.3	2.0	1.5	70.0	176.7	1.1	5.4