EEB QUARTERLY

UTM • St. George • UTSC

WINTER 2019

Volume 1 • Issue 2

NEWSLETTER

Toronto Winter • Mental Health • Job Board

FEATURES

Organism Profiles • Field Stories • Niagara

The Niagara cluster and some tips for running R scripts on it

Sean Anderson

THIS PAST MARCH, in a vast and windowless room at the SciNet data centre in Vaughan, 30 kilometres North of the St. George campus, a crew of technicians powered-up Canada's newest and fastest supercomputer: a cluster of 1500 Lenovo servers christened *Niagara*.

Of the four High Performance Computing (HPC) systems built in Canada since 2016 — part of a $120 million effort to replace a nation-wide set of ageing systems, including two recently decommissioned clusters at UofT — Niagara is by far the most powerful. Each of the 1500 servers or 'nodes' in Niagara contains 202 gigabytes of RAM and 40 Intel Skylake 2.4 gigahertz processing cores*. The nodes, piled in six foot stacks and arranged in rows, communicate to one another through thick bundles of baby-blue wire that run an ultra-high-speed connection system in the new "dragonfly+" topology. I won't go into much detail on dragonfly+, since I don't ~~have the first clue what it is~~ want to bore you. But the decision to run Niagara in this arrangement made tech-industry headlines, and you can trust me that it's all very new and exciting. You can also start bragging to your friends that the University of Toronto now houses the fastest academic research computer in the country.

Impressive, no doubt. But as we all know, a thing is only "good" if it helps one discern hidden truths regarding the evolution and distribution of life on earth. So what good is a room-sized cluster of liquid-cooled servers to the scholar in EEB?

The computer and its job-allocation system are operated by SciNet, the University of Toronto's HPC outfit. Scripts (or 'jobs') can be submitted freely to Niagara by anyone with a SciNet account, and SciNet accounts are available to anyone with a Compute Canada account, which is free. All faculty in EEB, by virtue of belonging to the University of Toronto, are eligible for such an account – as are any graduate students that a PI endorses. This means that with very minimal prodding to your advisor, you'll be able to run analyses on Niagara, an 18-million-dollar machine and the 53rd fastest computer on earth, at any time, from anywhere with internet access, absolutely free. The university basically brought a brand new Ferrari and you can take it for a spin just by signing out the keys.

The only cost to you is the time you'll spend adapting code for parallel runs and writing overhead scripts to communicate with Niagara's job-allocation software. That's where this tutori-

* Like all modern supercomputers, Niagara is a large collection of smaller computers joined by high-speed connections. Some terminology: A ***core*** is an independent processing unit – it does the work that a program tells it to do, independently of other cores. A ***server*** is a machine that takes instructions from other machines (called "***clients***") and carries out those instructions on its cores. The Lenovo servers in Niagara (called "***nodes***") accept instructions from a script you submit from your machine over the internet and run those instructions across each of their 40 cores. Niagara has 1500 servers*40 cores/server = 60,000 cores. For perspective, UofT's now-retired General Purpose Cluster had 30,240 cores. A macbook pro has 2.

Massively Parallel

Photo from www.scinethpc.ca

al can help. As I'll try to show, certain problems can readily be coded in R to run in parallel on multiple cores and across multiple nodes, and the overhead scripting can be automated. Once you're familiar with these steps and the Niagara interface, running jobs is fairly painless.

I'll focus on the R language since it's the *de facto* default data program in ecology and evolutionary biology. For many of us, getting use out of Niagara will mean running R on Niagara (for you genomicists using different software, the gnu-parallel approach below will probably apply to your work, same for those of you running Python or Octave*).

There is surprisingly little online instruction for adapting R scripts for clustered computers — and you'll often find that what does exist doesn't work. One reason is that R, while useful for many purposes, is not a high performance language; by HPC standards, interpreted languages like R, Python, and Matlab are very slow, with Python being the fastest. Nonetheless, R has some straightforward parallel functionality built-in, and running appropriate R tasks on the Niagara cluster can dramatically reduce your analysis time. A job that might take a week on a 16-core lab server might take a couple of hours using a handful of nodes on Niagara. So even though R

* It's important to note that running licensed programs on Niagara can be a hassle, so you may have to look for open-source alternatives. For matlab users that means Octave. You can refer questions about this to the scinet staff.

can't hang with Fortran in terms of speed, adapting R scripts for use in the cluster can definitely be worth your time. You still get to drive the Ferrari; you're just running on cheaper gas.

This tutorial is primarily intended for those with little or no experience using clustered computers. There are a number of ways to run R scripts on Niagara, but I'll focus here on a few very basic approaches. I can guarantee that the code here works as of December 2018, but I can't promise my examples illustrate the most efficient approach. If you're a regular Niagara user and have alternative suggestions, please send us a description and we'll add a note on the website. Some code, especially the forking parallel options, may not work in Rstudio.

I should stress that I am not — by any stretch of the imagination — an expert on high-performance computing, nor is this an exhaustive review of programming for clusters. But I have used Niagara and some of you haven't, so I've got a head start. I also spent a while testing out the performance of alternative approaches when I began parallel computing – and to be honest, it was a pain. My hope in writing this tutorial is that I can spare you some of that time and tedium and help you get started with what I think is a pretty incredible resource, one with the real potential to make you a more efficient researcher.

Before You Get Started

Is Niagara overkill?

The sports car analogy falls short in that – rich male insecurities aside – nobody *needs* a Ferrari. Your work, on the other hand, may very well need HPC capabilities. Even if you get by without Niagara, you may benefit greatly from its use. It all depends on what you're trying to program.

As a simple rule of thumb, if you think of your R script run time in terms of seconds, you'll probably gain very little from parallelization, and it could actually slow you down (running code in parallel comes with "overhead" time loss as information is sent to different cores or nodes). If you think in terms of minutes, running in parallel may benefit you, but a multi-core laptop or lab server will suffice. If you think in terms of hours or days, then you should be working in parallel if possible and should consider burdening Niagara with the heavy lifting.

If your work is computationally intensive to a much higher degree than the problem discussed in this tutorial. Niagara will certainly be valuable to you, but the R language probably won't be. You'll may have to use compiled programs written by others in your field or, if you've got the time and knack for it, write your own code in a compiled programming language like C or C++.

Is your code parallelizable?

The problems that we code range from "inherently serial" (i.e. can't be run in parallel) to "perfectly parallel" (sometimes called "embarrassingly parallel"). An inherently serial problem is, for example, a calculation in which one function's output is another function's input. You can't process these functions separately on different cores, since one depends on the other.

For a problem to be perfectly parallel, you need to be able to divide it into parts that can run independently. An example would be calculating the output of a function given multiple inputs values, which might be stored as elements in a list; it's easy to run the function on different parts of the list on different cores at the same time, since each run of the function is independent of the rest. This is a common parallel computing application and the example I'll use in the tutorial.

If your problem is inherently serial, you can try to re-write it so that it can be run in parallel. Once you find a way to break up the problem

into independent parts, you're in business.

Getting a SciNet account

You need a SciNet account to use Niagara. If you don't have one, the first step is to make sure your advisor has a Compute Canada account. Compute Canada (CC) is the national organization that facilitates the building and operation of HPC systems. SciNet works in partnership with CC, and you apply for a SciNet account through the CC online database.

If your advisor doesn't have a CC account, kindly ask them to take ten minutes and go to the CC website's Research Portal. They should click on the cryptically-labelled: "Apply for an account".* This will take them to a set of instructions and a link to the login page. Your advisor will have to fill out a short application which says they are faculty at a Canadian university. Processing takes two business days, after which they'll receive an email with a CCRI number (Compute Canada Role Identifier). This is the number they'll give to graduate students they wish to 'sponsor' for CC accounts.

You can now apply for your own account using your advisor's CCRI. Go to the same login page and input the number along with personal details. Your advisor will then receive an email asking if they really want to sponsor you. Try not to upset them beforehand.

Once you have a CC account, you can apply for a SciNet account (this may be sounding tedious but I swear it's easy). All you do is login to the CC database and under "My Account", select "Apply for a consortium account". This will bring up a nationwide list of consortiums. Choose SciNet from this list and follow the instructions.

The command-line interface

To work with any HPC cluster, you need some basic familiarity with command-line interfaces. All communication with Niagara is done through a secure-shell connection from your home machine. If you have no experience with shell commands, I highly recommend checking out the SciNet website and going through the first two lectures of their Intro to Biostatistics course. It's worth doing assignments one and two. The info in those two lectures is pretty much all you need in order to navigate through Niagara.

Overview of Niagara

You now have a SciNet account. So what does that get you, and how do you access it?

When you first sign into Niagara you'll be taken to the home directory of a login node. These are separate from the job-running compute nodes. You use login nodes to organize your files and job scripts. Once you submit a job, your instructions will be transferred to the compute nodes and executed.

For most of you, a SciNet account will grant access to two login-node directories: home and scratch. You can keep important files in $HOME, but you submit jobs from $SCRATCH. A key difference between the two is that any files you save in the home directory are regularly backed up, whereas the files you move to scatch are automatically deleted after two months of inactivity. The other difference is memory allotment. You can store up to 100 gigabytes worth of files in the home directory; in scratch you can store 25 terabytes at a time -- which really comes in handy if your thesis involves sequencing the genome of every species known to science.

Researchers who rely heavily on Niagara can apply for greater quotas of time and memory via the annual Resource Allocation Competition (RAC), which is similar to an NSERC application but for PIs only. I suspect most of these go to physicists doing climate science or hard-core simulations on the structure of the universe, but there are probably some genomicists in there too. If your advisor has a RAC allocation, you'll

*https://www.computecanada.ca/research-portal/account-management/apply-for-an-account/

have access to a third directory called 'project', which will be unique to your group. Project directories are backed up like the home directory but allow for greater storage. Under a RAC, you can also submit more jobs (max 1000, versus 200) that use more nodes (max 1000, versus 20) in a given time than regular users. Regardless of your account type, the maximum run time for a single job is 24 hours.

To login to Niagara via a secure shell connection, use the following command from your terminal:

```
ssh -Y username@niagara.scinet.utoronto.ca
```

Enter your password when prompted.

Detailed information on how to navigate through Niagara is available in the SciNet lectures from the courses page of their website and in the SciNet wiki page, but an important point to note here is that you should use the pre-set environment variables HOME, SCRATCH, and PROJECT to refer to domains — this will take you and your scripts to the right place even if these folders are moved in the future.

For example, to navigate from the home to the scratch directory:

```
cd $SCRATCH
```

Testing your scripts

You won't be testing much code on the login nodes, though Niagara does allow limited testing there. You can test your scripts instead by submitting shorter test jobs (~15 minutes) to the regular work nodes or by requesting time on a debugging node, which is my preferred choice. You can request up to four debugging nodes at a time for up to an hour. A key advantage is that you can run your code interactively, which is extremely helpful for finding bugs. You can also run whole scripts here to ensure your job is runs over the corect number of nodes and that all 40 cores are running at or close to 100%. You request a debugging node with the following command:

```
debugjob X --time=HH:MM:SS
```

Where X is the number of nodes you want.

Testing your code for efficiency is pretty important with Niagara. SciNet tracks your group's usage in units of node-hours, and this value helps calculate your priority in the job queue – the more node-hours you've used recently, the lower your place in line. It's important to know your scripts don't waste this time or inefficiently tie up a shared resource.

Getting your files to Niagara

If your script involves file input, you may have to transfer files from your home machine to your Niagara directories. There are a few ways to do this. If the files are less than 10 gigabytes you can copy them from your home machine to any login-node directory using the following secure copy push command:

```
scp PATH/filename username@niagara.scinet.utoronto
```

You'll be prompted to give your scinet password. **Note that you'll have to give an explicit destination PATH** — environment variables like $SCRATCH or $PROJECT won't work (the files will be copied to the home directory). To copy multiple files just add additional filenames after the first argument.

You can also copy whole directories by adding the recursive "-r" flag to the scp command:

```
scp -r PATH/directory username@niagara.scinet.utoro
```

An alternative approach is to use rsync instead of scp. This will sync a directory on your machine with a new one on the login node and can be more efficient than scp during

subsequent syncing events (future syncs will only copy the file pottions that have changed). Here again, an explicit PATH is required:

```
rsync –a DIRECTORY/ username@niagara.scinet.utoronto.ca:PATH/NEWDIRECTORY
```

Be sure to add the new direcory name, otherwise the files will just be dumped into the general PATH/ environment.

Lastly, a very simple way to send files from your machine to Niagra is to use Filezilla. This program has a straightforward GUI interface, and once you connect it to Niagara, copying files to the directory of your choice is a simple double-click.

When moving data in excess of 10 gigabytes, you'll have to conduct either an scp or rsync pull command directly from a Niagara datamover node. This means that you'll have to make your machine accessible to remote logins. From the Niagara login node:

```
ssh nia–datamover1
scp –r USER@YOURIP:~/PATH/dirname PATH/directory
```

Where User is your username on your home machine.

.ca:PATH/

An Example Problem

In the following practice examples I want to use a toy function that conducts a very simple operation but that also introduces some messiness you might encounter. So as to not alienate half the department, I want to avoid using a problem that's exclusively in the realm of either ecology or evolution. Thus, I've created an imaginary Frankenproblem* with some biology terms thrown in. If I've succeeded, it will be equally meaningless to everyone but will illustrate the important points.

nto.ca:PATH/

In this imaginary problem, we have a dataset containing measurements for two properties from some sample population. These measurements are stored in a data frame in two columns denoted Prop1 and Prop2. We know that the first property is related to the second via three parameters through the simple and totally-made-up expression:

Prop2=(param1+param2)xProp1/param3

We'll imagine that parameter 1 is an environmental trait, parameter 2 is a genetic trait, and parameter 3 is some scaling constant. We want to know the values of these parameters in our population.

So, we have a dataset, and we want to extract important parameter values from it. Our strategy is to plug a bunch of possible parameter combinations into the known expression and see which ones give Prop2 estimates that approximate our Prop2 measurements (you would then test this formally using some downstream criterion like maximum likelihood – but in this example we're just generating the Prop2s that would result from our Prop1s given different parameter combos).

As you can see, this problem is perfectly parallel – the expression can be run on each combination of parameters independently of all others. It's therefore straightforward to spread the work across multiple cores.

We'll be using a grid approach, meaning we'll define a set of values for each of the three parameters, mix them up into all possible 3-parameter combinations, then plug each combination into the expression. This can be a useful exploratory approach when trying to determine areas of parameter space that warrant further, more fine-grained searches. You can also use a grid to establish start values for search algorithms to employ. Running through each row in a grid can be computationally intensive — but we've got an $18 million supercomputer at our disposal.

DISCLAIMER: running through a tonne of very quick calculations is exactly the type of problem that benefits *least* from parallelization.

* Should you feel the need to write in and say "*Actually*, that would be a Frankenstein's *Monster* problem", just know that you are dead to me.

The real benefits of the parallel approach (and the efficient use of Niagara) kicks in when a single run of your function takes a while and you have many runs to complete.

Creating the R Code

All jobs on Niagara require a minimum of two scripts: a worker script containing functions and variables, and a submission script that tells Niagara's job allocation software which resources you want and how it should use them. This section deals with creating worker code to go in an Rscript. Later we'll see how to marry the Rscript and job submission script to achieve a succesful run.

Data and Parameters

For our problem, we'll first create a dummy dataset by drawing 200 'measurements' from different normal distributions. The mean of our imaginary Prop1 is six and that of Prop 2 is 20.

```
dataset <- data.frame(rnorm(200, 6, 1),
rnorm(200, 0.8, 3))
colnames(dataset)=c("Prop1", "Prop2")
```

Next we'll define our parameters and construct the grid of parameter combinations. We know from imaginary theory that the first two parameter value are positive, but we don't know their order of magnitude. We will therefore let them assume values between 0-1, 1-10, and 10-100. We also know that the scaling constant lies somewhere between 1 and 100, so we'll let it take all integers in that range. Thus:

```
enviro=c(seq(0.1, 0.9, 0.1), seq(1, 9, 1),
seq(10, 100, 2))
genetic=enviro
scaling=seq(1, 100, 1)
parcombo <- as.matrix(expand.grid(enviro,
genetic, scaling))
colnames(parcombo)=c("enviro", "genetic",
"scaling")
```

'Parcombo' is now a three-coumn matrix in which each row is a vector containing a unique combination of our three parameters.

Finally we can define our function. It needs to accept a datum (a measurement for Prop1) plus a combination of three parameters in order to calculate a Prop2 estimate. Ideally we can give it a whole vector of Prop1 data and have it return a whole vector of Prop2 calculations for a given parameter set. Our function will take two arguments: one a data vector containing our Prop1 measurements, and the other a parameter vector in which the first element is a value for the environmental parameter, the second a value for the genetic parameter, and the third a value for the scaling constant. This function returns a vector with one Prop2 estimate for each Prop1 measurement.

```
ecoevo_func <- function(params, data) {
  enviro=params[1]
  genetic=params[2]
  scale=params[3]
  Prop2Est <- (enviro+genetic)*data/scale
  return(Prop2Est)
}
```

Now that we have some data, a parameter grid, and a function, we can look at different approaches to running the problem. It can be helpful to understand some parallel procedures as extensions of their serial counterparts, so we'll start with a quick look at the serial options.

Running the problem in serial

Each row of our parameter matrix is a three-element vector formatted to match the 'param' argument in ecoevo_func. Our challenge is to find a nice way to work through the matrix and sequentially select each row as input. The three main ways to do this are 1) for loops, 2) the 'apply' family of functions, and 3) the 'foreach' package. Code for each is displayed in the next panel.

```
# using a for loop
res_loop <- as.list(rep(0, nrow(parcombo)))
for(i in 1:nrow(parcombo)) {
    res_loop[[i]] <- ecoevo_func(params=parcombo[i,], data=dataset$Prop1)
}

# using the 'apply' function
res_apply <- apply(X=parcombo, MARGIN=1, FUN=ecoevo_func, data=dataset$Prop1)

# using the 'foreach' package
res_FE <- foreach(i=1:nrow(parcombo)) %do% ecoevo_func(params=parcombo[i,], data=dataset$Prop1)
```

With the for loop we have to first designate an object, in this case a list, to be filled with the output of each loop. Setting this list to the known length of the output and filling it at first with arbitrary values (here I used the repeated number zero) sets aside memory in R and makes the loop proceed *much* faster. In the 'apply' regime, the argument MARGIN=1 tells R to apply each row of the parameter grid as input to ecoevo_func (using MARGIN=2 would apply the columns instead and result in an error). The output of 'apply' is a matrix instead of a list, and each column contains all Prop2 estimates from one parameter combination. The output of foreach is identical to that of the for loop (foreach returns a list by default, but this can be changed by experimenting with its '.combine' argument). In each case we give the function only one column from the dataset as input, which we denote using the $ operator.

The performance of each routine varies greatly. On my laptop, the for loop took 7.6 seconds, the apply function took 4.7 seconds, and foreach took a whopping five minutes. The abysmal performance of foreach reflects a mismatch between problem and method. Our function performs a trivial calculation that R carries out extremely fast. The foreach package does a lot of background work that becomes useful in parallel runs, but this dominates the run-time when the core calculation is so simple (in fact, the overhead lag means that with our toy function, all parallel options will be slower than the standard for loop or apply family of functions implemented in serial). But with meatier functions like the ones you're likely to use in real analyses, this handicap in overhead time loss is much reduced.

Foreach was designed for parallel computing. You can think of it as a for loop in which different loops are run concurrently on different cores (for loops, *per se*, cannot be run in parallel; the parallel version of the for loop is the parallel version of foreach). The parallel framework for foreach is extremely straightforward and when used to repeatedly run complex functions it performs well. It also allows for easy implementation of nested loops and can perform some handy back-end cleanup operations.

Parallel runs on one multi-core machine

We'll shift to the world of parallel computing by running our problem across multiple cores on a single machine. An advantage here is that you can make use of shared memory, which is memory that can be accessed simultaneously by each core.

The quick and easy way to do this is through a parallel routine known as 'forking'. With forking, the entire R environment of the 'master' process is copied to each additional core, so there's no need to waste time passing information between processors. Both the foreach package and apply family of functions offer convenient forking options for parallel runs. Forking is not available on windows, so if you're a windows user check out the socket cluster options in the next section. The package

'DoParallel' contains every function you need for the following few examples.

To adapt a foreach loop for parallel runs you first register a parallel 'back-end', then you replace the %do% operator with %dopar%. To register the back-end when forking you simply define the number of cores you want to use within the function registerDoParallel. You can find the number of cores available with the detectCores function. On Niagara you will always use the maximum number.

```
# library(doParallel)
# register parallel backend
ncor=detectCores()
registerDoParallel(cores=ncor)
# run your function in loops across all cores
result=foreach(i=1:nrow(parcombo)) %dopar% ecoevo_func(params=parcombo[i,], data=dataset$Prop1)
```

If your 'dopar' code fails, switch back to the %do% operator for debugging.

The parallel version of the apply function for our problem is parApply (almost all apply functions have a parallel version). Using this function requires you to first create a cluster, denoted 'cl' by convention. It's considered good practice to close this cluster after the function completes to save resources.

```
# library(doParallel)
# make a cluster object with forking
ncor = detectCores()
cl <- makeForkCluster(nnodes=ncor)
# run your function in parallel with parApply
result <- parApply(cl=cl, X=parcombo, MARGIN=1, FUN=ecoevo_func, data=dataset$Prop1)
stopCluster(cl)
```

An even simpler way to use forking with the apply family is through 'mclapply' -- the multicore version of lapply. With this function, you don't have to create or register a cluster; you simply designate the number of cores to use as an argument in the function. In our case, we can't use mclapply right away, since lapply takes a list and our parameters are stored in a matrix. However, it's easy to transform our matrix to a list such that each element contains a vector of unique parameter combinations.

```
# convert parameter matrix 'parcombo' to list
param_list <- as.list(as.data.frame(t(parcombo)))

# use each element in list as input for ecoevo_fu
res_listpar <- mclapply(X=param_list, FUN=ecoevo_
mc.cores=nc)
```

Parallel runs across multiple machines

Parallel computing on a single machine is useful, but the true advantage of a cluster like Niagara is the ability to run jobs across multiple nodes to make use of many cores. With multiple machines we can no longer rely on forking or shared memory. We'll instead have to use a networking protocol to explicitly send information between nodes and their cores. A simple method for this is the socket cluster.

Making a socket cluster is similar to making a fork cluster when using parApply. The key difference is that workers (i.e. cores that will run our program, with the lead processor known as the 'master' and additional cores known as 'slaves') are identified by the name of their host

machine. So if we have one node called ‘nia1492’, and we want to run a process across two of its cores, we would write:

```
cl=makePSOCKcluster(c(“nia1492”, “nia1492”))
```

You won't know the names of the nodes while you're writing your script, but the folks at SciNet provided code on their R wikipage to automate this step, shown below in the first code block.

```
nc and run in parallel
func, data=dataset$Prop1,
```

Both parApply and foreach work with socket clusters. The workflow here has four or five steps, depending on which function you use: 1) gather the names of each node in your allocation, 2) construct your socket cluster using these names, 3) register the cluster (only required when using foreach), 4) send the variables your function requires to each core, 5) run your function. Additional steps like saving output to files will be job-specific:

```
library(doParallel)

#### Here you load or define data, define parameter grid, and define the function ####

# Get the names of the nodes in your allocation (code cribbed from the SciNet wiki)
nodelist <- Sys.getenv("SLURM_JOB_NODELIST")
node_ids <- unlist(strsplit(nodelist,split="[^a-z0-9-]"))[-1]
if (length(node_ids)>0) {
  expanded_ids <- lapply(node_ids, function (id) {
    ranges <- as.numeric(
      unlist(strsplit(id, split="[-]"))
    )
    if (length(ranges)>1) seq(ranges[1], ranges[2], by=1) else ranges
  })

  nodelist <- sprintf("nia%04d", unlist(expanded_ids))
}

# create socket cluster
cl <- makePSOCKcluster(rep(nodelist, 40))

# isolate the appropriate data vector
Prop1=dataset$Prop1

## if using foreach, register the cluster
registerDoParallel(cl=cl)

# Export your parameter matrix and data vector to each core in the cluster
clusterExport(cl=cl, c("Prop1", "parcombo"))

# Run function using parApply
res_parApp <- parApply(cl=cl, X=parcombo, MARGIN=1, FUN=ecoevo_func, data=Prop1)

## OR run function using foreach
result <- foreach(i=1:nrow(parcombo), .packages="foreach") %dopar%
ecoevo_func(params=parcombo[i,], data=Prop1)

# Save result to file with informative name
save(res_parApp, file="test_res_parApply.Rdata")

stopCluster(cl)
```

If that were an actual Rscript, you would load your data if necessary, define the parameter grid, and define the function all above the first block of code (and obviously run only one of parApply or foreach). You'll note that we've added a step in which we isolate just the Prop1 vector we need from the dataset. This is because we have to export variables to each core and it's inefficient to send more than is necessary. We've also saved the result in the binary .Rdata format, which is smaller and more efficient than a csv or txt file.

Bored of this tutorial yet? Here's a picture I took of a caiman. Look at those teeth! And as a reward for sticking it out there's a comic strip after this. I suppose you could just flip to that page now, but then you'd know less about running code on an expensive machine with a cool name. Speaking of which, isn't "supercomputer" a bit much? I mean it's not like this thing wears a cape and leaps between tall buildings, amirite? Cause if you ask me I'd h

The Job Submission Script

Assume we've taken some code from this tutorial and tidied it up into an R script. Now we want to launch it. To do this we'll need to write and submit a job allocation script which informs SLURM – Niagara's job allocation software – that we want a certain number of nodes for a certain time, then loads an R module, launches our script, and, if necessary, passes arguments to it.

Two R modules are currently installed on all Niagara servers: "r/3.5.0" (which requires intel/2018.2) and "r/3.4.3-anaconda-5.1.0". I've only been able to run socket cluster scripts using the latter, so that's what's shown below. SciNet's R wiki instructs users to make a wrapper script for launching R on multiple nodes, but I've found this to be unnecessary using r/3.4.3 and it hasn't fixed my problems with r/3.5.0 – so if you can get by with the older version of R, you can safely ignore the wrapper.

The next panel contains the job script, "testjob.sh", we'd use to launch the R script, "test.R", on 3 nodes for 2 hours.

```
#!/bin/bash
#SBATCH --nodes=3
#SBATCH --time=02:00:00
#SBATCH --job-name R_cluster_test

# re-route to the submission directory
cd $SLURM_SUBMIT_DIR

# load an R module
ml r/3.4.3-anaconda5.1.0

# launch your R script
Rscript --no-restore test.R
```

You'd then submit this job from the login node with the command:

```
sbatch testjob.sh
```

A note on MPI

In the HPC world, the standard method for transferring information between nodes is the Message Passing Interface (MPI). Most HPC applications use it. The package 'Rmpi' contains functions for spawning worker processes and sending and retrieving data between cores on different nodes using MPI. This type of programming is slightly more involved, but only

slightly, and if you're doing a lot of Niagara work it's worth investigating as it may be more efficient.

A second package, 'doMPI', simplifies things by allowing you to start an MPI cluster and register it as a back-end to foreach. Writing R scripts with this framework is easy (it's one function off from the code on the previous page), but instead of launching them with the 'Rscript' shell command you use 'mpirun'. At the time of writing, I haven't found the right arguments to this command to achieve a successful run with doMPI. If I figure it out, I'll add a note on the website or a brief note in the next issue. If you have a regime that works, please let us know in an email.

Multiple serial runs and GNU parallel

A fundamentally different approach to parallel computing on Niagara is through the GNU parallel program. I haven't used it in my work, but Stephanie Penk has and she was kind enough to explain it to me and share her code. The key difference is that the division of labour doesn't take place in the worker script, which is stripped down to contain little more than the core functions. Instead, a single set of parameters (or some other defining feature of the independent parts of a problem) is fed through the bash command line as input for worker code on a single core, and different copies are executed independently on each core of a node. So instead of having one job run across 40 cores, you have 40 jobs that each do one run on their own core.

You can write this approach manually in a job submission script, but GNU parallel automates this writing and provides a 'load-balancing' framework – you can give it many more than 40 jobs and it will assign them to cores in your node allocation such that each core is always busy and finished runs are automatically replaced with the next job in your list.

This "multiple serial" approach can be powerful with the right automation. Stephanie uses a Latin hypercube (not the club drug) to randomly sample parameter space and feed unique parameter combinations to R scripts, which then convert them into command-line arguments unique to each job, which then get compiled into a joblist that gnu parallel runs through.

Unlike some of the options mentioned previously, there's good online information about multiple serial regimes for clusters*, so I'll be brief and leave a short block of code to illustrate the routine. Sticking with our problem, the R script would look this this:

```
#### Worker Code for use in each job ####
### Above this line you load data if necessary and define the function ####

# retrieve parameters for individual run from bash command line
par_vec = as.numeric(commandArgs(trailingOnly=TRUE))

# run the function
result = ecoevo_func(params=par_vec, data=dataset$Prop1)

# Give result a unique name so you don't end up with lots of objects you can't load together
assign(paste("par",par_vec[1],par_vec[2],par_vec[3],"result",sep="_"), result)

# Save your result with a unique name so you'll be able to distinguish it from all the others
save(list=paste("par",par_vec[1],par_vec[2],par_vec[3],"result",sep="_"),file=paste(par_vec[1
], par_vec[2], par_vec[3], "result.Rdata", sep="_"))
```

* *https://docs.scinet.utoronto.ca/index.php/Running_Serial_Jobs_on_Niagara*

If this script was called "serial.R", then the bash command that would launch it is launch it is:

```
Rscript serial.R 0.8 0.2 0.3
```

Where the arguments following the script name are parameters to be passed to the script. A whole series of these commands can be stored in a joblist and run by GNU parallel. The job list is a list file in which each line is a "job" like the one above for Niagara to run.

Once you have a job list, you can start running independent serial jobs on different cores and even multiple nodes. The job submission script, given a job list named Jobs.lst, is shown below, courtesy of Stephanie Penk.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=01:00:00
#SBATCH --job-name R_serial_test

module load intel/2018.2 r/3.5.0 gnu-parallel/20180322
parallel --jobs 10 < Jobs.lst
```

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

To monitor progress:

There may be a better way to do this, but I monitor jobs by opening separate tabs in Terminal, logging into Niagara in each one, then ssh-ing into the work nodes my jobs are running on and using "htop". To find which nodes your job is running on, enter the following command and look under 'resources'.

```
squeue -u YOURUSERNAME
```

Once you've transferred over to a work node, using the "htop" command will give you a nice intuitive visual representation of the work being done across all cores. You want to see the bars filled up to near 100 percent mark.

Best practices

Always use all 40 cores, and try hyperthreading by using 80 when memory allows (there are 40 physical cores on each node, but each one is contains two virtual or "logical" cores). It's unclear to me how hyperthreading affects performance on Niagara, but I can't see it hurting unless the memory gets out of control.

Another important note is that it is extremely inefficient to load or save large numbers of small files during your computation. If you have to load files, load them all at the beginning before your operations kick in. If you have to save files, it's much better to save one or a few large files at the end than a bunch of small ones throughout.

Be memory aware

You have to be memory conscious when working with Niagara. Computing in parallel can cause rapid jumps in memory demand. For example, if your code involves manipulating a larger object — say a 2 gigabyte data file — and you construct a socket cluster and export this object to each core, you're now taking up 40x2 gigabytes = 80 gigabytes of memory. All nodes give you just under 200 gigabytes to work with, so keep this in mind or you could run out of memory and crash your job.

Closing remarks

Don’t be afraid of Niagara. You're not going to break it. I found it intimidating when I started out since I didn't know what I was doing and didn't want to screw up my lab's fair share allotment. But it's really not that bad. Just start with some smaller jobs until you get

comfortable with the environment, how to debug, and how to track progress. And definitely don't be afraid to get in touch with the SciNet team. They are there to help and and they want you using the machine. In my experience they are excellent.

Finally, when Niagara inevitably asks you if you want to play a game, don't choose Global Thermonuclear War. It can be a bit of a bummer. Happy supercomputing!

Sean Anderson is a PhD student in the Weir lab at UTSC studying character evolution in relation to speciation in birds. He has used Niagara primarily to simulate trait divergence between sister pairs across a latitudinal gradient under different models of evolution. He's a scorpio and enjoys long walks on the beach. He's also an editor at The EEB Quarterly.

Special thanks to Stephanie Penk and Julia Kreiner for code and advice, to the SciNet team for providing email assistance, and to Dr. Daniel Gruner, the Chief Technical Officer at SciNet, who graciously took the time to provide background information and review part of this tutorial. Any errors are entirely the author's and will be rectified online as soon as they are discovered.

Squirrel monkey. Photo by Sean Anderson