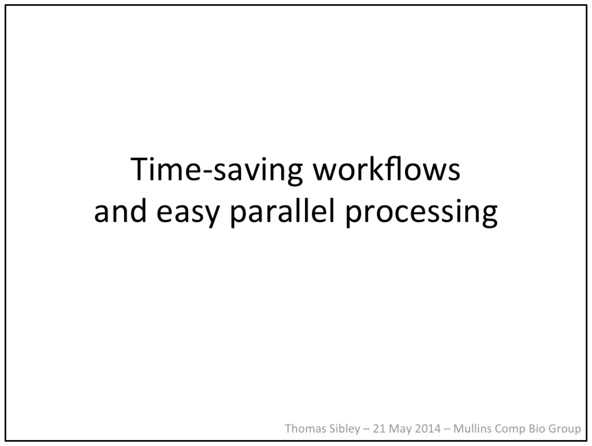
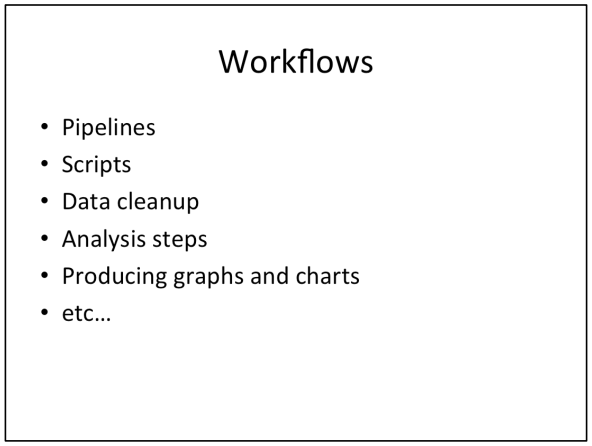


Time-saving workflows and easy parallel processing

Thomas Sibley – 21 May 2014 – Mullins Comp Bio Group

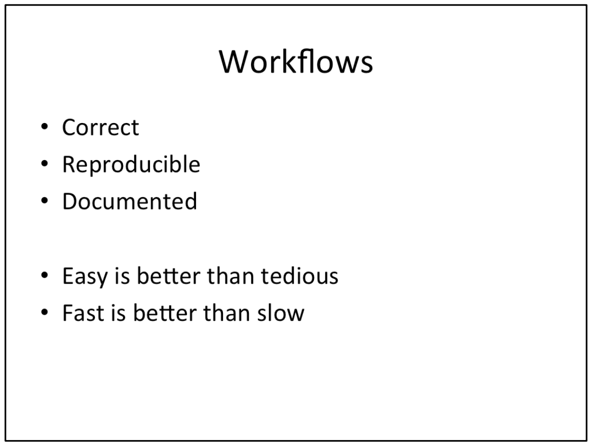


Workflows

- Pipelines
- Scripts
- Data cleanup
- Analysis steps
- Producing graphs and charts
- etc…

These are your computational methods in a paper.

Whatever you did to go from the raw data to your charts and graphs and final data tables.



Workflows

- Correct
- Reproducible
- Documented

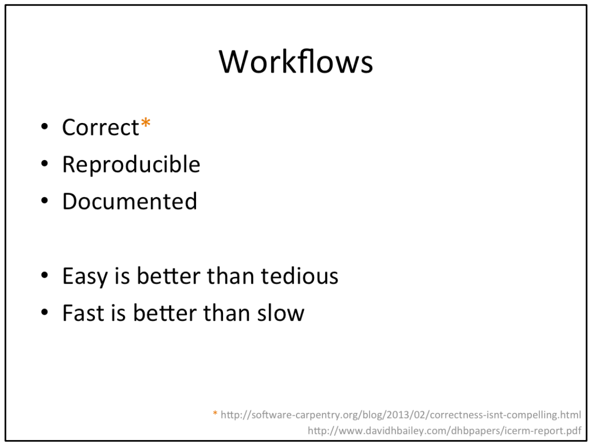
- Easy is better than tedious
- Fast is better than slow

These are your computational methods in a paper.

Reproducible doesn't just mean people in other labs. It means people in your lab and even you, a few months or few years later.

Documentation also includes software versions, sources, data input/output, and more, but the steps you took to process and analyze the data is a huge one.

If all of those hold about your workflow, then you'd also like it to be easy and fast rather than tedious and slow. Not only do you not spend needless time waiting, but there are tangible benefits in terms of thinking critically about a problem when you can iterate quickly trying out new ideas or fixing problems in the analysis. If it takes you 8 hours to run an analysis, fixing problems is frustrating and you're afraid to make changes and try new things.



Workflows

- Correct*
- Reproducible
- Documented

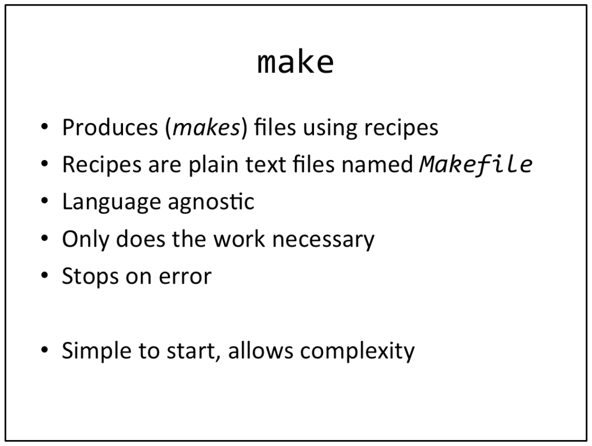
- Easy is better than tedious
- Fast is better than slow

* http://software-carpentry.org/blog/2013/02/correctness-isnt-compelling.html
http://www.davidhbailey.com/dhbpapers/icerm-report.pdf

A caveat: Correctness isn't compelling, and there was a study to prove it. Incentives are all wrong as few papers are subjected to reproduction attempts.

However, making your analyses reproducible and self-documenting will save you time puzzling over what you did and make it easy to redo analysis on an updated or new data set.

When you do similar analyses in the future or when another member of the lab or your collaborators want to run the same analysis, you'll all save time not reinventing what you already did. And when you reuse what you did previously, you're more certain that it works.



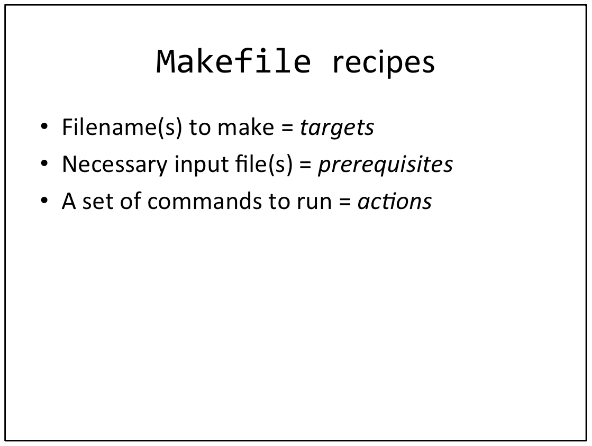
`make`

- Produces (*makes*) files using recipes
- Recipes are plain text files named *`Makefile`*
- Language agnostic
- Only does the work necessary
- Stops on error

- Simple to start, allows complexity

Scales to the size of your project, whether it's very simple or very complex.

There are some sharp corners with Makefiles and make, but it's a time-tested tool and no software doesn't have sharp corners somewhere.



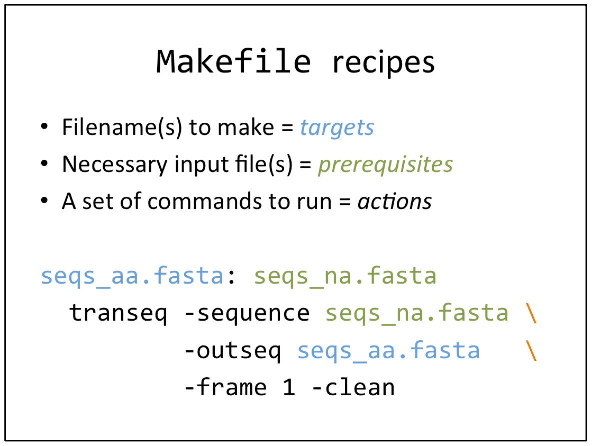
`Makefile` recipes

- Filename(s) to make = *targets*
- Necessary input file(s) = *prerequisites*
- A set of commands to run = *actions*

I often prototype Makefile recipes straight on the command line before putting them into a Makefile and further testing and tweaking from there.

Recipes just describe what to run, the necessary prerequisites (or dependencies), and what files are produced. You can run core Unix commands, your own Python, R, or Perl scripts, and use features of your shell to pipe data between commands.

The prerequisites are important because make uses those to determine what order to run the recipes. It also uses them to skip over recipes if the files they produce already exist and the input files haven't changed since then. This can save a lot of time when writing complex workflows with steps that take a while. Once you verify that the long running steps are correct, you can run them once and then move on to using them down the road without re-running it everytime. Best yet, make will figure this out for you and you don't need to remember what's changed.



Makefile recipes

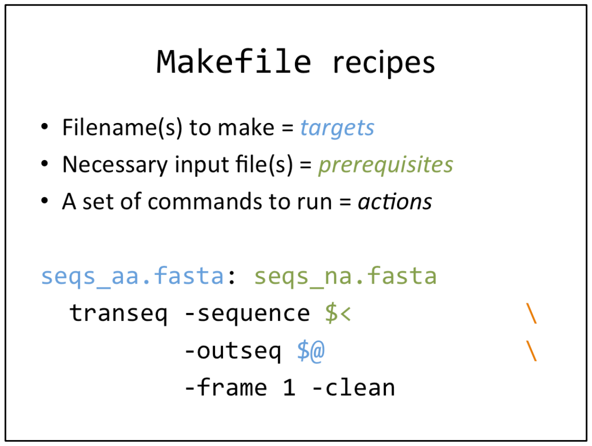
- Filename(s) to make = *targets*
- Necessary input file(s) = *prerequisites*
- A set of commands to run = *actions*

```
seqs_aa.fasta: seqs_na.fasta
  transeq -sequence seqs_na.fasta \
          -outseq seqs_aa.fasta   \
          -frame 1 -clean
```

An example!

Note the line continuations. This says that these three lines will be run as a single command. Without those, each line would be a separate command (and wouldn't work because -outseq isn't a valid command).

To run this, we'd simply type: make seqs_aa.fasta and presto, make follows our recipe to produce it from seqs_na.fasta. Nifty, though I guess not very impressive yet.



Makefile recipes

- Filename(s) to make = *targets*
- Necessary input file(s) = *prerequisites*
- A set of commands to run = *actions*

```
seqs_aa.fasta: seqs_na.fasta
  transeq -sequence $<            \
          -outseq $@              \
          -frame 1 -clean
```

This is an equivalent recipe which save some typing. It also provides you flexibility if you rename your targets or inputs later. It's especially useful for recipes which describe how to make multiple files, so I'll use these variables from now on. It's a good practice. Note that recipes can specify multiple targets or prerequisites and $< and $@ are only the first of the files being made/input. There are other variables to get all of them, which you'll see later.

```
seqs_aa.fa: seqs_na.fa
   transeq -sequence $< -outseq $@ \
            -frame 1 -clean

seqs_aa.aligned.fa: seqs_aa.fa
   muscle -quiet < $< > $@

seqs_aa_freq.tsv: seqs_aa.aligned.fa
   # CountAAFreq.pl only takes Nexus
   fasta2nexus < $< > seqs_aa.nxs
   perl CountAAFreq.pl seqs_aa.nxs $@ 0.25 0.5
   rm seqs_aa.nexus
```

A longer example showing multiple targets that depend on each other. This takes a fasta of nucleotide sequences, translates it to amino acids, aligns it with muscle, and then runs it through Wenjie's CountAAFreq.pl. It also has to convert the aligned fasta to nexus. Note how each subsequent recipe depends on another. These are ordered sequentially, but they don't have to be and multiple recipes can depend on the same recipe.

It's also worth noting the command line's (shell's) input and output redirection operators. fasta2nexus, for example, takes an input stream and prints an output stream rather than taking an input filename and output filename itself. CountAAFreq.pl on the other hand takes two filenames along with some cutoff values (gaps and frequency). Do you see why $< uses <?

Notice the temporary nexus file the last recipe creates and then deletes? We can do it that way, but it's a good practice to keep your recipes as short as possible to enable reuse and save time later. With make, this is easy! Let's see how you could modify the recipes.

To produce a table of amino acid frequencies from a seqs_na.fa file, we can now type: make seqs_aa_freq.tsv and make will follow all the necessary recipes.

```
seqs_aa.fa: seqs_na.fa
   transeq -sequence $< -outseq $@ \
            -frame 1 -clean

seqs_aa.aligned.fa: seqs_aa.fa
   muscle -quiet < $< > $@

seqs_aa.nxs: seqs_aa.aligned.fa
   fasta2nexus < $< > $@

seqs_aa_freq.tsv: seqs_aa.nxs
   perl CountAAFreq.pl $< $@ 0.25 0.5
```

Note our recipes are simpler again by making a separate recipe to describe how to make the nexus file out of the aligned fasta.

We can take it a step further though since there's little use for both a nexus file and the aligned fasta. Why not just produce a nexus file to begin with?

```
seqs_aa.fa: seqs_na.fa
   transeq -sequence $< -outseq $@ \
            -frame 1 -clean

seqs_aa.nxs: seqs_aa.fa
   muscle -quiet < $< | fasta2nexus > $@




seqs_aa_freq.tsv: seqs_aa.nxs
   perl CountAAFreq.pl $< $@ 0.25 0.5
```

Now we're getting somewhere! We can pipe the output of muscle directly to fasta2nexus and convert it on the fly. Using pipes not only simplifies the recipe, but it's faster than writing a bunch of files.

Great, we have a workflow to get amino acid frequencies from a set of nucleotide sequences!

But it only works for one file, and that's annoying. Ideally we'd like to generalize it so it works for any nucleotide fasta we have without renaming all our files to seqs_na.fa.

```
%_aa.fa: %_na.fa
   transeq -sequence $< -outseq $@ \
            -frame 1 -clean

seqs_aa.nxs: seqs_aa.fa
   muscle -quiet < $< | fasta2nexus > $@




seqs_aa_freq.tsv: seqs_aa.nxs
   perl CountAAFreq.pl $< $@ 0.25 0.5
```

Luckily, make supports this by writing recipes that use pattern matching in the target and prerequisites. The % is a wildcard here and represents the same name on each side of the colon.

This means we can take any file named sometext_na.fa and produce a sometext_aa.fa just by typing: make sometext_aa.fa

That's pretty nifty, but notice that our other recipes are still hardcoded. Let's fix that.

```
%_aa.fa: %_na.fa
   transeq -sequence $< -outseq $@ \
            -frame 1 -clean

%.nxs: %.fa
   muscle -quiet < $< | fasta2nexus > $@




seqs_aa_freq.tsv: seqs_aa.nxs
   perl CountAAFreq.pl $< $@ 0.25 0.5
```

This recipe will now take any fasta and produce a nexus file. With just a seqs_na.fa file, you can now type: make seqs_aa.nxs and get an aligned amino acid sequences in a nexus file. Make will run the first rule if it needs to, and then run the second rule with the output from the first.

But note that there's no restriction on the filenames other than the extensions, so if want to align your nucleotide sequences instead and get a nexus file of those, you can also do: make seqs_na.nxs. The first rule won't be run since no seqs_aa.fa needs to be made.

Let's keep going to make the whole workflow generalized.

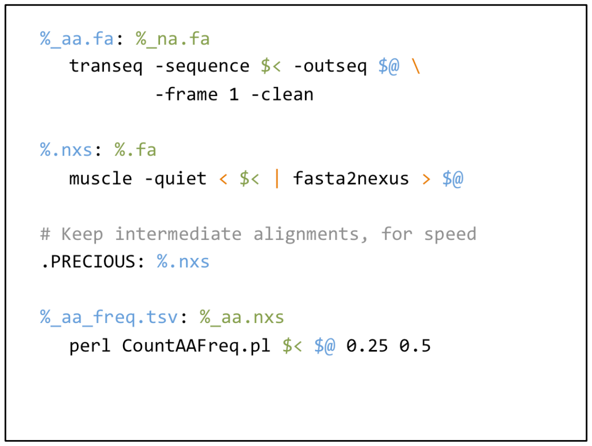
```
%_aa.fa: %_na.fa
   transeq -sequence $< -outseq $@ \
            -frame 1 -clean

%.nxs: %.fa
   muscle -quiet < $< | fasta2nexus > $@




%_aa_freq.tsv: %_aa.nxs
   perl CountAAFreq.pl $< $@ 0.25 0.5
```

This is the last piece of the puzzle. Our rule for counting amino acid frequencies now describes how to take any nexus file of amino acid sequences (using the convention _aa.nxs) and runs it through Wenjie's program to produce a frequency table.

If I have a file pic_na.fa, I can now run: make pic_aa_freq.tsv. It's important to note that if I already have a someseqs_aa.fa file from somewhere else, I can still run `make someseqs_aa_freq.tsv` and make will realize it doesn't need to run the first rule to translate from nucleotides.

It's also important to note that the filenames I'm using are just conventions. You can use whatever you want, for example, to distinguish between amino acid and nucleotide fastas, as long as you're consistent within your recipes.

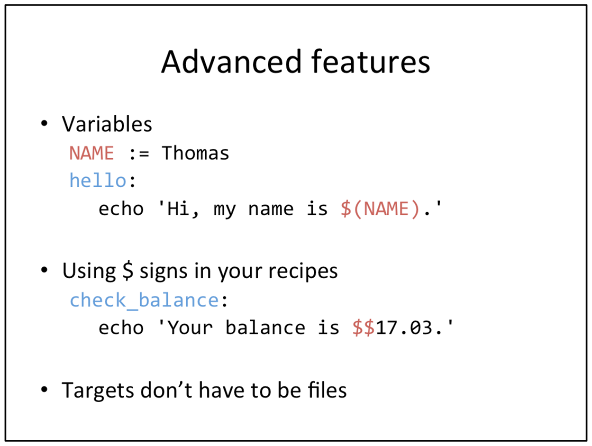
Normally make will delete intermediate files after it's done with them. Intermediate files are any files you didn't ask for, but that it had to produce to get from your input to the output you asked for. In the case of going from seqs_na.fa to seqs_aa_freq.tsv, there are two intermediate files: seqs_aa.fa and seqs_aa.nxs which make will delete when it's done. This is just a cleanliness thing so you have less files to look at in your directory. But sometimes you want to keep those files around, especially if they take a while to produce. muscle, for example, might take a long time on a large set of sequences. There's a way to tell make that it shouldn't delete certain intermediate files, that certain files are… precious.



```
%_aa.fa: %_na.fa
   transeq -sequence $< -outseq $@ \
            -frame 1 -clean

%.nxs: %.fa
   muscle -quiet < $< | fasta2nexus > $@

# Keep intermediate alignments, for speed
.PRECIOUS: %.nxs

%_aa_freq.tsv: %_aa.nxs
   perl CountAAFreq.pl $< $@ 0.25 0.5
```

The special target ".PRECIOUS" does this and any prerequisites you specify won't be deleted even if they're intermediate files.

The gray text is a comment, which you can put in your Makefiles by starting a line with a hash or pound sign.



Advanced features

- Variables

```
NAME := Thomas
hello:
	echo 'Hi, my name is $(NAME).'
```

- Using $ signs in your recipes

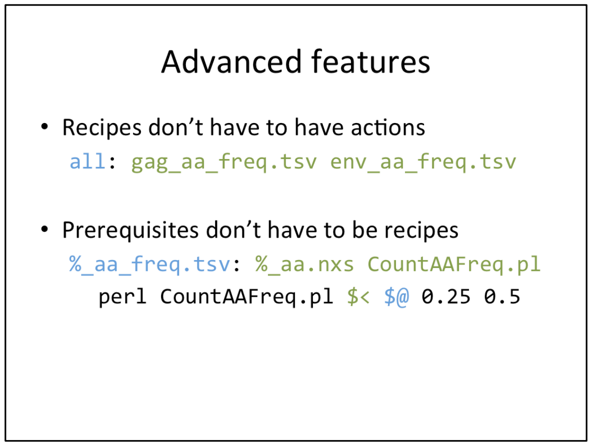
```
check_balance:
	echo 'Your balance is $$17.03.'
```

- Targets don't have to be files

Variables help reduce repetition in directory names or other commonly used parameters. You can override them when running make: `make hello NAME=Jim` Variable names longer than a single character need to be surrounded by parentheses.

Since dollar signs introduce a variable in Makefiles, to use an actual dollar sign you need to type it twice.

You can see that targets don't have to be files. Make doesn't create a target file itself, that's up to the recipe. So targets may just be a convenient name for a recipe to run a bunch of commands that doesn't actually produce a file.



Advanced features

- Recipes don't have to have actions

```
all: gag_aa_freq.tsv env_aa_freq.tsv
```

- Prerequisites don't have to be recipes

```
%_aa_freq.tsv: %_aa.nxs CountAAFreq.pl
	perl CountAAFreq.pl $< $@ 0.25 0.5
```

You can also create targets who's sole purpose is to list a bunch of other targets as prerequisites, which is a way of running multiple targets at once which don't depend on each other, or producing a number of specific files from a set of generalized recipes.

Prerequisites also don't have to be recipes. Remember that prerequisites are just the targets or files that a recipe needs to run and that when they change the recipe needs to be re-run. You can list your own programs as prerequisites and then make will know it needs to remake the files the next time you ask for them after updating your program. Make will know when you fix bugs!

Advanced features

- Recipes don't have to have actions

```
all: gag_aa_freq.tsv env_aa_freq.tsv
```

- Prerequisites don't have to be recipes

```
%_aa_freq.tsv: %_aa.nxs CountAAFreq.pl
  perl CountAAFreq.pl $< $@ 0.25 0.5

%_aa_freq.tsv: CountAAFreq.pl %_aa.nxs
  perl $^ $@ 0.25 0.5
```

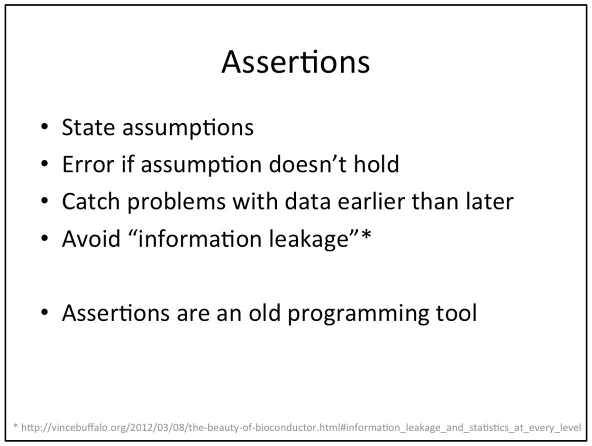
You could write that last recipe like this too.

$^ is an automatic variable that lists all the prerequisites, separated by spaces. In this case, it'll expand to the script name and the input filename to make the first two arguments to perl.

There are other variables too, such as $* which is the just the shared stem, or wildcard part, of the filename.

Validation

- Charts in R



Assertions

- State assumptions
- Error if assumption doesn't hold
- Catch problems with data earlier than later
- Avoid "information leakage"*

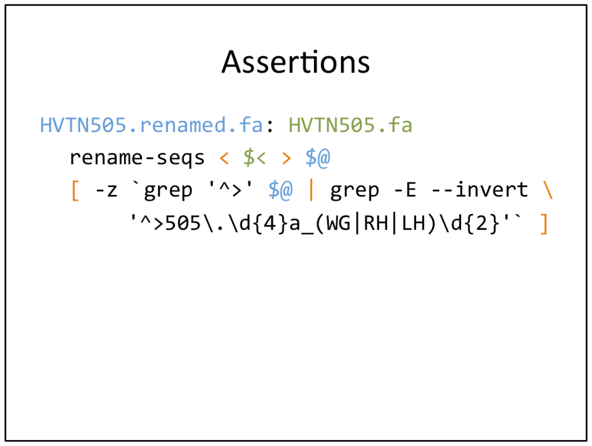
- Assertions are an old programming tool

* http://vincebuffalo.org/2012/03/08/the-beauty-of-bioconductor.html#information_leakage_and_statistics_at_every_level

Assertions declare that an assumption you're making about the data must be true or the computer shouldn't continue. You generally express this in terms of a simple condition that should hold, like, "All the sequence names should match this pattern."

They're useful for avoiding informational leakage in pipelines and catching problems before they go any further and become larger or harder to notice. Physical leaks are similarly caught and fixed in oil, water, and gas pipelines by checking expected pressure at intervals.

Assertions are relatively easy to add to Makefile recipes because if you cause an error make will abort processing.



Assertions

```
HVTN505.renamed.fa: HVTN505.fa
  rename-seqs < $< > $@
  [ -z `grep '^>' $@ | grep -E --invert \
      '^>505\.\d{4}a_(WG|RH|LH)\d{2}'` ]
```

`Makefile` gotchas

- Hard tabs vs. spaces
- Force updates after changing `Makefile`
- Change default behavior on error:

```
SHELL := /bin/bash
export SHELLOPTS := errexit:pipefail
.DELETE_ON_ERROR:
```

The action part of a recipe must use hard tabs for the first indent, not spaces. This is often a source of problems. All editors should have a way of highlighting hard tabs vs. spaces.

When a Makefile changes, you often need to rerun the recipes. Since the creation times of the input and output files don't change, just running make won't do that. To get around this, you can run `make -B` to force run a target and all dependent targets. You can also update the timestamps of all your input files using `touch` and then rerun your targets with make.

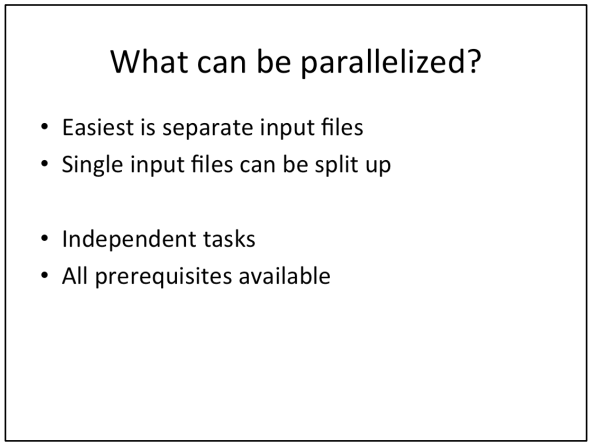
make's default behaviour on errors is less than ideal. Only the success/failure status of the last command in a pipeline is considered a failure, even if a command in the middle fails partway through the data.

When make does catch an error, it leaves any partially made target files around. You can include the special empty target .DELETE_ON_ERROR: to make it delete any partially-complete target files if the recipe fails.

This avoids running other recipes later which may use the partial data.

Parallel processing

How to do more than one thing at a time



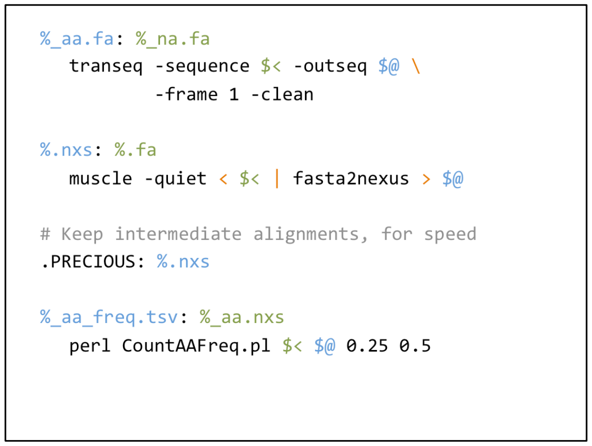
What can be parallelized?

- Easiest is separate input files
- Single input files can be split up

- Independent tasks
- All prerequisites available

The easiest thing to parallelize is when you have a bunch of separate input files to do the same thing to and it takes a while to do it. But that said, you can also split up a single input file if you're running tasks over individual things in that input file. For example, and I'll show this in a minute with blast, you might be doing something to every sequence in a file and it takes a while or there are a lot of sequences.

In any multi-step workflow, you must also think about dependencies and what steps of your workflow depend on other steps. For example, you can't run a step expecting amino acids as input when you haven't translated your nucleotide sequences yet. Hey wait… that sounds familiar. We just thought about this for Makefiles!



```
%_aa.fa: %_na.fa
   transeq -sequence $< -outseq $@ \
           -frame 1 -clean

%.nxs: %.fa
   muscle -quiet < $< | fasta2nexus > $@

# Keep intermediate alignments, for speed
.PRECIOUS: %.nxs

%_aa_freq.tsv: %_aa.nxs
   perl CountAAFreq.pl $< $@ 0.25 0.5
```

Remember our Makefile from earlier? It calculates amino acid frequencies from a set of nucleotide sequences. We generalized it so that filenames aren't hardcoded and it'll operate on anything named properly.

The work we put in to describe our workflow in a Makefile will pay back again.

Parallelizing with make

gag_na.fa

env_na.fa

```
make gag_aa_freq.tsv env_aa_freq.tsv
make {gag,env}_aa_freq.tsv
```

Given two files, you'd get amino acid frequencies like this. The second make example does the same thing.

Parallelizing with make

- What if you had those files for 20 patients?

Pt100_gag_na.fa, Pt100_env_na.fa

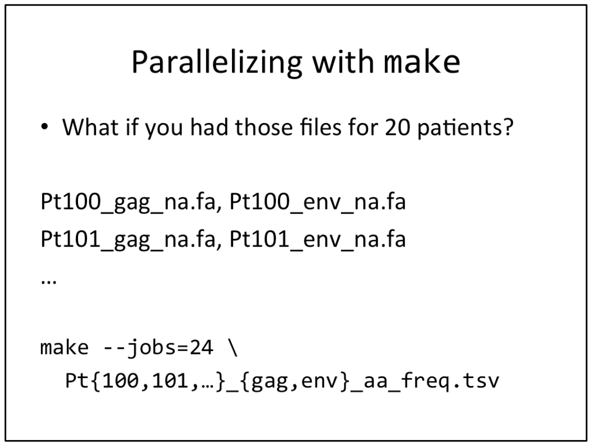
Pt101_gag_na.fa, Pt101_env_na.fa

…

```
make Pt{100,101,…}_{gag,env}_aa_freq.tsv
```

Maybe you'd do something like this. Make is going to process those one after the other though, and that'll take a while. Probably not too long for our simple example, but consider if the alignments were large or you were doing a more computationally intensively task.

Since you described your workflow in a Makefile, make knows what output files require what input files and how to put it all together. All you need to do is tell make how many jobs to run in parallel!



Parallelizing with make

- What if you had those files for 20 patients?

Pt100_gag_na.fa, Pt100_env_na.fa

Pt101_gag_na.fa, Pt101_env_na.fa

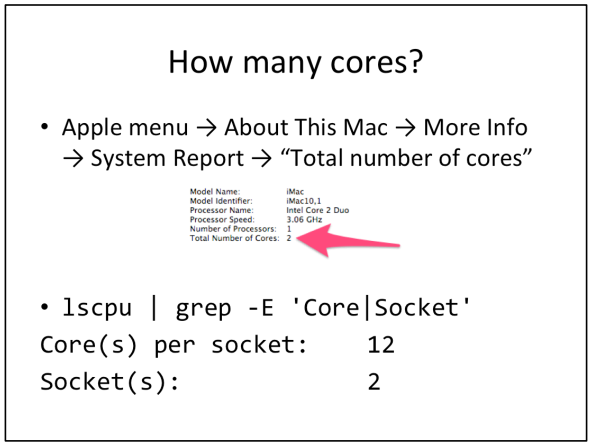
…

```
make --jobs=24 \
  Pt{100,101,…}_{gag,env}_aa_freq.tsv
```

make takes a --jobs or -j option which tells it how many jobs, or tasks, to run at one time. By default it only runs one recipe at a time, but it'll happily run as many as you tell it. Now instead of waiting for Pt100_gag_aa_freq.tsv to finish before starting Pt100_env_aa_freq.tsv, make will run up to 24 tasks at once! That should cut down on the runtime a lot.

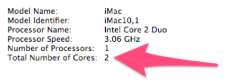
It's really that easy if you have a Makefile, and this is one of the benefits of describing your workflows with make. Like any software, of course, it's possible for it to break if you missed a prerequisite input file for a recipe. If you're running individual steps by hand, you might not notice until you try to run them in parallel because previously the file from recipe 1 always happened to exist before you ran recipe 2.

Note that you shouldn't ask to run more jobs than the number of cores if the tasks are computationally intensive (generally the case for bioinformatics). themis has 24 cores, your computer might have 1 or 2 or 4. Other servers will have different numbers. This leads to the question…



How many cores?

- Apple menu → About This Mac → More Info → System Report → "Total number of cores"



```
Model Name:                iMac
Model Identifier:          iMac10,1
Processor Name:            Intel Core 2 Duo
Processor Speed:           3.06 GHz
Number of Processors:      1
Total Number of Cores:     2
```

```
• lscpu | grep -E 'Core|Socket'
Core(s) per socket:     12
Socket(s):              2
```

…"How many cores do I have?" You can check yourself on Mac or Linux.

On Linux, each "socket" is a physical CPU, so multiply Cores per socket by the number of Sockets to get the total number of cores.

I've got 99 problems, but a `Makefile` ain't one.

Parallel processing power tools

No `Makefile`? No problem.

```
for file in *.fasta; do
  do_something -in $file -out $file.new
done
```

Maybe you have a command-line loop like this that you run over data files. It's straightforward to turn into a Makefile, but you don't want to. You may know about using an ampersand to run things in the background which makes the loop complete quickly, and then you wait around for all the .new files to pop into existence. That's fine for a handful of files, but if you have more than a couple dozen files, you'll bog down the computer with too many jobs.

No Makefile? No problem.

```
for file in *.fasta; do
  do_something -in $file -out $file.new
done

parallel do_something -in {} -out {}.new \
  ::: *.fasta
```

Enter a tool called “parallel”. It helps you run other programs in parallel and does a lot to handle input and output to each job. It’s the Leatherman of command-line parallel processing. I’d compare it to a Swiss Army knife, but that’s not fair to the knife.

In it’s most basic form, parallel easily replaces your loops. In more complicated forms, it can split up your input and rejoin the output back together.

Like make, it also has a -j option, but by default it will use as many jobs as the computer has cores. Handy! You can run the same command on your desktop as the server and it’ll just magically go faster on the server without thinking about the number of cores.

No Makefile? No problem.

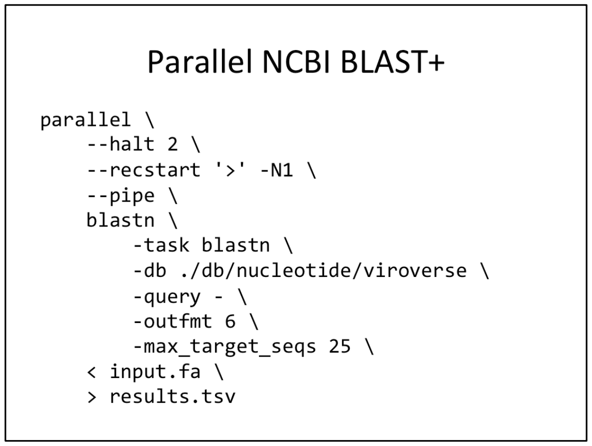
```
for file in *.fasta; do
  do_something -in $file -out $file.new
done

parallel do_something -in {} -out {}.new \
  ::: *.fasta
```

Enter a tool called “parallel”. It helps you run other programs in parallel and does a lot to handle input and output to each job. It’s the Leatherman of command-line parallel processing. I’d compare it to a Swiss Army knife, but that’s not fair to the knife.

In it’s most basic form, parallel easily replaces your loops. In more complicated forms, it can split up your input and rejoin the output back together.

Like make, it also has a -j option, but by default it will use as many jobs as the computer has cores. Handy! You can run the same command on your desktop as the server and it’ll just magically go faster on the server without thinking about the number of cores.

Let me show you more complicated example that will hopefully be immediately useful…

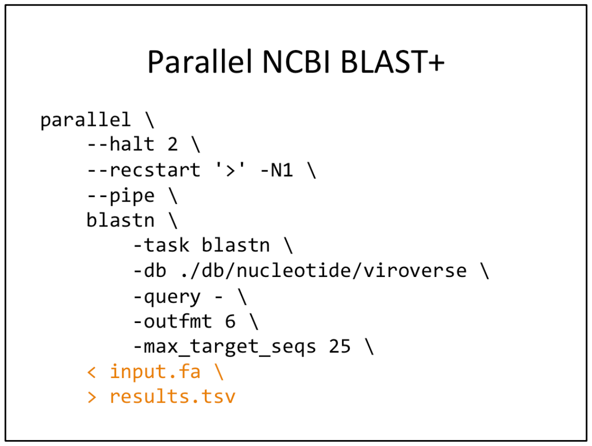


Parallel NCBI BLAST+

```
parallel \
    --halt 2 \
    --recstart '>' -N1 \
    --pipe \
    blastn \
        -task blastn \
        -db ./db/nucleotide/viroverse \
        -query - \
        -outfmt 6 \
        -max_target_seqs 25 \
    < input.fa \
    > results.tsv
```

Here's an example parallel command I put together to very efficiently run blastn on themis against a copy of the Viroverse blast database (the same thing used by the local Viroblast).

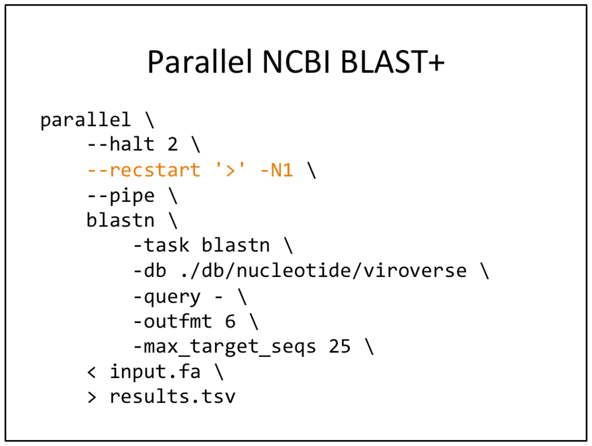
I'll walk through it piece by piece, but first, why should you care? Well, blasting 245 whole and half HIV genomes against Viroverse using our local Viroblast took an hour. Doing the same blast run on themis using this command takes 3 minutes.



Parallel NCBI BLAST+

```
parallel \
    --halt 2 \
    --recstart '>' -N1 \
    --pipe \
    blastn \
        -task blastn \
        -db ./db/nucleotide/viroverse \
        -query - \
        -outfmt 6 \
        -max_target_seqs 25 \
    < input.fa \
    > results.tsv
```

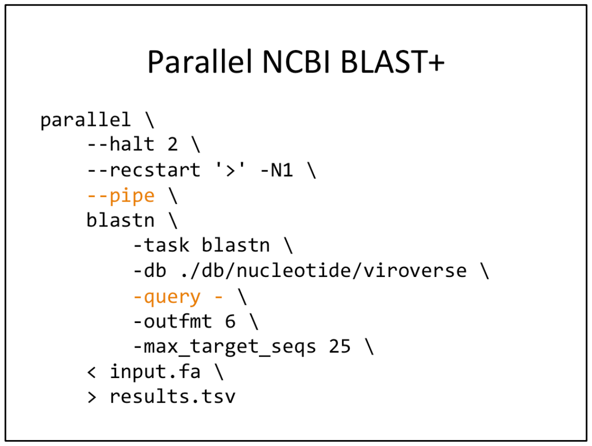
The first thing I want to note is actually at the end. The file input.fa is redirected as the input to parallel, and parallel's output is redirected to another file, results.tsv.



Parallel NCBI BLAST+

```
parallel \
    --halt 2 \
    --recstart '>' -N1 \
    --pipe \
    blastn \
        -task blastn \
        -db ./db/nucleotide/viroverse \
        -query - \
        -outfmt 6 \
        -max_target_seqs 25 \
    < input.fa \
    > results.tsv
```

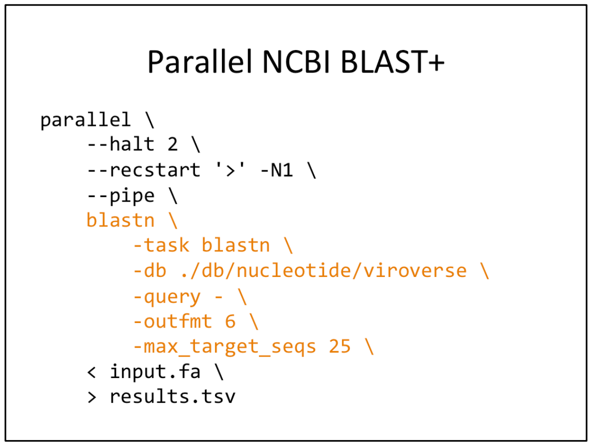
The next thing to note is what parallel is doing with that input for each job. --recstart tells parallel to split up the input into multiple “records”. In this case, I’m telling parallel that records start with a “>”, which should be familiar to you as the start of a FASTA sequence. I’m also telling parallel with -N1 to only pass one record at a time to blast. This means we’ll run one blast job for every sequence, but the number of jobs running at the same time is limited by the number of cores.



Parallel NCBI BLAST+

```
parallel \
    --halt 2 \
    --recstart '>' -N1 \
    --pipe \
    blastn \
        -task blastn \
        -db ./db/nucleotide/viroverse \
        -query - \
        -outfmt 6 \
        -max_target_seqs 25 \
    < input.fa \
    > results.tsv
```

Since parallel is splitting up the input file, we no longer have a filename for blast to process. Instead, we tell parallel to pipe the query sequence to blast as standard input. And we also tell blast that the query sequences are from stdin. The dash in place of a filename is a Unix convention to specify “stdin” rather than an actual filename.



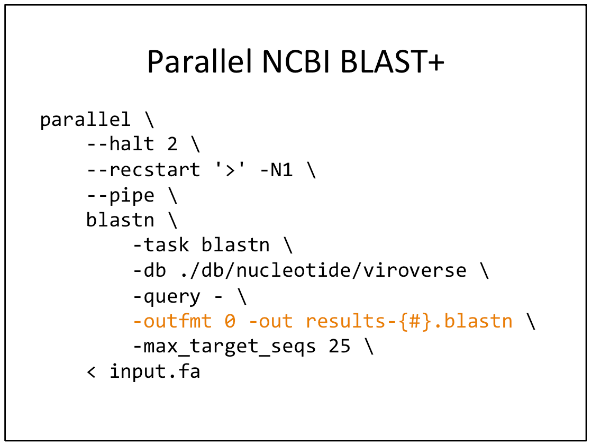
Parallel NCBI BLAST+

```
parallel \
    --halt 2 \
    --recstart '>' -N1 \
    --pipe \
    blastn \
        -task blastn \
        -db ./db/nucleotide/viroverse \
        -query - \
        -outfmt 6 \
        -max_target_seqs 25 \
    < input.fa \
    > results.tsv
```

Finally we have the blast command itself. We're using blastn, specifying a database, choosing a suitable output format and limiting the maximum hits to 25 per query. You can use whatever options make sense for your query and database, such as to adjust the blast scoring or output format.

Output format 6 is tabular data and blastn will print it to stdout, which parallel will capture and print as jobs finish. It's what we redirect into results.tsv.

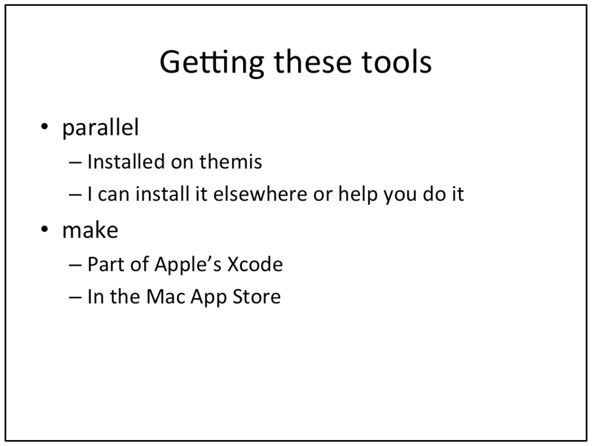
It's important to choose an output format which can be joined together easily. There are options for dealing with output formats that can't just be smushed together, but explaining those is for another day. If you find yourself in this boat, you can always have each blast job produce its own result file instead of joining them all together.



Parallel NCBI BLAST+

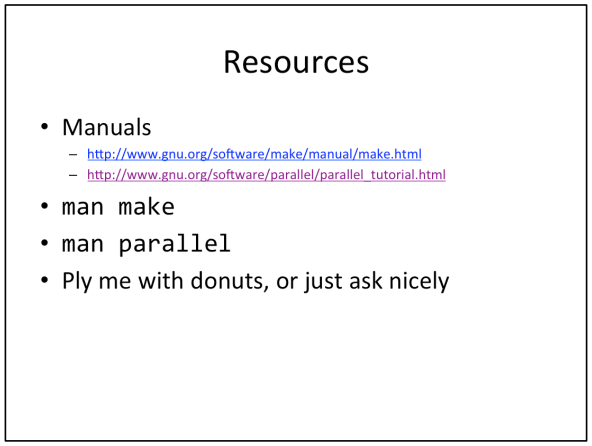
```
parallel \
    --halt 2 \
    --recstart '>' -N1 \
    --pipe \
    blastn \
        -task blastn \
        -db ./db/nucleotide/viroverse \
        -query - \
        -outfmt 0 -out results-{#}.blastn \
        -max_target_seqs 25 \
    < input.fa
```

For example, you could do this to get a pairwise output file for each blast job. Parallel substitutes {#} for the input record number, so you'll get as many files as you have input sequences.



Getting these tools

- parallel
  - Installed on themis
  - I can install it elsewhere or help you do it
- make
  - Part of Apple's Xcode
  - In the Mac App Store



Resources

- Manuals
  - http://www.gnu.org/software/make/manual/make.html
  - http://www.gnu.org/software/parallel/parallel_tutorial.html
- `man make`
- `man parallel`
- Ply me with donuts, or just ask nicely

The make manual is a bit dense, but it's the authoritative reference for how things work. It contains useful things such as descriptions of the automatic variables like $@, $<, $* and more.

The parallel tutorial is a little friendlier and better yet is chock full of examples showing how it works. There are other substitution patterns than just {}, for example, which are often useful.