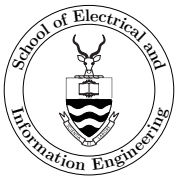


Nextflow: a tutorial through examples

Scott Hazelhurst

December 2016

1 Introduction

This talk in an introduction to Nextflow primarily through examples. Since the talk of brief, it is designed to whet your appetite – we're only going to dip in and out of some of its features in a superficial way.

- `www.bioinf.wits.ac.za/courses/nextflow`
- http://github.com/shaze/nextflow-course
- `https://www.nextflow.io/docs/latest/index.html`

Exercise: Throughout this tutorial there will be some practical examples. Not all will be covered in class for time reasons but you can come back and do them.

- git clone http://github.com/shaze/nextflow-course

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 for expressing workflows

- Portable – works on most Unix-like systems
- Very easy to install (NB: requires Java 7, 8)
- Scalable
- Supports Docker
- Supports a range of scheduling systems

Key concepts

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

Simple example

Input is a file – with 6 columns – column 2 is an index column

- If there are any rows with identical field 2, remove

```
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
```

This is easy to do from the shell – very simple example, not realistic for Nextflow

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

Simple nextflow program

```
#!/usr/bin/env nexflow

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

Here we create a new channel, and put one value into the channel – a file name.

```
process getIDs {
    input:
       file input from input_ch
    output:
       file "ids" into id_ch
       file "11.bim" into orig_ch
    script:
       " cut -f 2 $input | sort > ids "
}



process getDups {
    input:
       file input from id_ch
    output:
       file "dups" into dups_ch
    script:
       """
           uniq -d $input > dups
           touch ignore
       """
}


process removeDups {
    input:
       file badids  from dups_ch
       file orig    from orig_ch
    output:
       file "clean.bim" into output
    script:
       "grep -v -f $badids $orig > clean.bim "
}

output.subscribe { print "Done!" }
```

Note, the use of Nextflow variables: Within a double quoted string, there is string interpolation marked with the $. If you want to access a system environment variable you need to also escape with a backslash. So in the Nextflow program, you can normally just refer to Nextflow variables unadorned with their names (e.g. `input`) and environment variables with a dollar (e.g. `$HOME`) but within a double/triple-quoted string it's `\$input` and `\$HOME`.

File names can be relative (to the current working directory where the script is being run in, not to the location of the script), or absolute. Great care needs to be taken with using absolute path names since this reduces the portability of scripts, particualarly when you are using Docker.

We can now run our script:

```
$ ./cleandups.nf
```

```
N E X T F L O W  ~  version 0.22.3
Launching ‘cleandups.nf‘ [determined_blackwell] - revision: 795e2aa39d
[warm up] executor > local
[8b/e32746] Submitted process > getIDs (1)
[79/aee730] Submitted process > getDups (1)
[ad/2d955d] Submitted process > removeDups (1)
```

Note that nextflow creates a *work* directory, and inside of that are the working directories of each process – in the example above you can see that the *getIDs* process was launched in a directory with a prefix e32746, inside the directory 8b..

```
$ ls

cleandups.nf   data      work

work
-- 79
 |--- aee730e58313287261c8ff9bd9a22e
   |- dups
   |- ids
   |- ignore
-- 8b
 |--- e32746b56a4683ab5044aa59bc4ecd
   |- 11.bim
   |- ids
-- ad
 |--- 2d955d384560ed010709b121d6638f/
   |- 11.bim
   |- clean.bim
   |- dups
```

The names of the working directory are randomly chosen so if you run it, you will get different names. Also, each time you run a process ever, it will get a unique working directory. There is no danger of name clashes.

Instead of naming the file you get from a channel you can also

- specify *stdin* if your process expects data to come from `stdin` rather than a named file. Nextflow will pipe the file to standard input;
- specify *stdout* if your process produces data on `stdout` and you want that data to go into the channel.

Exercise: 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*. To see the solution, you can say `git checkout simple-std`

1.1 Partial execution

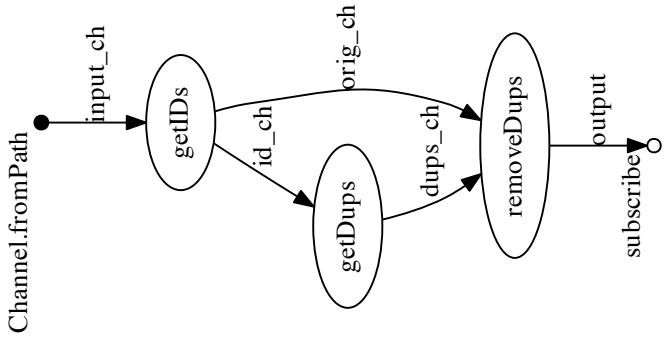
If execution of workflow is only partial (e.g., because of error), only need to resume from process that failed

```
./cleandups -resume
```

1.2 Visualising the workflow: I

Nextflow supports several visualisation tools – we'll explore one now.

```
nextflow cleandups.nf -with-dag
```



2 Groovy

Nextflow is a DSL built with Groovy

- Can inter-mix Nextflow, Groovy and Java code
- Very powerful, flexible
- Don't need to know much (any?) Groovy but a little knowledge is a powerful thing

Here we'll do some cookbook Groovy...

Closures

Closures are anonymous functions – similar to lambdas in Python.

- Don't want the overhead of naming a function we onlly use once
- Typically use with higher-order functions – functions that take other functions as arguments
  Very powerful and useful

Syntax for a closure that takes one argument:

```
{ parm -> expression }
```

This is an anonymous function that takes one parameter (I've called it *parm*, you can call it whatever you want) and *expression* is a valid expression, probably including the parameter. Let's look at some examples.

```
{ a -> a*a } (3)

{ a -> a*a+7*a - 2 } (3)

for (n in 1..5) print( {it*it} (n));

{ x, y ->  Math.sqrt(x*x + y*y) } (3,4)
```

OK, I am simplifying a bit here. Closures are a bit more than functions.

Now what we have seen so far isn't so useful but the power comes when we have a function that take another function. Consider this very simplistic example. Suppose we have a program where we are manipulating lists of numbers. Sometimes we want to sum a list, sometimes we want to sum the squares of the numbers; sometimes we want to sum the cubes of the cosines of the numbers. The business of going through the list and summing is the same in all cases. What differs is what we do to the numbers in each case – so rather than have a separate procedure for each type of summation, we just have one. But we pass this procedure a function that says what do do

```
int doX(f, nums) {
    sum=0;
    for ( n in nums ) {
       sum = sum+f(n);
    }
    return sum
}
```

We can call it thus:

```
print doX ( {a->a},   [4,5,16] );

print doX ( {a->a*a}, [4,5,16] );

print doX ( { it*it }, [4,5,16]);

m=10
print doX({a->m*a+2}, [1,2,3])
```

Note how you don't have to name the parameter – if you don't name the parameter then the name *it* is assumed.

Exercise Look at the sample Groovy code – you can run it with nextflow `git checkout groovy`

3 Generalising and extending

We'll now extend this example, introducing more powerful features of Nextflow as well as some of the complication of workflow design. x

- 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.

3.1 Parameters

In Nextflow file:

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

And run it like this

```
./phylo1.nf --data_dir data/polyseqs.fa
```

During debugging you may want to specify default values.

```
params.data_dir = 'data'
```

If you run the Nextflow program without parameters, it will use this as a default value; if you give it parameters, the default value is replaced. Of course, as a matter of good practice, default values for parameters are really designed for real parameters of the process (like gap penalties in a multiple sequence alignment) rather than data files.

Nextflow makes a distinction between parameters with a single dash (-) and with a double dash (--). The single dash ones are from a small, language defined subset modifying the behaviour of *Nextflow* — for example we've seen `-with-dag` already.

The double-dash parameters are user-defined and completely extensible – they are used to populate *params*. They modify the behaviour of *your* program.

3.2 Nexflow channels...

Channels support different types:

- file
- val
- set

NB: *val* is the most generic – could be a file name. But sending a *file* provides power since you can access Groovy's file handling capacity *and*, more importantly does staging of files.

Creating channels

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

There are others. Note that the *fromPath* method takes a Unix *glob* and creates a new channel which has all the files that match the glob. These files are then emitted one by one to processes that use these values. This default semantics can be changed using the channel operators that Nexflow provides, some of which are shown below.

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

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

3.3 Generalising our example...

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

process getIDs {
    input:
       file input from input_ch
    output:
       file "${input.baseName}.ids" into id_ch
       file "$input" into orig_ch
    script:
       " cut -f 2 $input | sort > ${input.baseName}.ids "
}
```

Here the *getIDs* process will execute once, for each file found in the initial glob. On a machine with multiple cores, these would probably execute in parallel, and as we'll see later if you are running on the head node of a cluster, each could run as a separate job.

Note that in this version of *getIDs* we name the output file dependant on the input file. This is convenient to do because now we are taking many input files. There is no danger of there being any name clashes during execution because each parallel execution of *getIDs* runs in a separate local directory. However, at the end we want to be able to distinguish which output came from which input without having to do detective work – so we name the files conveniently. Files that get created on the way but don't need at the end we can name boringly.

```
process getDups {
    input:
       file input from id_ch
    output:
```

```
       file "${input.baseName}.dups" into dups_ch
    script:
       out = "${input.baseName}.dups"
       """
       uniq -d $input > $out
       touch ignore
       """
}

process removeDups {
    input:
       file badids  from dups_ch
       file "orig.bim"    from orig_ch
    output:
       file "${badids.baseName}.bim" into cleaned_ch
    publishDir "output", pattern: "${badids.baseName}.bim",\
                  overwrite:true, mode:'copy'

    script:
       "grep -v -f $badids orig.bim > ${badids.baseName}.bim "
}
```

```
scott$ ./cleandups.nf
N E X T F L O W  ~  version 0.22.3
Launching `./cleandups.nf` [jovial_feynman] - revision: f968836869
[warm up] executor > local
[b0/21926b] Submitted process > getIDs (3)
[6e/5462ff] Submitted process > getIDs (1)
[b9/7345e2] Submitted process > getIDs (2)
[79/8f751f] Submitted process > getDups (1)
[94/f6a5f8] Submitted process > getDups (2)
[3b/abe610] Submitted process > removeDups (1)
[03/418526] Submitted process > removeDups (2)
Done![ec/82bc45] Submitted process > getDups (3)
Done![9d/f91454] Submitted process > removeDups (3)
```

Now I'm going to add a next step – say we want to split the IDs into groups using *split* but try different values of splitting
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 ...

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

process splitIDs  {
    input:
```

```
       file bim from cleaned_ch
    each split from splits
    output:
       file ("*-$split-*") into output_ch;

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

Exercise Do a `git checkout channels` and understand and play with the script

3.4 Managing grouped files

We've seen so far where we have a stream of file being processed independently. But in many applications there may be matched data sets.

We'll now look at an example, using a popular bioinformatics tool called PLINK. In its most commn usages, PLINK takes in three related files, typically with the same but different suffixed: .bed, .bim, .fam

Use PLINK as an example.

```
plink --bfile /path/YRI --freq --out /tmp/YRI
plink --bed YRI.bed --bim YRI.bim --fam YRI.fam --freq \
      --out /tmp/YRI
```

If you don't know what PLINK does, don't worry. It's the Swiss Army knife for bioinformatics. The above commands are equivalent (the first is the short-hand for the second when the bed, bim, and fam files have the same base). The command finds frequencies of genome variations – the output in this example will go into a file called *YRI.frq*.

Version 0

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 – Groovy 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 }
```

Similar to lambdas in Python and Java.

Version 1

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

This example takes a stream of values from *params.pops* and for each value (that's what map does) it applies a closure that takes a string and produces a tuple of files. That tuple is then bound to a channel called `plink_data`

Note that there are two **distinct** uses of the *set*.

- As a channel operator as shown here
- In an input/output clause of a channel

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

```
[/popdata/1k-2014/pops/YRI.bed, /popdata/1k-2014/pops/YRI.bim, /popdata/1k-2014/pops/YRI.fam]
[/popdata/1k-2014/pops/CEU.bed, /popdata/1k-2014/pops/CEU.bim, /popdata/1k-2014/pops/CEU.fam]
[/popdata/1k-2014/pops/BEB.bed, /popdata/1k-2014/pops/BEB.bim, /popdata/1k-2014/pops/BEB.fam]
```

Now let's look at more realistic exammple. To try it on your own computer do `git checkout plink1`

NB: Since you may not have plink on your computer, our code actually fakes the output. If you do have plink you can make the necessary changes.

```
process getFreq {
   input:
     set file(bed), file(bim), file(fam) from plink_data
   output:
     file "${bed.baseName}.frq" into result

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

Look at *plink1B.nf*. It's a slightly different ways of doing things. On examples of this size, none of these options are much better or worse but it's useful to see different ways of doing things for later.

Version 2

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"; }
```

Here we use standard globbing to find all the files in the /usr/bin directory in the *marks* directory. The closure takes the first letter of each file – all the files with the same letter are put together. The size parameter says how many we put togther : -1 means all.

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 }
```

fromFilePairs

- Matches the glob
- The first * is taken as the matching key
- For each unique match of the * we return
  - The thing that matches the *
  - The list of files that match the glob with that item
  - Up to 3 matching files (the default of size is 2 – hence the name)

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

[CEU, [/popdata/1k-2014/pops/CEU.bed, /popdata/1k-2014/pops/CEU.bim, /popdata/1k-2014/pops/CEU.fam]]
[YRI, [/popdata/1k-2014/pops/YRI.bed, /popdata/1k-2014/pops/YRI.bim, /popdata/1k-2014/pops/YRI.fam]]
[BEB, [/popdata/1k-2014/pops/BEB.bed, /popdata/1k-2014/pops/BEB.bim, /popdata/1k-2014/pops/BEB.fam]]

process checkData {
   input:
     set pop, file(pl_files) from plink_data
   output:
     file "${pl_files[0]}.frq" into result
   script:
     base = pl_files[0].baseName
    "plink --bfile $base --freq  --out ${base}"
```

OR:

```
process checkData {
   input:
     set pop, file(pl_files) from plink_data
   output:
     file "${pop}.frq" into result
   script:
    "plink --bfile $pop --freq  --out $pop"
```

Version 3

```
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 }
```

Exercise Do `git checkout pairs`. In the data directory are set of data files for different years and months. First, I want 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. If you get stuck do a `git check pairs-ans`

3.5 On absolute paths

Great care needs to be taken when referring to absolute paths. Consider the following script. Assumming that local execution is being done, this should work.

Absolute paths

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

  process show {
    input:
      file data from input
    output:
      file 'see.out'
    script:
        cp $data /home/scott/answer
        ...
```

However, there is a big difference in the two uses of absolute paths. While it might be more appropriate or useful to pass the first path as a parameter, there is no real problem. Netflow will transparently stage the input files to the working directories as appropriate (generally using hard links). But the second hard-coded file will cause failures when we try to use Docker.

4 Nextflow and Docker

Light-weight virtualisation abstraction layer:

- Currently runs on Unix like systems (e.g., Linux, macOS).
- Windows support coming . . .

Can create docker image locally, or get from repositories

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

NB: This can download hundreds of MB of data so don't do on a 3G link! For the exercises below do `docker pull quay.io/banshee1221/h3agwas-plink`

Running Docker

```
docker run someimage
```

Often runs image in background (e.g. webserver)
Can run interactively

```
sudo docker run -t -i quay.io/banshee1221/h3agwas-plink
```

Nextflow supports Docker

- Well designed script should be highly portable
- Each process gets run as a separate Docker call (e.g, under the hood, a *docker run* is called)
- Can use the same or different Docker images for each process, parameterisable

Simple example
Assuming all processes use the same Docker image

```
nextflow run plink2.nf \
      -with-docker quay.io/banshee1221/h3agwas-plink
```

Now, even if you *don't* have plink, your script will work because my docker image has PLINK insalled

Directory/file access

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

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

```
data = Channel.fromPath("/popdata/1k-2014/pops/YRI.bim")
process see {
     echo true
     input:
         file bim from data
     output:
         file count
     publishDir params.publish, overwrite:true, mode:'move'
     """
         hostname
         echo "Path is \$( pwd )\n "
         echo "Parent directory has \$( ls .. )\n"
         echo "My home directory has \$( ls /home/scott )\n"
         wc -l $bim > count
         ls
     """
}
```

The output that is produced is:

```
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/94/597f09ca6cc01c7be015052f7f072c
Parent directory has 597f09ca6cc01c7be015052f7f072c
My home directory has witsGWAS

YRI.bim
count
```

Note that although the script's pwd shows */home/scott/witsGWAS/dockerized/work/94/597f09ca6cc01c7be015052f7f0*

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

Any absolutely paths (other than those used in staging) will result in error.
A little more technical detail: In the above example, the YRI.bim file is staged into the working directory on the Docker image. To achieve this, the directories /popdata, /popdata/1k-2014 and /popdata/1k-2014/pops are mounted on the Docker image. But no other sub-directories of /popdata or /popdata/1k-2014 are available. But all the other files in /popdata/1k-2014/pops are available![1]

Exercise: Have a look at *dockersee.nf* and run it thus

```
nextflow nextflow run dockersee.nf -with-docker quay.io/banshee1221/h3agwas-plink
```

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
```

Docker profiles
This can be extended in many ways

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

[1] I suppose a concession to practicality.

5 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
-

Selecting executor

Annotating each process

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

Or, `nextflow.config` file

- either global or per-process

Running Nextflow on a cluster

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 = 'WitsLong'
    scratch = true
    cpus = 5
    memory = '2GB'
}
```

Nextflow at CHPC

Nextflow can run at CHPC

- you can download into your own directory

The code below can be obtained by saying `git checkout chpc`

Example NF script

```
inp = Channel.fromPath("check.nf")

process Sorter {
   input:
      file data from inp
   output:
      file 'lines.srt' into sorted
   """
      sleep 10
      hostname > whoami
      sort $data > lines.srt
   """
}




process Counter {
   input:
      file sortf from sorted
   output:
      file 'answer' into answer
   """
      sleep 2
      hostname  > whoami
      wc -l $sortf > answer
   """
}

answer.subscribe { print it }
```

And now use the following nextflow.config file, changing the project accordingly.

nextflow.config

```
profiles {
 standard {
      process.executor = 'local'
 }

 chpc {
   process.clusterOptions = '-P CBBI0930'
   process.executor = 'pbs'
   process.$Sorter.queue = 'serial'
   process.$Counter.queue = 'normal'
   process.$Counter.cpus = '48'
   process.$Sorter.clusterOptions = '-P CBBI0930 -l select=3'
 }
}

     ./nextflow run chpc.nf -profile chpc
```

Scheduler + Docker

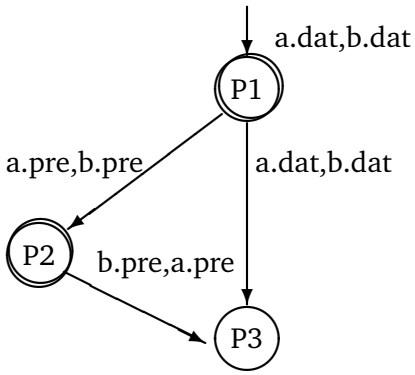
```
process.container = 'quay.io/banshee1221/h3agwas-plink:latest'
docker.enabled = false

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

6 Channel operations

Phasing
Nextflow tries to maximise concurrency

- processes are by default synchronised by channels
- when data arrives on all input channels, process executes

No problem if channels only ever have one value – but when multiple values, may be an issue.




```
Channel.fromPath("data/*.dat").set { data }

process P1 {
   input:
      file(data)
   output:
      file  "${fbase}.pre" into channelA
      file  data           into channelB
   script:
      fbase=data.baseName
      "echo dummy > ${fbase}.pre"
}



process P2 {
   input:
     file pre from channelA
   output:
     file pre into channelC
   script:
      if (pre.baseName == "a")
      "sleep 4"
      else
      "sleep 1"
}
```

Try

```
process P3 {
   input:
      file(data) from channelB
      file(pre)  from channelC
   echo true
   script:
```

```
    """
     echo "${data} - $pre"
    """
}
```

Solution: phase channels

x.phase(y): two channels, x and y get merged into one based on common key

```
ch1 = Channel.from( "a","b","c" )
ch2 = Channel.from( "a","d","e","a","c","f","b" )x
ch1 .phase(ch2) .subscribe { println it }
```

Yields

```
["a","a"]
["b","b"]
["c","c"]
```

By default: phase merged by combining

- If values are singletons, then the values must be the same
- If value is tuple if the, then the first element of the tuple must be the same

But can pass a *closure* to find the key

Phasing with a closure

```
process P3 {
   input:
      set file(pre), file(data) from \
         channelB.phase(channelC) { fn -> fn.baseName }
   echo true
   script:
    """
     echo "${data} - $pre"
    """
}
```

Copying channels

You often need to copy a channel

```
process do {
  ..
  output:
    file ("x.*") into out_ch
  ..
}

out_ch.separate(a_ch, b_ch, c_ch)
```

Alternatively

```
process do {
  ..
  output:
    file ("x.*") into (a_ch, b_ch, c_ch)
  ..
}
```

Acknowledgement

H3ABioNet
Funded by NHGRI grant number U41HG006941