Nextflow Hands-on

Jacopo Gasparetto

Workshop on
management of distributed resources for genomic communities

29-30 October 2024

INFN CLOUD

ICSC Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

DARE DIGITAL LIFELONG PREVENTION

POLICLINICO DI SANT'ORSOLA

Ministero dell'Università e della Ricerca

Italiadomani

PNC Piano nazionale per gli investimenti complementari al PNRR
Ministero dell'Università e della Ricerca

Data preparation

```
mkdir workshop && cd workshop
curl -O https://raw.githubusercontent.com/jacogasp/workshop-
nextflow/refs/heads/main/generate_data.sh
chmod +x generate_data.sh
./generate_data.sh data
```

Il contenuto della cartella dovrebbe essere

```
ls data
A.txt   B.txt   C.txt   D.txt   E.txt
```

Hello, World

Creiamo una cartella che conterrà l'intero progetto Nextflow e chiamiamola `pi-workflow`

```
mkdir pi-workflow
cd pi-workflow
```

Creiamo due file, uno chiamato `nextflow.config` e uno `main.nf`.

In `nextflow.config` inseriamo

```
// Params
params.data
params.output
// Runners
docker.enabled = true
```

Hello, World

In `main.nf`

```
log.info """\
===================================
W O R K S H O P  -  N E X T F L O W
===================================

data directory   : ${params.data}
output directory : ${params.output}
"""
```

Hello, World

Posizioniamoci *sopra* la cartella `pi-workflow` e lanciamo Nextflow con il comando

```
nextflow run pi-workshop --data $(pwd)/data --output $(pwd)/results
```

I parametri definiti in `nextflow.config` tramite la keyword `params` vengono passati a Nextflow tramite il *doppio trattino*. Ad esempio, per utilizzare il parametro `params.foo`, scriveremo

```
nextflow run my-workflow --foo some_value
```

Hello, World

L’ouput del run del comando

```
nextflow run pi-workshop --data $(pwd)/data --output $(pwd)/results
```

dovrebbe essere simile a

```
 N E X T F L O W   ~  version 24.04.4

Launching `pi-workshop/main.nf` [friendly_lavoisier] DSL2 - revision: ce254f739d

================================
W O R K S H O P  -  N E X T F L O W
================================

data directory   : /Users/jacopo/pi-ws/data
output directory : /Users/jacopo/pi-ws/results
```

Main Workflow

In `main.nf` creiamo il workflow principale tramite la keyword `workflow`. Continuando sotto `log.info` aggiungiamo

```
// log.info ...

workflow {
  // everything happens here
}
```

Main Workflow: `Channel.fromPath`

Per accedere ai dati input, utilizziamo la funzione `Channel.fromPath`, la quale genererà una coda di file da sottomettere agli step successivo/i. Utilizziamo la funzione `view` per mostrare il risultato

```
workflow {
  Channel.fromPath(params.data + "/*.txt") | view
}
```

Lanciamo il workflow con il precedente comando

```
nextflow run pi-workshop --data $(pwd)/data --output $(pwd)/results
```

Main Workflow: Channel.fromPath

```
workflow {
  Channel.fromPath(params.data + "/*.txt") | view
}
```

Nextflow mostrerà tutti i file con estensione `.txt` all'interno della cartella `data`

```
==================================
W O R K S H O P  -  N E X T F L O W
==================================

data directory   : /Users/jacopo/pi-ws/data
output directory : /Users/jacopo/pi-ws/results

/Users/jacopo/pi-ws/data/C.txt
/Users/jacopo/pi-ws/data/B.txt
/Users/jacopo/pi-ws/data/A.txt
/Users/jacopo/pi-ws/data/E.txt
/Users/jacopo/pi-ws/data/D.txt
```

Main Workflow: `Channel.fromPath`

Possiamo salvare l'output della funzione `Channel.fromPath` in una variabile usando l'espressione

```
def nome_variabile = valore
```

La seguente scrittura sarà equivalente a quello che abbiamo appena scritto

```
workflow {
  def inputs = Channel.fromPath(params.data + "/*.txt")
  inputs | view
}
```

Il mio primo processo

Un *processo* (`process`) è l'unità funzionale di lavoro che accetta un input e produce un output a in funzione del calcolo che si desidera effettuare. Nel mondo cloud, ogni processo sarà eseguito all'interno di un container Docker, la cui immagine di partenza conterrà il software necessario a svolgere il calcolo.

È buona norma organizzare in processi in file chiamati moduli suddivisi per «argomento logico». Creiamo dunque un cartella chiamata `modules` e all'interno un file chiamato `pi.nf`

```
├── main.nf
├── modules
│   └── pi.nf
└── nextflow.config
```

Il mio primo processo

Nel modulo `modules/pi.nf` definiamo il processo `ComputePi`. Il processo prenderà come input un singolo file tra i file di dati (A.txt, B.txt, C.txt, ecc) e come output produrrà un singolo file chiamato `pi.txt`. Definiamo, inoltre, l'immagine che vogliamo utilizzare per eseguire il processo, in questo caso utilizziamo l'immagine Docker ufficiale di Python 3

```
process ComputePi {
  input:
    path fileName
  output:
    path pi.txt
  container "python:3"
  shell:
    """
    // ... some code
    """
}
```

Il mio primo processo

Nella sezione `shell` è possibile eseguire qualunque comando bash oppure script in altri linguaggi, ad esempio Python. Nella sezione shell aggiungiamo il seguente script per calcolare Pi greco tramite il metodo Monte-Carlo:

```
process ComputePi {
  // input
  // output...
  shell:
  """
    #!/usr/local/bin/python
    import random
    with open("${fileName}") as f:
        N = int(f.read())
    inside_circle = 0
    for i in range(N):
        x = random.random()
        y = random.random()
        if x * x + y * y < 1:
            inside_circle += 1
    pi = 4.0 * inside_circle / N
    with open("pi.txt", "w") as f:
        f.write(str(pi))
  """
}
```

Il mio primo processo

Lo script legge il contenuto del file passato in input (in questo caso un numero intero) e scrive su file il valore calcolato di Pi Greco in funzione del numero N di iterazioni contenuto nel file di input.

```
process ComputePi {
  // input
  // output...
  shell:
  """
    #!/usr/local/bin/python
    import random
    with open("${fileName}") as f:
        N = int(f.read())
    inside_circle = 0
    for i in range(N):
        x = random.random()
        y = random.random()
        if x * x + y * y < 1:
            inside_circle += 1
    pi = 4.0 * inside_circle / N
    with open("pi.txt", "w") as f:
        f.write(str(pi))
  """
}
```

Il mio primo processo

Nel file `main.nf`, includiamo il processo `ComputePi` dal modulo `pi`. Alla prima riga aggiungiamo

```
include { ComputePi } from './modules/pi'
```

Dentro il workflow chiamiamo il processo `ComputePi` passandogli come argomento il channel `inputs`, e stampiamo il risultato a schermo

```
workflow {
  def inputs = Channel.fromPath(params.data + "/*.txt")
  ComputePi(inputs) | view
}
```

Il mio primo processo

Lanciamo il workflow con il solito comando

```
nextflow run pi-workshop --data $(pwd)/data --output $(pwd)/results
```

L'output sarà del tipo

```
executor >  local (5)
[f7/8276c4] ComputePi (5) | 5 of 5 ✓
/Users/jacopo/pi-ws/work/d5/c481a14ded426dd035af5aa90bc28b/pi.txt
/Users/jacopo/pi-ws/work/0a/25151cf2bc94be2bb0a868cf508c15/pi.txt
/Users/jacopo/pi-ws/work/ad/c99b2bce572ef790f6abd5acbb179d/pi.txt
/Users/jacopo/pi-ws/work/65/be8efc63864d57d2a342256c2ad16a/pi.txt
/Users/jacopo/pi-ws/work/f7/8276c42124877f8639fed231987a5d/pi.txt
```

Il mio primo processo

```
executor >  local (5)
[f7/8276c4] ComputePi (5) | 5 of 5 ✓
/Users/jacopo/pi-ws/work/d5/c481a14ded426dd035af5aa90bc28b/pi.txt
/Users/jacopo/pi-ws/work/0a/25151cf2bc94be2bb0a868cf508c15/pi.txt
/Users/jacopo/pi-ws/work/ad/c99b2bce572ef790f6abd5acbb179d/pi.txt
/Users/jacopo/pi-ws/work/65/be8efc63864d57d2a342256c2ad16a/pi.txt
/Users/jacopo/pi-ws/work/f7/8276c42124877f8639fed231987a5d/pi.txt
```

Nextflow ha lanciato 5 processi identici, uno per ogni file di input (A.txt, B.txt, C.txt, D.txt, E.txt e F.txt) e ha prodotto un file `pi.txt` ciascuno. Se guardiamo il contenuto di uno di questi file troviamo il nostro risultato

```
cat /Users/jacopo/pi-ws/work/d5/c481a14ded426dd035af5aa90bc28b/pi.txt
3.1442247658688864
```

Aggregazione degli output

A partire da 5 file di input abbiamo creato 5 risultati di output (relazione one-to-one) utilizzando il parallelismo di 5 processi. Ora vogliamo prendere questi 5 risultati e passarli come singolo input ad un processo che ne fa la media e ritorna il risultato.

Nota: ci sono innumerevoli modi per fare questa cosa, ad esempio gli operatori `map`, `flatMap`, `reduce` ecc. Per una conoscenza approfondita degli operatori fare riferimento alla guida ufficiale https://www.nextflow.io/docs/latest/reference/operator.html

```
executor >  local (5)
[f7/8276c4] ComputePi (5) | 5 of 5 ✓
/Users/jacopo/pi-ws/work/d5/c481a14ded426dd035af5aa90bc28b/pi.txt
/Users/jacopo/pi-ws/work/0a/25151cf2bc94be2bb0a868cf508c15/pi.txt
/Users/jacopo/pi-ws/work/ad/c99b2bce572ef790f6abd5acbb179d/pi.txt
/Users/jacopo/pi-ws/work/65/be8efc63864d57d2a342256c2ad16a/pi.txt
/Users/jacopo/pi-ws/work/f7/8276c42124877f8639fed231987a5d/pi.txt
```

Aggregazione degli output

Aggiungiamo in coda `ComputePi(inputs)` l'operatore `collectFile`. Come dice il nome, l'operatore colleziona tutti i file generati dalle 5 esecuzioni di `ComputePi()` e scrive i risultati in un unico file di testo, chiamato anch'esso `pi.txt`. Il parametro `newLine` indica che il risultato di ogni processo sarà scritto in una nuova riga.

```
workflow {
  def inputs = Channel.fromPath(params.data + "/*.txt")
  def results = ComputePi(inputs).collectFile(name: "pi.txt", newLine: true)
  results | view
}
```

Aggregazione degli output

Eseguendo il workflow troviamo

```
executor >  local (5)
[1e/fa7866] ComputePi (5) | 5 of 5 ✓
/Users/jacopo/pi-ws/work/tmp/f2/5ba06b66fdfee790ff92ebfaafec16/pi.txt
```

e ispezionando il contenuto del file

```
cat /Users/jacopo/pi-ws/work/tmp/f2/5ba06b66fdfee790ff92ebfaafec16/pi.txt
3.1624713958810067
3.1489698890649764
3.088865764828304
3.124588002636783
3.1569581537606006
```

Aggregazione degli output

All'interno del modulo `pi.nf`, creiamo un nuovo processo chiamato `Avg`. Il processo prende in ingresso il singolo file `pi.txt` contenente i risultati, come output restituisce la media degli N valori. In questo caso non andremo a scrivere il risultato su un file ma direttamente in standard output.

```
process Avg {
  input:
    path fileName
  output:
    stdout
  container "python:3"
  shell:
    """
    #!/usr/local/bin/python
    with open("${fileName}") as f:
        arr = [float(x) for x in f.readlines()]
    pi = sum(arr) / len(arr)
    print(pi)
    """
}
```

Aggregazione degli output

Includiamo il processo Avg all'interno del file `main.nf`

```
include { ComputePi; Avg } from './modules/pi'
```

e invochiamolo all'interno del workflow

```
workflow {
  def inputs = Channel.fromPath(params.data + "/*.txt")
  def results = ComputePi(inputs).collectFile(name: "pi.txt", newLine: true)
  Avg(results) | view
}
```

Aggregazione degli output

Eseguiamo il workflow e come output troveremo

```
executor >  local (6)
[11/b52fe9] ComputePi (5) | 5 of 5 ✓
[61/0c2112] Avg (1)       | 1 of 1 ✓
3.141324446513296
```

Pubblicazione dell'output

Nonostante si potrebbe salvare il risultato finale direttamente nella processo `Avg()`, per didattica ed esercizio di questo workshop creiamo un terzo processo la cui unica funzione è prendere il risultato in output di Avg e *pubblicarlo* nella cartella di destinazione.

Creiamo dunque un nuovo processo all'interno del modulo `pi.nf` chiamato `SaveResult`

```
process SaveResult {
  input:
    val pi
  publishDir params.output, mode: 'copy', overwrite: true
  container "python:3"
  output:
    path "pi.txt"
  shell:
    """
    echo "${pi}" >> pi.txt
    """
}
```

Pubblicazione dell'output

```
process SaveResult {
  input:
    val pi
  publishDir params.output, mode: 'copy', overwrite: true
  container "python:3"
  output:
    path "pi.txt"
  shell:
    """
    echo "${pi}" >> pi.txt
    """
}
```

Notiamo che questa volta il nostro input non è un file, bensì un *valore*. Utilizziamo invece la direttiva `publishDir` per pubblicare il nostro risultato nella cartella di destinazione identificata dal parametro `params.output`.

Aggregazione degli output

Includiamo il processo `SaveResult` all'interno del file `main.nf`

```
include { ComputePi; Avg; SaveResult } from './modules/pi'
```

e invochiamolo all'interno del workflow

```
workflow {
  def inputs = Channel.fromPath(params.data + "/*.txt")
  def results = ComputePi(inputs).collectFile(name: "pi.txt", newLine: true)
  def pi = Avg(results)
  SaveResult(pi)
}
```

Aggregazione degli output

Dopo aver lanciato il workflow troviamo

```
data directory   : /Users/jacopo/pi-ws/data
output directory : /Users/jacopo/pi-ws/results

executor >  local (7)
[5b/2f4937] ComputePi (5)  | 5 of 5 ✓
[b0/8700ba] Avg (1)        | 1 of 1 ✓
[cf/d9f2e1] SaveResult (1) | 1 of 1 ✓
```

e in `/Users/jacopo/pi-ws/results` troviamo il file finale

```
cat /Users/jacopo/pi-ws/results/pi.txt

3.1245344250724285
```

Bonus

La sintassi di Nextflow/Groovy permette scritture alternative che possono generare confusione. Invece di *definire* (`def`) l'output delle funzioni/processi tramite variabili, si possono concatenare i processi tramite l'operatore "|"

Ad esempio, il workflow

```
workflow {
  def inputs = Channel.fromPath(params.data + "/*.txt")
  def results = ComputePi(inputs).collectFile(name: "pi.txt", newLine: true)
  def pi = Avg(results)
  SaveResult(pi)
}
```

si può scrivere nella forma

```
workflow {
  Channel.fromPath(params.data + "/*.txt") |
  ComputePi | collectFile(name: "pi.txt", newLine: true) |
  Avg | SaveResult
}
```

Nextflow on K8S

nextflow.conf

Editiamo il file `nextflow.conf` per preparalo per lavorare con Kubernetes. Creiamo due profili, uno standard per eseguire il workflow in locale con Docker, e un profilo chiamato `k8s` per in chiediamo che i processi vengano eseguiti sul cluster

```
// Params
params.data
params.output
// Runners
docker.enabled = true
// Profiles
profiles  {
  standard {
    process.executor = 'local'
  }
  k8s {  // custom name!
    process.executor = 'k8s'
  }
}
```

nextflow.conf

Sempre nel file `nextflow.conf`, aggiungiamo i mount point dei volumi che verranno montanti all'interno dei pod

```
// Kuberetes
k8s.pod = [
  [volumeClaim: "pvc-input", mountPath: "/input/data"],
  [volumeClaim: "pvc-work", mountPath: "/work/nextflow"],
  [volumeClaim: "pvc-output", mountPath: "/output/results"],
]
```

Per concludere, aggiungiamo le informazioni sul namespace, context e user account da utilizzare, e la flag `debug.yaml` per persistere la configurazione utilizzata da Nextflow per generare i pod

```
k8s {
  namespace = "nextflow"
  context = "sorsola"
  serviceAccount = "nextflow-sa"
  debug.yaml = true
}
```

Driver pod - Generale

Per poter lanciare sul cluster, è necessario generare un manifest Kuberentes contente il comando Nextflow da eseguire e i volumi, service account e segreti da associare. Creiamo dunque un file chiamato `k8s-run.yaml`. La risorsa che vogliamo creare sarà di tipo `Pod` e come metadata gli associamo un nome a piacere e gli assegniamo il namespace `nextflow`

```
apiVersion: v1
kind: Pod
metadata:
  name: wf-workshop-change-me
  namespace: nextflow
```

Driver pod - Spec

Definiamo ora le specifiche nel campo `spec`. Assegniamo il service account `nextflow-sa` e ettiamo `Never` come restart policy.

```
// metadata:
// ...
spec:
  serviceAccountName: nextflow-sa
  restartPolicy: Never
  volumes: ...
  containers: ...
```

Driver pod - Volumes

Nella sezione `volumes` definiamo tre volumi di tipo `persistantVolumeClaim`, un volume di input contente i dati, un volume di work e un volume di output per salvare i risultati

```
spec:
  volumes:
    - name: input-vol           # nome di fantasia
      persistentVolumeClaim:
        claimName: pvc-input    # nome del Persistent Volume Claim definito nel cluster
    - name: output-vol
      persistentVolumeClaim:
        claimName: pvc-output
    - name: work-vol
      persistentVolumeClaim:
        claimName: pvc-work
```

Driver pod – Containers 1.

Nella sezione `containers` definiamo un solo container basato sull'immagine custom di Nextflow.

Settiamo la working directory del container nella stessa cartella di work (`/work/nextflow`) così di eseguire il processo `nextflow` all'interno di questa cartella; in questo modo Nextflow sarà in grado di salvare la cache all'interno di `/work/nextflow/.nextflow`

```
spec:
 containers:
    - name: nextflow
      image: gitlab-sorsola-integration.cnaf.infn.it:4567/sorsola/docker-images/nextflow
      imagePullPolicy: Always
      workingDir: "/work/nextflow/my-workflow"
      args: [ ... ]
      volumeMounts: ...
```

La direttiva `imagePullPolicy: Always` ci garantisce di utilizzare sempre l'ultima immagine disponibile, se non già scaricata sul nodo del cluster.

Driver pod – Containers 2.

Scriviamo nella sezione args tutto il comando che vogliamo eseguire sotto forma di lista di argomenti. Ricordiamo che un "-" rappresenta un parametro nativo di Nextflow, mentre due "--" rappresentano un parametro custom che abbiamo definito in `nextflow.config`.

```
spec:
 containers:
    - name: nextflow
      # ...
      args: [
        "nextflow", "run",
        "jacogasp/workshop-nextflow", "-r", "main", # repo, branch (revision) "main"
        "-profile", "k8s",                          # profile defined in nextflow.conf
        "-w", "/work/nextflow/my-workflow ",        # nextflow's workdir
        "--data", "/input/data/workshop",
        "--output", "/output/results/workshop",
        "-resume"                                   # resume workflow if any cache is found
]
```

Driver pod – Containers 3.

Come ultima cosa, è necessario montare i persistent volume claim che abbiamo definito in `spec.volumes` all'interno del container. I `mountPath` sono definiti dall'architettura di rete NFS del cluster e possono essere variati solo dall'amministratore.

```
spec:
 containers:
    - name: nextflow
      # ...
      volumeMounts:
        - name: input-vol
          mountPath: "/input/data"
        - name: work-vol
          mountPath: "/work/nextflow"
        - name: output-vol
          mountPath: "/output/results"
```

```
apiVersion: v1
kind: Pod
metadata:
  name: wf-workshop-change-me
  namespace: nextflow
spec:
  serviceAccountName: nextflow-sa
  restartPolicy: Never
  volumes:
    - name: input-vol
      persistentVolumeClaim:
        claimName: pvc-input
    - name: output-vol
      persistentVolumeClaim:
        claimName: pvc-output
    - name: work-vol
      persistentVolumeClaim:
        claimName: pvc-work
  containers:
    - name: nextflow
      image: gitlab-sorsola-integration.cnaf.infn.it:4567/sorsola/docker-images/nextflow
      imagePullPolicy: Always
      workingDir: "/work/nextflow/change-me"
      args: [
        "nextflow", "run",
        "jacogasp/workshop-nextflow", "-r", "main",
        "-profile", "k8s",
        "-w", "/work/nextflow/my-workflow ",
        "--data", "/input/data/workshop",
        "--output", "/output/results/workshop",
        "-resume"
      ]
      volumeMounts:
        - name: input-vol
          mountPath: "/input/data"
        - name: work-vol
          mountPath: "/work/nextflow"
        - name: output-vol
          mountPath: "/output/results"
```

Esecuzione su Kubernetes

Per lanciare il workflow sul cluster

```
token=$(oidc-token sorsola)
alias k="kubectl token=$token -n nextflow"
k apply -f k8s-run.yaml
```

Per monitorare lo stato del work flow

```
k get po
k logs -f wf-workshop-change-me
k exec <pod-name> -it -- bash
// oppure
k9s --token=$token
```

https://monitoring-sorsola-integration.cnaf.infn.it