Leibniz Supercomputing Centre
of the Bavarian Academy of Sciences and Humanities

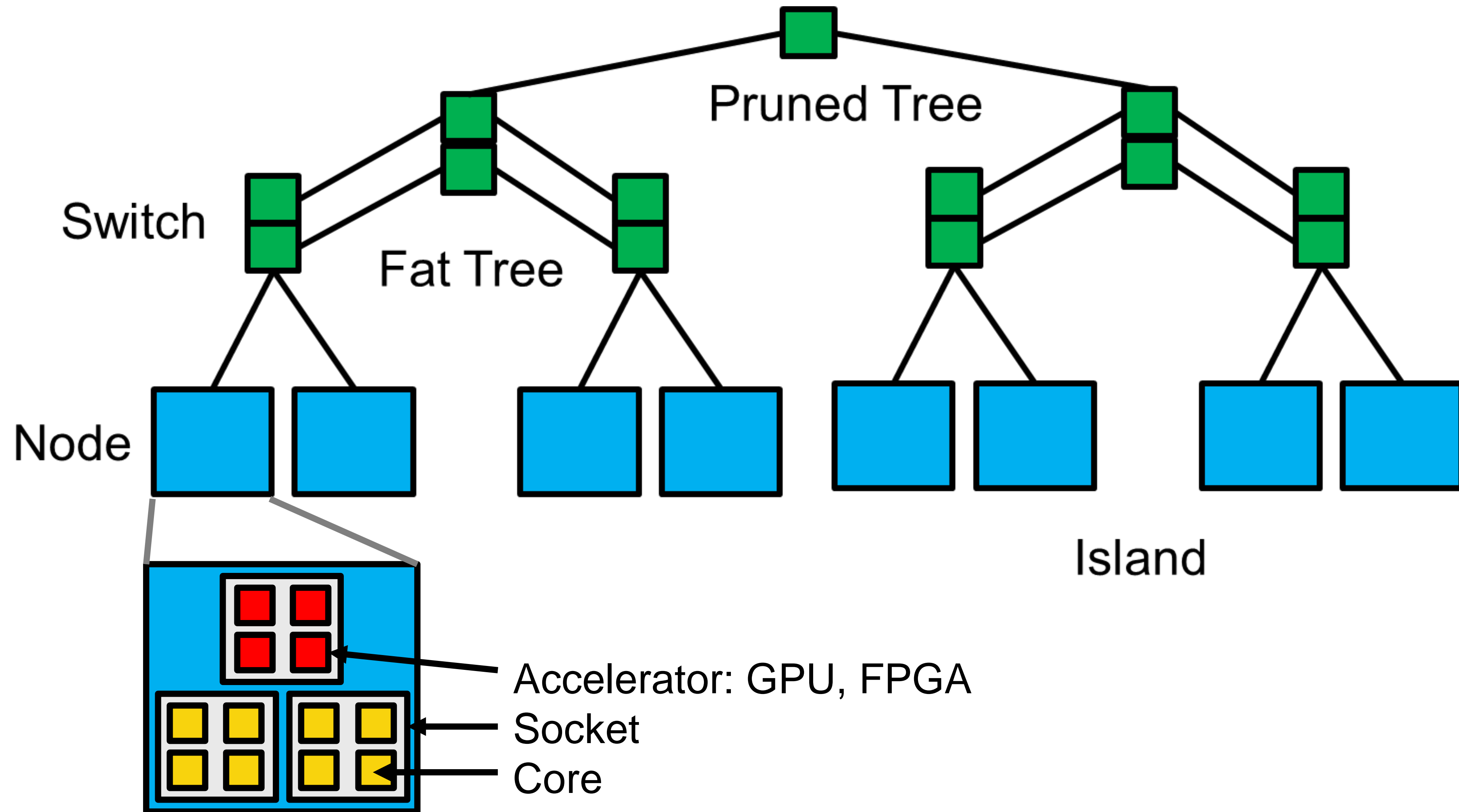
April 2021

# Parallelization Using R

Using R at LRZ | April 2021

lrz

# HPC Cluster Systems



# Parallelization

lrz

Motivation:

- You have a lot of (more or less) indepen-dent tasks or

- You want to accelerate a single complex task -> it might be possible to turn the single complex task into many (more or less) independent tasks

…and you have access to a (massively parallel) supercomputer!

# Parallelization Scenario: Embarrassingly/Pleasingly Parallel

- many independent processes (10 - 100.000)
- individual task (list) for each process
- private memory for each process
- no communication between processes
- results are stored separately on a (large) storage medium

# Parallelization Scenario: Worker Queue

- many independent processes (10 - 100.000)
- central task scheduler (database)
- private memory for each process
- results are sent back to task scheduler
- re-scheduling of failed tasks possible

# Parallelization Scenario: Shared Memory

lrz

- a few processes working closely together (10-100)
- single task list (script/program)
- shared memory
  (cache coherent non-uniform memory architecture aka ccNUMA)
- results are kept in shared memory

# Parallelization Scenario: Message Passing

- many independent processes (10 - 100.000)
- one task list (script/program) for all processes
- each process can (in principle) talk to every other process
- private memory
- needs communication strategy in order to scale (area of optimization, e.g. nearest neighbor communication)
- beware of deadlocks!

https://cran.r-project.org/web/views/HighPerformanceComputing.html

# CRAN Task View: High-Performance and Parallel Computing

**CRAN Task View: High-Performance and Parallel Computing with R**

**Maintainer:** Dirk Eddelbuettel
**Contact:** Dirk.Eddelbuettel at R-project.org
**Version:** 2018-08-27
**URL:** https://CRAN.R-project.org/view=HighPerformanceComputing

This CRAN task view contains a list of packages, grouped by topic, that are useful for high-performance computing (HPC) with R. In this context, we are defining 'high-performance computing' rather loosely as just about anything related to pushing R a little further: using compiled code, parallel computing (in both explicit and implicit modes), working with large objects as well as profiling.

Unless otherwise mentioned, all packages presented with hyperlinks are available from CRAN, the Comprehensive R Archive Network.

Several of the areas discussed in this Task View are undergoing rapid change. Please send suggestions for additions and extensions for this task view to the task view maintainer .

Suggestions and corrections by Achim Zeileis, Markus Schmidberger, Martin Morgan, Max Kuhn, Tomas Radivoyevitch, Jochen Knaus, Tobias Verbeke, Hao Yu, David Rosenberg, Marco Enea, Ivo Welch, Jay Emerson, Wei-Chen Chen, Bill Cleveland, Ross Boylan, Ramon Diaz-Uriarte, Mark Zeligman, Kevin Ushey, Graham Jeffries, Will Landau, Tim Flutre, Reza Mohammadi, Ralf Stubner, and Bob Jansen (as well as others I may have forgotten to add here) are gratefully acknowledged.

Contributions are always welcome, and encouraged. Since the start of this CRAN task view in October 2008, most contributions have arrived as email suggestions. The source file for this particular task view file now also reside in a GitHub repository (see below) so that pull requests are also possible.

The `ctv` package supports these Task Views. Its functions `install.views` and `update.views` allow, respectively, installation or update of packages from a given Task View; the option `coreOnly` can restrict operations to packages labeled as *core* below.

**Direct support in R started with release 2.14.0** which includes a new package **parallel** incorporating (slightly revised) copies of packages multicore and snow. Some types of clusters are not handled directly by the base package 'parallel'. However, and as explained in the package

**Parallel computing: Explicit parallelism**

- Several packages provide the communications layer required for parallel computing. The first package in this area was rpvm by Li and Rossini which uses the PVM (Parallel Virtual Machine) standard and libraries. rpvm is no longer actively maintained, but available from its CRAN archive directory.
- In recent years, the alternative MPI (Message Passing Interface) standard has become the de facto standard in parallel computing. It is supported in R via the Rmpi by Yu. Rmpi package is mature yet actively maintained and offers access to numerous functions from the MPI API, as well as number of R-specific extensions. Rmpi can be used with the LAM/MPI, MPICH / MPICH2, Open MPI, and Deino MPI implementations. It should be noted that LAM/MPI is now in maintenance mode, and new development is focused on Open MPI.
- The pbdMPI package provides S4 classes to directly interface MPI in order to support the Single Program/Multiple Data (SPMD) parallel programming style which is particularly useful for batch parallel execution. The pbdSLAP builds on this and uses scalable linear algebra packages (namely BLACS, PBLAS, and ScaLAPACK) in double precision based on ScaLAPACK version 2.0.2. The pbdBASE builds on these and provides the core classes and methods for distributed data types upon which the pbdDMAT builds to provide distributed dense matrices for "Programming with Big Data". The pbdNCDF4 package permits multiple processes to write to the same file (without manual synchronization) and supports terabyte-sized files. The pbdDEMO package provides examples for these packages, and a detailed vignette. The pbdPROF package profiles MPI communication SPMD code via MPI profiling libraries, such as fpmpi, mpiP, or TAU.
- An alternative is provided by the nws (NetWorkSpaces) packages from REvolution Computing. It is the successor to the earlier LindaSpaces approach to parallel computing, and is implemented on top of the Twisted networking toolkit for Python.
- The snow (Simple Network of Workstations) package by Tierney et al. can use PVM, MPI, NWS as well as direct networking sockets. It provides an abstraction layer by hiding the communications details. The snowFT package provides fault-tolerance extensions to snow.
- The snowfall package by Knaus provides a more recent alternative to snow. Functions can be used in sequential or parallel mode.

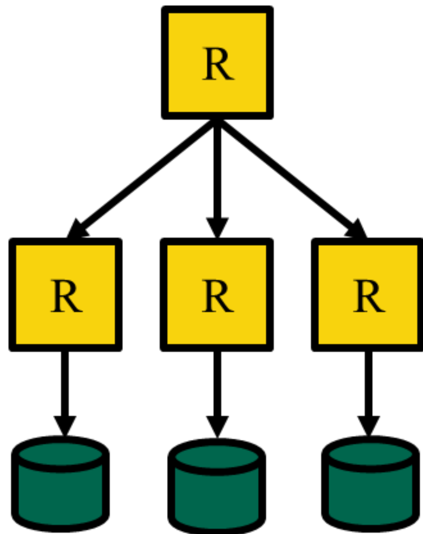
The foreach package allows general iteration over elements in a collection without the use of an explicit loop counter.

- The Rborist package employs OpenMP pragmas to exploit predictor-level parallelism in the Random Forest algorithm which promotes efficient use of multicore hardware in restaging data and in determining splitting criteria, both of which are performance bottlenecks in the algorithm.
- The h2o package connects to the h2o open source machine learning environment which has scalable implementations of random forests, GBM, GLM (with elastic net regularization), and deep learning.
- The randomForestSRC package can use both OpenMP as well as MPI for random forest extensions suitable for survival analysis, competing risks analysis, classification as well as regression
- The parSim package can perform simulation studies using one or multiple cores, both locally and on HPC clusters.
- The qsub package can submit commands to run on gridengine clusters.

# (Explicit) Parallelization Using R

- Embarrassingly/pleasingly parallel (independent processes):
  - basic approach: start as many R processes as you need in the shell with different scripts

# Parallelization Using R: Embarrassingly/Pleasingly parallel




Embarrassingly/Pleasingly
Parallel

```
$ R -f script.R &
```

# Parallelization Using R: Embarrassingly/pleasingly parallel

- Use the command line to start your R process (in the background):
  ```
  $ Rscript script0.R &
  ```
- If you do this repeatedly, the resulting R processes will be distributed by the OS to different cores (subject to availability):
  ```
  $ Rscript script1.R &
  $ Rscript script2.R &
  $ Rscript script3.R & …
  ```
- To further automate this procedure, you could write a bash script (run_all_R_scripts.sh) containing these commands and then run this single script:
  ```
  $ bash run_all_R_scripts.sh &
  ```

- Do not start more processes than cores!
- Do not use the (cluster) login nodes for this (e.g. request an interactive shell instead)!

# Parallelization Using R: Embarrassingly/pleasingly Parallel

- Let’s look at a toy problem:

```
for(i in 1:20) sum(sort(runif(1e7)))
```

- Are there parallelization opportunities?
- Add a time measurement:

```
system.time(for(i in 1:20) sum(sort(runif(1e7))))
```

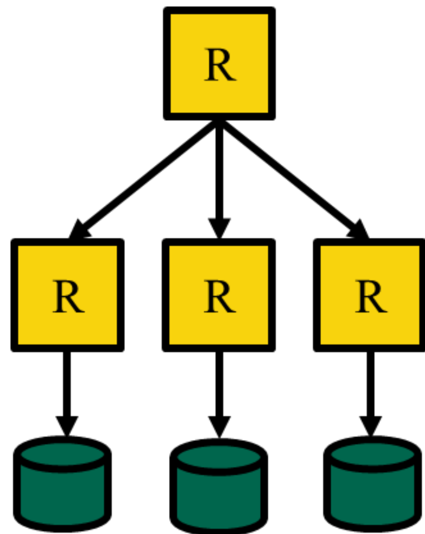
- You might also be familiar with alternatives like the following:

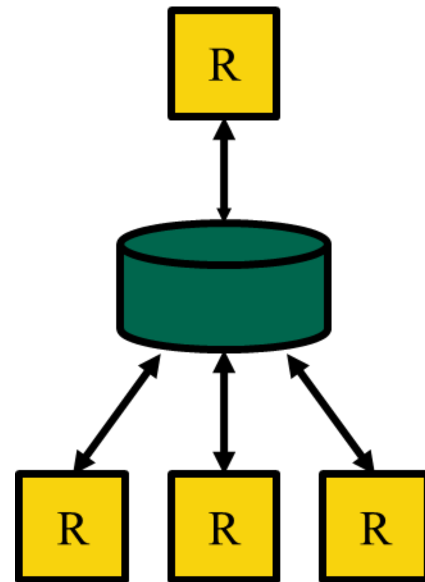
```
lapply(1:20, function(x) sum(sort(runif(1e7))))
```

# (Explicit) Parallelization Using R

- Embarrassingly/pleasingly parallel (independent processes):
  - basic approach: start as many R processes as you need in the shell with different scripts
- Worker Queue (weak coupling, shared file system or database):
  - a main process (with access to a database/shared file system) coordinates several R processes, potentially on different compute nodes (e.g. batchtools, rredis/doRedis)

# Parallelization Using R: Worker Queue




Embarrassingly/Pleasingly Parallel

```
$ R -f script.R &
```

Shared file system or database

job steps/srun, batchtools, rredis/doRedis

https://mllg.github.io/batchtools/

# Parallelization Using R: batchtools

“batchtools provides a parallel implementation of Map for high performance computing systems managed by schedulers like Slurm, …

- all relevant batch system operations (submitting, listing, killing) are either handled internally or abstracted via simple R functions
- with a well-defined interface, the source is independent from the underlying batch system - prototype locally, deploy on any high performance cluster”

i.e. a (interactive) R process is used in combination with the shared file system and the workload manager of the cluster to distribute workloads across nodes

# Parallelization Using R: rredis/doRedis

Redis is an open source, fast, persistent, networked database with many features, among them a blocking queue-like data structure (Redis “lists”). This feature makes Redis useful as a lightweight back end for parallel computing.

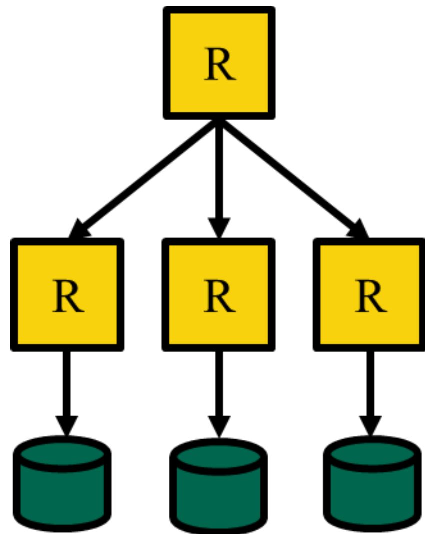
A Redis server has to be set up as part of the cluster (e.g. on a login node) or even somewhere else, containing the problem description(s). Worker processes connect to this server and tasks are assigned to them.

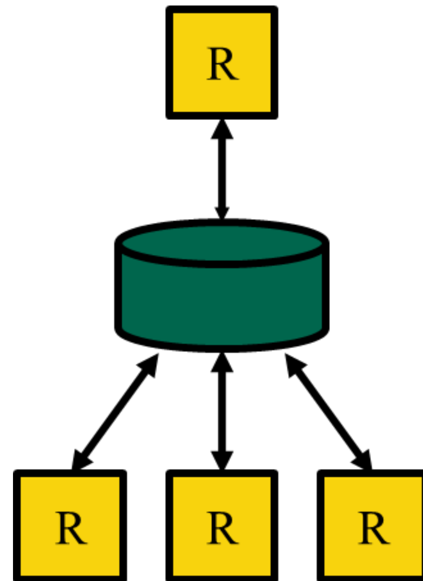
This is a very flexible and dynamic approach, as workers can basically run wherever you want (as long as they can connect to the server). When running on the cluster, you have to deal with resource allocation separately (via the Slurm workload manager) and potential firewall access restrictions.

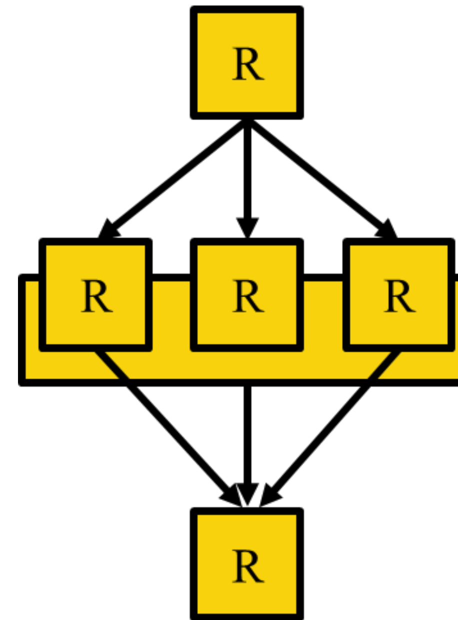
# (Explicit) Parallelization Using R

- Embarrassingly/pleasingly parallel (independent processes):
  - basic approach: start as many R processes as you need in the shell with different scripts
- Worker Queue (weak coupling, shared file system or database):
  - a main process (with access to a database/shared file system) coordinates several R processes, potentially on different compute nodes (e.g. batchtools, rredis/doRedis)
- Shared Memory (strong coupling):
  - one R process spawns sub-processes on a single node with many cores (e.g. parallel/doParallel; formerly multicore/doMC, snow/doSNOW)

# Parallelization Using R: Shared Memory




Embarrassingly/Pleasingly Parallel

```
$ R -f script.R &
```

Shared file system or database

job steps/srun, batchtools, rredis/doRedis

Shared memory

parallel/doParallel

# Shared Memory Parallelization: Multithreading with doParallel

- As seen earlier, the for loop construct in R:
  ```
  for(i in 1:20) sum(sort(runif(1e7)))
    # serial execution/single thread
  ```

- “The foreach package provides a new looping construct for executing R code repeatedly. […] it supports parallel execution, that is, it can execute those repeated operations on multiple processors/cores on your computer, or on multiple nodes of a cluster.”

```
library(foreach)
foreach(i = 1:20) %do% sum(sort(runif(1e7)))  # serial execution

foreach(i = 1:20) %dopar% sum(sort(runif(1e7)))
  # multithread execution (?)
```

# Shared Memory Parallelization: Multithreading with doParallel

- This is were the “do-back ends” (e.g. doParallel) come into play…
- By creating/registering a cluster, foreach’s %dopar% operator can rely on these parallel resources, e.g. using parallel’s multicore-like functionality (“forking”):

```
library(foreach)
library(doParallel)
registerDoParallel(cores=2)
  # define number of cores, this enables multicore-functionality
  # (preferred on GNU/Linux, but won’t work on Windows)
foreach(i = 1:20) %dopar% sum(sort(runif(1e7)))
```

# Shared Memory Parallelization: Multithreading with doParallel

lrz

- The procedure is similar for snow-like functionality:

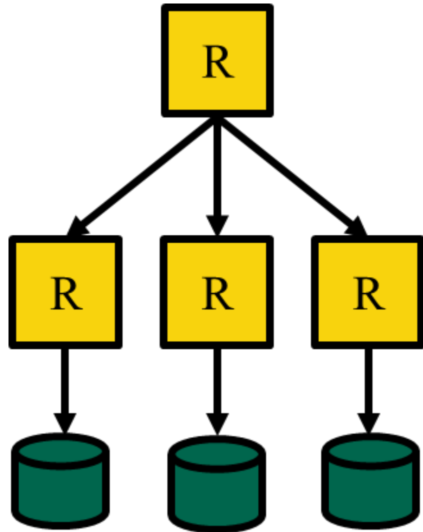
```
library(foreach)
library(doParallel)
cluster.object <- makePSOCKcluster(2)
registerDoParallel(cluster.object)
foreach(i = 1:20) %dopar% sum(sort(runif(1e7)))
stopCluster(cluster.object)
```

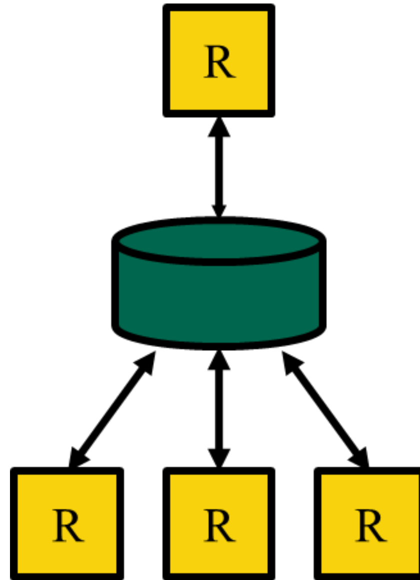
- This uses Rscript to launch further copies of R (on the same host or optionally elsewhere; in the latter case, hostnames need to be provided)
- [parallel‘s snow-like functionality also allows to create MPI-clusters (makeMPIcluster()-function) but Rmpi/doMPI is usually recommended to be used instead]

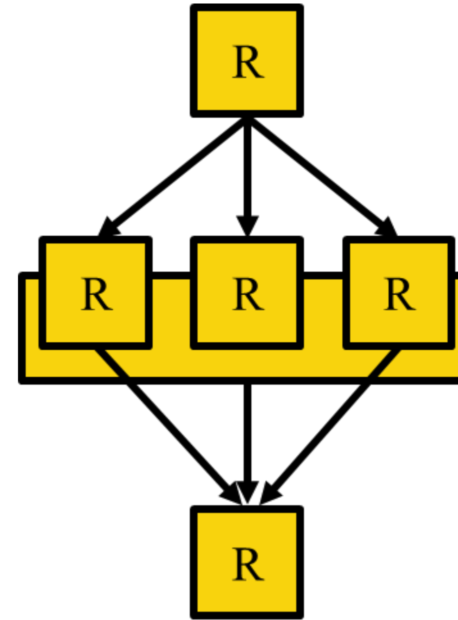
# (Explicit) Parallelization Using R

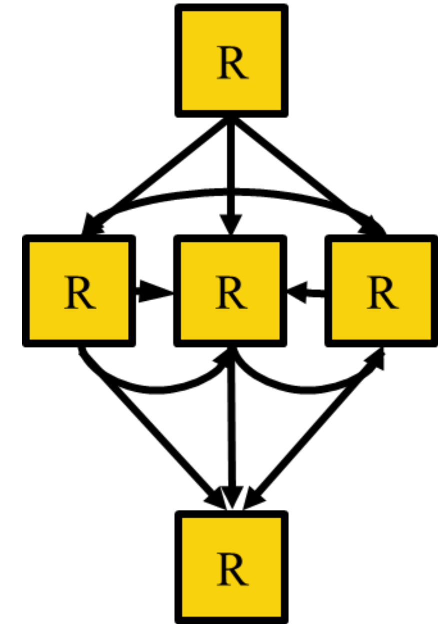
- Embarrassingly/pleasingly parallel (independent processes):
  - basic approach: start as many R processes as you need in the shell with different scripts
- Worker Queue (weak coupling, shared file system or database):
  - a main process (with access to a database/shared file system) coordinates several R processes, potentially on different compute nodes (e.g. batchtools, rredis/doRedis)
- Shared Memory (strong coupling):
  - one R process spawns sub-processes on a single node with many cores (e.g. parallel/doParallel; formerly multicore/doMC, snow/doSNOW)
- Message Passing (strong coupling):
  - several R processes talk to each other (across different nodes) by passing messages (e.g. Rmpi/doMPI), this also allows for a (single) main and (multiple) workers model

# Parallelization Using R: Message Passing




| Embarrassingly/Pleasingly Parallel | Shared file system or database | Shared memory | Message Passing |
|---|---|---|---|
| `$ R -f script.R &` | job steps/srun, batchtools, rredis/doRedis | parallel/doParallel | Rmpi/doMPI |

# Message Passing with doMPI

- To execute a doMPI script on multiple compute nodes a “message passing environment” needs to be set up, i.e. the R interpreter needs to be executed using a command such as `mpirun` (i.e. `mpirun R –f script.R`)

- Then, the already familiar „do-back end“-pattern is put to use within R:

```
library(foreach)
library(doMPI)
cluster.object <- startMPIcluster()
registerDoMPI(cluster.object)
foreach(i = 1:20) %dopar% sum(sort(runif(1e7)))
closeCluster(cluster.object)
```

# More foreach()

- use times() for simple repetitions:
```
times(10) %do% sum(sort(runif(1e7)))
```
- foreach is a function with several arguments…
```
foreach(i = 1:10, .combine = c, …) %do% sth()  #  process results
    as they get generated, e.g. c(), cbind(), list(), sum(), ...
```
- … evaluates iterators…
```
foreach(i = iter(input)) %do% sth()  # see package iterators
foreach(i = irnorm(100)) %do% sth()
```
- … and provides additional operators:
```
foreach(i = 1:10) %:% when(cond) %do%  sth()  # nesting operator
    and condition cf. Python's list comprehensions
```

# More parallel

lrz

- parallel provides parallel replacements of lapply and related functions (as have snow and multicore):

  - multicore-like: e.g. mclapply(1:10, function(x) sum(sort(runif(1e7)))),
    mcmapply (x, FUN, ...), mcMap(FUN, …)

  - snow-like: clusterApply(cl, x, fun, ...), e.g. parLapply(cl, x, FUN, ...)

# Even More parallel: Futures/Promises

- Constructs for synchronizing program execution. Describe objects that act as proxies for a result, which is yet unknown (because the computation is incomplete)
- Send command to background and return handle:
  handle <- mcparallel(some_expensive_function)
- Collect result at later point:
  result <- mccollect(handle)

# Futures/Promises

lrz

```
> system.time(sum(sort(runif(1e7))))
      user  system elapsed
     1.581   0.112   1.700

> system.time(sapply(1:20, function(x) sum(sort(runif(1e7)))))
       user  system elapsed
     28.875   2.998  31.883

> library(parallel)
> h <- mcparallel(sapply(1:20, function(x) sum(sort(runif(1e7)))))
> mccollect(h, wait = FALSE)
NULL
# wait approx. 30 seconds for job to finish
> mccollect(h, wait = FALSE)
[1] 5000214 4999121 5001166 …
```

https://github.com/HenrikBengtsson/future

# Futures/Promises

- Package future tries to unify the previous approaches:
  "The purpose of this package is to provide a lightweight and unified Future API for sequential and parallel processing of R expressions via futures. […] Because of its unified API, there is no need to modify any code in order switch from sequential on the local machine to, say, distributed processing on a remote compute cluster. "

- Implicit:
```
v %<-% { expr }  # future assignment , creates a future and a
    promise to its value (instead of regular assignment <-)
```

- Explicit:
```
f <- future({ expr })  # creates a future
v <- value(f)  # gets the value of the future
    (blocks if not yet resolved)
```

https://github.com/HenrikBengtsson/future

# Futures/Promises

- Function plan() allows the user to plan the future, i.e. it specifies how futures()s are resolved

- For example: plan(sequential) vs. plan(multiprocess)

```
> library("future")
> plan(multiprocess)
> v %<-% {
+   cat("Hello world!\n")
+   3.14
+ }
> v
Hello world!
[1] 3.14
```

https://github.com/HenrikBengtsson/future

# Futures/Promises

| Name | OSes | Description |
| --- | --- | --- |
| *synchronous:* | | *non-parallel:* |
| sequential | all | sequentially and in the current R process |
| transparent | all | as sequential w/ early signaling and w/out local (for debugging) |
| *asynchronous:* | | *parallel:* |
| multiprocess | all | multicore, if supported, otherwise multisession |
| multisession | all | background R sessions (on current machine) |
| multicore | not Windows | forked R processes (on current machine) |
| cluster | all | external R sessions on current, local, and/or remote machines |
| remote | all | simple access to remote R sessions |

- Additionally: package future.batchtools provides an implementation of the Future API on top of the batchtools package, i.e. it allows to process futures (as defined by the future package) on HPC infrastructure

https://github.com/HenrikBengtsson/doFuture

# doFuture: future and foreach

- Package doFuture provides a %dopar% adaptor for the foreach package such that any type of future (that is supported by the Future API of the future package) can be used for asynchronous (parallel/distributed) or synchronous (sequential) processing.
- Example:

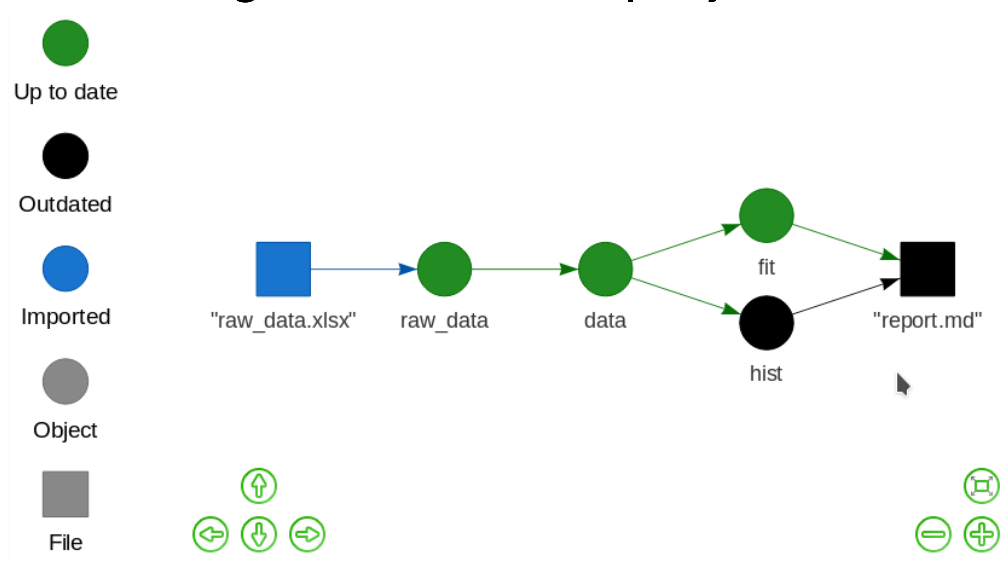
```
library(doFuture)
registerDoFuture()
plan(multiprocess)
foreach(i = 1:20) %dopar% sum(sort(runif(1e7)))
```

- Look out for the use of foreach (and the possibility to register all these different back ends) in other R packages!

https://ropensci.github.io/drake/

# … and beyond.

- Drake is a general-purpose workflow manager for data-driven tasks.
- It rebuilds intermediate data objects when their dependencies change, and it skips work when the results are already up to date. Not every run-through starts from scratch, and completed workflows have tangible evidence of reproducibility.
- drake supports scalability, parallel computing (relying on the parallel, future, batchtools, and future.batchtools packages), and a smooth user experience when it comes to setting up, deploying, and maintaining data science projects.



# Conclusion

lrz

- Parallel programming is here to stay (for the foreseeable future).
- Know your hardware…
- … and the possibilities of your software/programming environment.
- Applying proper (high level) abstractions (foreach, futures,…) to target the features of modern CPUs/GPUs and supercomputing infrastructure will allow you to write fast and scalable programs.