PARALLEL PROGRAMMING IN R

Are my results reproducible?

Hana Sevcikova
University of Washington

Random numbers in R

- Many statistical applications involve random numbers (RNs)
- Examples: MCMCs in Bayesian methods, bootstrap, simulations

- For reproducibility:
  - Set **seed of a random number generator** (RNG) prior to running the code

```
set.seed(1234)
rnorm(3)
[1] -1.2070657  0.2774292  1.0844412

rnorm(3)
[1] -2.3456977  0.4291247  0.5060559

set.seed(1234)
rnorm(3)
[1] -1.2070657  0.2774292  1.0844412

rnorm(3)
[1] -2.3456977  0.4291247  0.5060559
```

Naive (non)reproducibility in parallel code

```
library(parallel)
cl <- makeCluster(2)
```

```
set.seed(1234)
clusterApply(cl, rep(3, 2), rnorm)

[[1]]
[1] -1.891091 -1.351767 -1.456848

[[2]]
[1]  1.7346577  0.7855641 -2.2319774
```

```
set.seed(1234)
clusterApply(cl, rep(3, 2), rnorm)

[[1]]
[1]  0.4432499 -0.7896067  0.2659675

[[2]]
[1]  0.2229560  0.8323269 -0.4092570
```

Incorrect way of generating RNs in parallel code

- Using `set.seed()`, the RNG is initialized only on the master.
- Workers start with a clean environment, thus no RNG seed set.
- What happens when we set the RNG on each worker?

```
clusterEvalQ(cl, set.seed(1234))
clusterApply(cl, rep(3, 2), rnorm)
[[1]]
[1] -1.2070657  0.2774292  1.0844412

[[2]]
[1] -1.2070657  0.2774292  1.0844412
```

Another incorrect way of generating RNs in parallel code

- Quick and dirty solution:

```
for (i in 1:2) {
    set.seed(1234)
    clusterApply(cl, sample(1:10000000, 2), set.seed)

    print(clusterApply(cl, rep(3, 2), rnorm))
}

[[1]]
[1]  0.078249533  0.003019703 -1.314239709

[[2]]
[1]  1.3955357 -0.9935141 -0.3740712

[[1]]
[1]  0.078249533  0.003019703 -1.314239709

[[2]]
[1]  1.3955357 -0.9935141 -0.3740712
```

- NOT RECOMMENDED!!!

PARALLEL PROGRAMMING IN R

Let's practice!

PARALLEL PROGRAMMING IN R

Parallel random number generators

Hana Sevcikova
University of Washington

Random Number Generators (RNGs)

- Important parameters of an RNG:
  - long period (preferably $> 2^{100}$)
  - good structural (distributional) properties in high dimensions
- These parameters should hold when used in distributed environment

L'Ecuyer Multiple Streams RNG

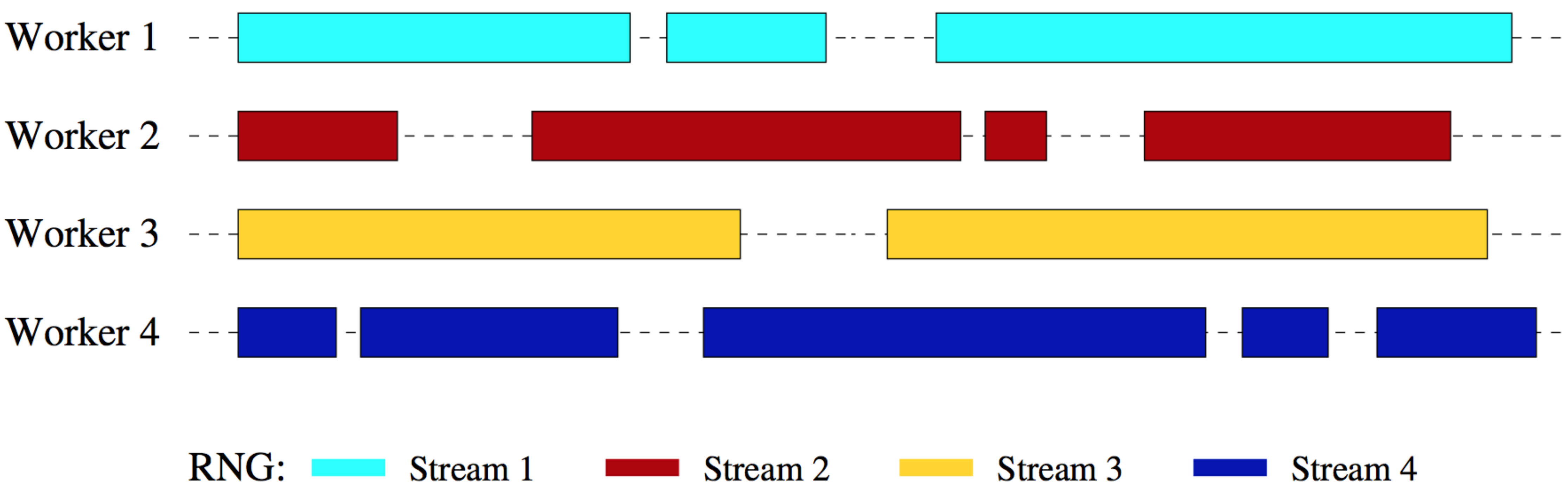
- A good quality RNG with multiple independent streams proposed by Pierre L'Ecuyer et al. (2002), RngStreams
  - Period 2^{191}
  - Streams have seeds 2^{127} steps apart
  - Parallel parts of user computation can use independent and reproducible streams
  - Direct interface in R: `rlecuyer`, `rstream`
  - In R core: `RNGkind("L'Ecuyer-CMRG")`

Using L'Ecuyer RNG in parallel

- Setting an RNG seed for cluster `cl`:

```
clusterSetRNGStream(cl, iseed = 1234)
```

- Initializes a reproducible independent stream on each worker



Reproducibility in the parallel package

- In `parallel`: one stream per worker
- Creates constraints on reproducibility
- Results only reproducible if:

  1. process runs on clusters of the same size
  2. process does **not** use load balancing, e.g. `clusterApplyLB()`

PARALLEL PROGRAMMING IN R

Let's practice!

PARALLEL PROGRAMMING IN R

Reproducibility in foreach and future.apply

Hana Sevcikova
University of Washington

doRNG: backend for foreach

Using doRNG via %dorng%

```
library(doRNG)
library(doParallel)
registerDoParallel(cores = 3)
```

```
set.seed(1)
res1 <- foreach(n = rep(2, 5), .combine = rbind) %dorng% rnorm(n)
```

```
set.seed(1)
res2 <- foreach(n = rep(2, 5), .combine = rbind) %dorng% rnorm(n)
```

```
identical(res1, res2)

[1] TRUE
```

Using doRNG via %dopar%

```
library(doRNG)
library(doParallel)
registerDoParallel(cores = 3)
```

```
registerDoRNG(1)
res3 <- foreach(n = rep(2, 5), .combine = rbind) %dopar% rnorm(n)
```

```
set.seed(1)
res4 <- foreach(n = rep(2, 5), .combine = rbind) %dopar% rnorm(n)
```

```
c(identical(res1, res3), identical(res2, res4))

[1] TRUE TRUE
```

Note: Cannot be used with the `%doSEQ%` backend.

Summary of using doRNG

Two ways of including `doRNG` into `foreach`:

1. Using **`%dorng%`**:
    - advantage of being explicit about using the L'Ecuyer's RNG
2. Using **`%dopar%`** and registering **doRNG**:
    - easy to make code/packages reproducible by only prepending

      `registerDoRNG()`

`doRNG` can be used with any parallel backend, including `doFuture`.

future.apply

- Uses independent streams of the L'Ecuyer's RNG
- As in `doRNG`, generates one stream per task
- Need only to assign `future.seed` argument

```
library(future.apply)
plan(sequential)
res5 <- future_lapply(1:5, FUN = rnorm, future.seed = 1234)

plan(multiprocess)
res6 <- future_lapply(1:5, FUN = rnorm, future.seed = 1234)

identical(res5, res6)

[1] TRUE
```

PARALLEL PROGRAMMING IN R

Let's practice!

PARALLEL PROGRAMMING IN R

Finishing Touch

Hana Sevcikova
Senior Research Scientist, University of Washington

Recommended R packages

- **parallel** (core package)
  - No need for dependencies on other packages
  - Important to understand as other packages are built on it
  - Often yields best performance
  - Reproducible results: only on clusters of the same size with no load balancing

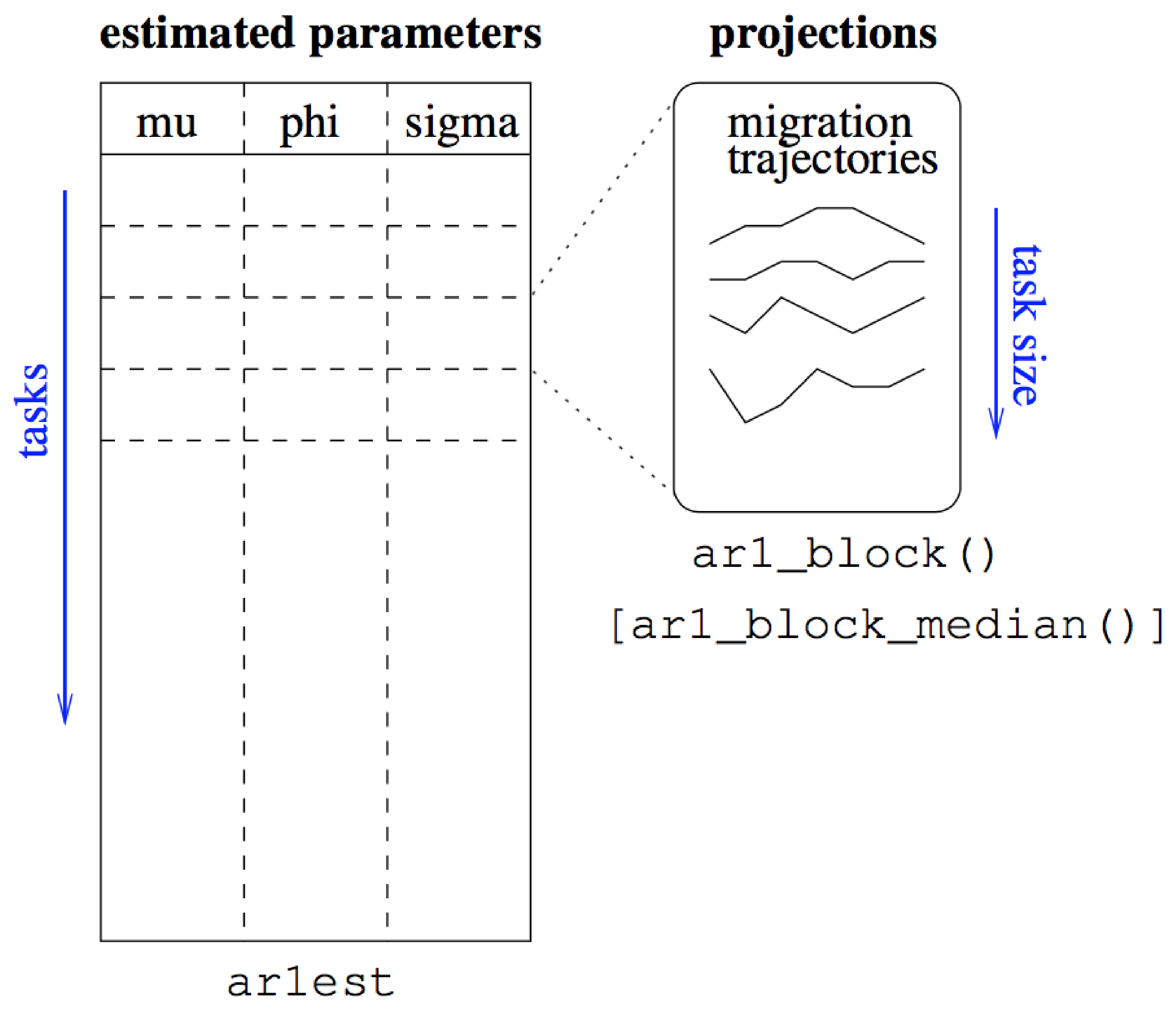
Recommended R packages (cont.)

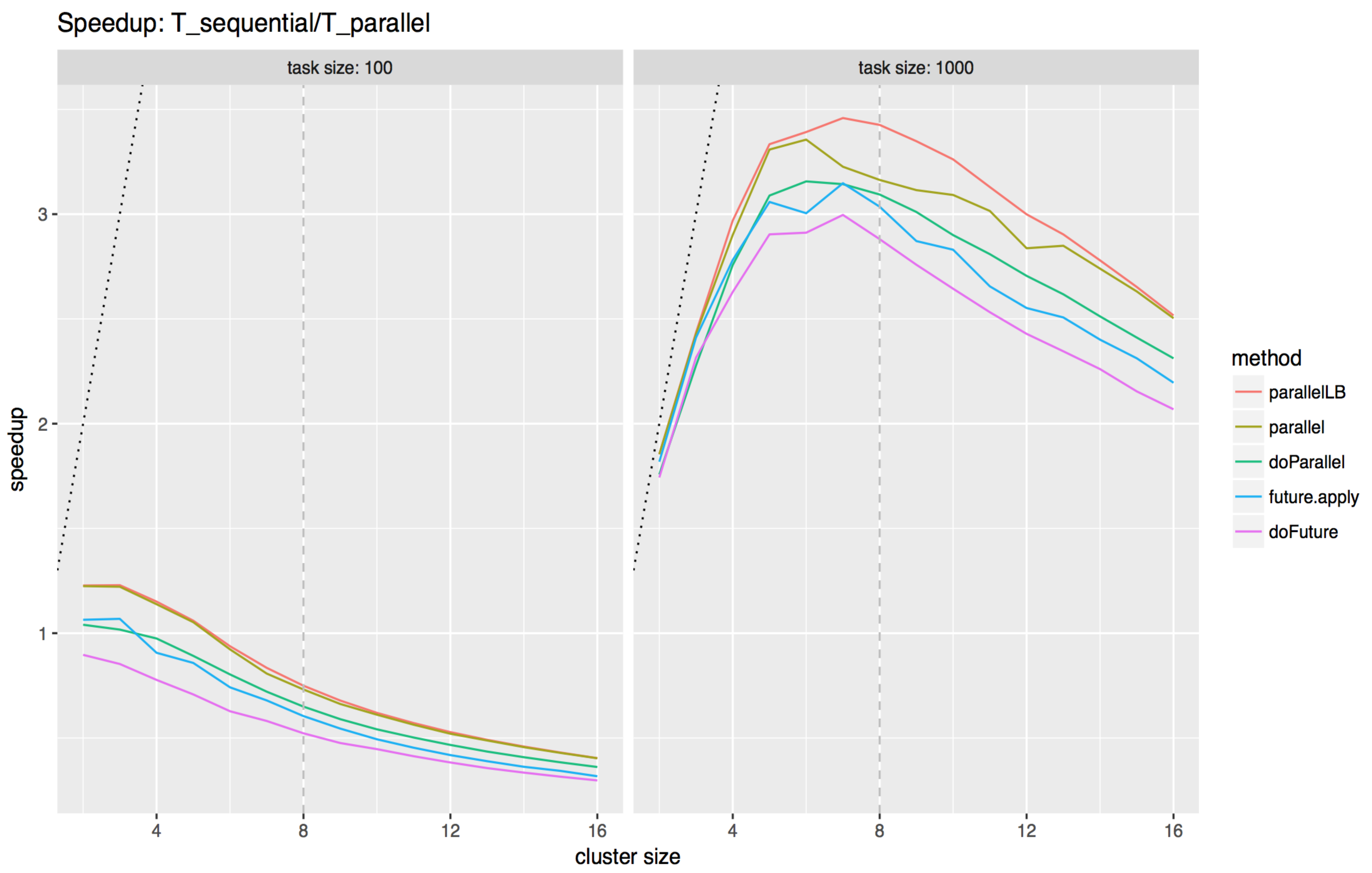
- **foreach** (with **doParallel**, **doFuture**)
  - Higher level programming
  - Intuitive syntax in form of `for` loops
  - Results reproducible via **doRNG**
- **future.apply** (based on **future**)
  - Unifies many parallel backends into one interface
  - Intuitive `apply()`-like syntax
  - Results always reproducible

Getting the best performance

- Minimize amount of **communication** (sending repeatedly big data is bad!)
- Use **scheduling** and **load balancing** appropriate for your application (e.g. group tasks into chunks evenly distributed across workers)
- Use **cluster size** appropriate for your hardware (i.e. number of physical cores)

Probabilistic projection of migration




Speedup: T_sequential/T_parallel
task size: 100
task size: 1000
speedup
cluster size
1
2
3
4
8
12
16
method
parallelLB
parallel
doParallel
future.apply
doFuture

PARALLEL PROGRAMMING IN R

Final Slide