A Brief Introduction to Using R for High-Performance Computing

George G. Vega Yon
vegayon@usc.edu

University of Southern California
Department of Preventive Medicine

August 27th, 2019

High-Performance Computing: An overview

Loosely, from R's perspective, we can think of HPC in terms of two, maybe three things:

1. Big data: How to work with data that doesn't fit your computer
2. Parallel computing: How to take advantage of multiple core systems
3. Compiled code: Write your own low-level code (if R doesn't has it yet…)

(Checkout CRAN Task View on HPC)

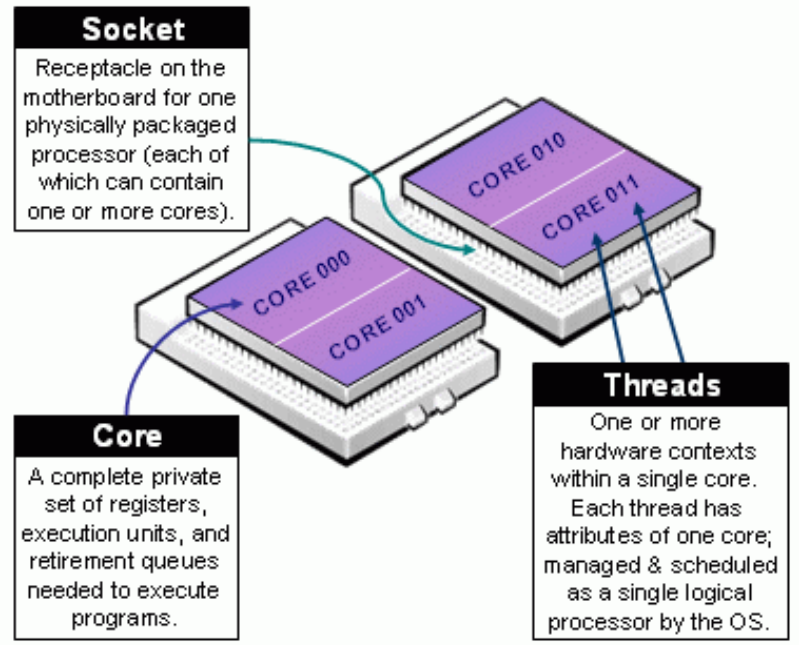
Some vocabulary for HPC

In raw terms

- Supercomputer: A **single** big machine with thousands of cores/gpus.
- High Performance Computing (HPC): **Multiple** machines within a **single** network.
- High Throughput Computing (HTC): **Multiple** machines across **multiple** networks.

You may not have access to a supercomputer, but certainly HPC/HTC clusters are more accesible these days, e.g. AWS provides a service to create HPC clusters at a low cost (allegedly, since nobody understands how pricing works)

What's "a core"?



Taxonomy of CPUs (Downloaded from de https://slurm.schedmd.com/mc_support.html)

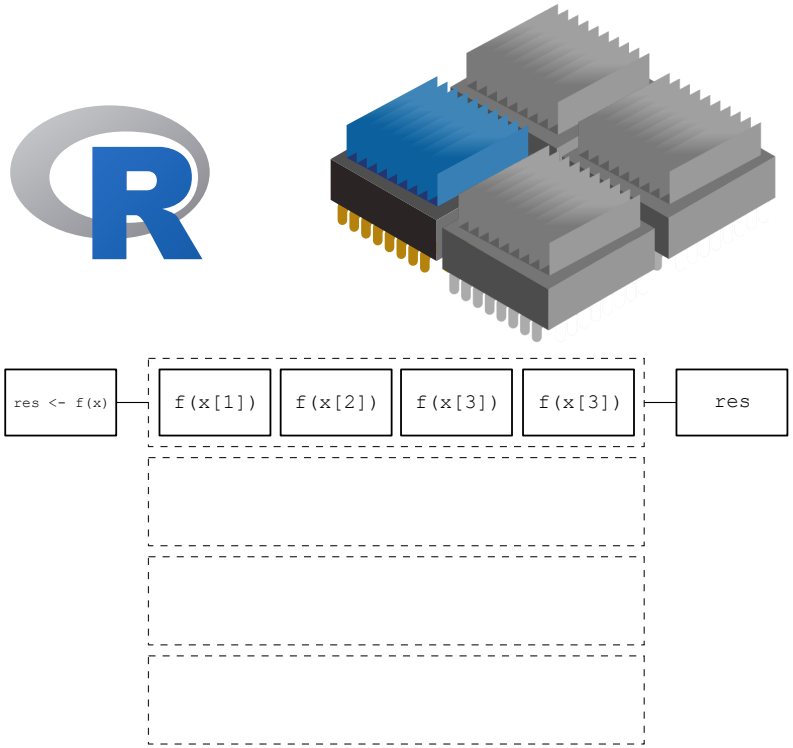
Now, how many cores does your computer has, the parallel package can tell you that:

```
parallel::detectCores()
```

```
## [1] 4
```

What is parallel computing, anyway?

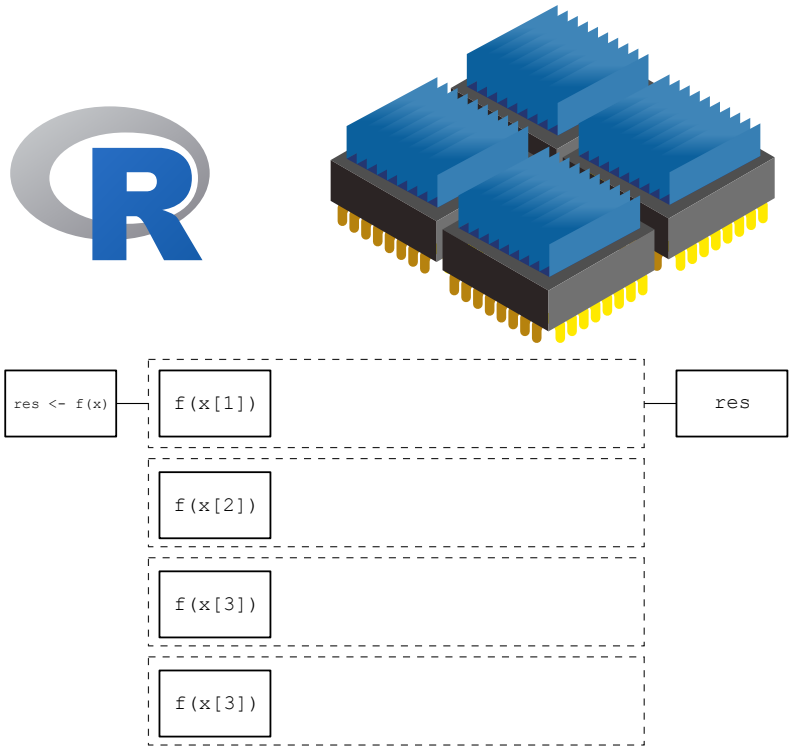
```
f <- function(n) n*2
f(1:4)
```



Here we are using a single core. The function is applied one element at a time, leaving the other 3 cores without usage.

What is parallel computing, anyway? (cont'd)

```r
f <- function(n) n*2
f(1:4)
```

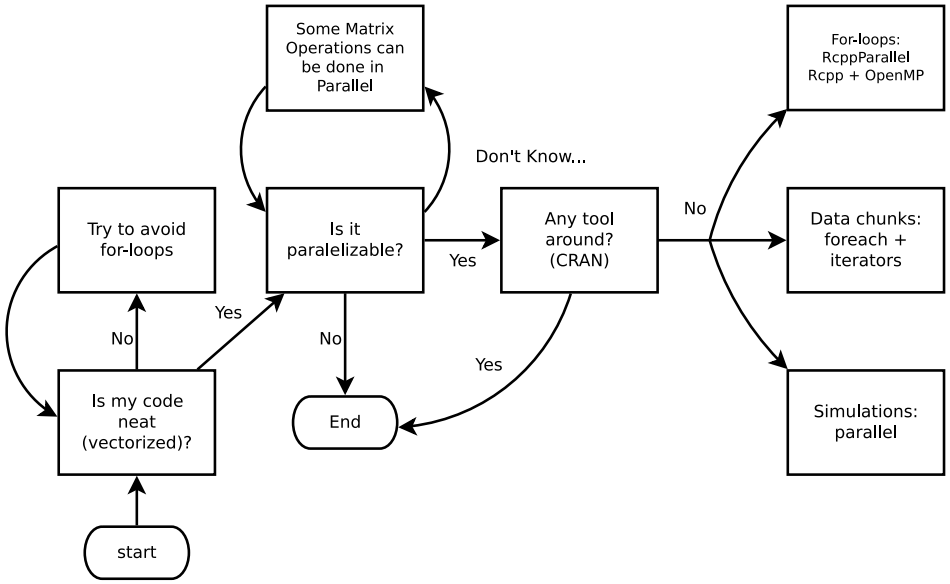


In this more intelligent way of computation, we are taking full advantage of our computer by using all 4 cores at the same time. This will translate in a reduced computation time which, in the case of complicated/long calculations, can be an important speed gain.

Let’s think before we start…

When is it a good idea to go HPC?

When is it a good idea?



Ask yourself these questions before jumping into HPC!

Parallel computing in R

While there are several alternatives (just take a look at the High-Performance Computing Task View), we'll focus on the following R-packages for **explicit parallelism**

Some examples:

- **parallel**: R package that provides '[s]upport for parallel computation, including random-number generation'.
- **foreach**: R package for 'general iteration over elements' in parallel fashion.
- **future**: '[A] lightweight and unified Future API for sequential and parallel processing of R expression via futures.' (won't cover here)

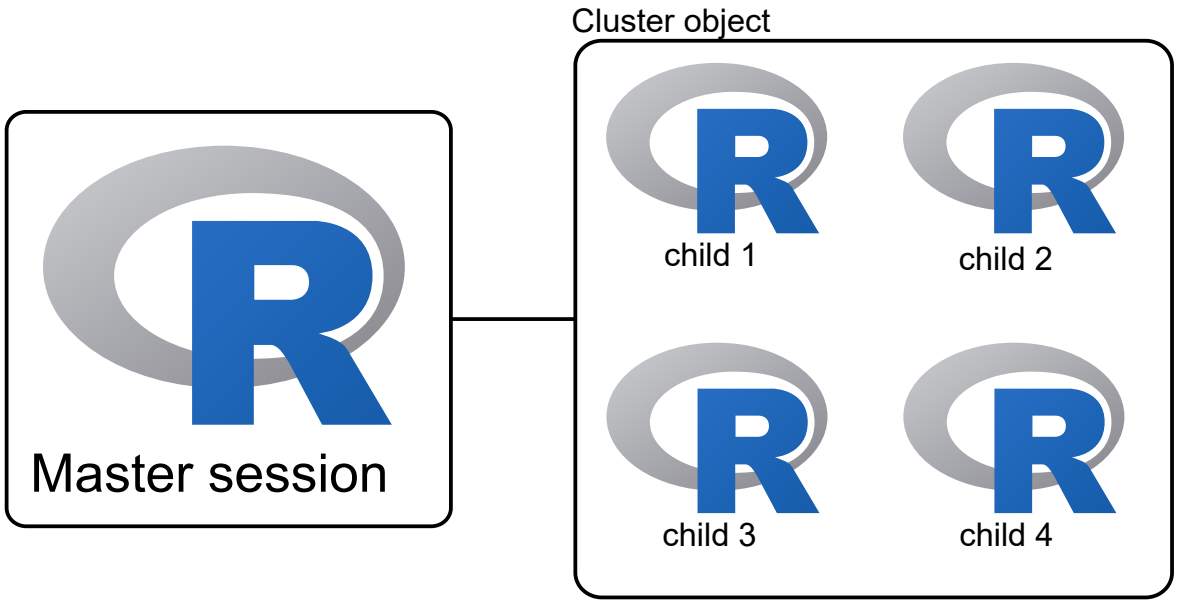
Implicit parallelism, on the other hand, are out-of-the-box tools that allow the programmer not to worry about parallelization, e.g. such as **gpuR** for Matrix manipulation using GPU, **tensorflow**

And there's also a more advanced set of options

- **Rcpp** + OpenMP: Rcpp is an R package for integrating R with C++, and OpenMP is a library for high-level parallelism for C/C++ and Fortran.
- A ton of other type of resources, notably the tools for working with batch schedulers such as Slurm, HTCondor, etc.

The parallel package

- Based on the `snow` and `multicore` R Packages.
- Explicit parallelism.
- Simple yet powerful idea: Parallel computing as multiple R sessions.
- Clusters can be made of both local and remote sessions
- Multiple types of cluster: `PSOCK`, `Fork`, `MPI`, etc.



Parallel workflow

(Usually) We do the following:

1. Create a PSOCK/FORK (or other) cluster using `makePSOCKCluster`/`makeForkCluster` (or `makeCluster`)
2. Copy/prepare each R session (if you are using a PSOCK cluster):
    1. *Copy objects with `clusterExport`*
    2. *Pass expressions with `clusterEvalQ`*
    3. *Set a seed*
3. Do your call: `parApply`, `parLapply`, etc.
4. Stop the cluster with `clusterStop`

Ex 1: Hello world!

```r
# 1. CREATING A CLUSTER
library(parallel)
cl <- makePSOCKcluster(4)
x  <- 20

# 2. PREPARING THE CLUSTER
clusterSetRNGStream(cl, 123) # Equivalent to `set.seed(123)`
clusterExport(cl, "x")

# 3. DO YOUR CALL
clusterEvalQ(cl, {
  paste0("Hello from process #", Sys.getpid(), ". I see x and it is equal to ", x)
})
```

```
## [[1]]
## [1] "Hello from process #4035. I see x and it is equal to 20"
##
## [[2]]
## [1] "Hello from process #4050. I see x and it is equal to 20"
##
## [[3]]
## [1] "Hello from process #4064. I see x and it is equal to 20"
##
## [[4]]
## [1] "Hello from process #4093. I see x and it is equal to 20"
```

```r
# 4. STOP THE CLUSTER
stopCluster(cl)
```

Ex 2: Parallel regressions

Problem: Run multiple regressions on a very wide dataset. We need to fit the following model:

$$y = X_i\beta_i + \varepsilon, \quad \varepsilon \sim N(0, \sigma_i^2), \quad \forall i$$

```
dim(X)
```

```
## [1] 500 999
```

```
X[1:6, 1:5]
```

```
##          x001        x002       x003       x004      x005
## 1  0.61827227  1.72847041 -1.4810695 -0.2471871 1.4776281
## 2  0.96777456 -0.19358426 -0.8176465  0.6356714 0.7292221
## 3 -0.04303734 -0.06692844  0.9048826 -1.9277964 2.2947675
## 4  0.84237608 -1.13685605 -1.8559158  0.4687967 0.9881953
## 5 -1.91921443  1.83865873  0.5937039 -0.1410556 0.6507415
## 6  0.59146153  0.81743419  0.3348553 -1.8771819 0.8181764
```

```
str(y)
```

```
##  num [1:500] -0.8188 -0.5438 1.0209 0.0467 -0.4501 ...
```

Ex 2: Parallel regressions (cont'd 1)

Serial solution: Use `apply` (forloop) to solve it

```
ans <- apply(

  X      = X,
  MARGIN = 2,
  FUN    = function(x, y) coef(lm(y ~ x)),
  y      = y
  )

ans[,1:5]
```

```
##                    x001        x002        x003        x004        x005
## (Intercept) -0.03449819 -0.03339681 -0.03728140 -0.03644192 -0.03717344
## x           -0.06082548  0.02748265 -0.01327855 -0.08012361 -0.04067826
```

Ex 2: Parallel regressions (cont'd 2)

Parallel solution: Use `parApply`

```
library(parallel)
cl <- makePSOCKcluster(4L)
ans <- parApply(
  cl     = cl,
  X      = X,
  MARGIN = 2,
  FUN    = function(x, y) coef(lm(y ~ x)),
  y      = y
  )

ans[,1:5]
```

```
##                    x001        x002        x003        x004        x005
## (Intercept) -0.03449819 -0.03339681 -0.03728140 -0.03644192 -0.03717344
## x           -0.06082548  0.02748265 -0.01327855 -0.08012361 -0.04067826
```

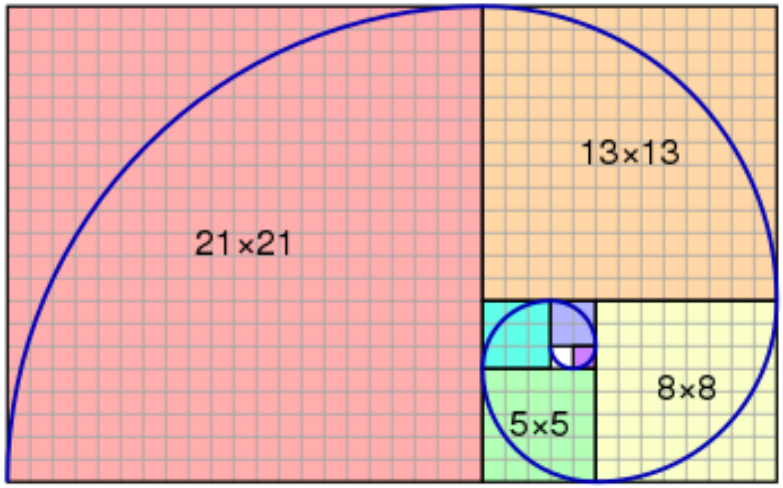
Are we going any faster?

```
library(bench)
mark(
  parallel = parApply(
    cl  = cl,
    X   = X, MARGIN = 2,
    FUN = function(x, y) coef(lm(y ~ x)),
    y   = y
    ),
  serial = apply(
    X   = X, MARGIN = 2,
    FUN = function(x, y) coef(lm(y ~ x)),
    y   = y
    )
)
```

```
## # A tibble: 2 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 parallel      374ms    410ms      2.44    11.8MB      0
## 2 serial        677ms    677ms      1.48    94.9MB     13.3
```

Rcpp: Hello world!

The Fibonacci series



$$Fib(n) = \begin{cases} n & \text{if } n \leq 1 \\ Fib(n-1) + Fib(n-2) & \text{otherwise} \end{cases}$$

Rcpp: Hello world! vers I

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
int fibCpp(int n) {

  if (n < 2) {
    return n;
  }

  return fibCpp(n - 1) + fibCpp(n - 2);

}
```

Back in R

```
c(fibCpp(1), fibCpp(2), fibCpp(3), fibCpp(4), fibCpp(5))
```

```
## [1] 1 1 2 3 5
```

Rcpp: Hello world! vers2 (with function overloading)

```
#include <Rcpp.h>

using namespace Rcpp;

// inline kind of implementation
int fibCpp(int n) {return (n < 2)? n : fibCpp(n - 1) + fibCpp(n - 2);}

// [[Rcpp::export]]
IntegerVector fibCpp(IntegerVector n) {

  IntegerVector res(n.size());
  for (int i = 0; i < n.size(); ++i)
    res[i] = fibCpp(n[i]);

  return res;

}
```

Back in R

```
fibCpp(1:5)
```

```
## [1] 1 1 2 3 5
```

RcppArmadillo and OpenMP

- Friendlier than **RcppParallel**… at least for 'I-use-Rcpp-but-don't-actually-know-much-about-C++' users (like myself!).
- Must run only 'Thread-safe' calls, so calling R within parallel blocks can cause problems (almost all the time).
- Use `arma` objects, e.g. `arma::mat`, `arma::vec`, etc. Or, if you are used to them `std::vector` objects as these are thread safe.
- Pseudo-Random Number Generation is not very straight forward… But C++11 has a nice set of functions that can be used together with OpenMP
- Need to think about how processors work, cache memory, etc. Otherwise you could get into trouble… if your code is slower when run in parallel, then you probably are facing false sharing
- If R crashes… try running R with a debugger (see Section 4.3 in Writing R extensions):

```
~$ R --debugger=valgrind
```

RcppArmadillo and OpenMP workflow

1. Add the following to your C++ source code to use OpenMP, and tell Rcpp that you need to include that in the compiler:

```
#include <omp.h>
// [[Rcpp::plugins(openmp)]]
```

2. Tell the compiler that you'll be running a block in parallel with openmp

```
#pragma omp [directives] [options]
{
  ...your neat parallel code...
}
```

You'll need to specify how OMP should handle the data:

- `shared`: *Default, all threads access the same copy.*
- `private`: *Each thread has its own copy (although not initialized).*
- `firstprivate` *Each thread has its own copy initialized.*
- `lastprivate` *Each thread has its own copy. The last value is the one stored in the main program.*

Setting `default(none)` is a good practice.

3. Compile!

Ex 3: RcppArmadillo + OpenMP

Computing the distance matrix (see `?dist`)

```
#include <omp.h>
#include <RcppArmadillo.h>

// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::plugins(openmp)]]

using namespace Rcpp;

// [[Rcpp::export]]
arma::mat dist_par(const arma::mat & X, int cores = 1) {

  // Some constants
  int N = (int) X.n_rows;
  int K = (int) X.n_cols;

  // Output
  arma::mat D(N,N);
  D.zeros(); // Filling with zeros

  // Setting the cores
  omp_set_num_threads(cores);

#pragma omp parallel for shared(D, N, K, X) default(none)
  for (int i=0; i<N; ++i)
    for (int j=0; j<i; ++j) {
      for (int k=0; k<K; k++)
        D.at(i,j) += pow(X.at(i,k) - X.at(j,k), 2.0);

      // Computing square root
      D.at(i,j) = sqrt(D.at(i,j));
      D.at(j,i) = D.at(i,j);
    }
```

```
  // My nice distance matrix
  return D;
}
```

```r
set.seed(1231)
K <- 1000
n <- 500
x <- matrix(rnorm(n*K), ncol=K)
```

```r
# Benchmarking!
rbenchmark::benchmark(
  dist(x),                # stats::dist
  dist_par(x, cores = 1),  # 1 core
  dist_par(x, cores = 2),  # 2 cores
  dist_par(x, cores = 4), #  4 cores
  replications = 10, order="elapsed"
)[,1:4]
```

```
##                     test replications elapsed relative
## 3 dist_par(x, cores = 2)           10   2.558    1.000
## 4 dist_par(x, cores = 4)           10   2.962    1.158
## 2 dist_par(x, cores = 1)           10   3.428    1.340
## 1                dist(x)           10   4.986    1.949
```

Thanks!

gvegayon
@gvegayon
ggvy.cl

Presentation created with rmarkdown::slidy_presentation

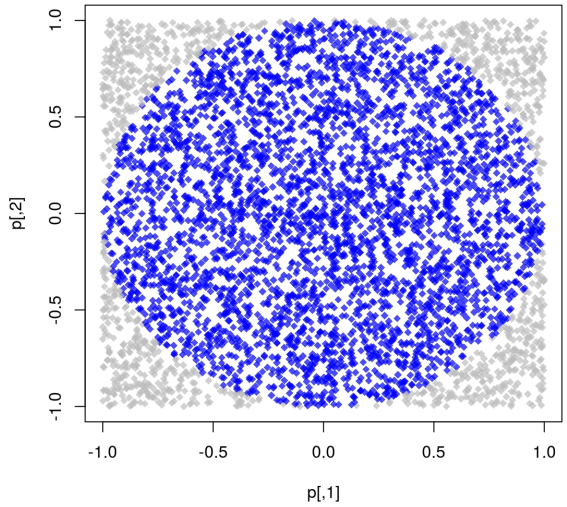
See also

- Package parallel
- Using the iterators package
- Using the foreach package
- 32 OpenMP traps for C++ developers
- The OpenMP API specification for parallel programming
- 'openmp' tag in Rcpp gallery
- OpenMP tutorials and articles

For more, checkout the CRAN Task View on HPC

Bonus track: Simulating π

- We know that $\pi = \frac{A}{r^2}$. We approximate it by randomly adding points x to a square of size 2 centered at the origin.
- So, we approximate π as $\Pr\{\|x\| \leq 1\} \times 2^2$



The R code to do this

```r
pisim <- function(i, nsim) {  # Notice we don't use the -i-
  # Random points
  ans  <- matrix(runif(nsim*2), ncol=2)

  # Distance to the origin
  ans  <- sqrt(rowSums(ans^2))

  # Estimated pi
  (sum(ans <= 1)*4)/nsim
}
```

```r
library(parallel)
# Setup
cl <- makePSOCKcluster(4L)
clusterSetRNGStream(cl, 123)

# Number of simulations we want each time to run
nsim <- 1e5

# We need to make -nsim- and -pisim- available to the
# cluster
clusterExport(cl, c("nsim", "pisim"))

# Benchmarking: parSapply and sapply will run this simulation
# a hundred times each, so at the end we have 1e5*100 points
# to approximate pi
rbenchmark::benchmark(
  parallel = parSapply(cl, 1:100, pisim, nsim=nsim),
  serial   = sapply(1:100, pisim, nsim=nsim), replications = 1
)[,1:4]
```

```
##       test replications elapsed relative
## 1 parallel            1   0.428     1.00
## 2   serial            1   0.796     1.86
```

Session info

```
## R version 3.6.1 (2019-07-05)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 16.04.6 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/libblas/libblas.so.3.6.0
## LAPACK: /usr/lib/lapack/liblapack.so.3.6.0
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
## [1] bench_1.0.2   future_1.14.0
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.2       rbenchmark_1.0.0 codetools_0.2-16 listenv_0.7.0
##  [5] digest_0.6.20    magrittr_1.5     evaluate_0.14    highr_0.8
##  [9] icon_0.1.0       stringi_1.4.3    rmarkdown_1.14   tools_3.6.1
## [13] stringr_1.4.0    xfun_0.9         yaml_2.2.0       compiler_3.6.1
## [17] globals_0.12.4   htmltools_0.3.6  knitr_1.24
```