1. Running R and RStudio on Chapman Research Technology Support Clusters

Released: 2019-11-04  
Author: jakelly@chapman.edu

Version: .01

Introduction

This document is not platform-specific. Its instructions will work on Macintosh, Windows, Linux and mobile clients with appropriate software installed.   
Other RTS documents cover installing and configuring the software needed on each platform, and using that software to connect to RTS servers and clusters.

This document covers:

1. Running R and RStudio in **interactive** sessions on cluster **login** nodes (for testing/debugging/visualization).
2. Using the **SLURM** scheduler to run R workloads in **interactive** sessions on **compute** nodes (for testing/debugging/parallel testing).
3. Using the **SLURM** scheduler to run R workloads in **batch** jobs (for production).
4. Running R **parallel** workloads for CPU using the **SLURM** scheduler.
5. Running R **parallel** GPU workloads using the **SLURM** scheduler.

R and RStudio single-threaded workloads can be run on Chapman RTS login nodes, with resource restrictions, for testing and debugging.   
On the Keck cluster, that includes the login node, keckcluster.chapman.edu.

R parallel workloads are available on all Chapman RTS compute nodes.   
On the Keck cluster, that includes partitions **defq** and gpuq.

R parallel GPU-based workloads are available on GPU-equipped Chapman RTS compute nodes.   
On the Keck cluster, that includes partition gpuq.

Network Connectivity Notes:

On campus, you will need to be connected through a physical ethernet cable or EduRoam wireless. Depending on your location on campus, you may also need to be connected through the campus VPN server, vpn.chapman.edu.

Off campus, or on-campus but connected to ChapmanOpen wireless, you will always need to connected through the campus VPN server.

Running R on RTS Clusters

There are several ways to run R workloads on RTS clusters.

In “interactive” sessions, the R or RStudio client is **opened**, running on some cluster node. The user runs scripts, edits code, and manages files using that remote R client. This mode is very similar to using R on a local computer, but provides access to RTS cluster computing resources **with some limits**.

When submitting “batch jobs”, the R client is **not directly opened**. R scripts and cluster job-definition files are submitted to the cluster’s job-scheduling software for distributed execution. This is the production mode on RTS clusters, and provides **full access** to all RTS cluster computing resources.

*Interactive* sessions on cluster *login* nodes using the R (terminal) or RStudio (graphical) clients

* Designed for testing, validation, debugging, some visualization
* This is the only way to access the R Studio graphical interface on clusters
* Always restricted to single-core, single-threaded code on login nodes
* Limited memory allocation
* Limited wall-clock duration (interactive sessions are always limited to six hours of wall-clock duration), with automatic disconnection

*Interactive* sessions on cluster *compute or login* nodes using the R (terminal) client

* Designed for testing, validation, and debugging
* Can include parallel debugging and performance testing on compute node sessions
* Available on both login and compute nodes.
* Full access to available resources on compute nodes (allocated through scheduling engine)
* Always restricted on login nodes to single-core, single-thread code, limited memory allocation
* Limited wall-clock duration on all nodes (interactive sessions are always limited to six hours of wall-clock duration), with automatic disconnection

*Batch job* submissions through the cluster job-scheduling engine, SLURM

* Required method for production R workloads
* Unrestricted resource access (up to cluster/partition/account limits)
* Unrestricted wall-clock duration (up to cluster/partition/account limits)
* Scalable to many concurrent parallel jobs (up to cluster/partition/account limits)

Running interactive sessions on login nodes using the RStudio graphical interface

On RTS clusters the only to access the RStudio interface is through single-threaded sessions running directly on a cluster login node.

These sessions are not designed for production work, but only for testing, debugging, and validation. Some visualization workloads may run, if their resource requirements fit within the session limits.

Please remember, on login nodes:

* Always run RStudio code on login nodes in single-threaded form
* Memory will often be limited (varies based on other login node operations)
* Session wall-clock duration will always be limited, with automatic session disconnection (six hours)
* Designed for syntax testing, validation, debugging, visualization

Starting a single-threaded, interactive, graphical **RStudio** session on a login node

Log into a cluster login node using SSH or X2Go. (Installing and using these connection tools are covered in other RTS documents.)   
  
If using SSH, include the -Y parameter to enable graphical display redirection.   
  
If using X2Go, open a terminal window on the login node.

Install R and RStudio in a user conda environment

Start by identifying the current (default) python settings and version:

[user@chapman.edu@keckcluster:~]$ module list

Currently Loaded Modulefiles:  
1) gcc/7.2.0 2) slurm/17.11.8

[user@chapman.edu@keckcluster:~]$ which python

/usr/bin/python

[user@chapman.edu@keckcluster:~]$ python --version

Python 2.7.5 <== default system python

Next, load the anaconda3 environment module:

[user@keckcluster]# module load anaconda3/current

[user@keckcluster]# module list  
 Currently Loaded Modulefiles:  
 1) gcc/7.2.0 2) slurm/17.11.12 3) anaconda3/current

The session is modified to use Anaconda's python instead of the system default.

Verify the altered python settings and version:

[user@chapman.edu@keckcluster:~]$ module list

Currently Loaded Modulefiles:

1) gcc/7.2.0 2) slurm/17.11.8 3) anaconda3/current <== anaconda3 module loaded

[user@chapman.edu@keckcluster:~]$ which python

/cm/shared/apps/anaconda3/bin/python

[user@chapman.edu@keckcluster:~]$ python –-version

Python 3.6.6 :: Anaconda custom (64-bit) <== python version changed

Create and then activate a user conda environment for your project/workload:

[user@chapman.edu@keckcluster:~]$ conda info --envs

# conda environments:

#

base \* /cm/shared/apps/anaconda3 <== base/root system conda environment (read-only)

[user@chapman.edu@keckcluster:~]$ conda create -n your\_project\_name python=3.6 anaconda <== choose python version

/ Solving environment: done

## Package Plan ##

environment location: /home/user@chapman.edu/.conda/envs/your\_project\_name

Proceed ([y]/n)? y

Preparing transaction: done

Verifying transaction: done

Executing transaction: done

#

# To activate this environment, use:

# > source activate your\_project\_name

#

# To deactivate an active environment, use:

# > source deactivate

#

*Activate a user conda environment (cont.):*

[user@chapman.edu@keckcluster:~]$ conda info --envs <== list existing environments

# conda environments:

#

base \* /cm/shared/apps/anaconda3

your\_project\_name /home/user@chapman.edu/.conda/envs/your\_project\_name <== new user environment

[user@chapman.edu@keckcluster:~]$ source activate your\_project\_name <== make user environment active

(your\_project\_name)

[user@chapman.edu@keckcluster:~]$ <== prompt now prefixed with active environment

The list of currently loadable Python packages is the sum of the active environment’s packages and the base environment’s packages:

(your\_project\_name)

[user@chapman.edu@keckcluster:~]$ conda list <== list packages installed in active/user environment

# packages in environment at /home/user@chapman.edu/.conda/envs/your\_project\_name:

#

# Name Version Build Channel

...list of packages installed by user...

(your\_project\_name)

[user@chapman.edu@keckcluster:~]$ conda list -n base <== list packages installed in base/root environment

# packages in environment at /cm/shared/apps/anaconda3:

#

# Name Version Build Channel

...list of packages installed by root...

To install additional packages into the current environment:

(your\_project\_name)

[user@chapman.edu@keckcluster:~]$ conda search package\_name <== verify installable package exists

Loading channels: done

# Name Version Build Channel

package\_name 0.11.0 np15py26\_2 pkgs/free

package\_name 0.12.0 np15py26\_3 pkgs/free

package\_name 0.13.0 np15py37\_ce1 pkgs/free <== newer versions last

...list of all available package versions...

(your\_project\_name)

[user@chapman.edu@keckcluster:~]$ conda install package\_name <== installs latest version plus dependencies

# Solving environment: done into active environment location (user homedir)

## Package Plan ##

environment location: /home/user@chapman.edu/.conda/envs/your\_project\_name

added / updated specs:

- package\_name

The following packages will be downloaded:

package | build

---------------------------|-----------------

mkl\_fft-1.0.6 | py37h7dd41cf\_0 150 KB

intel-openmp-2019.1 | 144 885 KB

...installation output...

(your\_project\_name)

[user@chapman.edu@keckcluster:~]$ conda install package1 package2 package3 <== install all required packages at

once for best dependency resolution

Run RStudio.

Start the graphical RStudio interface:

[user@chapman.edu@keckcluster]# RStudio

The graphical RStudio client will open. Interactive sessions are always limited to six hours of wall-clock time.

Once your work is complete, close RStudio normally.

End your interactive session

Deactivate your user conda environment:

(your\_project\_name) <== prefixed environment name

[user@chapman.edu@keckcluster:~]$ source deactivate

[user@chapman.edu@keckcluster:~]$ <== prefixed environment name is removed

Exit your terminal session:

[user@chapman.edu@keckcluster:~]$ exit

If you are using the X2Go remote desktop client, log out of your MATE desktop session.

Starting an interactive, text-mode **R** session on a login node

To start the R client in text-only mode.

[user@keckcluster]# r

R version 3.6.1 (2019-07-05) -- "Action of the Toes"

Copyright (C) 2019 The R Foundation for Statistical Computing

Platform: x86\_64-conda\_cos6-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.

You are welcome to redistribute it under certain conditions.

Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.

Type 'contributors()' for more information and

'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or

'help.start()' for an HTML browser interface to help.

Type 'q()' to quit R.

[Previously saved workspace restored]

>

Quit R normally when finished. Use exit to end your terminal session.

Starting an interactive, text-mode **R** session on a compute node, using **SLURM**

To avoid the limits of the login nodes, interactive, text-only R can be run on the compute nodes through the resource scheduler, SLURM. (Interactive sessions are always limited to six hours of wall-clock duration.) Using the utility srun, the below command:

* requests resources from the CPU partition (--partition defq),
* on one node (--nodes=1),
* allocates one CPU core per task (--cpus-per-task=1),
* allocates two megabytes of memory per CPU core (--mem-per-cpu=2048), for a total of 2gb of memory, and
* allocates the use of one task/process at a time (--ntasks=1).

*[--ntasks=2 would allocate* ***two*** *tasks/processes/threads, which would also double the requests for CPU and memory.]*

[user@chapman.edu@keckcluster:~]$ srun --pty --partition defq –-nodes=1 --cpus-per-task=1 --mem-per-cpu=2048 --ntasks=1 bash

The job scheduler waits for a node to become available with enough resources free, and then runs the program listed at the end of the request (bash).

[user@chapman.edu@node001:~]$ <== node name has changed to node001

You can then open an interactive text-mode R session as specified above. (Interactive sessions are always limited to six hours of wall-clock duration.)   
The main capability this kind of session adds is the ability to test parallel code (more on parallelism below).

R Batch job submission through the cluster resource scheduling engine, SLURM

Submitting R scripts as batch jobs is the expected way to run production workloads on RTS clusters. Once a workload has been debugged and validated elsewhere, it is ready to be packaged as a .R script for submission to the cluster. Combined with a SLURM job-definition script, these two files comprise a SLURM R batch job. Here is an example .R script, saved in the user’s home directory on the cluster (simple.R).

# Define a variable.

x <- rnorm(10)

# calculate the mean of x and print out the results.

mux = mean(x)

cat("The mean of x is ",mean(x),"\n")

Here is the associated .sbatch script, saved in the same directory (simple.sbatch). The parameters in this script are identical to the previous example.

#!/bin/env bash

#SBATCH --job-name=simpleML

#SBATCH --output=simple.out

#SBATCH --error=simple.err

#SBATCH --partition=defq

#SBATCH --nodes=1

#SBATCH --cpus-per-task=1

#SBATCH --mem-per-cpu=2048

#SBATCH --ntasks=1

module load anaconda3/current

conda create --name your\_project\_name <== skips quickly if already exists on node

source activate your\_project\_name

conda install package1 package2 package3 <== skips quickly if already installed on node

Rscript --vanilla simple.R

The job would then be submitted with the command:

[user@chapman.edu@keckcluster:~]$ sbatch simple.sbatch

This causes the job to be placed in the requested queue (defq).   
It will be run as soon as the requested resources become available in the requested partition.

Monitor and manage R SLURM jobs

The .sbatch script sets the output filename to simple.out. The .out file will contain stdout for all job processes on all allocated nodes. The simple.err file will contain the matching stderr messages.

You can use the cat utility to print the contents of a file to your terminal window:

[user@chapman.edu@keckcluster:~]$ cat simple.out

While the job is running you can watch the contents of the .out and .err files in real time using the tail utility (use ctrl+c to quit):

[user@chapman.edu@keckcluster:~]$ tail -F simple.out

To see all running SLURM jobs use this squeue:

[user@chapman.edu@keckcluster:~]$ squeue

To manage all SLURM jobs you have rights to with scontrol, or cancel jobs with scancel:

[user@chapman.edu@keckcluster:~]$ scontrol

[user@chapman.edu@keckcluster:~]$ scancel {job-id} <== get the job-id from squeue

Local documentation for each command is available with the Linux man utility:

[user@chapman.edu@keckcluster:~]$ man {command}

The vendor's official documentation for the current SLURM version installed on RTS clusters is [here](https://slurm.schedmd.com/archive/slurm-17.11.12/).

More complex R batch jobs and SLURM: Parallel Code

This document covers R jobs on RTS clusters scaling to all of the CPU cores on a single node and 256 gigabytes of memory.

With the installation of additional R packages and more complex code, R jobs can scale to use all of the nodes in the cluster, thousands of CPU cores, and multiple terabytes of memory concurrently. Multiple-node R code jobs are currently beyond the scope of this document.

Running R parallel workloads as SLURM jobs

Parallelization in R offers two set of tools: the parallel library with the mclapply function, or the doParallel and foreach libraries with the makeCluster function and the %dopar% keyword. A good, brief introduction to using these two sets of tools can be found [here](https://nceas.github.io/oss-lessons/parallel-computing-in-r/parallel-computing-in-r.html).

Here is a very brief .R script (parallel.R) and .sbatch script (parallel.sbatch) example illustrating the second strategy:

install.packages("doParallel")

library(doParallel)

cl <- makeCluster(as.numeric(args[1]))

registerDoParallel(cl)

x <- iris[which(iris[,5] != "setosa"), c(1,5)]

trials <- 1000000

ptime <- system.time({

r <- foreach(icount(trials), .combine=cbind) %dopar% {

ind <- sample(100, 100, replace=TRUE)

result1 <- glm(x[ind,2]~x[ind,1], family=binomial(logit))

coefficients(result1)

}

})[3]

ptime

First, in the job’s associated .sbatch script (which we will look at soon), a multi-threaded job ***must*** be requested, using the parameter --ntasks={n>1}.

The number of --ntasks controls how many CPUs are allocated, defaulting to one CPU core per requested task. The total number of allocated CPUs is saved in a script-level shell environment variable, SLURM\_CPUS\_ON\_NODE that was passed into Rscript as the first parameter (used here as as.numeric(args[1])).

It is important to note here that currently, as described here, ***doParallel clusters are limited to the CPU cores within a single compute node***.

Parallelizing beyond a single node requires additional complexity in your R code and possibly changes to the architecture of your batch workload. (Both of which are beyond the scope of this document.) (Note to interested users: multi-node, parallel R SLURM jobs in RTS clusters can be constructed based on [this](https://cran.r-project.org/web/packages/rslurm/vignettes/rslurm.html) design. Please contact [RTS](mailto:researchcomputing@chapman.edu) for more information.)

The maximum CPUs for a doParallel cluster are, therefore: ***144 CPU cores on the defq/cpu\* partitions***, and ***40 CPU cores on the gpu\* partitions***.

In line three of the .R script, a doParallel cluster object named ‘cl' is instantiated to enable the foreach with %dopar% function in line nine. The doParallel cluster is created with the number of CPUs that were allocated in the SBATCH script (using the parameter passed to Rscript). The foreach function distributes each iteration of the loop to a different worker (i.e. a different CPU), running the iterations as concurrently as the number of workers allows.

Here is the associated SBATCH script (parallel.sbatch):

#!/bin/env bash

#SBATCH --job-name=parallel

#SBATCH --output=parallel.out

#SBATCH --error=parallel.err

#SBATCH --partition=defq

#SBATCH --time=00:10:00

#SBATCH --nodes=1

#SBATCH --ntasks=10

#SBATCH --cpus-per-task=1

module load anaconda3/current

conda create --name your\_project\_name <== skips quickly if already exists on node

source activate your\_project\_name

conda install package1 package2 package3 <== skips quickly if already installed on node

Rscript --vanilla parallel.R $SLURM\_CPUS\_ON\_NODE

This script allocates 10 parallel threads (ntasks) and one CPU core for each task, for a total of 10 CPUs. The CPU count gets saved as the session variable SLURM\_CPUS\_ON\_NODE, which is used to set the number of worker processes created in the .R script's doParallel cluster.

The job is then submitted with the command:

[user@chapman.edu@keckcluster:~]$ sbatch parallel.sbatch

and placed into the queue for the requested partition (defq). As soon as the requested resources are free, the .R script will run. The foreach loop will distribute its 100 iterations across the allocated CPUs, so each CPU will run ten iterations.

Monitor the SLURM output file to see the resulting output. In this example, the .sbatch file named the output file parallel.out.   
(If no filename is specified, the default name is slurm-NNN.out, where NNN is the SLURM jobID. The single output file contains stdout and stderr for all nodes.)

[user@chapman.edu@keckcluster:~]$ cat parallel.out