DiFlow

DATA INTUITIVE

Graphical design and cover picture by Anneleen Maelfeyt (http://anneleenmaelfeyt.be/).

Typeset using LaTeX. More info to be found at http://www.data-intuitive.com.

Contents

Introduction

DiFlow[1] is an abstraction layer on top of NextFlow[2]'s DSL2[3]. DiFlow is a set of principles and guidelines for building NextFlow pipelines that allow the developer to declaratively define processing components and the user to declare the pipeline logic in a clean and intuitive way.

[1] https://pointer
[2] https://www.nextflow.io/
[3] https://www.nextflow.io/docs/latest/dsl2.html

Viash[4] is a tool that (among other things) allows us to *use* DiFlow and make it practical, without the burden of maintaining boilerplate or *glue* code.

[4] http://data-intuitive.com/viash_docs

Functional Reactive Programming (FRP)

If you're new to Functional Reactive Programming (FRP), here are a few pointers to posts and a video that introduce the concepts:

- An excellent Medium post[5] from Timo Stöttner
- The introduction[6] to Reactive Programming you've been missing from André Staltz.
- A very insightful presentation[7] by Staltz where he introduces FRP from first principles (with live coding).

[5] https://itnext.io/demystifying-functional-reactive-programming-67767dbe520b
[6] https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
[7] https://www.youtube.com/watch?v=fdol03pcvMA

In what follows, we will refer to *streams* in line with those authors but if you're used to working with Rx[8] you would call this an observable.

[8] http://reactivex.io/

FRP for pipelines

Other initiatives have consideren that FRP is a good fit for pipeline development. Recent research and development also confirms this[9,10].

[9] https://soft.vub.ac.be/~mathsaey/skitter/
[10] https://github.com/weng-lab/krews

NextFlow

FRP in NextFlow

The `Channel`[11] class used by NextFlow, itself based on the DataFlow Programming Model[12] can in fact be regarded as an implementation of a Functional Reactive Programming library. Having said that, NextFlow allows one to to mix functional and imperative programming to the point that a developer is able to shoot its own foot.

[11] https://www.nextflow.io/docs/latest/channel.html

[12] https://en.wikipedia.org/wiki/Dataflow_programming

Furthermore, `Channel`s can not be nested which complicates certain operations on the streams.

NextFlow DSL(2)

DSL2[13] is a crucial development in NextFlow because it avoid having to maintain large, monolithic pipeline definitions in one file. With DSL2, developer can spin off functionality in separate files and `import` what is needed.

[13] https://www.nextflow.io/docs/latest/dsl2.html

This also potentially opens up ways to build (reusable) modules that could be used in different projects. That is exactly what a lot of organizations need.

DiFlow

The NoPipeline approach

For developing the pipeline, we set out with a few goals in mind:

- Build modules where each modules deals with a specific (computational) task
- Make sure those modules can be reused
- Make sure the module functionality can be tested and validated
- Make sure modules have a consistent API, so that

  a. calling a module is straightforward
  b. including a module in a pipeline is transparent and seamless

Please note that nothing in these requirements has to do with running a pipeline itself. Rather, we consider this a bottom-up system whereby we first focus on a solid foundation before we actually start to tie things together.

That's why we call this the NoPipeline approach, similar to NoSQL where 'No' does not stand for *not*, but rather 'Not Only'. The idea is to focus on the pipeline aspect *after* the steps are properly defined and tested.

General Requirements and design principles

Reproducibility

I originally did not include it as a design principle for the simple reason that I think it's obvious. This should be every researcher's top priority.

Pipeline Parameters vs Runtime Parameters

We make a strict distinction between parameters that are defined for the *FULL* pipeline and those that are defined at runtime.

Pipeline Parameters We currently have 4 pipeline parameters: Docker prefix, `ddir`, `rdir` and `pdir`.

Runtime Parameters Runtime parameters differ from pipeline parameters in that they may be different for parallel runs of a process. A few examples:

- Some samples may require different filter threshold than others
- After concatenation, clustering may be run with different cluster parameters
- etc.

In other words, it does not make sense to define those parameters for the full pipeline because they are not static.

Consistent API

When we started out with the project and chose to use NextFlow as a workflow engine, I kept on thinking that the level of abstraction should have been higher. With DSL1, all you could do was create one long list of NextFlow code, tied together by `Channel`s.

With DSL2, it became feasible to *organise* stuff in separate NextFlow files and import what is required. But in larger codebases, this is not really a benefit because every modules/workflow may have its own parameters and output. No structure is imposed. `Workflow`s are basically functions taking parameters in and returning values.

I think it makes sense to define an API and to stick to it as much as possible. This makes using the modules/workflows easier…

Flat Module Structure

We want to avoid having nested modules, but rather support a pool of modules to be mixed and matched.

As a consequence, this allows a very low threshold for including third-party modules: just add it to the collection of modules and import it in the pipeline. In order to facilitate the inclusion of such third-party modules that are developed in their own respective repositories, we added one additional layer in the hierarchy allowing for such a splitting.

Job Serialization

We avoid requiring the sources of the job available in the runtime environment, i.e., the Docker container. In other words, all code and config is serialized and sent with the *process*.

An abstract computation step

The module concept inspired us to think of an abstract way to represent a computation step and implement this in NextFlow. We wrote [Portash] to this end. But Portash had its shortcomings. The most important of which was that it did not adhere to separation of concerns: execution definition (what?) where mixed up with execution context (how?/where?). Moreover, dynamic nature of Portash lends itself well to running a tool as a service, but not so much in a batch process.

Nevertheless, we were able to express a generic NextFlow step as pure *configuration* that is passed to a process at runtime. This allows for some very interesting functionality. Some prototypes were developed, the last one of which could run a single-cell RNA pipeline from mapping to generating an integrated dataset combining different samples.

The run-configuration was provided by means of a Portash YAML spec residing in the module directory. It must be stressed that not requiring the component *code* to be already available inside the container is a big plus. It means a container contains dependencies, not the actual run script so the latter can be updated more frequently. This is especially useful during component and pipeline development.

Our first implementation had a few disadvantages:

- It contained a mix of what to run and how to run it, but it did not contain information on the container to run in. This had to be configured externally, but then the module is not an independent entity anymore.
- Specifying and overriding YAML content in Groovy is possible, but not something that is intuitive. We worked around that by letting the user specify custom configuration using a Groovy nested `Map`.
- The module functionality was abstracted with a consistent API and the difference between 2 modules was just a few lines of code with a different name or pointer. But still, one had to maintain that and making a similar change in a growing set of module files is a recipe for mistakes.

But overall, the concept of an abstract computation step proved to work, it was just that a few ingredients were still missing it seemed. On the positive side, we showed that it's possible to have an abstract API for (NextFlow) modules that keeps the underlying implementation hidden while improving the readability of the pipeline code.

Toward implementation

What is needed as information in order to run a computation step in a pipeline?

1. First, we need data or generally speaking, **input**. Components/modules

and pipelines should run zero-touch, so input has to be provided at startup time.

2. Secondly, we need to know what to run en how to run it. This is in effect the definition of a modules or pipeline step.
3. Thirdly, in many cases we will require the possibility to change parameters for individual modules in the pipeline, for instance cutoff values for a filter, or the number of clusters for a clustering algorithm. The classical way to do that is via the `params` object.

One might wonder if there is a difference between input and parameters pointing to input is also a kind of parametrization. The reason those are kept apart is that additional validation steps are necessary for the data. Most pipeline systems trace input/output closely whereas parameters are ways to configure the steps in the pipeline.

In terms of FRP, and especially in the DataFlow model, we also have to keep track of the *forks* in a parallel execution scenario. For instance, if 10 batches of data can be processed in parallel we should give all 10 of them an ID so that individual forks can be distinguished. We will see that those IDs become crucial in most pipelines.

We end up with a model for a stream/channel as follows (conceptually):

```
[ ID, data, config ]
```

were

- `ID` is just a string or any object for that matter that can be compared later. We usually work with strings.
- `data` is a pointer to the (input) data. With NextFlow, this should be a `Path` object, ideally created using the `file()` helper function.
- `config` is a nested `Map` where the first level keys are chosen to be simply an identifier of the pipeline step. Other approaches can be taken here, but that's what we did.

This can be a triplet, or a list with mixed types. In Groovy, both can be used interchangeably.

The output of a pipeline step/mudules adheres to the same structure so that pipeline steps can easily be chained.

Step by step

Let us illustrate some key features of NextFlow together with how we use them in DiFlow.

POC1

Let us illustrate the stream-like nature of a NXF `Channel` using a very simple example: computing 1 + 1.

```
workflow poc1 {

    Channel.from(1) \
        | map{ it + 1 } \
        | view{ it }

}
```

This chunk is directly taken from `main.nf`, running it can be done as follows:

```
nextflow run . -entry poc1
```

POC2

NextFlow (and streams in general) are supposed to be a good fit for parallel execution. Let's see how this can be done:

```
workflow poc2 {

    Channel.from( [ 1, 2, 3 ] ) \
        | map{ it + 1 } \
        | view{ it }

}
```

Running it can be done using:

```
nextflow run . -entry poc3
```

POC3

In the previous example, we ran 3 parallel executions each time applying the same simple function: adding one. Let us simulate now a more real-life example where parallel executions will not take the same amount of time. We do this by defining a `process` and `workflow` that uses this process. The rest is similar to our example before.

```
process add {

    input:
        val(input)
    output:
        val(output)
    exec:
        output = input + 1

}

workflow poc3 {

    Channel.from( [ 1, 2, 3 ] ) \
        | add \
        | view{ it }

}
```

Running it is again the same.

```
nextflow run . -entry poc3
```

The result will be a permutation of 2,3 and 4. Try it multiple times to verify for yourself that the order is not guaranteed to be the same. Even though the execution times will not be that much different! In other words, a `Channel` does not guarantee the order, and that's a good thing.

POC4

An illustrative test is one where we do not use a `process` for the execution, but rather just `map` but such that one of the inputs *takes longer* to process, i.e.:

```
def waitAndReturn(it) { sleep(2000); return it }

workflow poc4 {

    Channel.from( [ 1, 2, 3 ] ) \
        | map{ (it == 2) ? waitAndReturn(it) : it } \
        | map{ it + 1 } \
        | view{ it }

}
```

Running it:

```
nextflow run . -entry poc4
```

The result may be somewhat unexpected, the order is retained there's just a 2 second delay between the first entry and the rest. The `sleep` in other words blocks all the parallel execution branches.

This is a clear indication of why it's better to use a `process` to execute computations. On the other hand, as long as we *stay* inside the `map` and don't run a `process`, the order is the same. This opens up some possibilities that we will exploit in what follows.

POC5

If we can not guarantee the order of the different parallel branches, we should introduce a *branch ID*. This may be a label, a sample ID, a batch ID, etc. It's the unit of parallelization.

```
process addTuple {

    input:
        tuple val(id), val(input)
    output:
        tuple val("${id}"), val(output)
    exec:
        output = input + 1

}

workflow poc5 {

    Channel.from( [ 1, 2, 3 ] ) \
        | map{ el -> [ el.toString(), el ]} \
        | addTuple \
        | view{ it }

}
```

We can ran this code sample in the same way as the previous examples.

Please note that the function to add 1 remains exactly the same, we only added the `id` as the first element of the tuple in both input and output. As such we keep a handle on which sample is which, by means of the *key* in the tuple.

Note: Later, we will extend this tuple and add configuration parameters to it… but this was supposed to go step by step.

POC6

What if we want to be able to configure the term in the sum? This would require a parameter to be sent with the process invocation. Let's see how this can be done.

```
process addTupleWithParameter {

input:
    tuple val(id), val(input), val(term)
output:
    tuple val("${id}"), val(output)
exec:
    output = input + term

}

workflow poc6 {

    Channel.from( [ 1, 2, 3 ] ) \
        | map{ el -> [ el.toString(), el, 10 ]} \
        | addTupleWithParameter \
        | view{ it }

}
```

This works, but is not very flexible. What if we want to configure the operator as well? What if we want to have branch-specific configuration? We can add a whole list of parameters, but that means that the `process` signature may be different for every `process` that we define. That is not a preferred solution.

POC7

Let us use a simple hash to add 2 configuration parameters.

```
process addTupleWithHash {

    input:
        tuple val(id), val(input), val(config)
    output:
        tuple val("${id}"), val(output)
    exec:
        output = (config.operator == "+") ? input + config.term : input - config.term
}

workflow poc7 {

    Channel.from( [ 1, 2, 3 ] ) \
        | map{ el -> [ el.toString(), el, [ "operator" : "-", "term" : 10 ]  ]} \
        | addTupleWithHash \
        | view{ it }

}
```

POC8

POC7 offers us a way to use a consistent API for a process. Ideally, however, we would like different `process` invocation to be chained rather than to explicitly add the correct configuration all the time. Let us add an additional key to the map, so that a process knows *it's scope*.

```
process addTupleWithProcessHash {

    input:
        tuple val(id), val(input), val(config)
    output:
        tuple val("${id}"), val(output)
    exec:
        def thisConf = config.addTupleWithProcessHash
        output = (thisConf.operator == "+") ? input + thisConf.term : input - thisConf.term

}

workflow poc8 {

    Channel.from( [ 1, 2, 3 ] ) \
        | map{ el -> [ el.toString(), el, [ "addTupleWithProcessHash" : [ "operator" : "-", "term" : 10 ] ] ] } \
        | addTupleWithProcessHash \
        | view{ it }

}
```

Please note that we used the process name as a key in the map, so that each process can tell what configuration parameter are relevant for its own scope.

POC9

We used native Groovy code in the `process` examples above. Let us now use a shell script:

```
workflow poc8 {

    Channel.from( [ 1, 2, 3 ] ) \
        | map{ el -> [ el.toString(), el, [ "addTupleWithProcessHash" : [ "operator" : "-", "term" : 10 ] ] ] } \
        | addTupleWithProcessHash \
        | view{ it }

}

process addTupleWithProcessHashScript {

    input:
        tuple val(id), val(input), val(config)
    output:
        tuple val("${id}"), stdout
    script:
        def thisConf = config.addTupleWithProcessHashScript
        def operator = thisConf.operator
        def term = thisConf.term
        """
        echo \$( expr $input $operator ${thisConf.term} )
        """

}
```

Running this (in the same way as before), we get something along these lines:

```
[3, -7
]
[1, -9
]
[2, -8
]
```

This is because the `stdout` qualifier captures the newline at the end of the code block. We could look for ways to circumvent that, but that is not the point here.

What's important to notice here:

1. We can not just retrieve individual entries in `config` in the shell, we have to let Groovy do that.
2. That means we either first retrieve individual values and store them in a variable,
3. or we use the `${...}` notation for that.
4. If we want to use `bash` variables, the `$` symbol has to be escaped.

Obviously, passing config like this requires a lot of typing (especially as additional parameters are introduced) and is error prone.

POC10

We used the pipe `|` symbol to combine different steps in a *pipeline* and we noticed that a `process` can do computations on parallel branches. That's nice, but we have not yet given an example of running 2 processes, one after the other.

There are a few things we have to note before we go to an example:

1. It's not possible to call the same process twice, a strange error occurs in that case[14].
2. If we want to pipe the output of one process as input of the next, the I/O signature needs to be exactly the same, so the `output` of the `process` should be a triplet as well.

[14] It *is* possible in some cases however to manipulate the `Channel` such that a process is effectively run twice on the same data, but that is a more advanced topic.

```
process process_poc10a {

    input:
        tuple val(id), val(input), val(term)
    output:
        tuple val("${id}"), val(output), val("${term}")
    exec:
        output = input.toInteger() + term.toInteger()

}

process process_poc10b {

    input:
        tuple val(id), val(input), val(term)
    output:
        tuple val("${id}"), val(output), val("${term}")
    exec:
        output = input.toInteger() - term.toInteger()

}

workflow poc10 {

    Channel.from( [ 1, 2, 3 ] ) \
        | map{ el -> [ el.toString(), el, 10 ] } \
        | process_poc10a \
        | process_poc10b \
        | view{ it }

}
```

The result of this is that first 10 is added and then the same 10 is subtracted again, which results in the same as the original. Please note that the output contains 3 elements, also the `term` passed to the `process`:

```
[3, 3, 10]
[1, 1, 10]
[2, 2, 10]
```

We can configure the second `process` (subtraction) by adding an additional `map` in the mix:

```
workflow poc10 {

    Channel.from( [ 1, 2, 3 ] ) \
        | map{ el -> [ el.toString(), el, 10 ] } \
        | process_poc10a \
        | map{ [ it[0], it[1], 5 ] } \
        | process_poc10b \
        | view{ it }

}
```

Please note that we define the closure in a different manner here, using the special variable `it`. We could also write (to the same effect):

```
        ...
        | map{ x -> [ x[0], x[1], 5 ] } \
        ...
```

or even

```
        ...
        | map{ id, value, term -> [ id, value, 5 ] } \
        ...
```

Let us tackle a different angle now and start to deal with files as input and output. We will leave

---

Appendix

Caveats and Tips

Resources

When you run or export with the `DockerTarget`, resources are automatically added to the running container and stored under `/resources`. In case of the `NativeTarget`, this is not the case and since `NextFlowTarget` uses the `NativeTarget` it's the same there. That does not mean that resources specified in `functionality.yaml` is not available in these cases, we only have to point to them where appropriate.

The following snippet (from `ct/singler`) illustrates this:

```r
par = list(
  input = "input.h5ad",
  output = "output.h5ad",
  reference = "HPCA",
  outputField = "cellType",
  pruningMADS = 3,
  outputFieldPruned = "celltype-pruned",
  reportOutputPath = "report.md"
)
## VIASH END
par$resources_dir <- resources_dir
```

In other words, `resources_dir` is automatically created by `viash` in all current 3 environments. This means that we can point to the `report.Rmd` file present in the resources like so:

```r
rmarkdown::render(paste0(par$resources_dir, "/", "report.Rmd"), output_file = par$reportOutputPath)
```

Default values

In functionality, no option should have an empty string as value!

`target_image`

It makes sense to add the `target_image` attribute in the `docker_platform.yaml` file. This way, the resulting container image is predictable, rather than an autogenerated tag from `viash`.

Running the Docker setup

We don't have a solution yet for pre-generating the Docker images prior to starting a NXF pipeline. For the moment, we ask the user to run the build script for the Docker targets with the `---setup` option. This only works locally, it would for instance not work on a different (clean) node or in a Kubernetes cluster.

We are working on solutions or workarounds for this. Keep you posted!

Open issues

1. Multiple files as input for a component: E.g. the concat component uses multiple files to be joined. At the moment this does not seems to be possible.
2. Use of additional input files into a specific component. Some components do not only have input/output but require additional input. How should we map this?