**TP HPC/Big Data**

# OpenMP et MPI

Supaéro
Module FITR35

Rédigé par
Gaëlle Usseglio

2022/2023

## Préparation du TP

Le code source se situe sur ce dépôt.
Vous pouvez le rapatrier en local via la commande *git clone*.

Vous trouverez une partie dédiée à OpenMP et une autre à MPI. Par défaut, la branche *main* contient le code à compléter et la branche *solution*, les solutions des excercices.

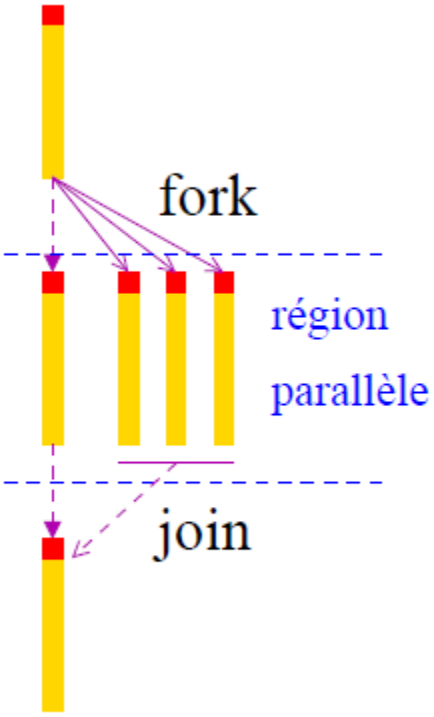
## OpenMP

## 1 Rappel des concepts

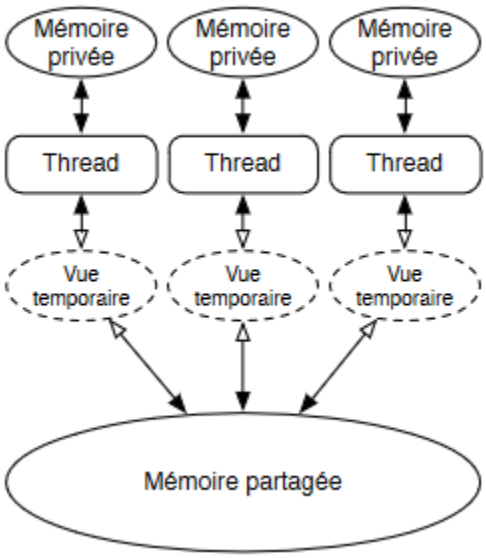
OpenMP manipule les concepts suivants :

- Thread : Entités d'exécution partageant un adressage (heap) et avec une mémoire locale (stack).
- Team : Un ensemble de un ou plusieurs threads qui participent à l'exécution d'une région parallèle.
- Task/Tâche : Une instance de code exécutable et ses données associées. Elles sont générées par les constructions PARALLEL ou TASK.
- Variable partagée : Une variable dont le nom permet d'accéder au même bloc de stockage au sein d'une région parallèle entre tâches.
- Variable privée : Une variable dont le nom permet d'accéder à différents blocs de stockage suivant les tâches, au sein d'une région parallèlle.

OpenMP est une alternance de régions séquentielles et parallèles : « fork and join » :




Une communication entre les threads implicite est faite grâce à la Mémoire partagée. :




## 2 Compilation - Run Time

Avant tout chose, il est important de vérifier son environnement et rendre la compilation "prête" pour OpenMP.

- Première condition, avoir un compilateur ! Regarder si le compulation GNU est présent.
- Seconde condition, activer OpenMP à la compilation. Le fichier *compil.sh* à la racine du répertoire openmp permet cette activation. Je vous laisse le regarder ....

Les principales APIs sont décrites dans le lien suivant **OpenMP cheat sheet**

## 3 Exemples

Deux exemples sont disponibles et permettent de mieux comprendre le fonctionnement des régions //, des scopes de variable et la notion de tâche :

- *just_parallel* :
  - La région parallèle permet-elle de différencier les actions réalisées dans les threads ?
  - Par défaut, quel est le scope des variables ? Qu'est ce que cela signifie ?
  - Quelles sont les différences entre les scopes ? Private vs shared ....
- *task* :
  - Quelle est la différence entre tâche et thread ?
  - Y-a-t-il besoin d'une synchronisation ?

## 4 Exercices

Maintenant, la question importante : Comment bénéficier de la parallélisation pour gagner en performance ? Essayer de paralléliser sur plusieurs threads les codes suivants :

- Estimation de **pi** via une intégrale :

$$pi = \int_0^1 f(x)\,\mathrm{d}x$$

avec

$$f(x) = \frac{4}{1+x^2}$$

- Exemple de reduction
- Exemple de boucle for avec protection en écriture de la variable d'accumulation

— Calcul **matriciel** : calculs arbitraires sur plusieurs tableaux avec une forte sollicitation mémoire :
- Manipulation du heap
- Utilisation de la stack

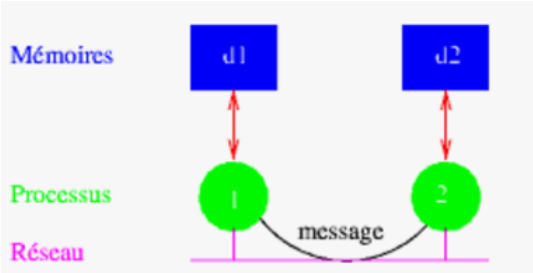
# MPI

## 5 Rappel des concepts

MPI manipule les concepts suivants :

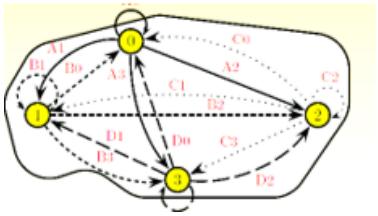
— Processus : Entité totalement indépendantes et isolées les unes des autres ne partageant pas d'adressage mémoire.

— Communication : Messages échangés entre processus via le réseau de manière explicite (visible dans le code).

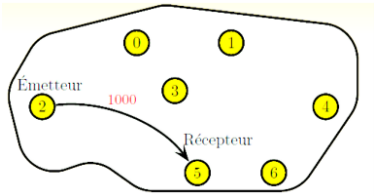



Deux types de communications sont à distinguer :

— Collectives :




— Point à Point




## 6 Compilation - Run Time

Pour vérifier son environnement et rendre la compilation/exécution "prête" pour MPI.

— Première condition, avoir un compilateur dédié ! Par exemple openMPI via les commandes *mpic++* et *mpirun*.

— Seconde condition, utiliser ce compilateur à la compilation et à l'exécution. Le fichier *compil.sh* à la racine du répertoire mpi/examples permet d'utiliser ce compilateur au moment de la compilation. Je vous laisse le regarder ....

Les principales APIs sont décrites dans le lien suivant **MPI cheat sheet**

# 7 Exemples

Les deux types de communications sont illustrés via des exemples :
- Collectives : APIs **Bcast** et **Scatter/Gather**
- Point à Point : APIs **Send/BSend** et **Recv**

# 8 Exercices

Maintenant à vous de jouer !

## 8.1 Exercice 1 : Les deadlocks

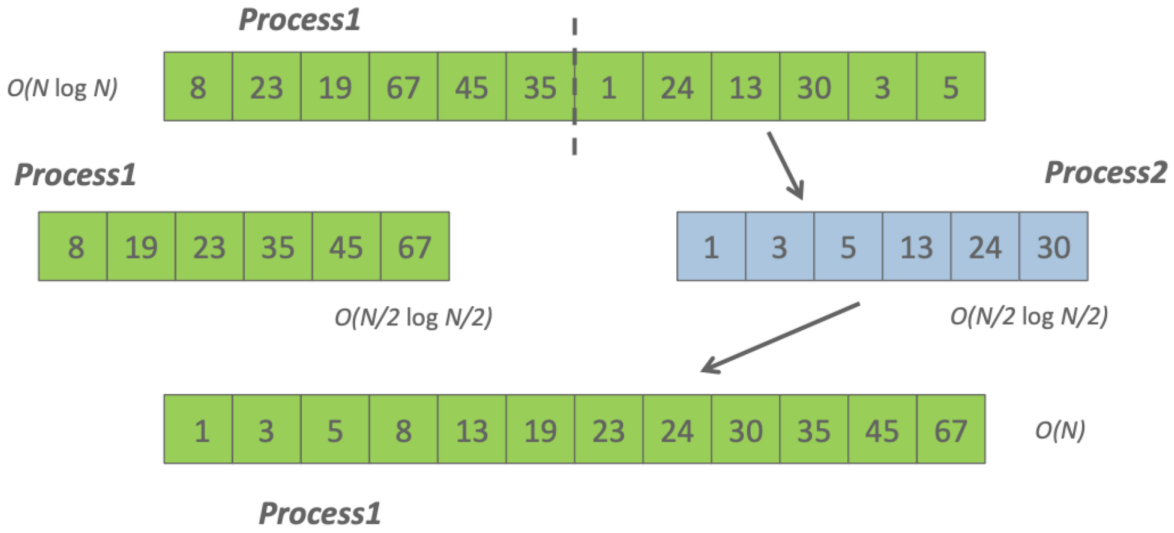
Il arrive que les communications MPI de par leurs natures bloquantes pour certains APIs, bloquent l'exécution d'un programme. Cela est par exemple le cas quand deux processus MPI veulent envoyer leur message respectif et ne sont pas prêtes à recevoir. Il est important d'identifier les cas ou cela se présente et trouver des alternatives.

Le programme *deadlock.cpp* illustre un cas de blocage.
Comment y remédier ? Il y a la possibilité d'utiliser d'autres APIs comme ISend et IRecv. Les commandes Wait et Test sont des compléments pour l'envoi de messages non bloquants.

## 8.2 Exercice 2 : Manipulation de tableaux

Tout comme OpenMP, un des enjeux de MPI est de découper le travail entre les processus. Ce découpage est entièrement à la charge du développeur avec la mise en place de conditions et communications.

Imaginons avoir comme traitement à trier un "gros" tableau d'entrée. Il est posible d'utiliser MPI afin de découper le tri à travers plusieurs processus.




Le répertoire *mpi/exercises/arrays* contient un programme séquentiel réalisant ce tri. Ce programme fait appel aux fonctions utilitaires de de la librairie *array_lib*, permettant de générer un tableau, de faire le tri et de rassembler un ensemble de tableau préalablement triés.

Aller plus loin : Imaginons que désormais le "gros" tableau d'entrée est d'ores et déjà découpé en un nombre égale au nombre de processus (ex : 4). Il serait alors possible d'effectuer le tri sur les 4 processus

avec un tableau par processus et un rassemblement à la fin. Les groupes/communicateurs pourraient rentrer en jeu pour découper le problème en deux avec un rassemblement intermédiaire avant un rassamblement global des 4 tableaux.