

# Introduction à OpenMP

## 1 Intro

### TP avec le cluster Muse

Pour vous connecter à muse :

Votre compte sur le cluster Meso@LR vous est donné par l'enseignant. Vous devez signer la charte et la lui remettre.

- Cluster : muse-login.meso.umontpellier.fr
- Login :
- Mot de passe :
- Account : lamps
- Partition(s) : upvd

Des informations supplémentaires, documentations relative au cluster MUSE sont accessible à : <https://meso-lr.umontpellier.fr/documentation-utilisateurs/>.

### TP sur votre machine perso

Configuration de OpenMP sur Ubuntu / Linux

1. Exécutez `>sudo apt-get install libomp-dev` dans votre terminal.
2. Sous code faites un click droit dans votre projet, sélectionné « propriété », « C/C++ », « Langage », et changez « Support d'OpenMP » pour « OUI ».

Ce document vous guide à travers les exercices. Veuillez suivre les instructions données pendant la conférence / séance d'exercices sur la façon de se connecter au cluster.

Lien vers l'archive .tar contenant les exercices C++ : <https://tinyurl.com/UPVD-OpenMP>

**Linux:** Veuillez télécharger l'archive correspondante et l'extraire dans votre répertoire \$HOME, via les commandes suivantes

```
cd $HOME
wget https://tinyurl.com/UPVD-OpenMP
unzip openmp_exercices.zip
```

**Linux :** Les makefiles fournissent plusieurs cibles pour compiler et exécuter le code :

- debug : le code est compilé avec OpenMP activé, toujours avec une prise en charge complète du débogage.
- release: Le code est compilé avec OpenMP et plusieurs optimisations de compilateur activées, ne pas utiliser en mode débogage.
- run : exécute le code compilé. La variable d'environnement OMP\_NUM\_THREADS doit être définie dans l'interpréteur de commandes appelant.
- clean : nettoyez tous les fichiers de build existants.

## 2 Hello World

Accédez au répertoire `hello`. Compilez le code `hello` via `'make [debug|release]'` et exécutez l'exécutable résultant via `'OMP_NUM_THREADS=procs make run'`, où `procs` indique le nombre de threads à utiliser.

**Exercice 1 :** Modifiez le code de tel sorte que (a) le numéro de thread (*ID de thread*) et (b) le nombre total de threads de la *team* sont affichés. Recompilez et exécutez le code afin de vérifier vos modifications.

**Exercice 2 :** Dans quel ordre attendiez-vous l'affichage des messages Hello World ? Etes-vous satisfait, sinon qu'est ce qui ne va pas ?

## 3 Parallélisation de Pi par intégration numérique

Accédez au répertoire `pi`. Ce code calcule Pi via une intégration numérique. Compilez le code `pi` via `'make [debug|release]'` et exécutez l'exécutable résultant via `'OMP_NUM_THREADS=procs make run'`, où `procs` indique le nombre de threads à utiliser.

**Exercice 1 :** Paralléliser le code Pi avec OpenMP. La partie à forte intensité de calcul réside dans une seule boucle dans la fonction `CalcPi()`, donc la *région parallèle* doit également y être placée. Recompilez et exécutez le code afin de vérifier vos modifications.

**Remarque :** Assurez-vous que votre code ne contient aucune *data race* (c'est-à-dire deux threads accédant à la même variable partagée sans synchronisation).

**Exercice 2 :** Si vous travaillez sur un système multicœur (par exemple, le cluster MUSE), mesurez l'accélération et l'efficacité du programme Pi parallèle.

# Threads	Runtime [sec]	Speedup	Efficiency
1			
2			
3			
4			
6			
8			
12			

## 4 Parallélisation d'un solveur Jacobi itératif

Allez dans le répertoire `jacobi`. Compilez le code `jacobi.c` via `'make [debug|release]'` et exécutez l'exécutable résultant via `'OMP_NUM_THREADS=procs make run'`, où `procs` indique le nombre de threads à utiliser.

**Exercice 1** : Utilisez Intel VTune Profiler XE (OneAPI) pour trouver les parties du programme à forte intensité de calcul du solveur Jacobi. Il doit y avoir trois hotspot dans le programme (en fonction du jeu de données d'entrée) :

Number	Line Number	Function Name	Runtime Percentage
1			
2			
3			

**Exercice 2** : Paralléliser les parties du programme à forte intensité de calcul avec OpenMP. Pour commencer simplement, créez une *région parallèle* pour chaque point d'accès de performances.

**Exercice 3** : Essayez de combiner des *régions parallèles* qui sont dans la même routine en une seule *région parallèle*.

**Exercice 4** : Si vous travaillez sur une machine NUMA, pensez à la distribution des données du code `jacobi`. Modifiez l'initialisation des données pour une meilleure distribution des données si nécessaire. Si vous le souhaitez, vous pouvez également paralléliser la vérification des erreurs.

## 5 Concept de tâches : Fibonacci

Au cours de cet exercice, vous examinerez la fonctionnalité de tâche introduite dans OpenMP 3.0.

**Exercice 1** : Allez dans le répertoire de Fibonacci. Ce code calcule le nombre de Fibonacci à l'aide d'une approche récursive – ce qui n'est pas optimal du point de vue des performances, mais bien adapté à cet exercice.

Examinez le code de Fibonacci. Parallélisez le code en utilisant le concept de tâche d'OpenMP 3.0. N'oubliez pas : la *région parallèle* doit résider dans `main()` et la fonction `fib()` doit être entrée la première fois avec un seul thread. Vous pouvez compiler le code via `'make [debug|release]'`.

**Exercice 2** : Dans le cours, vous avez entendu dire que la création de tâches peut se révéler inefficace (trop de petites tâches lancées). Implémentez cette idée dans votre code et arrêtez de créer de nouvelles tâches lorsque `n` est inférieur à 30. Pour les dernières valeurs de Fibonacci, écrivez une fonction distincte qui continue à calculer mais en série.

## 6 Work-Distribution

Accédez au répertoire `for`. Compilez le code `for` via `'make [debug|release]'` et exécutez l'exécutable résultant via `'OMP_NUM_THREADS=procs make run'`, où *procs* indique le nombre de threads à utiliser.

**Exercice 1** : Examiner le code et réfléchir à l'endroit où placer la ou les directives de parallélisation.

**Exercice 2** : Mesurer l'accélération et l'efficacité du code parallélisé. Quelle est la qualité de *scaling* du code et à quel *scaling* vous attendiez-vous ?

# Threads	Runtime [sec]	Speedup	Efficiency
1			

## 7 Min/Max-Reduction en C/C++

Accédez au répertoire `minmaxreduction`. Compilez le code `MinMaxReduction` via `'make [debug|release]'` et exécutez l'exécutable résultant via `'OMP_NUM_THREADS=procs make run'`, où *procs* indique le nombre de threads à utiliser.

**Exercice 1** : Depuis OpenMP 3.1, une opération de réduction pour min/max est prise en charge. Ajoutez le code nécessaire pour calculer `dMin` et `dMax` (comme indiqué aux lignes 29 et 30) en parallèle.

## 8 Quicksort

Quicksort est un algorithme récursif qui, dans ce cas, est utilisé pour trier un tableau de nombres entiers aléatoires. Son fonctionnement est décrit dans les étapes suivantes.

Un élément pivot est choisi. La valeur de cet élément est le point où le tableau est divisé dans ce niveau de récursivité.

5	8	1	7	4	9	2	1	0	3	4	6
---	---	---	---	---	---	---	---	---	---	---	---

Toutes les valeurs inférieures à l'élément pivot sont déplacées vers l'avant du tableau et tous les éléments plus grands que l'élément pivot vers la fin du tableau. L'élément pivot se trouve entre les deux parties. Notez que, selon l'élément pivot, les partitions peuvent différer en taille.

4	1	3	4	0	2	1	5	9	7	8	6
---	---	---	---	---	---	---	---	---	---	---	---

Les deux partitions sont triées séparément par appels récursifs à quicksort.

							5				
4	1	3	4	0	2	1		9	7	8	6

La récursivité se termine lorsque le tableau atteint une taille de 1, car un élément est toujours trié.

Accédez au répertoire quicksort. Compilez le code Quicksort via 'make [debug|release]' et exécutez l'exécutable résultant via 'OMP\_NUM\_THREADS=procs make run', où procs indique le nombre de threads à utiliser.

**Exercice 1 :** Les partitions créées à l'étape 3 peuvent être triées indépendamment les unes des autres, ce qui peut se faire en parallèle. Utilisez les tâches OpenMP pour paralléliser le programme quicksort.

**Exercice 2 :** La création de tâches pour de très petites partitions est inefficace. Implémentez une limite pour créer des tâches uniquement s'il reste suffisamment de travail. Par exemple, lorsque plus de 10 000 nombres doivent être triés, une tâche peut être créée, pour les tableaux plus petits, aucune tâche n'est créée.

**Astuce :** Vous pouvez ajouter des clauses `if` aux pragmas de tâche.

**Exercice 3 :** La clause `if` doit être évaluée chaque fois que la fonction est appelée, bien que la taille du tableau ne dépasse pas 10 000 éléments à un niveau inférieur. Implémentez une fonction `serial_quicksort` et appelez cette fonction lorsque le tableau devient trop petit. Cela peut aider à éviter les frais généraux de la clause `if`.



## 8.1 Finding Data Races: Primes

Accédez au répertoire des nombres premiers. Compilez le code PrimeOpenMP via 'make [debug|release]' et exécutez l'exécutable résultant via 'OMP\_NUM\_THREADS=procs make run', où procs indique le nombre de threads à utiliser.

**Exercice 1 :** Exécutez le programme deux fois, avec un nombre donné de threads (au moins deux). Vous constaterez que le nombre de nombres premiers trouvés dans l'intervalle spécifié changera - ce qui n'est bien sûr pas le résultat correct. Essayez de trouver la *Data Race* en regardant le code source ...

**Exercice 2:** Utilisez l'Intel Inspector XE (OneAPI) pour trouver la *Data Race*. Afin de ne pas attendre trop longtemps le résultat de l'analyse, raccourcissez l'intervalle de recherche. L'intervalle est fourni en tant qu'arguments au programme. Les arguments d'entrée doivent être spécifiés dans Inspector. *Remarque:* Pour utiliser Inspector, vous devez charger le module approprié (« module load intelixe »), mais vous n'avez pas besoin de passer à une autre machine. Définissez OMP\_NUM\_THREADS sur au moins 2 ('export OMP\_NUM\_THREADS=2') et démarrez l'interface graphique avec 'inspxe-gui'.

Linux : les étapes suivantes sont nécessaires pour rechercher des *Data Race*.

1. Cliquez sur « Fichier -> Nouveau projet -> »
2. Entrez le nom de votre choix et choisissez un emplacement pour stocker les données de résultat.
3. Choisissez « PrimeOpenMP.exe » comme application de votre répertoire d'exemple en utilisant le « parcourir » bouton.
4. Comme paramètres d'application spécifient un petit intervalle de recherche (par exemple « 0 1000 ») et quittez la fenêtre de dialogue en appuyant sur « ok ».
5. Cliquez sur le bouton « nouvelle analyse » .
6. Choisissez « Threading Error Analysis > Locate Deadlocks and Data Races » comme type d'analyse et appuyez sur le  bouton.

**Exercice 3 :** Corrigez le code PrimeOpenMP à l'aide des constructions de synchronisation OpenMP appropriées. Utilisez l'inspecteur XE pour vérifier que vous avez éliminé toutes les *Data Race*.

**Exercice 4 :** Quelles sont les limites des outils de détection Data Race tels que l'Intel Inspector XE ?

**Exercice 5 :** Pouvez-vous expliquer pourquoi la vérification du programme au moment de la compilation ne peut être que très limitée et pourquoi elle ne peut pas détecter les problèmes que les outils de vérification des threads sont en mesure de signaler ?

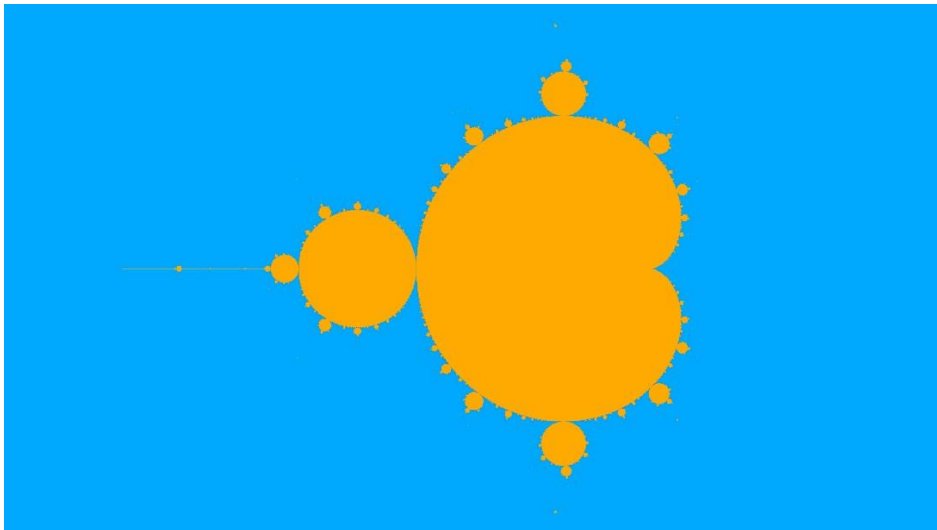
## 8.2 Finding Data Races: Pi

Accédez au répertoire datarace\_detection.

**Exercice 1 :** Suivez le manuel sur la façon d'exécuter une analyse de la course des données avec Intel Inspector et corrigez les erreurs dans le code source.

## 9 Mandelbrot

L'ensemble de Mandelbrot est un ensemble de nombres complexes qui a une limite fractale très « alambiquée » lorsqu'ils sont tracés. Le code donné calcule et trace l'ensemble de Mandelbrot. Le tracé généré ressemble à ceci :



Accédez au répertoire `mandelbrot`. Compilez le code `mandelbrot` via `'make [debug|release]'` et exécutez l'exécutable résultant via `'OMP_NUM_THREADS=procs make run'`, où `procs` indique le nombre de threads à utiliser.

**Exercice 1 :** Exécutez le code avec un thread et plusieurs threads et comparez les images résultantes. ressemblent-ils à l'image ci-dessus?

**Exercice 2 :** L'une des images est incorrecte. Avez-vous une idée de ce qui ne va pas? Connaissez-vous un outil qui peut vous aider à trouver l'erreur? Essayez de détecter et de corriger l'erreur dans le code.