

# Paralelização e programação em GPU

---

Douglas R. Mesquita Azevedo

Universidade Federal de Minas Gerais

# Motivação

---



## Somos multitarefas:

- Estudamos e escutamos música
- Fazemos comida e assistimos televisão
- Falamos no telefone e dirigimos
- ...

- Podemos realizar muitas tarefas **em paralelo**.
- Algumas tarefas **não são paralelizáveis**.
- Não necessariamente o resultado da paralelização é o ideal.



## Buscando cerveja...

- Você e mais 15 amigos foram ao Cabral...
- Missão: Comprar e distribuir 15 latões:
  - Uma cerveja de cada vez (sequencial).
  - Duas cervejas por vez (paralelo).
  - ...
  - Todas de uma vez (vai dar ruim).

Buscando mais de uma cerveja por vez economizamos tempo.

Paralelize com moderação...



# História

---



- **Quando:** 1958
- **Quem:** John Cocke e Daniel Slotnick
- **Onde:** Estados Unidos

## **Conceito:**

“Computação paralela é uma forma de computação em que vários cálculos são realizados ao mesmo tempo, operando sob o princípio de que grande problemas geralmente podem ser divididos em problemas menores, que então são resolvidos concorrentemente (em paralelo)”

Os processos são **Independentes**

## Exemplo: Produto matricial

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$



$$y_1 = 30$$



$$y_2 = 40$$



$$y_3 = 50$$

As multiplicações de linhas e colunas podem ser realizadas por pessoas  
(**processadores**) diferentes.



## Paralelização no R

---

Existem diversos pacotes de paralelização no R. Cada um com suas vantagens e desvantagens!

## Alguns pacotes...

- `snow`
- `foreach`
- `gpuR`
- ...

O mais simples de todos (na minha opinião) é o pacote `foreach`. Desta forma, esse será o pacote que iremos utilizar.

Um exemplo simples (e óbvio):

```
> require(dplyr)
> require(rbenchmark)
> require(foreach)
> #require(doParallel) # Ambos sistemas
> require(parallel)   # Ambos sistemas
> require(doMC)       # Alternativa Linux
> #require(doSNOW)    # Alternativa Windows
>
> #cl <- makeCluster(detectCores())      # parallel
> #registerDoParallel(cl)               # parallel
> registerDoMC(cores = detectCores())  # doMC
> #registerDoSNOW(cores = detectCores()) # doSNOW
> sleep_time <- 0.01
>
> bench <- benchmark("for" = for(i in 1:4){Sys.sleep(sleep_time)},
+                   "do" = foreach(n = 1:4) %do% Sys.sleep(sleep_time),
+                   "dopar" = foreach(n = 1:4) %dopar% Sys.sleep(sleep_time),
+                   columns = c("test", "replications",
+                               "elapsed", "user.self", "sys.self"))
> bench <- bench %>% arrange(elapsed)
> bench
  test replications elapsed user.self sys.self
1 dopar          100   1.759    0.412   0.316
2  for           100   4.036    0.008   0.008
3   do            100   4.530    0.496   0.012
```

Primeiramente vamos entender os operadores `%>%`, `%do%` e `%dopar%`

- `%>%`:
  - Não tem nada a ver com paralelização
  - Muito bom para manipulação de dados
  - Utilizo inconscientemente
- `%do%`:
  - Operador do pacote `foreach`
  - Não é paralelizado
  - Retorna o resultado de um loop no formato desejado (vetor, lista, ...)
- `%dopar%`:
  - Operador do pacote `foreach`
  - Paralelizado
  - Retorna o resultado de um loop no formato desejado (vetor, lista, ...)

Resumidamente precisamos de **três** passos para paralelizar um código

- **Passo 1:** Definir previamente o número de processadores
- **Passo 2:** Utilizar o operador **%dopar%**
- **Passo 3:** Estar atento na forma de retornar os objetos desejados!

Em um *loop* convencional cada iteração retorna um objeto que é alocado em uma posição. Utilizando o pacote **foreach** o resultado completo do *loop* é alocado em um objeto. Ahn?



## Exemplo de uso

```
> set.seed(1)
> ##-- Tamanho de amostra e do loop ----
> n <- 10000
> nRep <- 5000
> ##-- Loop sequencial ----
> objeto1 <- numeric(nRep)
> t1 <- Sys.time()
> for(i in 1:nRep){
+   objeto1[i] <- mean(rnorm(n))
+ }
> t2 <- Sys.time()
> ##-- Loop paralelizado ----
> t3 <- Sys.time()
> objeto2 <- foreach(i = 1:nRep) %dopar% {
+   mean(rnorm(n))
+ }
> t4 <- Sys.time()
> ##-- Resultados ----
> class(objeto1)
[1] "numeric"
```

```
> class(objeto2)
[1] "list"
```

```
> t2 - t1
Time difference of 4.550889 secs
```

```
> t4 - t3
Time difference of 2.14652 secs
```

## E se fizéssemos diferente?

```
> objeto2 <- numeric(nRep)
>
> aux <- foreach(i = 1:nRep) %dopar% {
+   objeto2[i] <- mean(rnorm(n))
+ }
>
> head(objeto2)
[1] 0 0 0 0 0 0
```

```
> head(aux)
[[1]]
[1] -0.004779611

[[2]]
[1] -0.002974301

[[3]]
[1] -0.002751514

[[4]]
[1] 0.003785111

[[5]]
[1] 0.003435557

[[6]]
[1] -0.017541
```

# Argumento “.combine”

Podemos **combinar** o resultado de *loop* de várias formas...

```
> n <- 100
> nRep <- 5000
>
> objeto1 <- foreach(i = 1:nRep, .combine = 'list') %dopar% {
+   mean(rnorm(n))
+ }
> class(objeto1)
[1] "list"
```

```
> objeto2 <- foreach(i = 1:nRep, .combine = 'c') %dopar% {
+   mean(rnorm(n))
+ }
> class(objeto2)
[1] "numeric"
```

```
> objeto3 <- foreach(i = 1:nRep, .combine = 'rbind') %dopar% {
+   rnorm(n)
+ }
> dim(objeto3)
[1] 5000 100
```

```
> objeto4 <- foreach(i = 1:nRep, .combine = 'cbind') %dopar% {
+   rnorm(n)
+ }
> dim(objeto4)
[1] 100 5000
```



## Sumarizando colunas

- Gere 10000 amostras de 5000 observações de uma distribuição normal
- Para cada amostra calcule:
  - Mínimo
  - Primeiro quartil
  - Mediana
  - Media
  - Terceiro quartil
  - Máximo
  - Variância
- Obtenha um objeto de dimensão 10000x7

Estas leis respondem à seguinte pergunta: **Como saber se vale a pena paralelizar meu código?**

- **Lei de Amdahl:**  $S = \frac{1}{1 - P + \frac{P}{S_p}}$ 
  - **S** é a aceleração do código
  - **P** é a fração paralelizável
  - **S<sub>p</sub>** é a aceleração potencial (nº de processadores)
- **Lei de Gustafson:**  $S = 1 - P + P * S_p$ 
  - **S** é a aceleração do código
  - **P** é a fração paralelizável
  - **S<sub>p</sub>** é a aceleração potencial (nº de processadores)

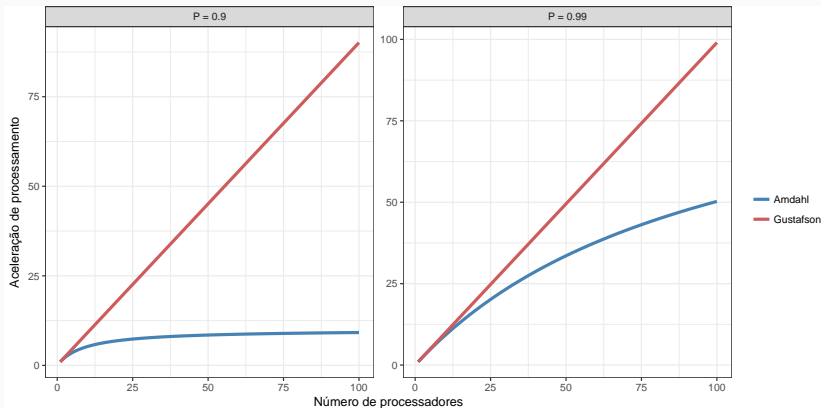
# Leis de Amdahl e Gustafson

## Exemplo 1: $P = 0.3$ e $S_p = 1$

- Amdahl:  $S = 1$
- Gustafson  $S = 1$

## Exemplo 2: $P = 0.3$ e $S_p = 4$

- Amdahl:  $S = 1.29$
- Gustafson  $S = 1.9$



Nem sempre paralelizar um processo é um bom negócio!

Windows

An error has occurred. To continue:

Press Enter to return to Windows, or

Press CTRL+ALT+DEL to restart your computer. If you do this,  
you will lose any unsaved information in all open applications.

Error: 0E : 016F : BFF9B3D4

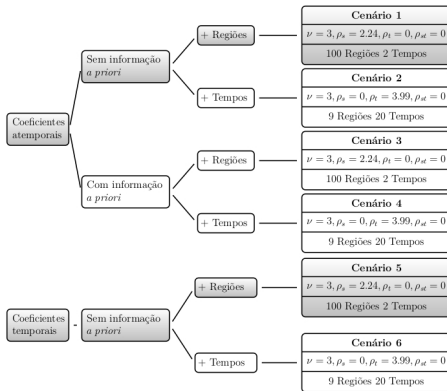
Press any key to continue \_

- Processos **muito caros** podem tornar o processamento lento
  - Exige demais do processador
- Processos **muito baratos** podem tornar o processamento lento
  - O tempo para paralelizar é maior que o tempo de execução



## Dissertação

- 6 Cenários
- 4 Modelos
- 100 Bases por cenário
- 2400 Bases utilizadas
- 9600 Modelos ajustados
- Paralelização:



## Validação cruzada para escolha de parâmetros do SVM

```
> require(e1071)
> require(mvtnorm)
> set.seed(26)
> ##-- Criando uma base de dados ----
> nCov <- 5
> n <- 2000
> X <- rmvnorm(n = n, mean = rep(0, nCov))
> betas <- rnorm(n = nCov)
> Xbeta <- X%*%betas
> y <- ifelse(Xbeta < 0, 0, 1)
> data <- data.frame(y = y, X)
> ##-- Separando a base em treino e teste ----
> id <- 1:nrow(data)
> testindex <- sample(id, trunc(length(id)/3))
> testset <- data[testindex,]
> trainset <- data[-testindex,]
> ##-- Criando os grids para avaliação dos parâmetros de tunagem ----
> scale = c(TRUE, FALSE)
> kernel = c("linear", "polynomial", "radial", "sigmoid")
> degree = 2:4
> gamma = 1/seq(nrow(trainset)-(n/10), nrow(trainset) + (n/10), length.out = 4)
> cost = seq(1, 10, length.out = 4)
> parameters <- expand.grid(scale = scale,
+                             kernel = kernel,
+                             degree = degree,
+                             gamma = gamma,
+                             cost = cost)
```

Tempo computacional utilizando um *loop sequencial*:

```
> t1 <- system.time(  
+   svm.output_1 <- foreach(i = 1:nrow(parameters), .combine = 'rbind') %do% {  
+     svm.model <- e1071::svm(formula = y ~ .,  
+                           data = trainset,  
+                           scale = parameters[i, 1],  
+                           kernel = parameters[i, 2],  
+                           degree = parameters[i, 3],  
+                           gamma = parameters[i, 4],  
+                           cost = parameters[i, 5],  
+                           type = "C-classification")  
+     svm.pred <- predict(svm.model, testset[,-1])  
+     svm.confusao <- table(pred = svm.pred, true = testset[,1])  
+     svm.acuracia <- sum(diag(svm.confusao))/sum(svm.confusao)  
+     cbind(parameters[i,], acuracia = svm.acuracia)  
+   }  
+ )  
>  
> t1[3]  
elapsed  
63.964
```



Tempo computacional utilizando um *loop* **paralelizado**:

```
> t2 <- system.time(  
+   svm.output_2 <- foreach(i = 1:nrow(parameters), .combine = 'rbind') %dopar% {  
+     svm.model <- e1071::svm(formula = y ~ .,  
+                           data = trainset,  
+                           scale = parameters[i, 1],  
+                           kernel = parameters[i, 2],  
+                           degree = parameters[i, 3],  
+                           gamma = parameters[i, 4],  
+                           cost = parameters[i, 5],  
+                           type = "C-classification")  
+     svm.pred <- predict(svm.model, testset[,-1])  
+     svm.confusao <- table(pred = svm.pred, true = testset[,1])  
+     svm.acuracia <- sum(diag(svm.confusao))/sum(svm.confusao)  
+     cbind(parameters[i,], acuracia = svm.acuracia)  
+   }  
+ )  
>  
> t2[3]  
elapsed  
19.131
```

Obtivemos os mesmos resultados?

```
> all.equal(svm.output_1, svm.output_2)
[1] TRUE
```

```
> best.config <- svm.output_2 %>% arrange(desc(acuracia)) %>% slice(1)
> best.config
  scale kernel degree      gamma cost  acuracia
1  TRUE linear      2 0.0008818342  1 0.9954955
```

70.09 %

de economia de tempo

Vamos comparar o AIC de todas as possíveis regressões:

```
> set.seed(26)
>
> nCov <- 12
> n <- 1000
> cov <- rmvnorm(n = n, mean = rep(0, nCov))
> betas <- as.matrix(c(rnorm(5), rep(0, nCov - 5)), ncol = 1)
> y <- cov%*%betas
> data <- data.frame(y = y, cov)
>
> vars <- paste0("X", 1:nCov)
> formulas <- unlist(sapply(X = 1:nCov, FUN = function(x) combn(x = vars, m = x, simplify = F)),
+                   recursive = F)
> formulas <- sapply(X = formulas, FUN = function(x) paste("y ~", paste(x, collapse = "+")))
```

Com 12 covariáveis temos 4095 possíveis regressões.

## Exemplo: Regressão linear

Tempo computacional utilizando um *loop* sequencial:

```
> aic_for <- data.frame(matrix(0, nrow = 2^nCov - 1, ncol = 2))
> names(aic_for) <- c("Índice", "AIC")
>
> t1 <- system.time(
+   for(i in 1:length(formulas)){
+     reg <- lm(formula = formulas[i], data = data)
+     aic_for[i, ] <- c(i, AIC(reg))
+   }
+ )
> t1[3]
elapsed
  9.744
```

```
> aic_for %>% arrange(AIC) %>% slice(1) %>% .$"Índice" %>% formulas[.]
[1] "y ~ X1+X2+X3+X4+X5+X9"
```

Tempo computacional utilizando um *loop* **paralelizado**:

### Paralelizando a regressão

- Torne o código do *slide* anterior **paralelizado**
- Retorne uma base das mesmas dimensões da base **aic\_for**
- Verifique se os resultados são iguais

## Processamento

---

Paralelizando um código nós utilizamos os processadores de forma ótima

```
douglas@azul: ~
top - 11:27:59 up 2:37, 1 user, load average: 0,56, 0,47, 0,31
Tasks: 212 total, 2 running, 210 sleeping, 0 stopped, 0 zombie
%Cpu0  : 1,0 us,  0,0 sy,  0,0 ni, 98,7 id,  0,3 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu1  : 100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu2  : 1,3 us,  0,0 sy,  0,0 ni, 98,7 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu3  : 1,3 us,  0,7 sy,  0,0 ni, 93,7 id,  4,3 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem : 16397120 total, 13557372 free, 1147396 used, 1692352 buff/cache
KiB Swap: 16740348 total, 16740348 free,  0 used, 14865696 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM    TIME+  COMMAND
 3426 douglas  20   0  727868 159192 28412 R  99,7  1,0   0:28.40 rsession
```

Loop não  
paralelizado

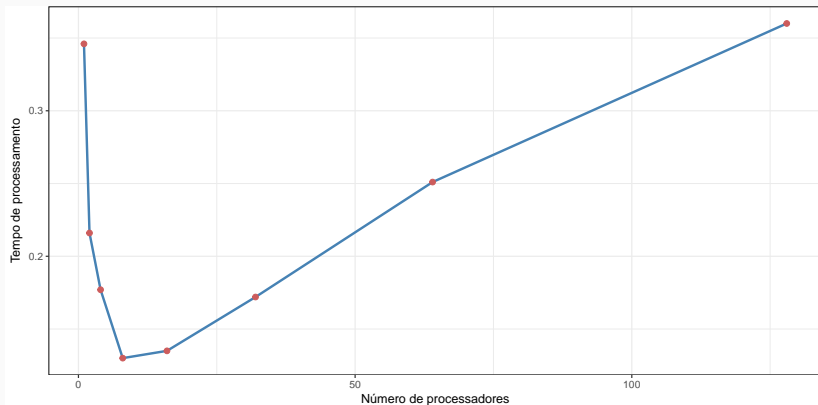
```
douglas@azul: ~
top - 11:34:36 up 2:43, 1 user, load average: 1,49, 1,10, 0,68
Tasks: 223 total, 5 running, 218 sleeping, 0 stopped, 0 zombie
%Cpu0  : 96,7 us,  3,3 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu1  : 99,7 us,  0,3 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu2  : 94,0 us,  6,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu3  : 93,0 us,  7,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem : 16397120 total, 12635292 free, 1979356 used, 1782472 buff/cache
KiB Swap: 16740348 total, 16740348 free,  0 used, 13957984 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM    TIME+  COMMAND
 6469 douglas  20   0  724820 132856  4872 R  94,4  0,8   0:08.50 rsession
 6467 douglas  20   0  724820 133168  5184 R  86,7  0,8   0:08.31 rsession
 6466 douglas  20   0  730628 138932  5340 R  83,7  0,8   0:07.92 rsession
 6468 douglas  20   0  730080 137952  5020 R  82,1  0,8   0:07.84 rsession
```

Loop paralelizado

## Definindo mais processadores

O que acontece se eu disser que tenho **mais processadores** do que eu realmente tenho? **Depende...**



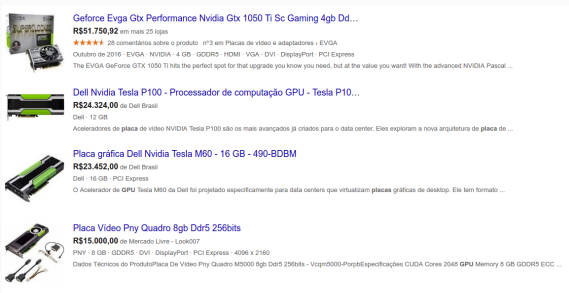


## **Paralelização utilizando uma placa gráfica**


---

# Paralelização utilizando uma placa gráfica


- **O que:** Graphics Processing Unit
- **Quando:** Anos 70
- **Para que servia?** Games
- **Para que serve?** Games e Machine Learning




**GeForce Evga Gtx Performance Nvidia Gtx 1050 Ti Sc Gaming 4gb Dd...**  
**R\$51.750,92** em mais 29 lojas  
★★★★★ 28 comentários sobre o produto - 11"3 em Placas de vídeo e adaptadores - EVGA  
Outubro de 2016 - EVGA - NVIDIA - 4 GB - GDDR5 - HDMI - VGA - DVI - DisplayPort - PCI Express  
The EVGA GeForce GTX 1050 Ti hits the perfect spot for that upgrade you know you need, but at the value you want! With the advanced NVIDIA Pascal ...



**Dell Nvidia Tesla P100 - Processador de computação GPU - Tesla P10...**  
**R\$24.324,00** de Dell Brasil  
Dell - 12 GB  
Aceleradores de placa de vídeo NVIDIA Tesla P100 são os mais avançados já criados para o data center. Eles exploram a nova arquitetura de placa de ...



**Placa gráfica Dell Nvidia Tesla M60 - 16 GB - 490-BDBM**  
**R\$23.452,00** de Dell Brasil  
Dell - 16 GB - PCI Express  
O Acelerador de GPU Tesla M60 da Dell foi projetado especificamente para data centers que virtualizam placas gráficas de desktop. Ele tem formato ...



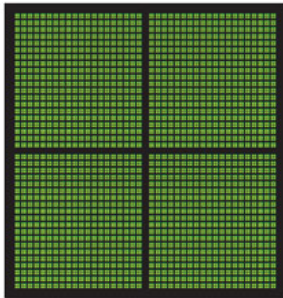
**Placa Vídeo Pny Quadro 8gb Ddr5 256bits**  
**R\$15.000,00** de Mercado Livre - Look007  
PNY - 8 GB - GDDR5 - DVI - DisplayPort - PCI Express - 4096 x 2160  
Dados Técnicos do Produto/Placa De Vídeo Pny Quadro M5000 8gb Ddr5 256bits - Vozm5000-Propb/Especificações CUDA Cores 2048 GPU Memory 8 GB GDDR5 ECC ...

## Paralelização utilizando uma placa gráfica

A ideia é que uma GPU pode realizar **milhares de operações**. O problema deve ser transformado em **pequenas operações**.



CPU  
MULTIPLE CORES

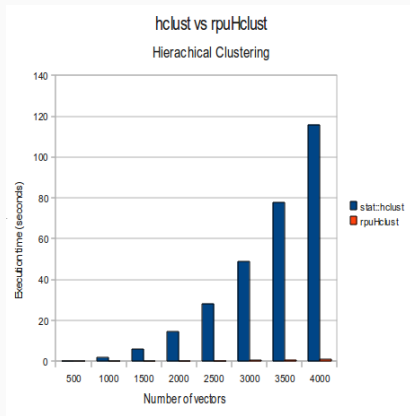
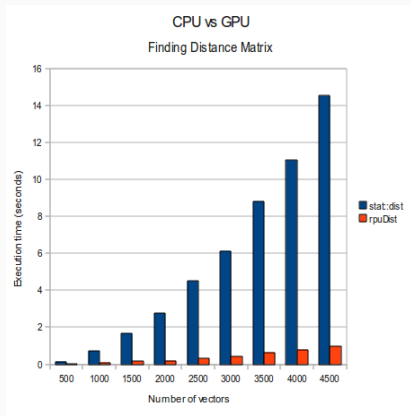


GPU  
THOUSANDS OF CORES



[Link](#) para o vídeo

Já existem pacotes no R capazes de trabalhar com **programação em GPU**. Um exemplo é o pacote **gpuR**.



## Populações do Censo 2010 - Parte 1

1. Leia as bases de dados por estado
2. Junte-as em um único arquivo
3. Observe que a base possui a população **segmentada**
4. Crie uma base auxiliar contendo as populações por município
5. Descubra quais são os municípios com mais de 500.000 habitantes
6. Restrinja a base inicial aos municípios encontrados no item 5
7. Crie 4 bases auxiliares
  - 7.1 Uma contendo as populações dos município por sexo
  - 7.2 Uma contendo as populações dos município por etnia
  - 7.3 Uma contendo as populações dos município por analfabetismo
  - 7.4 Uma contendo as populações dos município por situação do domicílio
8. O item 1 deve ser realizado sequencialmente e em paralelo
9. Valeu a pena paralelizar o código?

## Populações do Censo 2010 - Parte 2

1. Para cada **município** do **item 5 da parte 1** crie um relatório
2. Os parâmetros do relatório são (para cada cidade):
  - 2.1 **cidade** - Nome da cidade em análise
  - 2.2 **estado** - Sigla do estado em análise
  - 2.3 **pop\_sexo** - Base contendo a população do sexo Feminino e Masculino
  - 2.4 **pop\_etnia** - Base contendo a população Amarela, Branca, Ignorado, Indígena, Parda, Preta
  - 2.5 **pop\_analf** - Base contendo a população Analfabeta (Não), Alfabetizada (Sim) e NA
  - 2.6 **pop\_situacao** - Base contendo a população Rural e Urbana
3. O item 1 deve ser realizado sequencialmente e em paralelo
4. Há um ganho em paralelizar o código?

Para listar arquivos em um diretório pense em:

- `?list.files`

Sempre que o objetivo for calcular a **população agregada** por algum fator pense em:

- `?aggregate`
- `?dplyr::group_by`

Sempre que o objetivo for restringir a base pense em:

- `?subset`
- `?dplyr::filter`



## Exercício - Gerando um relatório

```
> rmarkdown::render(input = "relatorio/relatorio.Rmd",
+                   output_dir = "relatorio/cidades/sequencial/",
+                   output_file = paste0(4322004, ".html"),
+                   intermediates_dir = tempfile(),
+                   params = list(estado = "RS",
+                                cidade = "Triunfo",
+                                pop_sexo = data_sexo,
+                                pop_etnia = data_etnia,
+                                pop_analf = data_analf,
+                                pop_situacao = data_situacao))
```

- `data_sexo` - data.frame contendo as populações por sexo da cidade em questão
- `data_etnia` - data.frame contendo as populações por etnia da cidade em questão
- `data_analf` - data.frame contendo as populações por níveis de analfabetismo da cidade em questão
- `data_situacao` - data.frame contendo as populações urbana e rural da cidade em questão

Exemplo de dado de entrada para o relatório (`data_sexo`):

	COD_UF	NO_UF	COD_MUN	NO_MUN	SEXO	POP
1	23	CE	2304400	Fortaleza	Feminino	1304267
2	23	CE	2304400	Fortaleza	Masculino	1147918

- [https://en.wikipedia.org/wiki/Parallel\\_computing](https://en.wikipedia.org/wiki/Parallel_computing)
- <https://www.r-bloggers.com/how-to-go-parallel-in-r-basics-tips/>
- <http://blog.aicry.com/r-parallel-computing-in-5-minutes/>
- [https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)
- <https://cran.r-project.org/web/packages/doParallel/vignettes/gettingstartedParallel.pdf>
- <https://www.r-bloggers.com/the-wonders-of-foreach/>
- <ftp://cran.r-project.org/pub/R/web/packages/foreach/vignettes/foreach.pdf>
- <https://cran.r-project.org/web/packages/foreach/foreach.pdf>
- <http://www.nvidia.com/object/what-is-gpu-computing.html>
- <http://www.r-tutor.com/gpu-computing>