High Performance Computing, Cloud Computing

JSC 370: Data Science II

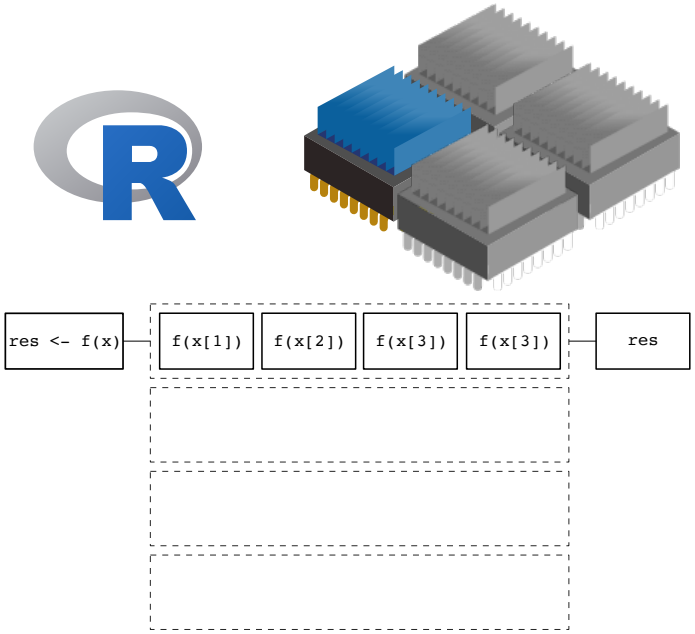
What is HPC

High Performance Computing (HPC) can relate to any of the following:

- **Parallel computing**, i.e. using multiple resources (could be threads, cores, nodes, etc.) simultaneously to complete a task.
- **Big data** working with large datasets (in/out-of-memory).

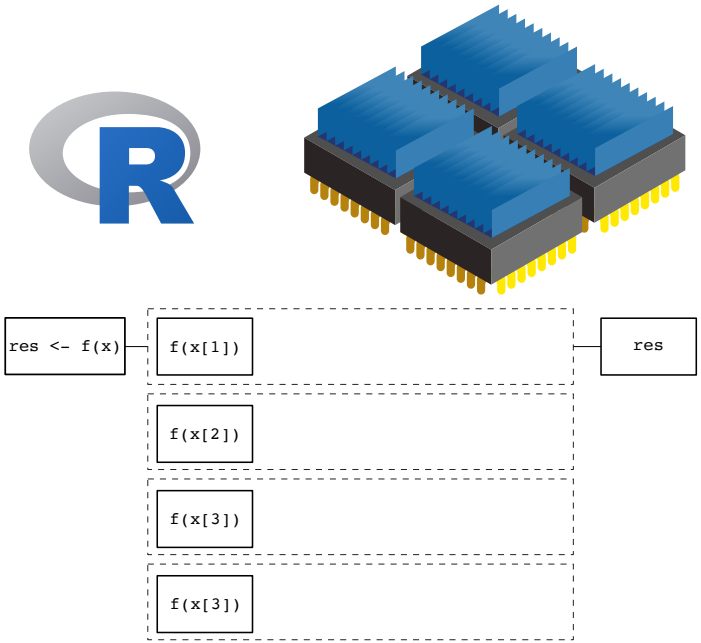
We will mostly focus on parallel computing.

Serial computation



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

Parallel computation



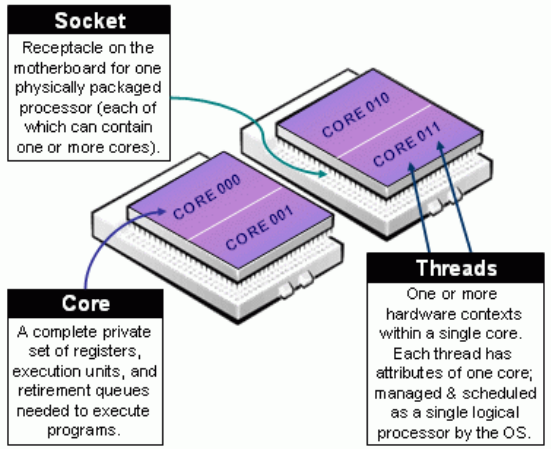
In this more intelligent way of computation, we are taking full advantage of our computer by using all 4 cores at the same time. This will translate in a reduced computation time which, in the case of complicated/long calculations, can be an important speed gain.

Parallel computing: Hardware

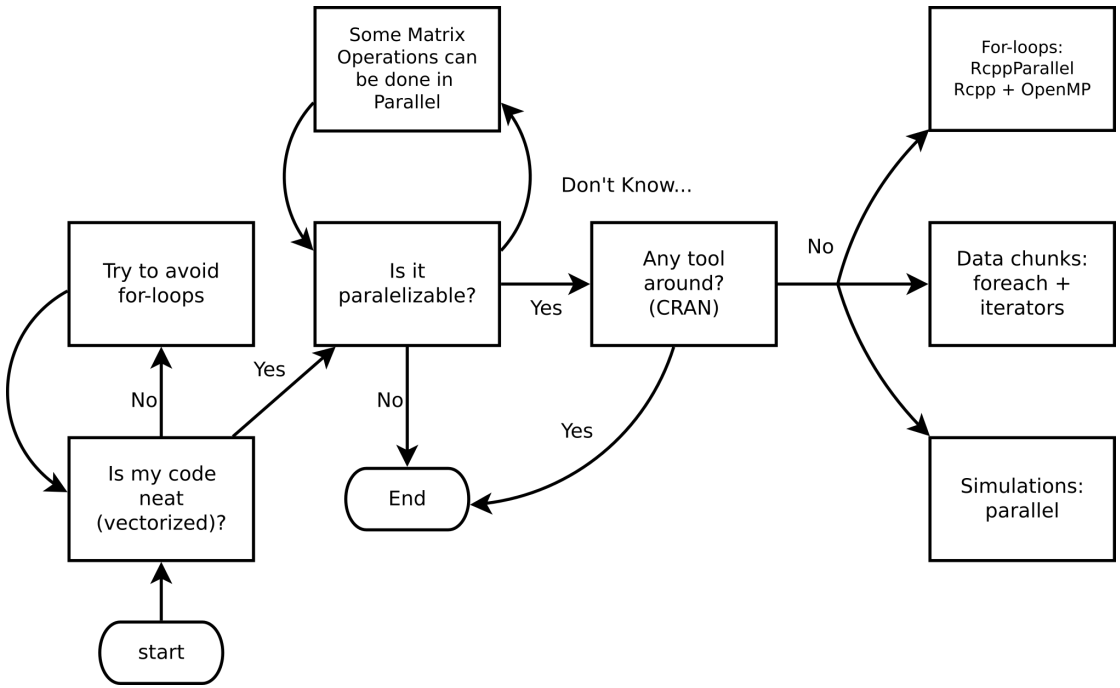
When it comes to parallel computing, there are several ways (levels) in which we can speed up our analysis. From the bottom up:

- **Thread level SIMD instructions**: In most modern processors support some level of what is called vectorization, this is, applying a single (same) instruction to streams of data, for example: adding vector `A` and `B`.
- **Hyper-Threading Technology** (HTT): Intel's hyper-threading generates a virtual partition of a single core (processor) which, while not equivalent to having multiple physical threads, does speedup things.
- **Multi-core processor**: Most modern CPUs (Central Processing Unit) have two or more physical cores. A typical laptop computer has about 8 cores.
- **General-Purpose Computing on Graphics Processing Unit** (GP-GPU): While modern CPUs have a couple of dozens of cores, GPUs can hold thousands of those. Designed for image processing, there's an increasing use of GPUs as an alternative of CPUs for scientific computing.
- **High-Performance Computing Cluster** (HPC): A collection of computing nodes that are interconnected using a fast Ethernet network.
- **Grid Computing**: A collection of loosely interconnected machines that may or may not be in the same physical place, for example: HTCondor clusters.

Parallel computing: CPU components



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



Ask yourself these questions before jumping into HPC!

Parallel computing in R

While there are several ways to do parallel computing in R (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)

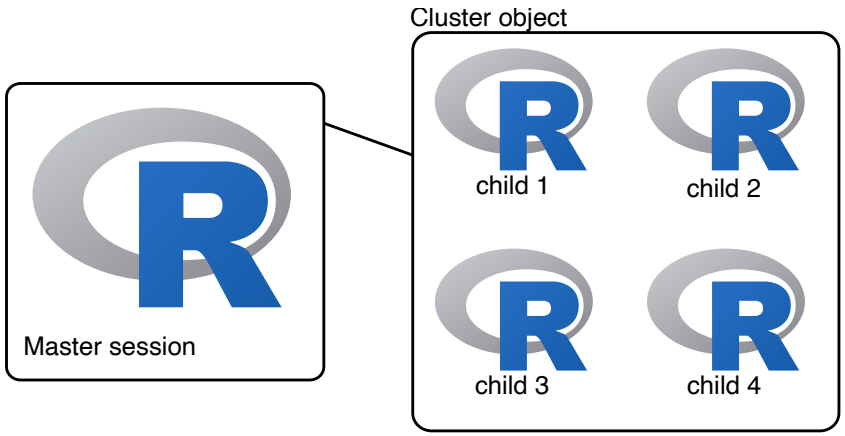
Implicit parallelism, on the other hand, are out-of-the-box tools that allow the programmer not to worry about parallelization, e.g. such as **gpuR** for Matrix manipulation using GPU, **tensorflow**

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):

   a. Copy objects with `clusterExport`

   b. Pass expressions with `clusterEvalQ`

   c. Set a seed
3. Do your call: `parApply`, `parLapply`, etc.
4. Stop the cluster with `clusterStop`

Ex 1: Hello world!

```
# 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 #38185. I see x and it is equal to 20"
##
## [[2]]
## [1] "Hello from process #38187. I see x and it is equal to 20"
##
## [[3]]
## [1] "Hello from process #38186. I see x and it is equal to 20"
##
## [[4]]
## [1] "Hello from process #38184. I see x and it is equal to 20"
```

```
# 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) coef(lm(y ~ x))
  )

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)
clusterExport(cl, "y")
ans <- parApply(
  cl     = cl,
  X      = X,
  MARGIN = 2,
  FUN    = function(x) coef(lm(y ~ x))
  )

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?

```
microbenchmark::microbenchmark(
  parallel = parApply(
    cl  = cl,
    X   = X, MARGIN = 2,
    FUN = function(x) coef(lm(y ~ x))
    ),
  serial = apply(
    X   = X, MARGIN = 2,
    FUN = function(x) coef(lm(y ~ x))
    ), unit="ms"
)
```

```
## Unit: milliseconds
##       expr      min       lq     mean   median       uq      max neval
##  parallel 189.3619 211.8326 226.1504 221.3169 230.7429 345.5169   100
##    serial 553.7184 586.4854 604.3711 598.8101 609.3490 776.1760   100
```

Extended Example: SARS-CoV2 simulation

An altered version of Conway's game of life

1. People live in torus, each individual having 8 neighbors.
2. A healthy individual interacting with a sick neighbor has the following probabilities of contracting the disease:

    a. 100% if neither wears a face-mask.

    b. 50% if only he wears the face-mask.

    c. 20% if only his neighbor wears the mask.

    d. 5% if both wear the face-mask.
3. Infected individuals may die with probability 10%.

We want to illustrate the importance of wearing face masks. We need to simulate a system with 2,500 (50 x 50) individuals, 1,000 times so we can analyze: (a) contagion curve, (b) death curve.

More models like this: The SIRD model (Susceptible-Infected-Recovered-Deceased)

Conway's Game of Masks

Download the program here.

```
source("sars-cov2.R", echo=FALSE)

# Looking at some constants
probs_sick # Sick individual's probabilities
```

```
##  deceased  infected recovered
##       0.1       0.4       0.5
```

```
probs_susc # Probabilities of i getting the disease
```

```
##                 j doesn't wear j wears
## i doesn't wear             0.9    0.20
## i wears                    0.5    0.05
```

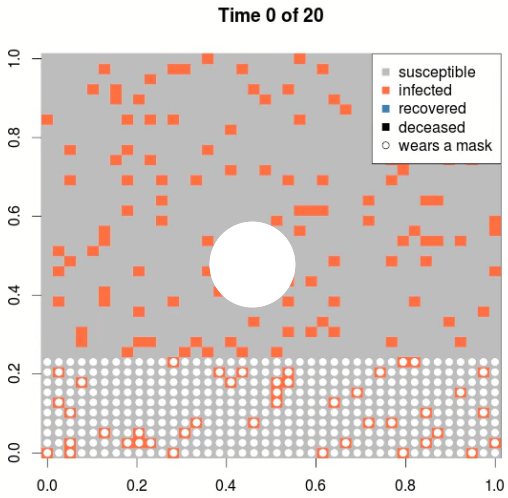
First look: How does the simulation looks like?

```
set.seed(7123)
one <- simulate_covid(
    pop_size    = 1600,
    nsick       = 160,
    nwears_mask = 1:400,
    nsteps      = 20,
    store       = TRUE
    )

one$statistics[c(1:5, 16:20),]
```

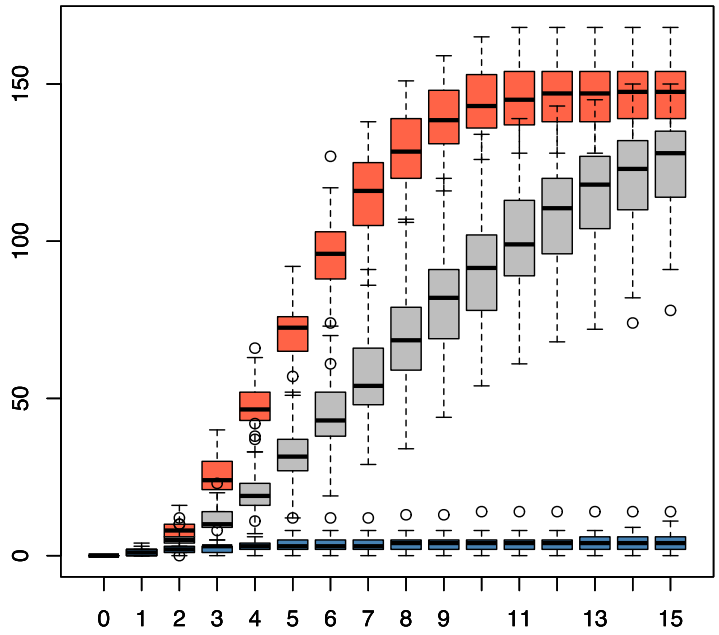
```
##    susceptible infected recovered deceased
## 0         1440      160         0        0
## 1         1265      234        85       16
## 2         1064      307       190       39
## 3          876      321       334       69
## 4          717      287       499       97
## 15         430        1       990      179
## 16         429        2       990      179
## 17         429        0       992      179
## 18         429        0       992      179
## 19         429        0       992      179
```

First look: How does the simulation looks like? (contd')

```
# Location of who wears the facemask. This step is only for plotting
wears <- which(one$wears, arr.ind = TRUE) - 1
wears <- wears/(one$nr) * (1 + 1/one$nr)

# Initializing the animation
fig   <- magick::image_device(600, 600, res = 96/2, pointsize = 24)
for (i in 1:one$current_step) {

  # Plot
  image(
    one$temporal[,,i], col=c("gray", "tomato", "steelblue","black"),
    main = paste("Time", i - 1L, "of", one$nsteps),
    zlim = c(1,4)
    )
  points(wears, col="white", pch=20, cex=1.5)
  legend(
    "topright",
    col    = c("gray", "tomato", "steelblue","black", "black"),
    legend = c(names(codes), "wears a mask"),
    pch    = c(rep(15, 4), 21)
  )
}

# Finalizing plot and writing the animation
dev.off()
animation <- magick::image_animate(fig, fps = 2)
magick::image_write(animation, "covid1.gif")
```


Time 0 of 20
susceptible
infected
recovered
deceased
wears a mask
0.0
0.2
0.4
0.6
0.8
1.0

```r
set.seed(123355)
stats_nobody_wears_masks <- replicate(50, {
  simulate_covid(
    pop_size    = 900,
    nsick       = 10,
    nwears_mask = 0,
    nsteps      = 15)$statistics[,"deceased"]
  }, simplify = FALSE
  )

set.seed(123355)
stats_half_wears_masks <- replicate(50, {
  simulate_covid(
    pop_size    = 900,
    nsick       = 10,
    nwears_mask = 450,
    nsteps      = 15)$statistics[,"deceased"]
  }, simplify = FALSE
  )

set.seed(123355)
stats_all_wears_masks <- replicate(50, {
  simulate_covid(
    pop_size    = 900,
    nsick       = 10,
    nwears_mask = 900,
    nsteps      = 15)$statistics[,"deceased"]
  }, simplify = FALSE
  )
```



Cumulative number of deceased as a function of whether none, half, or all individuals wear a face mask.

Speed things up: Timing under the serial implementation

We will use the function `system.time` to measure how much time it takes to complete 20 simulations in serial versus paralle fashion using 4 cores

```
time_serial <- system.time({
  ans_serial <- replicate(50, {
    simulate_covid(
      pop_size    = 900,
      nsick       = 10,
      nwears_mask = 900,
      nsteps      = 20)$statistics[,"deceased"]
    },
    simplify = FALSE
    )
})
```

Speed things up: Parallel a Forking Cluster

Alternative 1: If you are using Unix-like system (Ubuntu, OSX, etc.), you can take advantage of process forking, and thus, parallel's `mclapply` function:

```
set.seed(1231)
time_parallel_fork <- system.time({
  ans_parallel <- parallel::mclapply(1:50, function(i) {
    simulate_covid(
      pop_size    = 900,
      nsick       = 10,
      nwears_mask = 900,
      nsteps      = 20)$statistics[,"deceased"]
    }, mc.cores = 2L
  )
})
```

Speed things up: Parallel with a Socket Cluster

Alternative 2: Regardless of the operating system, we can use a Socket cluster, which is simply a group of fresh R sessions that listen to the parent/main/mother session.

```
# Step 1: Make the cluster
cl <- parallel::makePSOCKcluster(2L)
```

```
# Step 2: Prepare the cluster
# We could either export all the needed variables
parallel::clusterExport(
  cl,
  c("calc_stats", "codes", "dat", "get_neighbors", "init", "probs_sick",
    "probs_susc", "simulate_covid", "update_status", "update_status_all"
    )
  )
```

Or simply running the simulation script in the other sessions

```
# Step 2 (alt): Prepare the cluster
parallel::clusterEvalQ(cl, source("sars-cov2.R"))
parallel::clusterSetRNGStream(cl, 123) # Make sure it is reproducible!
```

```
(pids <- c(
  master    = Sys.getpid(),
  offspring = unlist(parallel::clusterEvalQ(cl, Sys.getpid()))
  ))
#      master offspring1 offspring2
#     14810      15998      16012
```

If you are using Unix, you can see more details:

```
UID        PID  PPID  C STIME TTY          TIME CMD
george   14810 10376  0 12:31 ?        00:00:09 /usr/lib/
george   15998     1  0 12:56 ?        00:00:00 /usr/lib/
george   16012     1  0 12:56 ?        00:00:00 /usr/lib/
```

Speed things up: Parallel with a Socket Cluster (cont'd)

```
# Step 3: Do your call
time_parallel_sock <- system.time({
  ans_parallel <- parallel::parLapply(cl, 1:50, function(i) {
    simulate_covid(
      pop_size    = 900,
      nsick       = 10,
      nwears_mask = 900,
      nsteps      = 20)$statistics[,"deceased"]
    }
  )
})

# Step 4: Stop
parallel::stopCluster(cl)
```

Using two threads/processes, you can obtain the following speedup

```
time_serial;time_parallel_sock;time_parallel_fork
```

```
##    user  system elapsed
##  20.514   0.238  21.166

##    user  system elapsed
##   0.001   0.000  11.585

##    user  system elapsed
##   0.003   0.004  11.925
```

Cloud Computing (a.k.a. on-demand computing)

HPC clusters, super-computers, etc. need not to be bought... you can rent:

- Amazon Web Services (AWS)
- Google Cloud Computing
- Microsoft Azure

These services provide more than just computing (storage, data analysis, etc.). But for computing and storage, there are other free resources, e.g.:

- The Extreme Science and Engineering Discovery Environment (XSEDE)

There are many ways to run R in the cloud

At USC:

- Center for Advanced Research Computing (CARC). USC users can request hundreds of cores (literally). Take a look at the slurmR package

Running R in:

- Google Cloud: https://cloud.google.com/solutions/running-r-at-scale
- Amazon Web Services: https://aws.amazon.com/blogs/big-data/running-r-on-aws/
- Microsoft Azure: https://docs.microsoft.com/en-us/azure/architecture/data-guide/technology-choices/r-developers-guide

Submitting jobs

- A key feature of cloud services > interact via command line.
- You will need to familiarize with `Rscript` and `R CMD BATCH`.
- Which is better? It depends on the application.

Submitting jobs (examples)

Imagine we have the following R script (download here):

```
library(data.table)
set.seed(1231)
dat <- data.table(y = rnorm(1e3), x = sample.int(5, 1e3, TRUE))
dat[,mean(y), by = x]
```

R CMD BATCH

This will run a non-interactive R session and put all the output (stdout and stderr) to the file `dummy.Rout`.

```
R CMD BATCH --vanilla dummy.R dummy.Rout &
```

Rscript

This will also execute R in the background, with the difference that the output `dummy.Rout` will not capture `stderr` (messages, warnings and errors from R).

```
Rscript --vanilla dummy.R > dummy.Rout &
```

The `&` at the end makes sure the job is submitted and does not wait for it to end. Try it yourself (5 mins)!

Rscript

The R script can be executed as program directly, if you specify where the `Rscript` program lives.

The following example works in Unix. This is an R script named `since_born.R` (download here)

```
#!/usr/bin/Rscript
args <- tail(commandArgs(), 0)
message(Sys.Date() - as.Date(args), " days since you were born.")
```

This R script, can be executed in various ways...

Rscript as a program

For this we would need to change it to an executable. In unix you can use the chmod command: `chmod +x since_born.R`. This allows to:

```
./since_born.R 1988-03-02
```

Rscript in a bash script (most common)

In the case of running jobs in a cluster or something similar, we usually need to have a bash script, In our case, here we have a file named `since_born_bash.sh` that calls `Rscript` (download here)

```
#!/bin/bash
Rscript since_born.R 1988-03-02
```

Which we would execute something like this

```
sh since_born_bash.sh
```

```
## 12419 days since you were born.
```

Summary

- Parallel computing can speed up things.
- Not always needed... need to make sure that you are taking advantage of vectorization.
- Most of the time we look at "Embarrassingly parallel computing."
- In R, explicit parallelism can be achieved using the **parallel** package:
    1. Load the package and create a cluster **parallel::makeCluster()**
    2. Setup the environment **parallel::clusterEvalQ()**, **parallel::clusterExport()**, and **parallel::clusterSetRNGStream()**
    3. Make the call, e.g., **parallel::parLapply()**
    4. Stop the cluster **parallel::stopCluster()**
- Regardless of the Cloud computing service we are using, we will be using either `R CMD BATCH` or `Rscript` to submit jobs.

Session info

```
## R version 4.1.2 (2021-11-01)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS Big Sur 10.16
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## loaded via a namespace (and not attached):
##  [1] digest_0.6.29   R6_2.5.1        jsonlite_1.7.2  magrittr_2.0.2
##  [5] evaluate_0.14   highr_0.9       xaringan_0.22   stringi_1.7.6
##  [9] rlang_1.0.0     cli_3.1.1       rstudioapi_0.13 jquerylib_0.1.4
## [13] bslib_0.3.1     rmarkdown_2.11  tools_4.1.2     stringr_1.4.0
## [17] xfun_0.29       yaml_2.2.1      fastmap_1.1.0   compiler_4.1.2
## [21] htmltools_0.5.2 knitr_1.37      sass_0.4.0
```

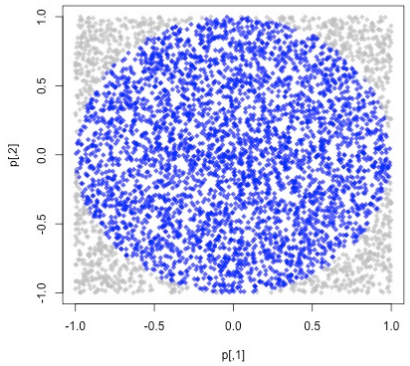
Resources

- 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{target="_blank"}

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

```
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
microbenchmark::microbenchmark(
  parallel = parSapply(cl, 1:100, pisim, nsim=nsim),
  serial   = sapply(1:100, pisim, nsim=nsim), times = 1, unit="ms"
)
```

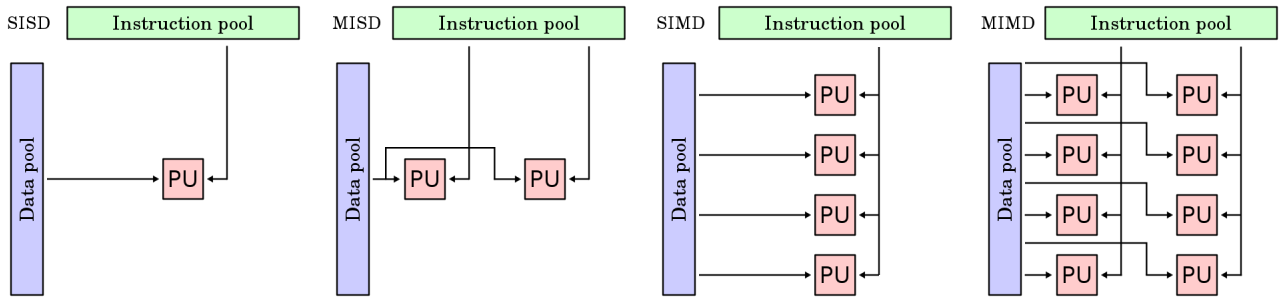
```
## Unit: milliseconds
##       expr      min       lq     mean   median       uq      max neval
##  parallel 320.6614 320.6614 320.6614 320.6614 320.6614 320.6614     1
##    serial 861.9523 861.9523 861.9523 861.9523 861.9523 861.9523     1
```

(Bonus) Overview of HPC

Using Flynn's classical taxonomy, we can classify parallel computing according to the following two dimmensions:

a. Type of instruction: Single vs Multiple

b. Data stream: Single vs Multiple



Michael Flynn's Taxonomy (wiki)

(Bonus) Parallel computing: Software

Implicit parallelization:

- tensorflow: Machine learning framework
- pqR: Branched version of R.
- Microsoft R: Microsoft's R private version (based on Revolution Analytics' R version).
- data.table (R package): Data wrangling using multiple cores.
- caret (R package): A meta package, has various implementations using parallel computing.

Explicit parallelization (DIY):

- CUDA (C/C++ library): Programming with GP-GPUs.
- Open MP (C/C++ library): Multi-core programming (CPUs).
- Open MPI (C/C++ library): Large scale programming with multi-node systems.
- Threading Building Blocks (C/C++ library): Intel's parallel computing library.
- Kokkos (C++ library): A hardware-agnostic programming framework for HPC applications.
- parallel (R package): R's built-in parallel computing package
- future (R package): Framework for parallelzing R.
- RcppParallel (R C++ API wrapper): Header and templates for building Rcpp+multi-threaded programs.
- julia (programming language): High-performing, has a framework for parallel computing as well.