Introduction to automation using Nextflow:
A tutorial through examples

Phelelani Mpangase

Sydney Brenner Institute for Molecular Bioscience
University of the Witwatersrand
Johannesburg
South Africa

Outline

Introduction to Nextflow

Introduction to Nextflow

Introduction

Resources

- `https://github.com/phelelani/nf-tut-2023`

Workflow Languages

Many scientific applications require

- Multiple data files
- Multiple applications
- Perhaps different parameters

General purpose languages not well suited

- Too low a level of abstraction
- Does not separate workflow from application
- Not reproducible

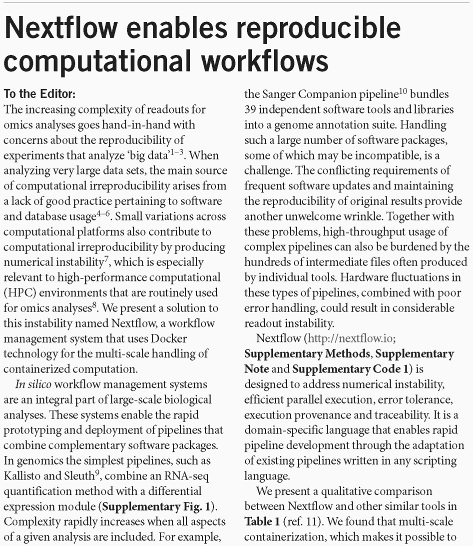
Workflow Languages

Many scientific applications require

- Multiple data files
- Multiple applications
- Perhaps different parameters

General purpose languages not well suited

- Too low a level of abstraction
- Does not separate workflow from application
- Not reproducible



Nextflow enables reproducible computational workflows

To the Editor:
The increasing complexity of readouts for omics analyses goes hand-in-hand with concerns about the reproducibility of experiments that analyze 'big data'[1–3]. When analyzing very large data sets, the main source of computational irreproducibility arises from a lack of good practice pertaining to software and database usage[4–6]. Small variations across computational platforms also contribute to computational irreproducibility by producing numerical instability[7], which is especially relevant to high-performance computational (HPC) environments that are routinely used for omics analyses[8]. We present a solution to this instability named Nextflow, a workflow management system that uses Docker technology for the multi-scale handling of containerized computation.

In silico workflow management systems are an integral part of large-scale biological analyses. These systems enable the rapid prototyping and deployment of pipelines that combine complementary software packages. In genomics the simplest pipelines, such as Kallisto and Sleuth[9], combine an RNA-seq quantification method with a differential expression module (**Supplementary Fig. 1**). Complexity rapidly increases when all aspects of a given analysis are included. For example, the Sanger Companion pipeline[10] bundles 39 independent software tools and libraries into a genome annotation suite. Handling such a large number of software packages, some of which may be incompatible, is a challenge. The conflicting requirements of frequent software updates and maintaining the reproducibility of original results provide another unwelcome wrinkle. Together with these problems, high-throughput usage of complex pipelines can also be burdened by the hundreds of intermediate files often produced by individual tools. Hardware fluctuations in these types of pipelines, combined with poor error handling, could result in considerable readout instability.

Nextflow (http://nextflow.io; **Supplementary Methods**, **Supplementary Note** and **Supplementary Code 1**) is designed to address numerical instability, efficient parallel execution, error tolerance, execution provenance and traceability. It is a domain-specific language that enables rapid pipeline development through the adaptation of existing pipelines written in any scripting language.

We present a qualitative comparison between Nextflow and other similar tools in **Table 1** (ref. 11). We found that multi-scale containerization, which makes it possible to

Nextflow

Groovy-based language

- Expressing workflows
- Portable
  - works on most Unix-like systems
- Very easy to install
  - NB: requires Java 7, 8
- Scalable
- Supports Docker/Singularity
- Supports a range of scheduling systems

Nextflow

Groovy-based language

- Expressing workflows
- Portable
  - works on most Unix-like systems
- Very easy to install
  - NB: requires Java 7, 8
- Scalable
- Supports Docker/Singularity
- Supports a range of scheduling systems

Key concepts of Nextflow

- **Processes**:
  - actual work being done (usually simple).
  - call program that does the analysis.
- **Channels**:
  - for communication between processes.
  - handles inputs and outputs.
- When all inputs ready, process is executed.
- Each process runs in its own directory (files are staged).
- Supports resumption of previous partial runs.

Introduction to Nextflow

Nextflow Script

Exercise 1

You have an input file with 6 columns (see below), where column 2 is an ”index” column. Identify rows that have identical indexes (column 2) and remove them from the file.

Your input file looks like this:

```
11   11:189256   0   189256   A   G
11   11:193788   0   193788   T   C
11   11:194062   0   194062   T   C
11   11:194228   0   194228   A   G
11   11:193788   0   193788   A   C
```

Simple Example: Using BASH

Input is a file

- With 6 columns
- Column 2 is an index column
- Identify rows with identical field 2
- Remove identical rows

```
11   11:189256   0   189256   A   G
11   11:193788   0   193788   T   C
11   11:194062   0   194062   T   C
11   11:194228   0   194228   A   G
11   11:193788   0   193788   A   C
```

Using BASH:

```
cut -f 2 data/11.bim  | sort | uniq -d  > dups
grep -v -f dups data/11.bim > 11.clean
```

Simple Example: Using `nextflow`

```
#!/usr/bin/env nextflow
nextflow.enable.dsl=2

input_ch = Channel.fromPath("data/11.bim")

process getIDs {
    input:
    path(input_ch)

    output:
    path("ids"), emit: id_ch
    path("11.bim"), emit: orig_ch

    """
    cut -f 2 ${input_ch} | sort > ids
    """
}

process getDups {
    input:
    path(id_ch)

    output:
    path("dups"), emit: dups_ch

    """
    uniq -d ${id_ch} > dups
    touch ignore
    """
}
```

Simple Example: Using `nextflow`

```
#!/usr/bin/env nextflow
nextflow.enable.dsl=2

input_ch = Channel.fromPath("data/11.bim")

process getIDs {
    input:
    path(input_ch)

    output:
    path("ids"), emit: id_ch
    path("11.bim"), emit: orig_ch

    """
    cut -f 2 ${input_ch} | sort > ids
    """
}

process getDups {
    input:
    path(id_ch)

    output:
    path("dups"), emit: dups_ch

    """
    uniq -d ${id_ch} > dups
    touch ignore
    """
}
```

```
process removeDups {
    input:
    path(dups_ch)
    path(orig_ch)

    output:
    path("clean.bim"), emit: output

    """
    grep -v -f ${dups_ch} ${orig_ch} > clean.bim
    """
}

workflow {
    getIDs(input_ch)
    getDups(getIDs.out.id_ch)
    removeDups(getDups.out.dups_ch, getIDs.out.orig_ch).
       subscribe { print "Done!" }
}
```

Simple Example: Using `nextflow`

```
$ nextflow run cleandups.nf

N E X T F L O W  ~  version 19.04.1
Launching `cleandups.nf` [soggy_jennings] - revision: 795e2aa39d
[warm up] executor > local
executor >  local (3)
[84/7e1ad1] process > getIDs     [100%] 1 of 1 ▯
[19/cc8bf9] process > getDups    [100%] 1 of 1 ▯
[f9/ed086d] process > removeDups [100%] 1 of 1 ▯
Completed at: 31-Jul-2019 09:00:50
Duration    : 1.5s
CPU hours   : (a few seconds)
Succeeded   : 3
```

```
|--work
|  |--90
|  |  |--cebf3649d883f88381e32b4912b560
|  |  |  |--ids -> /Users/phele/day4/work/b3/aa0380f2a1bca447259b7ffd390083/ids
|  |  |  |--ignore
|  |--9c
|  |  |--e0cb7d8d26682d7d4a1c44392f2bb3
|  |  |  |--11.bim -> /Users/phele/day4/data/11.bim
|  |  |  |--clean.bim
|  |  |  |--dups -> /Users/phele/day4/work/90/cebf3649d883f88381e32b4912b560/dups
|  |--b3
|  |  |--aa0380f2a1bca447259b7ffd390083
|  |  |  |--11.bim -> /Users/phele/day4/data/11.bim
|  |  |  |--ids
```

Exercise 2

Change the script so that you use **`stdin`** or **`stdout`** in the **`getIDs`** and **`getDups`** processes to avoid the use of the temporary file ids. You can see the solution: ex2-cleandups-stdin.nf!.

Introduction to Nextflow

Partial Execution

Partial Execution

If execution of workflow is only partial

- Because of error
- Only need to resume from process that failed

```
nextflow run cleandups.nf -resume
```

Introduction to Nextflow

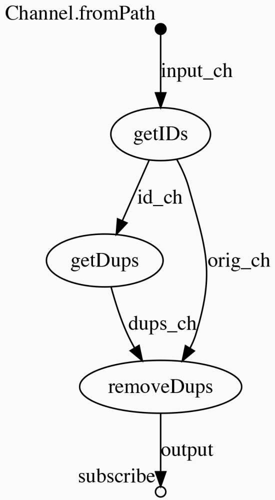
Visualising the Workflow

Visualising the Workflow

Nextflow supports several visualisation tools:

-with-dag

```
nextflow run cleandups.nf -with-dag <file-name>
```



Visualising the Workflow

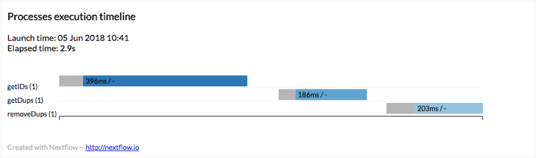
Nextflow supports several visualisation tools:

-with-dag

```
nextflow run cleandups.nf -with-dag <file-name>
```

-with-timeline

```
nextflow run cleandups.nf -with-timeline <file-name>
```



Processes execution timeline

Launch time: 05 Jun 2018 10:41
Elapsed time: 2.9s

Created with Nextflow -- http://nextflow.io

Visualising the Workflow

Nextflow supports several visualisation tools:

`-with-dag`

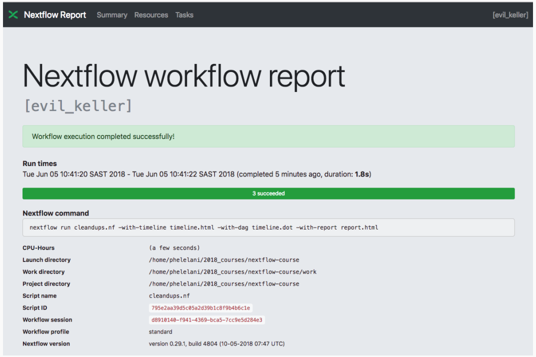
```
nextflow run cleandups.nf -with-dag <file-name>
```

`-with-timeline`

```
nextflow run cleandups.nf -with-timeline <file-name>
```

`-with-report`

```
nextflow run cleandups.nf -with-report <filename>
```



Nextflow workflow report

[evil_keller]

Workflow execution completed successfully!

Run times

Tue Jun 05 10:41:20 SAST 2018 - Tue Jun 05 10:41:22 SAST 2018 (completed 5 minutes ago, duration: **1.8s**)

3 succeeded

Nextflow command

```
nextflow run cleandups.nf -with-timeline timeline.html -with-dag timeline.dot -with-report report.html
```

CPU-Hours	(a few seconds)
Launch directory	/home/phelelani/2018_courses/nextflow-course
Work directory	/home/phelelani/2018_courses/nextflow-course/work
Project directory	/home/phelelani/2018_courses/nextflow-course
Script name	cleandups.nf
Script ID	795e2aa39d5c05a2d39b1c8f9b4b6c1e
Workflow session	d8910140-f941-4369-bca5-7cc9e5d284e3
Workflow profile	standard
Nextflow version	version 0.29.1, build 4804 (10-05-2018 07:47 UTC)

Generalising and Extending

Extending the Example

- Parameterise the input
- Want output to go to convenient place
- Workflow takes in multiple input files – processes are executed on each in turn.
- Complication : may need to carry the base name of the input to the final output;
- Can repeat some steps for different parameters.

Generalising and Extending

Parameters

Parameters

In Nextflow file:

```
input_ch = Channel.fromPath(params.data)
```

And run it like this

```
nextflow run phylo1.nf --data data/polyseqs.fa
```

Generalising and Extending

Channels

Data Types in Channels

Channels support different types:

- path
- stdin
- env
- tuple

Many, many operations you can do on channels and their contents

bind	buffer	close
filter	map/reduce	group
join, merge	mix	copy
split	spread	fork
count	min/max/sum	print/view

Creating Channels

```
Channel.create()
Channel.empty
Channel.from("blast","plink")
Channel.fromPath("data/*.fa")
Channel.fromFilePairs("data/{YRI,CEU,BEB}.*")
Channel.watchPath("*fa")
```

Generalising and Extending

Generalising Our Example

Workflow: Multiple Inputs

```
#!/usr/bin/env nextflow
nextflow.enable.dsl=2

params.data_dir = "data"
input_ch = Channel.fromPath("${params.data_dir}/*.bim")

process getIDs {
    input:
    path(input)

    output:
    path("${input.baseName}.ids"), emit: id_ch
    path("${input}"), emit: orig_ch

    """
    cut -f 2 ${input} | sort > ${input.baseName}.ids
    """
}

process getDups {
    input:
    path(input)

    output:
    path("${input.baseName}.dups"), emit: dups_ch

    """
    uniq -d ${input} > "${input.baseName}.dups"
    touch ignore
    """
}
```

Workflow: Multiple Inputs

```
#!/usr/bin/env nextflow
nextflow.enable.dsl=2

params.data_dir = "data"
input_ch = Channel.fromPath("${params.data_dir}/*.bim")

process getIDs {
    input:
    path(input)

    output:
    path("${input.baseName}.ids"), emit: id_ch
    path("${input}"), emit: orig_ch

    """
    cut -f 2 ${input} | sort > ${input.baseName}.ids
    """
}

process getDups {
    input:
    path(input)

    output:
    path("${input.baseName}.dups"), emit: dups_ch

    """
    uniq -d ${input} > "${input.baseName}.dups"
    touch ignore
    """
}
process removeDups {
    publishDir "output", pattern: "${badids.baseName}.bim",
       overwrite:true, mode:'copy'

    input:
    path(badids)
    path(orig)

    output:
    path("${badids.baseName}_clean.bim"), emit: cleaned_ch

    """
    grep -v -f ${badids} ${orig} > ${badids.baseName}_clean.bim
    """
}

workflow {
    getIDs(input_ch)
    getDups(getIDs.out.id_ch)
    removeDups(getDups.out.dups_ch, getIDs.out.orig_ch)
}
```

Workflow: Multiple Inputs

```
$ nextflow run cleandups.nf

Launching `cleandups.nf` [distracted_hodgkin] - revision: 29fdb384a6
[warm up] executor > local
executor >  local (9)
[1a/431eb7] process > getIDs     [100%] 3 of 3 ✔
[cc/fc0aaa] process > getDups    [100%] 3 of 3 ✔
[03/c31154] process > removeDups [100%] 3 of 3 ✔
Completed at: 31-Jul-2019 10:26:23
Duration    : 2s
CPU hours   : (a few seconds)
Succeeded   : 9
```

Exercise 4

Now try adding a process to our Nextflow example and for splitting the file but using different split values (solution: ex4-cleandups-multi-params.nf).

Workflow: Multiple Parameters

Now try splitting the file but use different split values

```
split  -l 400 data.txt dataX
```

will produce files dataXaa, dataXab, dataXac and so on ...

Try:

```
splits = [400,500,600]

process splitIDs  {
  input:
  path(bim)
  each split

  output:
  path("*-$split-*"), emit: output_ch

  """
  split -l ${split} ${bim} ${bim.baseName}-$split-
  """
}
```

Have a look at the modified Nextflow script: ex4-cleandups-multi-params-mod.nf.

Generalising and Extending

Managing Grouped Files

Grouped Files

Use PLINK as an example.

```
## Short version of the command
plink --bfile /path/YRI --freq --out /tmp/YRI

## Long version of the command
plink --bed YRI.bed \
    --bim YRI.bim \
    --fam YRI.fam \
    --freq \
    --out /tmp/YRI
```

Problem:

- Pass the files on another channel(s) to be staged
- Pass the base name as value/or work it out

Pros/Cons

- Simple
- Need extra channel/some gymnastics

Grouped Files

Use PLINK as an example.

```
## Short version of the command
plink --bfile /path/YRI --freq --out /tmp/YRI

## Long version of the command
plink --bed YRI.bed \
    --bim YRI.bim \
    --fam YRI.fam \
    --freq \
    --out /tmp/YRI
```

Problem:

- Pass the files on another channel(s) to be staged
- Pass the base name as value/or work it out

Pros/Cons

- Simple
- Need extra channel/some gymnastics

RECAP CLOSURES

Simply, a *closure* is an anonymous function

- Code wrapped in braces {, }
- Default argument called *it*

```
[1,2,3].each { print it * it }
[1,2,3].each { num -> print num * num }
```

Grouped Files - Version 1: map

```
#!/usr/bin/env nextflow
nextflow.enable.dsl=2

params.dir = "data/pops/"
dir = params.dir
params.pops = ["YRI","CEU","BEB"]

Channel
    .from(params.pops)
    .map { pop ->
        [ file("$dir/${pop}.bed"),
          file("$dir/${pop}.bim"),
          file("$dir/${pop}.fam")]
    }
    .set { plink_data }

plink_data.subscribe { println "$it" }
```

Grouped Files - Version 1: map

```
#!/usr/bin/env nextflow
nextflow.enable.dsl=2

params.dir = "data/pops/"
dir = params.dir
params.pops = ["YRI","CEU","BEB"]

Channel
    .from(params.pops)
    .map { pop ->
        [ file("$dir/${pop}.bed"),
          file("$dir/${pop}.bim"),
          file("$dir/${pop}.fam")]
    }
    .set { plink_data }

plink_data.subscribe { println "$it" }
```

```
[data/pops/YRI.bed, data/pops/YRI.bim, data/pops/YRI.fam]
[data/pops/CEU.bed, data/pops/CEU.bim, data/pops/CEU.fam]
[data/pops/BEB.bed, data/pops/BEB.bim, data/pops/BEB.fam]
```

Grouped Files - Version 1: map

```
#!/usr/bin/env nextflow
nextflow.enable.dsl=2

params.dir = "data/pops/"
dir = params.dir
params.pops = ["YRI","CEU","BEB"]

Channel
    .from(params.pops)
    .map { pop ->
        [ file("$dir/${pop}.bed"),
          file("$dir/${pop}.bim"),
          file("$dir/${pop}.fam")]
    }
    .set { plink_data }

plink_data.subscribe { println "$it" }
```

```
process getFreq {
  input:
  tuple path(bed), path(bim), path(fam)

  output:
  path("${bed.baseName}.frq"), emit result

  """
  plink --bed $bed \
    --bim $bim \
    --fam $fam \
    --freq \
    --out ${bed.baseName}"
  """
}

workflow {
  getFreq(plink_data).view()
}
```

```
[data/pops/YRI.bed, data/pops/YRI.bim, data/pops/YRI.fam]
[data/pops/CEU.bed, data/pops/CEU.bim, data/pops/CEU.fam]
[data/pops/BEB.bed, data/pops/BEB.bim, data/pops/BEB.fam]
```

Grouped Files - Version 2: `fromFilePairs`

Use `fromFilePairs`.

- Takes a closure used to gather files together with the same key

```
x_ch = Channel.fromFilePairs( files ) { closure }
```

- Specify the files as a glob
- Closure associates each file with a key
- `fromPairs` puts all files with same key together
- Returns a list of pairs (key, list)

Grouped Files - Version 2: `fromFilePairs`

Use `fromFilePairs`.

- Takes a closure used to gather files together with the same key

```
x_ch = Channel.fromFilePairs( files ) { closure }
```

- Specify the files as a glob
- Closure associates each file with a key
- `fromPairs` puts all files with same key together
- Returns a list of pairs (key, list)

```
#!/usr/bin/env nextflow

commands = Channel.fromFilePairs("/usr/bin/*", size:-1) {
                it.baseName[0]
           }

commands.subscribe { k= it[0];
  n=it[1].size();
  println "There are $n files starting with $k";
}
```

A more complex example – default closure

```
Channel
    .fromFilePairs
        ("${params.dir}/*.{bed,fam,bim}",size:3, flat : true)
    .ifEmpty { error "No matching plink files" }
    .set { plink_data }

plink_data.subscribe { println "$it" }
```

Grouped Files - Version 2: `fromFilePairs`

Use `fromFilePairs`.

- Takes a closure used to gather files together with the same key

```
x_ch = Channel.fromFilePairs( files ) { closure }
```

- Specify the files as a glob
- Closure associates each file with a key
- `fromPairs` puts all files with same key together
- Returns a list of pairs (key, list)

```
#!/usr/bin/env nextflow

commands = Channel.fromFilePairs("/usr/bin/*", size:-1) {
                it.baseName[0]
          }

commands.subscribe { k= it[0];
  n=it[1].size();
  println "There are $n files starting with $k";
}
```

A more complex example – default closure

```
Channel
    .fromFilePairs
        ("${params.dir}/*.{bed,fam,bim}",size:3, flat : true)
    .ifEmpty { error "No matching plink files" }
    .set { plink_data }

plink_data.subscribe { println "$it" }
```

```
[CEU, [data/pops/CEU.bed, data/pops/CEU.bim, data/pops/CEU.fam]]
[YRI, [data/pops/YRI.bed, data/pops/YRI.bim, data/pops/YRI.fam]]
[BEB, [data/pops/BEB.bed, data/pops/BEB.bim, data/pops/BEB.fam]]
```

Grouped Files - Version 2: `fromFilePairs`

```
process checkData {
    input:
    tuple val(pop), path(pl_files)

    output:
    path("${pl_files[0]}.frq"), emit: result

    """
    plink --bfile $base --freq --out pl_files[0].baseName
    """
}
```

Grouped Files - Version 2: `fromFilePairs`

```
process checkData {
    input:
    tuple val(pop), path(pl_files)

    output:
    path("${pl_files[0]}.frq"), emit: result

    """
    plink --bfile $base --freq --out pl_files[0].baseName
    """
}
```

```
process checkData {
    input:
    tuple val(pop), path(pl_files)

    output:
    path("${pop}.frq"), emit: result

    """
    plink --bfile $pop --freq  --out $pop
    """
}
```

Grouped Files - Final Version

```
#!/usr/bin/env nextflow
nextflow.enable.dsl=2

params.dir = "data/pops/"
dir = params.dir
params.pops = ["YRI","CEU","BEB"]

Channel
    .fromFilePairs("${params.dir}/{YRI,BEB,CEU}.{bed,bim,fam}",size:3) {
        file -> file.baseName
    }
    .filter { key, files -> key in params.pops }
    .set { plink_data }

process checkData {
    input:
    tuple val(pop), path(pl_files)

    output:
    path("${pop}.frq"), emit: result

    """
    plink --bfile $pop --freq  --out $pop
    """

workflow {
  checkData(plink_data).view()
}
```

Exercise 5

Have a look at ex5-weather.nf. In the data directory are set of data files for different years and months. First, I want you to use paste to combine all the files for the same year and month (paste joins files horizontal-wise). Then these new files should be concated.

Generalising and Extending

On absolute paths

Absolute paths

```
input = Channel.fromPath("/data/batch1/myfile.fa")

process show {
    input:
    path(data)

    output:
    path('see.out')

    """
    cp ${data} /home/scott/answer
    """
    ...
```

Nextflow and Docker

Nextflow and Docker

Docker & Singularity Containers

Docker & Singularity Containers

Light-weight virtualisation abstraction layer

- Currently runs on Unix like systems
  - Linux
  - macOS
- Windows support coming

Can create images locally or get from repositories

```
## Docker
docker pull ubuntu
docker pull quay.io/banshee1221/h3agwas-plink

## Singularity
singularity pull docker://ubuntu
singularity pull docker://quay.io/banshee1221/h3agwas-plink
```

Running images

```
## Docker
docker run <some-image-name>

## Singularity
singularity exec <some-image-name>
```

- Docker/Singularity often run images in background
- Can also run interactively

```
## Running Docker interactively
sudo docker run -t -i quay.io/banshee1221/h3agwas-plink

## Running Singularity interactively
singularity shell docker://quay.io/banshee1221/h3agwas-plink
```

Nextflow supports Docker & Singularity

- Well designed script should be highly portable
- Each process gets run as a separate image call
  - Under the hood, a `docker run` or a `singularity exec` is called
- Can use the same or different images for each process
  - Parameterisable

Assuming all processes use the same image:

```
## For Docker
nextflow run plink2.nf -with-docker quay.io/banshee1221/h3agwas-plink

## For Singularity
nextflow run plink.nf -with-singularity docker://quay.io/banshee1221/h3agwas-plink
```

Nextflow and Docker

Directory & File Access

Directory & File access

Nextflow Docker/Singularity support highly transparent – but pay attention to good practice

- For each process Docker/Singularity mounts the work directory for **that** process on the Docker/Singularity image.
- Files can be staged in and out using Nextflow mechanisms.
- Other files available: directories mounted through Docker/Singularity run time options or on the Docker image
- No other files on the host machine including the current directory
- Process executes in the Docker/Singularity environment

Directory & File access

```
#!/usr/bin/env nextflow
nextflow.enable.dsl=2

data = Channel.fromPath("data/pops/YRI.bim")

process see {
    publishDir "count_out", overwrite:true, mode:'move'
    echo true

    input:
    path(bim)

    output:
    path(count)

    """
    hostname
    echo "Path is \$( pwd )\n "
    echo "Parent directory has \$( ls .. )\n"
    echo "My home directory has \$( ls /home/phele )\n"
    wc -l ${bim} > count
    ls
    """
}

workflow {
    see(data)
}
```

```
N E X T F L O W  ~  version 0.21.2
Launching show_env.nf
[warm up] executor > local
[94/597f09] Submitted process > see (1)
89ad448ae0b2
Path is /home/scott/witsGWAS/dockerized/work/94/597f09ca6cc01c7be
Parent directory has 597f09ca6cc01c7be
My home directory has witsGWAS

YRI.bim
count
```

Directory & File access

Note that although the script's pwd shows:
`/home/scott/witsGWAS/dockerized/work/94/597f09ca6cc01c7be`

- Only these specific directories are mounted
- Only the files in the innermost directory are available

Any absolute paths (other than those used in staging) will result in error.

Profiles

In nextflow.config

```
profiles {
  ...
  docker {
    process.container = 'quay.io/banshee1221/h3agwas-plink:latest'
    docker.enabled = true
  }
}
```

Now can run as:

```
nextflow run gwas.nf -profile docker
```

This can be extended in many ways

- Different processes can use different containers
- Can mount other host directories
- Can pass arbitrary Docker parameters

Executors

Executors

Executors

Executors

A Nextflow *executor* is the mechanism which Nexflow runs the code in each of the processes

- Default is `local`: process is run as a script

Many others

- PBS/Torque
- SLURM
- Amazon (AWS Batch)
- SGE (Sun Grid Engine)

Selecting an executor Annotating each process

- `executor` directive, e.g. `executor 'pbs'`
- resource constraints

Or, `nextflow.config` file

- either global or per-process

Executors

Nextflow on a cluster (HPC)

Running Nextflow on a cluster (HPC)

Script runs on the *head* node

- Nextflow uses the `executor` information to decide how the job should run
- Each process can be handled differently
- Nextflow submits each process to the job scheduler on your behalf (e.g, if using PBS/Torque, `qsub` is done)

Example

```
process {
  executor = 'pbs'
  queue = 'batch'
  scratch = true
  cpus = 5
  memory = '2GB'
}
```

Executors

Scheduler + Docker

Scheduler + Docker

```
process.container = 'quay.io/banshee1221/h3agwas-plink:latest'
docker.enabled = false

process {
  executor = 'pbs'
  queue = 'batch'
  scratch = true
  cpus = 5
  memory = '2GB'
}
```

Executors

Amazon EC2

Amazon EC2

Netflow has native support for EC2

- You need an account on EC2
- Image (AMI) with the appropriate support

Launch your code:

```
nextflow cloud create GenomeCloud -c 5
```

If successful, Nextflow will give you the name of the headnode of your cluster

- `ssh` into into it
- run Nextflow on it.

Afterwards shut down:

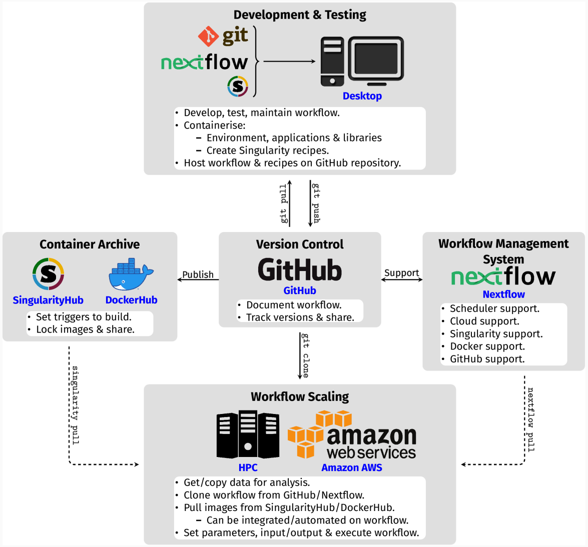
```
nextflow shutdown GenomeCloud
```

How it All Fits

Bioinformatics Workflows - Challenges & Solutions

• Multiple softwares • Repeating analyses • Handling intermediate files	→ nextflow ←	Workflow Management Systems
• Software installation • Software dependencies • Software version control • Platform dependencies	→ S docker ←	Software Containerisation & Portability
• Laptop/desktop computer • Parallelisation • Cluster environment • Cloud environment	→ amazon web services ←	Workflow Scaling & Computational Resources
• Documentation Version control • Automation • Publishing	→ git GitHub ←	Workflow Documentation & Sharing

Bioinformatics Workflows - Best Practices



Development & Testing

- Develop, test, maintain workflow.
- Containerise:
  - Environment, applications & libraries
  - Create Singularity recipes.
- Host workflow & recipes on GitHub repository.

Container Archive

- Set triggers to build.
- Lock images & share.

Version Control

- Document workflow.
- Track versions & share.

Workflow Management System

- Scheduler support.
- Cloud support.
- Singularity support.
- Docker support.
- GitHub support.

Workflow Scaling

- Get/copy data for analysis.
- Clone workflow from GitHub/Nextflow.
- Pull images from SingularityHub/DockerHub.
  - Can be integrated/automated on workflow.
- Set parameters, input/output & execute workflow.