A Future for R: Future Topologies

Futures can be nested in R such that one future creates another set of futures and so on. This may, for instance, occur within nested for loops, e.g.

```
library("future")
library("listenv")
x <- listenv()
for (ii in 1:3) {
  x[[ii]] %<-% {
    y <- listenv()
    for (jj in 1:3) {
      y[[jj]] %<-% { ii + jj / 10 }
    }
    y
  }
}
unlist(x)
## [1] 1.1 1.2 1.3 2.1 2.2 2.3 3.1 3.2 3.3
```

The default is to use synchronous futures unless otherwise specified, which is also true for nested futures. If we for instance specify, `plan(multiprocess)`, the first layer of futures (`x[[ii]] %<-% { expr }`) will be processed asynchronously in background R processes, and the futures in the second layer of futures (`y[[jj]] %<-% { expr }`) will be processed synchronously in the separate background R processes. If we wish to be explicit about this, we can specify `plan(list(multiprocess, sequential))`.

Example: High-Throughput Sequencing

Consider a high-throughput sequencing (HT-Seq) project with 50 human DNA samples where we have one FASTQ file per sample containing the raw sequence reads as they come out of the sequencing machine. With this data, we wish to align each FASTQ to a reference genome such that we generate 24 individual BAM files per sample - one per chromosome.

Here is the layout of what such an analysis could look like in R using futures.

```
library("future")
library("listenv")
htseq_align <- function(fq, chr) { chr }

fqs <- dir(pattern = "[.]fastq$")

bams <- listenv()
for (ss in seq_along(fqs)) {
  fq <- fqs[ss]
  bams[[ss]] %<-% {
    bams_ss <- listenv()
    for (cc in 1:24) {
      bams_ss[[cc]] %<-% htseq_align(fq, chr = cc)
    }
    as.list(bams_ss)
  }
}
bams <- as.list(bams)
```

The default is to use synchronous futures, so without further specifications, the above will process each sample and each chromosome sequentially. Next, we will consider what can be done with the following two computer setups:

- A single machine with 8 cores
- A compute cluster with 3 machines each with 16 cores

One multi-core machine

With a single machine of 8 cores, we could choose to process multiple samples at the same time while processing chromosomes sequentially. In other words, we would like to evaluate the outer layer of futures using multiprocess futures and the inner ones as sequential futures. This can be specified as:

```
plan(list(multiprocess, sequential))
```

The internals for processing multiprocess future queries `availableCores()` to infer how many cores can be used simultaneously, so there is no need to explicitly specify that there are 8 cores available.

Comment: Since synchronous is the default future, we could skip trailing sequential futures in the setup, e.g. `plan(list(multiprocess))` or just `plan(multiprocess)`. However, it does not hurt to be explicit.

If we instead would like to process the sample sequentially and the chromosomes in parallel, we can use:

```
plan(list(sequential, multiprocess))
```

We could also process the data such that we allocate two cores for processing two samples in parallel each using four cores for processing four chromosomes in parallel:

```
plan(list(tweak(multiprocess, workers = 2), tweak(multiprocess, worke
```

An ad-hoc compute cluster

With a compute cluster of 3 machines each with 16 cores, we can run up to 48 alignment processes in parallel. A natural setup is to have one machine process one sample in parallel. We could specify this as:

```
nodes <- c("n1", "n2", "n3")
plan(list(tweak(cluster, workers = nodes), multiprocess))
```

Comment: Multiprocess futures are agile to its environment, that is, they will query the machine they are running on to find out how many parallel processes it can run at the same time.

One possible downside to the above setup is that we might not utilize all available cores all the time. This is because the alignment of the shorter chromosomes will finish sooner than the longer ones, which means that we might at the end of each sample have only a few alignment processes running on each machine leaving the remaining cores idle/unused. An alternative set up is then to use the following setup:

```
nodes <- rep(c("n1", "n2", "n3"), each = 8)
plan(list(tweak(cluster, workers = nodes), multiprocess))
```

This will cause up to 24 (= 3*8) samples to be processed in parallel each processing two chromosomes at the same time.

Example: A remote compute cluster

Imagine we have access to a remote compute cluster, with login node `remote.server.org`, and that the cluster has three nodes `n1`, `n2`, and `n3`. Also, let us assume we have already set up the cluster such that we can log in via public key authentication via SSH, i.e. when we do `ssh remote.server.org` authentication is done automatically.

With the above setup, we can use nested futures in our local R session to evaluate R expression on the remote compute cluster and its three nodes. Here is a proof of concept illustrating how the different nested futures are evaluated on different machines.

```
library("future")
library("listenv")

## Set up access to remote login node
login <- tweak(remote, workers = "remote.server.org")
plan(login)

## Set up cluster nodes on login node
nodes %<-% { .keepme <- parallel::makeCluster(c("n1", "n2", "n3")) }

## Specify future topology
## login node -> { cluster nodes } -> { multiple cores }
plan(list(
  login,
  tweak(cluster, workers = nodes),
  multiprocess
))


## (a) This will be evaluated on the cluster login computer
x %<-% {
  thost <- Sys.info()[["nodename"]]
  tpid <- Sys.getpid()
  y <- listenv()
  for (task in 1:4) {
    ## (b) This will be evaluated on a compute node on the cluster
    y[[task]] %<-% {
      mhost <- Sys.info()[["nodename"]]
      mpid <- Sys.getpid()
      z <- listenv()
      for (jj in 1:2) {
        ## (c) These will be evaluated in separate processes on the s
        z[[jj]] %<-% data.frame(task = task,
                                top.host = thost, top.pid = tpid,
                                mid.host = mhost, mid.pid = mpid,
                                host = Sys.info()[["nodename"]],
                                pid = Sys.getpid())
      }
      Reduce(rbind, z)
    }
  }
  Reduce(rbind, y)
}

print(x)
##   task top.host top.pid mid.host mid.pid host    pid
## 1    1    login  391547       n1  391878   n1 393943
## 2    1    login  391547       n1  391878   n1 393951
## 3    2    login  391547       n2  392204   n2 393971
## 4    2    login  391547       n2  392204   n2 393978
## 5    3    login  391547       n3  392527   n3 394040
## 6    3    login  391547       n3  392527   n3 394048
## 7    4    login  391547       n1  391878   n1 393959
## 8    4    login  391547       n1  391878   n1 393966
```

Try the above `x %<-% { ... }` future with, say, `plan(list(sequential, multiprocess))` and see what the output will be.