Implementing scRNA-seq pipeline in Nextflow

Kane Toh*

21/04/22

Contents

Introduction to the Nextflow pipeline

What is Nextflow

Nextflow refers to two things: It is a workflow orchestration engine as well as a domain specific language (DSL) that extends the Groovy programming language. It was developed by Paolo Di Tommaso and colleagues in 2017 See Paper.

Why Nextflow

We use Nextflow to integrate our scripts (bash/python/perl/R etc.) into a cohesive pipeline, ensuring that our pipeline is portable, reproducible, scalable and checkpointed.

*kanetoh@nus.edu.sg; Cancer Science Institute, Genomics and Data Analytics Core (GeDaC)

Of course, Nextflow is not the only workflow orchestration engine around. Other popular workflow management systems include Snakemake, Cromwell (which runs WDL scripts) and Galaxy (See Popularity of workflow managers), and their relative merits have been extensively debated.

One oft-cited disadvantage of Nextflow is that it is an extension of Groovy, which is superset of the java-programmming language. For many bioinformaticians, this means having to pick up a new language. In contrast, Snakemake is written in a Python-based language and WDL is designed with a human-readable and writeable syntax. However, there are also practical computational advantages to using a Java-based language that are discussed elsewhere.

That aside, we can see 2 reasons, the first being more compelling, that make Nextflow especially conducive for developing our scRNA-seq pipeline:

1. nf-core community

The nf-core community curates many biological analysis pipelines and has an active community in a slack channel, making it easy for us to start using, experimenting and creating our own pipelines. In our case, we adapt ours from the nfcore/scrnaseq pipeline, simplifying it for our use case with `kallisto-bustools`.

2. Running on AWS Batch

Nextflow, Cromwell/WDL and Snakemake (via Tibanna on AWS) can all run on AWS Batch. We find that in contrast to the documentation provided by these workflow managers, the AWS genomics documentation is sketchy and is challenging to navigate; a feature aggravating the fact that several AWS services have to be deployed together in an AWS workflow (VPC, AWS batch, AWS ECR, AWS S3), making its deployment even more confusing. Therefore, it is especially useful for us that AWS has a dedicated page describing how to use third party workflow managers. However, this page only includes Nextflow and Cromwell, and not Snakemake.

Why implement Kallisto-bustools first (kb-python)

Currently, our scrnaseq workflow implements kb-python which wraps `kallisto and bustools`. We are fond of this pseudoaligner primarily due to its computational speed and memory efficiency. According to the authors (we are in the midst of documenting this properly ourselves), `kb-python`, which runs the `kallilsto` pseudoaligner, outperforms other aligners (cellRanger, Alevin and STARsolo) in terms of speed and memory efficiency (defined in terms of maximum memory used). Informally, we have also compared the genes x cell matrix output of `kb` on several datasets with the `cellranger` matrix output, and the results appear, at least qualitatively, similar. Thus, it appears that for common use cases, `kb` is more efficient than other aligners without sacrificing accuracy. Of course, this particular advantage makes a massive difference when running a large number of workflows in parallel on the cloud.

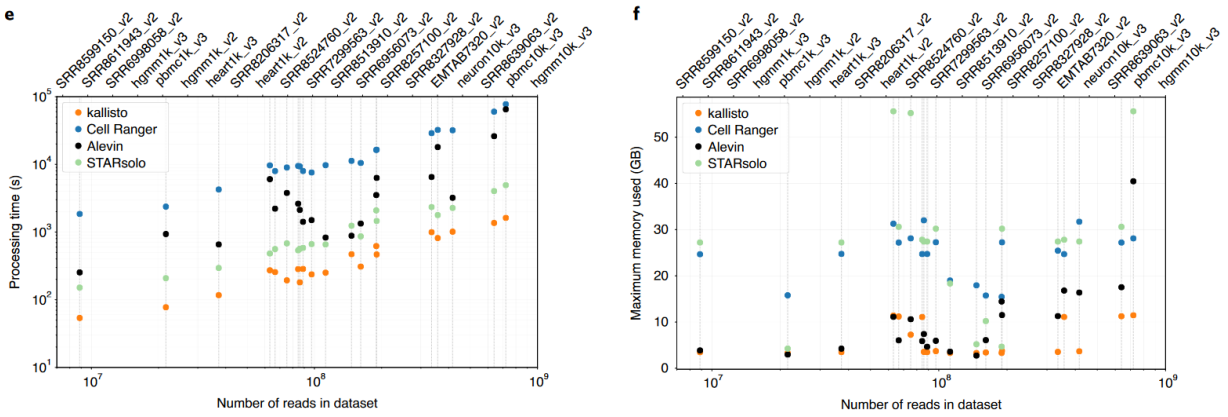

Figure 1: Benchmark-kb-against-other-aligners

Nextflow project directory

For latest updates, please refer to the github repository.

Here, we sketch the main files and directories that compose the Nextflow directory for the scRNAseq pipeline that implements kallisto-bustools. The idea is to map the interrelationships between files, thus providing us with an entry point to extending or debugging our nextflow files. We leave fine-grained dissections of the nextflow groovy-like syntax to the reader, who can find much comfort in the thorough nextflow documentations.

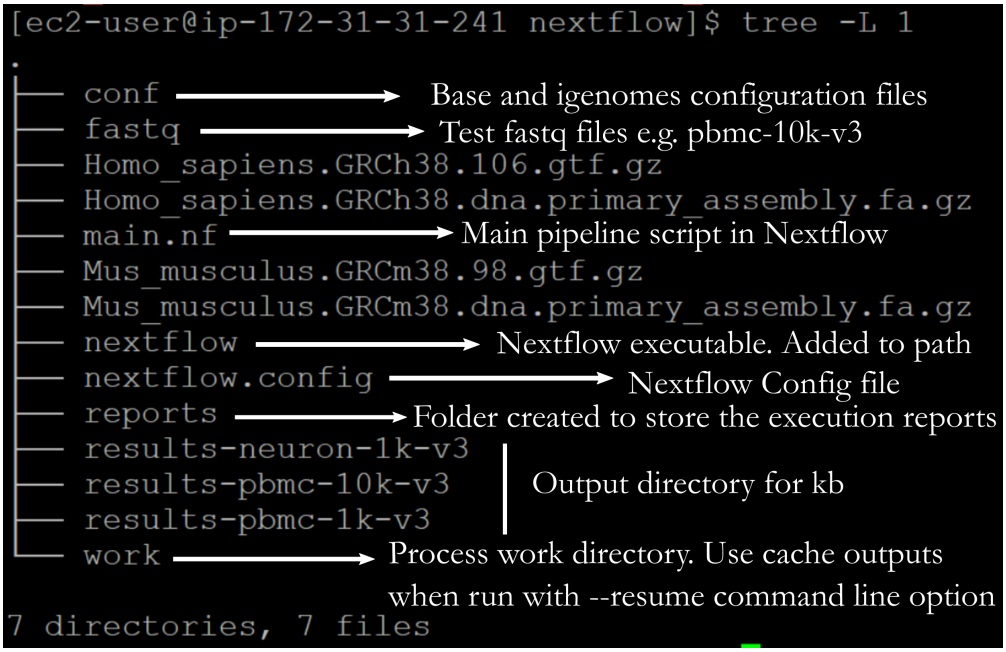

Figure 2: Nextflow-directory

Our nextflow project directory also contains the genome `fasta` and `gtf` annotation files for human and mouse. This is (currently) a required command line option - see below

Note - the pipeline is currently written in DSL 1. For Nextflow version 22-03-0-edge onwards which uses DSL 2 by default, we need to specify `export NXF_DEFAULT_DSL=1` *in the configuration file.*

main.nf

The `main.nf` file contains the meat of our scRNAseq pipeline, and thus far has only 3 sections.

Configuration checks

First, we perform a few preliminary configuration checks.

We check that:

1. If the `--genome` pipeline parameter is set when calling nextflow, then the particular genome should be available in the `iGenomes` file.
2. If the `--aligner` pipeline parameter is set when calling nextflow, then the particular aligner type should be within our list of implemented aligners (currently only `kb` is supported).

In addition, we use the configuration checks to raise errors if

1. The `index` and `t2g` files are not supplied when `kb_prebuilt` = true (default is set to false)

2. The `gtf` and `fasta` files are not supplied when `kb_prebuilt = false`, and if the species is not human or mouse. This is because `kb_download_ref` can only run to download the prebuilt human or mouse indices without needing the fasta or gtf files, whereas for all other species, we need to run `kb_build_index` as the prebuilt indices are not available.

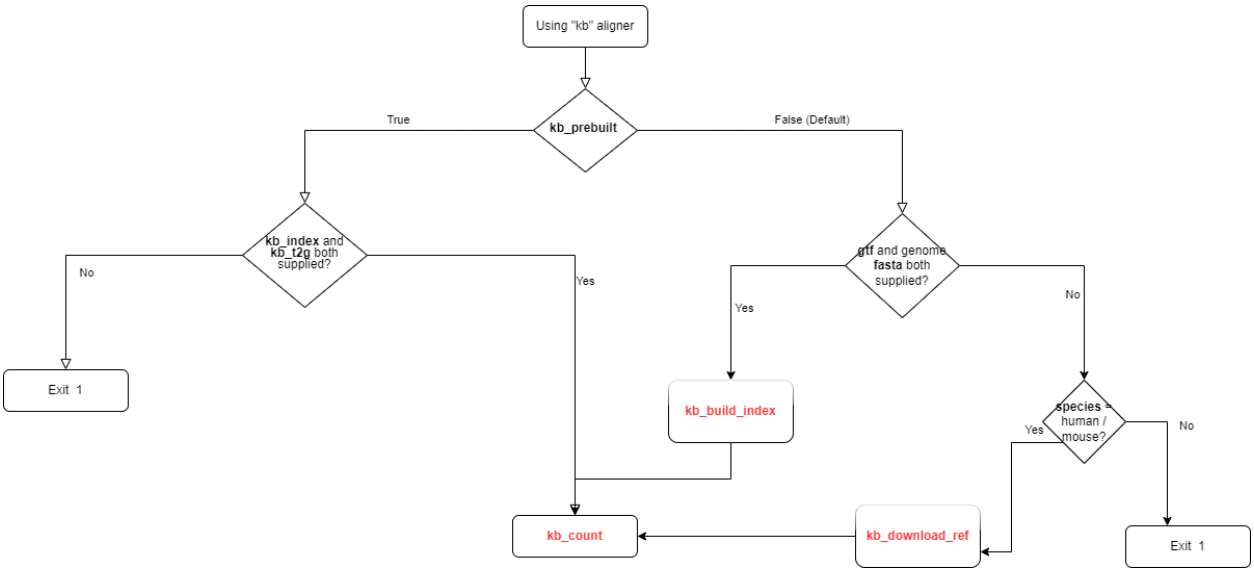

Figure 3: kb-scrnaseq-logic

Channels

Channels are the critical data structure of Nextflow, and allows Nextflow to be built on the Dataflow programming paradigm (DFP). One of its key advantages over the traditional von Neumann architecture is that the DFP can be easily parallelised (See devopedia). In Nextflow, processes cannot communicate to one another directly unless connected via channels.

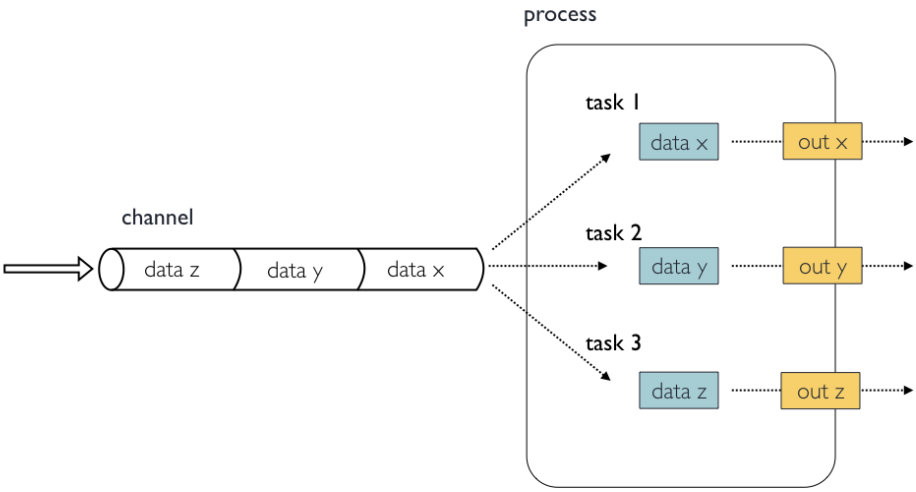

Figure 4: channel-schematic

In the above figure, we see that a channel containing 3 data inputs is passed into a process, which spawns 3

new tasks. Each task takes in a single input and outputs a single file. All 3 input items are delivered in the same order that they have been sent, but since the tasks run in parallel, there is no gurantee that they will complete in the order that they have been received by the process.

We create 5 input queue channels (`read_files_kb`, `gtf_kb`,`fasta_kb`, `kb_prebuilt_index` and `kb_prebuilt_t2g`) to pass our input files into our processes. All 5 channels are built explicitly with the `fromPath` factory method, which creates a channel that returns files corresponding to the file path. We also check whether the file exists, and then set the channel name to the above names.

Processes

We currently implement 3 processes that run the `kb-python` tool. See Figure 3 above for the logic that chains the processes together:

1. `kb_build_index`

See kb-ref for more.

This process takes as input the gtf and fasta files from the `gtf_kb` and `fasta_kb` channels and outputs the `index.idx` and `t2g.txt` files, which are then saved at the `params.kb_ref_files` directory as specified by the `publishDir` directive. These output files are then passed into the `kb_built_index` and `kb_built_t2g` channels. We run this process if the kb indices have not been prebuilt i.e. when `params.kb_prebuilt = false`.

2. `kb_download_ref`

Instead of `kb_build_index`, we run the `kb_download_ref` process if the gtf and fasta files are not supplied, and if we are working with human or mouse genomes. Our run script includes the `-d ${params.species}` flag. For both human and mouse genomes, we can directly download the index and t2g files directly from the Caltech server, which saves time (if the connection is good) by avoiding the need to build it on our own.

3. `kb_count`

See kb_count for more.

Taking the input fastq files, as well as the index and t2g files, we run the `kb count` command to pseudoalign reads with `kallisto` and quantify the data with `bustools`. For both the index and t2g files, we use the `mix` operator to combine the items emitted by these 3 channels into a single channel, which is then returned as a List object with the `collect` operator. We then parse all 3 input files into the script.

nextflow.config

See Nextflow configuration for more detail.

The nextflow.config file, as the name suggests, sets the configurations for the pipeline script. In our case, it is the primary configuration file and other configuration files are stored in the `/conf` directory.

It currently contains the following main *scopes*

- Manifest: pipeline metadata information
- Params: define parameters that will be made accessible in the pipeline script (main.nf). These are the default parameters.
  - The curly brackets makes it neater to view the respective parameters. Another way to define the parameter is via a prefix i.e. `params.aligner = "kb"`.
  - The `params.outdir` parameter is set to `"${projectDir}/results"`. There are two important components to this definition:
    1. We use the `projectDir` script implicit variable, which is already implicitly defined in the global execution scope. It expands to the directory where the main script (main.nf) is located.

2. We put the strings under double-quotation, instead of a single-quotation, following the rules of string interpolation, to insert the value of the implicit variable via the `${expression}` or `$expression` syntax.

- Process: default configuration of the process directives.
  - The `withLabel` process selector is used to configure all processes (`kb_ref` and `kb_count` in main.nf) labelled with `kb`. This makes it easy to group processes together based on properties. In our case, they are grouped based on the aligner type used.
  - We use the `container` directive to execute the kb-labelled processes in a Docker container. Specifically, we pull and run the docker image `kanetoh/kb-python` stored in the Dockerhub registry. This image was built on top of the `nf-core:base` image and currently has the `kb-python` tool installed via conda.
- Docker: We set the docker setting `enabled` to true as we are running the Nextflow pipeline using the Docker engine to run the container by default. We can put this under a `profile` scope if there are other container engines such as `Singularity`, `Charliecloud`, `Podman`etc. that we want to select. This could also have been written in the curly brackets notation.
- Profiles: Config profiles allow us to activate certain attributes using the `--profiles` command line option.

We also used other scopes to control the execution trace file (`trace`), execution graph file (`DAG`), execution timeline report (`timeline`) and execution report (`report`). For consistency, these files are all saved to the `${params.tracedir}` directory.

We also include additional configuration files from the `/conf` directory via the `includeConfig` keyword.

/work

See Nextflow resume for more detail.

The `work` directory contains the `task work directories` and functions as a scratch storage area that can (and should) be periodically cleaned up once the process is completed. The work directory is where the respective `tasks` are launched and is different from the project directory. Each task is assigned a unique ID obtained by hashing the input values and files, and this unique ID is used to create a work directory. Nextflow caches output files from each task to its respective work directory.

If the `-resume` command line option is set, Nextflow uses the UID to check if the working directory

1. exists
2. contains the expected output files
3. contains a valid command exit status

If so, it skips the task execution and uses previously computed outputs instead.

/conf

We include a couple of additional configuration files.

- The `base.config` file contains a few default parameters such as the `max_memory` allowable per process.
- The optional `igenomes.config` file from Nextflow compiles information on the reference genome builds across multiple species, including links to useful files such as the genome fasta and bowtie2/bwa indices. These resources were compiled by Phil Ewels and uploaded into an S3 bucket (ngi-igenomes).
  - We can use this list to check whether the genome name supplied by the user matches any one of those listed. If so, we can download the necessary files from the s3 bucket if required.

Run the nextflow pipeline locally

```
# The --paramName pipeline parameters overrides
# the param.paramName values specified
# in the nextflow.config file or main.nf script.
# Example run command for human pbmc-1k-v3

nextflow run main.nf \
--input "./fastq/pbmc_1k_v3/*R{1,2}*.fastq.gz" \
--gtf Homo_sapiens.GRCh38.106.gtf.gz \
--fasta Homo_sapiens.GRCh38.dna.primary_assembly.fa.gz
```

Benchmarking performance

We are currently testing the pipeline on an `AWS EC2` instance of the `t3.2xlarge` instance type.

Most commonly, the NextFlow scRNAseq pipeline would be applied by CSI researchers to the analysis of human and mouse sequences. As a preliminary test of the pipeline on the most common use cases, we downloaded the following 3 `test datasets` (2 datasets of the human peripheral blood mononuclear cells (PBMC) and 1 dataset of the E18 mouse brain) from the 10x Genomics datasets webpage:

1. Human: PBMC_1k_v3 - A relatively small human dataset of 1,222 PBMCs for fast prototyping of Nextflow on the 10x chromium platform (V3 chemistry). Tar file of approx 5.5 Gb.
2. Human: PBMC_10k_v3 - A larger dataset of 11,769 PBMCs to benchmark performance of Nextflow pipeline. Tar file of approx 51 Gb.
3. Mouse Neuron_1k_v3 - A relatively small mouse dataset of 1,301 cells from the cortex, hippocampus and sub ventricular zone of an E18 mouse to apply the pipeline to a non-human species.

The following table compares our run performance of the `kb_count` process across the 3 datasets. We examine the metrics for this process as it is significantly more expensive than `kb_ref`, which merely downloads the index / t2g files from the web. The relevant information was obtained from the `execution-trace` file stored in `${params.tracedir}`. First, we allocate only 1 cpu for the process.

name	realtime	allocated_cpus	%cpu	peak_vmem
pbmc-1k	11m 19s	1	93.8%	8.685GB
neuron-1k	19m 11s	1	95.4%	5.481GB
pbmc-10k	1h 55m	1	96.7%	8.691GB

As expected, we find that pbmc-1k and neuron-1k run faster than pbmc-10k. We see that pbmc-10k takes around 10x longer to run than pbmc-1k, consistent with the 10x number of cells between the two datasets.

To see the efficiency gain when using multiple CPUs, we run the process with 3 or 8 CPUs allocated. This was as easy as specifying the directive `cpu <3/8>` in the `kb_count` process.

name	realtime	allocated_cpus	%cpu	peak_vmem
pbmc-1k	4m 44s	3	219.8	8.658GB
pbmc-1k	3m 28s	8	295.3	8.685GB

We find that by tripling the number of CPUs allocated, we take about 1/3 of the time to run `kb_count`. Allocating 8 CPUs (max number of vCPUs for our instance type), however, does not reduce the run time significantly, as it appears that `peak_vmem` plateaus.

Attaching new EBS storage volume for the test FASTQ files

Given the large size of these files, we mount an additional EBS storage volume to a `/fastq` folder to store all the Fastq files.

```
lsblk #Display details abou block devices
mkdir ./fastq
sudo mount /dev/nvme1n1 ./fastq #Mount the NVMe block device
cd ./fastq
sudo chown `whoami` . #Give user permissions

# To mount attached volume automatically after reboot

lsblk # Copy the device's universally unique identifier (UUID)
sudo vim /etc/fstab
```

Insert a new line with the following format:

```
UUID= \<UUID>  \<mount-directory>  \<file-system>  defaults,nofail  0  2
```

- Nofail means that if the device is not enabled after you boot and mount it using fstab, no errors will be reported
- Specify `0` to prevent the file system from being dumped
- Specify `2` to indicate that this is a non-root device

Next steps

- Translate to DSL2

Resources

Nextflow-docs

- Nextflow training by Sequera labs
- Official Nextflow documentation

Nextflow-vs-other-workflow-orchestrators

- Biostars thread
- Reddit
- Epi2me
- Table 1 from Nextflow paper