Nextflow

A book for nextflow workflow tutorials

Nextflow Introduction

Nextflow is a workflow management system designed to streamline the execution of complex data pipelines.
It supports scalable and reproducible workflows, making it ideal for bioinformatics and other data-intensive fields.

Benefits Over Traditional Bash Scripting

- **Scalability**: Automatically manages parallel execution and scales from local to cloud-based environments.
- **Reproducibility**: Uses a declarative language to ensure consistent results and easy sharing.
- **Resource Management**: Integrates with various environments and schedulers for efficient resource allocation.
- **Modularity**: Allows for reusable, modular processes, simplifying pipeline maintenance.
- **Error Handling**: Includes built-in error handling and retry mechanisms.
- **Data Provenance**: Tracks data flow for better debugging and result tracking.
- **Caching and Resume**: Features caching of intermediate results and the ability to resume failed or interrupted workflows, saving time and resources.

Nextflow enhances efficiency, scalability, and maintainability compared to traditional bash scripting.

How do I use Nextflow?

Below is a video going through some of the Nextflow basics on our HPC

https://www.youtube.com/watch?v=5UkWwRcIytw

There are a number of different docs and training materials that can be found regarding nextflow usage.

https://www.nextflow.io/docs/latest/index.html

Official Nextflow training video

https://www.youtube.com/watch?v=wbtMbJTo1xo

Example RNA-seq Nextflow script from video

```
#!/usr/bin/env nextflow
// Defining necessary variables including the path to the reference genome and fastq files
params.ref = "/hpc/faculty/azeezoe/rna_seq/refs/22.fa"
params.reads = "/hpc/faculty/azeezoe/rna_seq/reads/*_R{1,2}.fq.gz"
params.outdir = "./results"
params.PE = true

// Fastqc process, creates zip files used for multiqc
process FASTQC {
    publishDir "${params.outdir}/fastqc", mode: 'copy'

    // Input are the reads channel
    input:
    tuple val(sample), val(extension), path(reads)

    output:
    path("*.zip")

    script:
    """
    fastqc ${reads}
    """
}

// FASTP_PE, fastp processing for paired end files. if params.PE (top) set to false then this process is not called
process FASTP_PE {
    publishDir "${params.outdir}/fastp", mode: 'copy'

    input:
    tuple val(sample), val(extension), path(reads)

    output:
```

```
    // Defining channel output emitted as "reads", and the fastp.json report as "json"
    tuple val(sample), val(extension),
path("${sample}_{1,2}.trimmed.{${extension[0]},${extension[1]}}"), emit: reads
    path("${sample}.fastp.json"), emit: json

    script:
    // Splitting the reads and extension lists into individual variables
    def read1 = reads[0]
    def read2 = reads[1]
    def extension1 = extension[0]
    def extension2 = extension[1]

    """
    fastp -i ${read1} -I ${read2} -o ${sample}_1.trimmed.${extension1} -O
${sample}_2.trimmed.${extension2} -j ${sample}.fastp.json
    """
}

// Multiqc allows for easy visualization of fastp and fastqc reports
process MULTIQC{
    publishDir "${params.outdir}/multiqc", mode: 'copy'

    input:
    // takes the fastp.json output and all the zip files from fastqc
    path(fastp_out)
    path(fastqc_out)

    output:
    path('*')

    script:
    // "multiqc ." effectively means "run multiqc on everything in the entire directory you
are in".
    // Because we aren't "in" any directory as we have to specify inputs to a process,
    // we will need to use the ".collect()" method for inputs when we call this process in the
workflow block (see below),
    // so as to hand multiqc everything we need at once.
    """
    multiqc .
    """
```

```
}

// Indexing our transcriptome
process SALMON_IDX {
    input:
    path(transcriptome)

    output:
    path("salmon_idx")

    script:
    """
    salmon index --threads ${task.cpus} -t ${transcriptome} -i salmon_idx
    """
}

// Salmon quantification step, requires our trimmed reads and the salmon index
process SALMON_QUANT_PE {
    publishDir "${params.outdir}/salmon", mode: 'copy'

    input:
    path(salmon_idx)
    tuple val(sample), val(extension), path(reads)

    output:
    path("*")

    script:
    def read1 = reads[0]
    def read2 = reads[1]

    """
    salmon quant --threads ${task.cpus} -i ${salmon_idx} -l A -1 ${read1} -2 ${read2} -o
${sample}.quant
    """
}


workflow {
```

```
    // Handles .fastq, .fq, .fastq.gz, .fq.gz files automatically, no need to modify
    if ( params.PE ) {
        reads_ch = channel.fromFilePairs( params.reads, checkIfExists: true )
        .map { sample_id, files ->
            def extensions = files.collect { file ->
                def file_name = file.name
                def idx = file_name.indexOf(".")
                def extension = file_name.substring(idx+1)
                extension
                }
            tuple(sample_id, extensions, files)
        }
    }
    else {
        reads = channel.fromPath( params.reads, checkIfExists: true )
        reads_ch = reads.map { file ->
            def file_name = file.name
            def idx = file_name.indexOf(".")
            def sample_id = file_name.substring(0, idx)
            def extension = file_name.substring(idx+1)
            tuple( sample_id, extension, file )
        }
    }

    // Viewing what the reads channel looks like
    reads_ch.view()

    // Fastqc and fastp takes reads channel input
    fastqc_out = FASTQC(reads_ch)
    fastp_out = FASTP_PE(reads_ch)

    // Multiqc requires just the "json" output from fastp, and everything from fastqc.
    // By default nextflow passes inputs as a stream to a process, but in this case we need
    // to hand everything to the multiqc process at once. To do this we use the ".collect()"
method
    MULTIQC(fastp_out.json.collect(), fastqc_out.collect())

    salmon_idx = SALMON_IDX(params.ref)
    // Notice again that we specify we only want to supply the reads output from fastp to
salmon quant
```

```
    // To do this we use "dot notation", fastp_out.reads (which we defined with the "emit"
keyword in the process block)
    SALMON_QUANT_PE(salmon_idx, fastp_out.reads)
}
```

Nextflow.config file with Slurm compatibility

```
process {
    executor = 'slurm'
    time = < number of hours ('h') days ('d') to let job run eg. '3d' >
    cpus = < number of cpus to request eg. 32>
    memory = < amount of RAM to request eg.'100.GB'>
}
```

PLEASE NOTE

These values need to be chosen carefully. If the `time` variable is not high enough, then the nextflow job may quit before it can complete. This is because nextflow submits these jobs through slurm which have a default max run time of 30 minutes. The longest your job can run on the HPC is 7 days.

Some tools required a certain amount of memory to be allocated (usually aligners). If you don't reserve enough, your job may error out.

Most tools have a flag to specify the number of cpus to use (may look like `-cpu`, `-threads`, `--threads` etc. Please see the documentation of the tool you're using to correctly specify) to supply that information in the workflow block there is a parameter called `${task.cpus}`

If you do not specify the number of threads for some processes (blastx is an offender) then it may default to a number of cpus specified in the tool - despite the fact you have requested to use a large number of cpus. This can ***severely*** increase the amount of time it will take your job to complete.

To correctly indicate the number of cpus for a process, see below.

(In nextflow.config)

```
process {
    executor = 'slurm'
    time = '5d'
    cpus = 38
    memory = '300.GB'
}
```

(In <your_nf_script>.nf)

```
process my_task {
    script:
    """
    my_command --threads ${task.cpus}
    """
}
```

The command in the process block above will effectively evaluate to `my_command --threads 38`

TEST DELETE