Introduction

ANF R pour le calcul

Daphné Giorgi

23 septembre 2024

Villa Clythia, Fréjus

Calcul parallèle avec R, *Vincent Miele, Violaine Louvet*, EDP sciences, 2016

Calcul HPC

High Performance Computing

Haute Performance :

- Accélérer la vitesse du code
- Diminuer l'empreinte mémoire

Mais attention ! Avant d'être performant un code doit être **exact** et **clair**.

Développement

Comment développer un code exact et clair ?

- Modularité du code
- Intégration Continue
- Gestionnaire de version (git, svn, forges, ...)
- Tests
- Convention de nommage
- Refactoring

Temps de calcul

`system.time` : précision faible

```
> x <- runif(100)
> system.time(for(i in 1:1e5){sqrt(x)})
   user  system elapsed
  0.031   0.013   0.045
```

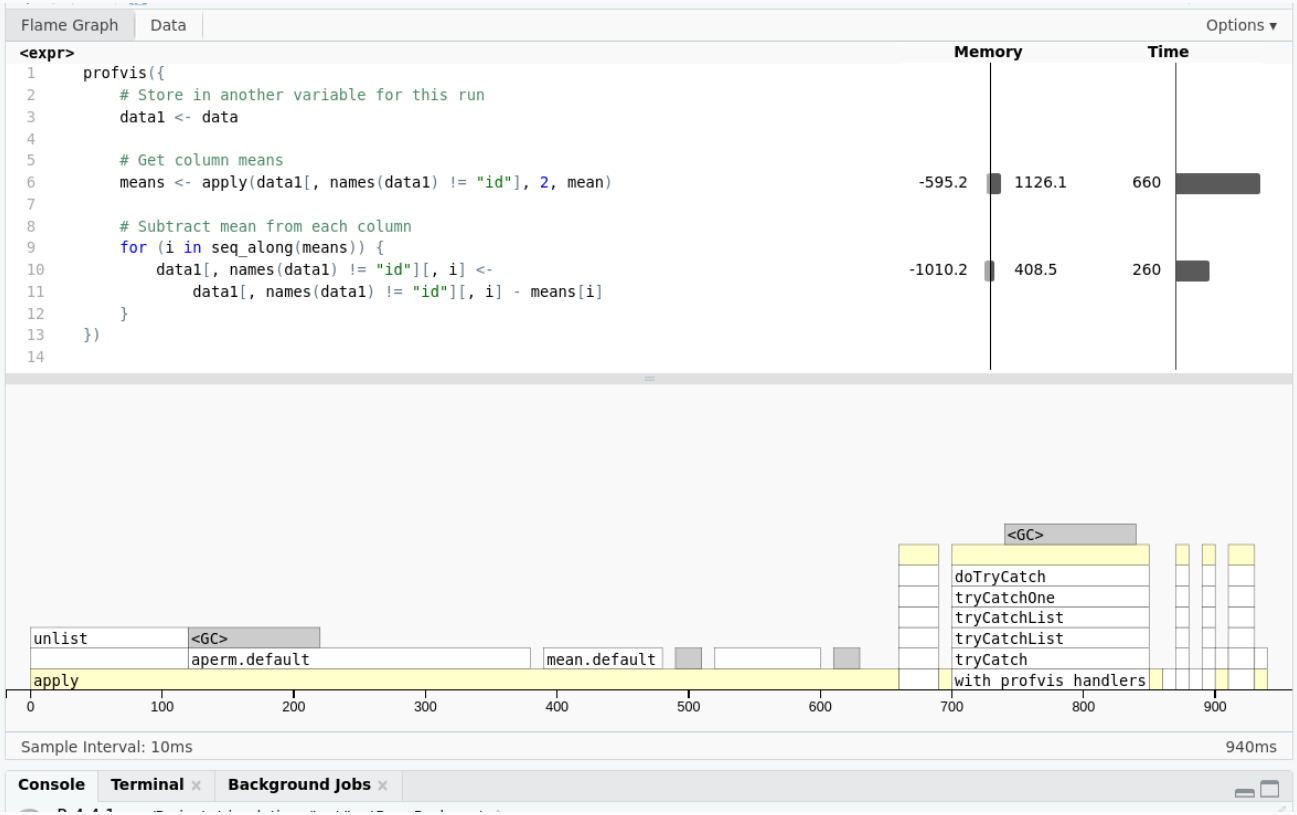
`microbenchmark` : plus grande précision et distribution du temps de calcul

```
> library(microbenchmark)
> microbenchmark(sqrt(x), x^(1./2.), times=1e5)
Unit: nanoseconds
     expr  min   lq       mean median     uq      max neval cld
  sqrt(x)  210  225  352.0576    231  262.5 7151792 1e+05  a
 x^(1/2) 1452 1476 1806.9053   1499 1593.0 9142488 1e+05   b
```

Profilage de code

Très souvent, quelques lignes de code sont responsables d'un grand pourcentage du temps de calcul global.

Un profileur (`Rprof`, `profvis`, ...) aide à trouver ces lignes.



Gestion de la mémoire

La librairie `pryr` permet d'obtenir une bonne compréhension de la gestion de la mémoire de `R`.

Ainsi il est possible de prévoir la quantité de mémoire nécessaire pour une tâche donnée et tirer le meilleur parti de la mémoire disponible.

Cela peut même aider à écrire un code plus rapide, car les copies accidentelles sont une cause majeure de lenteur du code.

Fonctions principales :

- `object.size(x)` : taille de l'objet `x`
- `mem_used()` : taille de tous les objets en mémoire
- `tracemem(x)` : adresse de l'objet `x`

Premiers pas d’optimisation

Vectorisation

- Éviter les boucles `for`/`while`
- Utiliser les fonctions

  `*apply` (`apply`, `lapply`, `sapply`, `vapply`), mais pas toujours concluant
- Utiliser les *fonctions vectorisées* (écrites en `C`) par exemple

  `rowSum`, `colSum`, `rowMeans`, `max.col`, `cumsum`, `diff`, `pmin`, `pmax`.

Vectorisation

Exemple : somme des éléments des lignes d'une matrice.

Boucle `for` :

```
myRowSum <- function(mat){
  sums <- rep(0, nrow(mat))
  for (i in 1:nrow(mat))
    for (j in 1:ncol(mat))
      sums[i] <- sums[i] + mat[i,j]
  sums
}
```

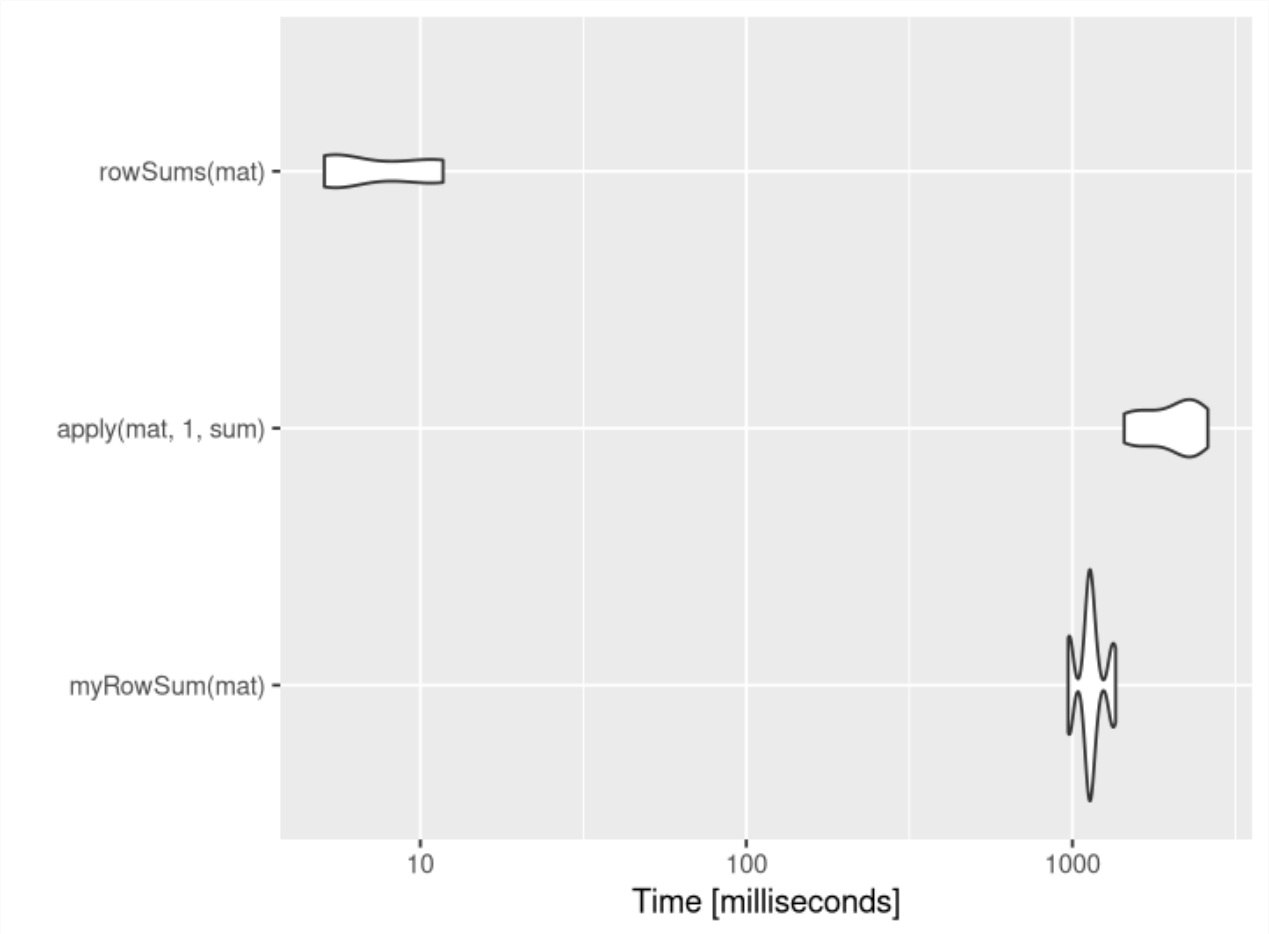
Fonctions `*apply` :

```
apply(mat,1,sum)
```

Fonction vectorisée : `rowSums`.

Vectorisation

Figure 1 – `mat <- matrix(rnorm(1e3*1e3, 1e3,1e3))`
`microbenchmark(myRowSum(mat), apply(mat,1,sum), rowSums(mat), times=10)`



Préallocation de la mémoire

Dans R la mémoire est allouée **dynamiquement**.

La création et la destruction de mémoire ont un coût.

Lorsque la taille des objets est connue à l'avance, il est recommandé de préallouer la mémoire.

Préallocation de la mémoire

Exemple : préallocation.

Sans :

```
noprealloc <- function(n){
  result <- c()
  for (i in 1:n) result <- c(result, rnorm(1))
  result
}
```

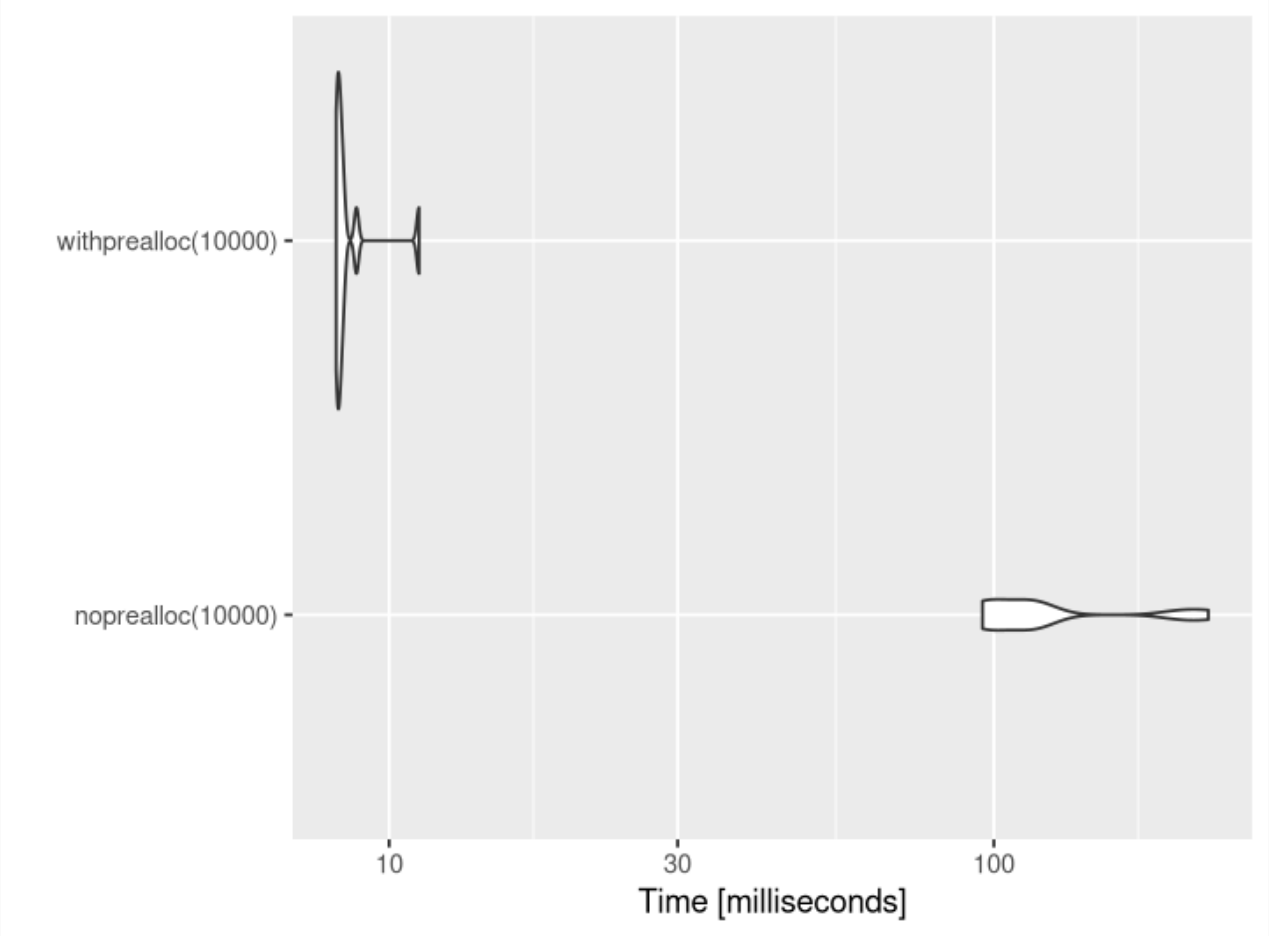
Avec :

```
withprealloc <- function(n){
  result <- double(n)
  for (i in 1:n) result[i] <- rnorm(1)
  result
}
```

Préallocation de la mémoire

Figure 2 –

```
microbenchmark(noprealloc(1e4), withprealloc(1e4), times=10)
```



Éviter les duplications en mémoire

Un entier doit être suffixé de la lettre `L`, sinon il sera considéré comme un réel.

Tracer régulièrement la mémoire.

Détruire les objets temporaires avec la fonction `rm()`.

Éviter les duplications en mémoire

```
> v <- 1:1e8
> class(v)
[1] "integer"
> tracemem(v)
[1] "<0x564dde5a9380>"
> format(object.size(v), standard = "SI", units="MB")
[1] "400 MB"
> v[1] <- 0
> tracemem(v)
[1] "<0x7f9f08400010>"
> format(object.size(v), standard = "SI", units="MB")
[1] "800 MB"
```

Utiliser les librairies existantes

Ne pas recoder quelque chose qui existe déjà.

La première étape de programmation est la phase de recherche de l'existant.

Reconnaître ses communautés.

Réfléchir à la structure de données

Bien réfléchir à la structure de données.

- Passer de `matrix` à `sparseMatrix` lorsqu'on travaille avec des matrices creuses.
- Trier les vecteurs et utiliser des fonctions spécifiques si on fait plusieurs recherche dans le vecteur (ex `binsearch`)

Un pas de plus

Rcpp

Implémenter les parties plus coûteuses en **langage compilé**.

`Rcpp` est un paquet `R` qui permet

- coder des fonctions en langage `C++`
- les compiler depuis `R`
- les appeler depuis `R`

Calcul parallèle

Calcul parallèle

La fréquence des unités de calcul a cessé d'augmenter : *The free lunch is over.*

Nécessité du calcul parallèle.

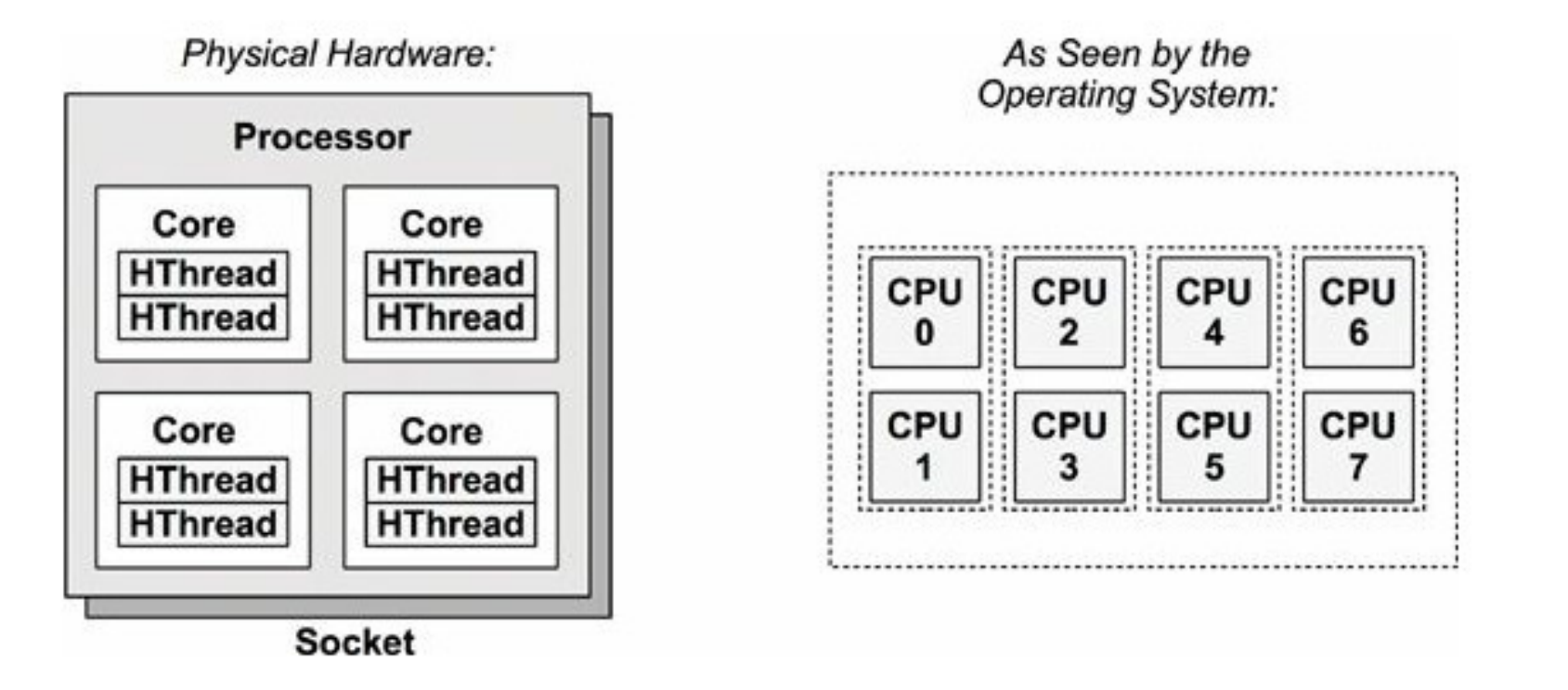
Architectures parallèles

Nœuds et CPUs

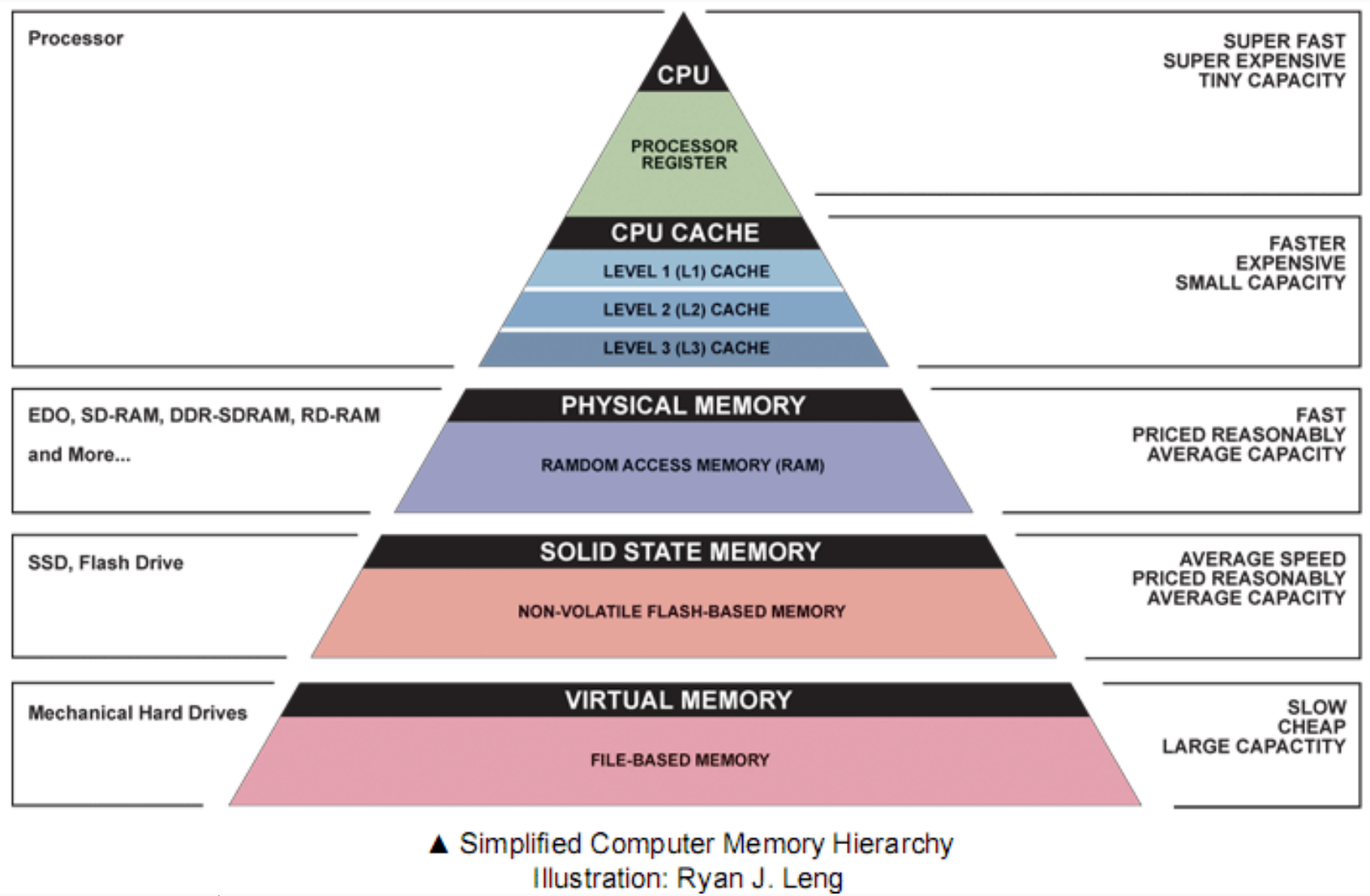
- Un nœud de calcul correspond à une machine de calcul
- Un nœud contient des processeurs physiques (sockets) contenant eux-mêmes des cœurs physiques (cores)

```
 bash$ lscpu
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          39 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   8
  On-line CPU(s) list:    0-7
Vendor ID:                GenuineIntel
  Model name:             11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz
    CPU family:           6
    Model:                140
    Thread(s) per core:   2
    Core(s) per socket:   4
    Socket(s):            1
```

CPUs physiques ou logiques



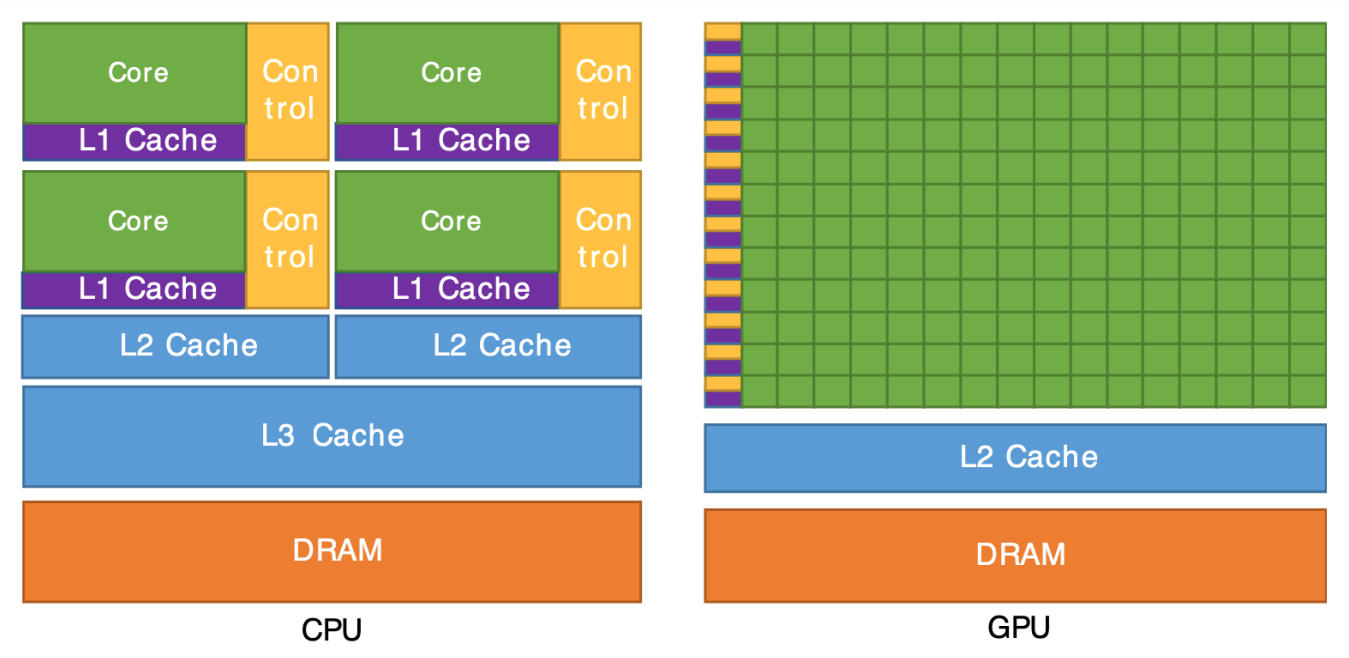
Accès à la mémoire



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

GPGPU

Il y a 20 ans, les GPU étaient utilisés principalement pour accélérer les rendus graphiques 3D, tels que les jeux.
Depuis les scientifiques ont réalisé que les GPU peuvent résoudre des problèmes numériques de (très grande) taille : **General Purpose GPU**



Architectures parallèles

Infrastructures

- Un **nœud** multicœurs : du portable au serveur de calcul
- Un nœud multicœurs avec **GPU**
- Un **cluster** de calcul : un ensemble de nœuds qui communiquent entre eux à travers un réseau

Calcul parallèle

Calcul séquentiel : Exécuter un même code en séquentiel sur une unité de calcul

Calcul réparti : Exécuter un même code en séquentiel sur plusieurs jeux de données différent

Calcul parallèle :

- à mémoire *partagée*
- à mémoire *distribuée*

Calcul parallèle

Granularité :

- grain-fin : nombreuses tâches de petite taille
- gros-grain : petit nombre de tâches conséquentes

Speed up : Accélération obtenue après une amélioration du code

Scalabilité : Capacité d'un programme à passer à l'échelle, augmenter les performances lorsqu'on augmente les ressources.

- Scalabilité forte : On fixe la taille du problème et on augmente le nombre d'unités de calcul.
- Scalabilité faible : On augmente la taille du problème en même temps que le nombre d'unités de calcul.

Overhead : Surcoût engendré par la mise en place du parallélisme (lecture/écriture mémoire, ...).

Programme

Mardi 24	*Mercredi 25*	*Jeudi 26*	*Vendredi 27*
`Rcpp` `R` parallel	`Rcpp` parallel	`Rmpi` `RKeOps`	Snakemake