Introduction au parallélisme multitâche par directives OpenMP

Master DFE – année 2020/2021

Mathieu Lobet, Maison de la Simulation
Mathieu.lobet@cea.fr

Introduction au parallélisme multitâche par directives OpenMP

1) Description de l'approche

Cours et matériel supplémentaires sur internet

Le cours de l'IDRIS est à mon sens le plus complet en français et anglais:
http://www.idris.fr/formations/openmp/

Le site du consortium fournit une documentation en ligne complète :
https://www.openmp.org/resources/refguides/

Le guide de référence résume très bien l'ensemble des options disponibles :
https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf

Ce cours n'est qu'une introduction très superficielle à OpenMP

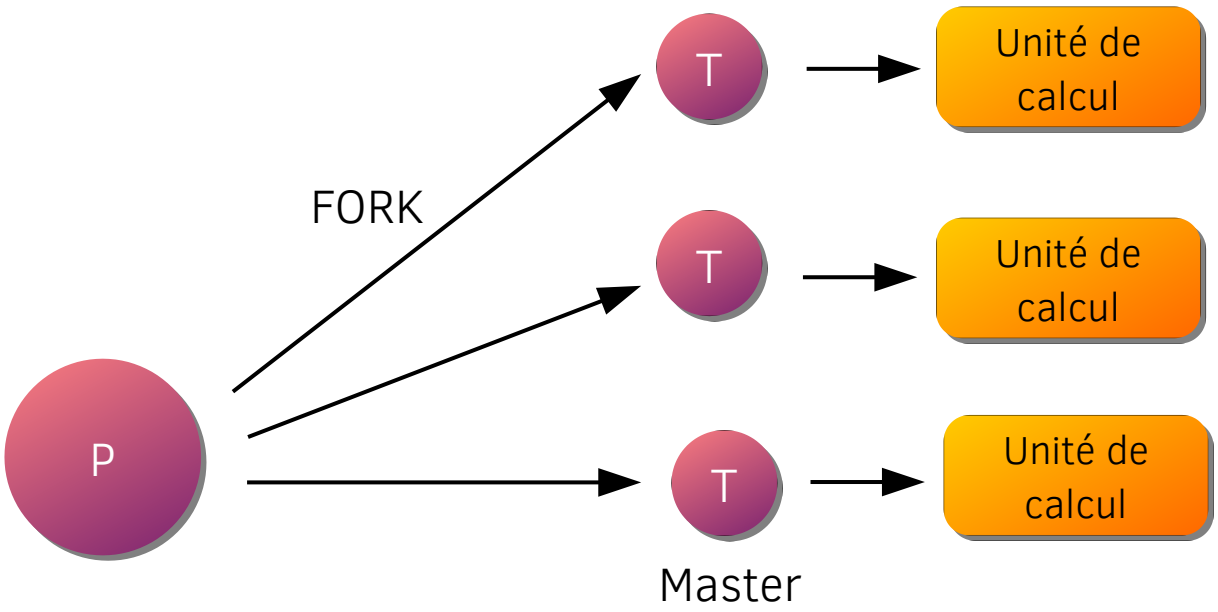
Fonctionnement d'un modèle multitâche *fork-join*

Multithreaded model

Démarrage séquentiel du programme

Fonctionnement d'un modèle multitâche *fork-join*

Multithreaded model

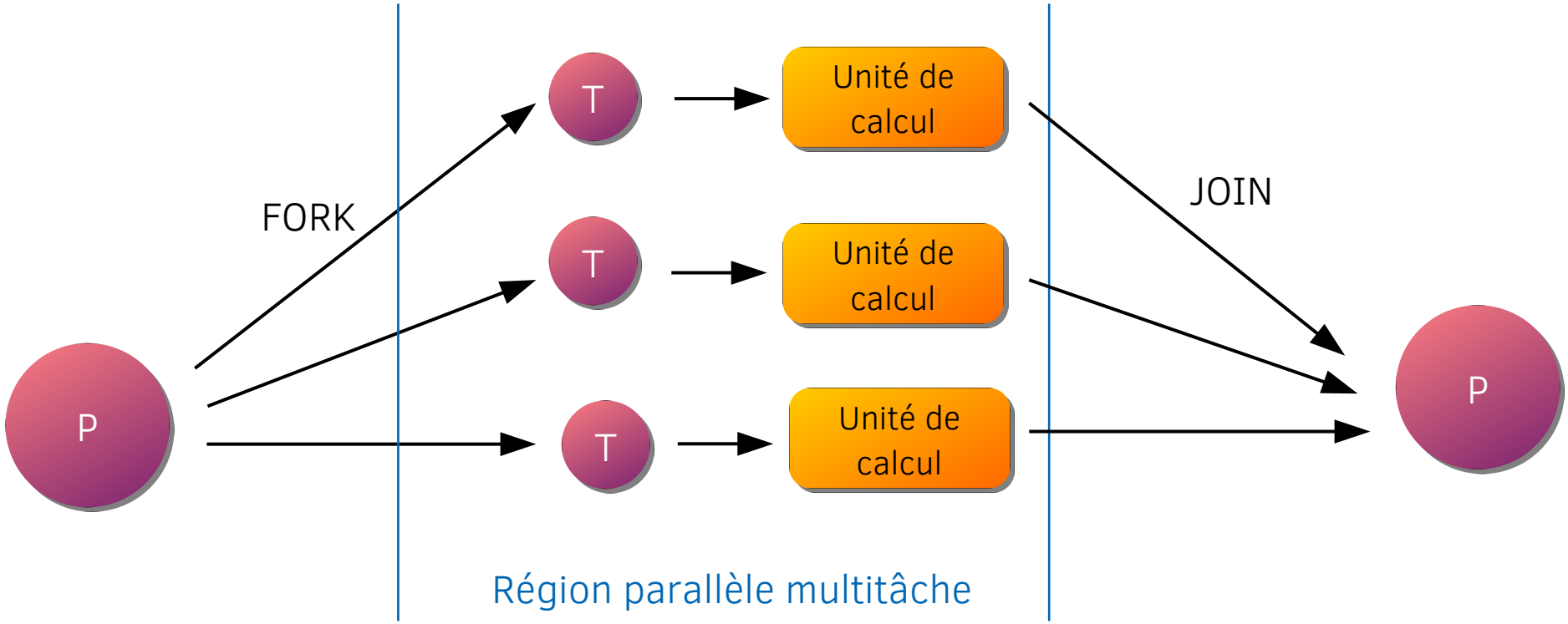


Le processus principal génère des threads secondaires (*slave*) pour résoudre un problème en parallèle (boucle par exemple).

Le processus initial devient le thread maître (*master*).

Fonctionnement d'un modèle multitâche *fork-join*

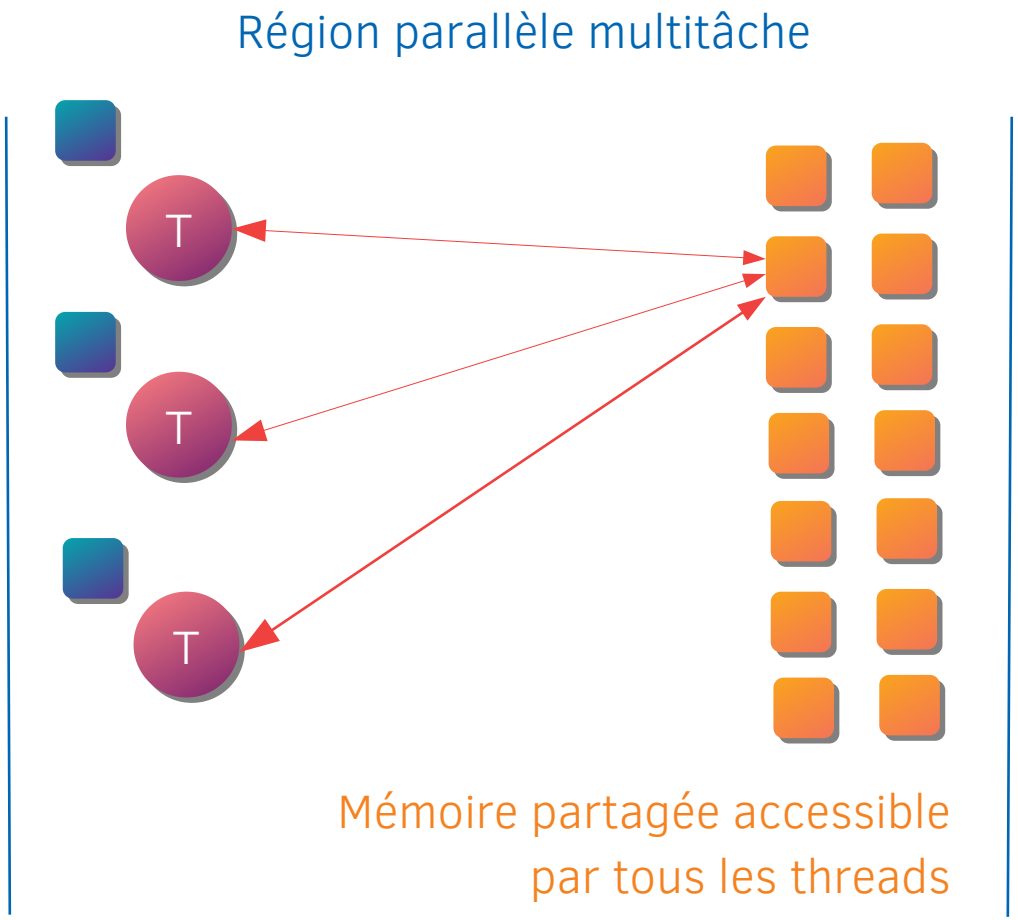
Multithreaded model



Fin de la région parallèle, les threads secondaires sont détruits et le programme redevient séquentiel.

Fonctionnement d'un modèle multitâche *fork-join*

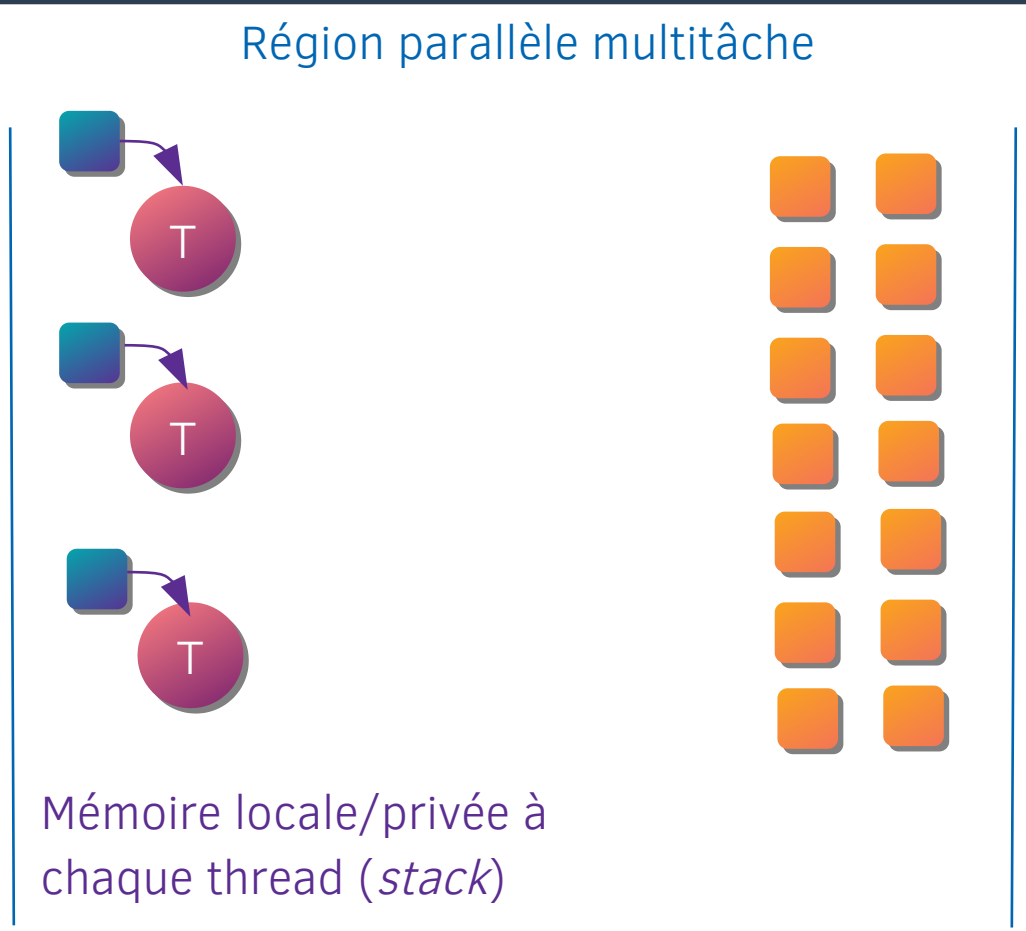
Multithreaded model



La mémoire est partagée par défaut entre tous les threads mais il existe une mémoire privée (déclaration locales de certaines variables) appelée *stack*.

Fonctionnement d'un modèle multitâche *fork-join*

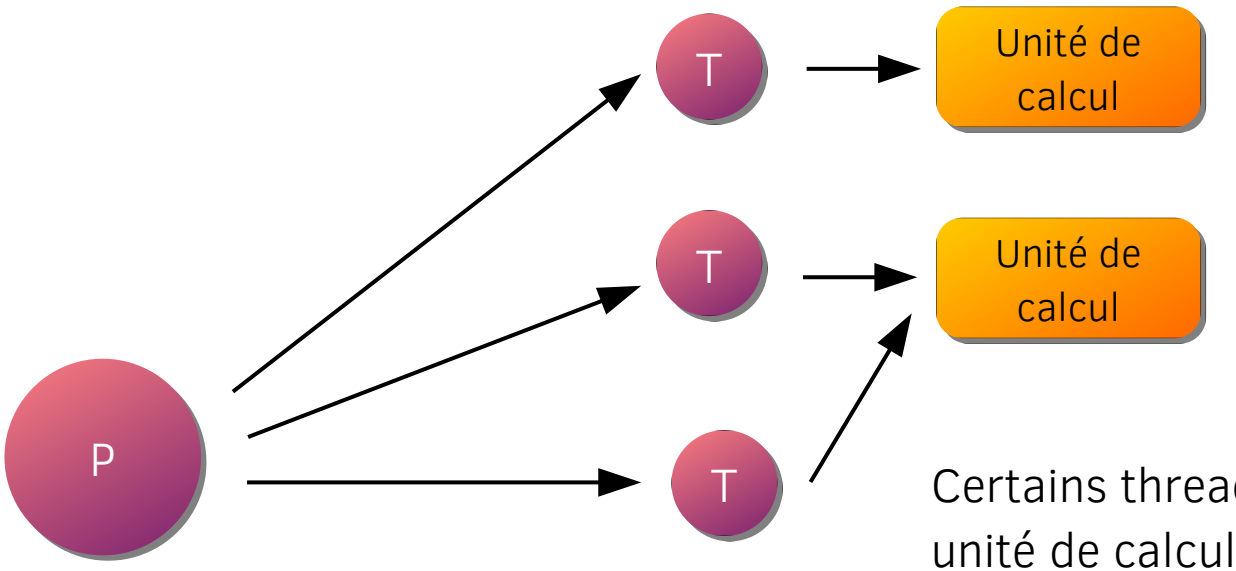
Multithreaded model



La mémoire est partagée par défaut entre tous les threads mais il existe une mémoire privée (déclaration locales de certaines variables) appelée *stack*.

Fonctionnement d'un modèle multitâche *fork-join*

Multithreaded model



Certains threads ici partagent la même unité de calcul

Le nombre de thread OpenMP est indépendant du nombre d'unité de calcul (on parle de thread logique et non de thread physique).

Les threads OpenMP sont répartis entre les unités de calcul réelles.

Principe de la programmation par directives

Une directive est une ligne de code compréhensible par le compilateur qui va aiguiller la compilation pour un bloc de code dans le langage utilisé.

.f90

```
Mon code fortran…

!$omp … (directive OpenMP en Fortran)

La suite de mon code…
```

.cpp

```
Mon code C++...

#pragma omp … (directive OpenMP en C++)

La suite de mon code…
```

Bibliothèque de fonctions

OpenMP fournit également une bibliothèque de fonctions appelables directement depuis le code

.f90

```
Use omp_lib

Mon code fortran…

Call fonction_openmp()...

La suite de mon code…
```

.cpp

```
#include « omp.h »

Mon code fortran…

fonction_openmp()...

La suite de mon code…
```

Où trouver OpenMP ?

La plupart des compilateurs récents (GNU, INTEL, LLVM...) possèdent une implémentation d'OpenMP installée par défaut.

Attention tout de même, les implémentations Fortran sont souvent en retard sur les implémentations C/C++.

Compilation d'un programme OpenMP

La compilation nécessite simplement l'ajout d'un paramètre à la ligne de compilation classique.

Par exemple, pour compiler avec gfortran en activant OpenMP :

```
> gfortran -fopenmp -O3 program.f90 -o executable
```

Par exemple, pour compiler avec g++ en activant OpenMP :

```
> g++ -fopenmp -O3 program.f90 -o executable
```

Attention, la syntaxe du paramètre dépend du compilateur, par exemple `-qopenmp` avec Intel

Exécution d'un programme OpenMP

L'exécution se fait comme un programme classique mais des paramètres d'environnement sont à spécifier pour contraindre son déroulement (si non spécifiés dans le code).

Par exemple, pour spécifier le nombre de threads dans les régions parallèles

```
> export OMP_NUM_THREADS=4
> ./executable
```

Par exemple, pour spécifier le type ordonnanceur lors de la parallélisation des boucles

```
> export OMP_SCHEDULER=DYNAMIC
> ./executable
```

Introduction au parallélisme multitâche par directives OpenMP

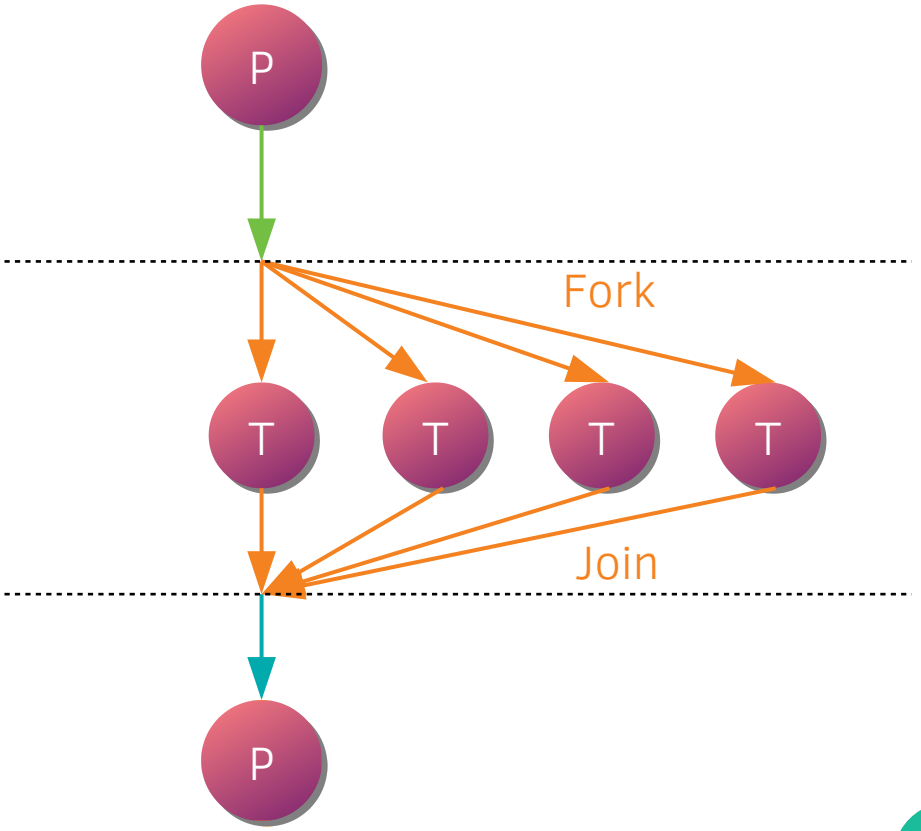
2) Ouvrir une région parallèle et gérer la mémoire

Création d'une région parallèle `!$ omp parallel`

Cette directive permet de spécifier la création d'une région parallèle : Le code présent dans la région parallèle est exécuté par chaque thread.

.f90

```
Program openmp

Implicit none

… partie séquentielle

!$omp parallel

… partie parallèle

!$omp end parallel

… partie séquentielle

End program
```

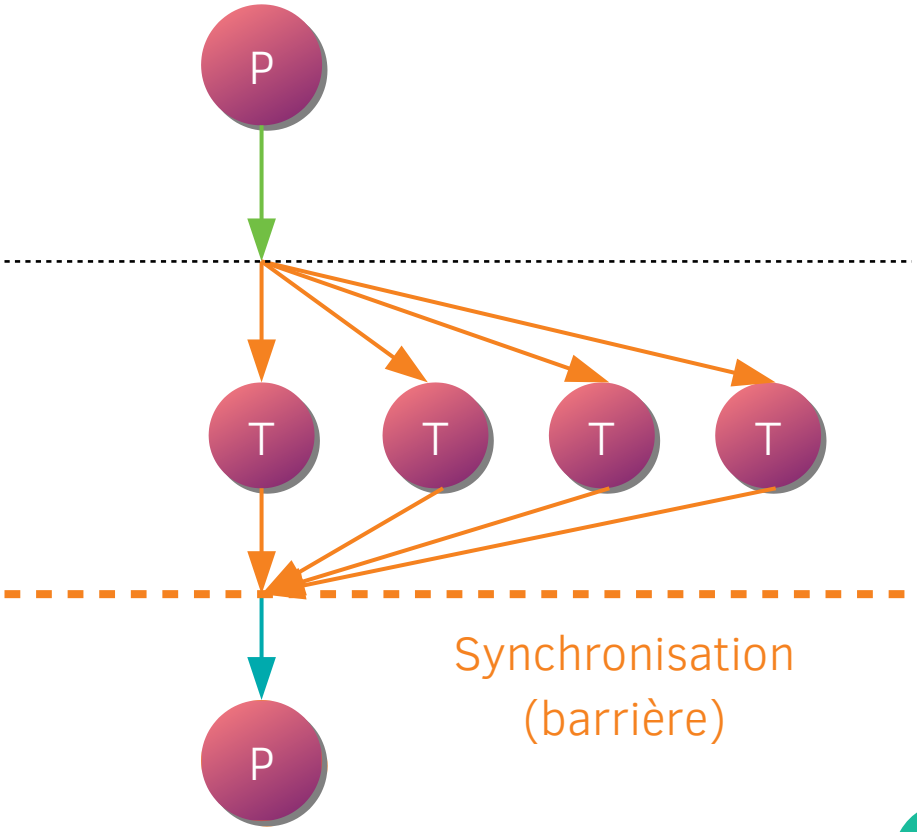


Création d'une région parallèle `!$ omp parallel`

Au moment de la fermeture de la région parallèle (`!$omp end parallel`), il y a une synchronisation de tous les threads (attente que chacun a fini son travail)

.f90

```
Program openmp

Implicit none

… partie séquentielle


!$omp parallel

… partie parallèle

!$omp end parallel


… partie séquentielle

End program
```



Notion de variable partagée
shared

Les variables déclarées avant la région parallèle sont par défaut partagées par les threads.

Une variable partagée ne peut pas être modifiée par tous les threads en même temps au risque d'avoir de la concurrence sur l'écriture en mémoire et d'obtenir une valeur aléatoire.


.f90

```
Integer :: A

!$omp parallel

… région parallèle
… la même variable A est partagée par les threads par défaut

!$omp end parallel
```

Notion de variable partagée
shared

Les variables déclarées avant la région parallèle sont par défaut partagées par les threads.

Une variable partagée ne peut pas être modifiée par tous les threads en même temps au risque d'avoir de la concurrence sur l'écriture en mémoire et d'obtenir une valeur aléatoire.

.cpp

```
int A ;

#pragma omp parallel {

… région parallèle
… la même variable A est partagée par les threads par défaut

}
```

Notion de variable partagée
shared

Il est tout de même conseillé de spécifier le comportement par défaut grâce à la clause **`default`**.
Pour ne plus avoir de comportement par défaut : **`default(none)`**


.f90

```
Integer :: A

!$omp parallel default(shared)

… région parallèle
… la même variable A est partagée par les threads explicitement

!$omp end parallel
```

Notion de variable privée
private

Lorsqu'une variable prendra des valeurs différentes pour chaque thread (utilisation locale/privée). Il est préférable de la déclarer comme une valeur privée en utilisant la clause `private`.

.f90

```
Program openmp

Implicit none

Integer :: A, B

!$omp parallel default(none) shared(A) private(B)

… partie parallèle
… la même variable A est partagée par les threads
… la variable B est dupliquée pour chaque thread (dans la stack)

!$omp end parallel

End program
```

Notion de variable privée
private

Lorsqu'une variable prendra des valeurs différentes pour chaque thread (utilisation locale/privée). Il est préférable de la déclarer comme une valeur privée en utilisant la clause `private`.

.cpp

```
int A, B ;

#pragma omp parallel default(none) shared(A) private(B) {

… partie parallèle
… la même variable A est partagée par les threads
… la variable B est dupliquée pour chaque thread (dans la stack)

}
```

Notion de variable privée
private

La valeur d'une variable privée initialisée et définie avant une région parallèle n'est pas copiée par défaut dans les versions locales aux threads

.f90

```
Program openmp

Implicit none

Integer :: A, B

B = 5

!$omp parallel default(none) shared(A) private(B)

… B existe mais n'a pas été définie (valeur inconnue)

!$omp end parallel

End program
```

Notion de variable privée
private

La valeur d'une variable privée initialisée et définie avant une région parallèle n'est pas copiée par défaut dans les versions locales aux threads

.cpp

```
int A, B ;

B = 5 ;

#pragma omp parallel default(none) shared(A) private(B) {

… B existe mais n'a pas été définie (valeur inconnue)

}
```

Variable privée : récupérer la valeur définie dans la région séquentielle *firstprivate*

La clause **`firstprivate`** permet d'initialiser une variable privée avec la valeur définie dans la zone séquentielle.

.f90

```
Integer :: A, B

B = 5

!$omp parallel default(none) shared(A) firstprivate(B)

… B existe localement pour chaque thread avec la valeur 5 par défaut

!$omp end parallel
```

Variable privée : récupérer la valeur définie dans la région séquentielle *firstprivate*

La clause **`firstprivate`** permet d'initialiser une variable privée avec la valeur définie dans la zone séquentielle.

.cpp

```
int A, B ;

B = 5 ;

#pragma omp parallel default(none) shared(A) firstprivate(B) {

… B existe localement pour chaque thread avec la valeur 5 par défaut

}
```

Variable privée : récupérer la valeur d'une variable privée à la fin d'une région parallèle DO
lastprivate

La clause **lastprivate** permet de conserver le dernier résultat enregistré dans une variable privée en sortie de région parallèle DO.


.f90

```
Integer :: A, B

B = 5

!$omp parallel do default(none) shared(A) lastprivate(B)

DO i=1,N

… B existe localement pour chaque thread

B = i

ENDDO

!$omp end parallel do

… B possède la valeur N
```

Variable privée : récupérer la valeur d'une variable privée à la fin d'une région parallèle for
lastprivate

La clause **lastprivate** permet de conserver le dernier résultat enregistré dans une variable privée en sortie de région parallèle for.


.cpp

```
int A, B ;

B = 5 ;

!$omp parallel for default(none) shared(A) lastprivate(B)

For(int i=0 ; i< N ; i++) {

  // B existe localement pour chaque thread

  B = i

}

!$omp end parallel do

// B possède la valeur N
```

Introduction au parallélisme multitâche par directives OpenMP

3) Fonctions de la bibliothèque

Fonctions de la bibliothèque OpenMP
Runtime library routines

Il existe un grand nombre de fonctions permettant de récupérer des informations sur l'environnement d'exécution ou de modifier cet environnement.

Il est nécessaire d'inclure le header openmp pour les utiliser :

.f90

```
Use omp_lib
```

.cpp

```
#include « omp.h »
```

`omp_get_num_threads` : récupérer le nombre de threads

`omp_get_num_threads` renvoie sous forme d'un entier le nombre de threads utilisé dans les régions parallèles (par exemple spécifié par la variable d'environnement `OMP_NUM_THREADS`)

.f90

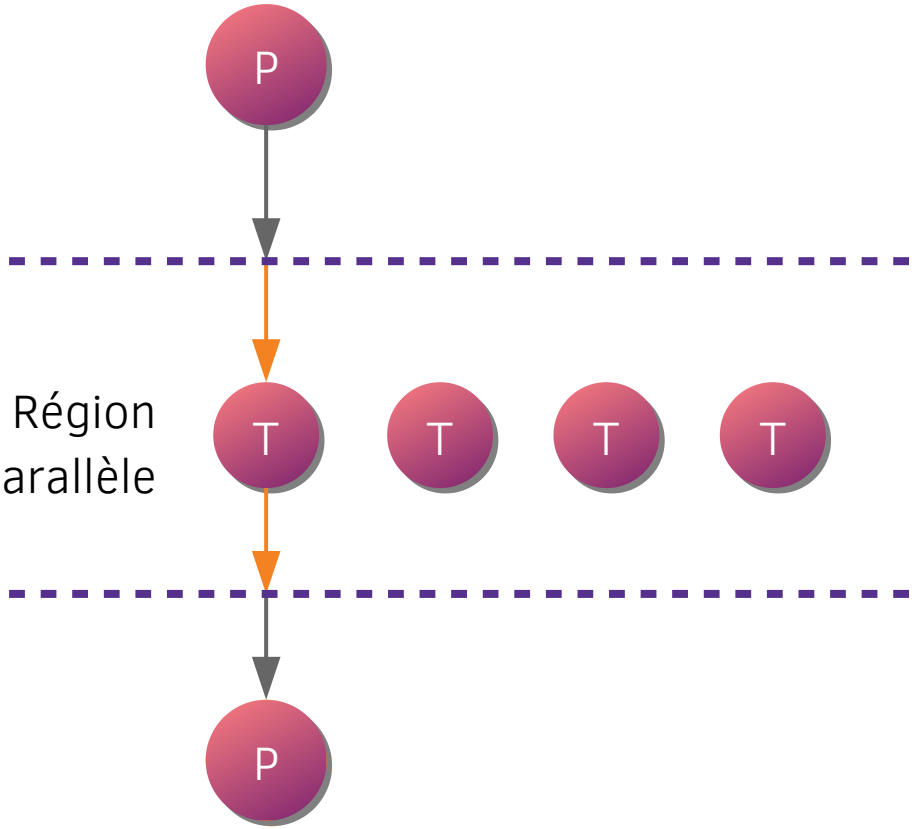
```
Integer ::  num_threads
num_threads = omp_get_num_threads()
```

.cpp

```
Int num_threads = omp_get_num_threads() ;
```

`omp_get_num_threads` : récupérer le nombre de threads

`omp_get_num_threads` renvoie sous forme d'un entier le nombre de threads utilisé dans les régions parallèles (par exemple spécifié par la variable d'environnement `OMP_NUM_THREADS`)



`omp_get_num_threads` renverra 4 peu importe qui appelle cette fonction.

`Int omp_get_thread_num` : récupérer l'identifiant du thread appelant

`omp_get_thread_num` renvoie sous forme d'un entier l'identifiant du thread qui appelle cette fonction. Cette fonction n'a de sens que dans la région parallèle.

.f90

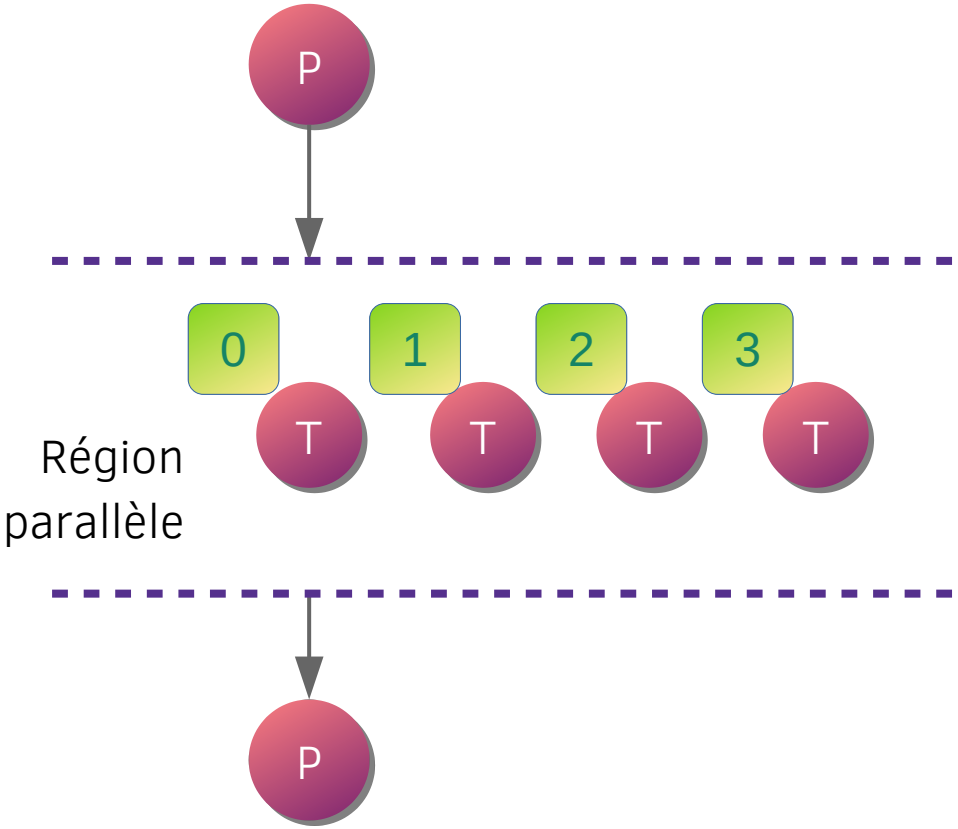
```
Integer ::  thread_id
thread_id = omp_get_thread_num()
```

.cpp

```
Int thread_id = omp_get_thread_num() ;
```

`Int omp_get_thread_num` : récupérer l'identifiant du thread appelant

`omp_get_thread_num` renvoie sous forme d'un entier l'identifiant du thread qui appelle cette fonction. Cette fonction n'a de sens que dans la région parallèle.



`omp_get_thread_num` renverra en fonction du thread appelant l'identifiant 0, 1, 2 ou 3.

Cela permet des traitements différenciés entre thread (print, écriture...)

`Double omp_get_wtime(void)` : récupérer le temps avec OpenMP

omp_get_wtime renvoie le temps d'exécution depuis une date arbitraire fixe pour chaque thread.

.f90

```
real(8) ::  time
time = omp_get_wtime()
```

.cpp

```
double time = omp_get_wtime() ;
```

Par sécurité, il est conseillé d'utiliser ces fonctions pour gérer le temps dans un code purement OpenMP.

Fonctions de la bibliothèque OpenMP
Runtime library routines

Il en existe d'autres qui sont toutes décrites dans la documentation OpenMP et notamment la fiche simplifiée.

Exercice n°1 : votre premier programme OpenMP

- Rendez vous sur le Gitlab des exercices : https://gitlab.maisondelasimulation.fr/mlobet/cours-hpc-m2-dfe
- Télécharger les exercices sur votre session de travail
- Décompressez l'archive en ligne de commande

```
> tar xvf archivedossier.tar
```

- Rendez vous dans le dossier de l'exercice n°1 OpenMP appelé `1_omp_parallel`

```
> cd exercises/openmp/1_omp_parallel
```

- Ouvrez les instructions contenues dans le fichier README.md avec votre éditeur de fichier favori (vim, emacs, atom, gedit...)

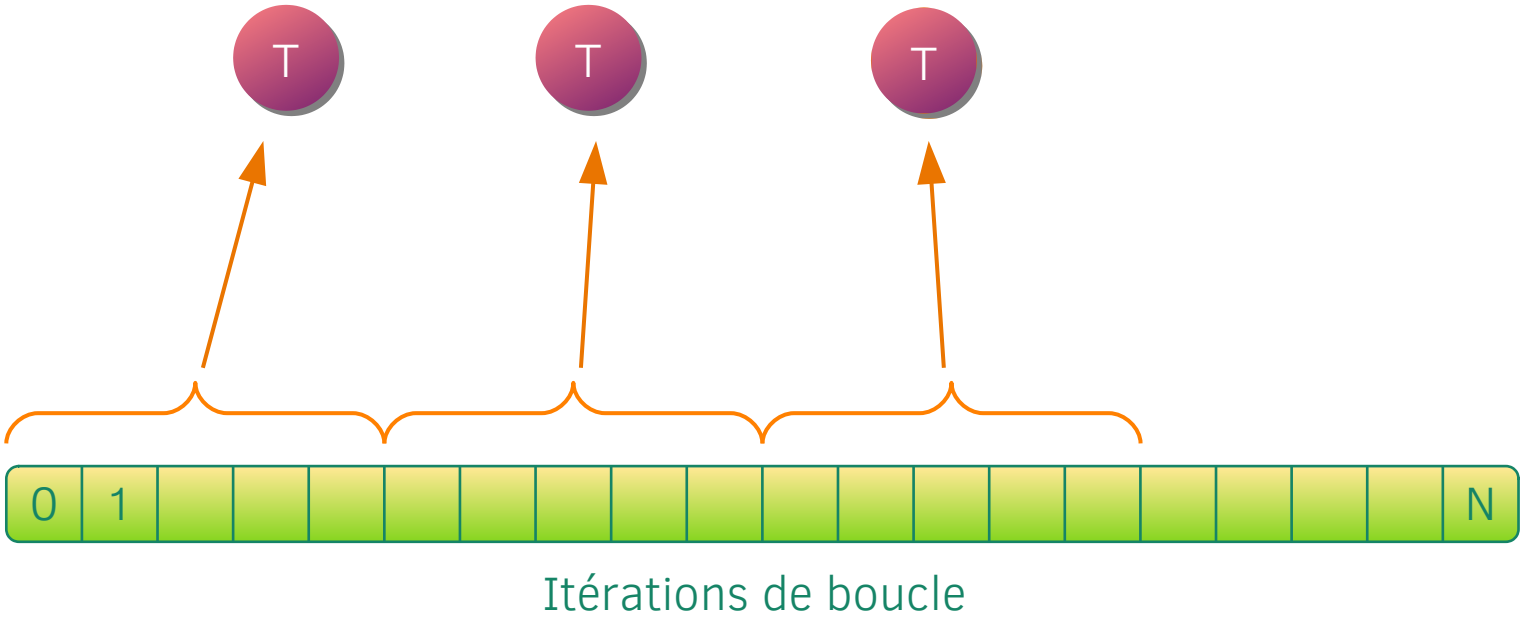
Vous pouvez lire le README directement depuis le Gitlab et c'est plus confortable comme ça.

Introduction au parallélisme multitâche par directives OpenMP

4) Partager le travail

Définition d'une concurrence mémoire
Memory race

La directive **!$OMP DO** permet de distribuer le travail d'une boucle dans une région parallèle entre tous les threads.



C'est une des sources principales d'erreurs en OpenMP

Parallélisation d'une boucle DO

La directive **!$OMP DO** permet de distribuer le travail d'une boucle dans une région parallèle entre tous les threads.

.f90

```
Program openmp

Implicit none

Real, dimension(N) :: A, B, C, D
Integer            :: i

!$omp parallel default(shared) private(i)

DO i=1,N
  A(i) = B(i) + C(i)*D(i)
ENDDO

!$omp end parallel

End program
```

La boucle est ici exécuté par tous les threads en même temps

Parallélisation d'une boucle DO

La directive **!$OMP DO** permet de distribuer le travail d'une boucle dans une région parallèle entre tous les threads.

.f90

```
Program openmp

Implicit none

Real, dimension(N) :: A, B, C, D
Integer            :: i

!$omp parallel default(shared) private(i)

!$omp do
DO i=1,N
  A(i) = B(i) + C(i)*D(i)
ENDDO
!$omp end do

!$omp end parallel

End program
```

La charge est maintenant partagée entre les threads disponibles

Parallélisation d'une boucle for en C et C++

La directive `#pragma omp for` permet, en C et C++, de distribuer le travail d'une boucle dans une région parallèle entre tous les threads.

.cpp

```
Int N ;
double A[N], B[B], C[N], D[N];

#pragma omp parallel default(shared)

for(int i= 0 ; i < N ; i++) {
  A[i] = B[i] + C[i]*D[i] ;
}
```

La boucle est ici exécuté par tous les threads en même temps

Parallélisation d'une boucle for en C et C++

La directive **#pragma omp for** permet, en C et C++, de distribuer le travail d'une boucle dans une région parallèle entre tous les threads.

.cpp

```
Int N ;
double A[N], B[B], C[N], D[N];

#pragma omp parallel default(shared)

#pragma omp for
for(int i= 0 ; i < N ; i++) {
  A[i] = B[i] + C[i]*D[i] ;
}
```

La charge est maintenant partagée entre les threads disponibles

Partage du travail avec la clause `SCHEDULE(type,chunk)`

La clause **`schedule`** permet d'expliciter la manière de partager le travail

.f90

```
Program openmp

Implicit none

Real, dimension(N) :: A, B, C, D
Integer            :: i

!$omp parallel default(shared) private(i)

!$omp do schedule(static, 10)
DO i=1,N
  A(i) = B(i) + C(i)*D(i)
ENDDO
!$omp end do

!$omp end parallel

End program
```

Partage du travail avec la clause `SCHEDULE(type,chunk)`

La clause **`schedule`** permet d'expliciter la manière de partager le travail :

- **`Static`** : partage des itérations en paquet de taille **`chunk.`** Les paquets sont attribués de façon cyclique.
- **`Dynamic`** : partage des itérations en paquets de taille **`chunk`**. Les paquets sont distribués dynamiquement dès qu'un thread est disponible.
- **`Guided`** : partage des itérations suivant une taille de paquet qui décroît. La taille est supérieure ou égale à **`chunk`**.
- **`Auto`** : le compilateur décide
- **`Runtime`** : la décision est prise en utilisant la variable d'environnement **`OMP_SCHEDULE`** avant l'exécution

```
> export OMP_SCHEDULE= « DYNAMIC, 100 »
```

Retour au séquentiel au sein des régions parallèles

Ouvrir et fermer régulièrement une région parallèle (`!$omp parallel`) possède un coût (overhead). Il est recommandé de le faire le moins possible au cours d'un programme, en particulier dans les régions intensives.

Il est pourtant parfois nécessaire d'effectuer des calculs sur un seul thread (affichage à l'écran par exemple, lecture ou sortie de fichier, appel de fonctions non multitâches…). Pour cela, on peut utiliser :

- `!$omp master` : seul le thread master exécute ce qui est dans la région qui suit. Les autres threads passent leur chemin et exécutent la suite.

- `!$omp single` : le premier thread qui arrive au niveau de cette directive se charge de la suite, les autres attendent qu'il finisse.

- `!$omp critical` : Tous les threads exécutent cette partie mais à tour de rôle dans un ordre non déterminé. Cette région est protégée par le thread en cours.

Retour au séquentiel au sein des régions parallèles : `!$omp master`

`!$omp master` : seul le thread master exécute ce qui est dans la région qui suit. Les autres threads passent leur chemin et exécutent la suite.

.f90

```
Program openmp

Implicit none

Real, dimension(N) :: A, B, C, D
Integer            :: i

!$omp parallel default(shared) private(i)

… partie parallèle

!$omp master

Print*, « je suis le master »

!$omp end master

!$omp end parallel

End program
```

Seul le thread master affichera l'information à l'écran

Retour au séquentiel au sein des régions parallèles : `!$omp single`

- `!$omp single` : le premier thread qui arrive au niveau de cette directive se charge de la suite, les autres attendent qu'il finisse.

.f90

```
Program openmp
Implicit none
Integer             :: id
!$omp parallel default(shared) private(id)
… partie parallèle
!$omp single
Id = omp_get_thread_num()
Print*, « je suis le thread », id
!$omp end single
!$omp end parallel
End program
```

Le premier thread qui arrive ici affichera son rang dans le terminal

Exercice n°2 : Parallélisation d'une boucle OpenMP

- Rendez vous sur le Gitlab des exercices : https://gitlab.maisondelasimulation.fr/mlobet/cours-hpc-m2-dfe
- Télécharger les exercices sur votre session de travail
- Décompressez l'archive en ligne de commande

```
> tar xvf archivedossier.tar
```

- Rendez vous dans le dossier de l'exercice n°2 OpenMP appelé `2_omp_do`

```
> cd exercises/openmp/2_omp_do
```

- Ouvrez les instructions contenues dans le fichier README.md avec votre éditeur de fichier favori (vim, emacs, atom, gedit...)

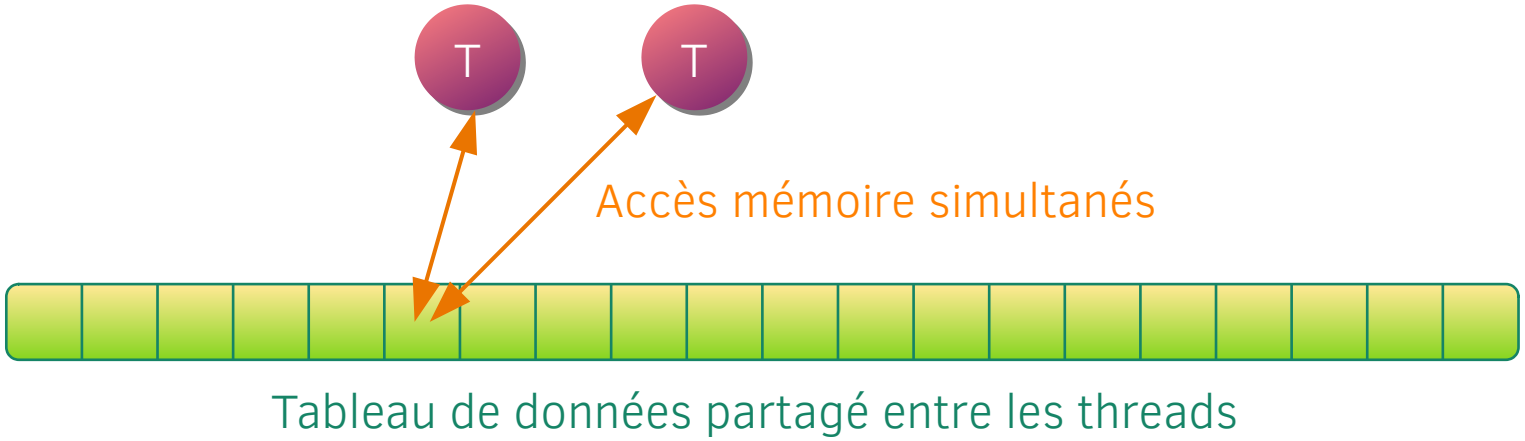
Vous pouvez lire le README directement depuis le Gitlab et c'est plus confortable comme ça.

Introduction au parallélisme multitâche par directives OpenMP

5) La concurrence mémoire

Définition d'une concurrence mémoire
Memory race

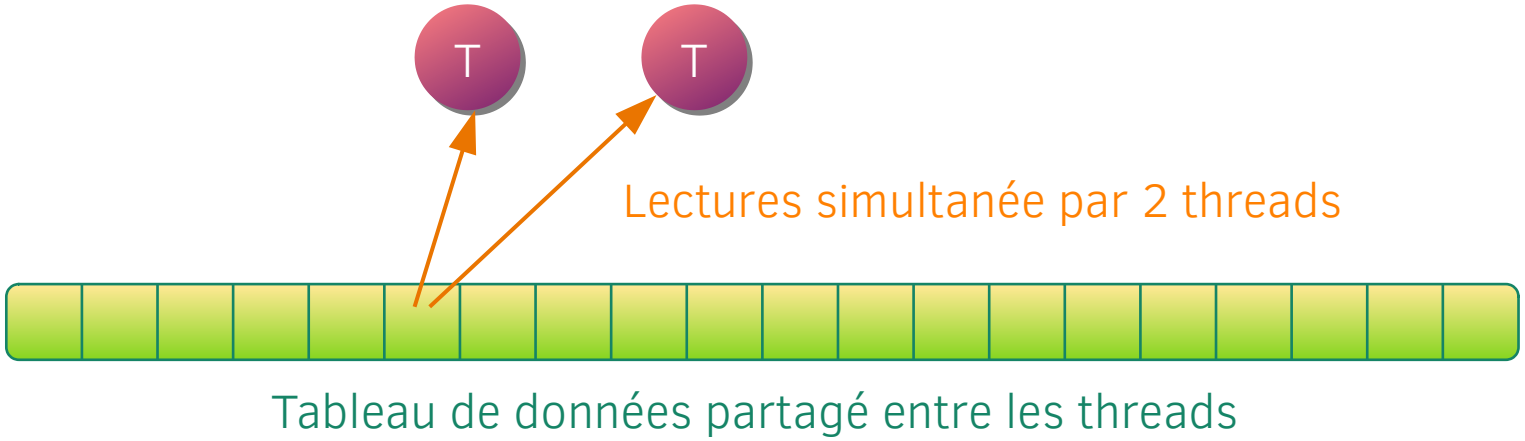
Il y a concurrence mémoire lorsque plusieurs threads accèdent au même espace mémoire (même variable) en même temps.



C'est une des sources principales d'erreurs en OpenMP

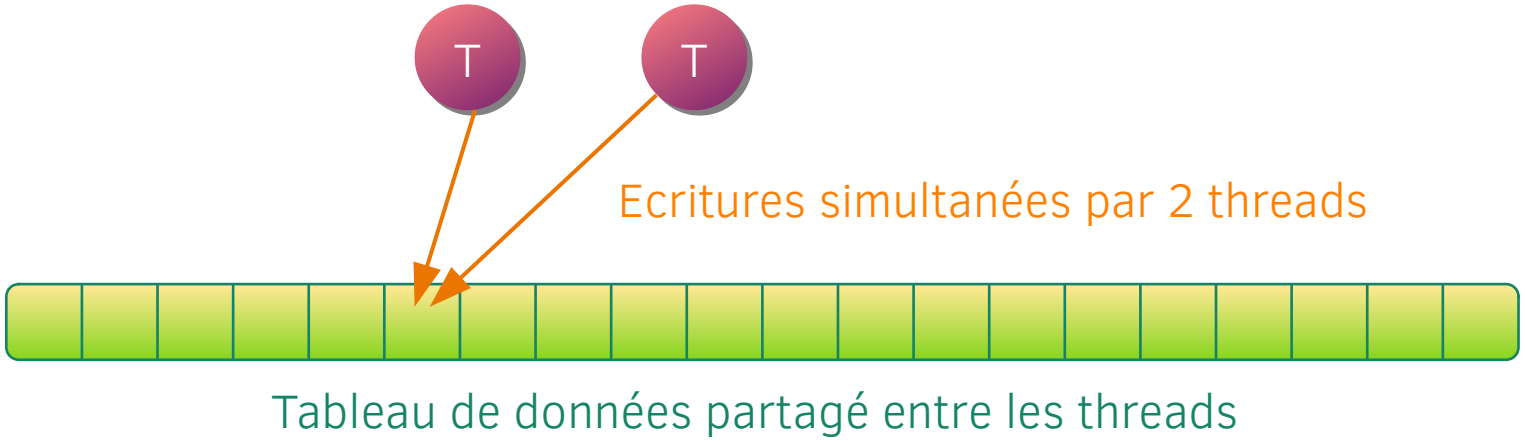
Accès simultanés en lecture

Tous les accès ne sont pas problématiques. Plusieurs threads peuvent accéder en lecture à la même donnée sans problème.



Accès simultanés en écriture

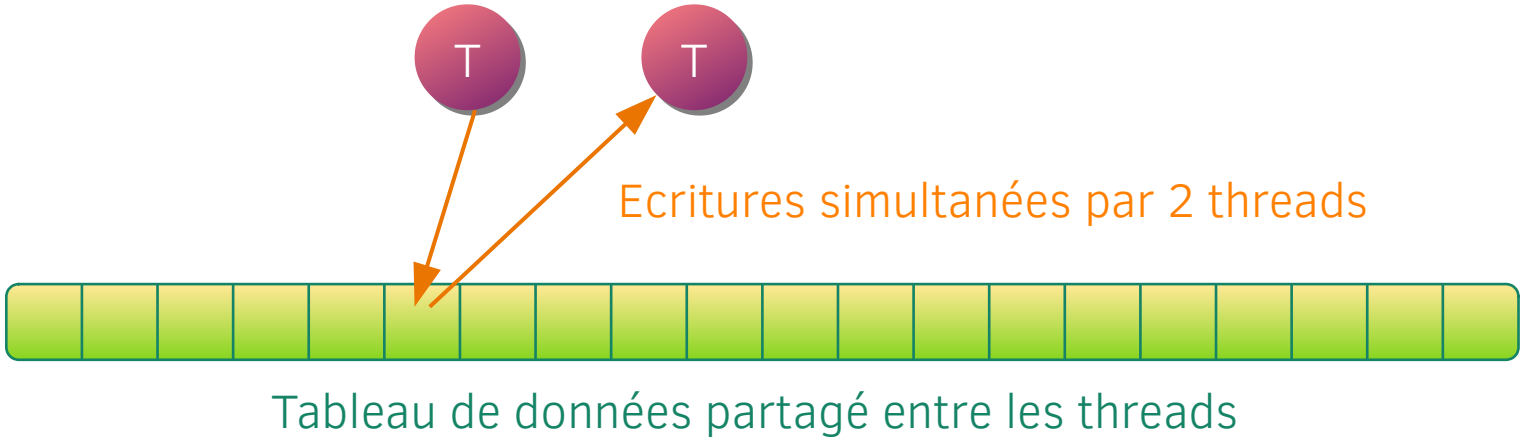
Si plusieurs threads écrivent au sein du même espace mémoire alors le contenu de la variable n'est pas garantie (pas *thread-safe*)



Problématique lors d'un accès « aléatoire » aux données

Accès simultanés en écriture et lecture

Si un thread écrit pendant qu'un autre lit le résultat alors le résultat de la lecture est aléatoire.



Problématique lors d'un accès « aléatoire » aux données

Gestion de la concurrence mémoire

Pour éviter ce genre de situation, il existe deux solutions :

- Réécrire l'algorithme problématique pour empêcher la concurrence mémoire (tableaux indépendants)
- Utiliser des directives OpenMP spécifiques permettant de protéger l'accès à la mémoire

Région critique (OpenMP critical)

Une région critique au sein d'une région parallèle permet de séquentialiser l'exécution de ce qui s'y trouve. Seul un thread à la fois peut entrer dans une telle région.

.f90

```
Real, dimension(N) :: A, B, C, D
Integer            :: i

!$omp parallel default(shared) private(i)

!$omp critical
DO i=1,N
  A(i) = B(i) + C(i)*D(i)
ENDDO
!$omp end critical

!$omp end parallel
```

Chaque thread effectuera cette opération à tour de rôle

L'ordre d'exécution dépend de l'ordre d'arrivée au niveau de la région critique.
L'ordre d'exécution n'est pas garanti.

Région critique (OpenMP critical)

Une région critique au sein d'une région parallèle permet de séquentialiser l'exécution de ce qui s'y trouve. Seul un thread à la fois peut entrer dans une telle région.

.cpp

```
Int N ;
double A[N], B[B], C[N], D[N];

#pragma omp parallel default(shared)

#pragma omp critical {
  for(int i= 0 ; i < N ; i++) {
    A[i] = B[i] + C[i]*D[i] ;
  }
}
```

Chaque thread effectuera cette opération à tour de rôle

L'ordre d'exécution dépend de l'ordre d'arrivée au niveau de la région critique. L'ordre d'exécution n'est pas garanti.

Région atomics
Atomics

Une région atomics permet de protéger l'accès mémoire spécifiquement. Les variables accédées dans une telle région sont protégées dès qu'un thread y effectue une action. Les autres threads doivent patienter pour accéder à la même donnée.

.f90

```
Real, dimension(N) :: A, B

!$omp parallel default(shared)

!$omp atomics
A(0) += 2*B(1)
!$omp end atomics

!$omp end parallel
```

Un thread à la fois peut modifier A(0)

Si une telle directive peut paraître la solution miracle, elle occasionne des synchronisations pouvant impacter la performance du code.

Région atomics
Atomics

Une région atomics permet de protéger l'accès mémoire spécifiquement. Les variables accédées dans une telle région sont protégées dès qu'un thread y effectue une action. Les autres threads doivent patienter pour accéder à la même donnée.

.cpp

```
double A[N], B[N] ;

#pragma omp parallel default(shared)

#pragma omp atomics {
A(0) += 2*B(1)
}

!$omp end parallel
```

Un thread à la fois peut modifier A(0)

Si une telle directive peut paraître la solution miracle, elle occasionne des synchronisations pouvant impacter la performance du code.

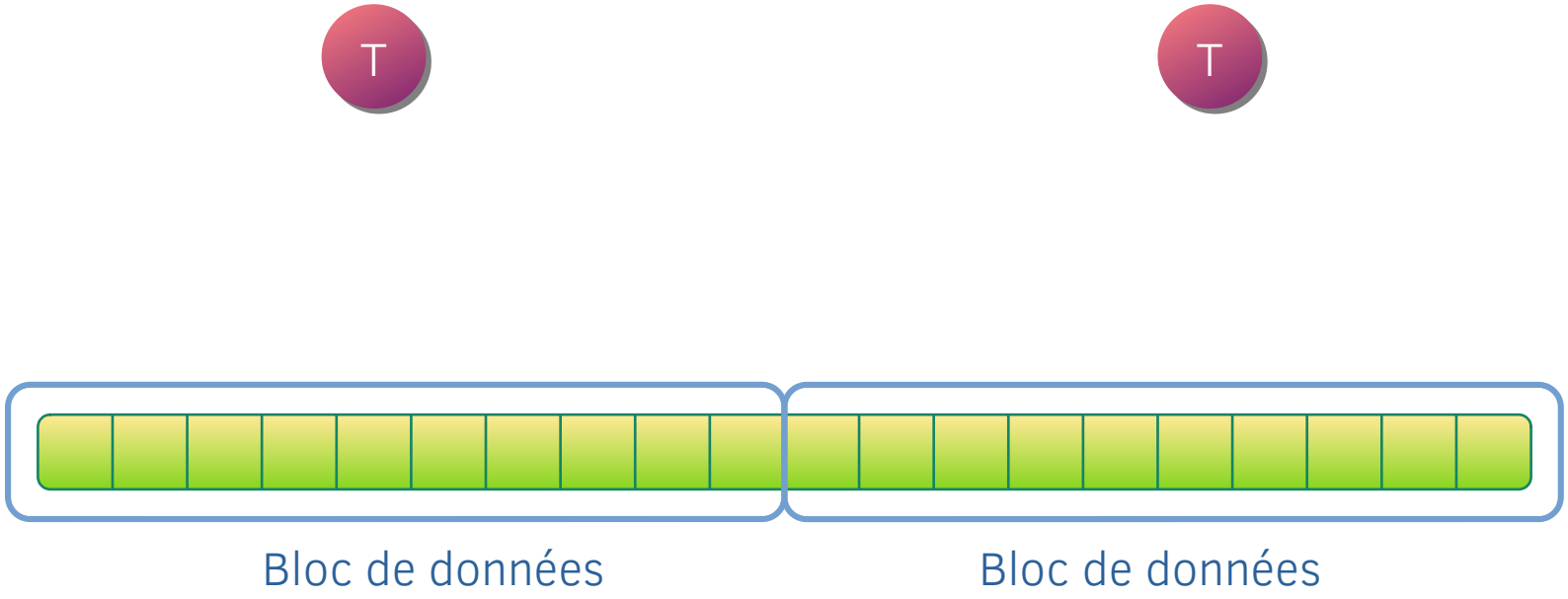
Les options pour les région atomics
Atomics

Il existe plusieurs clauses permettant d'affiner le comportement des régions atomics :

- Update : bloque la variable en lecture/écriture le temps de mettre à jour sa valeur – comportement par défaut.
- Read : protège la variable en lecture
- Write : protège la variable en écriture
- Capture : protège la variable tout en capturant la valeur originale ou finale (par exemple a = b ++)

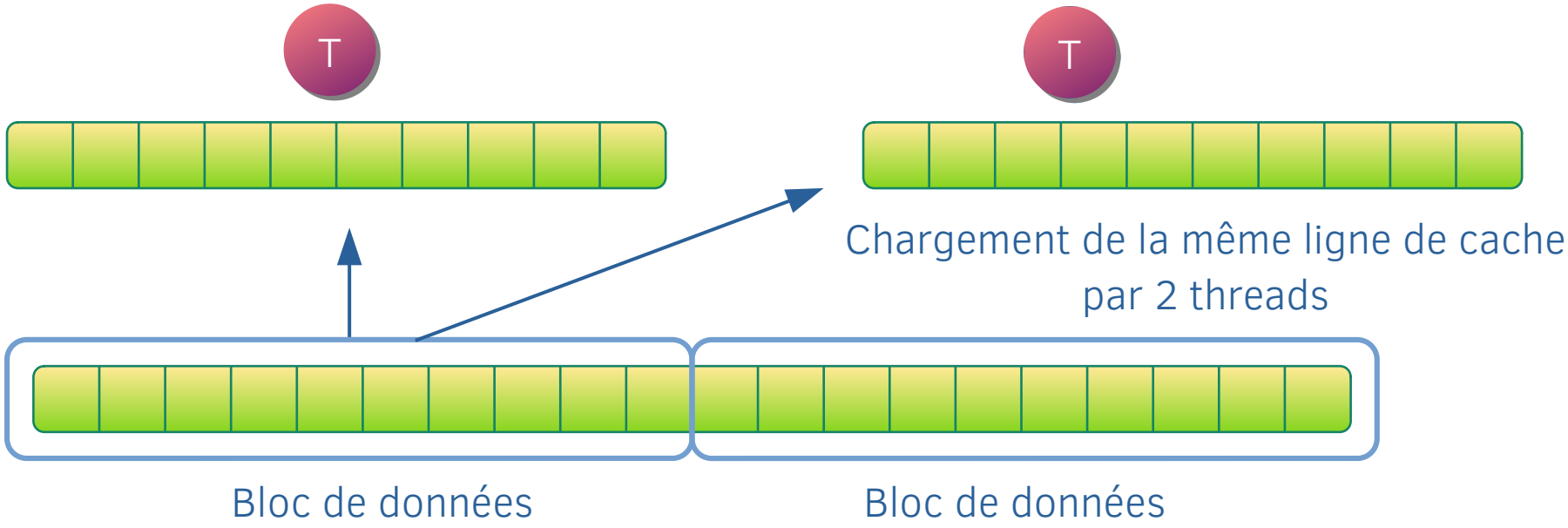
Se méfier du faux partage
False Sharing

Au sein des différents niveaux de cache, les données sont traitées par bloc dont la taille dépend du processeur (cache lines). Si deux threads accèdent au même bloc alors la cohérence du cache implique que chaque thread doit mettre à jour les données si ces dernières sont modifiées.



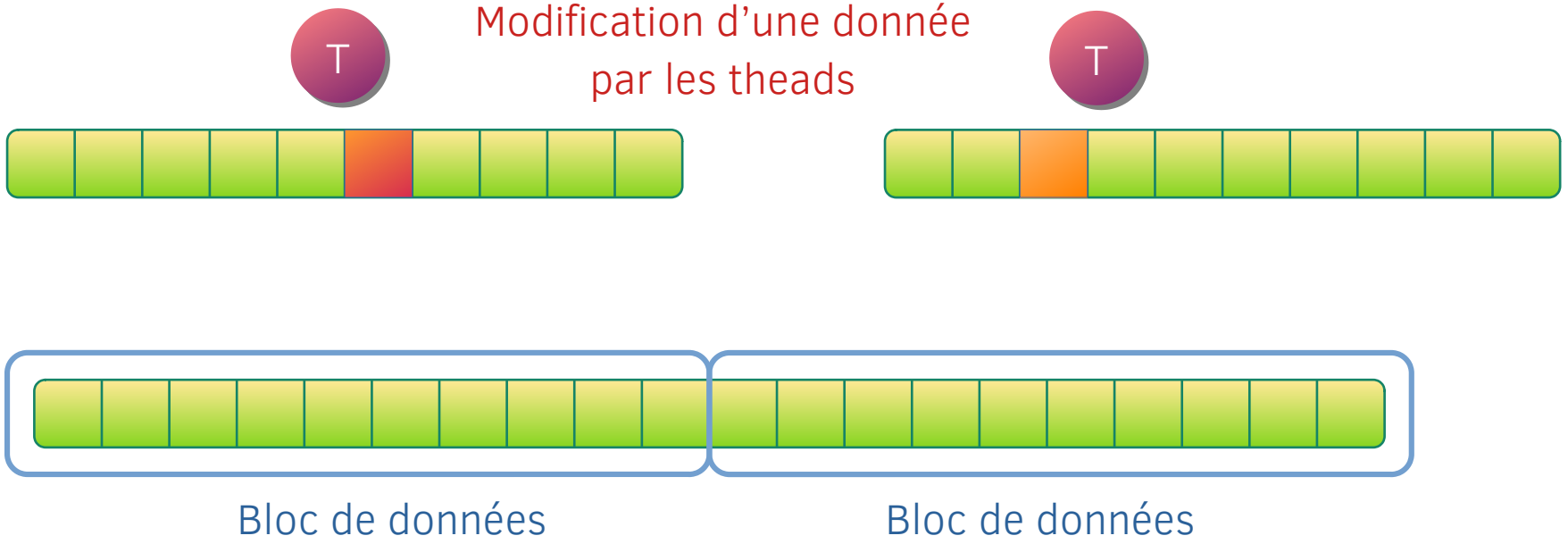
Se méfier du faux partage
False Sharing

Au sein des différents niveaux de cache, les données sont traitées par bloc dont la taille dépend du processeur (cache lines). Si deux threads accèdent au même bloc alors la cohérence du cache implique que chaque thread doit mettre à jour les données si ces dernières sont modifiées.



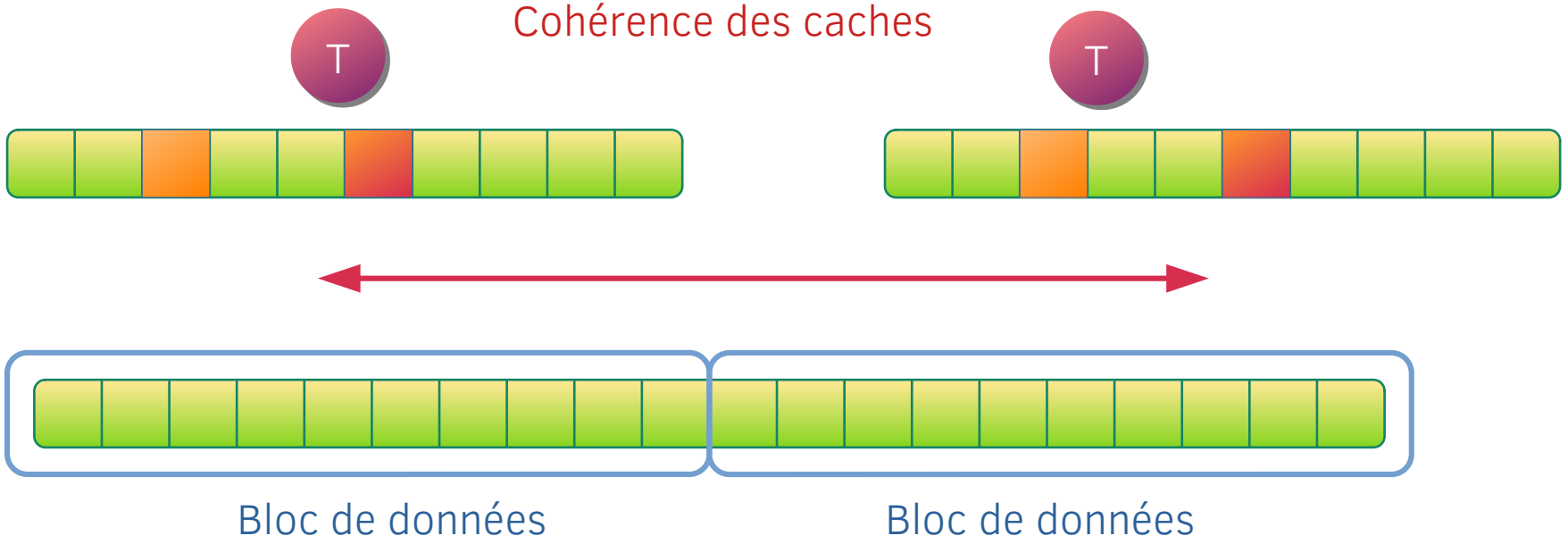
Se méfier du faux partage
False Sharing

Au sein des différents niveaux de cache, les données sont traitées par bloc dont la taille dépend du processeur (cache lines). Si deux threads accèdent au même bloc alors la cohérence du cache implique que chaque thread doit mettre à jour les données si ces dernières sont modifiées.



Se méfier du faux partage
False Sharing

Au sein des différents niveaux de cache, les données sont traitées par bloc dont la taille dépend du processeur (cache lines). Si deux threads accèdent au même bloc alors la cohérence du cache implique que chaque thread doit mettre à jour les données si ces dernières sont modifiées.



Exercice n°3 : concurrence mémoire

- Rendez vous sur le Gitlab des exercices : https://gitlab.maisondelasimulation.fr/mlobet/cours-hpc-m2-dfe
- Télécharger les exercices sur votre session de travail
- Décompressez l'archive en ligne de commande

```
> tar xvf archivedossier.tar
```

- Rendez vous dans le dossier de l'exercice n°2 OpenMP appelé `2_omp_do`

```
> cd exercises/openmp/3_omp_critical
```

- Ouvrez les instructions contenues dans le fichier README.md avec votre éditeur de fichier favori (vim, emacs, atom, gedit...)

Vous pouvez lire le README directement depuis le Gitlab et c'est plus confortable comme ça.

Introduction au parallélisme multitâche par directives OpenMP

6) Infos supplémentaires et conclusion

Les nouvelles versions d'OpenMP

A partir d'OpenMP 5, de nombreux changements apparaissent dans la nomenclature mais les compilateurs ne sont pas encore à jour.

Fin de la partie sur OpenMP

A ce stade du cours, vous savez maintenant :

- Utiliser les directives de base d'OpenMP et leurs clauses
- Paralléliser un programme composé de boucles simples en OpenMP
- Gérer la concurrence mémoire