sshfs usage on the sun grid engine

Hin-Tak Leung

April 27, 2007

1 Introduction

This document is essentially my notes on how to process some data stored on a portable external hard disk plugged into the USB port of my desktop with the CIMR sun grid cluster (on a separate network segment from my desktop), without doing any explicit or manual copying of the data to the cluster nor the manual copying of the data back to the external hard disk.

This mode of operation is particularly suitable when one of these factors applies:

- Where the files required for the calculation are many, and also of different type and serves many different purposes — sshfs (`http://fuse.sourceforge.net/sshfs.html`) can access a whole directory hierachy in the remote system e.g. it is entirely possible (but not recommended!) to mount the root directory of the remote file server!
- when the operation required is essentially one-off and there is no requirement of accessing the same piece of data in a later time or a later job.
- where the data is huge and the storage on the grid is limited.

I am sure there are other reasons. The drawbacks are a little effort of setting it up, and the jobs are not particularly fast. However, if you think about it, you are essentically having an sftp session running on-demand, so part of your CPU (and hopefully, a different CPU on multi-CPU machines) is used for encryption; and any comparision of timing with jobs using data on local discs must include the overhead of transferring the data across, (and especially an overhead involving human time, not machine time) and also moving the files into the right places in the destination relative to the directory from which some program is run.

So the German data, which consists of about 316GB of Affymetrix CEL files, to be digested by a relatively simple program which doesn't take long to run but requires a lot of memory, down to about 22GB, and write back to the same portable 500GB hard disc for storage, is a fair candidate, provided the network connectivity is fast enough.

2 Co-operation needed from the SGE admins

On every execution host, there is a one-off set up of:

- Installing the `fuse` package and the `sshfs` package. On Fedora Core 4 onwards, they are available as part of Fedora Extra and `yum install fuse fuse-sshfs` should just do it.
- If using the redhat packages, the users who is going to run sshfs needs to be added to part of the `fuse` group (in `/etc/group`). All that means, is that the user needs to be able to run `fusermount` with root privilege — this, on redhat systems, is achieved by (in addition to having the setuid bit set) disabling the world-executable bit, and making the binary's group `fuse`, so that only users to the `fuse` group can execute it. For installing from source, `fusermount` seems to be world-executable and also have the setuid bit set, which can be viewed as a security-related risk for possible escalation of privileges.

Also, it is necessary to make sure the `fuse` kernel module is loaded[1]. Normally the loading of the kernel module `fuse` should be automatic — I am not sure whether it is loaded on demand or on

[1] This requires a fairly recent linux kernel — the official inclusion of the `fuse` kernel module was 2.6.14 I think, but redhat kernels started to include it in as early as 2.6.12

start up, but I have never needed to manually load it much. It may occasionally require a per-reboot `modprobe fuse` manually.

3 Encryption key setup of the user

Note: It is extremely important to keep the private part of the ssh encryption key safe, and away from eyes of anybody who you don't trust! Firstly, it is a good idea to backup your `$HOME/.ssh` and also **keep the backup somewhere safe**.

The details outlined here is extremely important from the security point of view, and one must understanding the why's and the implication of each step.

I do `ssh-keygen -t dsa -b 2048` to generate a new pair of keys, and

- Choose any place else to store the generated key pairs, **any place else but the default, which is** `${HOME}/.ssh`.
- Just press return when asked for a passphase, so that the key pairs are **password-less**.

Now, this particular key pair (`id_dsa`, `id_dsa.pub`) can authenticate each other without a password. I would recommend for general ssh usage, one configures the default keys to have a passphase, but only do manual-override with the grid job to use the different and passwordless private key; and also only enable the password-less public key only during the duration of the grid jobs.

4 Enabling password-less sshfs

Copy `id_dsa.pub` from the grid machines and **append** it to `${HOME}/.ssh/authorized_keys` on the file server.

- The public key is not usually a secret and can be safely e-mailed, printed, etc — all its function is to let somebody who has the private key to get in; but the fact that its corresponding private key is password-less is. So be very careful.
- I would back up `${HOME}/.ssh/authorized_keys` on the file server and only **append** the additional public key for the duration of necessity; and removing the public key entry afterwards from the file server; and possibly also the private key from the grid machines as well. That's why **back up** `${HOME}/.ssh/` is important.

5 An example job script

This is the full actual content of one of my job script in its final form:

```
#!/bin/sh
#$ -S /bin/sh

echo ‘uname -a‘
export USEDIR=${HOME}/‘uname -n‘-$$
echo $USEDIR

mkdir $USEDIR

echo "before"
df
sshfs -C -o IdentityFile=~/.ssh.alt1/id_dsa hin-tak@192.168.55.29:german $USEDIR \
-o workaround=all -o reconnect -o sshfs_sync

#################

cd ${USEDIR}/norm/${1} && uname -a > which2w.log
cd ${USEDIR}/norm/${1} && /usr/bin/time -v -o time2w.log  ~/bin/gtype_cel_to_pq \
```

```
-refintensity ../../STYintfile.german \
-cdf ~/WTCCC-axil/Mapping250K_Sty.cdf \
-split ~/WTCCC-axil/Mapping250K_Sty_annot.split \
 -subset ~/WTCCC-axil/Mapping250K_Sty_annotAB.padded \
-log-average *.CEL* >& q2w.log


#################

cd ${HOME}
sync
sleep 10

fusermount -u $USEDIR && echo "umount successful"
echo "after"
df
```

6 Discussion

There is one shocking thing I found during this work about sshfs and the fuse framework in general — because as much of `fuse` is meant to as non-privileged as possible (unlike NFS and others which requires root priviledge at one or both end points to set up per-connectivity), which means the system will not allow `fusermount -u` to block indefinitely or appear to hang.

In more accessible terms, it means, if the unmounting process takes too long to finish, it would get forceably killed. i.e. Any cached-write which hasn't been finished would be lost. We are talking about **data lost at a massive scale here**. e.g. In a typical job, at the very end of the job, the program writes 23 files (one per chromosome) totalled to about 2GB. In 2 batches of "unsuccessful" runs, sometimes I get file 1, sometimes up to file 4, and sometimes up to file 10; so I know the bulk of the calculation has finished, but the result has not been written back completely.

So here are a few note-worthy points about the example job script:

- sometimes it is useful to know on which machine a job ran — particularly when/if it fails. It is possible to look it up retrospectively with `qacct` (and manually from examining the SGE logs), but having it in the job output is easier.
- I create a per-job mount point with `${HOME}/`uname -n`-$$` using the machine name and process id as part of the directory name; this make the directory unique, if more than one jobs are run on the same machine. I probably should add an `rmdir` at the very end of the script to tidy up itself. (`rmdir` is probably safe as it won't do any bad thing if the directory fails to unmount; `rm -rf` is obviously not a good idea!)
- the `df` commands are for showing what's mounted and what's not.
- The `cd; sync; sleep;` was my initial (and not effective) attempt at the data lost problem. The current shell process (the one that's running the script itself) needs to get outside of the mount point to allow the directory to be unmounted.
- The `-C -o workaround=all -o reconnect -o sshfs_sync` options were the final change that solved (or worked around?) the data lost problem. I suppose enabling compression and the synchronous write option are the two important ones among these four, but I am happy to leave the other two in.

The 316GB of data is broken into 12 jobs. So a typical job reads about 26GB (316GB/12), and does something fairly simple, and write 2GB back. The large memory requirement comes from a table transposition — i.e. writing result derived from entry 1 of file 1, entry 1 of file 2, entry 1 of file 3, etc.

According to the result of `time`, without compression/synchronous-writes the typical job takes 8-9 hours, whereas with compression/synchronous-writes, it takes 4-5 hours. CPU utilization is only 20% in the former case and 30% in the latter. In the end 5 jobs completed without compression/synchronous-writes in two batches, i.e. 2 out of 12, 3 out of 10, then 7 out of 7 with compression/synchronous-writes.

According to the time stamps of the 23 output files[2], asynchronous-writes (without compression) of 2GB takes about 15 minutes, whereas synchronous-write (with compression) of 2GB takes about 50 minutes.

All this (the timing with and without compression for reading, the time stamps of synchronous-write) seems to imply a bottleneck of about 5-7 Mbits/s somewhere.

One issue of note is that the ntfs driver (the disc on which the data is located) is very slow on linux. The ntfs driver can takes 30-50% of the CPU for continuous reading, and it is my experience that it takes about 40% CPU time on my desktop.

7 A dirty trick

I believe `qrsh` or some such provides a similiar functionality, but this is a dirty-trick job script, which I called `script_qsub_slave`, just runs a command in a particular directory:

```
#!/bin/sh
#$ -S /bin/sh
cd $1 && uname -a && $2
df
```

It is quite useful if one needs to run a small command (like doing `make`, or checking the mount table on a specific machine with:

```
$ qsub -q all.q@galapagos1 script_qsub_slave ‘pwd‘ df

$ qsub -q all.q@stats8 script_qsub_slave /home/hin-tak/my-C-code-src make
```

[2]They are in chromosome order and written one after the other and the file sizes are known. Although it is not possible to know the start time of the first file writing, it is possible to guess.