Adventures in Supercomputing with R

Lecture 3: Parallel Speedup, Shared Memory Multicore

George Ostrouchov

Oak Ridge National Laboratory and University of Tennessee

2022/02/20 (updated: 2022-03-02)

Review of Last Lecture

- Shared memory and distributed memory
    - One program in both cases
    - Distributed programming works in shared memory

- Shared memory devices range between MIMD and SIMD extremes

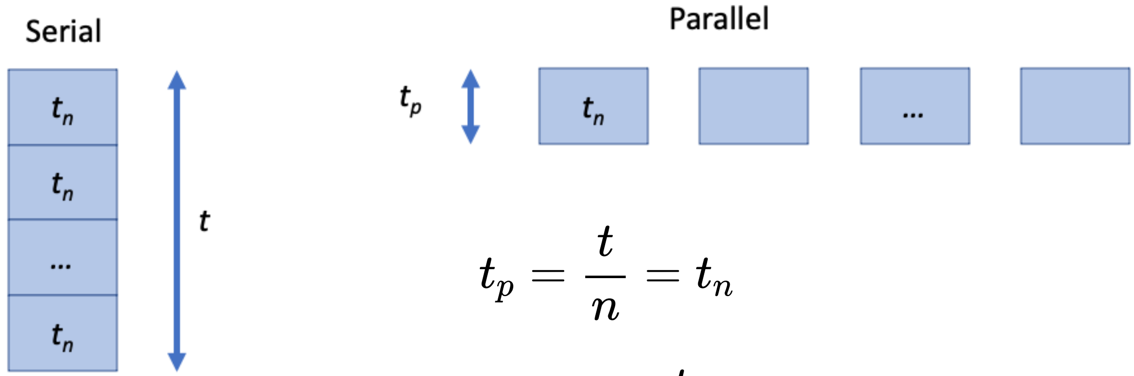
    - Processor and co-processor
    - Multicore processors are MIMD and more easily programmable
    - GPUs co-processors are very efficient at SIMD
    - Manycore processors can be best of both worlds

- "Fork a repository" versus "Unix fork"

- There are other ways to edit remote code besides RStudio and GitHub

Measurement and terminology of parallel speedup ...

Embarrassingly (Pleasingly) Parallel

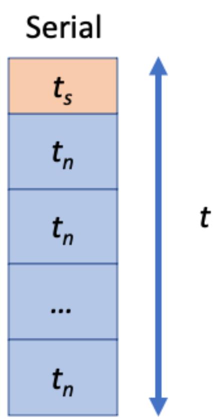


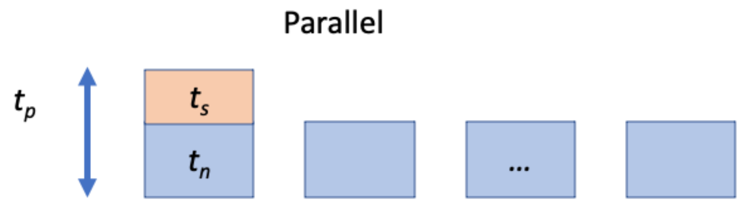
$$t_p = \frac{t}{n} = t_n$$

$$\text{Speedup:} \quad \frac{t}{t_p} = n$$

- t: Serial time
- n Number of chunks (or processes)
- t_n: Single chunk time with n chunks
- t_p: Parallel time

Serial Section (Amdahl's Law)





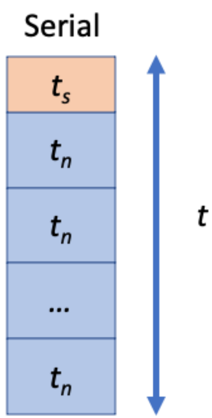
$$t_p = t_s + t_n > t_s$$

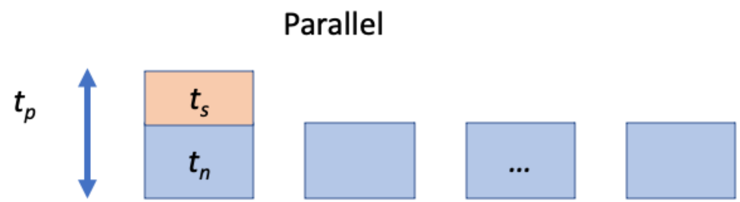
$$\text{Max Speedup:} \quad \lim_{n \to \infty} \frac{t}{t_p} = \frac{t}{t_s}$$

- t: Serial time (**fixed**)
- n Number of chunks (or processes)
- t_n: single chunk time with n chunks
- t_p: Parallel time
- t_s: Serial section time

Strong Scaling: fixed work, increasing resources

Serial Section (Gustafson's Law)





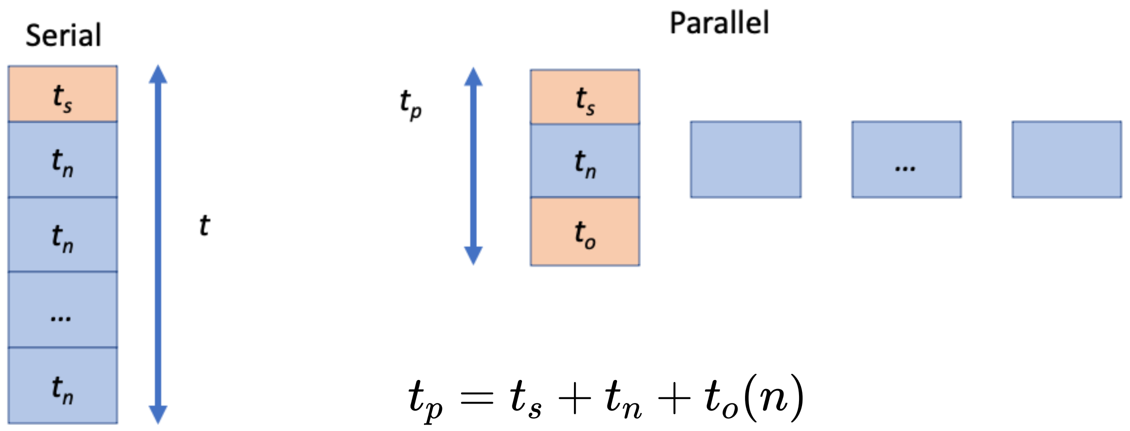
$$t_p = t_s + t_n$$

$$\text{Speedup:} \quad \frac{t}{t_p} = \frac{t_s + n t_n}{t_s + t_n} = O(n)$$

- t: Serial time (**growing**: $t_{2n} = 2t_n$)
- n: Number of chunks (or processes)
- t_n: single chunk time with n chunks
- t_p: Parallel time
- t_s: Serial section time

Weak Scaling: increasing work, increasing resources

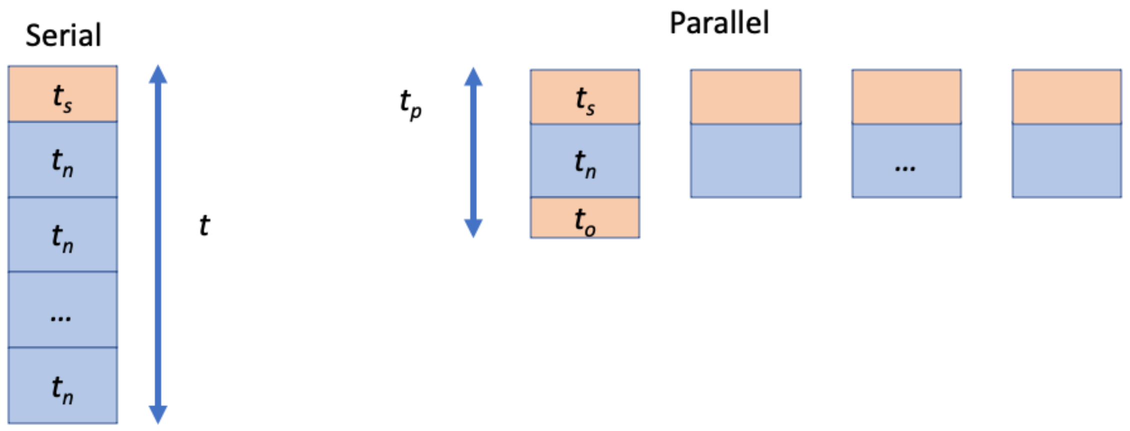
Parallel Overhead



$$t_p = t_s + t_n + t_o(n)$$

- t: Serial time
- n Number of processes
- t_n: single chunk time with n chunks
- t_p: Parallel time
- t_s: Serial section time
- $t_o(n)$: Parallel overhead time

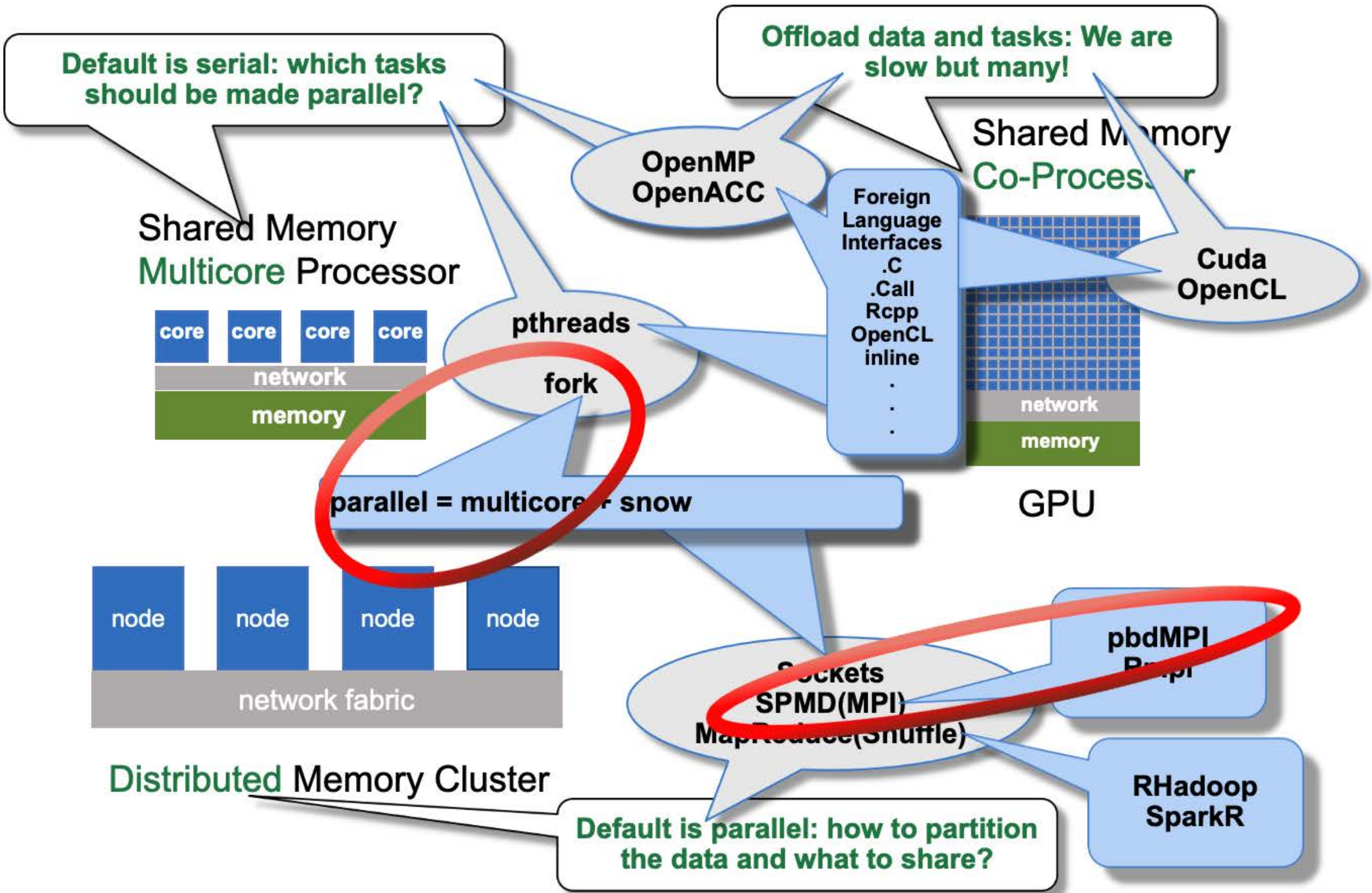
Preview - Distributed Compute Replication



- t: Serial time
- n Number of processes
- t_n: single chunk time with n chunks
- t_p: Parallel time
- t_s: Serial section time
- $t_o(n)$: Parallel overhead time

Replication can reduce communication overhead

R Interfaces to Low-Level Native Tools

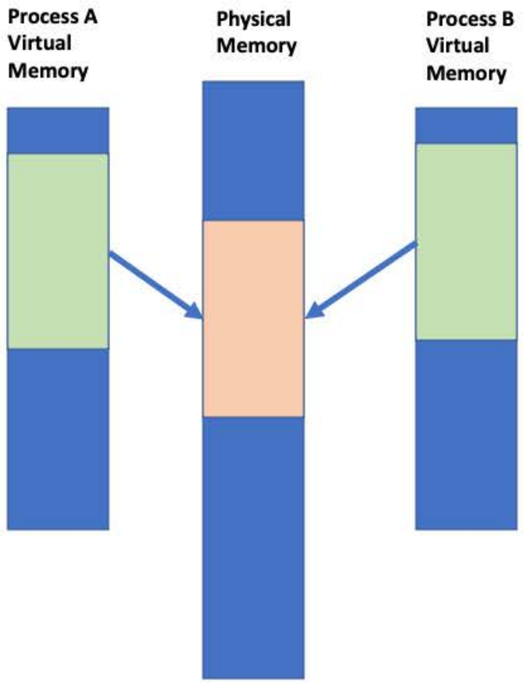


Unix `fork`

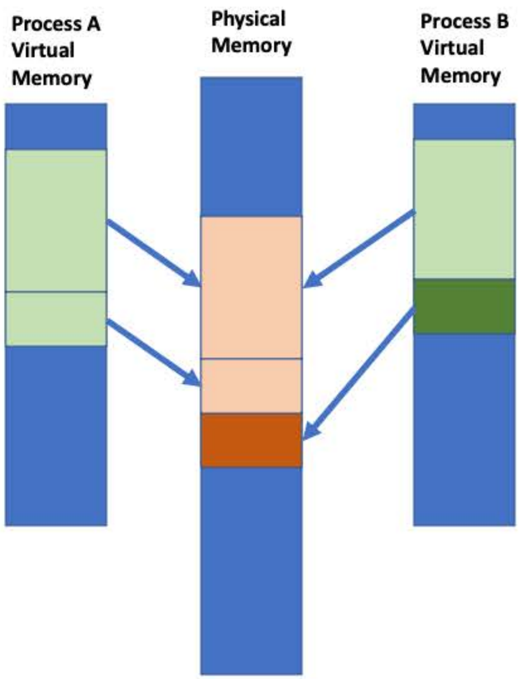
- Copy-on-write
- A memory efficient parallelism on shared memory devices
- **parallel** package `mclapply` and friends
- Use for numerical sections only
  - Avoid GUI, I/O, and graphics sections
- Convenient for data (not modified)
- Convenient for functional languages like R
- Avoid or manage nested parallelism
  - OpenBLAS takes all cores by default
  - data.table automatically switches to single threaded mode upon fork

A deeper discussion of `fork` memory (if you have interest) on YouTube by Chris Kanich (UIC)

Copy-on-write



Shared memory after forking Process B



After Process B writes to copy-on-write shared memory page

Mapping Threads to Cores

Theory and Reality

- Operating system manages core affinity
- Operating system tasks can compete
- Core switching occurs frequently
- **But it works rather well!**

Drop-in replacements (almost) for `lapply`, `mapply`, and `Map`

```
mclapply(X, FUN, ..., mc.preschedule = TRUE, mc.set.seed =
TRUE, mc.silent = FALSE, mc.cores = getOption("mc.cores", 2L),
mc.cleanup = TRUE, mc.allow.recursive = TRUE, affinity.list =
NULL)
```

```
mcmapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES
= TRUE, mc.preschedule = TRUE, mc.set.seed = TRUE, mc.silent =
FALSE, mc.cores = getOption("mc.cores", 2L), mc.cleanup =
TRUE, affinity.list = NULL)
```

```
mcMap(f, ...)
```

Example Random forest Code

Letter recognition data ($20\,000 \times 17$)

```
[,1] lettr capital letter
[,2] x.box horizontal position of box
[,3] y.box vertical position of box
[,4] width width of box
[,5] high height of box
[,6] onpix total number of on pixels
[,7] x.bar mean x of on pixels in box
[,8] y.bar mean y of on pixels in box
[,9] x2bar mean x variance
[,10] y2bar mean y variance
[,11] xybar mean x y correlation
[,12] x2ybr mean of x^2 y
[,13] xy2br mean of x y^2
[,14] x.ege mean edge count left to right
[,15] xegvy correlation of x.ege with y
[,16] y.ege mean edge count bottom to top
[,17] yegvx correlation of y.ege with x
```

Figure 1: Letter Recognition data (image: [Frey and Slate, 1991], description: **mlbench** package).

*Parallel Statistical Computing with R: An Illustration on Two Architectures
arXiv:1709.01195

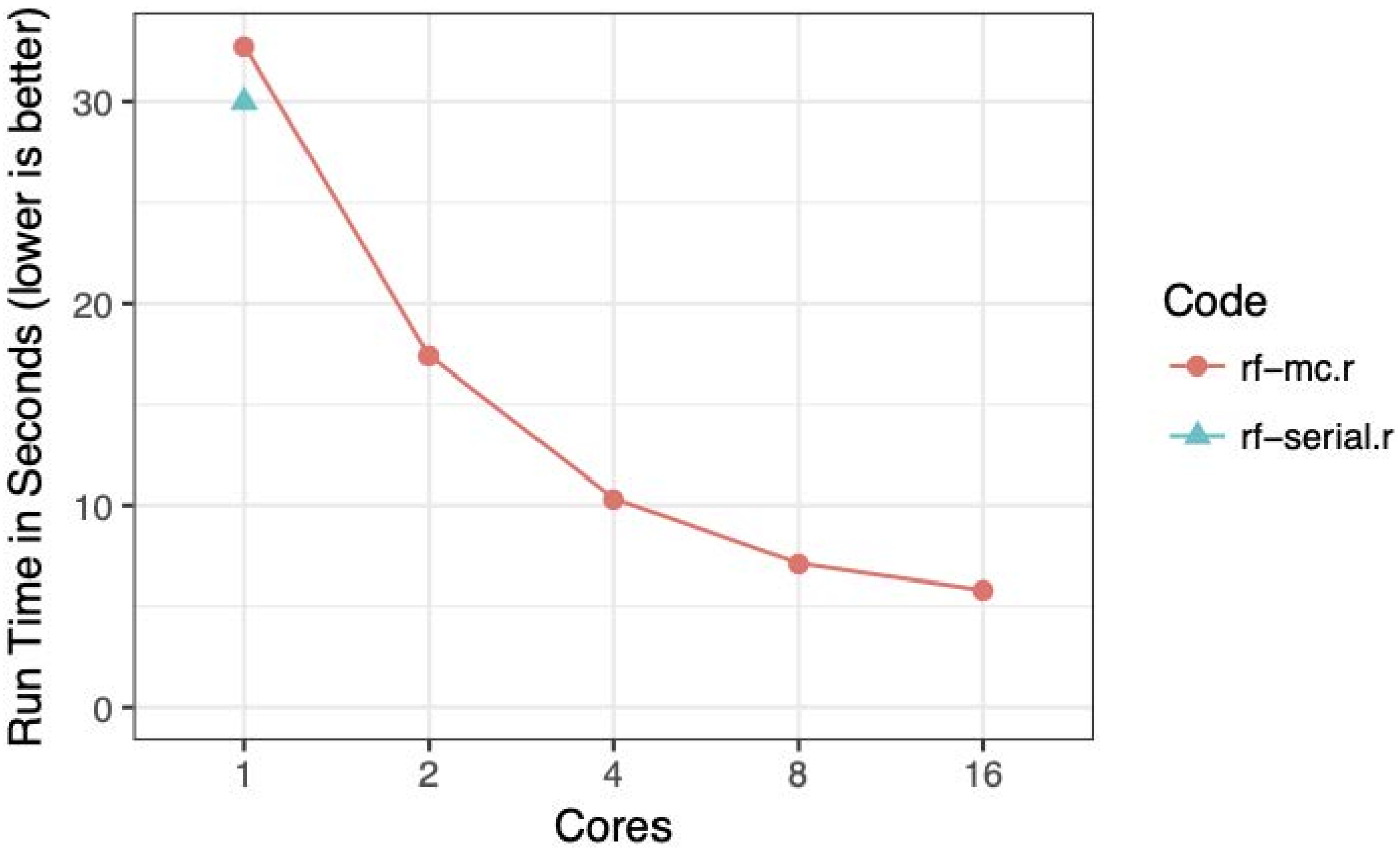
Random Forest Classification

Build many decision trees from random subsets of variables

Use their majority votes to classify

Example Random Forest Classification Code

Letter recognition data ($20\,000 \times 17$)



KPMS-IT4I-EX/code/rf_serial.r

```
library(randomForest)
data(LetterRecognition, package = "mlbench")
set.seed(seed = 123)

n = nrow(LetterRecognition)
n_test = floor(0.2 * n)
i_test = sample.int(n, n_test)
train = LetterRecognition[-i_test, ]
test = LetterRecognition[i_test, ]

rf.all = randomForest(lettr ~ ., train, ntree = 500, norm.votes = FAL
pred = predict(rf.all, test)

correct = sum(pred == test$lettr)
cat("Proportion Correct:", correct/(n_test), "\n")
```

KPMS-IT4I-EX/code/rf_mc.r

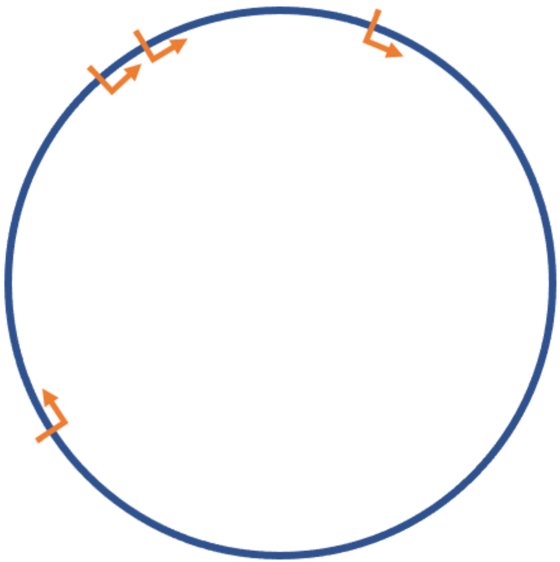
```r
library(parallel)
library(randomForest)
data(LetterRecognition, package = "mlbench")
set.seed(seed = 123, "L'Ecuyer-CMRG")

n = nrow(LetterRecognition)
n_test = floor(0.2 * n)
i_test = sample.int(n, n_test)
train = LetterRecognition[-i_test, ]
test = LetterRecognition[i_test, ]

nc = as.numeric(commandArgs(TRUE)[1])
ntree = lapply(splitIndices(500, nc), length)
rf = function(x) randomForest(lettr ~ ., train, ntree=x, norm.votes =
rf.out = mclapply(ntree, rf, mc.cores = nc)
rf.all = do.call(combine, rf.out)

crows = splitIndices(nrow(test), nc)
rfp = function(x) as.vector(predict(rf.all, test[x, ]))
cpred = mclapply(crows, rfp, mc.cores = nc)
pred = do.call(c, cpred)

correct <- sum(pred == test$lettr)
cat("Proportion Correct:", correct/(n_test), "\n")
```

Pseudo Random Number Generators (RNG)

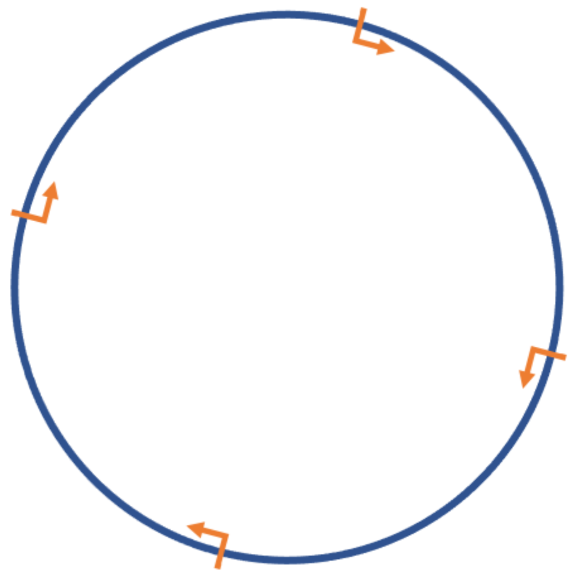
- Guaranteed reproducibility
- Possibly overlapping streams

- Reproducibility for same number of streams
- Guaranteed independent streams

Random seeds with serial RNG



Parallel RNG



KPMS-IT4I-EX/code/rf_mc.r

```r
library(parallel)
library(randomForest)
data(LetterRecognition, package = "mlbench")
set.seed(seed = 123, "L'Ecuyer-CMRG")

n = nrow(LetterRecognition)
n_test = floor(0.2 * n)
i_test = sample.int(n, n_test)
train = LetterRecognition[-i_test, ]
test = LetterRecognition[i_test, ]

nc = as.numeric(commandArgs(TRUE)[1])
ntree = lapply(splitIndices(500, nc), length)
rf = function(x) randomForest(lettr ~ ., train, ntree=x, norm.votes =
rf.out = mclapply(ntree, rf, mc.cores = nc)
rf.all = do.call(combine, rf.out)

crows = splitIndices(nrow(test), nc)
rfp = function(x) as.vector(predict(rf.all, test[x, ]))
cpred = mclapply(crows, rfp, mc.cores = nc)
pred = do.call(c, cpred)

correct <- sum(pred == test$lettr)
cat("Proportion Correct:", correct/(n_test), "\n")
```

KPMS-IT4I-EX/code/rf_karolina_pbs.sh

```bash
#!/bin/bash
#PBS -N rf
#PBS -l select=1:ncpus=128,walltime=00:05:00
#PBS -q qexp
#PBS -e rf.e
#PBS -o rf.o

cd ~/KPMS-IT4I-EX/code
pwd

module load R
echo "loaded R"

time Rscript rf_serial.r
time Rscript rf_mc.r 1
time Rscript rf_mc.r 2
time Rscript rf_mc.r 4
```

Demo ...