Cornell University
Center for Advanced Computing

Programming for Stampede 2 with Python or R

Adam Brazier
Computational Scientist
Cornell University Center for Advanced Computing (CAC)
brazier@cornell.edu

High Performance Computing on Stampede 2, with KNL

Overview

- Introduction
  - Themes, Overview
  - Scope
  - Resources
  - Visualization Portal

- Python
  - Compiled code
  - Parallelization: MKL/automagic, multiprocessing, MPI
- R
  - Parallelization: MKL/automagic, SNOW, RMPISNOW

Themes

- Using the right libraries and interpreters
- Integration with compiled code (in Python)
- Most importantly, parallelization
  - Automagic, MKL
  - Multicore operations
  - MPI

HPC? In a high-level language?

- Both Python and R are used commonly in scientific research, research which is producing increasing amounts of data
  - Data products you are trying to analyze may have been produced on Stampede
- Necessary data analysis in Python or R may become too slow, or computers may run out of memory
  - Stampede nodes have more cores and more RAM than your laptop
  - Re-implementing in C or Fortran may not be feasible or desirable!
- Parallelism can improve performance of many Python/R applications, even without fine-grained control over what is happening in the hardware

Scope

- Not an “introduction to programming Python/R” course, but assumes no particular level of expertise
  - Assumes no more Stampede expertise than discussed in preceding lectures in this workshop
- Two key strands:
  - What sort of things can I do to make it run faster/better?
  - Basic examples of some technologies that will server many/most Stampede2 use cases in Python and R
- I will use “Stampede” as the descriptor, because we’ll largely be running on old Stampede, learning what will work on Stampede-KNL
  - I will avoid some techniques which might not work on Stampede-KNL

For this workshop

- We will be using:
  - Standard Stampede logins, so ssh to stampede.tacc.utexas.edu
  - Allocation: : TG-TRA140011
  - Reservation: CAC1
  - Queue (if needed): normal-mic
  - Scripts are under R_Python_Workshop
    - /python_scripts
    - /R_scripts
  - One correction: one file in R_Python_Workshop/Rscripts needs to be replaced; you can get the corrected version at:

    ~tg459572/LABS/labsJan2017/R_Python_Workshop/R_scripts/Run_SimpleSNOW.sh

Other resources

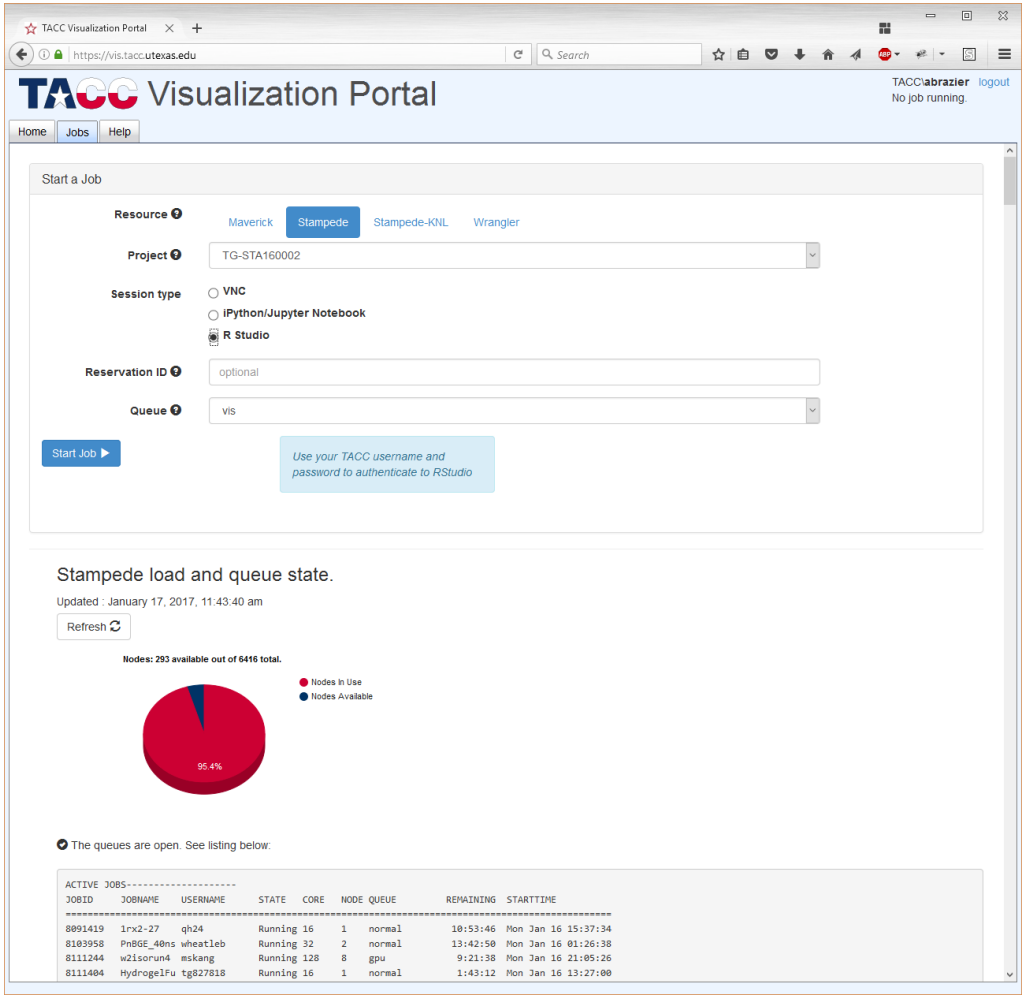
- All of the Python and R functions and libraries used are documented on the official Python and R documentation (or via CRAN, for R)

- All of the examples in this talk are from the Cornell Virtual Workshops Python for High Performance and An Introduction to R on XSEDE resources, which contain additional information to that covered here.

- Stampede documentation on the TACC portal contains some good information, and a search engine query of something like “TACC Stampede HPC [R/Python]” works pretty well for finding material.
  - Eg, David Walling’s presentation on “High Performance R”

Visualization portal

- http://vis.tacc.utexas.edu



Access through the Visualization Portal

- Gives access to *one* compute node.
- Shows current utilization on chosen resource.
- OMP_NUM_THREADS may not be set, should default to number of cores, and MKL should be able to use multithreading automatically. However, you can set it by calling a shell or setting system environment variables in code
- Visualization portal has Jupyter (allowing Python and R), R Studio and VNC. Typically asks for four hours but session can be terminated earlier. Choice of queues, should typically use “vis”.

Python

- Python very popular in the sciences
- Examples here use Python 2.7 but much of it works the same in Python 3 (however, no mpi4py in Python 3 on Stampede)
- We aren't covering "writing good code", but of course, writing good code is desirable if good performance is required
- We will use console submission of jobs, but Jupyter (fkas iPython) is available on the Visualization Portal
- Exploiting Stampede compute node capabilities requires parallelisation

You can run C/FORTRAN from Python

- Several ways to call code in a lower-level language from Python:
  - SWIG: create Python-callable libraries, from code written in C/C++
  - F2PY: allows calling Fortran (mostly F77) code from Python
  - Cython: generates compiled code from Python, callable *from* Python
  - Write your own C to call from Python!
  - Use subprocess to call compiled C code as if from command-line

Use the right packages/modules!

- If your software is built against the Intel Math Kernel Library (MKL)
- In particular, using the Numpy and Scipy provided by TACC will result in optimized calls to LAPACK and BLAS
- You get these with:

```
$ module load python
```

```
$ module load python3*
```

  - first, type `$ module spider python3` to get instructions on other required modules

** But not for MPI jobs!*

Multiple processes I—threading is still sequential

- Python has a `threading` module, which seems promising…

Multiple processes I—threading is still sequential

- Python has a `threading` module, which seems promising…

- But it produces sequential code. From the python documentation:
  - In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use multiprocessing or concurrent.futures.ProcessPoolExecutor. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

- This isn't what we normally want (unless we *are* I/O-bound, or need large amounts of RAM so that additional processes aren't viable)

Multiple processes I—threading is still sequential

- Python has a `threading` module, which seems promising…
- But it still produces sequential code:
  - In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use multiprocessing or concurrent.futures.ProcessPoolExecutor. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.
- This isn't what we normally want (unless we *are* I/O-bound, or need large amounts of RAM so that additional processes aren't viable)

Multiple processes—the multiprocessing package

- We can use the `multiprocessing` package
- `multiprocessing` creates separate processes which run in parallel and offers a similar API to the `threading` package
- Creating a process does have some extra overhead, but if the process runs long enough it's worth it
  - You create a pool of processes to which you then assign a function
  - Not as fast as a genuine threaded environment as inter-process communication slower than inter-thread communication, but performance benefits can still be considerable

Lab 1: Multiprocess example

- python_multiprocessing.py

```
from multiprocessing import Pool
def f(x):
    return x*x

p = Pool(4)                    # starts 4 worker processes
print(p.map(f, range(10)))   # prints [0, 1, 4,..., 81]
```

Lab 1: Multiprocess example

- python_multiprocessing.py

Importing Pool to let us create processes

```
from multiprocessing import Pool
def f(x):
    return x*x

p = Pool(4)                    # starts 4 worker processes
print(p.map(f, range(10)))   # prints [0, 1, 4,..., 81]
```

Lab 1: Multiprocess example

- python_multiprocessing.py

```
from multiprocessing import Pool
def f(x):
    return x*x

p = Pool(4)                         # starts 4 worker processes
print(p.map(f, range(10)))  # prints [0, 1, 4,..., 81]
```

Defining the function we're going to run in our processes

Lab 1: Multiprocess example

- python_multiprocessing.py

```
from multiprocessing import Pool
def f(x):
    return x*x

p = Pool(4)                    # starts 4 worker processes
print(p.map(f, range(10)))   # prints [0, 1, 4,..., 81]
```

Lab 1: Multiprocess example

- python_multiprocessing.py

```
from multiprocessing import Pool
def f(x):
    return x*x

p = Pool(4)                    # starts 4 worker processes
print(p.map(f, range(10)))     # prints [0, 1, 4,..., 81]
```

Chunks up and sends the iterable, `range(10)`, to the pooled processes and prints their output; like the built-in `map()` function but only takes one iterable

Lab 1: Multiprocess example

- python_multiprocessing.py

```
from multiprocessing import Pool
def f(x):
    return x*x

p = Pool(4)                    # starts 4 worker processes
print(p.map(f, range(10)))   # prints [0, 1, 4,..., 81]
```

Run in an interactive session:

```
$ idev -r
$ module load python
$ python python_multiprocessing.py
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Python and MPI

- The mpi4py package allows us to run MPI Python, across nodes
- mpi4py initializes MPI when imported and contains all the standard MPI calls
- mpi4py is already present on Stampede in Python 2.7
- For production code, exchange data as numpy arrays (see Cornell Virtual Workshop “Python for High Performance” for an example)

Python and MPI

- mpi_python.mpi

```
from mpi4py import MPI
import socket

comm = MPI.COMM_WORLD
print "Hello! I am rank %02d from %02d on host %s \n"
% (comm.rank , comm.size , socket.gethostname())
```

Lab 2: mpi4py

- mpi_python.mpi

```
from mpi4py import MPI
import socket

comm = MPI.COMM_WORLD
print "Hello! I am rank %02d from %02d on host %s \n"
% (comm.rank , comm.size , socket.gethostname())
```

Imports the `MPI` functionality, and also `socket` for our “here I am” test

Lab 2: mpi4py

- mpi_python.mpi

```
from mpi4py import MPI
import socket

comm = MPI.COMM_WORLD
print "Hello! I am rank %02d from %02d on host %s \n"
% (comm.rank , comm.size , socket.gethostname())
```

Creates intracommunicator instance

Lab 2: mpi4py

- mpi_python.mpi

```
from mpi4py import MPI
import socket

comm = MPI.COMM_WORLD
print "Hello! I am rank %02d from %02d on host %s \n"
% (comm.rank , comm.size , socket.gethostname())
```

Each process reports back

Lab 2: mpi4py

- mpi_python.mpi

```
from mpi4py import MPI
import socket

comm = MPI.COMM_WORLD
print "Hello! I am rank %02d from %02d on host %s \n"
% (comm.rank , comm.size , socket.gethostname())
```

```
$ idev -N 2 -n 24
$ module load python
$ ibrun python mpi_python.mpi
Hello! I am rank 09 from 24 on host c557-303.stampede.tacc.utexas.edu
Hello! I am rank 00 from 24 on host c557-303.stampede.tacc.utexas.edu
…
```

R on Stampede 2: basics

- *Don't* run R Scripts on login nodes!
- *Do* use:

```
module load Rstats
```

  - Note that `module load R` also works, but you don't get the optimized builds that way.

- Options for R include:
  - sbatch for traditional batch
  - idev for interactive sessions on compute notes
  - RDesktop on the visualization portal

R on Stampede 2: basics

- *Don't* run R Scripts on login nodes!
- *Do* use:

```
module load Rstats
```

  - Note that `module load R` also works, but you don't get the optimized builds that way.

- Options for R include:
  - sbatch for traditional batch
  - idev for interactive sessions on compute notes
  - RDesktop on the visualization portal ← We will be using this

R on Stampede 2: basics

- *Don't* run R Scripts on login nodes!
- *Do* use:

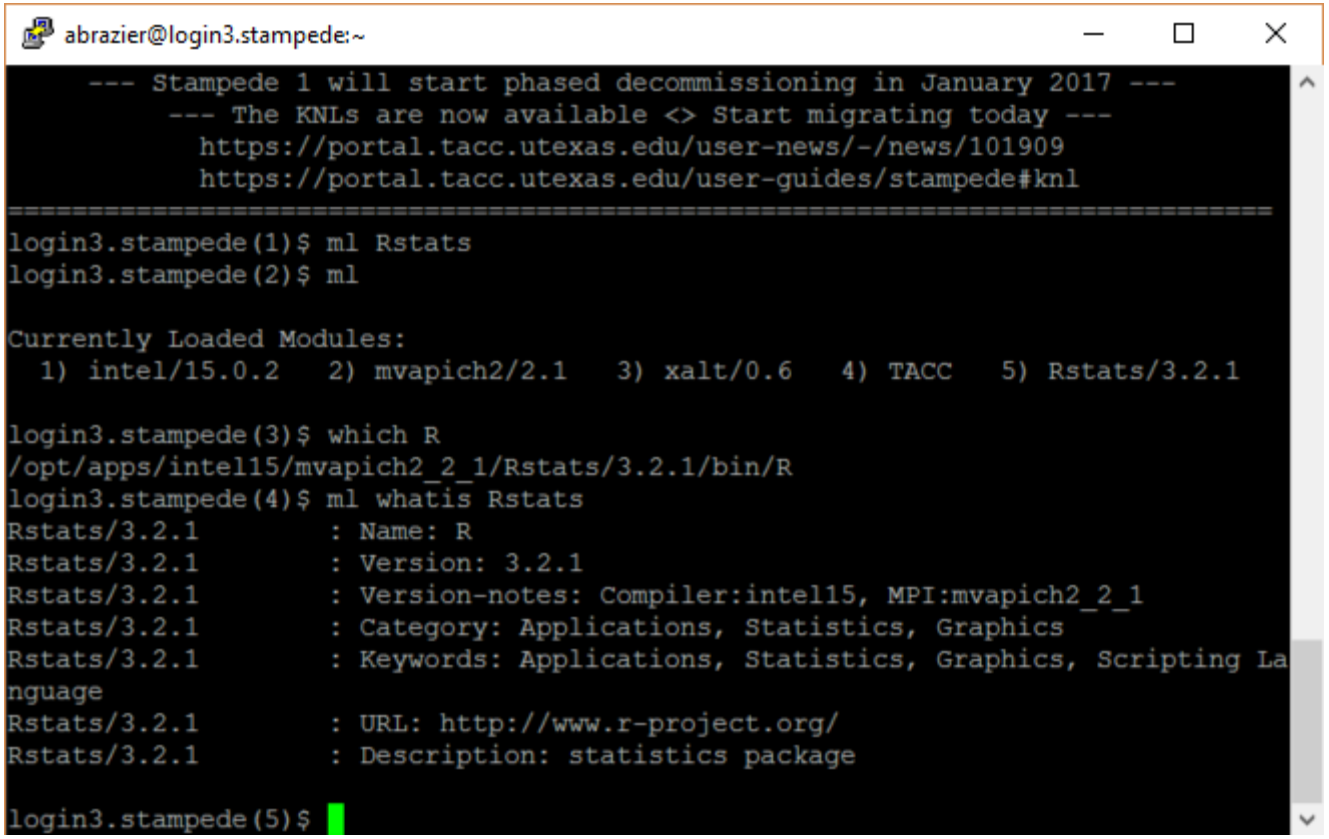
```
module load Rstats
```

  - Note that `module load R` also works, but you don't get the optimized builds that way.

- Options for R include:
  - sbatch for traditional batch
  - idev for interactive sessions on compute notes
  - RDesktop on the visualization portal

} But also using console

The bare-bones environment



Rstats?

- Includes a TACC-maintained optimized build of R
- Compiled with Intel compilers and linked against MKL math library
- We already told you this, but on Stampede, make sure you `module load Rstats` because although `module load R` also works on Stampede, you don't want to use that.
- Much of what you already know about Stampede, including batch and interactive jobs, is relevant to R on Stampede.

Multicore operations: the secret sauce

- R is, by default, single-threaded (as is the case with Python)
- On Stampede-KNL, as you have learnt, all the performance benefits come from running on multiple cores
- How to run on multiple cores in R?
  - The version of R built with MKL will give you automatic multithreading based on library heuristics, as we discussed for Python, earlier. The R Studio on the Vis portal also gives you this
  - You can use packages which have parallelism built in
  - You can use SNOW/RMPI
  - You can use Snowfall

Multicore operations: the secret sauce

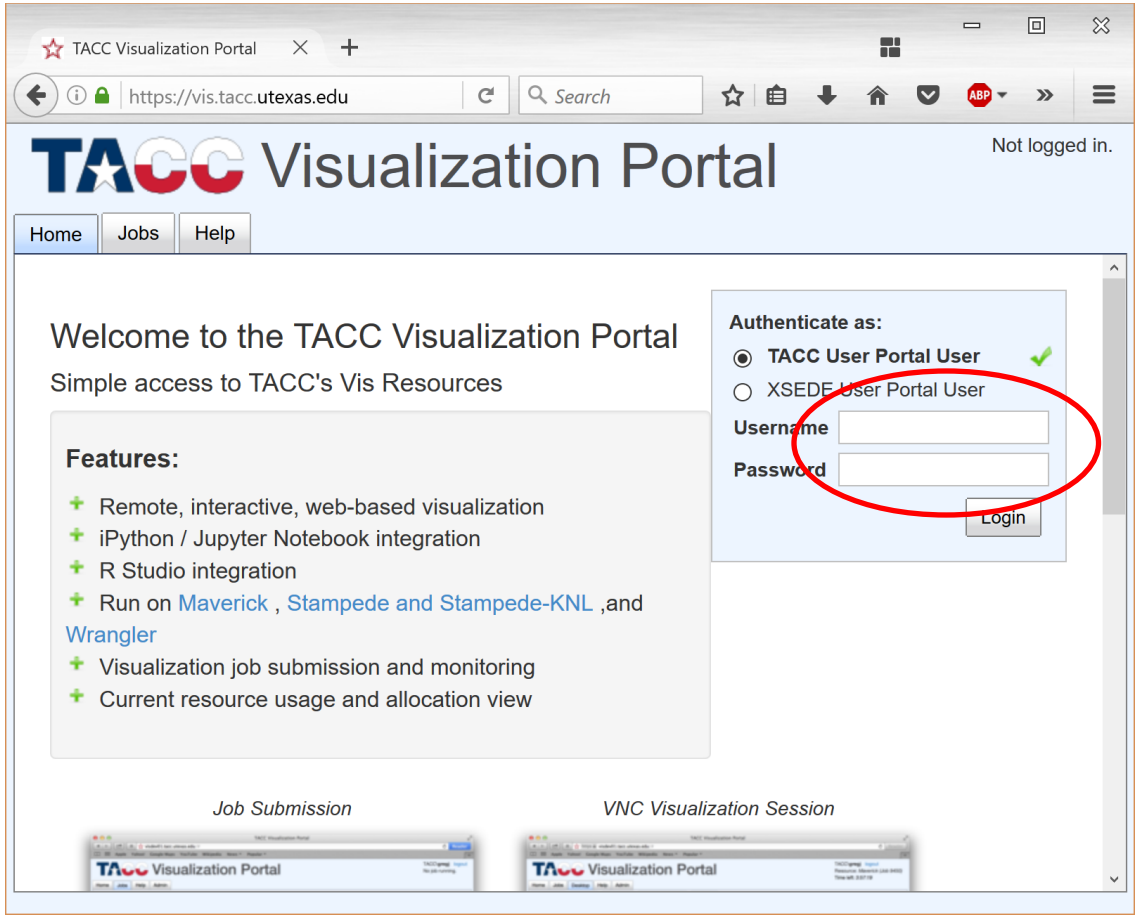
- R is, by default, single-threaded
- On Stampede-KNL, as you have learnt, all the performance benefits come from running on multiple cores
- How to run on multiple cores in R?
  - The version of R built with MKL will give you automatic multithreading based on library heuristics, as we discussed for Python, earlier. The R Studio on the Vis portal also gives you this
  - You can use packages which have parallelism built in
  - You can use SNOW/RMPI
  - You can use Snowfall

IMPORTANT!

Use the right package: multicore

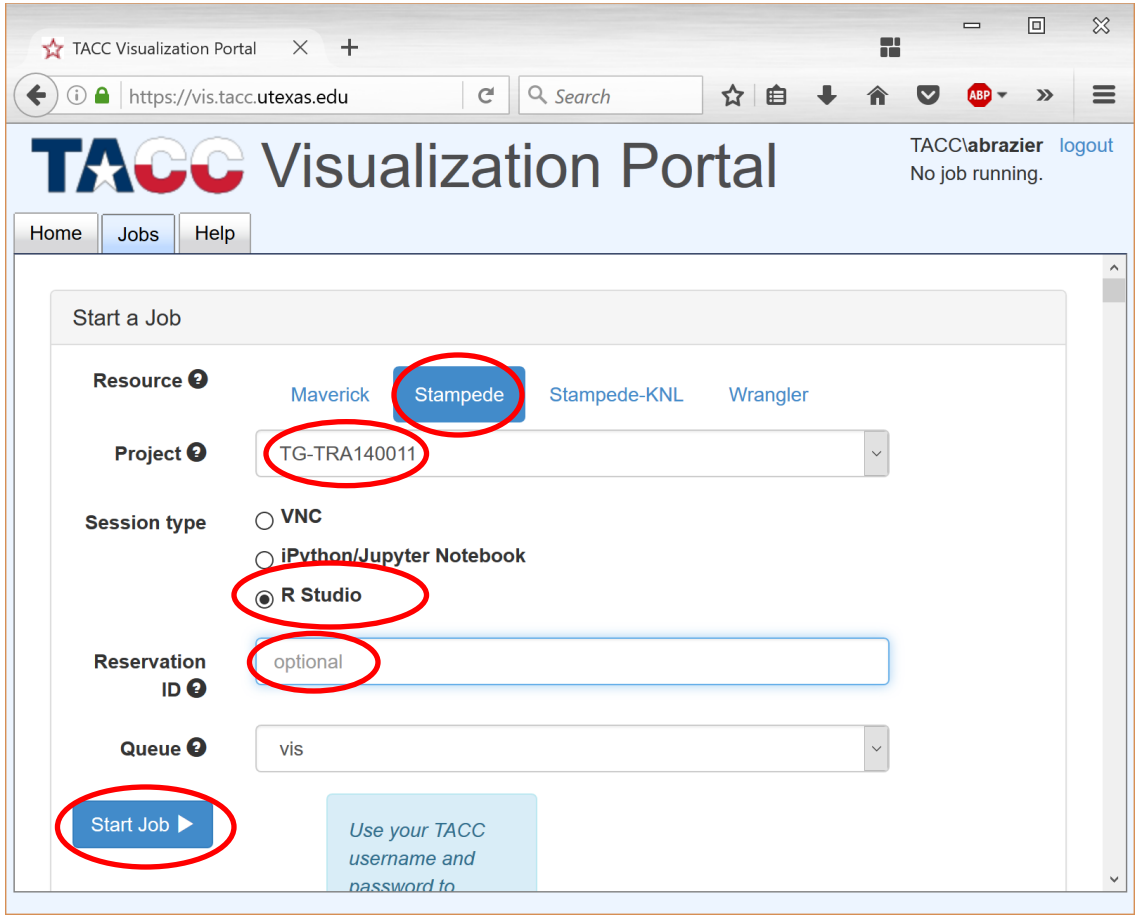
- The multicore package contains functions for parallel execution, where all spawned processes share the full state of R at spawning
- Configurable value for `cores` but defaults to all the available cores.
- A key function is `mclapply`, a multicore version of `lapply`
- `parallel` and `collect` are used to spawn processes and collect results

Lab 3: Rstudio and Multicore



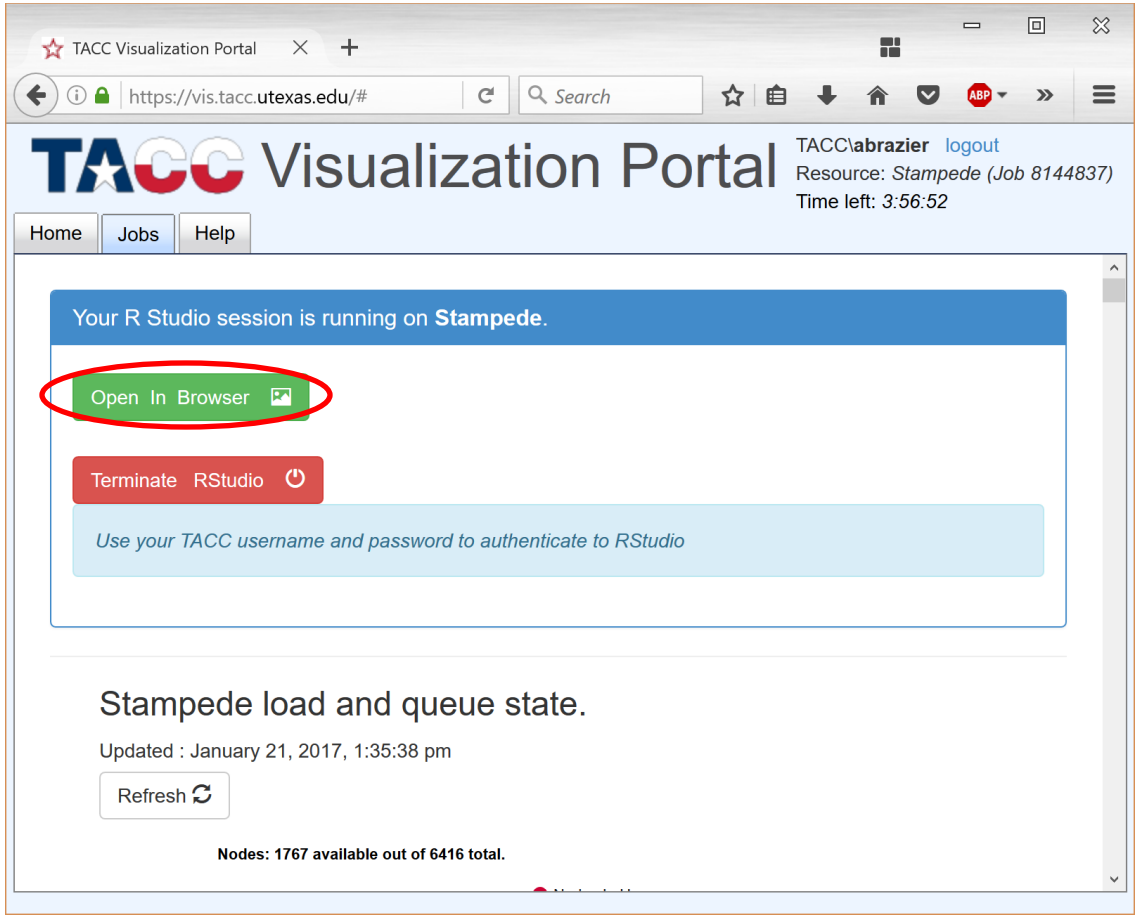
Login here

Lab 3: Rstudio and Multicore



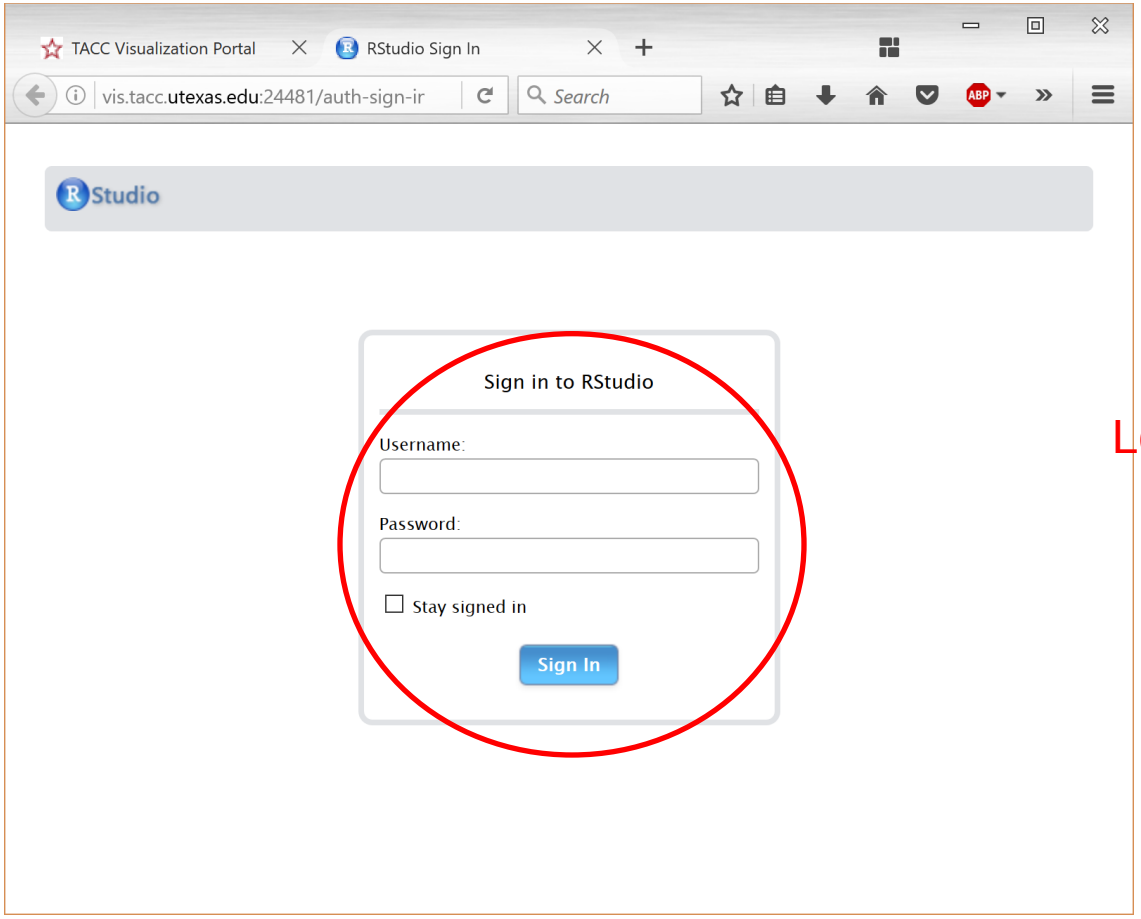
Important selections highlighted

Lab 3: Rstudio and Multicore



Start it up! (you will use the terminate button later)

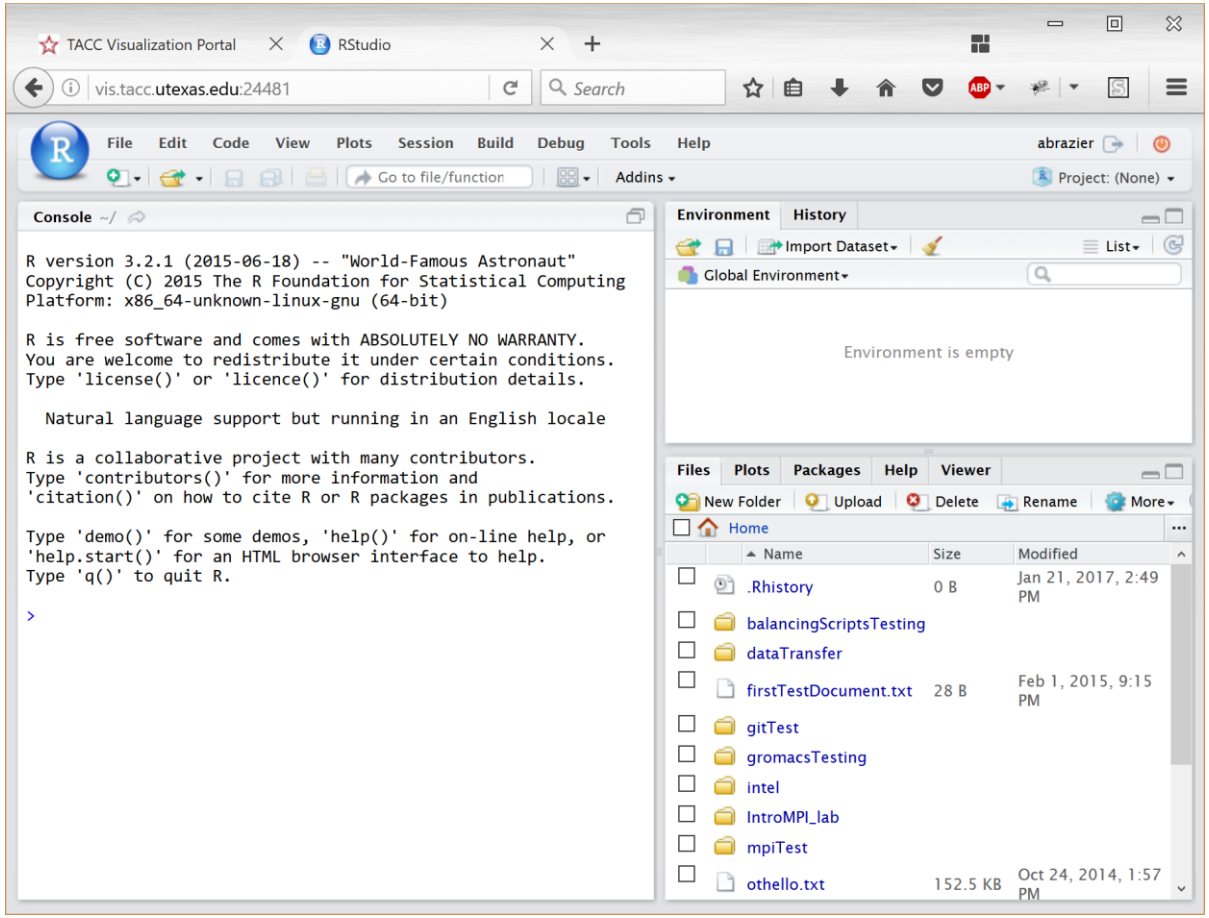
Lab 3: Rstudio and Multicore



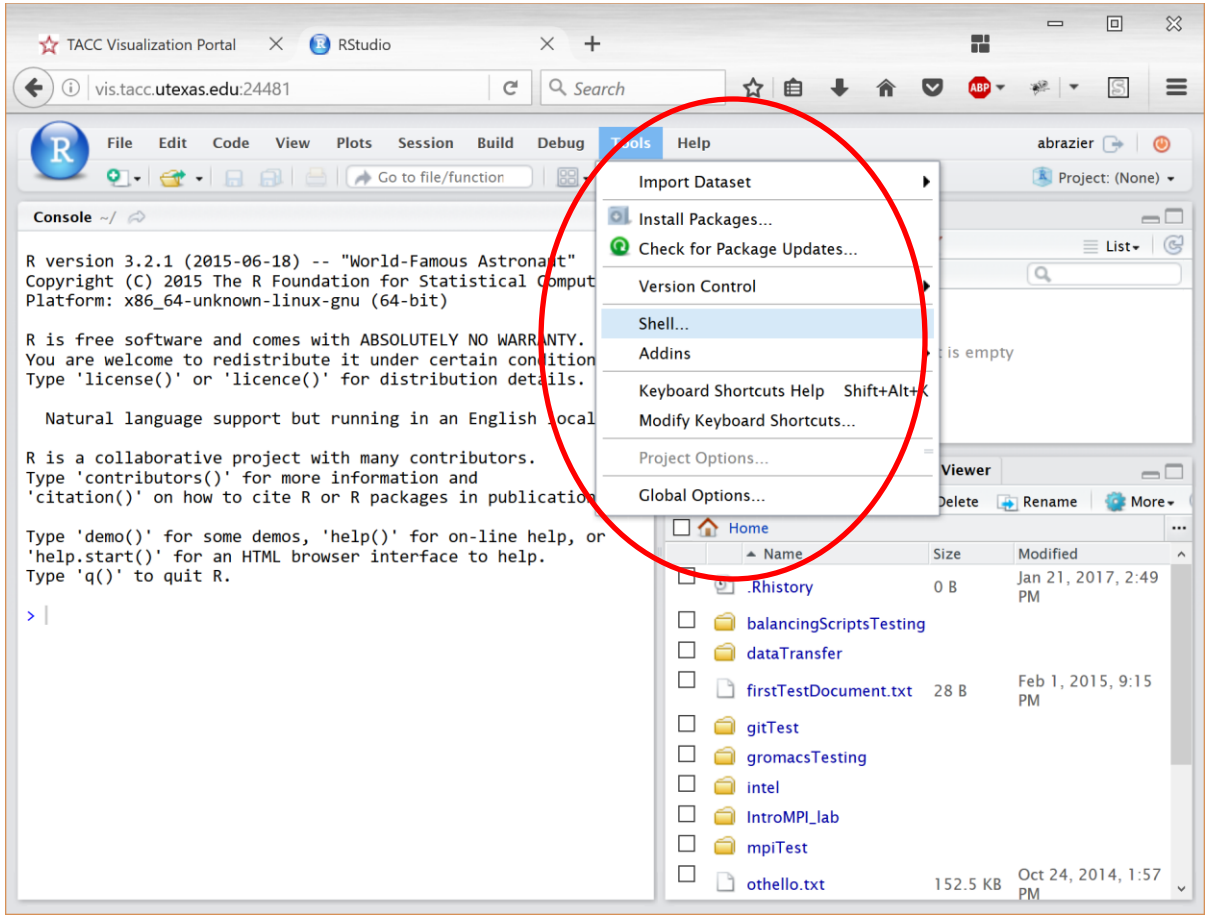
Login again!

Lab 3: Rstudio and Multicore

You are now on a compute node

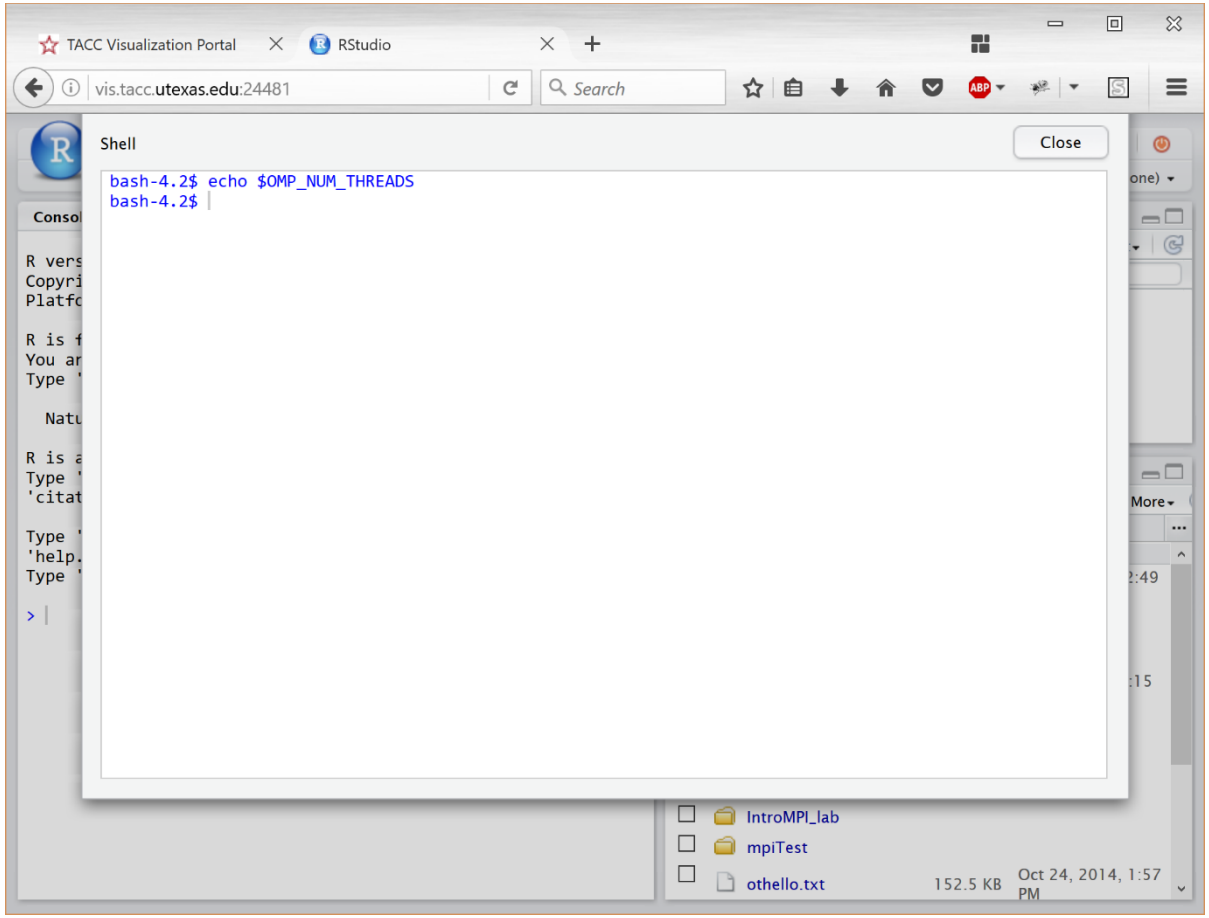


Lab 3: Rstudio and Multicore



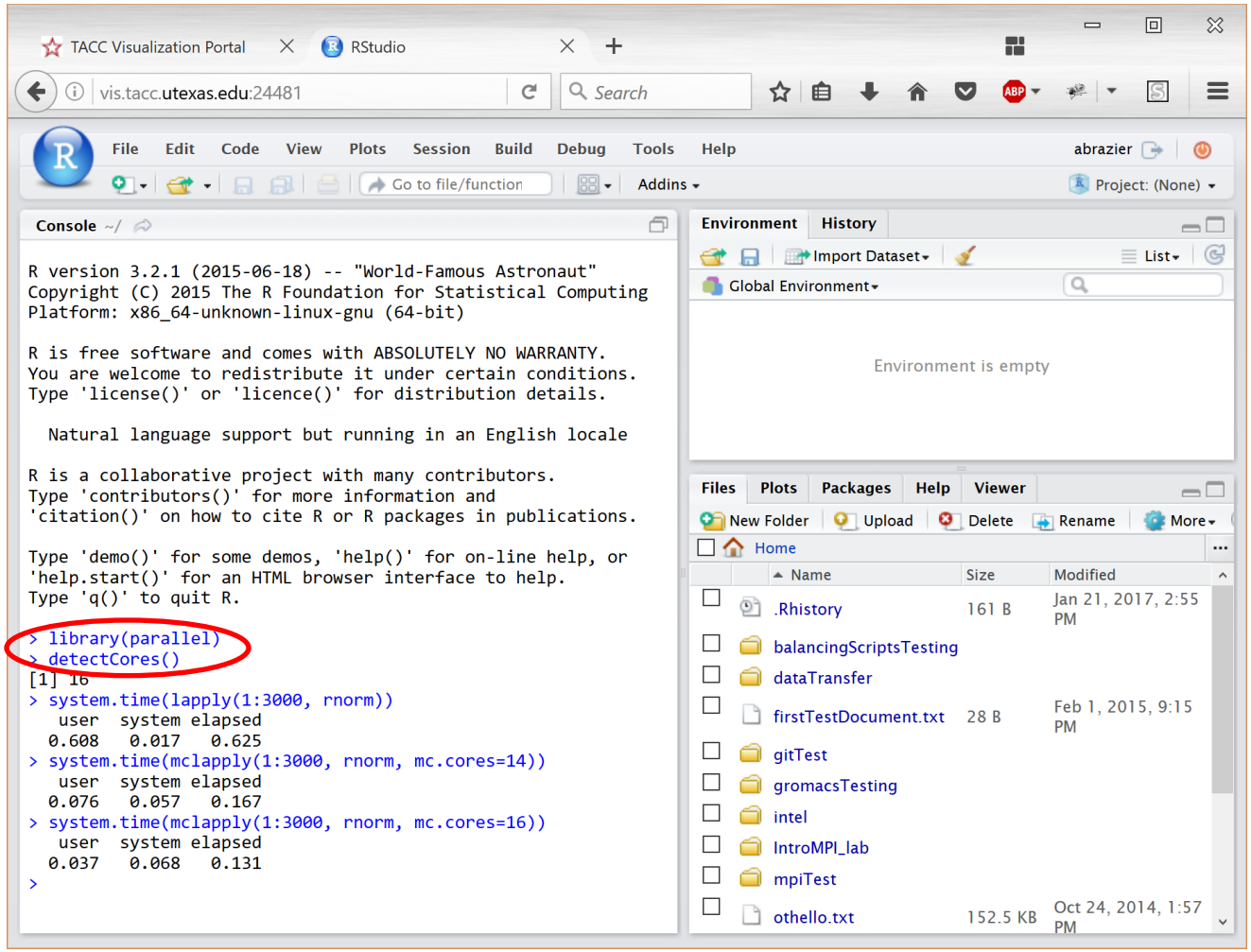
Lab 3: Rstudio and Multicore

OMP_NUM_THREADS is not set. You could set it here in shell



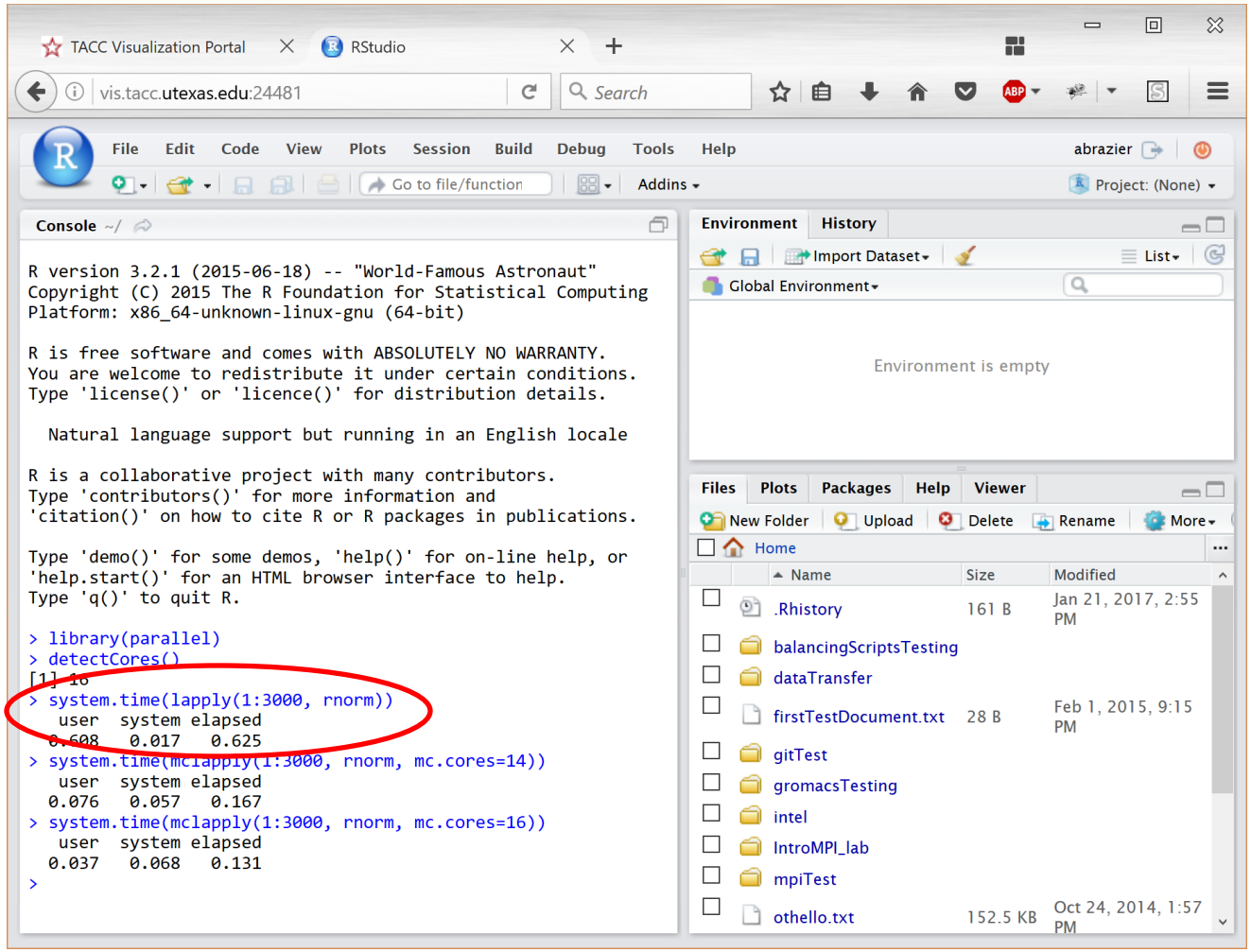
Lab 3: Rstudio and Multicore

Call up parallel library, check number of cores



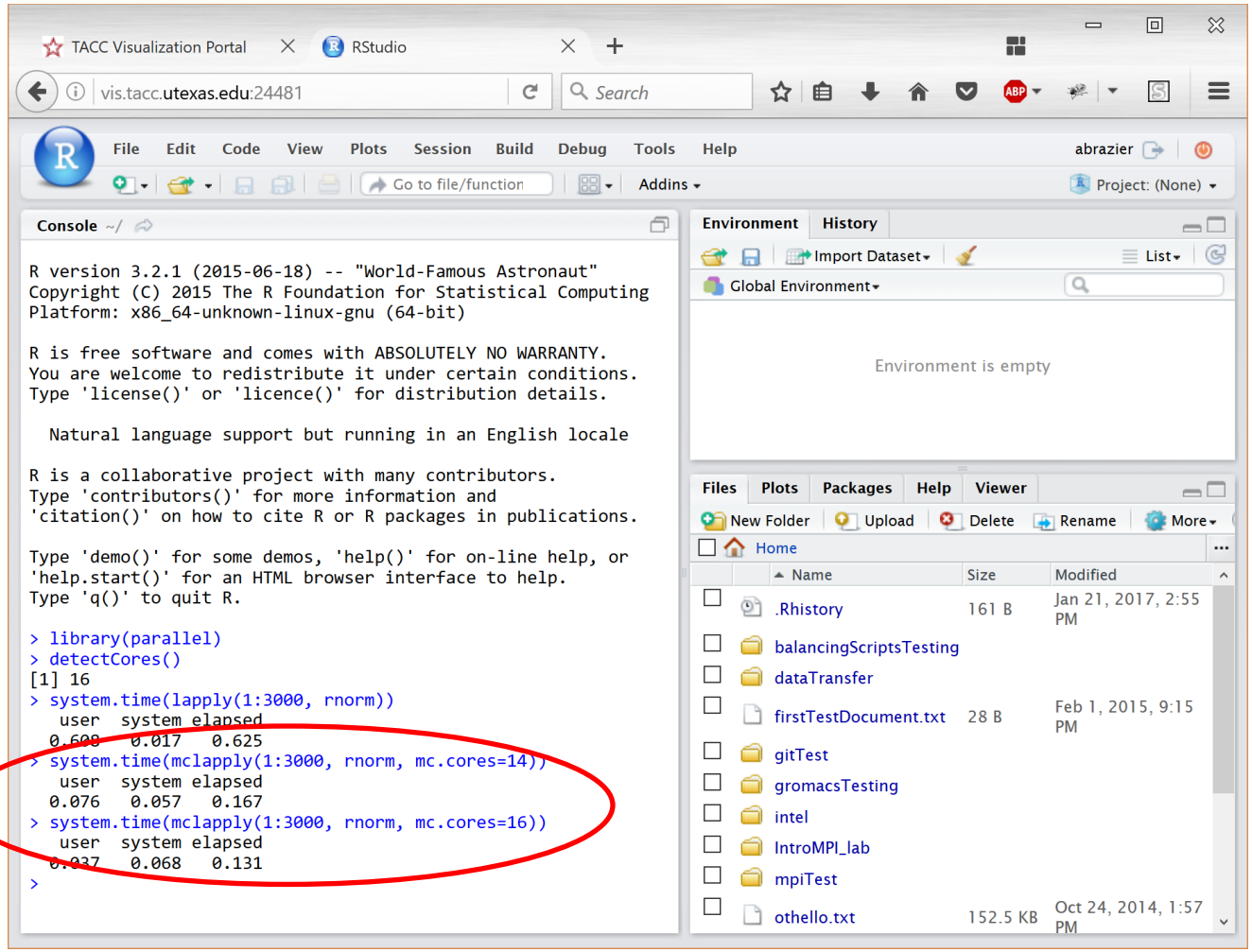
Lab 3: Rstudio and Multicore

Benchmark, single-core lapply generating normally distributed variables



Lab 3: Rstudio and Multicore

Use mclappy, try different numbers of cores



Details

Call up parallel library,
check number of cores

```
> library(parallel)
> system.time(lapply(1:3000, rnorm))
   user  system elapsed
  0.713   0.012   0.725
> system.time(mclapply(1:3000, rnorm, mc.cores=14))
   user  system elapsed
  0.145   0.082   0.252
> system.time(mclapply(1:3000, rnorm, mc.cores=16))
   user  system elapsed
  0.072   0.082   0.173
```

Details

```
> library(parallel)
> system.time(lapply(1:3000, rnorm))
   user  system elapsed
  0.713   0.012   0.725
> system.time(mclapply(1:3000, rnorm, mc.cores=14))
   user  system elapsed
  0.145   0.082   0.252
> system.time(mclapply(1:3000, rnorm, mc.cores=16))
   user  system elapsed
  0.072   0.082   0.173
```

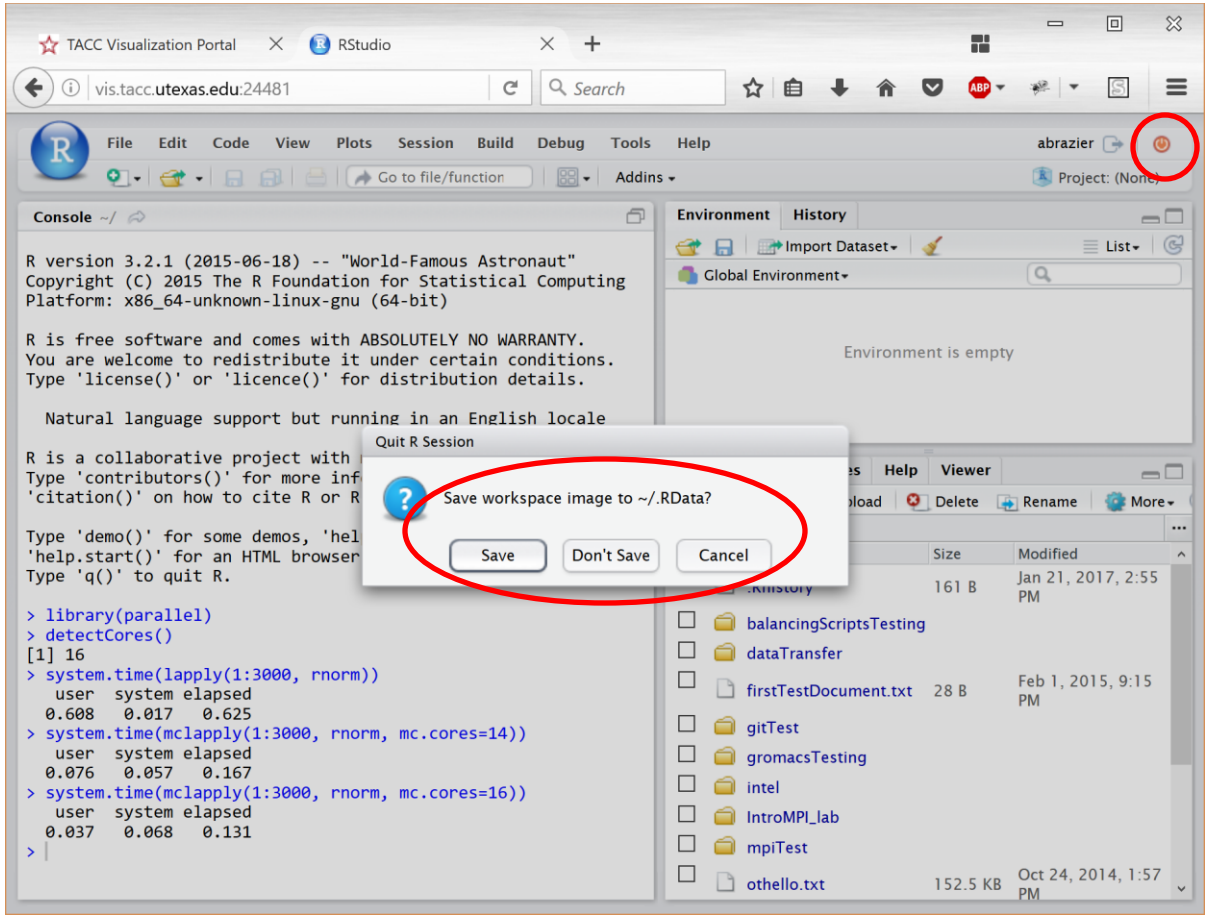
Benchmark, single-core lapply generating normally distributed variables

Details

```
> library(parallel)
> system.time(lapply(1:3000, rnorm))
   user  system elapsed
  0.713   0.012   0.725
> system.time(mclapply(1:3000, rnorm, mc.cores=14))
   user  system elapsed
  0.145   0.082   0.252
> system.time(mclapply(1:3000, rnorm, mc.cores=16))
   user  system elapsed
  0.072   0.082   0.173
```

Use mclappy, try different numbers of cores

Lab 3: Rstudio and Multicore

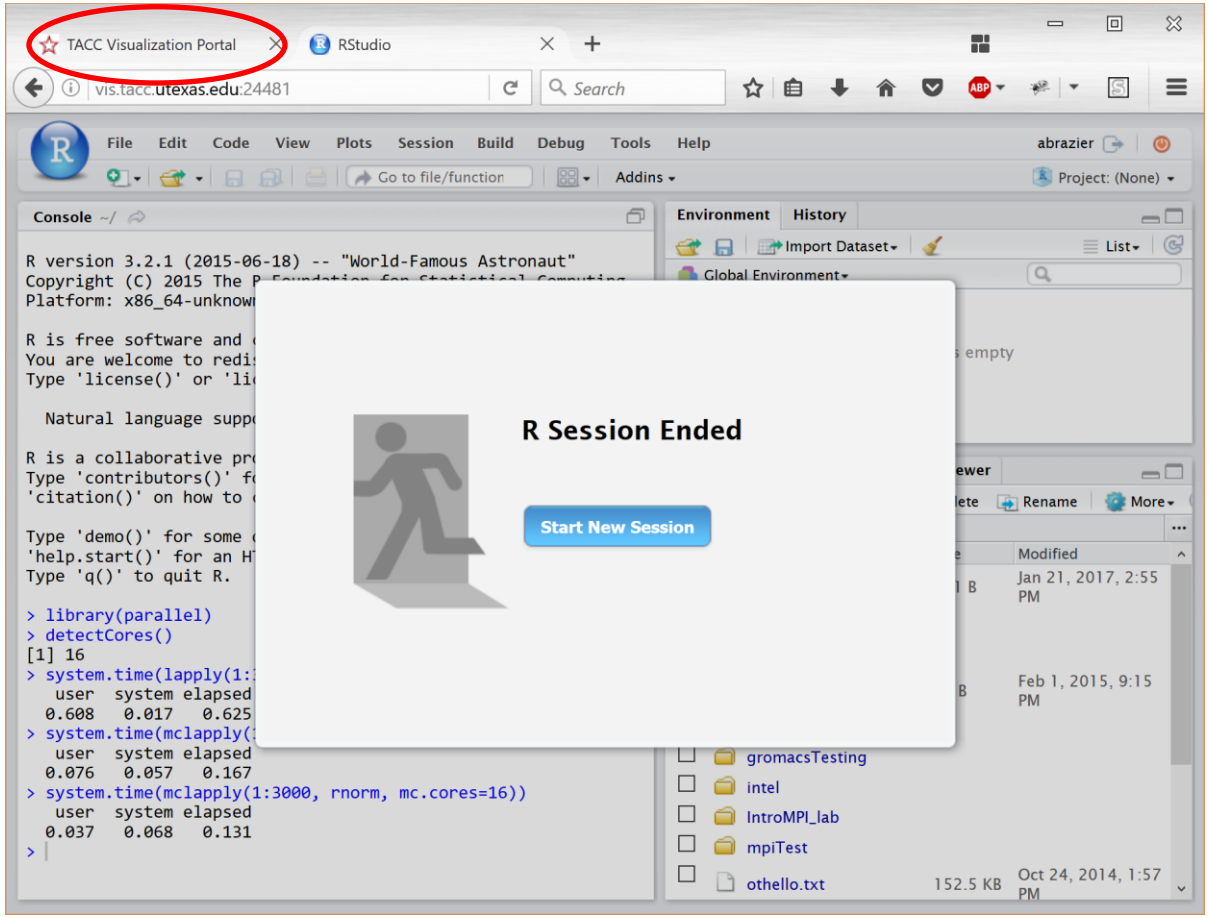


1st: quit session

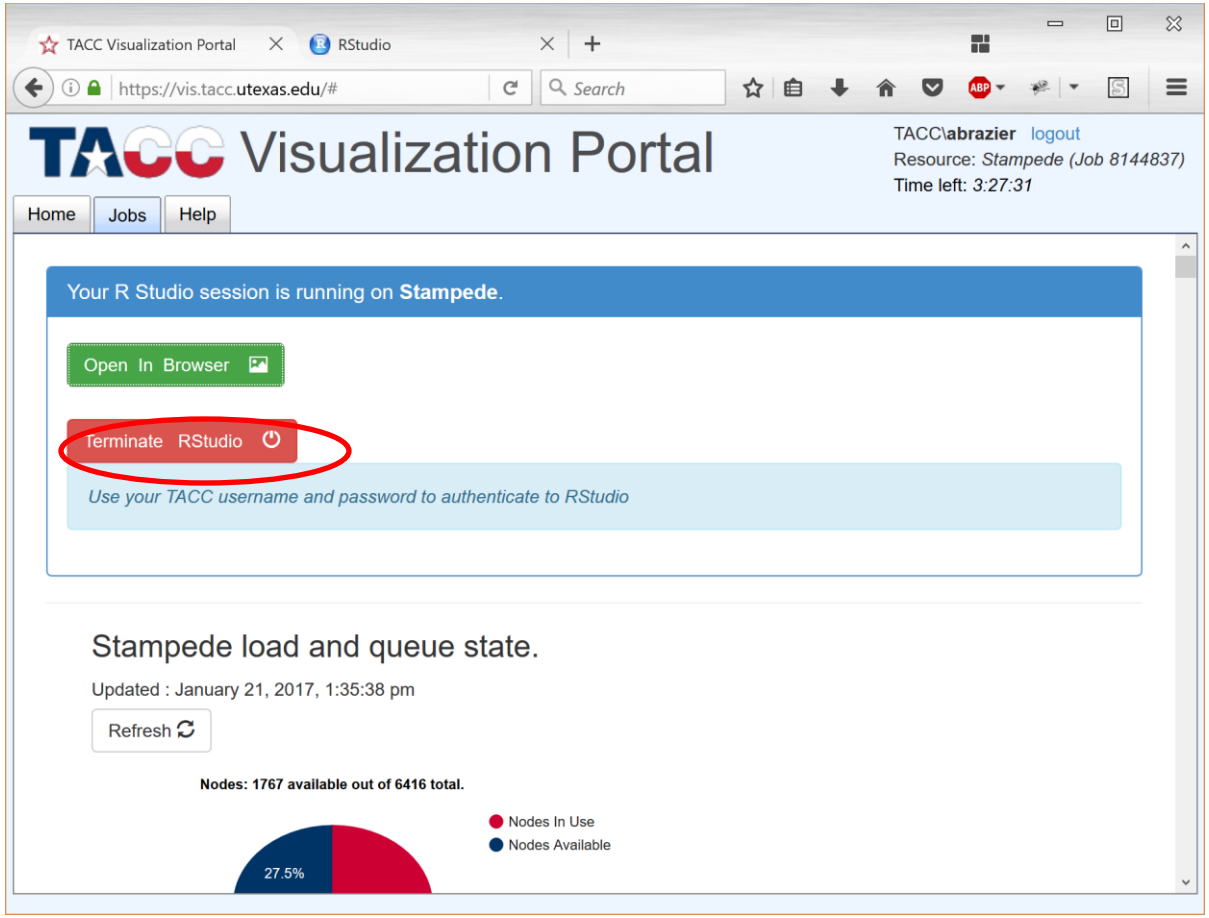
2nd: can save workspace to your home directory

Lab 3: Rstudio and Multicore

Return to vis portal page



Lab 3: Rstudio and Multicore



Terminate Rstudio/job

MPI with SNOW

- SNOW stands for Simple Network of Workstations. For embarrassingly parallel applications.
- SNOW is built atop RMPI, but you do not need to know MPI to use it
- Has a master/servant model, one master process controls the other processes, gathers the output and can perform additional processing
- Can be used on one node (Lab 4) or multiple nodes (Lab 5)

Lab 4: Let it SNOW on one node

- Look at birthday.R: `$ less -N birthday.R`

```
1 library(snow)
2
3 nmax = 50
4 nworkers <- as.numeric(Sys.getenv("SLURM_NPROCS"))
5
6 cl <- makeCluster(nworkers, type='SOCK')
7
```

Set up “cluster”

Lab 4: Let it SNOW on one node

- Look at birthday.R: `$ less -N birthday.R`

```
8  pbday <- function(n) {
9       ntests <- 1000
10      pop <- 1:365
11      anydup <- function(i)
12      any(duplicated(sample(pop, n,replace=TRUE)))
13      sum(sapply(seq(ntests), anydup)) / ntests
14 }
15 clusterExport(cl, list('pbday'))
16
17 # print the time to do nmax tests, after
distributing them to the workers
18 system.time( x <- clusterApply(cl, 1:nmax,
function(n) { pbday(n) })       18 )
```

Lab 4: Let it SNOW on one node

- Look at birthday.R : `$ less -N birthday.R`

```
8  pbday <- function(n) {
9       ntests <- 1000
10      pop <- 1:365
11      anydup <- function(i)
12      any(duplicated(sample(pop, n,replace=TRUE)))
13      sum(sapply(seq(ntests), anydup)) / ntests
14 }
15 clusterExport(cl, list('pbday'))
16
17 # print the time to do nmax tests, after
distributing them to the workers
18 system.time( x <- clusterApply(cl, 1:nmax,
function(n) { pbday(n) })      18 )
```

Experimentally evaluate probability of at least one shared birthday given n people

Lab 4: Let it SNOW on one node

- Look at birthday.R : `$ less -N birthday.R`

```
8   pbday <- function(n) {
9       ntests <- 1000
10      pop <- 1:365
11      anydup <- function(i)
12      any(duplicated(sample(pop, n,replace=TRUE)))
13      sum(sapply(seq(ntests), anydup)) / ntests
14  }
15  clusterExport(cl, list('pbday'))
16
17  # print the time to do nmax tests, after
distributing them to the workers
18  system.time( x <- clusterApply(cl, 1:nmax,
function(n) { pbday(n) })       18 )
```

Export to cluster and print time to evaluate for values of n from 1 to nmax, and assign computation output to x

Lab 4: Let it SNOW on one node

Calculate theoretical probability that no birthdays shared, for n up to nmax

- Look at birthday.R : `$ less -N birthday.R`

```
20 # compute the theoretical probability for each n
21 prob <- rep(0.0,nmax)
22 probnot <- 1.0
23 for (i in 2:nmax) {
24   probnot <- probnot*(366.0-i)/365.0
25   prob[i] = 1.0 - probnot
26 }
27
28 # print results, comparing tests to theory
29 z <- cbind(x,prob)
30 print(z)
```

Lab 4: Let it SNOW on one node

- Look at birthday.R : `$ less -N birthday.R`

```
20 # compute the theoretical probability for each n
21 prob <- rep(0.0,nmax)
22 probnot <- 1.0
23 for (i in 2:nmax) {
24   probnot <- probnot*(366.0-i)/365.0
25   prob[i] = 1.0 - probnot
26 }
27
28 # print results, comparing tests to theory
29 z <- cbind(x,prob)
30 print(z)
```

Output the experimental versus theoretical values, for each test

Lab 4: Let it SNOW on one node

- Now we run birthday.R (note, can use `$ Rscript ./birthday.R` if you don't want to see it line-by-line)

```
$ idev
$ module load Rstats
$ R --no-save < ./birthday.R
```

- Look for the runtime output and the displayed results comparing the two methods.

Lab 5: Let it SNOW on more than one node

- For this, we use RMPISNOW
- Unfortunately, we can't use the latest Rstats build for this on Stampede, but our batch script takes care of that.
- We will execute `SimpleSNOW.R` and call it from `Run_SimpleSNOW.sh`

Lab 4: Let it SNOW on more than one node

- Read Run_SimpleSNOW.sh: `$ less -N Run_SimpleSNOW.sh`

```
 1 #!/bin/bash
 2 #SBATCH -A XXXXXXXXXXX
 3 #SBATCH -N 2 -n 24
 4 #SBATCH -p XXXXXXXXXXX
 5 #SBATCH -t 00:10:00
 6 #SBATCH -J hello
 7 #SBATCH -reservation=XXXXXXXX
 8 module purge
 9 module load TACC
10 module load intel/14.0.1.106
11 module load Rstats
12
13 echo "say hello"
14 ibrun  RMPISNOW < ./SimpleSNOW.R
15 echo "done"
```

Lab 4: Let it SNOW on more than one node

- Read Run_SimpleSNOW.sh: `$ less -N Run_SimpleSNOW.sh`

```
 1 #!/bin/bash
 2 #SBATCH -A XXXXXXXXXXX
 3 #SBATCH -N 2 -n 24
 4 #SBATCH -p XXXXXXXXXXX
 5 #SBATCH -t 00:10:00
 6 #SBATCH -J hello
 7 #SBATCH -reservation=XXXXXXXX
 8 module purge
 9 module load TACC
10 module load intel/14.0.1.106
11 module load Rstats
12
13 echo "say hello"
14 ibrun  RMPISNOW < ./SimpleSNOW.R
15 echo "done"
```

Edit in allocation here

Edit in queue name here

Edit in reservation here (optional)

Getting right Rstats build

Send code to nodes

Lab 4: Let it SNOW on more than one node

- Read SimpleSNOW.R: `$ less -N SimpleSNOW.R`

```
2 cluster <- getMPIcluster()
3
4 # Print the hostname for each cluster member
5 sayhello <- function()
6 {
7   info <- Sys.info()[c("nodename", "machine")]
8   paste("Hello from", info[1], "with CPU type", info[2])
9 }
10
11 names <- clusterCall(cluster, sayhello)
12 print(unlist(names))
13
14 # stopCluster will call mpi.finalize, no need for mpi.exit
15 stopCluster(cluster)
```

Function to execute

Collect output, flatten and print to screen

End job, clear up

Lab 4: Let it SNOW on more than one node

- Run the code

```
$ sbatch Run_SimpleSNOW.sh
```

- Read the output, in a file something like `slurm-XXXXXX.out` and ignore the warnings about `.find.package`

- Just let it run and look for the output package; open it when it's visible

```
$ less -N slurm-XXXXXX.out
```

Lab 4: Let it SNOW on more than one node

```
$ less –N slurm-XXXXXX.out
```

```
137  [1] "Hello from c557-703.stampede.tacc.utexas.edu with CPU type x86_64"
138  [2] "Hello from c557-703.stampede.tacc.utexas.edu with CPU type x86_64"
....
158 [22] "Hello from c557-704.stampede.tacc.utexas.edu with CPU type x86_64"
159 [23] "Hello from c557-704.stampede.tacc.utexas.edu with CPU type x86_64"
160 >
161 > # stopCluster will call mpi.finalize, no need for mpi.exit
162 > stopCluster(cluster)
163 >
164
165 TACC: Shutdown complete. Exiting.
166 done
```

- Note that only 23 worker processes were used despite our request for 24: this is because it is assumed one process is needed to run it all

Snowfall. Rmpi

- "Snowfall" allows n(processes) > n(cores), but only on one Stampede node
  - Example on the Cornell Virtual Workshop "An Introduction to R on Stampede Resources"
- RMPI and pbdrMPI are also available. Requires more work from the coder but allows finer-grained control; some helpful advice can be found on David Walling's presentation "High Performance R"

Conclusions

- You need to use multiple cores!
  - In ascending difficulty/inconvenience, MKL, multithreading/processes, MPI
- You need to benchmark to find out how many threads/processes to run
- Visualization Portal is very good for many purposes (including, but not limited to, visualization!)
- Demonstrated effort to speed up your code is very helpful/necessary in getting more Stampede time