

Programmation concurrente

Techniques avancées en programmation statistique R

Patrick Fournier

Automne 2019

Université du Québec à Montréal

Φρόνησις

Le dîner des philosophes

Adapté de Dijkstra [1].

Le dîner des philosophes

Adapté de Dijkstra [1].

↪ La vie d'un philosophe est une alternance entre deux actions :

Le dîner des philosophes

Adapté de Dijkstra [1].

↪ La vie d'un philosophe est une alternance entre deux actions :

↪ penser et

Le dîner des philosophes

Adapté de Dijkstra [1].

↪ La vie d'un philosophe est une alternance entre deux actions :

↪ penser et

↪ manger.

Le dîner des philosophes

Adapté de Dijkstra [1].

- ↪ La vie d'un philosophe est une alternance entre deux actions :
 - ↪ penser et
 - ↪ manger.
- ↪ Les philosophes ont un appétit infini.

Le dîner des philosophes

Adapté de Dijkstra [1].

- ↪ La vie d'un philosophe est une alternance entre deux actions :
 - ↪ penser et
 - ↪ manger.
- ↪ Les philosophes ont un appétit infini.
- ↪ Approvisionnement en nourriture infini.

Le dîner des philosophes

Adapté de Dijkstra [1].

- ↪ La vie d'un philosophe est une alternance entre deux actions :
 - ↪ penser et
 - ↪ manger.
- ↪ Les philosophes ont un appétit infini.
- ↪ Approvisionnement en nourriture infini.
- ↪ Pour manger, un philosophe a besoin de 2 baguettes.

Le dîner des philosophes

Adapté de Dijkstra [1].

- ↪ La vie d'un philosophe est une alternance entre deux actions :
 - ↪ penser et
 - ↪ manger.
- ↪ Les philosophes ont un appétit infini.
- ↪ Approvisionnement en nourriture infini.
- ↪ Pour manger, un philosophe a besoin de 2 baguettes.
- ↪ Un philosophe n'a accès qu'aux baguettes lui étant adjacentes.

Le dîner des philosophes



Le dîner des philosophes



$\#baguettes = 2 \times \#\text{philosophes} \Rightarrow \text{philosophes vertueux}$

Le dîner des philosophes



Le dîner des philosophes



#baguettes = *#philosophes* \Rightarrow philosophes vicieux

La solution naïve est la suivante :

-
- 1: Penser jusqu'à ce que la baguette gauche soit disponible.
 - 2: Prendre la baguette gauche.
 - 3: Penser jusqu'à ce que la baguette droite soit disponible.
 - 4: Prendre la baguette droite.
 - 5: Manger pendant s secondes.
 - 6: Déposer la baguette droite.
 - 7: Déposer la baguette gauche.
 - 8: Répéter.
-

↪ Les 5 philosophes prennent la baguette gauche en même temps.

Solution naïve

- ~→ Les 5 philosophes prennent la baguette gauche en même temps.
- ~→ Or, la baguette gauche de l'un est la baguette droite de l'autre.

- ~> Les 5 philosophes prennent la baguette gauche en même temps.
- ~> Or, la baguette gauche de l'un est la baguette droite de l'autre.
- ~> Plus personne n'a de baguette droite!

Solution naïve

- ~> Les 5 philosophes prennent la baguette gauche en même temps.
- ~> Or, la baguette gauche de l'un est la baguette droite de l'autre.
- ~> Plus personne n'a de baguette droite!
- ~> Les philosophes sont alors condamnés à penser jusqu'à mourir d'inanition.

Solution naïve

- ~> Les 5 philosophes prennent la baguette gauche en même temps.
- ~> Or, la baguette gauche de l'un est la baguette droite de l'autre.
- ~> Plus personne n'a de baguette droite!
- ~> Les philosophes sont alors condamnés à penser jusqu'à mourir d'inanition.
- ~> Problème classique en programmation concurrente.

Solution naïve

- ~> Les 5 philosophes prennent la baguette gauche en même temps.
- ~> Or, la baguette gauche de l'un est la baguette droite de l'autre.
- ~> Plus personne n'a de baguette droite!
- ~> Les philosophes sont alors condamnés à penser jusqu'à mourir d'inanition.
- ~> Problème classique en programmation concurrente.
- ~> Pour une analyse détaillée, voir [1].

Problèmes en programmation concurrente

Programme séquentiel

Programme séquentiel

↪ Ordre d'exécution des instructions : dépend
uniquement du programme.

Séquentiel vs. concurrent

Programme séquentiel

↪ Ordre d'exécution des instructions : dépend
uniquement du programme.

Programme concurrent

Séquentiel vs. concurrent

Programme séquentiel

↪ Ordre d'exécution des instructions : dépend *uniquement* du programme.

Programme concurrent

↪ Ordre d'exécution des instructions : dépend

Séquentiel vs. concurrent

Programme séquentiel

- ~> Ordre d'exécution des instructions : dépend *uniquement* du programme.

Programme concurrent

- ~> Ordre d'exécution des instructions : dépend
 - ~> du programme et

Séquentiel vs. concurrent

Programme séquentiel

- ~> Ordre d'exécution des instructions : dépend *uniquement* du programme.

Programme concurrent

- ~> Ordre d'exécution des instructions : dépend
 - ~> du programme et
 - ~> de l'environnement dans lequel il s'exécute (système d'exploitation).

Séquentiel vs. concurrent

Programme séquentiel

- ~> Ordre d'exécution des instructions : dépend *uniquement* du programme.

Programme concurrent

- ~> Ordre d'exécution des instructions : dépend
 - ~> du programme et
 - ~> de l'environnement dans lequel il s'exécute (système d'exploitation).
- ~> De manière générale, on ne contrôle pas le système d'exploitation.

Séquentiel vs. concurrent

Programme séquentiel

- ~> Ordre d'exécution des instructions : dépend *uniquement* du programme.

Programme concurrent

- ~> Ordre d'exécution des instructions : dépend
 - ~> du programme et
 - ~> de l'environnement dans lequel il s'exécute (système d'exploitation).
- ~> De manière générale, on ne contrôle pas le système d'exploitation.
- ~> En conséquence, l'ordre d'exécution d'un programme concurrent est aléatoire.

Séquentiel vs. concurrent

Exemple 1 : Remplissage d'un vecteur.

```
1  > vec_seq <- numeric(0)
2  > vec1 <- numeric(0)
3  > vec2 <- numeric(0)
4  > for (kk in 1:10)
5  +   vec_seq %<>% c(kk)
6  > for (kk in sample(1:10, 10))
7  +   vec1 %<>% c(kk)
8  > for (kk in sample(1:10, 10))
9  +   vec2 %<>% c(kk)
10 > vec_seq
11 [1] 1 2 3 4 5 6 7 8 9 10
12 > vec1
13 [1] 5 8 1 3 7 6 2 10 4 9
14 > vec2
15 [1] 9 4 7 1 3 2 8 10 5 6
```

Exemple 2 : Opération non commutative (soustraction)

```
1 > Reduce(`-`, 1:10)
2 [1] -53
3 > Reduce(`-`, sample(1:10, 10))
4 [1] -51
5 > Reduce(`-`, sample(1:10, 10))
6 [1] -37
```


Séquentiel vs. concurrent

Exemple 2 : Opération non commutative (soustraction)

```
1 > Reduce(`-`, 1:10)
2 [1] -53
3 > Reduce(`-`, sample(1:10, 10))
4 [1] -51
5 > Reduce(`-`, sample(1:10, 10))
6 [1] -37
```

(contre)Exemple 3 : Opération commutative (addition)

```
1 > Reduce(`+`, 1:10)
2 [1] 55
3 > Reduce(`+`, sample(1:10, 10))
4 [1] 55
5 > Reduce(`+`, sample(1:10, 10))
6 [1] 55
```

En programmation concurrente,

~→ Plusieurs travailleurs exécutent des tâches distinctes.

En programmation concurrente,

- ~> Plusieurs travailleurs exécutent des tâches distinctes.
- ~> L'exécution des tâches nécessite un partage des ressources.

En programmation concurrente,

- ~> Plusieurs travailleurs exécutent des tâches distinctes.
- ~> L'exécution des tâches nécessite un partage des ressources.

Ces deux faits sont la cause d'un grand nombre de problèmes, entre autres

En programmation concurrente,

- ~> Plusieurs travailleurs exécutent des tâches distinctes.
- ~> L'exécution des tâches nécessite un partage des ressources.

Ces deux faits sont la cause d'un grand nombre de problèmes, entre autres

- ~> deadlock,

En programmation concurrente,

- ~> Plusieurs travailleurs exécutent des tâches distinctes.
- ~> L'exécution des tâches nécessite un partage des ressources.

Ces deux faits sont la cause d'un grand nombre de problèmes, entre autres

- ~> deadlock,
- ~> ressource starvation et

En programmation concurrente,

- ~> Plusieurs travailleurs exécutent des tâches distinctes.
- ~> L'exécution des tâches nécessite un partage des ressources.

Ces deux faits sont la cause d'un grand nombre de problèmes, entre autres

- ~> deadlock,
- ~> ressource starvation et
- ~> race condition.

Deadlock

Soit un pool de τ travailleurs t_1, \dots, t_τ .

\rightsquigarrow t_1 veut exécuter une tâche.

Deadlock

Soit un pool de τ travailleurs t_1, \dots, t_τ .

\rightsquigarrow t_1 veut exécuter une tâche.

\rightsquigarrow Pour cela, il a besoin d'une ressource possédée en ce moment par t_2 .

Deadlock

Soit un pool de τ travailleurs t_1, \dots, t_τ .

\rightsquigarrow t_1 veut exécuter une tâche.

\rightsquigarrow Pour cela, il a besoin d'une ressource possédée en ce moment par t_2 .

\rightsquigarrow *Donc, t_1 attend.*

Deadlock

Soit un pool de τ travailleurs t_1, \dots, t_τ .

- ~> t_1 veut exécuter une tâche.
- ~> Pour cela, il a besoin d'une ressource possédée en ce moment par t_2 .
- ~> *Donc, t_1 attend.*
- ~> t_2 va libérer la ressource aussitôt que sa tâche sera accomplie.

Deadlock

Soit un pool de τ travailleurs t_1, \dots, t_τ .

↪ t_1 veut exécuter une tâche.

↪ Pour cela, il a besoin d'une ressource possédée en ce moment par t_2 .

↪ *Donc, t_1 attend.*

↪ t_2 va libérer la ressource aussitôt que sa tâche sera accomplie.

↪ Pour cela, il a besoin d'une ressource possédée en ce moment par t_3 .

Deadlock

Soit un pool de τ travailleurs t_1, \dots, t_τ .

↪ t_1 veut exécuter une tâche.

↪ Pour cela, il a besoin d'une ressource possédée en ce moment par t_2 .

↪ *Donc, t_1 attend.*

↪ t_2 va libérer la ressource aussitôt que sa tâche sera accomplie.

↪ Pour cela, il a besoin d'une ressource possédée en ce moment par t_3 .

↪ *Donc, t_2 attend.*

Deadlock

Soit un pool de τ travailleurs t_1, \dots, t_τ .

↪ t_1 veut exécuter une tâche.

↪ Pour cela, il a besoin d'une ressource possédée en ce moment par t_2 .

↪ *Donc, t_1 attend.*

↪ t_2 va libérer la ressource aussitôt que sa tâche sera accomplie.

↪ Pour cela, il a besoin d'une ressource possédée en ce moment par t_3 .

↪ *Donc, t_2 attend.*

↪ ...

Deadlock

\leadsto Si t_τ a besoin d'une ressource possédée par $t_k, k \neq \tau$, on voit que *le programme ne terminera jamais*.

Deadlock

- ↪ Si t_τ a besoin d'une ressource possédée par $t_k, k \neq \tau$, on voit que *le programme ne terminera jamais*.
- ↪ Il est en situation de *deadlock*.

Deadlock

- ↪ Si t_τ a besoin d'une ressource possédée par $t_k, k \neq \tau$, on voit que *le programme ne terminera jamais*.
- ↪ Il est en situation de *deadlock*.
- ↪ Ce genre d'état peut sembler abstrait et difficile à atteindre.

Deadlock

- ↪ Si t_τ a besoin d'une ressource possédée par t_k , $k \neq \tau$, on voit que *le programme ne terminera jamais*.
- ↪ Il est en situation de *deadlock*.
- ↪ Ce genre d'état peut sembler abstrait et difficile à atteindre.
- ↪ Toutefois, il s'agit exactement de celui de nos philosophes foodies!

Deadlock

- ~> Si t_τ a besoin d'une ressource possédée par t_k , $k \neq \tau$, on voit que *le programme ne terminera jamais*.
- ~> Il est en situation de *deadlock*.
- ~> Ce genre d'état peut sembler abstrait et difficile à atteindre.
- ~> Toutefois, il s'agit exactement de celui de nos philosophes foodies!
- ~> En fait, il s'agit d'un problème très courant.

Deadlock

- ↪ Si t_τ a besoin d'une ressource possédée par t_k , $k \neq \tau$, on voit que *le programme ne terminera jamais*.
- ↪ Il est en situation de *deadlock*.
- ↪ Ce genre d'état peut sembler abstrait et difficile à atteindre.
- ↪ Toutefois, il s'agit exactement de celui de nos philosophes foodies!
- ↪ En fait, il s'agit d'un problème très courant.
- ↪ Pour les programmes complexes, très difficile à prévoir.

Deadlock

- ↪ Si t_τ a besoin d'une ressource possédée par t_k , $k \neq \tau$, on voit que *le programme ne terminera jamais*.
- ↪ Il est en situation de *deadlock*.
- ↪ Ce genre d'état peut sembler abstrait et difficile à atteindre.
- ↪ Toutefois, il s'agit exactement de celui de nos philosophes foodies!
- ↪ En fait, il s'agit d'un problème très courant.
- ↪ Pour les programmes complexes, très difficile à prévoir.
 - ↪ Algorithme de l'autruche.

Deadlock

- ↪ Si t_τ a besoin d'une ressource possédée par t_k , $k \neq \tau$, on voit que *le programme ne terminera jamais*.
- ↪ Il est en situation de *deadlock*.
- ↪ Ce genre d'état peut sembler abstrait et difficile à atteindre.
- ↪ Toutefois, il s'agit exactement de celui de nos philosophes foodies!
- ↪ En fait, il s'agit d'un problème très courant.
- ↪ Pour les programmes complexes, très difficile à prévoir.
 - ↪ Algorithme de l'autruche.
 - ↪ Miser sur la détection.

Race condition

↪ Survient lorsque le comportement d'un programme dépend de l'ordre d'exécution des instructions.

Race condition

- ~→ Survient lorsque le comportement d'un programme dépend de l'ordre d'exécution des instructions.
- ~→ Si le programme ne contrôle pas cet ordre, le comportement est imprévisible.

Race condition

- ~> Survient lorsque le comportement d'un programme dépend de l'ordre d'exécution des instructions.
- ~> Si le programme ne contrôle pas cet ordre, le comportement est imprévisible.
- ~> Cela est généralement le cas en programmation concurrente.

Race condition

- ~> Survient lorsque le comportement d'un programme dépend de l'ordre d'exécution des instructions.
- ~> Si le programme ne contrôle pas cet ordre, le comportement est imprévisible.
- ~> Cela est généralement le cas en programmation concurrente.
- ~> Souvent, survient lors de l'application d'une opération non associative ou non commutatives à un ensemble d'éléments.

Race condition

- ~> Survient lorsque le comportement d'un programme dépend de l'ordre d'exécution des instructions.
- ~> Si le programme ne contrôle pas cet ordre, le comportement est imprévisible.
- ~> Cela est généralement le cas en programmation concurrente.
- ~> Souvent, survient lors de l'application d'une opération non associative ou non commutatives à un ensemble d'éléments.
- ~> Problème rencontré dans l'exemple du calcul des différences.

Ressource starvation

↪ Survient lorsqu'il existe une probabilité non nulle qu'un travailleur ne puisse accéder à une ressource dont il a besoin.

Ressource starvation

- ~> Survient lorsqu'il existe une probabilité non nulle qu'un travailleur ne puisse accéder à une ressource dont il a besoin.
- ~> Généralement, l'erreur se situe plutôt au niveau du scheduler.

Ressource starvation

- ↪ Survient lorsqu'il existe une probabilité non nulle qu'un travailleur ne puisse accéder à une ressource dont il a besoin.
- ↪ Généralement, l'erreur se situe plutôt au niveau du scheduler.
 - ↪ Les ressources ne sont pas distribuées équitablement.

Ressource starvation

- ~> Survient lorsqu'il existe une probabilité non nulle qu'un travailleur ne puisse accéder à une ressource dont il a besoin.
- ~> Généralement, l'erreur se situe plutôt au niveau du scheduler.
 - ~> Les ressources ne sont pas distribuées équitablement.
- ~> Peut aussi survenir si l'ensemble des ressources sont utilisée.

Ressource starvation

- ~> Survient lorsqu'il existe une probabilité non nulle qu'un travailleur ne puisse accéder à une ressource dont il a besoin.
- ~> Généralement, l'erreur se situe plutôt au niveau du scheduler.
 - ~> Les ressources ne sont pas distribuées équitablement.
- ~> Peut aussi survenir si l'ensemble des ressources sont utilisée.
 - ~> Les travailleurs ne libère pas les ressources après les avoir utilisé.

Ressource starvation

- ~> Survient lorsqu'il existe une probabilité non nulle qu'un travailleur ne puisse accéder à une ressource dont il a besoin.
- ~> Généralement, l'erreur se situe plutôt au niveau du scheduler.
 - ~> Les ressources ne sont pas distribuées équitablement.
- ~> Peut aussi survenir si l'ensemble des ressources sont utilisée.
 - ~> Les travailleurs ne libère pas les ressources après les avoir utilisé.
 - ~> Fork bomb.

Ressource starvation

- ~> Survient lorsqu'il existe une probabilité non nulle qu'un travailleur ne puisse accéder à une ressource dont il a besoin.
- ~> Généralement, l'erreur se situe plutôt au niveau du scheduler.
 - ~> Les ressources ne sont pas distribuées équitablement.
- ~> Peut aussi survenir si l'ensemble des ressources sont utilisée.
 - ~> Les travailleurs ne libère pas les ressources après les avoir utilisé.
 - ~> Fork bomb.
- ~> Exemple : enlever toutes les baguettes aux philosophes!

Programmation concurrente en R

Idée générale

↪ La programmation concurrente permet plusieurs calculs simultanés exécutés par des *équipes* de *travailleurs*.

Idée générale

- ~> La programmation concurrente permet plusieurs calculs simultanés exécutés par des *équipes* de *travailleurs*.
- ~> Potentiellement, gain en performance.

Idée générale

- ~> La programmation concurrente permet plusieurs calculs simultanés exécutés par des *équipes* de *travailleurs*.
- ~> Potentiellement, gain en performance.
- ~> Toutefois, cela a un prix.

Idée générale

- ~> La programmation concurrente permet plusieurs calculs simultanés exécutés par des *équipes* de *travailleurs*.
- ~> Potentiellement, gain en performance.
- ~> Toutefois, cela a un prix.
- ~> Les travailleurs partagent des ressources.

Idée générale

- ~> La programmation concurrente permet plusieurs calculs simultanés exécutés par des *équipes* de *travailleurs*.
- ~> Potentiellement, gain en performance.
- ~> Toutefois, cela a un prix.
- ~> Les travailleurs partagent des ressources.
- ~> Nécessité de communiquer entre eux pour éviter les problèmes inhérent au partage.

Idée générale

- ~> La programmation concurrente permet plusieurs calculs simultanés exécutés par des *équipes* de *travailleurs*.
- ~> Potentiellement, gain en performance.
- ~> Toutefois, cela a un prix.
- ~> Les travailleurs partagent des ressources.
- ~> Nécessité de communiquer entre eux pour éviter les problèmes inhérent au partage.
- ~> Pour un ordinateur,

Idée générale

- ~> La programmation concurrente permet plusieurs calculs simultanés exécutés par des *équipes* de *travailleurs*.
- ~> Potentiellement, gain en performance.
- ~> Toutefois, cela a un prix.
- ~> Les travailleurs partagent des ressources.
- ~> Nécessité de communiquer entre eux pour éviter les problèmes inhérent au partage.
- ~> Pour un ordinateur,
 - ~> calcul : rapide

Idée générale

- ~> La programmation concurrente permet plusieurs calculs simultanés exécutés par des *équipes* de *travailleurs*.
- ~> Potentiellement, gain en performance.
- ~> Toutefois, cela a un prix.
- ~> Les travailleurs partagent des ressources.
- ~> Nécessité de communiquer entre eux pour éviter les problèmes inhérent au partage.
- ~> Pour un ordinateur,
 - ~> calcul : rapide mais
 - ~> communication : lente.

Idée générale

- ~> La programmation concurrente permet plusieurs calculs simultanés exécutés par des *équipes* de *travailleurs*.
- ~> Potentiellement, gain en performance.
- ~> Toutefois, cela a un prix.
- ~> Les travailleurs partagent des ressources.
- ~> Nécessité de communiquer entre eux pour éviter les problèmes inhérent au partage.
- ~> Pour un ordinateur,
 - ~> calcul : rapide mais
 - ~> communication : lente.
- ~> Plus de travailleurs \Rightarrow plus de calculs

Idée générale

- ~> La programmation concurrente permet plusieurs calculs simultanés exécutés par des *équipes* de *travailleurs*.
- ~> Potentiellement, gain en performance.
- ~> Toutefois, cela a un prix.
- ~> Les travailleurs partagent des ressources.
- ~> Nécessité de communiquer entre eux pour éviter les problèmes inhérent au partage.
- ~> Pour un ordinateur,
 - ~> calcul : rapide mais
 - ~> communication : lente.
- ~> Plus de travailleurs \Rightarrow plus de calculs :)

Idée générale

- ~> La programmation concurrente permet plusieurs calculs simultanés exécutés par des *équipes* de *travailleurs*.
- ~> Potentiellement, gain en performance.
- ~> Toutefois, cela a un prix.
- ~> Les travailleurs partagent des ressources.
- ~> Nécessité de communiquer entre eux pour éviter les problèmes inhérent au partage.
- ~> Pour un ordinateur,
 - ~> calcul : rapide mais
 - ~> communication : lente.
- ~> Plus de travailleurs \Rightarrow plus de calculs :)
- ~> Plus de travailleurs \Rightarrow plus de communication

Idée générale

- ~> La programmation concurrente permet plusieurs calculs simultanés exécutés par des *équipes* de *travailleurs*.
- ~> Potentiellement, gain en performance.
- ~> Toutefois, cela a un prix.
- ~> Les travailleurs partagent des ressources.
- ~> Nécessité de communiquer entre eux pour éviter les problèmes inhérent au partage.
- ~> Pour un ordinateur,
 - ~> calcul : rapide mais
 - ~> communication : lente.
- ~> Plus de travailleurs \Rightarrow plus de calculs :)
- ~> Plus de travailleurs \Rightarrow plus de communication :(

~→ En conséquence, on tente de réduire la surcharge associée à la communication au maximum.

Idée générale

- ~> En conséquence, on tente de réduire la surcharge associée à la communication au maximum.
- ~> *Très difficile* dans un langage de haut niveau tel que R.

Idée générale

- ~> En conséquence, on tente de réduire la surcharge associée à la communication au maximum.
- ~> *Très difficile* dans un langage de haut niveau tel que R.
- ~> Il faut donc modérer nos attentes.

- ~> En conséquence, on tente de réduire la surcharge associée à la communication au maximum.
- ~> *Très difficile* dans un langage de haut niveau tel que R.
- ~> Il faut donc modérer nos attentes.
- ~> Si la performance est nécessaire, possibilité d'appeler des bibliothèques C/C++ ou Fortran.

- ~> En conséquence, on tente de réduire la surcharge associée à la communication au maximum.
- ~> *Très difficile* dans un langage de haut niveau tel que R.
- ~> Il faut donc modérer nos attentes.
- ~> Si la performance est nécessaire, possibilité d'appeler des bibliothèques C/C++ ou Fortran.
 - ~> OpenMP

- ~> En conséquence, on tente de réduire la surcharge associée à la communication au maximum.
- ~> *Très difficile* dans un langage de haut niveau tel que R.
- ~> Il faut donc modérer nos attentes.
- ~> Si la performance est nécessaire, possibilité d'appeler des bibliothèques C/C++ ou Fortran.
 - ~> OpenMP
 - ~> Armadillo

- ~> En conséquence, on tente de réduire la surcharge associée à la communication au maximum.
- ~> *Très difficile* dans un langage de haut niveau tel que R.
- ~> Il faut donc modérer nos attentes.
- ~> Si la performance est nécessaire, possibilité d'appeler des bibliothèques C/C++ ou Fortran.
 - ~> OpenMP
 - ~> Armadillo
 - ~> MPI (mémoire distribuée, ex. Calcul Québec)

⇒ R est livré avec la librairie `parallel`.

- ~> R est livré avec la librairie `parallel`.
- ~> Fourni des fonctions telles que `parApply`, `parLapply`, `parSapply`.

- ~> R est livré avec la librairie `parallel`.
- ~> Fourni des fonctions telles que `parApply`, `parLapply`, `parSapply`.
- ~> L'implémentation est relativement simple.

- ~> R est livré avec la librairie **parallel**.
- ~> Fourni des fonctions telles que **parApply**, **parLapply**, **parSapply**.
- ~> L'implémentation est relativement simple.
 - ~> On construit une équipe.

- ~> R est livré avec la librairie **parallel**.
- ~> Fourni des fonctions telles que **parApply**, **parLapply**, **parSapply**.
- ~> L'implémentation est relativement simple.
 - ~> On construit une équipe.
 - ~> Le travail est réparti entre les travailleurs

- ~> R est livré avec la librairie **parallel**.
- ~> Fourni des fonctions telles que **parApply**, **parLapply**, **parSapply**.
- ~> L'implémentation est relativement simple.
 - ~> On construit une équipe.
 - ~> Le travail est réparti entre les travailleurs
 - ~> Une session R est lancée pour chaque travailleur.

- ~> R est livré avec la librairie **parallel**.
- ~> Fourni des fonctions telles que **parApply**, **parLapply**, **parSapply**.
- ~> L'implémentation est relativement simple.
 - ~> On construit une équipe.
 - ~> Le travail est réparti entre les travailleurs
 - ~> Une session R est lancée pour chaque travailleur.
 - ~> Lorsque tout le travail est fini, R réunit les résultats.

parallel

```
1  cl <- makeCluster(detectCores())
2  clusterExport(cl, "sims1")
3  vario1 <- parApply(cl, sims1 %>% as.tibble, 2, function(x){
4      geoR::variog(geoR::as.geodata(cbind(sp::coordinates(sims1),
        ↪      x))),
5          max.dist = 0.8,
6          option = "bin",
7          messages = FALSE)$v
8  }) %>% t
9  stopCluster(cl)
```

Possibilité d'exécuter du code arbitraire à l'aide de `clusterEvalQ` (ex.

```
clusterEvalQ(cl, {library(spatstat); set.seed(42)}))
```

R règle les problème discutés auparavant.

Deadlock

Chaque travailleur possède sa propre copie des ressources (1 session par travailleur).

R règle les problème discutés auparavant.

Deadlock

Chaque travailleur possède sa propre copie des ressources (1 session par travailleur).

Race condition

Indépendance entre les “itérations” (application).

R règle les problème discutés auparavant.

Deadlock

Chaque travailleur possède sa propre copie des ressources (1 session par travailleur).

Race condition

Indépendance entre les “itérations” (application).

Ressource starving

Malheureusement, toujours une possibilité!

~> Le paquet foreach fournit une interface simple au calcul parallèle.

- ~> Le paquet foreach fournit une interface simple au calcul parallèle.
- ~> doParallel permet à foreach d'utiliser la librairie parallel.

- ~> Le paquet foreach fournit une interface simple au calcul parallèle.
- ~> doParallel permet à foreach d'utiliser la librairie parallel.
- ~> L'interface est simplifiée. Entre autres :

- ~> Le paquet foreach fournit une interface simple au calcul parallèle.
- ~> doParallel permet à foreach d'utiliser la librairie parallel.
- ~> L'interface est simplifiée. Entre autres :
 - .packages :**
librairies à exporter.

- ~> Le paquet foreach fournit une interface simple au calcul parallèle.
- ~> doParallel permet à foreach d'utiliser la librairie parallel.
- ~> L'interface est simplifiée. Entre autres :
 - .packages :**
librairies à exporter.
 - .export :**
variables à exporter.

- ~> Le paquet foreach fournit une interface simple au calcul parallèle.
- ~> doParallel permet à foreach d'utiliser la librairie parallel.
- ~> L'interface est simplifiée. Entre autres :
 - .packages :**
librairies à exporter.
 - .export :**
variables à exporter.
- ~> Autres fonctionnalités intéressantes (ex. listes en compréhension).

Example

```
1  lmSlow <- function(m){
2      res <- matrix(nrow = m, ncol = 21)
3
4      for (kk in 1:m){
5          random_values <- rnorm(1e6)
6          X <- matrix(random_values, ncol = 20)
7          y <- rnorm(5e4)
8          reg <- lm(y ~ X)
9          res[kk,] <- coef(reg)
10     }
11
12     res
13 }
```


Example

```
1  library(doParallel)
2  registerDoParallel(cores = detectCores())
3  lmPar <- function(m){
4      foreach(kk = 1:m, .inorder = FALSE,
5              .combine = rbind, .multicombine = TRUE) %dopar%{
6          random_values <- rnorm(1e6)
7          X <- matrix(random_values, ncol = 20)
8          y <- rnorm(5e4)
9          reg <- lm(y ~ X)
10         coef(reg)
11     }
12 }
```

Exemple

```
1 > microbenchmark(lmSlow(20), lmPar(20), times = 5)
2 Unit: seconds
3      expr   min    lq mean median    uq   max neval
4  lmSlow(20) 3.38 3.40 3.42   3.40 3.46 3.47     5
5  lmPar(20)  2.28 2.34 2.38   2.35 2.38 2.54     5
```

En utilisant 4 coeurs, on obtient un speedup d'environ 1.45...

Conclusion

~→ La programmation concurrente est très intéressante d'un point de vue performance.

Conclusion

- ~→ La programmation concurrente est très intéressante d'un point de vue performance.
- ~→ Elle vient avec son lot de problèmes.

Conclusion

- ~> La programmation concurrente est très intéressante d'un point de vue performance.
- ~> Elle vient avec son lot de problèmes.
- ~> R permet dans une certaine mesure l'exécution de calculs en parallèle.

Conclusion

- ~> La programmation concurrente est très intéressante d'un point de vue performance.
- ~> Elle vient avec son lot de problèmes.
- ~> R permet dans une certaine mesure l'exécution de calculs en parallèle.
- ~> Malheureusement, les gains en performances sont souvent limités.

Conclusion

- ~> La programmation concurrente est très intéressante d'un point de vue performance.
- ~> Elle vient avec son lot de problèmes.
- ~> R permet dans une certaine mesure l'exécution de calculs en parallèle.
- ~> Malheureusement, les gains en performances sont souvent limités.
- ~> Toutefois, le code déjà vectorisé se parallélise relativement aisément.

Conclusion

- ~> La programmation concurrente est très intéressante d'un point de vue performance.
- ~> Elle vient avec son lot de problèmes.
- ~> R permet dans une certaine mesure l'exécution de calculs en parallèle.
- ~> Malheureusement, les gains en performances sont souvent limités.
- ~> Toutefois, le code déjà vectorisé se parallélise relativement aisément.
- ~> Cela peut valoir le coup d'essayer!

Références

- [1] Edsger W. DIJKSTRA. “Hierarchical Ordering of Sequential Processes”. en. In : *The Origin of Concurrent Programming : From Semaphores to Remote Procedure Calls*. Sous la dir. de Per Brinch HANSEN. New York, NY : Springer New York, 2002, p. 198-227. ISBN : 9781475734720. DOI : [10.1007/978-1-4757-3472-0_5](https://doi.org/10.1007/978-1-4757-3472-0_5). URL : https://doi.org/10.1007/978-1-4757-3472-0_5.