

OPENMPI WITH FORTRAN

A guide to writing and running parallel code

Contents

1	Obtaining and installing the necessary tools	2
1.1	Windows tools and installation	2
1.2	OS X tools and installation	3
2	Writing parallel code	4
2.1	A simple Hello World! example	4
2.2	An advanced Hello World! example	6
3	Running parallel code	9
3.1	Compiling and running code for Windows users	10
3.2	Compiling and running code for OS X users	20
3.3	Running the hello_world_advanced example code and its output	28
4	Additional command line statements, links, and guides	31

1 Obtaining and installing the necessary tools

This portion of the guide will focus on the required tools needed to run and write code on a parallel computer. There will be two subsections; a guide for Windows users, and a guide for OS X users. Linux users can follow the OS X guide. The following tools are recommended/required:

1. A text editor (semi-optional)
2. An X11 server client (semi-optional)
3. A command line interface (required)
4. An SFTP GUI client (optional)

An advanced text editor is great for writing code. You can get by on the standard editors that are bundled with Windows (Notepad) and OS X (TextEdit), but a more advanced editor will make your life much easier. An X11 server is required only if you wish to check the status of Kruskal via Ganglia. Ganglia is a local site that allows you to check the status of each individual node, and it is highly recommended you check the status of all the nodes before you send out your code to run. Some nodes may be down - others may be in use by someone else - you should check first to avoid headaches and cryptic command line errors. The command line interface is required for compiling and running the code, and an SFTP client will make moving your files from your computer to the parallel computer much easier.

1.1 Windows tools and installation

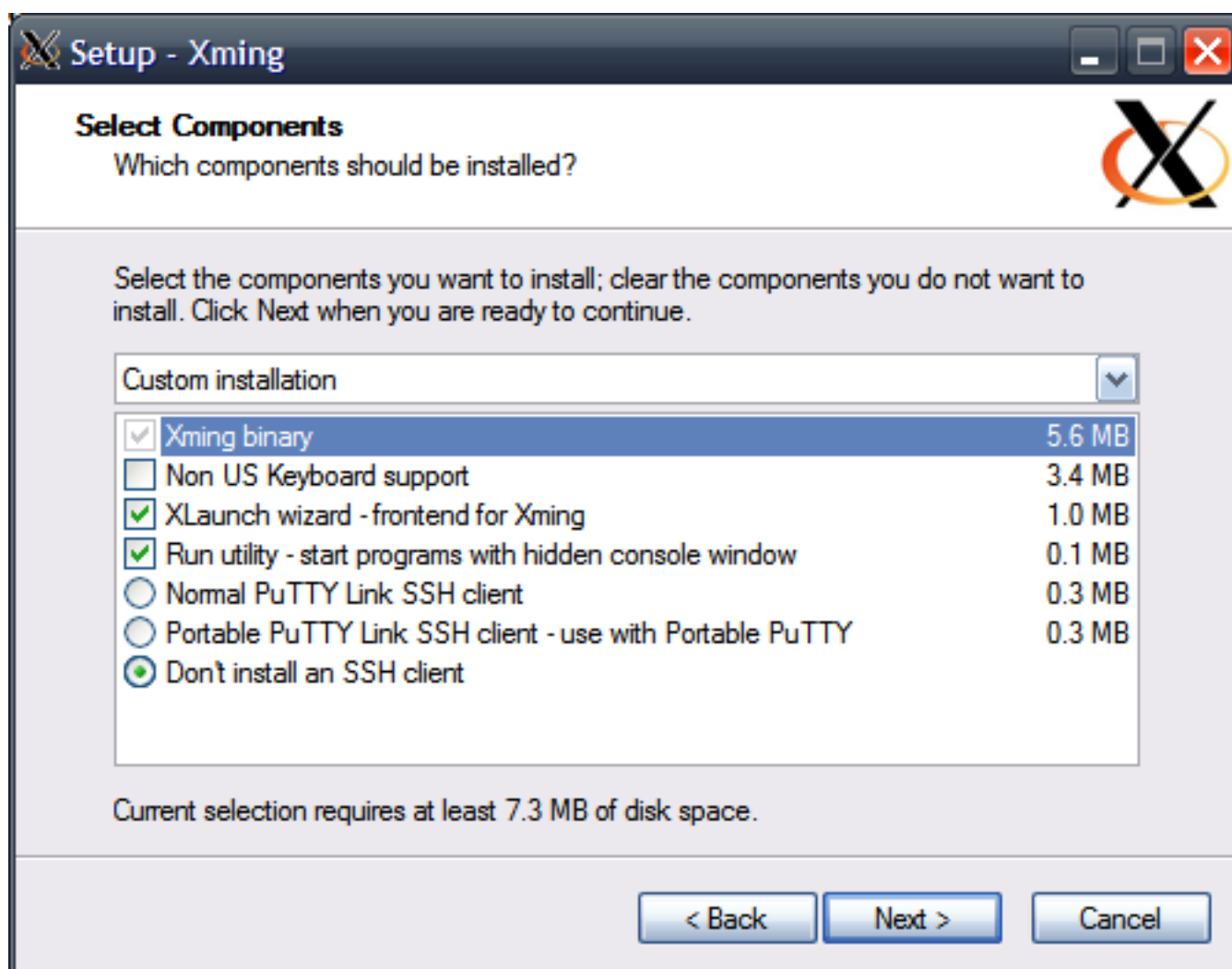
First thing is first. You'll want to make sure you have a capable text editor for writing OpenMPI code with Fortran. Several options are:

1. Sublime Text 2
2. Notepad++

There are plenty more, but these two seem to be the most popular as of late. Notepad++ does Fortran syntax highlighting natively, and as such, is recommended. You can install additional packages to get Sublime Text 2 to do Fortran syntax highlighting, but it doesn't work out of the box. Next up is getting an X11 client up and running. A minimal and easy to use X11 client will be Xming, which you can download here:

<http://sourceforge.net/projects/xming/>

Follow the installation instructions. When prompted, do a custom installation:



As for a command line interface, this guide will use PuTTY. Another option is Cygwin, but this guide will not be supporting it. To download PuTTY, head to:

<http://www.putty.org>

PuTTY is a portable application - no installation is required. Lastly, we need an SFTP client. This guide will use CyberDuck, but there are many options out there. A more feature rich client would be FileZilla. To download CyberDuck head on over to:

<https://cyberduck.io>

You now have all the tools!

1.2 OS X tools and installation

First thing is first. You'll want to make sure you have a capable text editor for writing OpenMPI code with Fortran. Several options are:

1. Sublime Text 2
2. Vim
3. Emacs
4. TextWrangler

Of these, TextWrangler is the only one that offers Fortran syntax highlighting natively and is recommended. You can install additional packages for Sublime Text 2 to allow it to do Fortran syntax highlighting, but it doesn't work out of the box. With a decent text editor loaded, the next step is to install an X11 client. On OS X, the only option is to download X11 from Xquartz located here:

<http://xquartz.macosforge.org/landing/>

Download and install this package. You'll need to log out of OS X for it to take effect. Once this is finished, you'll need to install an SFTP client. OS X comes with terminal, so you're covered for a command line interface.

The recommended SFTP client will be CyberDuck, which can be downloaded from here:

<https://cyberduck.io>

Note: CyberDuck is free if you download the install files directly from the website. Downloading it from the Mac App Store is not free.

You now have all the tools!

2 Writing parallel code

OpenMPI is an open source message passing interface (MPI), which allows several computers to run code simultaneously over a network. This section will have two examples; a simple Hello World! program, and a more advanced Hello World! program that has each node run a subroutine. All code will be written in Fortran 90. OpenMPI also works with C, however, this guide will not cover using OpenMPI with C. I assume you already know how to use Fortran.

It is important to remember that when using MPI variables are not shared across nodes unless explicitly coded to do so. Each child process (instance) of the code uses its own memory.

2.1 A simple Hello World! example

Let's dive in! Below is the code from the simple Hello World! program. I'll outline everything that is occurring and what everything does, as well as the outline the flow control.

```
program HELLOWORLD
use mpi
IMPLICIT NONE
```

```
integer :: ierr,num_procs,my_id
integer status(MPI_STATUS_SIZE)

CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_id, ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, num_procs, ierr)

if (my_id==0) then

    write(*,*) 'Hello world from the master processor!'

else

    write(*,*) 'Hello world from processor: ', my_id

end if

CALL MPI_FINALIZE(ierr)

stop

end Program HELLOWORLD
```

To begin, we define a new program called HELLOWORLD. Underneath that we have the line 'use mpi', a line which is required for writing parallel code. This line is for Fortran 90/95 - if you're using Fortran 77, you'll want to use:

```
include mpif.h
```

Next, we have our standard IMPLICIT NONE command which stops Fortran from defining variables that start with i,j,k,l,m, and n as integers. Don't forget this!

We define several integer variables next: `ierr`, `num_procs`, `my_id`, and `status(MPI_STATUS_SIZE)`. Next, we begin the MPI subroutine `CALL MPI_INIT(ierr)`, which initializes MPI. The variable `ierr` is an error variable, and is just one of those things necessary for MPI to function correctly. Next, every processor available gets assigned a specific rank, or ID, starting at 0 via the `CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_id, ierr)` statement. The variable `my_id` will contain the rank of each processor. Remember, this variable will have an instance on each processor.

Next, we have `CALL MPI_COMM_SIZE(MPI_COMM_WORLD, num_procs, ierr)`, which will determine the maximum amount of processors we have available to use, and assign it to the variable `num_procs`.

In general, every MPI program will contain those three `CALL` statements.

We start the Hello World logic with a simple if statement - we're checking to determine which

processor is considered the master, and which are considered the slave (or child) processes. At this point, the code is running in parallel - each processor is running an instance of this code. The processor that has the `my_id` instance that is equal to 0 will be our master, and as such, it will print a Hello World! message as the master. Any processor that has an instance of the `my_id` variable that is not equal to 0 (which will be all of them except the one with the ID of 0, duh) will then print a statement to the console that says Hello World from that specific processor ID.

Finally, we close out MPI using the `CALL MPI_FINALIZE(ierr)` statement, followed immediately by the statement `stop`. We end the program `HELLOWORLD`. Note we did not make use of the `num_procs` variable - we will make use of this in the more advanced example.

Output from this program will be shown in the section about running MPI code.

2.2 An advanced Hello World! example

```

program HELLOWORLD
use mpi
IMPLICIT NONE

integer :: ierr,num_procs,my_id,i,j
integer status(MPI_STATUS_SIZE)
real(8) :: random1, random2
real,dimension(2) :: mpi_values

CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_id, ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, num_procs, ierr)

OPEN(17, file="hello_world_random_numbers_output.dat", ACCESS = 'APPEND')

do i = 1,5

if (my_id == 0) then

write(*,*) 'Hello world from the master processor!'

      do j = 1,(num_procs-1)

          CALL MPI_RECV(mpi_values,2,MPI_REAL,MPI_ANY_SOURCE,i,MPI_COMM_WORLD,status,ierr)
          write(17,*) i,mpi_values(1),mpi_values(2)

      end do

else

      write(*,*) 'Hello world from processor: ', my_id

```

```

      CALL Hello(my_id,random1,random2)

      mpi_values(1) = random1
      mpi_values(2) = random2

      CALL MPI_SEND(mpi_values,2,MPI_REAL,0,i,MPI_COMM_WORLD,ierr)

end if

end do

CALL MPI_FINALIZE(ierr)

stop

end Program HELLOWORLD

!-----Subroutine

subroutine Hello(my_id, random1, random2)

real(8) :: random1, random2
integer :: idum, my_id

idum = -1-my_id

random1 = ran1(idum)
random2 = ran1(idum)

end subroutine Hello

!-----RNG Function

FUNCTION ran1(idum)
INTEGER idum,IA,IM,IQ,IR,NTAB,NDIV
REAL ran1,AM,EPS,RNMX
PARAMETER (IA=16807,IM=2147483647,AM=1./IM,IQ=127773,IR=2836,NTAB=32,NDIV=1+(IM-1)/NTAB,EPS=
INTEGER j,k,iv(NTAB),iy
SAVE iv,iy
DATA iv /NTAB*0/, iy /0/
if (idum.le.0.or.iy.eq.0) then
idum=max(-idum,1)
do j=NTAB+8,1,-1
k=idum/IQ
idum=IA*(idum-k*IQ)-IR*k

```



```

if (idum.lt.0) idum=idum+IM
if (j.le.NTAB) iv(j)=idum
enddo
iy=iv(1)
endif
k=idum/IQ
idum=IA*(idum-k*IQ)-IR*k
if (idum.lt.0) idum=idum+IM
j=1+iy/NDIV
iy=iv(j)
iv(j)=idum
ran1=min(AM*iy,RNMX)
return
END

```

To start, I'll describe what this program does. The master processor (with `my_id` equal to 0) will print to console a Hello World! statement, and then it will wait to receive a message from each processor that is not considered the master. This is accomplished via the `CALL MPI_RECV` statement on the master, and the `CALL MPI_SEND` statement on each of the child processes via a loop statement that increments from 1 to `num_procs-1` (recall that `num_procs` is our total number of processors available, including the master, hence the -1). We will loop through this process five times (as in, we will have all processors compute something and send it to the master processor a total of five times), as can be seen from the loop with counter variable `i`. The master processor, upon receiving the data, will save that data to a text file named `hello_world_random_numbers_output.dat` (note: `ACCESS='APPEND'` in the declaration statement towards the top of the code allows us to continuously write to this file without overwriting previous data, a must if you're writing data to file!).

Each child (slave) processor will run a Fortran subroutine named `Hello`, which shares the variables `my_id`, `random1`, and `random2`. These variables are declared in both the MPI portion of the program as well as the subroutine itself. Each child processor, before running the Fortran subroutine, will print a Hello World statement to the console.

When the `CALL Hello` statement is executed, it delegates flow to the Fortran subroutine located towards the bottom of the code. It will use each processors unique ID to create a negative seed value in the variable `idum`. This seed variable will be used to generate random, uniform numbers via the `rand1` function (bottom most code). Once this is complete, the child processor assigns these values to the array `mpi_values` (in index 1 and 2). This information is then sent from each child processor to the master processor via `CALL MPI_SEND`. I'll break down the syntax of the `CALL MPI_RECV` and `CALL MPI_SEND` statements next. The `CALL MPI_SEND` statement is as follows:

`MPI_SEND (data_to_send, send_count, send_type, destination_ID, tag, comm, ierr)`

where `data_to_send` is going to be whatever data you wish to send to the master processor. In our example, I sent an array of `real(8)` values of which has two index places (1 and 2). The `send_count` portion is how many pieces of data are being sent - in our case, I'm sending an array that has two

indices, and as such, we are essentially sending two points of data. The variable `send_type` is what type of data is being sent - we're sending an array of `real(8)` variables, so the type is `MPI_REAL`. This is pretty easily adjusted for other datatypes and will generally be what you expect it to be (for example, integers will use `MPI_INTEGER`, etc). The variable `destination_id` is going to be the processor which will receive the data (it will use the ID value that is in `my_id!`). We're sending this data to the master processor, so we use ID 0. The last variable that needs explanation is the tag variable. This variable assigns a integer value to the transmission - here we use the variable `i`, which is our counter variable for running through the entire process of sending out the data to all processors a total of five times. When this code is running in parallel there is no order in which it will complete, due to several factors (namely, variation between computation time and network latency), so it is common to see the results returned out of order (for example, hello world from processor 8 before a hello world from processor 3 - 8 just happened to get done before 3). When running simulations where we increment a specific parameter, it is good to finish running all child processes for that specific parameter value before we move onto the next one. The tag value will allow us to do that. The `MPI_SEND` statement, once called, is blocked from sending data again until the data gets received. If the master processor is waiting for a child processor to send data from something with a tag of 1, it will not receive anything from any other tag until all processes with tag 1 are received. The last two variables, `comm` and `ierr`, are just the communicator handle (just leave it as `MPI_COMM_WORLD`) and the error variable that MPI needs. Now we can look at the syntax of the `MPI_RECV` statement:

`MPI_RECV (received_data, receive_count, receive_type, sender_ID, tag, comm, status, ierr)`

The `MPI_RECV` statement should be fairly easy to understand now knowing what the `MPI_SEND` statement does. We're receiving the data `mpi_values`, an array of two indices (hence `receive_count` is set to 2). The variable `receive_type` is the type of data we're receiving (in this case, `real(8)` variables, hence `MPI_REAL`). Now we have the variable `sender_ID` - this is the ID of the processor that is sending the data. We use in our program `MPI_ANY_SOURCE`, which allows us to receive the data from any processor. Our tag is simply `i` (which prevents us from receiving data from separate iterations of the entire processes until the preceding process is finished (i.e., we won't receive data from tag 2 until all data from tag 1 is received). Our communicator handle, the `comm` variable, is `MPI_COMM_WORLD`. Now we have the status variable. We declared this at the top of our code - the integer `status(MPI_STATUS_SIZE)`. Lastly, we have our error variable (in our case, `ierr`).

I've included the actual code files with this guide, they will contain more information in the comments. This more advanced Hello World program will be a good template that can be used for running population simulations - simply write the simulation as a subroutine, and define which data from the subroutine you wish to send/receive to the master and compute/store. Now that we can write programs, we need to see how we run them on Kruskal.

3 Running parallel code

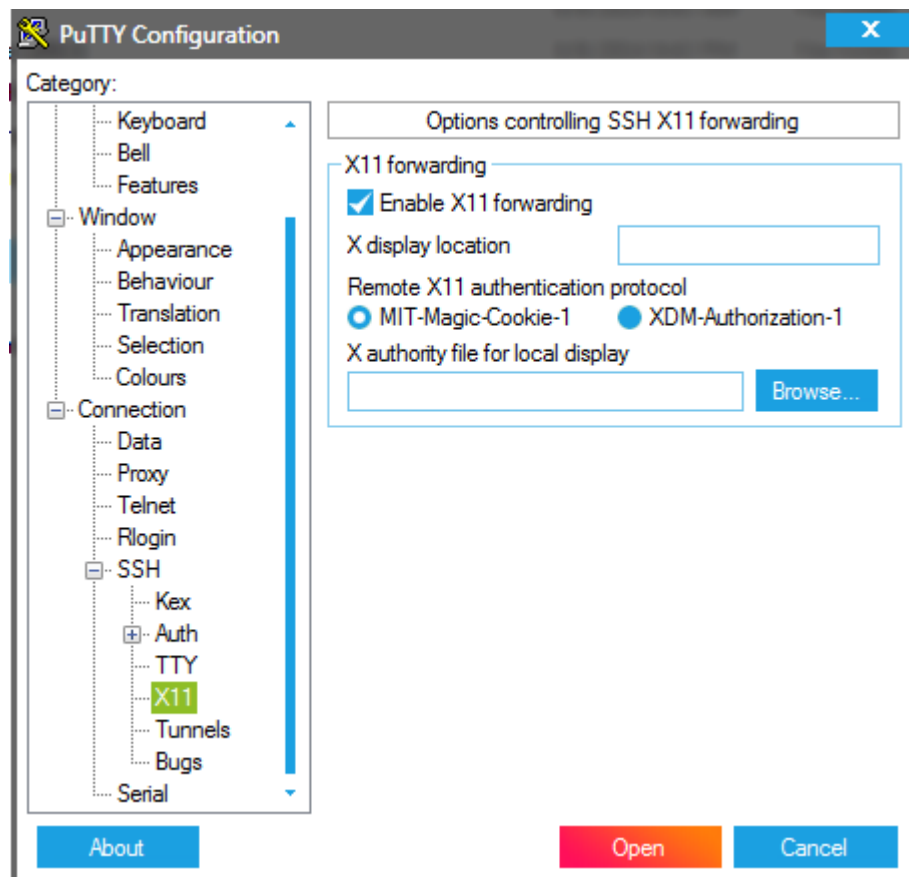
This section will be broken into three subsections; one for Windows users and one for OS X users on how to compile and run the code, and a final subsection which will describe the output from our two Hello World! examples once they've been ran.

3.1 Compiling and running code for Windows users

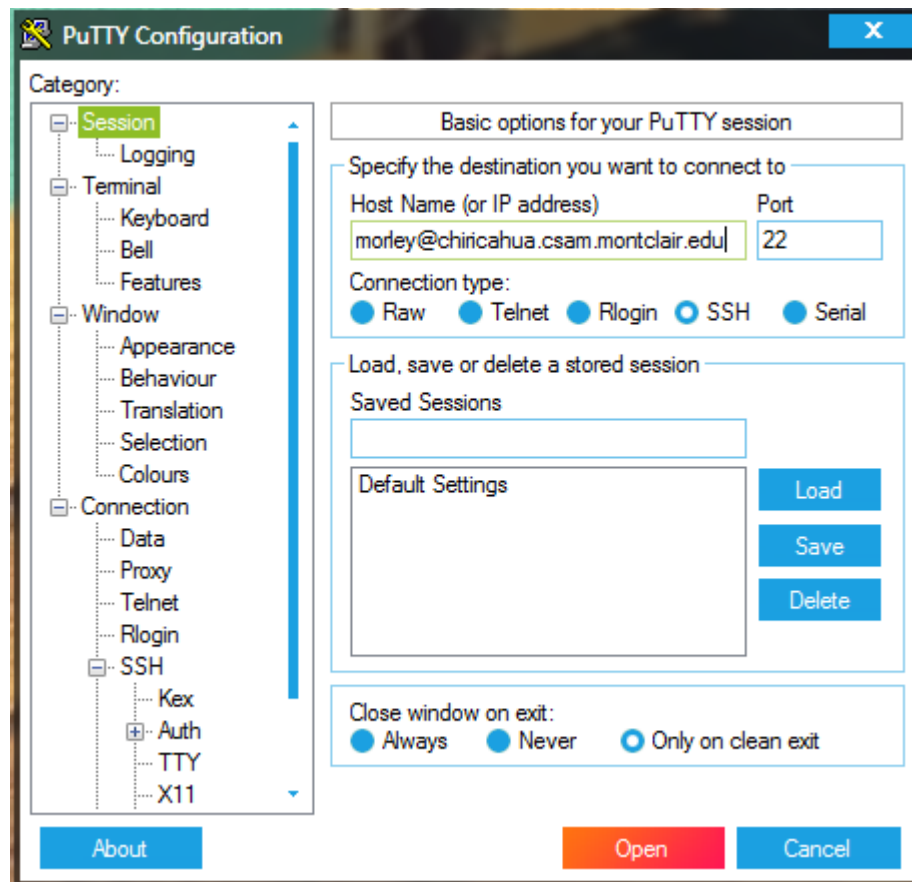
By now you should have Xming installed, a text editor installed (that you've used to write some parallel code!), and PuTTY downloaded with CyberDuck installed. Depending on where you are (on campus or at home), the process will be different. Kruskal is not a web-facing server, so if you're not on campus you cannot directly tunnel into it. You'll need to tunnel into a web-facing server first, and then tunnel into Kruskal. I'll go through the steps assuming you're at home first.

Okay, first thing we're going to do is get into Kruskal using PuTTY. My assumption here is you have a user account already setup on a web-facing server (Dr. Forgoston provided me a user account on his office server, which allowed me to tunnel in. You may have one from a different professor, but the steps will more or less be the same with a simple change of server name). Lets open PuTTY.exe and change some settings.

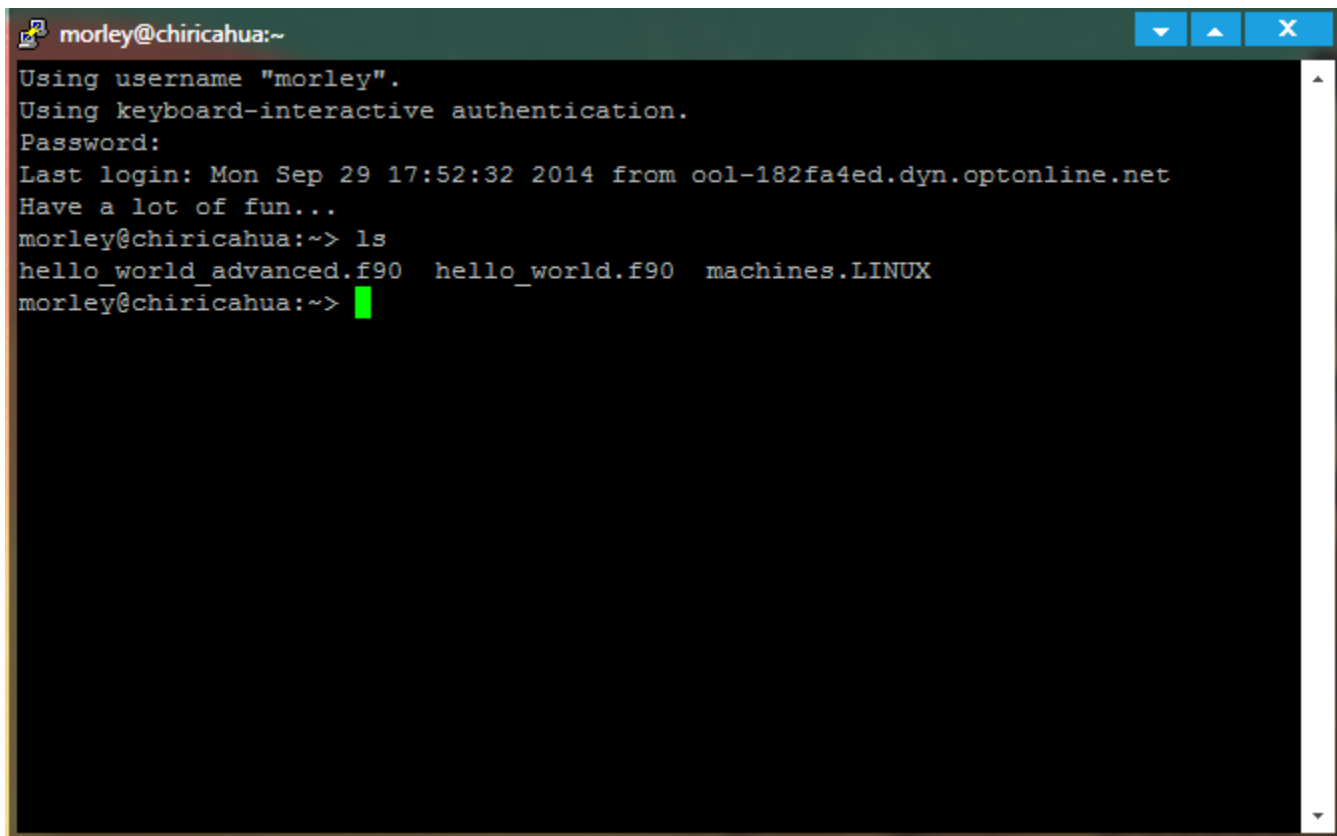
We're going to navigate to Connection→SSH→X11. Enable the option to Enable X11 Forwarding. Every time you tunnel in you'll need to do this.



Next, we're going to go to the Session category at the top of the list. We're going to enter the address of the web-facing server. Dr. Forgoston's server address is `chirichua.csam.montclair.edu`, and as such, my address will be `my username@chirichua.csam.montclair.edu`, which is `morley@chirichua.csam.montclair.edu`.

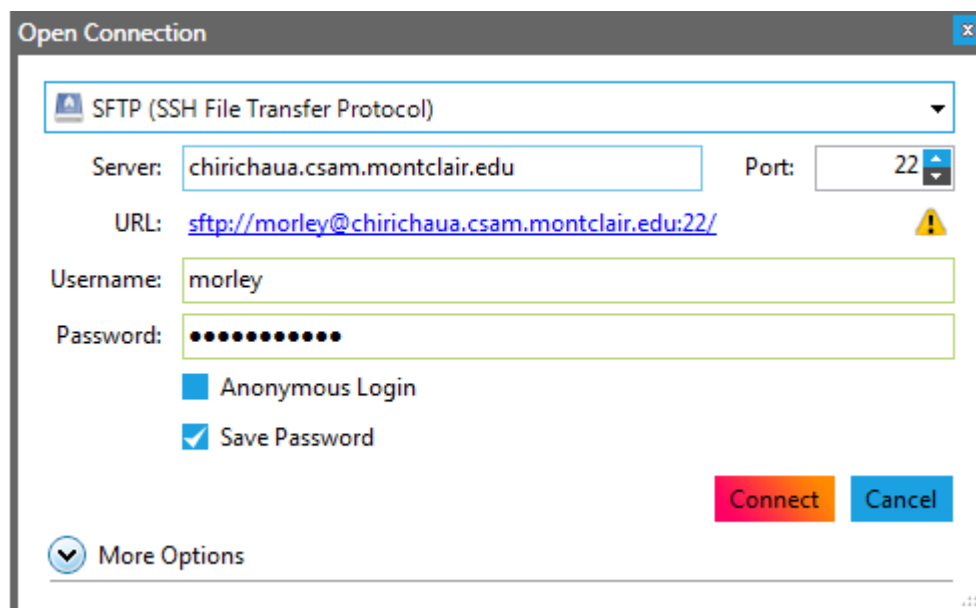


Ensure SSH is ticked, and hit the open button. You'll see a command prompt pop up, which will ask you for your password. Type your password and hit enter. You've now tunneled into the web-facing server!

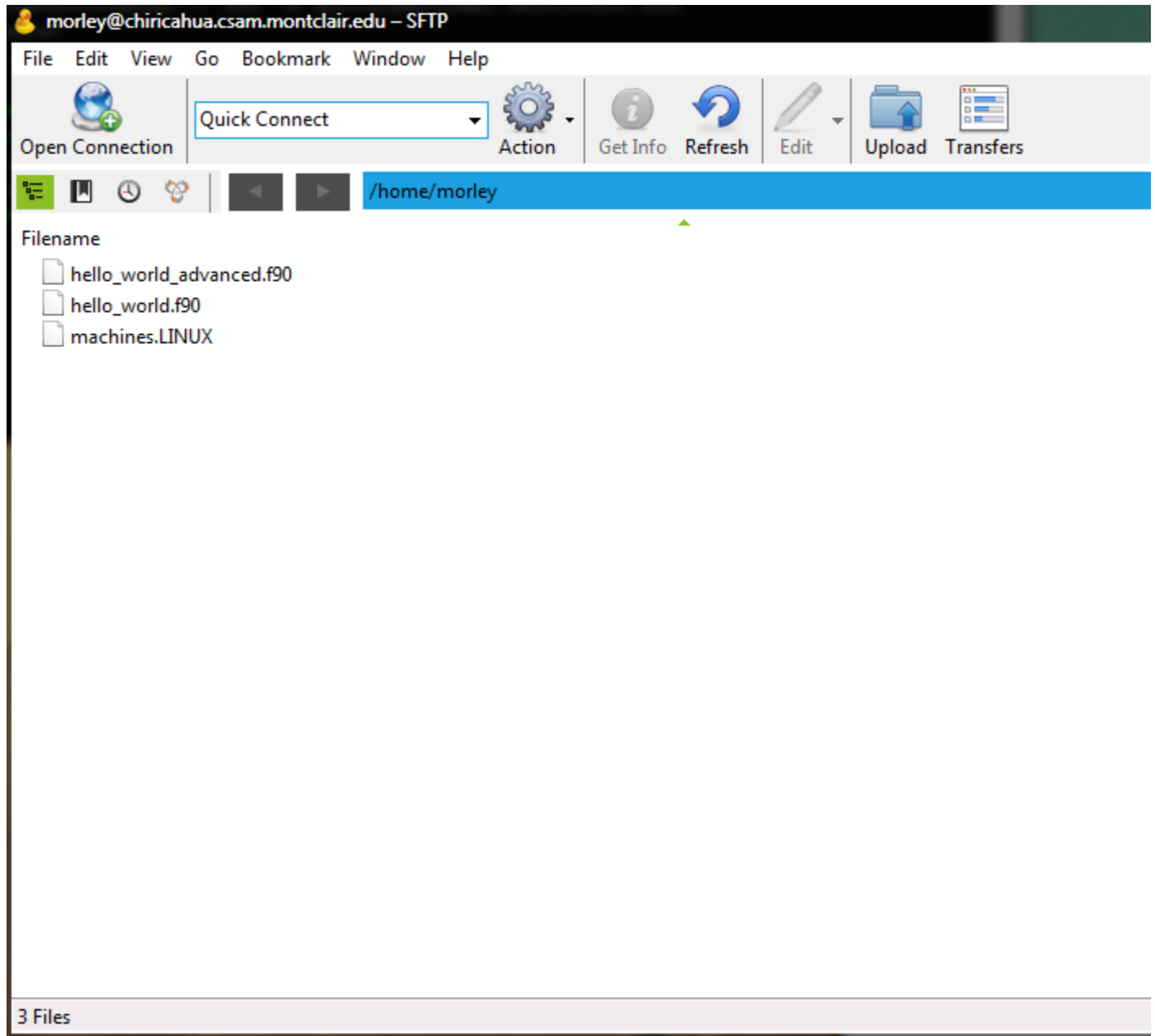
A terminal window titled 'morley@chiricahua:~' with standard window controls. It shows the output of an SSH login: 'Using username "morley".', 'Using keyboard-interactive authentication.', 'Password:', 'Last login: Mon Sep 29 17:52:32 2014 from ool-182fa4ed.dyn.optonline.net', and 'Have a lot of fun...'. The user then runs 'ls', which lists 'hello_world_advanced.f90', 'hello_world.f90', and 'machines.LINUX'. The prompt returns to 'morley@chiricahua:~>' with a green cursor.

```
morley@chiricahua:~  
Using username "morley".  
Using keyboard-interactive authentication.  
Password:  
Last login: Mon Sep 29 17:52:32 2014 from ool-182fa4ed.dyn.optonline.net  
Have a lot of fun...  
morley@chiricahua:~> ls  
hello_world_advanced.f90  hello_world.f90  machines.LINUX  
morley@chiricahua:~>
```

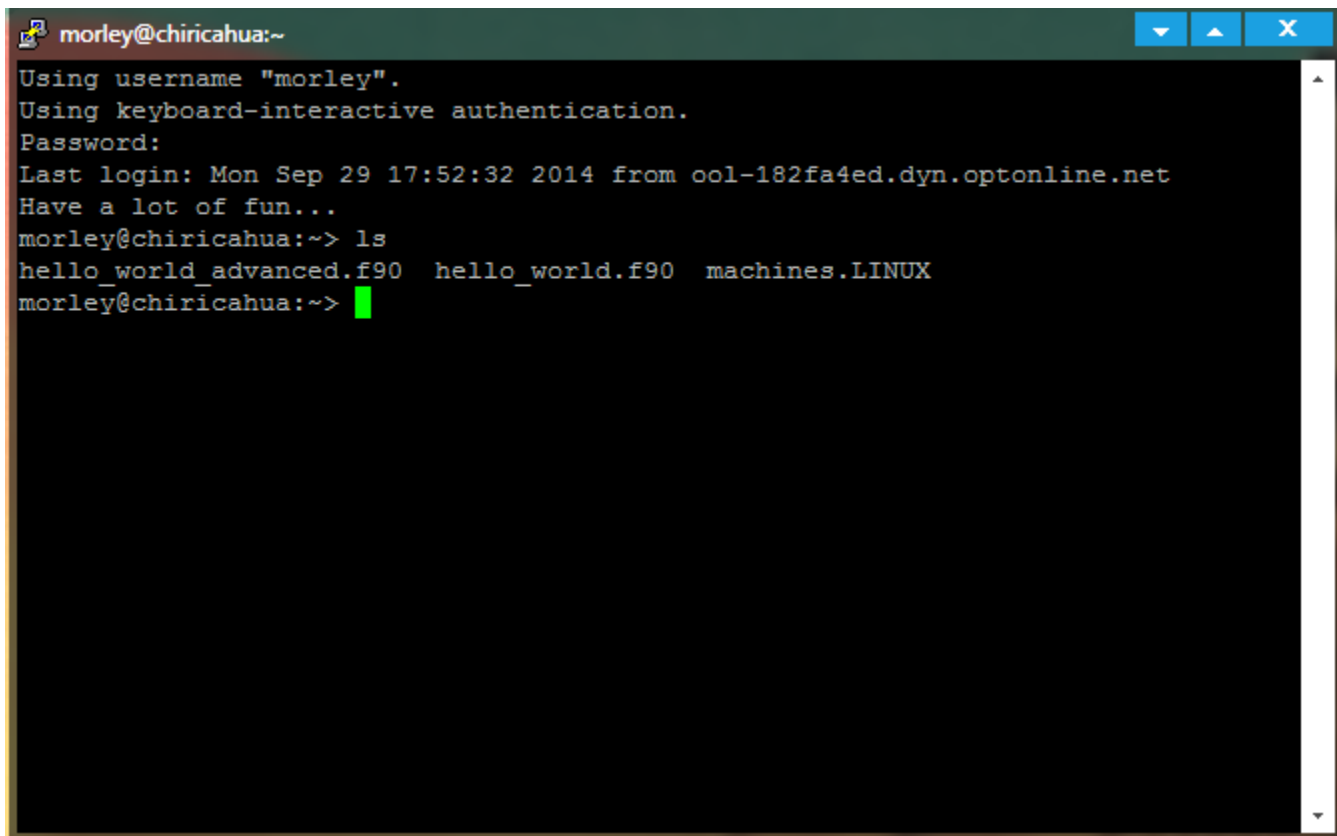
Seen in the above image, I do an `ls` command to get a list of all the files in my directory. I have the files `hello_world_advance.f90`, `hello_world.f90`, and a file called `machines.LINUX`. There are two ways to transfer these files from my computer to this server, I'll show the easy way first, and the more advanced way once we tunnel into Kruskal. Load up CyberDuck! You'll see a big button that says Open Connection. Go ahead and click that. A window will pop up, and you'll want to select SFTP from the drop down menu. Enter the server name and your username and password.



Once finished, hit connect. If this is your first time connecting, the following window will be empty as you have no files in your directory.

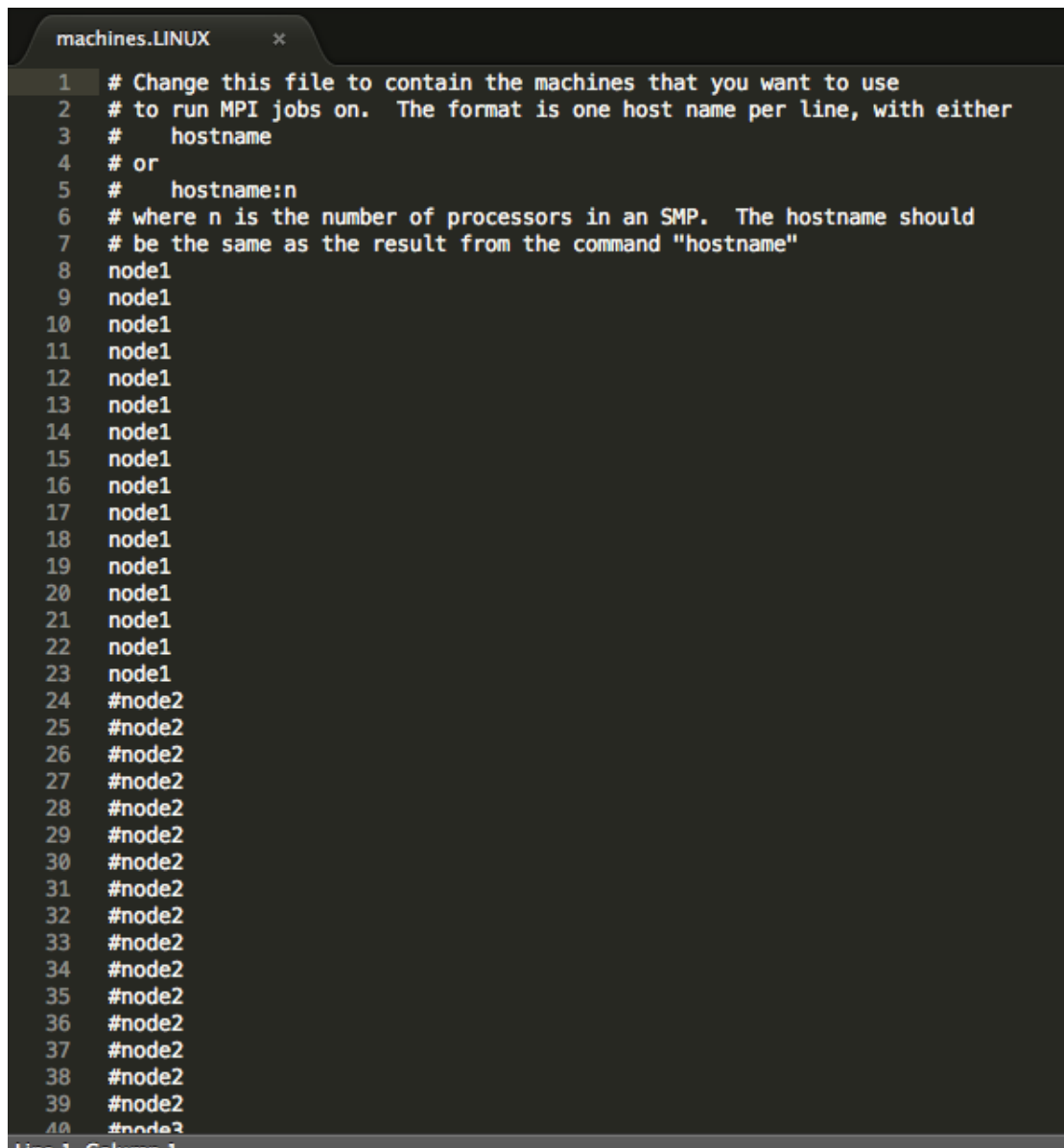


You can simply drag and drop files into this window (both to the window, and out of the window) to transfer files over to the web-facing server. You can verify that they have made it to the server by typing `ls` into the command prompt window, which will give you the listing of all the files in that directory.

A terminal window titled 'morley@chiricahua:~' with standard window controls. The terminal shows an SSH login process: 'Using username "morley".', 'Using keyboard-interactive authentication.', 'Password:', and a login message: 'Last login: Mon Sep 29 17:52:32 2014 from ool-182fa4ed.dyn.optonline.net'. After 'Have a lot of fun...', the user runs 'ls', which outputs 'hello_world_advanced.f90', 'hello_world.f90', and 'machines.LINUX'. The prompt returns to 'morley@chiricahua:~>' with a green cursor.

```
morley@chiricahua:~  
Using username "morley".  
Using keyboard-interactive authentication.  
Password:  
Last login: Mon Sep 29 17:52:32 2014 from ool-182fa4ed.dyn.optonline.net  
Have a lot of fun...  
morley@chiricahua:~> ls  
hello_world_advanced.f90  hello_world.f90  machines.LINUX  
morley@chiricahua:~>
```

By now, hopefully you're asking what `machines.LINUX` is. This file is necessary for running parallel code as it tells the compiler which nodes we're going to be using. Kruskal has 32 nodes, each with 16 processors (for a total of 512 available processors!). As stated previously in the guide, you'll want to check a site called Ganglia for the status of each node, and if a node is down or in use, you'll want to edit the `machines.LINUX` file and comment out that block of nodes to prevent code from being sent/ran on it. Here is what the `machines.LINUX` file looks like:



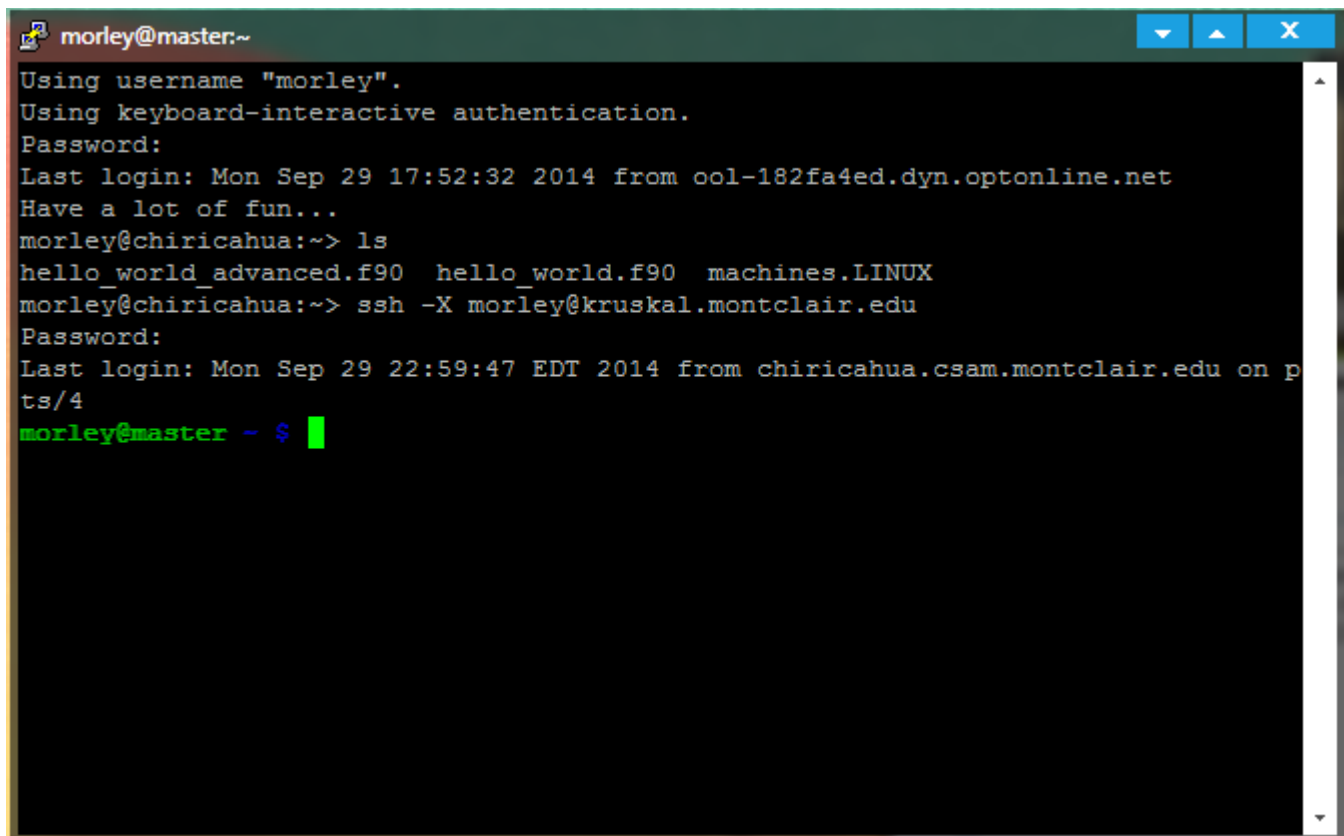
```
machines.LINUX x
1 # Change this file to contain the machines that you want to use
2 # to run MPI jobs on. The format is one host name per line, with either
3 # hostname
4 # or
5 # hostname:n
6 # where n is the number of processors in an SMP. The hostname should
7 # be the same as the result from the command "hostname"
8 node1
9 node1
10 node1
11 node1
12 node1
13 node1
14 node1
15 node1
16 node1
17 node1
18 node1
19 node1
20 node1
21 node1
22 node1
23 node1
24 #node2
25 #node2
26 #node2
27 #node2
28 #node2
29 #node2
30 #node2
31 #node2
32 #node2
33 #node2
34 #node2
35 #node2
36 #node2
37 #node2
38 #node2
39 #node2
40 #node3
```

Use the `#` symbol to comment out blocks of nodes. Here, I am only using `node1` and every other node is commented out. When I run this parallel code, I will specify that I only want to use 16 processors, and it will use this `machines.LINUX` file to determine which node it can use those processors from.

Now let's tunnel into Kruskal and check the status in Ganglia. If a node is down, we'll want to edit our `machines.LINUX` file and re-upload it to the web-facing server before we continue. In your command prompt, you'll reach Kruskal via SSH using the command:

```
ssh -X username@kruskal.montclair.edu
```

Here the `-X` flag allows X11 to be forwarded over SSH. This will allow us to open a remote web browser on Kruskal and navigate to the internal Ganglia site. Again, you'll be asked for a password when trying to connect:

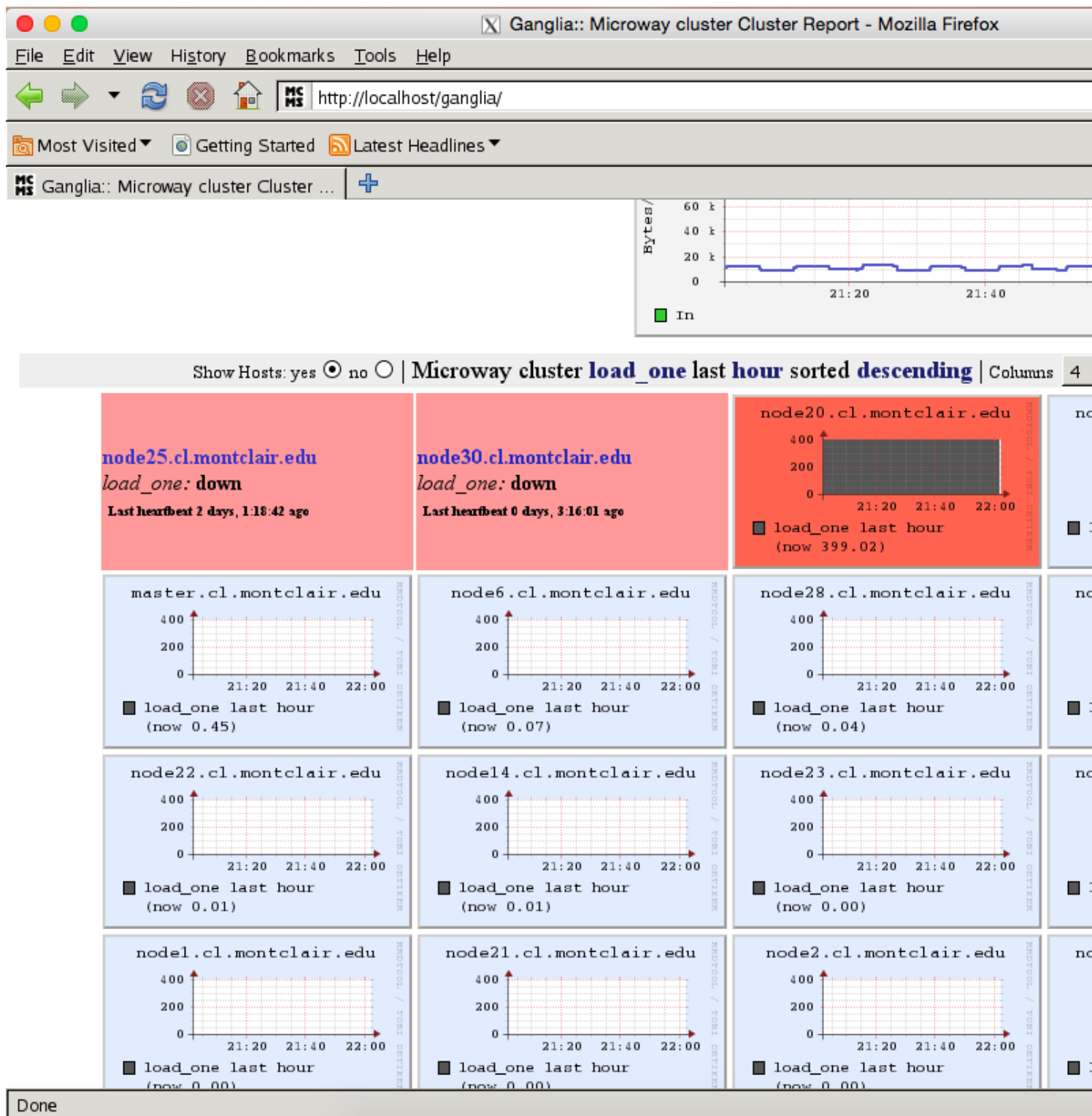
A terminal window titled 'morley@master:~' with standard window controls. The terminal shows an SSH login process for the user 'morley' on a system named 'chiricahua'. It displays the last login time and IP address, followed by a 'Have a lot of fun...' message. The user then runs 'ls', showing a directory listing with files like 'hello_world_advanced.f90' and 'machines.LINUX'. Finally, the user runs 'ssh -X morley@kruskal.montclair.edu', and the terminal shows the start of a new SSH session to 'kruskal.montclair.edu'.

```
morley@master:~  
Using username "morley".  
Using keyboard-interactive authentication.  
Password:  
Last login: Mon Sep 29 17:52:32 2014 from ool-182fa4ed.dyn.optonline.net  
Have a lot of fun...  
morley@chiricahua:~> ls  
hello_world_advanced.f90  hello_world.f90  machines.LINUX  
morley@chiricahua:~> ssh -X morley@kruskal.montclair.edu  
Password:  
Last login: Mon Sep 29 22:59:47 EDT 2014 from chiricahua.csam.montclair.edu on p  
ts/4  
morley@master ~ $
```

And we're in! The first thing we're going to do is open a web browser and check the status of all our nodes. Type `firefox` into the command prompt and a firefox browser will open (be patient, this can take a little bit over a slower connection). Once the browser is open, navigate to the following site:

`http://localhost/ganglia/`

Scrolling down will give you a list of all the nodes, as well as their status:

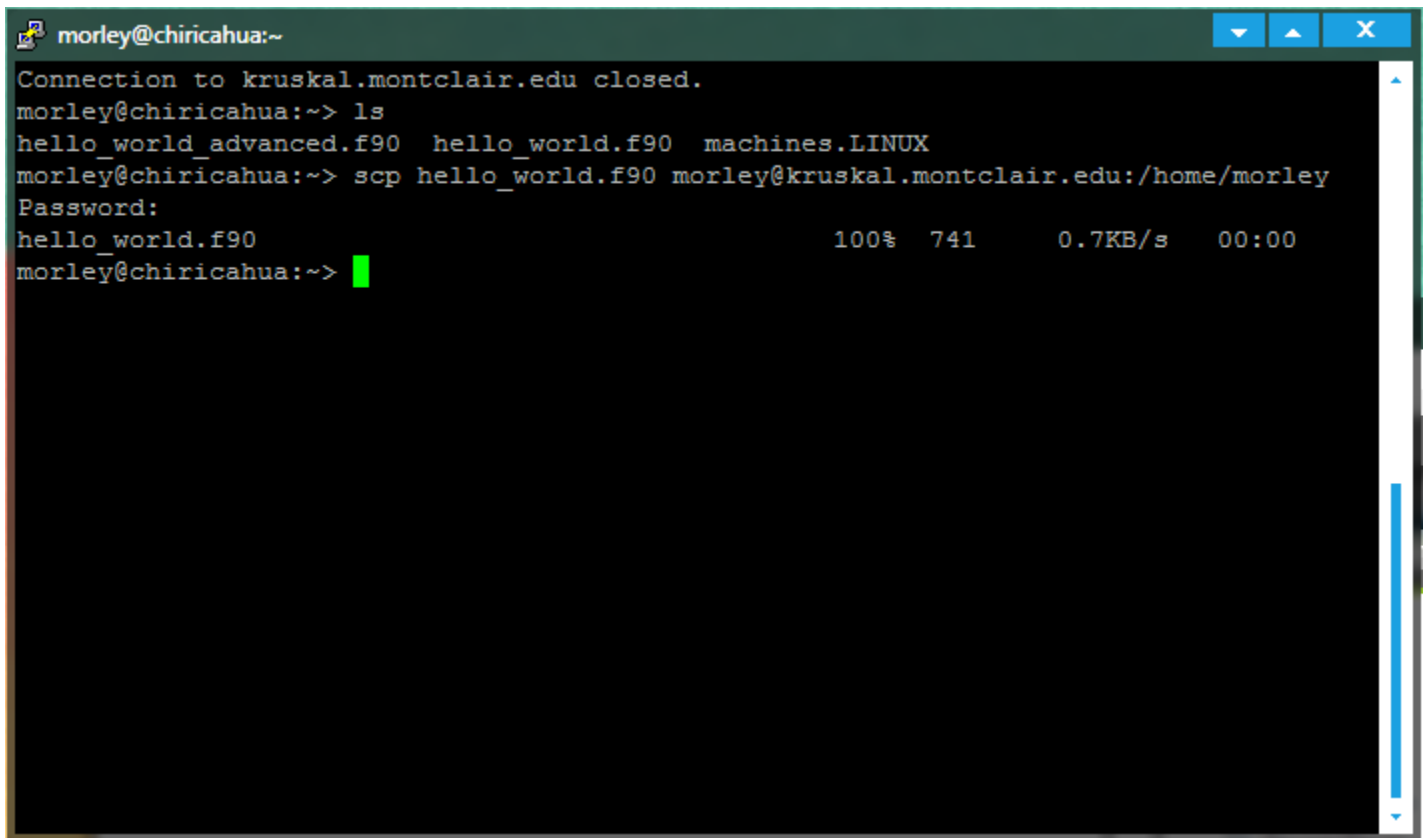


From this we can see that node25 and node30 are down, and node20 is in heavy use. You'll want to comment all instances of node25, node30, and node20 out of your machines.LINUX file and re-drag it into your CyberDuck window.

Okay, now we need to get all the files from the web-facing server over to Kruskal. Since Kruskal isn't web-facing, and since you're doing this from home, we will have to transfer the files over using the command prompt. First, close out the firefox window. Your command prompt will yell at you that the connection was broken and X11 is angry, It's fine. Type exit, and you'll log out of your ssh session on Kruskal and return to your ssh session on the web-facing server. To transfer the files to Kruskal we're going to use the scp command, which has the following syntax:

```
scp filename.extension username@kruskal.montclair.edu:/home/username
```

In the image below I transfer the file hello_world.f90. I'm prompted for my password, and then I see a completion percentage. If this doesn't display 100%, something went wrong.

A terminal window titled 'morley@chiricahua:~' with standard window controls. The terminal output shows a message 'Connection to kruskal.montclair.edu closed.' followed by a list command 'ls' showing files 'hello_world_advanced.f90', 'hello_world.f90', and 'machines.LINUX'. Then, an 'scp' command is used to transfer 'hello_world.f90' to 'morley@kruskal.montclair.edu:/home/morley'. A password prompt is shown, followed by the progress of the transfer: 'hello_world.f90' with '100%' completion, '741' bytes, '0.7KB/s' speed, and '00:00' time. The prompt returns to 'morley@chiricahua:~>' with a green cursor.

```
morley@chiricahua:~  
Connection to kruskal.montclair.edu closed.  
morley@chiricahua:~> ls  
hello_world_advanced.f90  hello_world.f90  machines.LINUX  
morley@chiricahua:~> scp hello_world.f90 morley@kruskal.montclair.edu:/home/morley  
Password:  
hello_world.f90                                100% 741      0.7KB/s   00:00  
morley@chiricahua:~> █
```

You'll want to transfer over your code files, as well as your machines.LINUX file. Once all the files are copied over, we can again ssh back into Kruskal and compile the code. Use the command:

```
ssh -X username@kruskal.montclair.edu
```

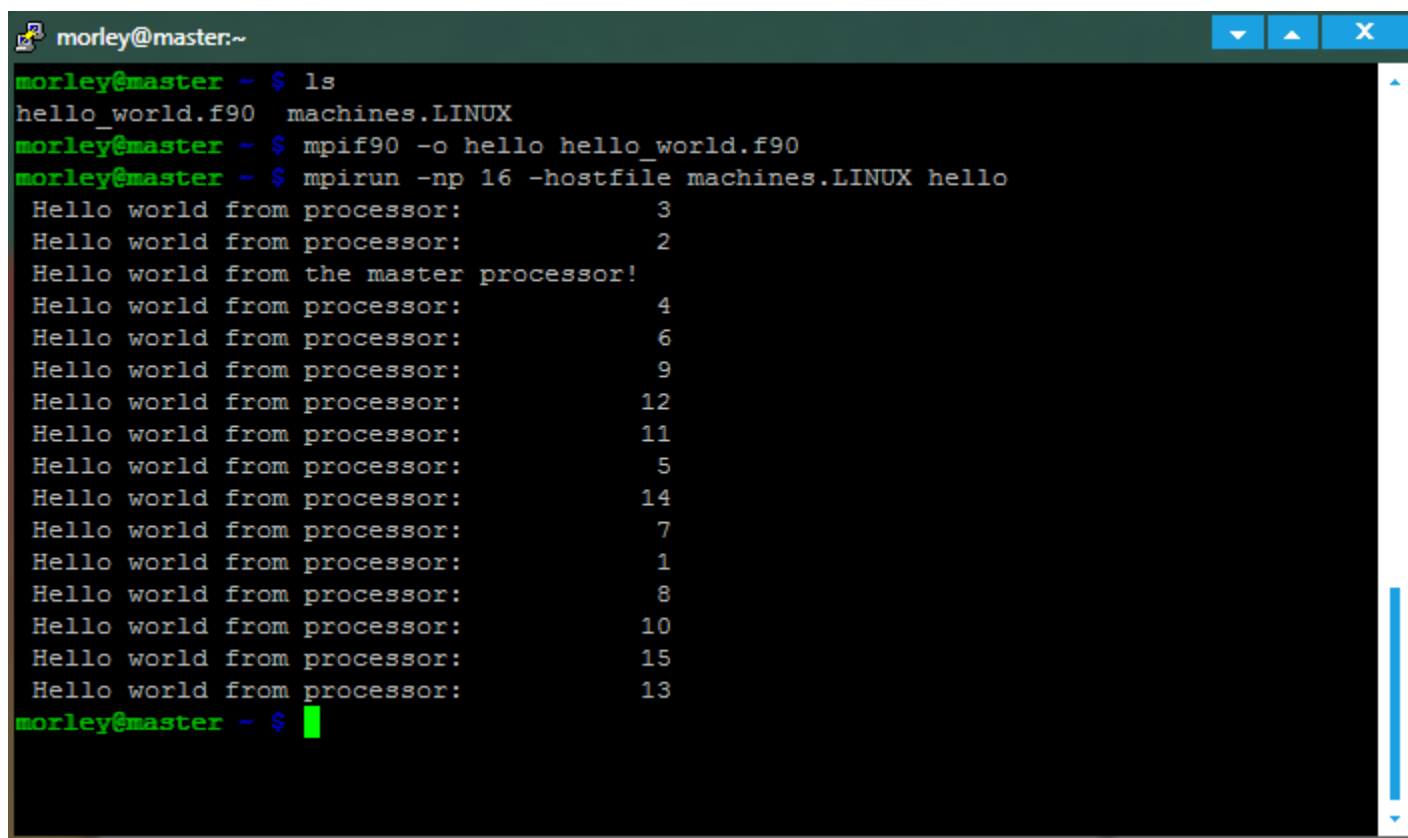
to log back into Kruskal. Now, do an ls to ensure the files have copied over. To compile parallel code, we use the following command:

```
mpif90 -o compiled_file_name program_name.f90
```

where `compiled_file_name` is a name for the compiled program (try to be descriptive!), and `program_name.f90` is the name of the Fortran program you're compiling. If there are no errors with your program, the command prompt will progress one line down and not spit out a large block of angry gibberish. Now that we've compiled the code, we need to run it, using a command with the following syntax:

```
mpirun -np number -hostfile machines.LINUX compiled_file_name
```

where `-np` is the flag where you can specify the number of processors to use, and `-hostfile` is the flag that lets you specify `machines.LINUX` to be used in determining which nodes we use. The `compiled_file_name` is the file that we're going to run (the compiled code). Below is an example where I ran the `hello_world.f90` code with 16 processors, using the `compiled_file_name` `hello`:

A terminal window titled 'morley@master:~' with standard Linux window controls (minimize, maximize, close) in the top right. The terminal shows the following commands and output:

```
morley@master ~ $ ls
hello_world.f90  machines.LINUX
morley@master ~ $ mpif90 -o hello hello_world.f90
morley@master ~ $ mpirun -np 16 -hostfile machines.LINUX hello
Hello world from processor:      3
Hello world from processor:      2
Hello world from the master processor!
Hello world from processor:      4
Hello world from processor:      6
Hello world from processor:      9
Hello world from processor:     12
Hello world from processor:     11
Hello world from processor:      5
Hello world from processor:     14
Hello world from processor:      7
Hello world from processor:      1
Hello world from processor:      8
Hello world from processor:     10
Hello world from processor:     15
Hello world from processor:     13
morley@master ~ $
```

We can see we received a Hello World! statement from the master processor as well as all the child processors! Success! To exit out of Kruskal, type the command `exit` into the terminal. You'll be taken back to your web-facing server. To exit out of this, again type `exit`. You're now logged out of everything.

Now let's consider how this is all done when you're actually on campus. It's actually much easier to work with the parallel machine when you're on campus, as you can use CyberDuck to connect directly to Kruskal and you can bypass the web-facing server all together. Simply load up CyberDuck and when you use Open Connection, select SFTP from the drop down menu enter the

server as `kruskal.montclair.edu` and use your Kruskal username and password. You can no directly drag and drop files to Kruskal. Likewise, in PuTTY, under session, you'll just want to log into `uesrname@kruskal.montclair.edu`. Easy!

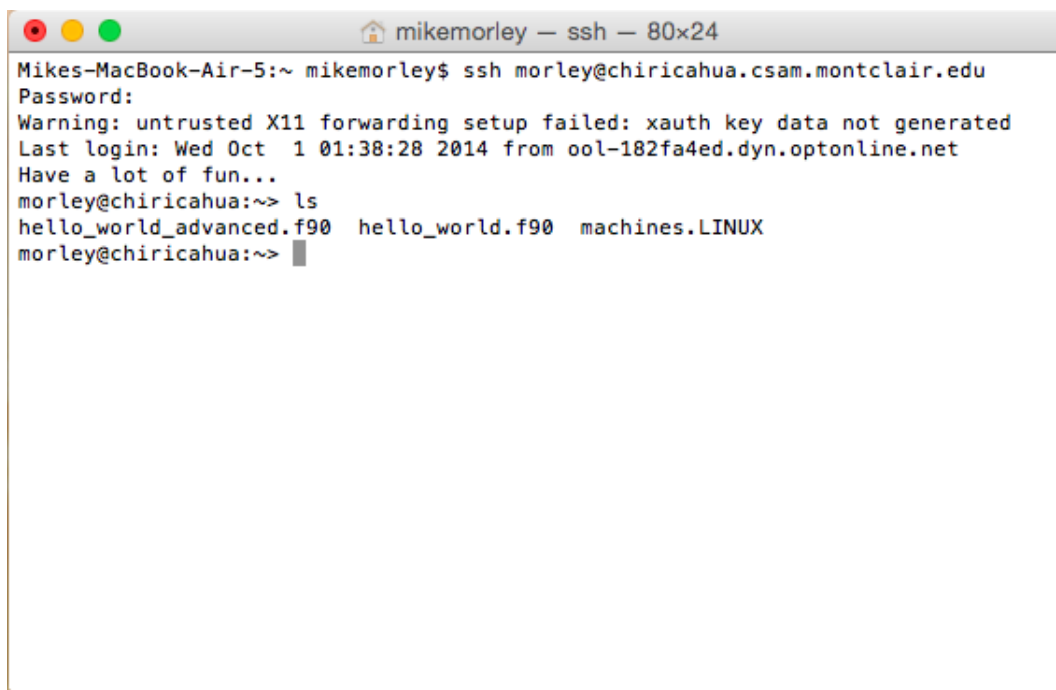
3.2 Compiling and running code for OS X users

Compiling and running parallel code on OS X is slightly easier than on Windows, due to the fact that OS X contains the terminal application built in. At this point, you should have Xquartz's X11 server installed, a decent text editor, and CyberDuck installed.

Let's start by logging into Kruskal under the assumption that you're working from home. In order to do this, you'll need to load up the terminal application that is included in OS X. This can be found in the Applications → Utilities folder. Since Kruskal is not a web-facing server, you'll need access (a username and password) to a web-facing server to tunnel into first (Dr. Forgoston provided me with access to his office server for this). To connect to the web-facing server, we're going to use the command:

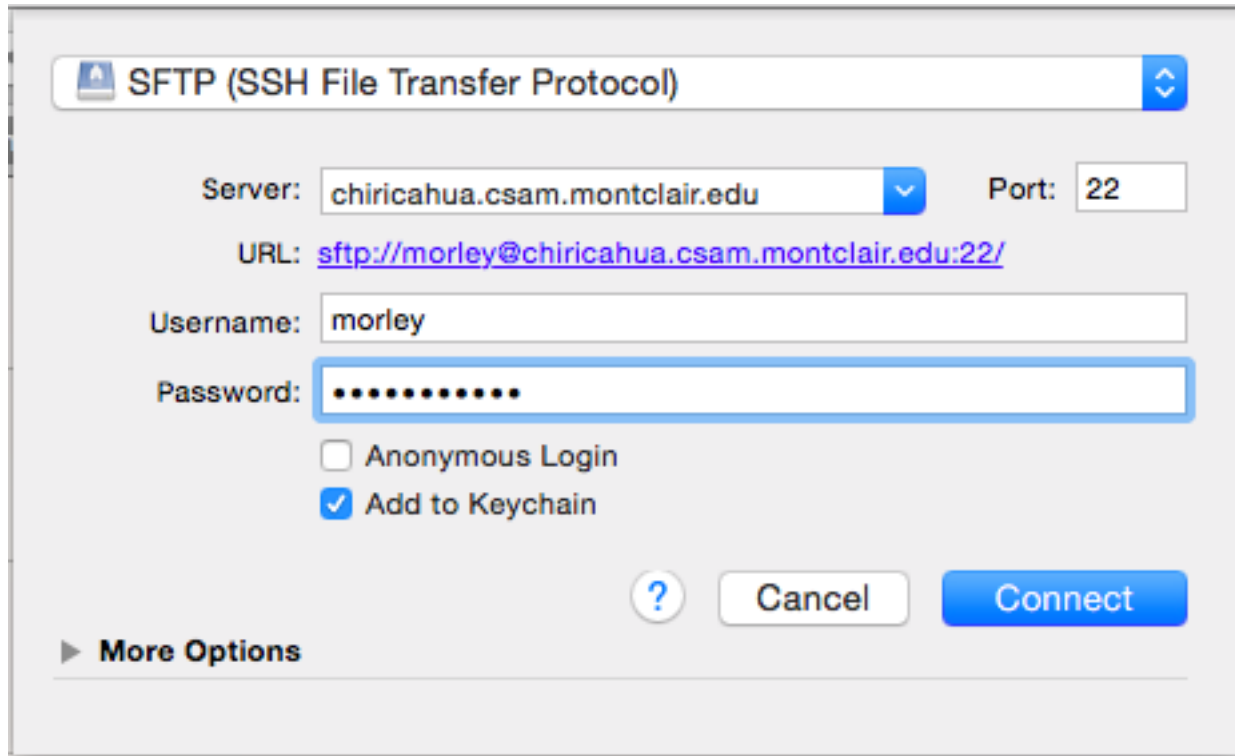
```
ssh username@serveraddress
```

In my specific case, the server address for Dr. Forgoston's server was `chiricahua.csam.montclair.edu`, and my username was `morley`. When you enter this ssh command you'll be prompted to enter your password - upon doing so will log you into the server and place you in your home directory.

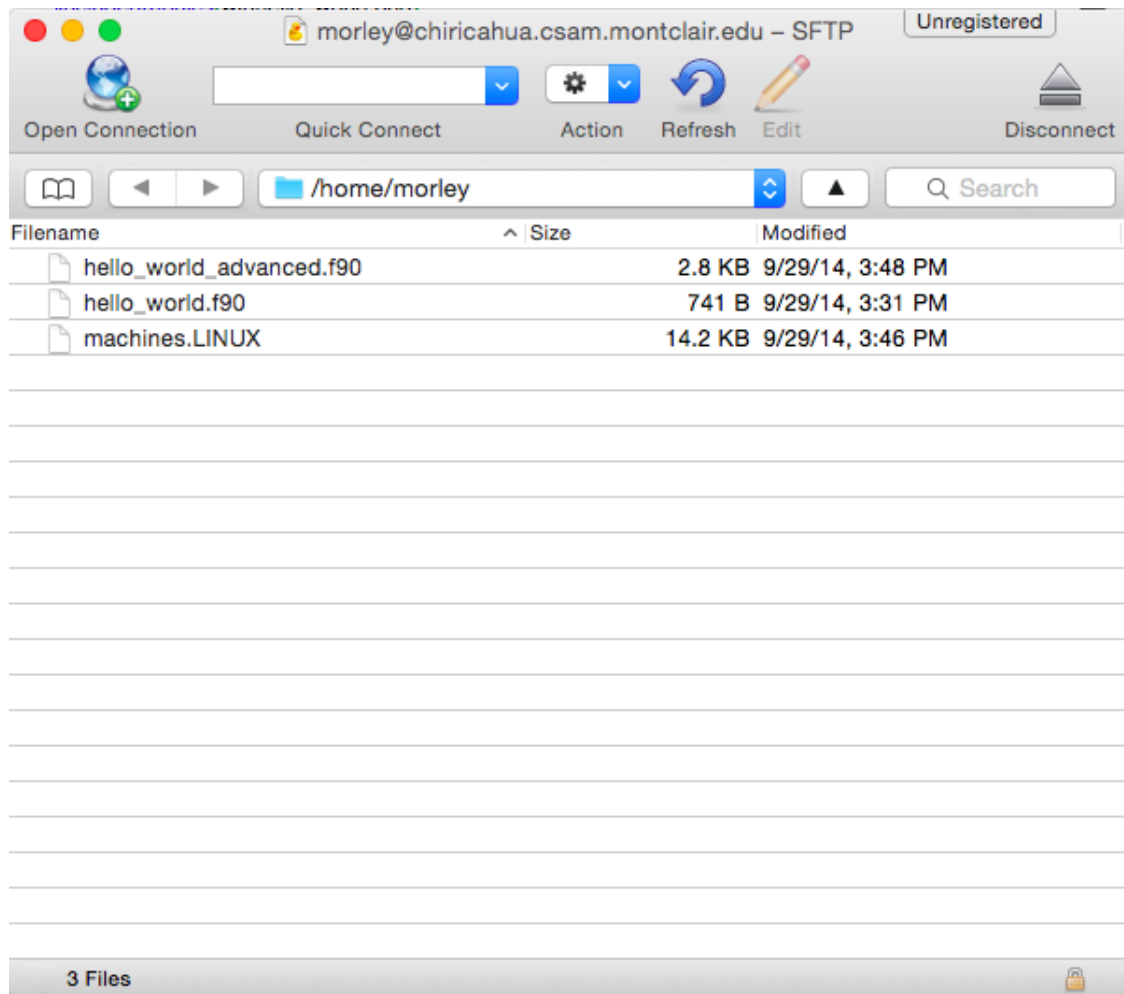
A screenshot of a macOS terminal window titled "mikemorley — ssh — 80x24". The terminal shows the command `ssh morley@chiricahua.csam.montclair.edu` being executed. The prompt changes to `morley@chiricahua:~>`. A warning message is displayed: "Warning: untrusted X11 forwarding setup failed: xauth key data not generated". The user then runs the `ls` command, which lists the files `hello_world_advanced.f90`, `hello_world.f90`, and `machines.LINUX` in the current directory. The terminal window has a standard macOS title bar with red, yellow, and green window control buttons.

The image above shows me logging into Dr. Forgoston's server. You'll note the warning about X11 - don't worry if you see this while reaching a web-facing server. We don't need X11 to work from these servers while tunneling in from home. You'll also see I did an `ls` command - which displays all the files in my directory. I already have two hello world codes in the directory, as well

as a machines.LINUX file. Next I'll show you how to connect in using CyberDuck so that we can easily drag and drop files. Load up CyberDuck, and click the Open Connections button. Enter the server name as well as your username and password.



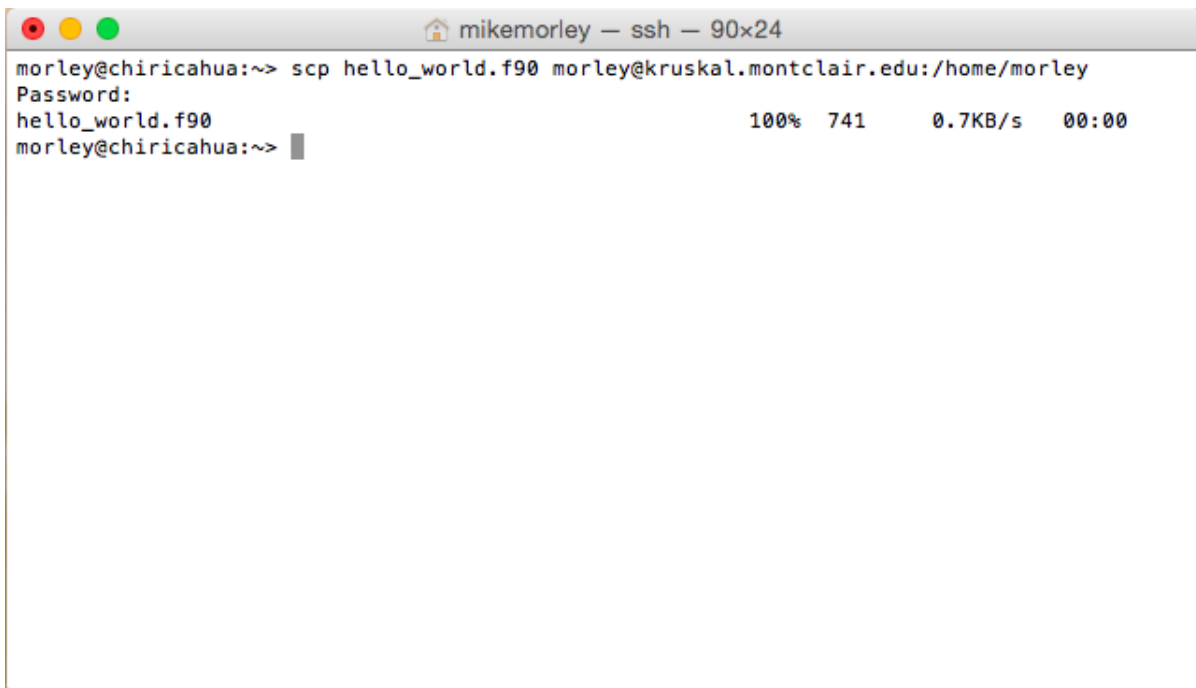
Hit connect and you'll be presented with an empty list (assuming this is your first time logging in). In my case, I already had files present so I see this:



You can now drag and drop files directly into this window (or out of it) which will allow you to easily transfer files to and from the web-facing server. To get these files onto Kruskal, we're going to need to use the terminal and some commands. Head back to the terminal, we're going to transfer `hello_world.f90` over to Kruskal using the `scp` command, which has the following syntax:

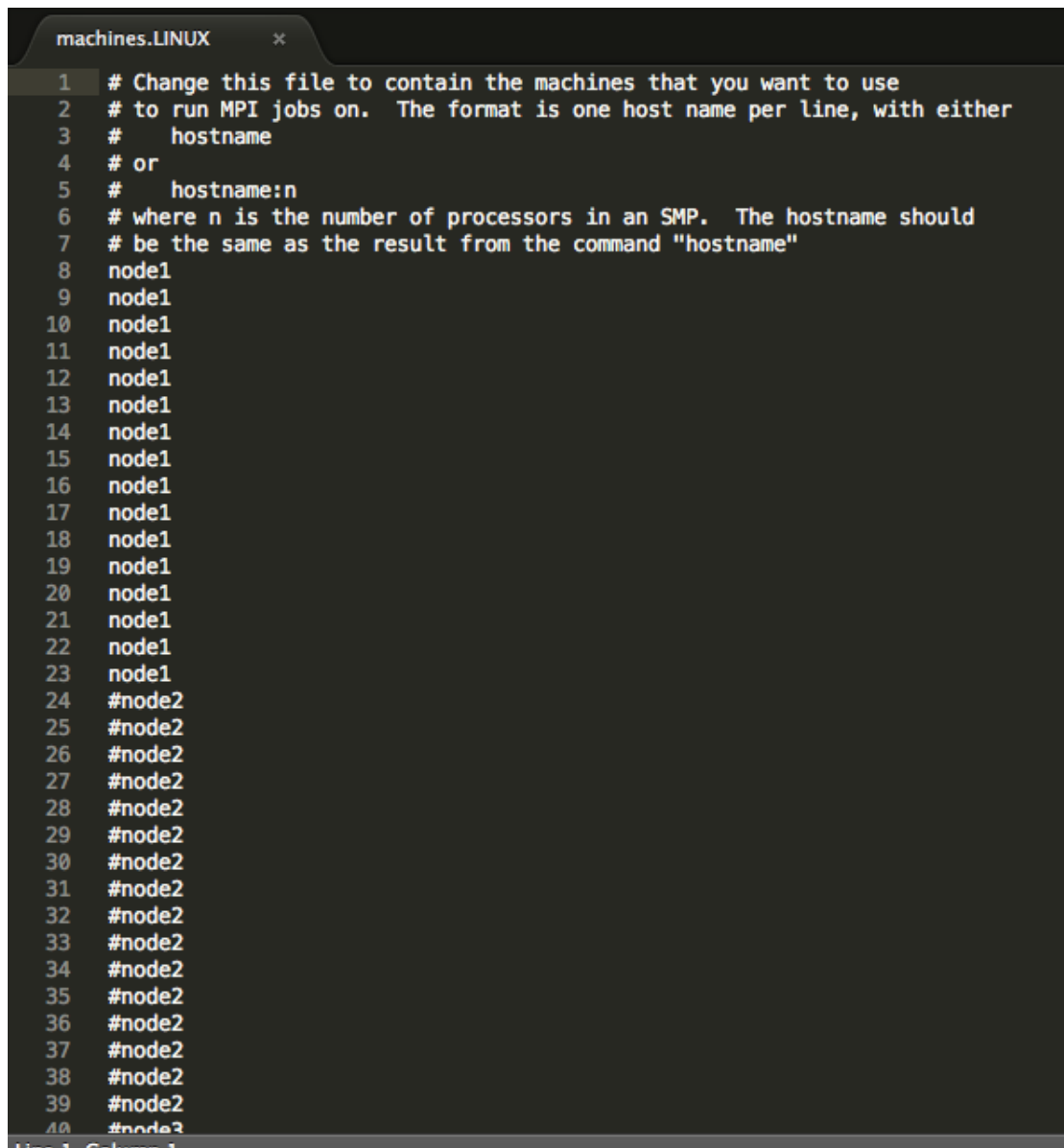
```
scp filename.extension username@serveraddress:/home/username
```

where the server address for Kruskal will be `kruskal.montclair.edu`. Since I am transferring over the file `hello_world.f90` to Kruskal, with my username being `morley` (make sure you're setup with a username and password on Kruskal before trying this!), my command looks like this:

A screenshot of a macOS terminal window titled "mikemorley — ssh — 90x24". The terminal shows a user "morley" at "chiricahua" running the command "scp hello_world.f90 morley@kruskal.montclair.edu:/home/morley". The prompt "Password:" is shown, followed by the user input "hello_world.f90". The output shows the file transfer progress: "100% 741 0.7KB/s 00:00". The terminal then returns to the prompt "morley@chiricahua:~>".

```
mikemorley — ssh — 90x24
morley@chiricahua:~> scp hello_world.f90 morley@kruskal.montclair.edu:/home/morley
Password:
hello_world.f90                                100% 741      0.7KB/s   00:00
morley@chiricahua:~> █
```

You will be prompted for a password before the transfer will occur - once this is entered you'll see a completion percentage. If you don't see a completion percentage, something likely went wrong. Repeat this process for the machines.LINUX file. This specific file is required every time you wish to run parallel code, as it tells the parallel computer which nodes we will be using. This is what the file looks like:

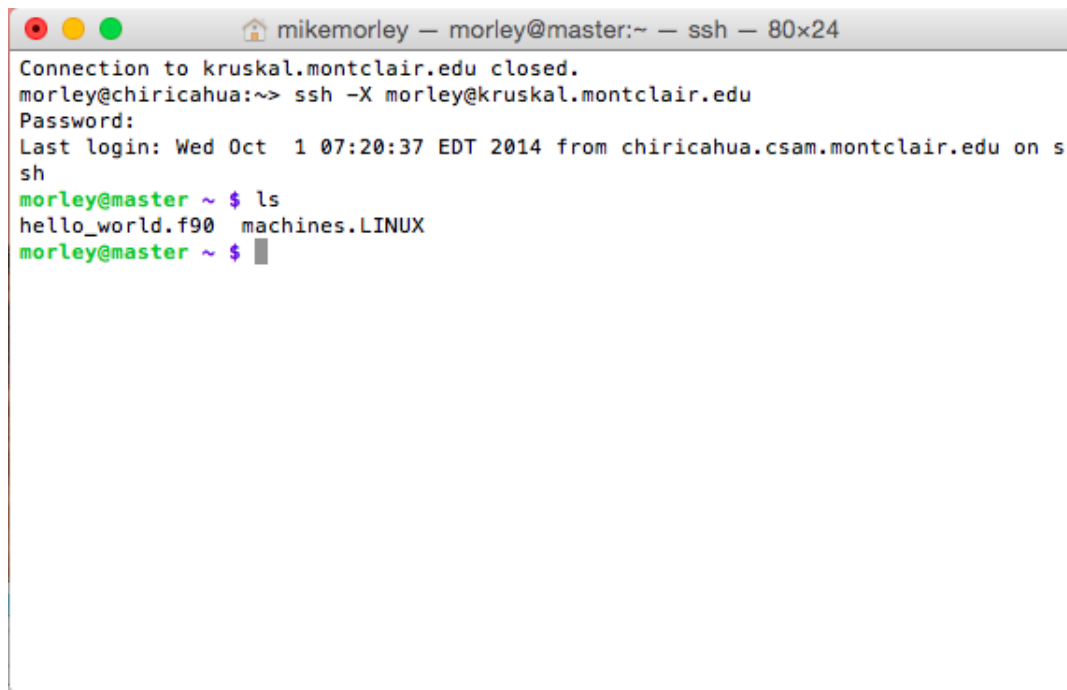


```
machines.LINUX x
1 # Change this file to contain the machines that you want to use
2 # to run MPI jobs on. The format is one host name per line, with either
3 # hostname
4 # or
5 # hostname:n
6 # where n is the number of processors in an SMP. The hostname should
7 # be the same as the result from the command "hostname"
8 node1
9 node1
10 node1
11 node1
12 node1
13 node1
14 node1
15 node1
16 node1
17 node1
18 node1
19 node1
20 node1
21 node1
22 node1
23 node1
24 #node2
25 #node2
26 #node2
27 #node2
28 #node2
29 #node2
30 #node2
31 #node2
32 #node2
33 #node2
34 #node2
35 #node2
36 #node2
37 #node2
38 #node2
39 #node2
40 #node3
```

We can see that I've commented out all nodes except node1. This will tell the parallel computer that we're only using to be using node1, and all 16 processors on node1 are eligible for use. You may be asking how to determine which nodes to use - that is what I'll show you next. We're going to put that X11 package we installed to use. Next step, we're going to finally log into Kruskal! Head back to your terminal window and type the following command:

```
ssh -X username@kruskal.montclair.edu
```

The flag -X will allow us to use X11 forwarding over ssh, which will allow us to remotely open a web browser window. You'll be prompted for your password for your kruskal username - enter it. Once connected, you can do an ls command to ensure the files we just copied over are in your home directory:

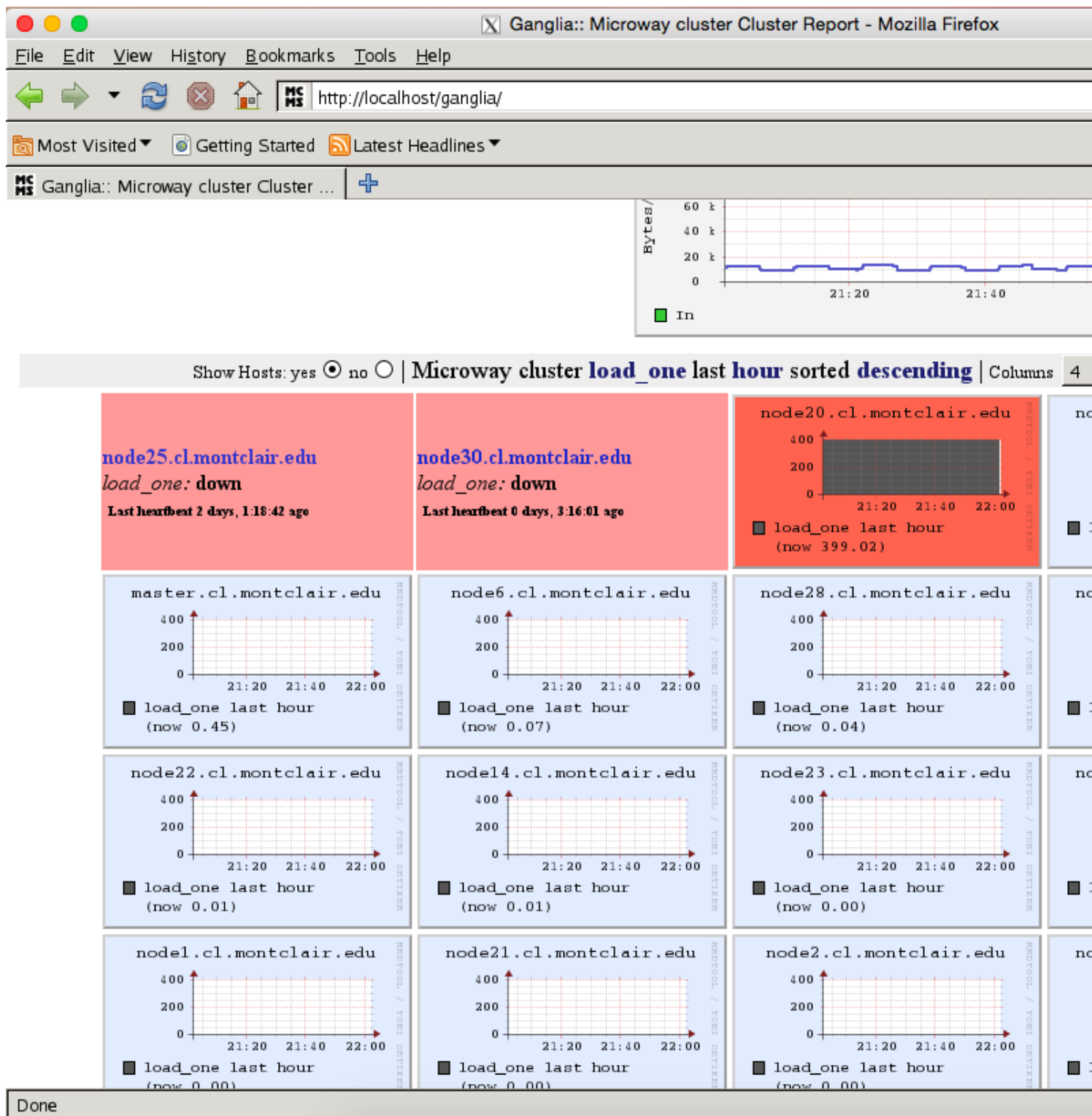


```
mikemorley — morley@master:~ — ssh — 80x24
Connection to kruskal.montclair.edu closed.
morley@chiricahua:~> ssh -X morley@kruskal.montclair.edu
Password:
Last login: Wed Oct  1 07:20:37 EDT 2014 from chiricahua.csam.montclair.edu on s
sh
morley@master ~ $ ls
hello_world.f90  machines.LINUX
morley@master ~ $
```

We're going to open up an instance of Firefox so we can head over to Ganglia and check the status of all the nodes. Simply type `firefox` into the terminal and give it a few seconds to open the browser window. Next, navigate to:

`http://localhost/ganglia/`

Scrolling down you can see a list of all the nodes like so:



We can see in this image that node25, and node30 are down. Node20 is being used heavily. We'll want to avoid those nodes all together - so you'll want to go back and comment all instances of node25, node30, and node20 out of your machines.LINUX file using `#`. Once this is done, you'll want to drag and drop that file back into CyberDuck, and then use the `scp` command to transfer it

back to your directory on Kruskal (Note: if your terminal doesn't let you input text, it's because the browser window is open. You'll need to close the browser window using `command+Q`, and then type `exit` into the terminal to leave your current Kruskal session and go back to your session on the web-facing server, where you can again use the `scp` command to transfer the `machine.LINUX` file back to Kruskal). You should make a habit of checking Ganglia to see what nodes are available for use!

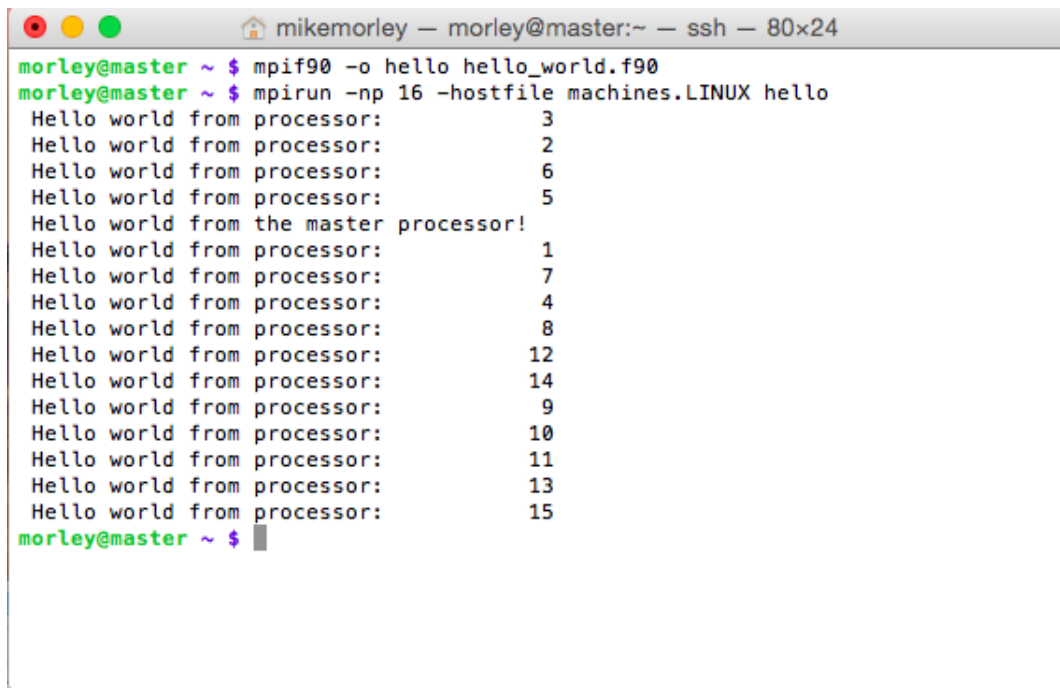
Okay, to recap what we've done: we've successfully tunneled into a web-facing server from home, we were able to drag and drop files to this server, as well as copy them to Kruskal using a command line command. We've checked Ganglia and adjusted our `machine.LINUX` file to ensure we don't use nodes that are down or in heavy use. We've left in a state where we can `ssh` into Kruskal, and our code file and `machine.LINUX` file are in our home directory on Kruskal. Now let's compile and run some code! `SSH` back into Kruskal. The syntax of the compile command is as follows:

```
mpif90 -o compiled_file_name program_name.f90
```

where `compiled_file_name` is a name for the compiled program (try to be descriptive!), and `program_name.f90` is the name of the Fortran program you're compiling. If there are no errors with your program, the command prompt will progress one line down and not spit out a large block of angry gibberish. Now that we've compiled the code, we need to run it, using a command with the following syntax:

```
mpirun -np number -hostfile machine.LINUX compiled_file_name
```

where `-np` is the flag where you can specify the number of processors to use, and `-hostfile` is the flag that lets you specify `machine.LINUX` to be used in determining which nodes we use. The `compiled_file_name` is the file that we're going to run (the compiled code). Below is an example where I ran the `hello_world.f90` code with 16 processors, using the `compiled_file_name` `hello`:



```
mikemorley — morley@master:~ — ssh — 80x24
morley@master ~ $ mpif90 -o hello hello_world.f90
morley@master ~ $ mpirun -np 16 -hostfile machines.LINUX hello
Hello world from processor:      3
Hello world from processor:      2
Hello world from processor:      6
Hello world from processor:      5
Hello world from the master processor!
Hello world from processor:      1
Hello world from processor:      7
Hello world from processor:      4
Hello world from processor:      8
Hello world from processor:     12
Hello world from processor:     14
Hello world from processor:      9
Hello world from processor:     10
Hello world from processor:     11
Hello world from processor:     13
Hello world from processor:     15
morley@master ~ $
```

We can see a Hello World! statement coming from the master processor as well as all child processors! Success! To exit out of Kruskal, type the command `exit` into the terminal. You'll be taken back to your web-facing server. To exit out of this, again type `exit`. You're now logged out of everything.

Now let's consider the case where you're connecting to Kruskal from campus. You can, essentially, completely ignore having to connect to a web-facing server first, and connect directly to Kruskal. This means you can use CyberDuck to directly connect to Kruskal and drag and drop files (connect to the server `kruskal.montclair.edu` through SFTP, using your username and password), and you can simply skip the ssh session with the web-facing server. So from campus, you can load up terminal, and ssh directly into Kruskal:

```
ssh -X username@kruskal.montclair.edu
```

Easy!

3.3 Running the hello_world_advanced example code and its output

This will be written using the OS X terminal, but the commands and output will be the same across OS X and Windows. If you've followed the guide to this point, you'll understand how to compile and run the code, so let's take a look at the output that you'll see in the console:

```
morley@master ~ $ mpif90 -o helloadv hello_world_advanced.f90
morley@master ~ $ mpirun -np 16 -hostfile machines.LINUX helloadv
Hello world from processor: 5
Hello world from processor: 5
Hello world from processor: 5
Hello world from processor: 5
Hello world from processor: 5
Hello world from processor: 6
Hello world from processor: 6
Hello world from processor: 6
Hello world from processor: 6
Hello world from processor: 6
Hello world from processor: 14
Hello world from processor: 14
Hello world from processor: 14
Hello world from processor: 14
Hello world from processor: 14
Hello world from the master processor!
Hello world from the master processor!
Hello world from the master processor!
Hello world from the master processor!
Hello world from the master processor!
Hello world from processor: 10
Hello world from processor: 10
Hello world from processor: 10
Hello world from processor: 10
Hello world from processor: 10
Hello world from processor: 4
Hello world from processor: 4
Hello world from processor: 4
Hello world from processor: 4
Hello world from processor: 9
Hello world from processor: 9
Hello world from processor: 9
Hello world from processor: 9
Hello world from processor: 2
Hello world from processor: 2
Hello world from processor: 2
Hello world from processor: 2
Hello world from processor: 3
```

Recall that this advanced example is having the master and all child processors print a hello world statement to the console. Each child processor is going to generate two random numbers, which will be sent to the master processor and stored in a text file. This entire process is done five times, and from the output we can see this is successful since each processor did its hello world statement five times. Lets take a look at the text file we created, using the cat command:

```

morley@master ~ $ ls
hello          hello_world_advanced.f90      helloadv
hello_world.f90 hello_world_advanced.o        machines.LINUX
hello_world.o  hello_world_random_numbers_output.dat
morley@master ~ $ cat hello_world_random_numbers_output.dat
 1  0.34229878      0.40238118
 1  0.10771511      0.53055429
 1  0.28478709      0.46192497
 1  0.12283650      0.42846999
 1  6.80166185E-02  5.12838885E-02
 1  1.87253784E-02  0.71742773
 1  0.84702754      0.97158414
 1  0.72494197      0.21150552
 1  0.29544553      0.56513327
 1  0.89791572      0.55824912
 1  0.17286055      0.68040943
 1  0.19288747      0.42718062
 1  0.66842240      0.83415651
 1  0.85693997      0.63351482
 1  0.69250333      0.11547279
 2  0.34229878      0.40238118
 2  0.10771511      0.53055429
 2  0.28478709      0.46192497
 2  0.12283650      0.42846999
 2  6.80166185E-02  5.12838885E-02
 2  1.87253784E-02  0.71742773
 2  0.84702754      0.97158414
 2  0.72494197      0.21150552
 2  0.29544553      0.56513327
 2  0.89791572      0.55824912
 2  0.17286055      0.68040943
 2  0.19288747      0.42718062
 2  0.66842240      0.83415651
 2  0.85693997      0.63351482
 2  0.69250333      0.11547279
 3  0.34229878      0.40238118
 3  0.10771511      0.53055429
 3  0.28478709      0.46192497
 3  0.12283650      0.42846999
 3  6.80166185E-02  5.12838885E-02
 3  1.87253784E-02  0.71742773
 3  0.84702754      0.97158414
 3  0.72494197      0.21150552
 3  0.29544553      0.56513327
 3  0.89791572      0.55824912
 3  0.17286055      0.68040943
 3  0.19288747      0.42718062
 3  0.66842240      0.83415651
 3  0.85693997      0.63351482
 3  0.69250333      0.11547279
 4  0.34229878      0.40238118
 4  0.10771511      0.53055429
 4  0.28478709      0.46192497
 4  0.12283650      0.42846999

```

We have three columns, the first column is from the variable `i`, which is keeping track of which full iteration is being done (out of the five total iterations), the second and third columns are the random numbers that we generated on each child processor. Everything works!

This should conclude the guide. By now, you should have a reasonable understanding of how to get onto Kruskal and compile and run parallel code. You should also know how to get to Ganglia and ensure you're using nodes that are not down or in heavy use in your machines.LINUX file. You should also have a basic understanding of how to write code using MPI. The included code files can

be used as templates if your goal is to run population simulations. If you'd like to make corrections or add to this guide, the tex files should be available from Dr. Forgoston. Good luck!

4 Additional command line statements, links, and guides

The following command line statements may help you when working with Kruskal in a command line environment:

`http://mally.stanford.edu/~sr/computing/basic-unix.html`

The following contains additional information about MPI:

`http://condor.cc.ku.edu/~grobe/docs/intro-MPI.shtml`

Additional fortran90 and MPI examples (also in C!):

`http://people.sc.fsu.edu/~jburkardt/f_src/mpi/mpi.html`