Basic Read Alignment

Dan MacLean

2020-01-14

Contents

1 About this course **5**
1.1 Prerequisites 5

2 The Alignment Pipeline **9**
2.1 Align all reads to a reference 9
2.2 Filter out badly scoring reads 9
2.3 Sort and compress output 10
2.4 Further Reading 10

3 Running `minimap2` **11**
3.1 The `minimap2` command and options 11
3.2 Further Reading 12

4 Filtering Badly Aligned Reads **13**
4.1 SAM Format 13
4.2 `samtools` 14
4.3 The `samtools` command and options 14
4.4 Checking the filtering 14
4.5 Are we done? 15
4.6 Further Reading 15

5 Connecting Programs and Compressing output **17**
5.1 BAM Files 18
5.2 Connecting Program Input and Output With Pipes 18
5.3 From reads to filtered alignments in one step 19
5.4 Sorting BAM files 19
5.5 Indexing the sorted BAM 20
5.6 Further Reading 20

6 Automating The Process **21**
6.1 Shell scripts 21
6.2 Creating a script that automates our alignment pipeline. 22
6.3 Running the script 22
6.4 Running on different input files 22

7 Running an alignment on the HPC **25**
7.1 An HPC is a group of slave computers under control of a master computer 25
7.2 Logging into the submission node 27
7.3 Preparing a job 27
7.4 Submitting with `sbatch` 29
7.5 Checkout tasks 30
7.6 Further Reading 30

Chapter 1

About this course

In this short course we'll look at a method for running an alignment of sequence reads against a reference genome. The course is very brief and will show you how to use a program called `minimap2` and `samtools` to create a binary alignment file that you can use in further work.

I acknowledge that there are lots of other programs and methods - this course is *not* meant to be comprehensive, it is meant to get you being productive. Seek out further advice if you need to run other programs. Do be encouraged though, 99 % of what you learn here will be applicable to other tools for the same job (they all run in a *very* similar manner) so this is a good place to start.

The course has two main parts - first we'll learn to do this 'locally', that is to say on the computer that you are actually physically sitting at and have direct control over. Once we've done that and know how to run the actual programs then we shall switch to running the programs 'remotely' on a HPC environment using a submission system that you have to log in to.

1.1 Prerequisites

This course assumes that you are familiar with the basics of running stuff from a command-line. You'll need to have some experience not lots. If you've done the TSL command-line course you'll know plenty

For the first 'local' part of this you'll need some software on your machines. Most bioinformatics software has to run on Unix style computers, which for most of us means Macs. The installation instructions below only apply to Macs.

For the second part you'll need an HPC account. See the Bioinformatics Team to get one of these.

1.1.1 Local Software Installation

We need to install `minimap2` and `samtools`. Installing bioinformatics software is often *not* straightforward so we'll take a path of least resistance and install some tools that manage software for us. This is a roundabout way of doing things, but it greatly simplifies the hard parts and means that we can isolate our installations from the rest of our computer and not mess up anything already installed.

1. **Get `conda`** `conda` is an environment and package manager for software projects (initially Python - hence the name). Its purpose is to create a sandbox area on your computer where you can safely install software without it interfering or overwriting any of the existing software already on there. This safe space is called a conda 'environment'. To install `conda`:
    1. Click this link https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-x86_64.pkg and wait for the package to download. When it has double-click it and go through the installation process.
2. **Get `bioconda`** `bioconda` is plugin that makes `conda` aware of the bioinformatics software we will need. To make `conda` aware of `bioconda`
    1. Open `Terminal` and type the following:

```
conda config --add channels defaults
conda config --add channels bioconda
conda config --add channels conda-forge
```

3. **Create a new environment** We can now create our new safe space environment. Type in the following.

```
conda create --name alignment_env
```

This step can take a moment or two and nothing seems to happen for a while - be patient. Accept the defaults (answer `y` when questioned).

4. **Activate the environment** Everytime we want to use the new environment, including to install something into it, we have to activate it. That means each time we leave or start a new Terminal we need to type this into it

```
conda activate alignment_env
```

You should see the name at the `$` prompt change, telling you that this Terminal is in the `alignment_env` environment. If you ever can't find programs that you're sure you installed, it means that you probably didn't activate the right environment.

5. **Install the alignment software `minimap2` and `samtools`** All the steps up to now have been so we can do this one! Install each of the pieces of software in turn by using `conda` with:

```
conda install minimap2
```

```
conda install samtools
```

These steps can also take a while. Again accept the defaults (answer `y`).

Now you are done! Everything is installed ready for you to work with. Next we need to get the sample data

1.1.2 Sample reference genome and reads

You'll need this zip file of data: sample_data.zip which contains a reference genome and a set of paired end reads. Download it, extract the files and put them into a folder on your machine. I suggest something like `Desktop/align_tut`. This will be the directory we'll work from in the rest of the course.

That's all you need to do the lesson. If you have any problems getting this going, then ask someone in the Bioinformatics Team and we'll help.

Chapter 2

The Alignment Pipeline

In this chapter we'll look at an overview standard paired-end read run of `minimap2`, what it outputs and how to manipulate the output with `samtools`.

The overall pipeline is very straightforward…

1. Align all reads to a reference
2. Filter badly scoring reads
3. Sort and compress output

2.1 Align all reads to a reference

This is the main step, and with `minimap2` it can be accomplished with a single command-line. In this step each read is aligned against the reference, and its best aligning position found. That position, along with a metric of the quality of the single alignment is reported in a `SAM` format file.

2.2 Filter out badly scoring reads

This is the quality control step. We remove reads that don't have a good alignment score because in most contexts it means the read is a bad read with bad sequence in it. Of course in some contexts it isn't - it depends what you're aligning to what, but for the RNAseq situation or SNP calling situation where we expect the reads to be very like the reference this is appropriate. This step is done with `samtools`

2.3 Sort and compress output

Once filtering is done and we have the set of reads we wish to retain we can take our output file and convert it to a sorted binary format that uses less disk space and is optimised for searching in downstream analysis. This step is a kind of housekeeping step that makes everything later easier. We do it with `samtools`

2.4 Further Reading

2.4.1 FastQ quality scores

Typically the reads we use will be in the FastQ format, this is a two line format that encodes not only the sequence read, but the 'believability' of each base. You can see more here

2.4.2 Why minimap2 and not bwa|bowtie|other

You may have heard of other aligners and that people in your group are using them. `minimap2` is currently the quickest and most accurate of all the aligners and has general ability with all modern sequence types. Those using others (in particular bwa and bowtie) are pretty out of date. bwa *was* a favourite for a long time, but has been superceded. The same author created both bwa and minimap2 and minimap2 was written to address issues with bwa.

Chapter 3

Running `minimap2`

Running `minimap2` takes only one step. Assuming we've already `cd`'d into the directory with the reads and reference we can use this command

```
minimap2 -ax sr ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq > aln.sam
```

Try running that and see what happens… You should get an output file in the working directory called `aln.sam`. On my machine this takes just a few seconds to run.

Let's look at the command in detail.

3.1 The `minimap2` command and options

First we get this

```
minimap2
```

which is the name of the actual program we intend to run, so it isn't surprising that it comes first. The rest of the command are options (sometimes called arguments) telling the program how to behave and what it needs to know. Next up is this

```
-ax sr
```

which gives option `a` meaning print out SAM format data. And option `x` meaningwe wish to use a preset parameter set. The preset we wish to use comes after `x` and is `sr`, which stands for `short reads` and tells `minimap2` to use settings for short reads against a long genome. Next is this

```
ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq
```

which are the input files in the 'reference' 'left read' 'right read' order. Finally, we have

```
> aln.sam
```

which is the `>` output redirect operator and the name of an output file to write to. This bit specifies where the output goes.

So the structure of the `minimap2` command (like many other commands) is simply `program_name options input output`.

And this one command is all we need for a basic alignment with `minimap2`. We can now move on to the next step in the pipeline.

3.2 Further Reading

3.2.1 The > operator

The `>` symbol is actually not part of the `minimap2` command at all, it is a general shortcut that means something like 'catch the output from the process on the left and put it in the file on the right. `Think of the`>' as being a physical funnel catching the datastream! Because it's a general operator and not an option in a program, we can almost always use'>' to make output files. You'll see it pop up quite often

3.2.2 `minimap2` further instructions and github

The commands given here for `minimap2` are just a small selection of what are available. You can see the user guide at GitHub

Chapter 4

Filtering Badly Aligned Reads

Once we have an alignment, the next step is often to throw out the reads that align badly or not in pairs as we we expect. To do this we need to look at the alignments and assess them one-by-one. We'll need first to have some understanding of the output from our alignment, in this case `aln.sam` a SAM format file.

4.1 SAM Format

Alignments are generally stored in SAM format, a standard for describing how each read aligned one-by-one. Each line carries the results for a single read. Let's examine a single reads alignment. Recall that we can look at one line in a file called `aln.sam` using `tail -n 1 aln.sam` (this gives the bottom line in the file). Running this prints the following

```
NC_011750.1_1004492_1005000_1:0:0_3:0:0_1869f   147 NC_011750.1 1004931 33  70M =   1004492 -509
```

On close inspection we can see this mess (which is only a single line) contains things like the read name, the position it maps to on the reference sequence, the read sequence, and lots of other strange things like `70M` and `de:f:0.0429`. The important thing to note is that these weird things are encoded quality information for this alignment, so we can - if we know how to manipulate those codes - select read alignment of the proper quality.

Thankfully the program `samtools` makes this easy for us.

4.2 samtools

We can accomplish read filtering with the following command.

```
samtools view -S -h -q 25 -f 3 aln.sam > aln.filtered.sam
```

Try running that and looking at the output file that is generated. You should have another SAM format file called `aln.filtered.sam` in your working directory.

Let's take a look at that command in detail

4.3 The samtools command and options

Straight away, the command seems to fit the familiar `program name options files` pattern. It starts with

```
samtools
```

which is the program name. Then we get the options

```
view -S -h -q 25 -f 3
```

The first option to `samtools` must be the name of the sub-program to run. There are lots of these as `samtools` is a suite of sub-programs. `view` is the option for working with alignments directly. The second option `-S` tells `samtools view` that we are handing it a SAM format file (soon we will hand it a different type) and `-h` tells it to show the header as well (each SAM file has a header that we sometimes don't want). The next two options are the important ones. `-q 25` will remove reads with a mapping quality (a measure of how well a read is aligned) lower than 25 (a reasonable score) and `-f 3` is a 'flag' a really complex way of encoding alignment attributes (see Further Reading for more details). The important thing is that `3` means `keep reads that are paired and whose pair is mapped too`.

At the end of the command is the input and output file information

```
aln.sam > aln.filtered.sam
```

which means the input file is our `aln.sam` and that the output should be redirected to `aln.filtered.sam`

4.4 Checking the filtering

As an exercise to show that we did filter stuff out lets compare the input `aln.sam` file with the output `aln.filtered.sam` file. Recall that `wc -l` will give us the number of lines in a text file. Run it like this, on both files at once

```
wc -l aln.sam aln.filtered.sam
```

I get this as output

```
  200002 aln.sam
  166905 aln.filtered.sam
  366907 total
```

The number of lines (alignments) in the filtered files is less than that in the unfiltered, so we can casually assume the command worked.

And that's all there is to getting the reads filtered. In real-life you have many options for filtering and you may choose to do it at other points (for instance, lots of RNAseq quantification programs will allow you to filter when you use them), but the process will be similar and take advantage of the same mapping quality and flag metrics you've been introduced to here.

4.5 Are we done?

On the face of it then, it looks like we've come to the end of what we intended to do - we did an alignment, and we've filtered out the poor ones. In practice though, we'll be dealing with many millions of reads, many files of many Gb size. This complicates the housekeeping we have to do, not the procedure we've learned *per se*, so before we jump to the HPC we need to look at that. That's the next chapter.

4.6 Further Reading

4.6.1 SAM Format

I only really alluded to the SAM format above, but there's a lot to it. This Wikipedia page gives a lot of detail.

4.6.2 Mapping Quality

A metric that describes how well overall the read aligned, it takes into account not just the alignment, but the nubmer of other possible alignments that were rejected. Consider that a read mapping well equally at a number of places in the genome cannot be said to be mapping well at all. Different aligners make arbitrary decisions about how to score such alignments. See this short summary for information on how it can be calculated.

4.6.3 Flags

The flags option is the most powerful way to describe a filter to `samtools view`, it is also really complicated. The number you pass (e.g `-f 3`) is calculated as a sum of lots of options. The way they're are described in the documentation is a bit more complex than I want to go into, but there are helpful web-apps that can simplify things - try this one

Chapter 5

Connecting Programs and Compressing output

Now that we've been through the whole alignment and filtering pipeline, let's look at the output. Specifically lets compare the sizes of the files we used. Recall that we can do that with `ls -alh`

On my folder I get this (some columns and files removed for clarity)

```
49M 29 Nov 10:46 aln.filtered.sam
59M 28 Nov 16:28 aln.sam
5.0M  2 Jul 15:04 ecoli_genome.fa
18M 28 Nov 15:53 ecoli_left_R1.fq
18M 28 Nov 15:53 ecoli_right_R2.fq
```

The file sizes are in the left-most column. Check out the relative size of the two read files (18M each) and the alignment SAM files (59M and 49M). The output file is much larger than the input. This has implications for storage when the files are really large (many GB) and there are lots of them. The disk space gets used really quickly. Consider also the redundancy we have - that `aln.filtered.sam` is the one we're interested in, not the `aln.sam` so it is taking up unnecesary disk space. It's easy to see that when you are doing a real experiment with lots of samples and hundreds of GB file size, you're going to eat up disk space. Also larger files take longer to process, so you're going to have a long wait. This has implications too when you get to later stages in the analysis

In this chapter we're going to look at a technique for reducing those housekeeping overheads and speeding things up.

5.1 BAM Files

BAM files are a binary compressed version of SAM files. They contain identical information in a more computer friendly way. This means that people can't read it, but it is rare in practice that you'll directly read much of a SAM file with your own eyes. Let's look at the command to do that

```
samtools view -S -b aln.filtered.sam > aln.filtered.bam
```

Again we're using `samtools view` and our options are `-S` which means SAM format input and the new one is `-b` means BAM format output. Our input file is `aln.filtered` and we're sending the output to `aln.filtered.bam`.

If we check the files with `ls -alh` now we get

```
9.2M 29 Nov 14:05 aln.filtered.bam
49M 29 Nov 10:46 aln.filtered.sam
59M 28 Nov 16:28 aln.sam
5.0M  2 Jul 15:04 ecoli_genome.fa
18M 28 Nov 15:53 ecoli_left_R1.fq
18M 28 Nov 15:53 ecoli_right_R2.fq
```

The BAM file is about a fifth of the size of the SAM file. So we can save space in this way. We have another trick up our sleeve though. We can connect together command lines, so that we don't have to create intermediate files - this reduces the number of files we have to save. We can do this by using something called pipes.

5.2 Connecting Program Input and Output With Pipes

Most command line programs print their results straight out without sending it to a file. This seems strange, but it adds a lot of flexibility. If also set up our programs to read in this output then we can connect them together. We can do this with pipes. The usual way to do this is to use the `|` operator. Let's look at a common example.

Here we'll use the command `ls` and `shuf` to see how this works. We know `ls` will 'list' our directory contents, `shuf` shuffles lines of text sent to it. If we use `|` in between we can connect the output of one to the other. Try running `ls` a couple of times to verify you get the same output both times and then try this a few times

```
ls | shuf
```

you should get different output everytime. The important thing to note is that `shuf` is doing its job on the data sent from `ls`, which sends consistent data every

time. We don't have to create an intermediate file for `shuf` to work from. The `|` character joing two commands is the key.

We can apply this to our `minimap2` and `samtools` commands.

5.3 From reads to filtered alignments in one step

So let's try reducing the original alignment pipeline to one step with pipes. We'll work in the BAM file bit later.

Simply take away the output file names (except the last one!) and replace with pipes. It looks like this

```
minimap2 -ax sr ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq | samtools view -S -h -q 25 -f
```

when you do `ls -alh` you should see the new `aln.filtered.from_pipes.sam` file, its size is identical to the file we generated when we created the intermediate `aln.sam` file, but this time we didnt need to, saving that disk space.

5.3.1 From reads to filtered alignments in a BAM file in one step

Let's modify the command to give us BAM not SAM, saving a further step. We already know that `samtools view` can output BAM instead of SAM, so lets add that option (`-b`) in to the `samtools` part.

```
minimap2 -ax sr ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq | samtools view -S -h -b -q 25
```

If you check the files with `ls -alh` now you should see that you have the new `aln.filtered.from_pipes.bam` file with no extra intermediate file and the smallest possible output file. Congratulations, you know now the fastest and most optimal way to make alignments and filter them.

5.4 Sorting BAM files

In practice a BAM file of alignments needs to be ordered with the alignments at the start of the first chromosome at the start of the file and the alignments on the end of the last chromosome at the end of the file. This is for computational reasons we don't need to worry about, but it does mean we need to do another sorting step to make our files useful downstream.

Because all the alignments need to be present before we can start we can't use the pipe technique above. So we use an input and output file. The command is `samtools sort` and looks like this.

```
samtools sort aln.filtered.from_pipes.bam -o aln.filtered.from_pipes.sorted.bam
```

Doing `ls -alh` shows a new sorted BAM `aln.filtered.from_pipes.sorted.bam` that contains the same information but is actually a little smaller due to being sorted. We can safely delete the unsorted version of the BAM file.

5.4.1 Automatically deleting the unsorted BAM

If the sorting goes fine, we have two BAM files with essentially the same information and don't need the unsorted file. We can of course remove this with `rm aln.filtered.from_pipes`. A neat space saving trick is to combine the `rm` step with the successful completion of the sort. We can do this by joining the commands with `&&`.

That looks like this

```
samtools sort aln.filtered.from_pipes.bam -o aln.filtered.from_pipes.sorted.bam && rm a
```

The `&&` doesn't connect the data between the two commands, it just doesn't let the second one start until the first one finishes successfully (computers have an internal concept of whether a command finished properly).

This means if the `samtools sort` goes wrong the `rm` part will not run and the input file won't be deleted so you won't have to remake it. This is especially useful later when we wrap all this into an automatic script.

5.5 Indexing the sorted BAM

Many downstream applications need the BAM file to have an index, so they can quickly jump to a particular part of the reference chromosome. This is a tiny file and we usually don't need to worry about it. To generate it use `samtools index`

```
samtools index aln.filtered.from_pipes.sorted.bam
```

Using `ls -lah` we can see a tiny file called `aln.filtered.from_pipes.sorted.bam.bai`, this is the index.

5.6 Further Reading

For a primer on some more aspects of `samtools` see this tutorial

Chapter 6

Automating The Process

We now know everything we need to do an alignment of reads against a reference in an efficient way. What's next is to consider that this process needs to be done for every set of reads you might generate. That's a lot of typing of the same thing over and over, which can get tedious. In this section we'll look at how we can automate the process to make it less repetitive using a script.

6.1 Shell scripts

Scripts that contain commands we usually run in the Terminal are called shell scripts. They're generally just the command we want to do one after another and saved in a file. We can then run that file as if it were a command and all the commands we put in the file are

Shell scripts must be a simple text file, so you can't create them in programs like Word, you'll need a special text editor. On most systems we have one called `nano` built into the Terminal.

6.1.1 Using `nano` to create a shell script

To open a file in `nano` type `nano` and the name of the file, if the file doesn't exist it will be created.

```
nano my_script.sh
```

Will create a file and open it. To save and exit type press `Ctrl` then `X` (thats what `^X` means in the help at the bottom. You can enter your script in here. Remember its not a word processor, its a Terminal text editor, so you have to use the mouse to move round and cutting and pasting is a bit clunky.

6.2 Creating a script that automates our alignment pipeline.

Let's enter our script into `nano`. We'll do it as we did in the earlier chapters, but we'll change file names to make it clear which files are coming from the script.

First, create a script called `do_aln.sh`

`nano do_aln.sh`

Once `nano` opens, add the following into it

```
minimap2 -ax sr ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq | samtools view -S -
samtools sort aln.script.bam -o aln.script.sorted.bam && rm aln.script.bam
samtools index aln.script.sorted.bam
```

That's all the steps we want to do. Use `Ctrl-X` to save the changes to the file.

6.3 Running the script

To run the script we use the `sh` command and the script name. Try

`sh do_aln.sh`

You should see progress from the script as it does each step in turn. When it's done you can `ls -alh` to see the new sorted BAM file from the script.

Congratulations! You just automated an entire analysis pipeline!

6.4 Running on different input files

So our script is great but the input filenames will be the same every time we run it meaning we'd need to go through the whole file and change them which is error prone. Also the output files are the same each time, meaning we could accidentally overwrite any previous work in there, which is frustrating. We can overcome this with a couple of simple changes in our script that make use of variables.

Variables are place holders for values that the script will replace when it runs. Consider these two commands

```
MY_MESSAGE="Hello, world!"
echo $MY_MESSAGE
```

Recall that `echo` just prints whatever follows it. Try running this, you get `Hello, world!` which shows that the process created a variable called `MY_MESSAGE` and stored the message in it. When used by a command the `$`

showed the command that it should use the message stored in the variable and printed `Hello, world!`. We can use this technique in our scripts. Note the command `MY_MESSAGE="Hello, world!"` must not have spaces around the equals sign.

Now we can expand our script to take advantage. Look at this script.

```
LEFT_READS="ecoli_left_R1.fq"
RIGHT_READS="ecoli_right_R2.fq"
REFERENCE_SEQUENCE="ecoli_genome.fa"
SAMPLE_NAME="ecoli"

minimap2 -ax sr $REFERENCE_SEQUENCE $LEFT_READS $RIGHT_READS | samtools view -S -h -b -q 25 -f 3
samtools sort $SAMPLE_NAME.bam -o $SAMPLE_NAME.sorted.bam && rm $SAMPLE_NAME.bam
samtools index $SAMPLE_NAME.sorted.bam
```

Right at the top we create a variable for each of our read files (`LEFT_READS` and `RIGHT_READS`), our reference files (`REFERENCE_SEQUENCE`) and a unique sample name (`ecoli`). These variables get used whenever we need them, saving us from typing the information over and over. The practical upshot of this being that we only need to change the script in one place every time we reuse it for a different sample and set of reads.

Now try this out. Save the new script in a file called `do_aln_variables.sh` and run it as before with `sh do_aln_variables.sh`. When it's run you should see an output called `ecoli.sorted.bam`.

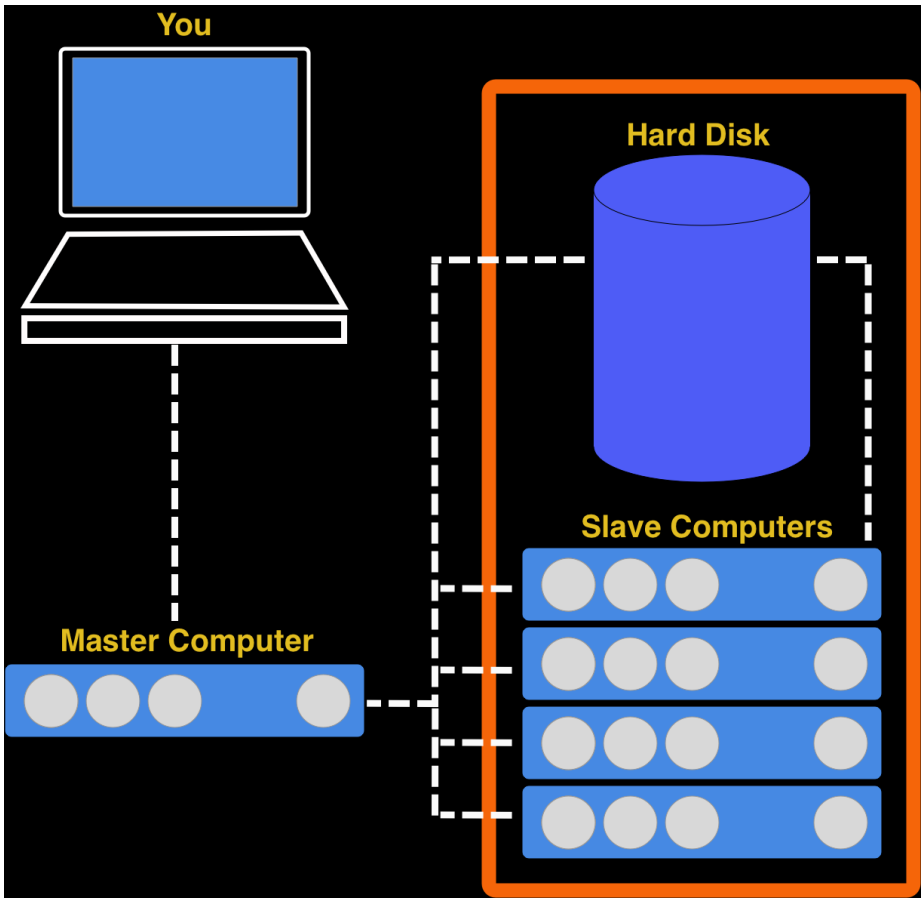
Chapter 7

Running an alignment on the HPC

In this chapter we'll look at how to run the alignment on an HPC cluster. First, we need to know a few things about that HPC before we can begin.

7.1 An HPC is a group of slave computers under control of a master computer

Most of the computers in an HPC cluster are really just there to do what one other computer tells them. They cannot be contacted directly (by the normal user) and they have very little software already on them. As the user you must get a master computer to tell them what to do. A key thing about an HPC cluster is that all the computers share one massive hard-drive. Look at the diagram below.



It shows the computer you are working at, the master computer and it's relation to the slave computers and the hard disk. Note that there is no way for you to contact the slaves directly, even though the slaves (or more properly 'worker nodes') are where the actual job runs. So the workflow for running an HPC job goes like this

1. Log into master (more usually called 'submission node')
2. Prepare a task for the submission node to send to the nodes
3. Submit the task to the submission node
4. Wait for the submission node to send the job to the worker nodes
5. Wait for the worker nodes to finish the job

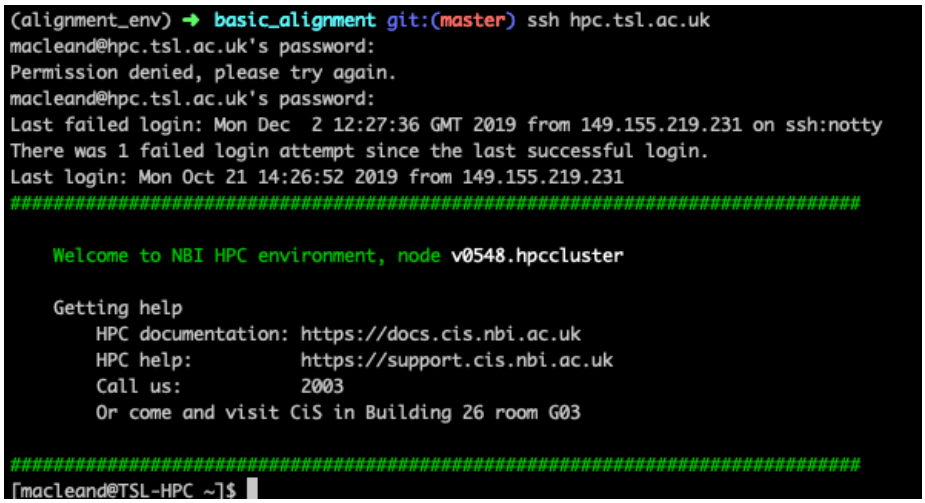
In the rest of this chapter we'll look at how to do these steps

7.2 Logging into the submission node

This is pretty straightforward, you need to use the `ssh` command to make a connection between your computer and the submission node. The TSL submission node has the address `hpc.tsl.ac.uk` so use this command

```
ssh hpc.tsl.ac.uk
```

You'll be asked for a user name and password, it's your usual NBI details. When it's done you should see something like this



This terminal is now working on the submission node (you can tell from the prompt `macleand@TSL-HPC`)

7.3 Preparing a job

To run a job we need to create a submission script. `nano` is available on the submission node, so we can use that. But what goes inside? Here's a typical one.

```
#!/bin/bash

#SBATCH -p tsl-short
#SBATCH --mem=16G
#SBATCH -c 4
#SBATCH -J alignments
#SBATCH --mail-type=begin,end,fail
#SBATCH --mail-user=dan.maclean@tsl.ac.uk
#SBATCH -o alignments.%j.out
```

```
#SBATCH -e slurm.%j.err

source minimap2-2.5
source samtools-1.9

srun minimap2 -ax sr ecoli_genome.fa ecoli_left_R1.fq ecoli_right_R2.fq | samtools view
```

Not much of this is going to be familiar, but it isn't complicated.

The first line of this file `#!/bin/bash` is one that should always be there. Always put it in and never worry about it again. It just tells the computer this file is a script.

7.3.1 The `#SBATCH` options

The second block of statements, all beginning `#SBATCH` are the resource options for the job. It tells the submission node what resources the job needs to run. These need to go at the top of the script. Let's look at them individually.

7.3.1.1 `#SBATCH -p`

This tells the submission node which queue (or partition in the jargon) the job should run on. We have three basic partitions `tsl-short`, `tsl-medium` and `tsl-long`. The main difference is that jobs that run for a short time shouldn't be held back by jobs that run for ages, so the submission node uses this to run all of its jobs optimally.

7.3.1.2 `#SBATCH -c`

The number here tells the machine how many CPU's (processors) to use. Most tools will be able to make use of more than one and will run faster as a consequence. The job (usually) won't fail if you get this wrong, but it will take longer to start as it waits for more CPU's to come free.

7.3.1.3 `#SBATCH --mem=`

This tells the submission node how much memory your job will need to run. Jobs that exceed their stated memory by too much are killed. REquestiing the lowest possible memory means your job will be executed more quickly. Memory is requested in units of `G` gigabytes, usually.

7.3.1.4 #SBATCH -J

This is a helpful little name for you to identify your jobs with. eg `#SBATCH -J my_jobs`

7.3.1.5 #SBATCH --mail-type=

These are the times during the job that the submission node will email you to let you know of a status change in your job. Always use this option as presented for quickest information.

7.3.1.6 #SBATCH --mail-user

This is simply the address your update emails will be sent to.

7.3.1.7 #SBATCH -o and #SBATCH -e

These are the names of files that output and errors will be sent to. On a long running process the output can get long so it goes to a file, not the email. The weird `%j` is a job ID number that uniquely identifies the job.

7.3.2 The source options

The next lines all begin with the word `source` followed by some software name. No software is loaded into the worker nodes by default, so we need to say which tools we want to use. Do this by using the `source` keyword followed by the software name, e.g `source BLAST-2.2.2`. Many versions of the same tool are available on the HPC, and are differentiated by the version number at the end. You can see which software is available to source by typing `source` then hitting the tab key twice. It should give a very long list of tools.

7.3.3 The srun command

Finally, we get to the actual commands we want to run. This is exactly as we did before but with the command `srun` in front.

7.4 Submitting with sbatch

All of this information should be saved in a single script. You can call it what you want, but use the extension `.sh`. Once you've got this script, you can ask

the submission node to add your job to the queue with `sbatch`. This doesn't go in the script, it goes on the command-line, so if you'd added all the details above to a file called `do_my_alignments.sh` you can submit it by typing `sbatch do_my_alignments.sh`

7.5 Checkout tasks

So that's all you need to know to submit a job. Let's test how that works by creating a simple job and running that. Then we'll try a bigger alignment job. These are

1. Create a job using a submission script that runs this command `date`. Check what the `date` command does on the command line. Note that it runs very quickly (is a short job) and uses very little memory (< 1G) and only needs one CPU.
2. What happened to the output? Check the contents of your directory when the job is done and examine the new files (`less` is useful for this).
3. Explicitly create an output file by running this command through the HPC instead `date > date.txt`. What is the contents of the folder now? What effect did explicitly naming an output file have. What is the `slurm_xxxx.out` file for?
4. Run an alignment job using the information we learned in the earlier chapters. The reference file `ecoli_genome.fa`, `ecoli_left_R1.fq`, `ecoli_right_R2.fq` are available in the HPC filesystem in the folder. `/tsl/data/reads/bioinformatics/tutorial/alignments/`

7.6 Further Reading

You can see more information about the cluster submission system and options at the CiS documentation site