Using Google Compute Engine

Chris Paciorek

January 30, 2014

WARNING: This document is now out-of-date (January 2014) as Google has updated various aspects of Google Compute Engine. But it may still be helpful as a general guide.

This document provides a tutorial on using Google Compute Engine (GCE) to do cloud computing, with a focus on statistical users. The syntax here is designed for SCF users, but should provide guidance for non-SCF users as well.

The basic idea of GCE (and other cloud computing resources such as Amazon's EC2) is to allow you to start up one or more virtual computers that run on Google's servers. You can link together multiple virtual computers into your own computer cluster. Each virtual computer is called an 'instance' (in Google's terminology). I'll also refer to it as a 'node', mimicing the idea of having a cluster with multiple nodes. The advantage of all this is that you have access to as many computers as you want, but you only pay for them for as long as you use them.

The instructions below assume that you are logged in to an SCF Linux machine and have access to your SCF home directory.

1 Getting an account and setting up payment

In general you'll need a Google account.

1. If your work is affiliated with the SCREMS grant (SCREMS faculty PIs are Nolan, Huang, Kaufman, Purdom, Yu), get permission from the appropriate PI and then contact consult@stat.berkeley.edu (cc'ing the PI). You will be added to the *scf-gvm0* project and will use that as the project ID in the instructions below.

2. For most users, you'll need to establish an account with Google that you either pay for directly yourself or link with a grant. See https://developers.google.com/compute/docs/signup. You may want to use your campus Google account (i.e., bMail with email address <username>@berkeley.edu) account for this rather than a personal Google account. To bill

to a grant, please contact consult@stat.berkeley.edu and Jane Muirhead (statgrants@stat.berkeley.edu), and we'll work with you to set up the billing so that Google bills the university. This will establish a new project ID that you will use.

You can view information about your project, including billing, through the Google APIs console: https://code.google.com/apis/console/

IMPORTANT: You will be billed for the entire time an instance is running, regardless of whether you are running anything on it. So make sure to terminate the instance when you are done. See Section 5 for details.

2 Starting a compute instance in the cloud

All management of your cloud computing can be done via shell commands using the *gcutil* tools. These are installed on SCF Linux machines, so the basic mode of operation in what follows in this document is to logon to an SCF Linux machine and execute the commands at a bash shell command line. *gcutil* is a set of commands that allow one to start and stop nodes, interact with the nodes, and generally manage your Google Compute Engine resources. You can see some documentation at https://developers.google.com/compute/docs/gcutil/. For basic command-line help that lists the various *gcutil* commands, do

```
gcutil help
```

For help on a specific command, do

```
gcutil help <command>
```

2.1 Getting set up

If you are authorized (see above) to use the SCREMS grant, in the instructions below, *<projectID>* will be *scf-gvm0*. Otherwise use whatever project ID you got when signing up with Google.

When you first use *gcutil* it will require you to authenticate. Do the following:

```
gcutil auth --project=<projectID>
```

Then follow the instructions presented. Google will create a public-private key pair to allow you to do password-less ssh. We recommend that you do enter a passphrase (e.g., it might just be your SCF password). The private key will be deposited in *~/.ssh/google_compute_engine* by default and the public key will be deposited in all the Google machine instances that you create so you can ssh to them without a password.

To avoid having to enter your passphrase every time, you can run the following and just enter the passphrase once in a given session.

```
ssh-add ~/.ssh/google_compute_engine
```

In general, whenever you issue a *gcutil* command, you need to enter the project ID. To avoid this, do the following:

```
gcutil getproject --project=<projectID> --cache_flag_values=True
```

Also, I'm not sure how to do it from the command line, but you can also enter your zone in the configuration file that is created in the step just above. Just add

```
--zone=us-central2-a
```

to the *.gcutil.flags* file in your home directory.

2.2 Starting an instance and installing software

Note that a basic setup that you can run in a single script is available at *setupGoogleCluster.sh*. You'll need to set the number of nodes, number of cores per node and a few other variables at the beginning of the script file (and modify the software and R packages installed, if desired).

2.2.1 Starting an instance

In the future, we will have an SCF image that will mimic the environment and most or all software that exists on the SCF Linux servers. At the moment, this image does not exist. Here are the steps for using a default Ubuntu image that relies on the same version of Linux as running on the SCF servers and for adding software to that image. For the moment, these instructions do not cover Matlab or hadoop.

We'll set up an environment variable containing the imageID:

```
imageID=projects/google/global/images/gcel-12-04-v20121106
```

Note that that image may be out of date when you read this. You can look for standard Google images using

```
gcutil listimages --project=google
```

For a basic Ubuntu image, choose the one with the most recent version date, i.e. gcel-12-04-vYYYYMMDD and set *imageID* to be this string.

Next we'll set up a variable containing the names of the nodes. This isn't necessary but will help in automating some of the steps below. Here we'll name the nodes vm0, vm1, ... ('vm' standing for 'virtual machine'), but you can name them whatever you want.

```
numNodes=4 # choose the number of nodes you want here
nodes=$(eval echo vm{0..$(($numNodes-1))})
# this should just confirm that you have a shell variable
# containing node names:  vm0 vm1 vm2 ...:
echo $nodes
```

Next you need to decide what kind of hardware you want for your virtual nodes. You can figure out what machine types are available by doing:

```
gcutil listmachinetypes
```

Decide upon one depending on how much memory, CPU, cores, and disk space you want. You'll enter the name of the machine_type below after the *–machine_type* flag. Here I've chose a standard 8-core machine:

```
gcutil addinstance --image=$imageID --machine_type=n1-standard-8
--zone=us-central2-a --wait_until_running $nodes
```

Ok, your virtual nodes should now be running. You can ssh to a given node (vm0 in this case) as:

```
gcutil ssh vm0
```

On the virtual node, you'll have the same username as the system you are coming from. The machine is running a variant of Ubuntu Linux, which makes thing easy because we run Ubuntu on the SCF Linux machines.

2.2.2 Putting software on your instance

If you are using an image provided by SCF or an image you've already created, then you can skip this step provided the image has all the software you need already.

If not, the next step is to put the software you want on your instances. In general, we'll use the *apt-get* mechanism of Ubuntu to install pre-existing Ubuntu packages, but you can also build and install software from scratch on the instances. If you need help with this, contact consult@stat.berkeley.edu.

Next we install R, Octave (an open source version of Matlab), Java, a threaded BLAS for linear algebra, and openMPI for distributed computing. You can manually ssh to each node and run the following commands. However, to do this automatically on multiple nodes, insert the following two lines in a bash script file, which I'll call *install.sh* in the syntax below.

```
sudo apt-get update
sudo apt-get install -y libopenblas-base openmpi-bin libopenmpi-dev
r-base octave3.2 openjdk-7-jre openjdk-7-jdk
echo "DONE with Ubuntu Linux package installation on $(hostname
-s)."
```

Now run the script file on each node using the following commands:

```
for node in $nodes; do
   gcutil push $node install.sh .
   gcutil ssh $node "sudo /bin/bash ./install.sh >& install.log.$node" &
```

```
done
```

You'll need to wait a few minutes for this to finish - to check on it, you can do the following every so often until all the nodes report as being DONE.

```
for node in $nodes; do
   gcutil ssh $node "grep DONE install.log.$node"
done
```

If you need additional software to include in the list of packages indicated in *install.sh*, you can search amongst available Ubuntu packages (here, e.g., for octave packages) with

```
apt-cache search octave
```

And you can see if a specific package is installed and what version would be installed if you did install it:

```
apt-cache policy octave3.2
```

Next we'll install some standard R packages for parallel computing. You should add whatever additional R packages you need to the list of packages below, or you can of course omit this if you won't be using R. Place the following line in a file; we'll call it *installRpkgs.R,* and push to the nodes.

```
echo "install.packages(c('Rmpi', 'foreach', 'doMC', 'doParallel',
   'doMPI'), repos = 'http://cran.cnr.berkeley.edu');
    print(paste('DONE with R package installation on ',
    system('hostname -s', intern = TRUE), '.')" > installRpkgs.R
for node in $nodes; do
   gcutil push $node installRpkgs.R . &
done
```

Now we'll invoke R and run *installRpkgs.R* on each node to install the packages:

```
for node in $nodes; do
gcutil ssh $node "sudo R CMD BATCH --no-save installRpkgs.R
        install.Rpkgs.log.$node" &
done
```

You can monitor progress as you did for the Ubuntu package installation above.

2.3 Creating your own image

Rather than adding the software to Google's base image, you can create your own image by using Google's image, adding software and then saving the result. You need access to Google cloud storage to save the image.

If you're using *scf-gvm0*, you can save the image on the disk space associated with the project (see code.google.com/apis/console)

If you're not using *scf-gvm0*, you'll need to set up Google cloud storage (and billing) for your project.

Now ssh to your instance. and do the following within the instance.

1. Add the software you want to your running instance (e.g., vm0 above).

2. `sudo python /usr/share/imagebundle/image_bundle.py -r / -o /tmp/ --log_file=/tmp/image.log`

3. If you are NOT using *scf-gvm0*, do the following, choosing a name for your 'share' in the cloud storage and wait a few minutes for the account to be activated.

   ```
   gsutil config

   # follow the instructions for authentication

   gsutil mb gs://<nameOfShare>
   ```

4. Now add the image (created in step 2) to the share, where <*file*> is based on the file name of the .tar.gz file created in /tmp. <*nameOfShare*> is either the name you chose above or *scf-gs0* in the event you are using *scf-gvm0*. <*imageName*> is the name you choose for the image.

   ```
   gsutil cp /tmp/<file>.image.tar.gz

   gcutil addimage <imageName> gs://nameOfShare/<file>.image.tar.gz
   ```

You should now be set to use <*imageName*> as the argument to *–image* when using gcutil addinstance.

3 Getting files to and from GCE and disk usage

Note that the most basic machine types only have 10 Gb of disk space and this space disappears when the instance is shut down. Such disk is called *ephemeral* disk. Furthermore, files on one instance are not available to another instance.

3.1 Transferring files

You can copy files to and from an instance using *gcutil* (here I just copy to vm0, but you could wrap it in a loop to copy to all the nodes, as I've done above.

```
gcutil push vm0 path/to/file/on/SCF/filename path/to/file/on/node/.
gcutil pull vm0 path/to/file/on/node/filename path/to/file/on/SCF/.
```

3.2 Additional ephemeral disk space

If you want more than 10 Gb of disk space, but are fine with ephemeral space, you can choose one of the machine types with "-d" in the name of the machine type; just invoke

```
gcutil listmachinetypes
```

Now start your instances. You might start one instance that will be the master node with extra storage and the other nodes with just the basic storage, e.g., if your code writes output only from the master process while the slave nodes just do processing and send results directly from memory back to the master process.

Once you've started your instances, do the following (these instructions are taken from https://developers.google.com/compute/docs/disks#additional).

The instructions that follow assume that you only need extra space on your primary node, but if all your nodes have additional space, you could embed the instructions in a loop over nodes.

You'll need to choose the directory (by setting *mount* below) at which to mount your disk. This might be */data*, */scratch*, *~/data*, etc., where the latter would place it in your home directory.

```
mount=/data
gcutil ssh vm0 "sudo mkdir $mount"
gcutil ssh vm0 "ls -l /dev/disk/by-id/google-ephemeral-disk-*"
```

Check that the result of the previous command is */dev/disk/by-id/google-ephemeral-disk-0*, in which case that can be used in the next command:

```
gcutil ssh vm0 "sudo /usr/share/google/safe_format_and_mount
    -m 'mkfs.ext4 -F' /dev/disk/by-id/google-ephemeral-disk-0 $mount"
```

You can check that it is mounted and available for use and has the expected disk space with:

```
gcutil ssh vm0 "mount | grep 'sd.'"
gcutil ssh vm0 "df -h $mount"
```

3.3 Persistent disk space

By default, the disk space associated with your instance is removed when you delete the instance. This is fine if you're careful about getting your results back before deleting. However,

if you have results (or input data) that persist between instances, you'll need to create a persistent disk. https://developers.google.com/compute/docs/disks#persistentdisks has the details on how to do this.

Note that a persistent disk must be attached to an instance when the instance is created. You can attach a persistent disk to more than one instance, but if you do, it is attached read-only, so this is useful only for reading in data and not for writing output. For writing output, you would need to attach the disk just to the master node and only write results from this node.

Also, the persistent disk needs to be in the same zone as the instances that you launch.

4 Running jobs

Note that once you use *gcutil* to ssh to a node, you can ssh to another node simply by using ssh (not *gcutil ssh*) and the name of the node (vm0, vm1, etc. in our naming convention from before). One could also copy files between nodes simply using *scp*. This may save data transfer costs, though I haven't investigated this.

4.1 Single node (shared memory) jobs

For the most part, the steps you need to follow are described in the SCF tutorial on shared memory parallel programming, also available on Chris Paciorek's website.

Suppose that you have an R code file, *testShared.R*, you want to run (the same basic approach should work for Matlab code or a shell script). Also suppose that the R code produces the *testShared.RData* file as output. We can run the job remotely as follows:

```
gcutil push vm0 testShared.R . # copy R file to node
gcutil ssh vm0 "R CMD BATCH --no-save testShared.R testShared.Rout"

# copy results back:
gcutil pull vm0 testShared.Rout .
gcutil pull vm0 testShared.RData .
```

4.2 Multiple node (distributed memory) jobs

For the most part, the steps you need to follow are described in the SCF tutorial in the SCF tutorial on distributed memory parallel programming, also available on Chris Paciorek's website.

First, you'll need a file that lists the names of the nodes on which the distributed job will be processed. You can automate creation of a hostfile (following our creation of the *$nodes* variable above), which I call *.hosts* here:

```
   echo $nodes > .hosts; sed -i 's/ /\n/g' .hosts
```

To choose the number of processes to assign per node, you can do:

```
numProcsPerNode=2
for ((i=0; i <$numNodes; i++)); do
   echo vm$i slots=$numProcsPerNode >> .hosts
done
```

What follows is an example for R, but analogous commands should work for non-R jobs.

Suppose you have a file of R code, *testDistrib.R*, that implements a distributed job, such as by using the *Rmpi* package. The R file should specify the number of cores being used. E.g., if you have two nodes with two cores per node in your cloud cluster for 4 total cores, you would want to have `startMPIcluster(3)` in your R code file, leaving one core for the master R process. Obviously you would adjust this based on how many nodes you have and how many virtual cores are in your instance type. You may want to use fewer than the number of virtual cores on a machine to allow for threaded calculations to use multiple cores.

You'll need to push the R code file and the *.hosts* file to your master node.

```
   gcutil push vm0 testDistrib.R .
   gcutil push vm0 .hosts .
```

Now run the R job by invoking *mpirun* with the *.hosts* file so that Rmpi knows what nodes to distribute the work to:

```
   gcutil ssh vm0 "mpirun -hostfile .hosts -np 1 R CMD BATCH --no-save
testDistrib.R testDistrib.Rout"
```

Assuming that the R code in *testDistrib.R* writes output to *testDistrib.RData* (on the master only) and writes text to *testDistrib.Rlog,* we need to get these files back from the nodes. Note that if the directory containing *testDistrib.Rlog* is not shared amongst the nodes, then there will be one *testDistrib.Rlog* file on each node.

```
for node in $nodes; do
   gcutil pull $node testDistrib.Rlog ./testDistrib.Rlog.$node
done
gcutil pull vm0 testDistrib.RData .
gcutil pull vm0 testDistrib.Rout .
```

4.3 Threaded computations

Note that the installation above includes openBLAS, a threaded BLAS, as the default BLAS. Any code that uses the BLAS, including R linear algebra functionality, will use as many cores

as possible on a given node to do the calculations. In some cases (e.g., small matrix problems), threaded BLAS calculations might actually take longer than unthreaded, and there are cases of conflicts between threaded BLAS and other software. So in some cases you may want to fix the number of threads used by threaded programs that use the openMP protocol, which includes openBLAS. You can do this by setting the OMP_NUM_THREADS environment variable, e.g., `export OMP_NUM_THREADS=1`, as described in more detail, in the links in the above subsections.

5 Terminating instances

You will be billed for the entire time an instance is running, regardless of whether you are running any jobs on it. So make sure to terminate the instance when you are done. You can manually terminate instances as follows:

```
gcutil deleteinstance -f $nodes
```

If you want to do this automatically after your job is finished, you could write a shell script that submits your job via *gcutil ssh*, copies the output back and then deletes the instances. As an example with a single node (and using R in this example, though this should work more generally), put the following in a file, say, *testStop.sh* (and assuming you have a file of R code you want to run, *test.R*, that saves output in *test.RData*). The crux of the shell code below is to check if the output file, *test.RData*, exists on the SCF machine, as a test for whether the job has completed and written output files back to the SCF machine, in which case it should be safe to shut down the cluster.

```
# begin testStop.sh
$gceuser = $USER
gcutil push vm0 test.R . # copy R file to node
if [ -f test.RData ]; then # does test.RData exist already?
    # if so, move any existing file aside, since we'll test
    # for its existence to check that job completes ok
    # before deleting the instance
    mv test.RData test.RData-
fi
# run job in foreground, waiting until done
gcutil ssh vm0 "R CMD BATCH --no-save test.R test.out"
gcutil pull vm0 test.out . # copy results back
gcutil pull vm0 test.RData .
if [ -f test.RData ]; then # has key file been transferred back
```

```
   gcutil deleteinstance -f vm0 # if so, delete instance
   echo "test.RData successfully copied back from node(s);
         instances are being shut down."
   echo "test.RData successfully copied back from node(s);
      instances are being shut down." | mailx -s "job succeeded"
      $gceuser@stat.berkeley.edu # automatic email notification
else
    echo "test.RData not copied back from node(s);
        instances are NOT being shut down"
    echo "test.RData not copied back from node(s);
       instances are NOT being shut down." | mailx -s "job failed"
       $gceuser@stat.berkeley.edu # automatic email notification
fi
# end testStop.sh
```

Now do

```
chmod u+x testStop.sh
./testStop.sh >& job.out &
```

You can then monitor *job.out* to see the progress of your job. Note that *testStop.sh* runs in the background (note the & at the end of the line above) so you can log out of the SCF machine. But the contents of *testStop.sh* run "in the foreground" in the sense that they run sequentially, so the copying of results back should only occur after the computation has finished.