**Introduction**

In this post, we look at various tips that can be useful when automating R application testing and continuous integration, with regards to orchestrating parallelization, combining sources from multiple git repositories and ensuring proper access right to the Jenkins agent.

**Contents**

1. Running stages in parallel
   1. Parallel computation using R
   2. Orchestrating parallelization of R jobs with Jenkins
   3. Failing early
2. Cloning multiple git repositories
   1. Cloning into a separate subdirectory
   2. Cleaning up
3. Changing permissions to allow the Jenkins user to read
4. References

**Running stages in parallel**

**Parallel computation using R**

There are numerous way to achieve parallel computation in the context of an R application, those native to R are for example

* the [parallel package](https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf), which is included with base R since version 2.14 and very stable, or
* the more recent [future package](https://cran.r-project.org/package=future)
* the CRAN Task View: [High-Performance and Parallel Computing with R](https://cran.r-project.org/web/views/HighPerformanceComputing.html) provides a useful and extensive overview of multiple topics, including parallelism with R

Governing parallelism directly within R code requires tackling many aspects, starting with logging and ending in handling conditions and exception. We might therefore also be interested in leaving the orchestration of parallelism to a layer above the R application code itself. This approach has both benefits and limitations, so careful consideration should be taken before the implementation starts.

**Orchestrating parallelization of R jobs with Jenkins**

Declarative Jenkins pipelines are one of the ways to orchestrate parallelism with many options, a very simple example of a parallelized process can look as follows:

pipeline {

agent any

stages {

stage('Preparation') {

steps {

// Cleanup, Environment setup, etc.

}

}

stage('Tests') {

parallel {

stage('Unit Tests') {

steps {

// Invoke unit tests

}

}

stage('Integration Tests') {

steps {

// Invoke integration tests

}

}

stage('Regression Tests') {

steps {

// Invoke regression tests

}

}

stage('Technical checks') {

steps {

// Invoke Technical checks

}

}

}

}

}

}

Note the parallel directive, which will ensure that the (sub)stages within it

* Unit Tests
* Integration Tests
* Regression Tests and
* Technical checks

will be executed in parallel.

The parallelization will be orchestrated only after the first stage – “Preparation” was finished first. This is useful in case we need a stage that is shared among the parallel stages to be executed first.

**Failing early**

If we want to fail the parallel stages early (as soon as any of them fails), we can add failFast true into the parallel stage:

stage('Tests') {

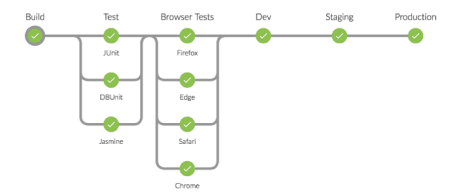
failFast true

parallel {

// ...

}

}



An example parallel Jenkins pipeline shown by BlueOcean.

**Cloning multiple git repositories**

In certain situations, we may need to clone not just the main repository that is subject to our multibranch pipeline, but also secondary repositories.

An example of such setup is when we store modeling parameters for our run in a separate repository, or when configurations governing the runs are stored in a separate repository.

The git directive allows us to clone another repository.

stage('Clone another repository') {

steps {

git branch: 'master',

credentialsId: 'my-credential-id',

url: 'git@github.com:user/repo.git'

}

}

**Cloning into a separate subdirectory**

Note however this will clone the repository into the current working directory, where the main repository subject to the pipeline is likely already checked out. This may have unintended consequences, so a safer approach is to checkout the secondary repository into a separate directory. We can achieve this using the dir directive:

stage('Clone another repository to subdir') {

steps {

sh 'rm subdir -rf; mkdir subdir'

dir ('subdir') {

git branch: 'master',

credentialsId: 'my-credential-id',

url: 'git@github.com:user/repo.git'

}

}

}

**Cleaning up**

After the pipeline is done, it may be useful do perform cleanup steps, for example removing unneeded directories. Since we likely want to clean those up regardless of the pipeline results, we can take advantage of the post directive running always, which will be executed regardless of the outcome of the pipeline stages.

One example use is to remove the hidden .git directories from both the working directory, where the main repository is checked out and the "subdir", where we checked out the secondary repository:

post {

always {

sh 'rm .git -rf'

sh 'rm subdir/.git -rf'

}

}

**Changing permissions to allow the Jenkins user to read**

One aspect of using Jenkins to execute our R code is to ensure that the Jenkins user executing the code on the worker node has access to all the necessary files. The following is a list of useful Linux commands that can help with the setup. These should, of course, be used with care.

# Add user `jenkins` to group `somegroup`

usermod -a -G somegroup jenkins

# Change group of somedir/ to somegroup, recursively

chgrp -R somegroup somedir/

# Allow group to read `somedir`, recursively

chmod -R g+r somedir/

# Find all directories in a path and allow group to traverse

find /dir/moredir/somedir -type d -exec chmod g+x {} \;