

# 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

- 1 Introduction to Nextflow
  - Introduction
  - Nextflow Script
  - Partial Execution
  - Visualising the Workflow
- 2 Generalising and Extending
  - Parameters
  - Channels
  - Generalising Our Example
  - Managing Grouped Files
  - On absolute paths
- 3 Nextflow and Docker
  - Docker & Singularity Containers
  - Directory & File Access
- 4 Executors
  - Executors
  - Nextflow on a cluster (HPC)
  - Scheduler + Docker
  - Amazon EC2

## Introduction to Nextflow

---

# Introduction to Nextflow

---

## Introduction

## Resources

- <https://github.com/phelelani/nf-tutorial>

## 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

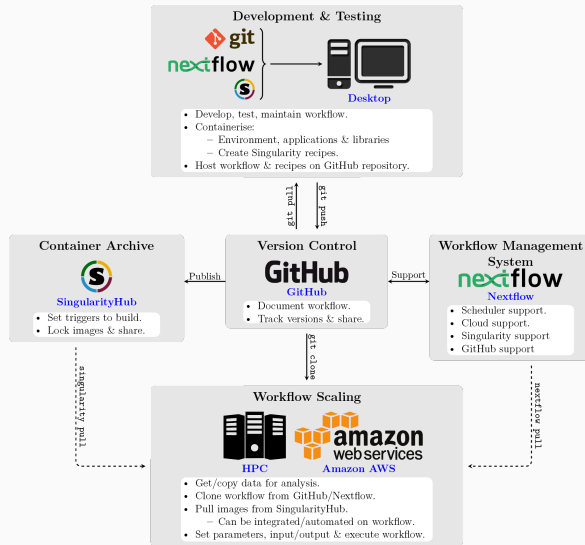
# Workflow Languages

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

## 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

```
1  #!/usr/bin/env nextflow
2  nextflow.enable.dsl=2
3
4  input_ch = Channel.fromPath("data/11.bim")
5
6  process getIDs {
7      input:
8      path(input_ch)
9
10     output:
11     path("ids"), emit: id_ch
12     path("11.bim"), emit: orig_ch
13
14     """
15     cut -f 2 ${input_ch} | sort > ids
16     """
17 }
18
19 process getDups {
20     input:
21     path(id_ch)
22
23     output:
24     path("dups"), emit: dups_ch
25
26     """
27     uniq -d ${id_ch} > dups
28     touch ignore
29     """
30 }
```

## Simple Example: Using nextflow

```
1  #!/usr/bin/env nextflow
2  nextflow.enable.dsl=2
3
4  input_ch = Channel.fromPath("data/11.bim")
5
6  process getIDs {
7      input:
8      path(input_ch)
9
10     output:
11     path("ids"), emit: id_ch
12     path("11.bim"), emit: orig_ch
13
14     """
15     cut -f 2 ${input_ch} | sort > ids
16     """
17 }
18
19 process getDups {
20     input:
21     path(id_ch)
22
23     output:
24     path("dups"), emit: dups_ch
25
26     """
27     uniq -d ${id_ch} > dups
28     touch ignore
29     """
30 }
```

```
31 process removeDups {
32     input:
33     path(dups_ch)
34     path(orig_ch)
35
36     output:
37     path("clean.bim"), emit: output
38
39     """
40     grep -v -f ${dups_ch} ${orig_ch} > clean.bim
41     """
42 }
43
44 workflow {
45     getIDs(input_ch)
46     getDups(getIDs.out.id_ch)
47     removeDups(getDups.out.dups_ch, getIDs.out.orig_ch).
48         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](https://github.com/nextflow-io/nextflow/blob/master/examples/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

---

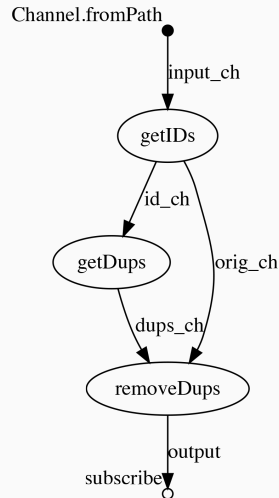
### Visualising the Workflow

# 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>
```

The screenshot displays a web-based report titled "Nextflow Report" with tabs for Summary, Resources, and Tasks. The user is identified as [evil\_keller]. The main heading is "Nextflow workflow report" followed by the user name. A green status bar indicates "Workflow execution completed successfully!". Below this, the "Run times" section shows the execution period from Tue Jun 05 10:41:20 SAST 2018 to Tue Jun 05 10:41:22 SAST 2018, completed 5 minutes ago with a duration of 1.8s. A green progress bar shows "3 succeeded". The "Nextflow command" section contains the command: `nextflow run cleandups.nf -with-timeline timeline.html -with-dag timeline.dot -with-report report.html`. At the bottom, a table lists various metadata:

CPU-Hours	(a few seconds)
Launch directory	/home/phelani/2018_courses/nextflow-course
Work directory	/home/phelani/2018_courses/nextflow-course/work
Project directory	/home/phelani/2018_courses/nextflow-course
Script name	cleandups.nf
Script ID	705c0ee79d5c8fa2d2961cd9b48dc1e
Workflow session	68916148-f943-4369-bca5-7c8e5d29e3
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

```
1  #!/usr/bin/env nextflow
2  nextflow.enable.dsl=2
3
4  params.data_dir = "data"
5  input_ch = Channel.fromPath("${params.data_dir}/*.bim")
6
7  process getIDs {
8      input:
9      path(input)
10
11      output:
12      path("${input.baseName}.ids"), emit: id_ch
13      path("${input}"), emit: orig_ch
14
15      """
16      cut -f 2 ${input} | sort > ${input.baseName}.ids
17      """
18  }
19
20  process getDups {
21      input:
22      path(input)
23
24      output:
25      path("${input.baseName}.dups"), emit: dups_ch
26
27      """
28      uniq -d ${input} > "${input.baseName}.dups"
29      touch ignore
30      """
31  }
```

## Workflow: Multiple Inputs

```
1  #!/usr/bin/env nextflow
2  nextflow.enable.dsl=2
3
4  params.data_dir = "data"
5  input_ch = Channel.fromPath("${params.data_dir}/*.bim")
6
7  process getIDs {
8      input:
9          path(input)
10
11      output:
12          path("${input.baseName}.ids"), emit: id_ch
13          path("${input}"), emit: orig_ch
14
15      """
16      cut -f 2 ${input} | sort > ${input.baseName}.ids
17      """
18  }
19
20  process getDups {
21      input:
22          path(input)
23
24      output:
25          path("${input.baseName}.dups"), emit: dups_ch
26
27      """
28      uniq -d ${input} > "${input.baseName}.dups"
29      touch ignore
30      """
31  }
```

```
32  process removeDups {
33      publishDir "output", pattern: "${badids.baseName}.bim",
34          overwrite:true, mode:'copy'
35
36      input:
37          path(badids)
38          path(orig)
39
40      output:
41          path("${badids.baseName}_clean.bim"), emit: cleaned_ch
42
43      """
44      grep -v -f ${badids} ${orig} > ${badids.baseName}_clean.bim
45      """
46  }
47
48  workflow {
49      getIDs(input_ch)
50      getDups(getIDs.out.id_ch)
51      removeDups(getDups.out.dups_ch, getIDs.out.orig_ch)
52  }
```

## 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:

```
1 splits = [400,500,600]
2
3 process splitIDs {
4   input:
5     path(bim)
6   each split
7
8   output:
9     path("*-${split-*}"), emit: output_ch
10
11   """
12   split -l ${split} ${bim} ${bim.baseName}-${split}-
13   """
14 }
```

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

```
1  #!/usr/bin/env nextflow
2  nextflow.enable.dsl=2
3
4  params.dir = "data/pops/"
5  dir = params.dir
6  params.pops = ["YRI", "CEU", "BEB"]
7
8  Channel
9      .from(params.pops)
10     .map { pop ->
11         [ file("${dir}/${pop}.bed"),
12           file("${dir}/${pop}.bim"),
13           file("${dir}/${pop}.fam") ]
14     }
15     .set { plink_data }
16
17 plink_data.subscribe { println "$it" }
```

## Grouped Files - Version 1: map

```
1  #!/usr/bin/env nextflow
2  nextflow.enable.dsl=2
3
4  params.dir = "data/pops/"
5  dir = params.dir
6  params.pops = ["YRI", "CEU", "BEB"]
7
8  Channel
9      .from(params.pops)
10     .map { pop ->
11         [ file("${dir}/${pop}.bed"),
12           file("${dir}/${pop}.bim"),
13           file("${dir}/${pop}.fam") ]
14     }
15     .set { plink_data }
16
17 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

```
1  #!/usr/bin/env nextflow
2  nextflow.enable.dsl=2
3
4  params.dir = "data/pops/"
5  dir = params.dir
6  params.pops = ["YRI", "CEU", "BEB"]
7
8  Channel
9      .from(params.pops)
10     .map { pop ->
11         [ file("$dir/${pop}.bed"),
12           file("$dir/${pop}.bim"),
13           file("$dir/${pop}.fam")]
14     }
15     .set { plink_data }
16
17 plink_data.subscribe { println "$it" }
```

```
16  process getFreq {
17      input:
18          tuple path(bed), path(bim), path(fam)
19
20      output:
21          path("${bed.baseName}.frq"), emit result
22
23      """
24      plink --bed $bed \
25            --bim $bim \
26            --fam $fam \
27            --freq \
28            --out "${bed.baseName}"
29      """
30  }
31
32  workflow {
33      getFreq(plink_data).view()
34  }
```

```
[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)

```
1  #!/usr/bin/env nextflow
2
3  commands = Channel.fromFilePairs("/usr/bin/*", size:-1) {
4      it.baseName[0]
5  }
6
7  commands.subscribe { k= it[0];
8      n=it[1].size();
9      println "There are $n files starting with $k";
10 }
```

A more complex example – default closure

```
1  Channel
2      .fromFilePairs
3      ("${params.dir}/*.{bed,fam,bim}",size:3, flat : true)
4      .ifEmpty { error "No matching plink files" }
5      .set { plink_data }
6
7  plink_data.subscribe { println "$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)

```
[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]]
```

```
1  #!/usr/bin/env nextflow  
2  
3  commands = Channel.fromFilePairs("/usr/bin/*", size:-1) {  
4      it.baseName[0]  
5  }  
6  
7  commands.subscribe { k= it[0];  
8      n=it[1].size();  
9      println "There are $n files starting with $k";  
10 }
```

A more complex example – default closure

```
1  Channel  
2      .fromFilePairs  
3      ("${params.dir}/*.{bed,fam,bim}",size:3, flat : true)  
4      .ifEmpty { error "No matching plink files" }  
5      .set { plink_data }  
6  
7  plink_data.subscribe { println "$it" }
```

## Grouped Files - Version 2: fromFilePairs

```
1  process checkData {  
2      input:  
3      tuple val(pop), path(pl_files)  
4  
5      output:  
6      path("${pl_files[0]}.frq"), emit: result  
7  
8      """  
9      plink --bfile $base --freq --out pl_files[0].baseName  
10     """  
11 }
```

## Grouped Files - Version 2: fromFilePairs

```
1 process checkData {  
2   input:  
3   tuple val(pop), path(pl_files)  
4  
5   output:  
6   path("${pl_files[0]}.frq"), emit: result  
7  
8   """  
9   plink --bfile $base --freq --out pl_files[0].baseName  
10  """  
11 }
```

```
1 process checkData {  
2   input:  
3   tuple val(pop), path(pl_files)  
4  
5   output:  
6   path("${pop}.frq"), emit: result  
7  
8   """  
9   plink --bfile $pop --freq --out $pop  
10  """  
11 }
```

## Grouped Files - Final Version

```
1  #!/usr/bin/env nextflow
2  nextflow.enable.dsl=2
3
4  params.dir = "data/pops/"
5  dir = params.dir
6  params.pops = ["YRI", "CEU", "BEB"]
7
8  Channel
9      .fromFilePairs("${params.dir}/{YRI,BEB,CEU}.{bed,bim,fam}",size:3) {
10      file -> file.baseName
11      }
12      .filter { key, files -> key in params.pops }
13      .set { plink_data }
14
15  process checkData {
16      input:
17      tuple val(pop), path(pl_files)
18
19      output:
20      path("${pop}.frq"), emit: result
21
22      """
23      plink --bfile $pop --freq --out $pop
24      """
25
26  workflow {
27      checkData(plink_data).view()
28  }
```

## 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

```
1 input = Channel.fromPath("/data/batch1/myfile.fa")
2
3 process show {
4     input:
5     path(data)
6
7     output:
8     path('see.out')
9
10    """
11    cp ${data} /home/scott/answer
12    """
13    ...
```

## 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
```

- 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
```

Running images

### ## Docker

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

### ## Singularity

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

## 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

```
1  #!/usr/bin/env nextflow
2  nextflow.enable.dsl=2
3
4  data = Channel.fromPath("data/pops/YRI.bim")
5
6  process see {
7      publishDir "count_out", overwrite:true, mode:'move'
8      echo true
9
10     input:
11     path(bim)
12
13     output:
14     path(count)
15
16     """
17     hostname
18     echo "Path is \$( pwd )\n "
19     echo "Parent directory has \$( ls .. )\n"
20     echo "My home directory has \$( ls /home/phele )\n"
21     wc -l ${bim} > count
22     ls
23     """
24 }
25
26 workflow {
27     see(data)
28 }
```

```
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

```
1 profiles {  
2   ...  
3   docker {  
4     process.container = 'quay.io/banshee1221/h3agwas-plink:latest'  
5     docker.enabled = true  
6   }  
7 }
```

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 Nextflow 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

```
1 process {  
2   executor = 'pbs'  
3   queue = 'batch'  
4   scratch = true  
5   cpus = 5  
6   memory = '2GB'  
7 }
```

Executors

---

Scheduler + Docker



## Scheduler + Docker

```
1 process.container = 'quay.io/banshee1221/h3agwas-plink:latest'
2 docker.enabled = false
3
4 process {
5     executor = 'pbs'
6     queue = 'batch'
7     scratch = true
8     cpus = 5
9     memory = '2GB'
10 }
```

## 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
```