Chapter 20

WORKING ON REMOTE COMPUTERS

The final chapters of this book delve into more specialized subjects, some of which won't apply to everyone, and others of which are unlikely to be used every day. First among these topics is working on remote computers. Remote tools are necessary if you need to conduct analyses that require more processors, more memory, or more software than you have on your personal computer. The remote computer can be a large cluster shared by many investigators, or a single desktop computer in your lab. We also introduce tools that control how programs run, including their priority and whether they should stop when you log out.

Connecting to a remote computer

Many biologists venture to the command line only when forced to, usually because they have to run analyses on a remote computer that does not provide a graphical user interface. It can be daunting to learn how to use the command line at the same time you are figuring out how to connect to other computers, transfer files to remote accounts, launch programs on large shared clusters, and manage program execution. This is one reason that it is so valuable to become familiar with the command line on your own computer—it provides a foundation upon which you can incrementally add the skills for remote work. In this chapter, which builds directly on the command-line lessons of Chapters 4–6, and 16, you will learn how to get command-line access to remote computers, transfer files, and control analyses. Some of these skills will also be relevant to how you run larger analyses on your own computer.

Clients and servers

You will often hear the words **client** and **server** when discussing network connections. In most situations the client is the computer where the user is sitting and where the connection is initiated from, while the server is the remote computer

that accepts the connection and provides a service. There are many kinds of servers. Physically, a server can be anything from a regular personal computer—even a laptop that just happens to be configured as a server—to a massive data center occupying several city blocks. A server can serve many different things to its clients, including disk space, web pages, databases, and computational power. Each such service has its own particular connection needs and therefore requires different software and protocols (i.e., agreed-upon standards for how to interact with other computers).Though client and server speak to each other in a common language, they too require different software and different configuration settings. Finally, for each network protocol, a computer may be set up as a client, a server, or both.

Almost all scientific servers run a version of Linux or Unix optimized for large-scale, computationally intensive tasks. These operating systems and the programs bundled with them can differ in subtle ways from the command-line interfaces of Mac OS X and Ubuntu Linux which we have focused on in this book. The operating systems on remote servers have a longer history and are more specialized than those found on personal computers, and are therefore more varied. As a result, there are some limitations as to the generality of what we can present here. Even things like using the ↑ key to cycle through your command history, or using the backspace key to delete characters, may not work as expected on a remote system. If you find that you have questions about a particular server, or that the material in this chapter doesn't seem to apply, look for help on the server's support Web site (if there is one), or contact the administrator for the machine you are working on.

Typical scenarios for remote access

There are a few typical scenarios that require the remote access tools described in this chapter. An overview of a couple of these at the outset will help you understand which parts of the chapter are most relevant to you, and how each component fits into the overall objective.

Running analyses on a large cluster First scenario: You need to perform an analysis that would take weeks or months on your laptop. After investigating the computational resources at your university, you find that you can use a Linux cluster available on campus. The cluster consists of a group of computers linked together via a local area network, effectively comprising a single computer with several hundred processors. You contact the system administrator, who confirms that the software you need is already installed on the cluster and is configured to use multiple processors at once.

The administrator creates an account for you, and provides all the information you need to log in remotely. You verify that you can log in. On your laptop you prepare the material you will be working with, making sure that you have all the files you need, that their data are correctly formatted, and that folders are in place to hold both data and analyses. You compress all this into a file archive and upload the archive to the cluster via a secure file transfer program. You then log into the cluster at the command line and expand the file archive. Following the instructions on the cluster Web site, you start the analysis with a special shell script that

provides access to multiple processors. You log out, and several days later get an e-mail notifying you that your job has completed. You log into the cluster, compress the result files into an archive, and then transfer that archive back to your laptop to investigate the results.

Checking in on a lab computer from home Second scenario: Imagine that your lab has a large multiprocessor desktop computer off in a corner. It is connected to your favorite instrument, which is automated and takes a few days to collect data on your samples. The lab manager has configured this computer as a server, so that lab members can gain command-line access to it and transfer files remotely. You live a half hour from the lab. One Sunday morning, you wonder how the current run on the instrument is going. You log in to the lab computer and check the data files. Everything is on track, and you can go to the North Shore instead of making an unneeded trip to the lab just to check the status.

Finding computers: IP addresses, hostnames, and DNSs

In order for computers to connect to each other on the Internet, they must be able to find each other. Understanding a bit about how this is done will make the process more transparent and help you troubleshoot some of the most common connection problems.

All computers on the Internet—whether a web server or a laptop computer connected by WiFi at a library—have a network number, known as an Internet Protocol address. Traditional **IP addresses**, such as `173.194.33.104`, consist of four 8-bit numbers—that is, four numbers ranging from 0 to 255.[1] You can think of the progression of these four numbers as successively narrowing the address down to a particular location. The first number in the series is the broadest, and the fourth and final number is the most specific, identifying an individual network connection (e.g., a computer, instrument, smartphone, etc.). Your school, company, and even your local coffee shop might therefore have addresses where the first few numbers are the same, and only the final one or two numbers differ between computers.

IP addresses are cumbersome for humans to remember and type, so most computers are also assigned a text-based hostname. The hostname is familiar as the base of a Uniform Resource Locator, or URL, in Web addresses, such as `sinauer.com` or `practicalcomputing.org`. In order to connect to a computer with a given hostname, however, the hostname first has to be translated to an IP address. When you type a URL into your web browser, your computer finds the IP address for the specified hostname and then makes a request for the indicated files from the computer identified by the IP address.

Translation between names and addresses is the job of Domain Name System servers, which collectively function as the Internet's equivalent of a phone

[1]This traditional scheme for encoding IP addresses is known as IPv4. Under IPv4, each IP address is a 32-bit number (four 8-bit numbers together). However, only about four billion devices can be addressed under this scheme, and these addresses have nearly all been used. To get around this problem, a new protocol called IPv6 has been developed which uses up to 128-bits per address; this will provide more than 3×10^{38} addresses altogether.

book. There are many DNS servers, each with their own copy of the master DNS database. This database is updated many times a day as new hostnames and IP addresses come online. Often your Internet Service Provider will specify a DNS server to use, or the local network may be configured so that computers can automatically connect to the local DNS server without custom configuration. There are also several publicly available DNS servers, such as Google Public DNS (IP address `8.8.8.8`), which will work from almost anywhere.

As the Internet has grown, more flexibility has been added to the relationship between hostnames and IP addresses. Multiple hostnames can point to the same IP address; among other things, this allows more than one Web site to be hosted on the same computer. Because there is not always a one-to-one correspondence between IP addresses and hostnames, when you look up the IP address from a hostname, and then do the reverse search, you may get a different IP address than you started with. The situation is further confused by the fact that many hostnames are aliases (shortcuts) to other hostnames.

Translating between IP addresses and hostnames The command-line program `host` allows you to find the IP address for a hostname. (The program `nslookup` does the same thing.) In some cases, it can also find a hostname if you give it an IP address. This command is dependent on having a network connection, because it sends your query out to a DNS server and returns the result:

```
myhost:~ lucy$ host www.jellywatch.org
www.jellywatch.org has address 134.89.10.74
myhost:~ lucy$ host 134.89.10.74
74.10.89.134.in-addr.arpa domain name pointer pismo.shore.mbari.org.
```

Sometimes, the name or IP number you are trying to find with the `host` command won't have any available information and the lookup will fail. Even when `host` fails, however, you can get some general information with the `whois` command. The `whois` command requires that you provide the name of an online `whois` database with the `-h` parameter, which is then followed by the address you want more information for:

```
whois -h whois.arin.net 75.119.192.137
```

This will return the registration information for the entire block of addresses from `75.119.192.0` to `72.119.223.255`, which includes the IP address you specified.

The special hostname `localhost` The hostname `localhost` always points to the computer that you are logged onto at the moment, and it always has the IP address `127.0.0.1`. Try checking this using the command `host localhost`. If you have web sharing enabled on your computer, `http://localhost` can even be used as an address in your web browser. (You will get a "could not connect to

localhost" error if you don't have web sharing turned on for your computer. For OS X, this can be turned on in the Sharing pane in System Preferences.)

Security

When working on computers remotely, security is of paramount concern. You should be thinking of security every step of the way. It should go without saying that you never give your password to anybody, not even system administrators. If a system administrator needs access to your account, they can reset the password themselves, and you can change it again later. This is standard etiquette, and any deviation from it may be grounds for concern. Use different passwords for different computers so that they won't all fall victim if the password for just one is compromised.

If you are configuring a server, keep in mind that you are opening it up to possible attacks. Don't enable any network protocols or services that aren't essential, as each protocol is a possible vulnerability. Make sure that all software is kept up to date, since many updates patch security vulnerabilities that will shortly become common knowledge to hackers. One common strategy to prevent unwanted access is to restrict the IP addresses that can access particular services, for instance by limiting access to only computers on the same campus. The method for doing this depends on the operating system and network protocols you are using.

Security should also be a concern when you are logging into a remote server from your own computer. Keep your own computer software up to date; this will help ward off keyloggers and other malicious programs that could steal login information and then compromise the other computers you connect to. Wherever possible, use encrypted connections that prevent others from listening in on your connection to the remote computer. Avoid connecting to servers over open network connections, such as unsecured WiFi at a coffee shop, without turning on tools (such as VPN, described later) that encrypt your connection.

Secure command-line connections with `ssh`

Much of this book has focused on learning to interact with the computer in front of you, using the command line. You are now familiar with the process of opening a terminal window, which launches a shell program and gives you a prompt, and then using it to navigate and control the computer attached to the screen and keyboard. As you will see, it is just as simple to use the command line to interact with a remote computer over a network connection. You will still sit at your own computer with your own screen, keyboard, and terminal window, but now the window will be relaying your commands to the remote computer and returning the results. It is as if you were physically sitting at the remote computer (which, in reality, may not even have a keyboard or screen).

Remote command-line logins are made with a network protocol called `ssh` (**s**ecure **sh**ell). The `ssh` program creates an encrypted connection between your client computer and the remote server, so that when you type at the terminal window in your computer this window is a connection to the remote computer. The data

transmitted via `ssh` are encoded so that it would be difficult for anyone listening in along the way to figure out what is being said.

The `ssh` *command*

Since we don't have a server that we can open up to every reader of the book, it won't be possible to follow along with remote access commands in this chapter unless you have your own server access. If you are affiliated with a university or company, you probably already have an account on a server, though you might not know about it. The end of the chapter has information on setting up a server, which you can use to configure a server of your own to test with some of the examples here.

You need at least three pieces of information to log in to a remote computer with `ssh`. First, you need the remote computer's address. This can be the IP address or a human-readable machine name, such as `myhost.myuniversity.edu`. Second, you need a username, sometimes called an account name. Third, you need a password. The system administrator will create your account and provide these pieces of information to you. Don't be surprised if they won't email them to you—the content of an email is visible to many computers en route and is a dangerous way to send security-critical information such as login credentials. Some administrators will email your username and the address of the computer, and then call you with the password.

The first thing you should do when you log in is change your password with the `passwd` command. You can follow the simple prompts to accomplish this. The command to launch a connection to a remote server with `ssh` has the following general structure:

```
ssh username@address
```

where `username` and `address` are specific to the server you are connecting to. After entering this command and pressing return, you will be prompted for a password. Enter the password provided by the administrator, and press return. If your login credentials are correct, you will then be greeted by a command-line prompt served up by the remote computer.

The first time you connect, the server will give you a warning that it has received a new key, and ask if you want to add it to your keychain. This key is a bit of information that allows the two computers to set up an encrypted communication channel. Accept the new key. If you are ever prompted to accept a new key for the site again, contact the administrator for the remote system. It could simply be a change to the remote computer that necessitates a new key, but it could also be an attempt by a nefarious party to intercept your communication.

Troubleshooting `ssh`

When logging into a remote system, you might get this error:

```
ssh_exchange_identification: Connection closed by remote host
```

This is usually an indication that the host on the other end did not approve of the address of your computer. Depending on what the system allows, there are two possible solutions. One is to set up a Virtual Private Network (VPN) connection to make your computer's address seem as though it is located internally alongside the system you are trying to connect with; see the section about VPN later in this chapter. If this capability isn't available, the other solution is to contact the system administrator of the remote machine and get him or her to add your address (or the range of addresses from which you might connect) to the list of allowed clients.

Once you have logged into a server from one computer, if you later log in from a different computer you may get an error of varying severity. For example:

```
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now !
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
...
Host key verification failed.
```

In such cases, the secure key passed between the systems has changed. Edit the `~/.ssh/known_hosts` file in your home directory to remove the line for the host you are trying to connect with. When you reconnect again the next time, it will generate a new key pair for you.

Working on the remote machine

The `ssh` program provides seamless command-line access to a remote computer, in the sense that it is no different from being physically located in the same room with the server. You may find, however, that your command-line experience on the remote computer is quite different from working on your own computer. This is because the remote computer may be running different software and have unfamiliar configurations. One of the most common surprises is that the shell program itself may be different. Even if you are using the `bash` shell on your local machine when you launch `ssh`, you will interact with the remote computer using its own default shell, which may or may not be `bash`. This shell starts once the `ssh` connection is active. If it is a different shell than `bash`, you may notice some subtle differences in the way it responds and how the prompt is presented. It is therefore a good idea to check which type of shell it is, by typing `echo $SHELL`. If the remote shell isn't `bash`, but instead one of the common alternatives such as `tcsh`, `sh`, or `zsh`, consult the online documentation for that type of shell to familiarize yourself with the most critical differences between it and `bash`.

Your account on the remote machine will have a home directory, independent of the home directory on your own computer. When you first log in you will have default shell configurations (these will be located in `.bash_profile` if the remote shell is `bash`); your `PATH` will only include directories set at the system level; and your home directory will be largely empty.

Once you have command-line access to the remote system, you can interact with the remote file system with the same command-line tools you use on your local computer. For instance, you can create folders on the remote computer with `mkdir`, change directories with `cd`, and list folder contents with `ls`. One of the first things you'll want to do upon logging into a remote system is explore the files and folders in your home directory (there is often a `README` file that explains system policy and use) and set up new folders to store data and analysis results.

While `ssh` provides a way to log in and operate a remote computer, it doesn't provide direct access to files on the remote machine. You will have to use other tools, as described in the next section, to transfer files back and forth. You can accomplish other tasks entirely on the remote machine. For example, to make small changes to a remote file, it is often more convenient to edit it with a command-line editor such as `nano`, rather than transfer the file, edit it on your own computer, and then transfer the edited file back to the remote machine. (See Chapter 5 or Appendix 3 for a `nano` refresher.) Being comfortable with a command-line editor when logged in via `ssh` will be an important skill for your remote computing.

Transferring files between computers

Once you are able to use the command line on a remote machine, you will probably want to get some work done. First, however, you have to be able to transfer data and analysis files between your computer and the remote computer. There are several options. For the most part, these options are interchangeable, but you may find that some are more convenient than others for your particular situation.

File archiving and compression

You will often want to transfer a collection of files rather than one single file. Not all file transfer tools can be used to transfer directories of files. You could transfer the files one by one, but this quickly gets tedious as the number of files grows. If the tools are available on your server, it is often easiest to use the `zip` command to merge the files into a single archive and compress them before transferring them. Try the `man zip` command to see some of the compression options. If `zip` is not an option, then a solution that is assured to be present on all Unix servers is creating a **tarball**. You can combine multiple files into a single file archive with the command `tar`:

```
myhost:~ lucy$ tar -cf ~/Desktop/scripts_25Jun2011.tar ~/scripts
```

The example above creates a file archive called `scripts_25Jun2011.tar` in the `Desktop` folder. The `-cf` argument specifies that you want to create an archive with the name `~/Desktop/scripts_25Jun2011.tar`. (By convention, archive files are given the extension `.tar`.) The files and directories you would like to archive are listed last; there can be one or more entries, separated by spaces. In this example, the directory being archived is your `scripts` folder.

File archives created with `tar` contain all the data in the original files; the files are just joined together into one big file. Making an archive with the command above leaves the original files in place, exactly as they were. It is good practice to include the date in the name of the archive. This makes it easy to understand at a glance what the archive contains, and allows you to store archives of the same files in the same folder without any name conflicts.

A `tar` archive is roughly the same size as the total size of the files included in it. To save Internet bandwidth and hard drive space, archives are often compressed with the command-line program `gzip`. Applying `gzip` compression to ordinary files produces files with a `.gz` or `.Z` file extension; applying it to a `tar` archive typically produces a file with a combined file extension of `.tar.gz` or `.tgz`.

The following command will compress the archive you created above:

```
myhost:~ lucy$ gzip ~/Desktop/scripts_25Jun2011.tar
```

This creates a new file on the Desktop, `scripts_25Jun2011.tar.gz`, and deletes the original uncompressed file. The compressed file will typically be much smaller than the original file.

The `gzip` command has a counterpart, `gunzip`, for uncompressing whatever has been compressed. Likewise, the `tar` command has arguments which allow an archive file to be expanded back into its constituent files and directories. So it is quite common to use these two commands in sequence to get the original files back out of a compressed archive:

```
myhost:~ lucy$ cd ~/Desktop
myhost:~ lucy$ gunzip scripts_25Jun2011.tar.gz
myhost:~ lucy$ tar -xvf scripts_25Jun2011.tar
```

The `tar` argument `-xvf` specifies that you want to expand the indicated file archive, which in this case is `scripts_25Jun2011.tar`. Once expanded, the directory and file names will be the same as before they were archived. (That is why we created the archive on your Desktop: we didn't want to clobber your original scripts directory). When restoring files, the archive is not deleted after it has been expanded.

Together, `tar`, `gzip`, and `gunzip` constitute a powerful trilogy of programs for storing and compressing all types of files, one you will likely find useful in many situations. You may frequently use these programs before transferring data to and from servers. Many data and software files you download from the Internet will be gzipped tar files, and you will use `gunzip` and `tar` to expand them. Compressed archives are also a convenient format for sharing collections of files with colleagues.

File transfer with `sftp`

The command-line program `sftp` (**s**ecure **f**ile **t**ransfer **p**rogram) can be used to transfer files over an `ssh` connection, via Secure File Transfer Protocol, or SFTP. Because it uses an `ssh` connection, the transfer is encrypted, you don't need to enable other types of connections to the remote machine, and it can be used anywhere

you have ssh access. Opening an sftp connection is similar to running ssh, but once connected, a link is maintained between the local and remote file systems, allowing you to use the commands get and put to move files back and forth. To transfer files, move to the directory on your local machine where you want to send or receive files. Then connect to the remote server with the command:

```
myhost:~ lucy$ sftp lucy@practicalcomputing.org
Connecting to practicalcomputing.org...
lucy@practicalcomputing.org's password: ←Type password
sftp> ls
```

Once connected, you will have a new command-line prompt, provided by the sftp program. The most common commands for sftp are listed in Table 20.1. The get and put commands can take wildcards such as *.txt as part of the file-name descriptors. If you get an error when trying to transfer several files, try the command mput or mget instead, which is set up for transferring multiple files.

Copying files with scp

It can be confusing to use a command line to simultaneously navigate through remote and local file systems, so you may prefer using the secure remote copy command scp. This functions just like the cp command, except that instead of just providing paths, you provide a username and a server name as well. The basic format for downloading a file is:

```
scp user@hostname:directory/remotefile localfile
```

TABLE 20.1 Some common commands when using sftp

Command	Usage
cd	Change directories on the remote machine
get A B	Download file A from remote machine and save as B locally
put B A	Upload file B from the local machine to the remote file A
lcd	Change directories on the local machine
!ls	List the files on the local machine
!command	Perform the specified shell *command* on the local machine
exit	Close the sftp connection

For uploading, you just specify the hostname as part of the second argument:

```
scp localfile.txt user@hostname:remotefile.txt
```

Here you can move to the appropriate local directory before running the command, so that you don't have to type the full path to the localfile. You will however have to specify the path to the file on the remote system, giving the directory name relative to your home directory on that system.

Other file transfer programs using SFTP

An even simpler SFTP option is to use a GUI program to handle file transfers and renaming, allowing you to drag files as you might in the Finder. You can try the shareware programs Cyberduck or Fetch, or the commercial program Transmit. All of these will present a window that will let you drag files to and from your remote machine. In Linux, you can get equivalent capability using FileZilla, which is available for Windows and OS X as well.

Other file sharing protocols

Many operating systems have built-in file sharing protocols which let you connect to a remote computer as though it were a disk drive. In OS X, you can use the Finder menu bar (or ⌘K) to select Go ▸ Connect to Server and enter the address of the remote computer. Windows computers can be accessed using the prefix `cifs://`, and other Macs can be accessed using `afp://` followed by the `username@servername` format used with `scp`. The Connect to Server dialog box can also be used for VNC connections, as described in the next section.

Full GUI control of a remote computer with VNC

Although we are focusing on command-line interaction with remote computers, it is possible to access the complete graphical user interface of a remote computer using Virtual Network Computing, or VNC. While `ssh` enables you to work as if you were looking directly at a terminal window on the remote computer, VNC enables you to work as if you were not only looking at the entire screen of that remote computer, but physically sitting in front of it and able to use its keyboard and mouse. This extra capability can be useful for modifying system settings, or for running programs unavailable at the command line. It can also be used to access your own desktop system while traveling.

VNC has a few limitations. It requires much more network bandwidth than `ssh`, and if there is a slow connection between the computers it may be so sluggish as to be entirely unusable. Some implementations of VNC do not encrypt data before it travels over the network, meaning that someone could eavesdrop on your activities and even control your computer. (One of the authors lost several weeks' worth of work when a computer was hacked in this way.) Be sure you have encryption enabled. Also, some remote servers, particularly large clusters, don't

have a GUI installed in the first place—and VNC can't provide access to a remote GUI that isn't there.

There are two components to a VNC set-up: VNC server software running on the remote computer, and VNC client software on your own computer that lets you interact with the remote server. Such software is available for all platforms, even including smartphones. In OS X, VNC capability is built into the file sharing system. You can set up your computer as a VNC server that supports incoming VNC connections using the Sharing panel in System Preferences, enabling either Screen Sharing to allow people to connect to your computer, or Remote Management to enable screen sharing in addition to other remote login options. You must set up a password for access at the same time, to keep people from controlling your computer without your permission. For Windows, RealVNC offers both a server and a client. For Linux, VNC server capability is also available, but to configure it securely follow the instructions at `help.ubuntu.com/community/VNC`.

Starting up a VNC server opens port 5900 (a portion of your network connection to the outside world) for traffic. Once you have turned on the VNC server on a computer, you will be able to connect to it using its IP number or address. Before you try connecting from an outside client, you can test whether the server is set up correctly by visiting this address in a web browser:

```
http://www.realvnc.com/cgi-bin/nettest.cgi
```

It will tell you what your IP address is, and will try to send traffic to you through port 5900.

To view the screen of a remote computer running a VNC server, you access the connection as a client. VNC client software is available in OS X without downloading additional software, although you can also download and install a third-party program such as Chicken of the VNC. To connect using OS X's built-in client, use the Finder to select Go ▸ Connect to Server (or ⌘K); in the dialog box which appears, type the prefix `vnc://` followed by the address of the remote machine. Although you can mouse around and otherwise operate a remote computer with VNC, you can't typically use it to transfer files.

Troubleshooting remote connections

Getting local with a Virtual Private Network (VPN)

Some services (e.g., access to an institutional library's Web site) and some network connections (e.g., `ssh`) block computers attempting to connect from outside the local network. Because every local network uses a particular range of IP addresses, your IP address off-campus will be different than the one you have on-campus. If you are off-campus, servers can see this and deny you access, if configured to do so.

A way around this is to use a virtual private network, or VPN.[2] Upon verification of your credentials, a VPN connection will pass your network traffic through

[2]VPN and VNC are often confused with each other in the heat of battle, but they are very different network tools.

a computer on the local network, effectively giving you an on-campus IP number. This will allow to access resources otherwise forbidden to computers off campus. VPN has the added benefit of being encrypted. If you are in a coffee shop using an unsecure WiFi connection, you can fire up the VPN to encrypt all Internet traffic between you and the VPN server on campus. This will prevent anyone from "sniffing" your passwords and other data over WiFi.

On OS X, setup a new VPN connection in the Network panel in System Preferences. Click the + below the list of settings, and choose VPN from the pop-up list. For VPN Type, you can try selecting PPTP and typing `vpn.` before your institution's domain name (e.g., `vpn.yourschool.edu`), but if that doesn't work, you will have to search your computer support pages or contact a system administrator to get the exact settings to use. Some VPNs require that you install special software.

Mapping network connections with `traceroute`

A useful command for diagnosing your inability to connect to a remote site—whether you are attempting an `ssh` connection or accessing the site via the Web—is to trace the network steps between your system and that address. This can be accomplished with the `traceroute` command. Simply type `traceroute` *`remoteaddress`* in a local terminal window, and you will see the hops between you and the server at your remote study site on a tropical island a hemisphere away. Test this by tracing the route between yourself and some of your favorite Web sites.

Configuring the backspace key

Your backspace key may not behave as expected when you are logged into some servers. Why? Because as with end-of-line characters, two different characters have been historically used to indicate a deletion or backspace. If you see `^H` (ctrl H) whenever you try to delete something at the command line, try changing the backspace character using the `stty` command.

Begin by typing:

```
stty erase '
```

After the single quote, press the backspace key on your keyboard. Instead of deleting the quote, it should cause the terminal to display the `^H` or `^?` (a single keypress will cause these two characters to appear) that you were getting previously. Close the string with another single quote mark and press return. Your ability to backspace should be restored.

To make this change permanent on the remote machine, use `nano` to edit your `.bash_profile` (or whatever the appropriate configuration file happens to be, if `bash` is not the default shell on the remote machine). Add the line:

```
stty erase '^H'
```

To type the ctrl H within this file as you edit it, you will have to use the trick of typing ctrl V first before the backspace key. Remember from Chapter 16 that this forces a literal interpretation of the next key typed, whether a return, backspace, or esc.

Two other features are sometimes missing on remote machines. The first of these is tab auto-complete at the command line, which is sometimes disabled on servers. Like the backspace, enabling it requires shell-specific modifications. The second is the use of the ↑ key to step back through your command history; this is controlled by the readline library, and it may be missing entirely or simply not yet configured. Search for the version of the system software you are accessing, or contact your system administrator to get these features configured to your liking.

Controlling how programs run

In most cases when using a remote server, you will transfer files to the remote machine and then launch computationally intensive analysis programs. How you launch and manage programs on the remote machine will depend on the type of computer you are accessing, who manages it, how many people are using it, what the system policy is, and how it is configured. Remotely managing analyses on a dedicated lab desktop computer is different than launching programs on a large cluster with many users, where special job management software ensures fair access to the processors and keeps different analyses from interfering with each other.

In this section, we describe tools for managing how programs run on regular computers—in other words, computers that aren't large shared clusters. Tools more relevant to large-scale analyses are described in the next section. Some of the tools addressed here will be helpful even for controlling how programs run on your own computer, so they really aren't specific to remote access. We then follow up with a brief overview of running analyses on clusters. It is difficult in this context to provide anything more than general advice, since there are so many different types of clusters.

To help demonstrate how to play with operations on a regular computer, we will use the `sleep` command as a stand-in for an analysis program. This program does nothing for the specified number of seconds, then stops. Try it out by typing:

```
host:~ lucy$ sleep 15
ls
```

Notice that when the `sleep` command is running, you don't have a shell prompt, and you lose the ability to run other commands until the operation is complete. (For example, type `ls` while you are waiting.) Any commands you type while `sleep` is running will be run after `sleep` quits.

Terminating a process

A running program is referred to as a **process**. The ability to terminate a process is an important skill for remote operations. This is because you may not be able to physically access that machine if something goes wrong, yet will still need to be able to stop a crashed program or a faulty analysis that is taking up valuable computing resources.

We have mentioned ctrl C a few times in passing. This is the interrupt command. It tries to terminate a program that is active at the command line. You type it by holding the ctrl key while pressing the C key. Because ^ represents ctrl in shell operations, this command is frequently abbreviated ^C.

Try typing the `sleep` command below and then pressing return:

```
host:~ lucy$ sleep 1000
```

This `sleep` command, if left undisturbed, would stop after 1000 seconds. To interrupt the running process before then, type ctrl C and you will get your shell prompt back. Some programs—for example, `nano`, and of course anything on Windows—interpret ctrl C differently. It isn't a universal command, but it is very widespread.

Starting jobs in the background with `&`

Usually when you run a program, you lose the command line for as long as the program will take to execute. However, if you type an ampersand (`&`) by itself at the end of the command line before pressing return, the program will run in the background, and you will get your shell prompt back immediately. Although these backgrounded commands are running unattended, they would cease operation if you were to close the window.

Try a `sleep` command with `&` at the end of the line:

```
host:~ lucy$ sleep 15 &
[1] 4990
host:~ lucy$
```

When you background a task like this, the prompt is immediately available, but the program is still running in the background (in this case, with job number `[1]` and the unique process ID number 4990) until the 15 seconds have passed. This is very different from terminating a process, in which case the process isn't running anymore at all. You can't use `&` to send a process to the background after it has begun, only at the time you start it.

Checking job status with `ps` *and* `top`

To check the status of all programs that are running, including those in the background, you can use the commands `ps` and `top`. The `ps` program gives a snapshot

of current processes. Try entering your `sleep` command as above, and then at the command line, run `ps`:

```
host:~ lucy$ sleep 15 &
[2] 4992
host:~ lucy$ ps
 4800 ttys001    0:00.02 -bash
 4992 ttys001    0:00.00 sleep 15
```

First you are informed of the process identifier, or PID, for the `sleep` process that was put into the background. Every running process has a unique PID that can be used for job control operations. The `ps` command then tells you the PID for the processes that are running in the current shell. One of these is `sleep`, and the other is the `bash` shell itself. This list is not a full list of everything your computer is doing. To see that list, use the command:

Other systems may use lower-case –a.

```
ps -A
```

The output of this command will likely be cut off at the right edge, since many of the program names are quite long. This command shows all the processes that are running, including all the housekeeping utilities that do things like listen for when you plug in a new mouse and check periodically for new network connections. Even the Finder and your graphical programs are in this list; this may seem strange at first, but they are processes running on your computer as well.

To find a certain process in the long list, you can pipe the output of `ps -A` to a `grep` command to look for the name in the output stream:

```
ps -A | grep Finder
```

This should show you just one line of output with information for that program.[3] This command is most often used to find runaway processes that you might want to stop, using the `kill` command described later in this section.

The `top` program provides a real-time (continuously updating) view of the same list of processes as `ps`. To see a basic list of processes, sorted by PID, just type `top` by itself. You can quit the `top` display by typing `q`, as for the program `less`. To see a list sorted reverse-alphabetically by command names, use the `-o` modifier and specify that you want to sort by command name:

```
top -o command
```

Other options for `top` are shown in Table 20.2.

[3] In Linux or Cygwin, you won't have a Finder program, but the command should work the same with other program names.

Open a second terminal window, and place it to the right of the window running the above command. Now you can use the left window to watch your activities in the right window. Rerun the `sleep 15` command in the right terminal window. You should see the `sleep` program listed alphabetically in the top window for 15 seconds before it eventually finishes and disappears.

In addition to the basic list of processes, `top` provides information on how much memory and Central Processing Unit (CPU) power each process is taking. This is a convenient way to keep an eye on computationally intensive programs and quickly get a sense of system status. If you use `ssh` to log into a shared lab computer to start an analysis, the first command you should run is `top` to make sure that someone else isn't already using all the processors for other analyses. If your own computer is getting hot for no apparent reason, you can run `top -u` and see what the cause might be.[4] We will show you in a bit how to terminate these processes.

TABLE 20.2 Options for the `top` command

`top -o`	Sort by the parameter that follows; default is `pid`, but with `-o`, command will sort by program or process name
`top -u`	Sort by CPU usage; useful for detecting runaway processes
`?`	While `top` is running, show a list of display options
`q`	While `top` is running, quit

Suspending jobs and sending them to the background

If you run a command without the final `&` and only later realize that you want to send it to the background, you can first suspend the operation by typing ctrl Z. This is different from ctrl C in that the program is merely frozen, not terminated. It is also distinct from putting a job into the background in that no operations are proceeding. Try this out with the `sleep` command, set for a relatively short duration, followed by ctrl Z:

```
host:~ lucy$ sleep 15
^Z
[1]+  Stopped                 sleep 15
host:~ lucy$ ps
  PID TTY           TIME CMD
 5760 ttys001    0:00.02 -bash
 5954 ttys001    0:00.00 sleep 15   ← This job is stopped, not just backgrounded
```

[4] Ahem, Flash content?

The output looks very similar to what you got when running `ps` after you launched `sleep 15 &`. In both cases the process is listed along with its PID, and you are able to interact with the shell even though sleep was started and has not finished. However, the big difference is that if you keep checking `ps` for more than 15 seconds, the process will still be listed; it is suspended and not running to completion in the background.

To list all the background processes, use the `jobs` command. First add another `sleep` operation so that it is running in the background:

```
host:~ lucy$ sleep 20 &
[2] 5989
host:~ lucy$ jobs
[1]+  Stopped                 sleep 15
[2]-  Running                 sleep 20 &
```

Notice that our new operation gets job number `[2]`, and that it is listed as `Running` while the first `sleep` job that was suspended with ctrl Z is listed as `Stopped`. To resume job `[1]` (the stopped `sleep` command in the background), use the `bg` command followed by the job number:

```
haeckelia:~ haddock$ bg 1
[1]+ sleep 15 &
[2]   Done                    sleep 20
```

Our updated job list now shows `sleep 15` now running in the background, while job `[2]` has already finished.

There is also a `fg` command that will take a suspended job and move it into the foreground. You can use this combination of ctrl Z, `jobs`, `bg`, and `fg` to move processes between active and inactive states and between the background and foreground.

Stopping processes with `kill`

Once you send a job into the background with `&` or `bg`, it will be impervious to interruption by ctrl C. This is because keyboard input can only be passed to programs that are in the foreground. You will also not be able to interrupt it by normal means. The command to terminate any process is suitably named `kill`. This operates using the PID as displayed by the `ps` and `top` commands (aha!).

To test your powers, use a background `sleep` process, but this time set it to go for 60 seconds. Then use `ps` or `top` to find out the PID it has been assigned. Finally, `kill` the process, and then check again with `ps` to see that it is indeed gone:

```
host:~ lucy$ sleep 60 &
[1] 5563
host:~ lucy$ ps
  PID TTY           TIME CMD
 5089 ttys000    0:00.06 -bash
 5563 ttys000    0:00.00 sleep 60   ← The first number is the PID
host:~ lucy$ kill 5563
host:~ lucy$ ps
  PID TTY           TIME CMD
 5089 ttys000    0:00.06 -bash
[1]+  Terminated              sleep 60
host:~ lucy$
```

On some occasions, a recalcitrant program may not be terminated by the `kill` command. When this happens, the way to force it to quit is to use the top secret command `kill -9`. If the process still can't be terminated because you don't have permission to interfere with it, you can also override this with `sudo`. The combined effect is lethal:

```
sudo kill -9 5563
```

That will stop just about any process, so use restraint (especially when the processes belong to other people on a shared computer).

There is another `kill` command, `killall`, that lets you interrupt programs by name, rather than by PID. This includes forcing termination of GUI programs that are running in a Finder window. This command too must be used with caution because it will kill every process whose name matches. For example, if you run several `sleep 30 &` commands in a row, a single `killall sleep` command will terminate all of them. If you are running OS X, see what happens when you type this:

```
killall Dock
```

Under certain circumstances, the `kill` command can save you from losing data or unsaved work, even when working in a GUI and not via the terminal. When a computer freezes up due to a certain program, such as a web browser, you are often still able to log into that computer remotely using `ssh`, as long as it was set up to do so in the first place. Once connected, you can then use `top -u` to find the PID or name of the offending process (look for the CPU hog or the name of the program you know is frozen). The `kill` command issued from this remote machine should be able to free up the system and let you save work in other programs that were running at the time.

Keeping jobs alive with `nohup`

When you terminate a shell by closing a terminal window or logging out of an `ssh` session, all the programs running in the shell stop too. This happens even if the processes are in the background. It means that when you connect to a remote server to start a program, the program will stop when you log out. If the program is likely to take hours, days, or even weeks to complete, needing to keep your personal computer continuously connected to the server for that entire time would be a huge inconvenience.

To get around processes terminating when the shell closes, you can insert the `nohup` command (**no h**ang-**up**) before the name of your program when you launch it. This indicates that the program should keep running even when the shell used to start it is long gone.

Using `nohup` is only one of the steps you need to assure that your program keeps running. If your program provides output or error messages along the way, when they are generated but their destination (your terminal window) is no longer present, that can cause the job to cease. We have not often asked you to use a command without explanation, but describing the inner workings of the full command we are about to present is beyond the scope of this book. If you find it useful, and it works, then you can just take it on faith that there are reasons for the strange syntax that follows.

Imagine that your program would normally be run with the following command, redirecting the output to the file `log.txt`:

```
phrap infile.sff -new_ace > log.txt
```

To rephrase that so it will run unattended, insert your command between the bolded code like this:

```
nohup phrap infile.sff -new_ace > log.txt 2> /dev/null < /dev/null &
```

The initial `nohup` command lets the program run without interaction after you log out. The rest of the text following `2>` sends any error messages to an imaginary place called `/dev/null`—essentially redirecting them to nowhere. Lastly, the ampersand at the end causes the job to run in the background.

It is often the case that a process takes much longer than anticipated. If you started it on a remote computer and expected it to take only a few minutes, you probably wouldn't have launched it with `nohup`. If it were still running several hours later when you needed to pack up your computer to go home, you might not want to terminate the process and lose all the progress it has made.

Fortunately, there is a way to modify a process after it has started so that it won't terminate when the shell closes. First, suspend the process with ctrl Z, and send it to the background using `bg 1` (or whatever actual job number is reported).

Although the process is in the background, it still belongs to the shell and will close when the shell does. To free it from the shell so that it doesn't close when the shell does, run the `disown -h` command. You should probably test this command before leaving an important job for a long time: if the program is producing output and wasn't invoked in the output-suppressing manner of `nohup` above, then it may still terminate when you log out.

Changing program priority with `renice`

Unless an analysis program is limited by other factors, such as hard disk speed or RAM, it will use all the processor power it can until it has completed. Many programs can only use one processor at a time, regardless of how many processors the computer has, meaning that they will max out a single processor. Other programs are designed and configured to use multiple processors at a time. In either case, if there are multiple programs running at the same time, they can start to compete for resources. When the computer is saturated with running programs, each new program that is started will slow down the rest.

However, usually some processes are more important than others. This being the case, you can adjust the priority of each process with `renice` so that it gets a larger or smaller fraction of processor power than other programs. The `renice` command takes two arguments: the priority you want to assign to a process, and the process's PID, which you can get with `ps`. The priority can be an integer running from `-20`, the highest priority, to `19`, the lowest priority. Processes are assigned a priority of 0 by default.

A typical `renice` event looks like the following:

```
lucy$ ps
 PID TTY           TIME CMD
29932 ttys000  0:00.01 -bash
29938 ttys000  0:00.06 raxmlHPC -b 13 -# 100 -s alg -m GTRCAT -n TEST
lucy$ renice 19 29938
```

In this case, the `raxmlHPC` process is reduced to the lowest priority.

If you are running long analyses on all the processors of a computer that is intermittently used for other purposes, such as taking pictures at a microscope, you should `renice` the analyses to the lowest priority, `19`. The analyses will then take all the computer power they can when the processor isn't being used for other tasks, but they will cede that power to any other programs that request it (unless other system resources are limiting).

High-performance computing

In some cases, the remote server you are connecting to will be a large cluster specifically designed for high-performance computing needs. Running software on these machines can be different than launching analyses on your own computer

or on a small server. This is because the analysis software must by configured and called in such a way that it can use multiple processors, with processes strictly controlled by job management tools that allocate the available resources to many different users.

Here we simply describe some of the issues at stake and mention some of the common tools used to address them. Consult the system administrator of the cluster you are using for details specific to your system.

Parallel programs

Some analysis tasks can only be run on one processor at a time. This is because each calculation requires the result of the calculation that came before it, so spreading the task across processors wouldn't speed anything up. However, many analyses can be divided up into pieces to be analyzed in parallel on multiple processors. Each piece might be an independent replicate in a statistical analysis, or a single part of a very large summation. Tackling a problem with multiple processors requires more than the problem being of the sort that can take advantage of multiple processors: the software itself must be written to use multiple processors.

In the simplest case, an analysis involves going line by line through an input file and performing some calculation that depends only on the contents of that line. This analysis can be made parallel by the poor man's strategy of just breaking the input file up into chunks, analyzing them separately, and then combining the results. You can even do this by hand. If you have four microprocessor cores on your computer, for example, you could break the input into four equal parts, launch four analysis processes (one on each part of the input data, using an ampersand at the end of each command to get the prompt back), and then combine all the result files.

In most cases, parallel analyses are more complicated, and require extensive communication between the processors to coordinate calculations and exchange data. Rather than write software from scratch to handle all these complicated tasks, parallel data analysis programs usually draw on third-party software. The reason this is relevant here is that many software tools that would be used for large analyses on remote computer clusters fall into this category, and it affects the way the analyses are started. Rather than call the analysis program directly, it may be necessary to call the parallel software and tell it the name of the program you want to run, including all the relevant parameters for your analysis. It must be emphasized that these tools cannot magically turn any analysis into a parallel analysis; the analysis software must have been designed from the ground up to be compatible with and use these tools.

One of the most widespread software tools for facilitating parallel analyses is the Open MPI library (`www.open-mpi.org`). To launch a parallel analysis with Open MPI, you call the `mpirun` or `mpiexec` command, depending on how your system is configured. It is possible to install and use Open MPI on your own com-

puter if it has multiple processors and you have programs that can use it. Other such technologies include Pthreads and the multithreading libraries in Boost C++.

Job management tools on clusters

If everyone on a large shared cluster were free to log in and start software however they liked, pandemonium would ensue and nothing would get done. Job management tools provide a way to structure how analyses are run and resources are allocated. If you are running your analysis with a job manager, you will not need the general commands for controlling programs described above (e.g., `renice`, `nohup`, and `bg`). The job manager takes care of all these issues, and using these other commands could actually interfere with your analyses as well as those of others.

When you log into a remote cluster with `ssh` you will have access to many, or even all, of the standard command-line tools you are already familiar with. While it is possible to start programs as you would on your own computer by typing their name and specifying the required arguments, it is standard practice that they will be terminated if they take too long or use too many resources. It is fine to run `gunzip` for a couple of minutes to uncompress your data files, but it isn't okay to call `phrap` at the command line for a five-day genome assembly. Instead, you need to prepare a configuration file for each analysis and submit it to the job manager.

This configuration file is often a shell script that includes supplemental commands to be read by the job manager. There are variables that determine how many processor cores the analysis needs and how these should be distributed across computers. The file sets up log files, and arranges other housekeeping. You then add one or more lines with the commands for the analyses you want to do.

Once the configuration file is ready, you submit it to the job manager. This is akin to getting in line. If there are enough free resources to start your job, it will begin immediately. Otherwise it will wait until other analyses finish.

Two of the most common job managers on academic clusters are Portable Batch System, also known as PBS, and Oracle Grid Engine, previously known as Sun Grid Engine and still commonly called SGE. Each has its own format for configuration files, as well as different commands for managing jobs. If you are using a cluster, ask the system administrator for an example job submission script that you can modify for your own analyses, and for the appropriate commands for submitting and monitoring jobs.

Setting up a server

Up until this point, this chapter has focused on how to connect to and use a remote server. In some cases, though, you will want to build your own server. You may want to configure your personal computer as a server to test and learn particular software packages, or you may want to configure a lab desktop as a server to run analyses or provide remote access to instrument data.

Building a server isn't necessarily as daunting as it might sound. Remember that a server is just a computer that accepts requests for a particular service and then delivers that service. The main tasks in setting up a server are as follows:

- Install and configure server software for the service you want to provide, i.e., `ssh` access or a web server. Some of the most common server software, such as that for serving web pages and `ssh` access, is installed by default on many operating systems.
- Configure the firewall on the server to allow access to the service you will provide, and to prevent access for services you are not using. A firewall provides an added level of security against attacks aimed at unused services, but can interfere with proper function of active services if not configured.
- Make sure your local network (i.e., the network in your building or on your campus) allows connections to computers from the outside world. Most local networks have firewalls that protect the entire network, just as each computer's firewall protects that computer. This provides an added level of security, but if all incoming requests are forbidden, you will need to work with your network administrator to allow remote access for particular services on your computer. In extreme cases, institutional policy may forbid remote access to all machines under all circumstances.
- Get the IP address or hostname of your server so that other computers can find it.

Building a server can consist of just configuring and turning on an already installed piece of software. In other cases, things may be a bit more complicated. You may need to install software or work with your network administrator to configure access to the server. Below we walk through some of these tasks in greater detail, focusing on `ssh` access. The material is relevant to configuring other types of servers, including web servers.

The `ssh` server is not installed by default on Ubuntu, but `openssh-server` can be added with the Synaptic Package Manager.

Configuring the `ssh` *server*

Before you enable remote `ssh` login, make sure that you have chosen a login password for your computer, and that this password is not something that can be easily guessed. This same password will be used to authenticate `ssh` access. A weak password is a serious security vulnerability.

OS X comes with the `ssh` server software already installed. To activate it, open System Preferences and select the Sharing pane. Click the Remote Login checkbox. This will start the `ssh` server software and enable remote `ssh` connections.

The OS X firewall is configured via the Security pane in System Preferences. If your firewall is running, any service enabled in the Sharing pane will automatically be added to the list of permitted services to which the firewall allows access.

Finding your addresses

In order to connect to a computer, you will need to know its address. During your shell session, the hostname of your computer is stored as the system variable `HOSTNAME`. To retrieve this name, simply type `echo $HOSTNAME`. To find your IP address, use this variable as input to the `host` command:

```
myhost:~ lucy$ host $HOSTNAME
myhost.practicalcomputing.org has address 100.200.10.20
```

There are other ways to find your hostname. You can use the shell command `ifconfig`:

```
myhost:~ lucy$ ifconfig
...
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    inet6 fe80::11a:bbff:33bb:aa11%en1 prefixlen 64 scopeid 0x5
    inet 100.200.10.20 netmask 0xffffff00 broadcast 100.200.10.255
    ether 00:11:22:bb:cc:ee
...
```

Because you probably have several kinds of connection options on your computer (Ethernet cable and wireless card, for example) this command typically returns a lot more information than just your IP address, meaning you have to dig through the results. The cabled network address should be found in the lines below the `en0:` entry, next to where it says `inet`. A WiFi address will be listed in the lines following the `en1:` entry. (Unfortunately, the addresses do not appear on the same lines as the `en0` entry, so they are not easily obtained by piping through a `grep` command.)

In our fictitious example, the IP number for the system is 100.200.10.20. This number may change each time you connect to the network, because your local system may dynamically assign you a new number. (You will hear this called DHCP, which stands for Dynamic Host Configuration Protocol.) This number may change every day or so, as well as every time your computer is unconnected and then reattached to the network. This is less than ideal since the number may change while you are away, meaning you will have no idea what the IP address is when you try to log in remotely. There are a couple of ways around this. First, you can use your system name as returned by `$HOSTNAME`. This is more stable than a dynamic IP address. Second, you can make a request to your network administrator for a fixed IP address. A fixed IP address is dedicated to a particular computer and does not change, unlike an address assigned by DHCP.

There is another potential address complication for remote connections. Some routers, including most WiFi base stations, are configured so that a single IP address is assigned to the router, with the router then setting up a subnetwork that

shares this connection among all the computers. The outside world sees only one machine with one address (the router) while the router assigns each computer on the subnetwork its own IP address that only works within that subnetwork. This means that the IP address of the computer on the subnetwork is not the same as the IP address the computer has on the Internet. You can recognize subnet addresses because they usually take the form `10.1.X.X`, `172.X.X.X`, or `192.168.X.X`. In order to remotely connect to a computer on a subnetwork, you will need to specially configure the router to use a technique called **port forwarding**. Search the Internet for that phrase along with the brand and model number of your router hardware to find the relevant instructions.

There are a variety of Web sites that will tell you what your IP address is on the Internet. These include `whatismyipaddress.com`. If your Internet IP address reported differs from the IP address shown by `ifconfig`, it is because the Internet IP address is assigned to your router and the IP address indicated by `ifconfig` is your IP address on the subnetwork.

Connecting to your own computer with `ssh`

Once the `ssh` server software is running, you can also connect *to* your own computer, *from* your own computer. When you do this, your computer is acting as both the server and the client. Connecting to your own computer works even if you aren't connected to a network. This is a bit pointless, in the sense that you are already sitting at your own computer, meaning you don't need `ssh` to gain access to it. However, it provides an opportunity to get experience with `ssh` when you don't have access to a server. It also comes in handy when you are having trouble with `ssh` and want to test parts of your connection to isolate the problem.

Once you have configured your computer as an `ssh` server as described above, try connecting to it using `localhost` as the address, your own username in place of `lucy`, and the same password that you use when logging in to your computer at startup:

```
ssh lucy@localhost
```

After you press return and enter your password, you will see the same greeting that you get when you first open a terminal window. This is because `ssh` starts a new shell when it logs into your computer. You can navigate your system and enter commands just as you would if you had not used `ssh`. To leave this `ssh` session, type `exit`, and you will get back to your original login prompt.

SUMMARY

You have learned:

- That computers are found on the Internet with hostnames and IP addresses
- The distinction between clients and servers
- How to gain command-line access to remote machines with `ssh`
- How to package and compress files using `zip` or `tar`, `gzip`, and `gunzip`
- How to transfer files to and from remote computers with `sftp`, `scp`, or graphical interfaces
- How to manage programs using ctrl Z, `bg`, `fg`, and the ampersand
- How to terminate processes with ctrl C, `kill`, or `sudo kill -9`
- How to determine process identities and CPU usage with `ps` and `top`
- How to keep programs running when the shell closes with `nohup` and `disown`
- Some of the major issues involved with running large parallel analyses on remote, high-performance clusters
- The steps needed to set up a standard computer as a server, with a focus on configuring an `ssh` server