

Université des Sciences et de la Technologie Houari Boumediene
Faculté d'Informatique



ACP

TP2 : Calcul parallèle

Modèle de programmation pour machines à mémoire partagées

Fait par :

Nom et prénom : ABDELMALEK BENMEZIANE

Matricule : 171731046778

Spécialité : M2 BIOINFO

Section : A

Contents

1	Définition	1
1.1	Présentation d'OpenMP	1
1.2	Le modèle de programmation de openMP	1
2	Premier programme	3
2.1	Le code source	3
2.2	Compilation et exécution	3
3	Les directives de openMP	4
3.1	Les types des directives	4
3.1.1	Directives de compilation	4
3.1.2	Partage de taches indépendantes	5
3.1.3	Partage de taches spécial boucles	7
3.1.4	Directive single	7
4	Les clauses de openMP	9
4.1	Les types des clauses	9
4.1.1	Réduction	9
4.1.2	If	10
4.1.3	Schedule	10
4.1.4	Liées aux status des variables	11
4.1.5	Principales clauses de statuts	11
5	Les fonctions de openMP	12

5.1	Les fonctions de la bibliothèque openMP	12
5.2	Les fonctions de mesure du temps	12
6	Exercices	13
6.1	Exercice 1	13
6.1.1	Compilation et exécution	13
6.1.2	L'ajout des sections	13
6.2	Exercice 2 (la clause Schedule)	15
6.3	Exercice 3	16
6.3.1	Static schedule	16
6.3.2	Dynamic schedule	16
6.4	Exercice 4 (calcul de pi)	16

List of Figures

1.1	Output	2
2.1	Output	3
3.1	Syntax	6
3.2	Syntax	7
3.3	Syntax	8
4.1	Syntax	10
6.1	Output	13
6.2	Output	14
6.3	Output	14
6.4	Output	15
6.5	Output	16
6.6	Output	16
6.7	Output	17

Chapter 1

Définition

1.1 Présentation d'OpenMP

API de programmation parallèle pour machines à mémoire partagée (supporte C, C++ et Fortran sur la majorité des systèmes d'exploitation Linux, macOS et Windows. Elle est basée sur 3 composants essentiels:

- Des directives de compilation : spécifier ce qu'on parallélise, les synchronisations et le **statut** des données (privé, partagé).
- Des routines (ou fonctions) de librairie : pour obtenir des informations générales sur le programme.
- Des variables d'environnement permettant de préciser des valeurs pour certains paramètres sans passer par le programme.

OpenMP permet une programmation multithread de plus haut niveau comparativement aux bibliothèques threads comme **pthread**s par exemple.

1.2 Le modèle de programmation de openMP

Le modèle est basé sur le principe fork join. Le processus principal (thread principale , ou thread master) crée un groupe de threads (étape fork) au début d'une « région parallèle » délimitée par le programmeur à l'aide de directives de compilation. Les threads s'exécutent en parallèle avec une

synchronisation implicite à la fin de la région parallèle (étape join) et les threads sont aussi détruits. A la fin d'une région parallèle, le programme redevient séquentiel (exécuté par le processus principal).

Le nombre de threads à utiliser peut être déterminé de plusieurs façons. Une région parallèle peut être une boucle à exécuter par plusieurs threads, ou simplement plusieurs sections (tâches) indépendantes qui seront distribuées sur les threads. Pour la synchronisation entre threads il existe aussi des directives permettant de protéger l'accès à des données partagées, ou délimiter une région de code pour une section critique, etc.

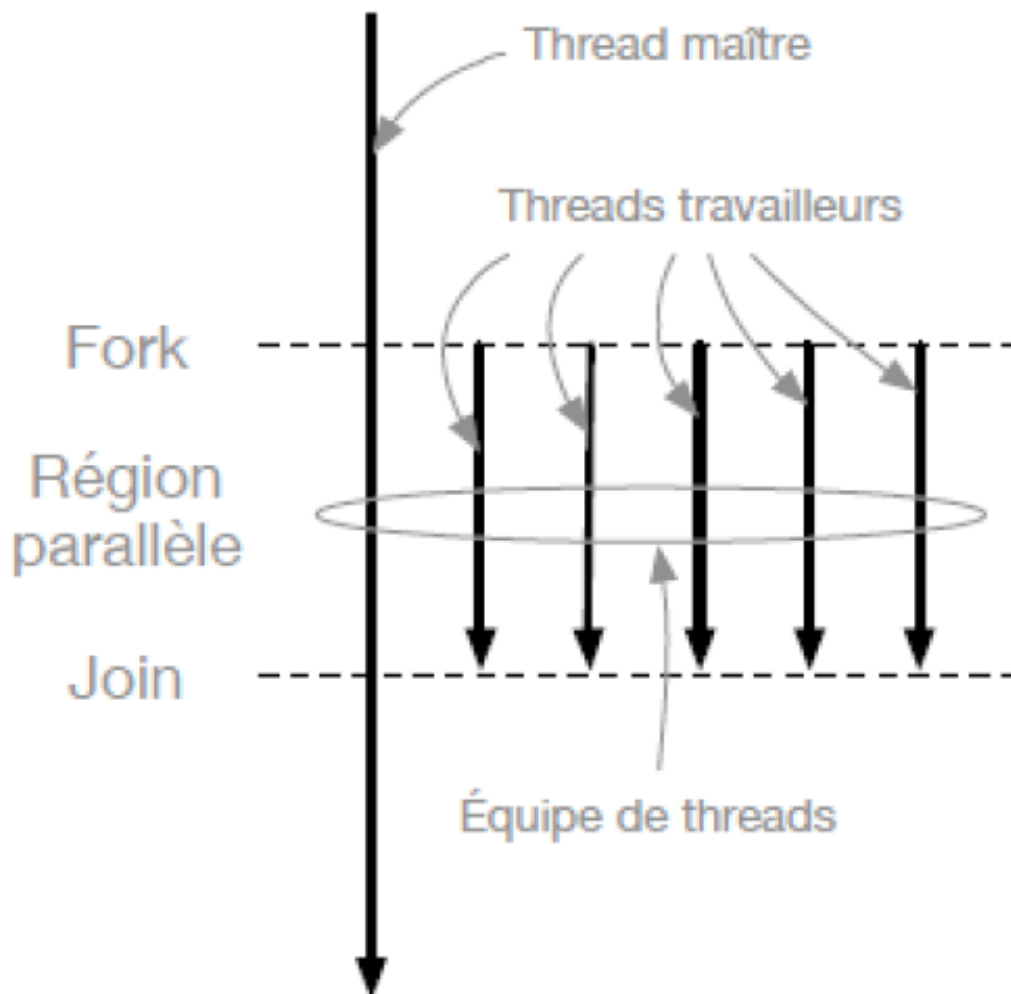


Figure 1.1: Output

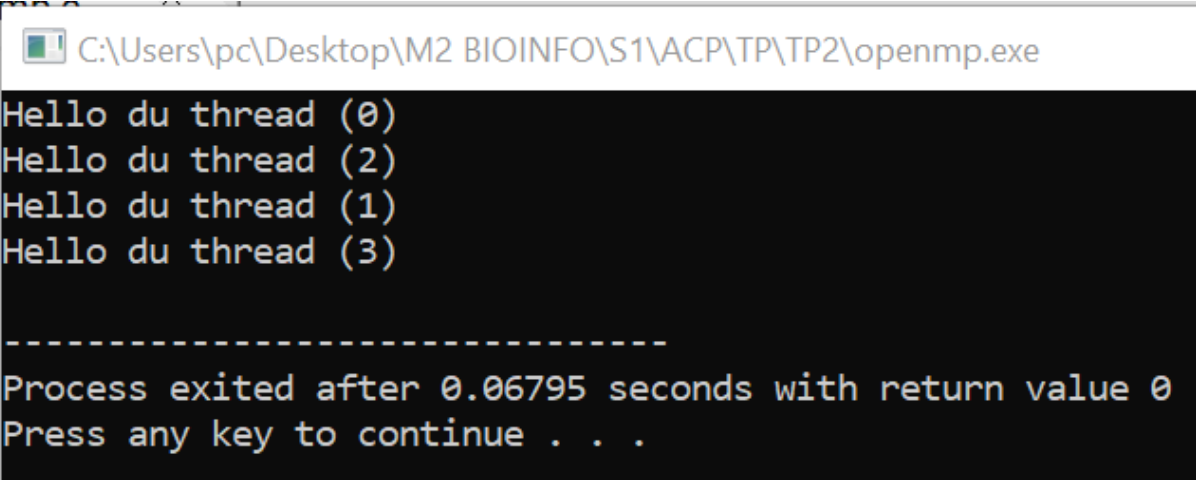
Chapter 2

Premier programme

2.1 Le code source

Le programme est dans le fichier source "openmp.c".

2.2 Compilation et exécution



```
C:\Users\pc\Desktop\M2 BIOINFO\S1\ACP\TP\TP2\openmp.exe
Hello du thread (0)
Hello du thread (2)
Hello du thread (1)
Hello du thread (3)
-----
Process exited after 0.06795 seconds with return value 0
Press any key to continue . . .
```

Figure 2.1: Output

Chapter 3

Les directives de openMP

3.1 Les types des directives

Le langage openMP contient plusieurs directives sont:

- directives de compilation
- partage de tâches indépendantes (sections)
- partage de tâches spécial boucles (for)
- directive (single)

3.1.1 Directives de compilation

La directive `#pragma omp parallel` permet de spécifier le début et la fin d'une région parallèle. Un groupe de threads est créé au début de la région et tous vont exécuter le même code.

Pour la clause, cela peut être une des commandes suivantes :

- Schedule (permet de spécifier le mode d'ordonnancement pour le groupe de threads).
- nowait (permet d'annuler la barrière de synchronisation qui existe par défaut à la fin de chaque directive de compilation).

- `num_threads(expression)` : permet de spécifier le nombre de threads à créer à l'entrée de la région parallèle.
- `shared (listeVariables)` (permettent de spécifier la visibilité des variables spécifiées dans la liste).
- `private (listeVariables)`.
- `firstprivate (listeVariables)`.

Exemples

1. `#pragma omp parallel nowait`
2. `#pragma omp parallel schedule(static/dynamic, [chunksize])`
3. `#pragma omp parallel shared(A, B, C) private (tid)`

3.1.2 Partage de tâches indépendantes

C'est sous formes de sections de code indépendantes à exécuter en parallèle par plusieurs threads. On peut alors créer une région avec la directive `sections` et à l'intérieur on délimite plusieurs régions parallèles avec la directive `section`. Par défaut il y a une barrière de synchronisation à la fin de la directive `omp sections` sauf si on précise la clause `nowait` spécifiée dans `omp sections`.

```
#pragma omp parallel
{
#pragma omp sections
{
#pragma omp section
X  calculation();
#pragma omp section
y  calculation();
#pragma omp section
z_calculation();
}
}
```

Figure 3.1: Syntax

3.1.3 Partage de tâches spécial boucles

partager les itérations d'une boucle for entre plusieurs threads qui vont exécuter le même code sur soit des données différentes. Le compteur de la boucle est par défaut privé à chaque thread. Ce qui permet aussi à chaque thread d'accéder aux bonnes données (quand il s'agit de boucle for itérant une structure de données de type tableau, matrice).

```
#pragma omp parallel for  
for(i=0;I<N;i++) { a[i] = a[i] + b[i];}  
}
```

Figure 3.2: Syntax

3.1.4 Directive single

Elle permet de préciser qu'un bloc de code à l'intérieur d'une région parallèle est à exécuter par un seul thread. Peut être utile dans le cas de partie de code qui ne peuvent pas être exécutées par plusieurs threads (par exemple les entrées/sorties).

```
#pragma omp single  
{  
// Bloc  
}
```

Figure 3.3: Syntax

Chapter 4

Les clauses de openMP

4.1 Les types des clauses

On peut associer une ou plusieurs clauses aux directives de compilation openmp.

- Réduction
- If
- Schedule
- Liées aux statuts des variables

4.1.1 Réduction

Le code suivant permet de calculer la somme puis la moyenne des éléments du tableau A. Un moyen de paralléliser ce code est d'utiliser la directive parallèle `for` avec la clause réduction (`+:moy`) la variable `moy` doit être partagée dans toute la boucle. Le compilateur fera une copie locale pour chaque thread (transparent à l'utilisateur). Les copies sont initialisées à 0 au début des calculs. Puis à la fin, toutes les copies seront réduites en une seule variable (la variable partagée) en utilisant l'opération spécifiée (ici `+`).

```
double moy=0.0, A[MAX]; int i;
#pragma omp parallel for reduction (+:moy)
for (i=0;i< MAX; i++) {
moy += A[i];
}
```

Figure 4.1: Syntax

4.1.2 If

Permet de conditionner la création de threads par la valeur d’une condition, si condition=0, la région parallèle va être exécutée par le thread maître en séquentiel. Si condition !=0, création d’un groupe de threads. Cela peut être utile si on souhaite conditionner la création de threads par la taille de la tâche à partager.

4.1.3 Schedule

La clause schedule définit comment les itérations de la boucle sont placées sur les threads.

- schedule (static [,chunk]) : distribue des blocs d’itérations de taille “chunk” sur chaque thread.
- schedule(dynamic [,chunk]) : Chaque thread traite dynamiquement “chunk” itérations . Un thread qui termine aura plus de travail jusqu’à ce que toutes les itérations soient exécutées.
- schedule(guided [, chunk]) : Les threads récupèrent dynamiquement des blocs d’itérations. La taille des blocs diminue selon une fonction exponentielle jusqu’à la taille chunk (après ça devient comme dynamic)
- schedule(runtime) : Récupérer le mode de distribution et la taille des blocs depuis la variable d’environnement OMP_SCHEDULE.

4.1.4 Liées aux status des variables

Comme openMP a été principalement créé pour tourner sur des architectures parallèles à mémoire partagée, le statut par défaut pour les variables déclarées avant les régions parallèles est partagées. Pour spécifier un autre statut on peut utiliser les clauses prévues à cet effet.

4.1.5 Principales clauses de statuts

private : définit une liste de variables privées à chaque thread. Les valeurs de la variable au début et à la fin de la région parallèle sont indéfinies.

firstprivate : private avec initialisation automatique (initialisé par la valeur déclarée avant la région parallèle).

lastprivate : private avec mise à jour automatique. Les variables listées sont mises à jour avec la valeur de la variable privée correspondante à la fin du thread qui exécute soit la dernière itération d'une boucle soit la dernière section par rapport à l'exécution séquentielle.

shared : définit une liste de variables partagées. Tous les threads d'une même équipe peuvent accéder aux variables partagées simultanément (sauf si une directive OpenMP l'interdit, comme atomic ou critical).

Les modifications d'une variable partagée sont visibles par tous les threads de l'équipe (mais pas toujours immédiatement, sauf si une directive OpenMP le précise, comme flush).

default : change le statut par défaut.

reduction : définit une liste de variables à réduire.

Chapter 5

Les fonctions de openMP

5.1 Les fonctions de la bibliothèque openMP

- Pour modifier/vérifier le nombre de threads : `omp_set_enum_threads()`, `omp_get_enum_threads()`.
- Combien y a t il de processeurs dans le système : `omp_num_procs()`.
- Autoriser ou pas l'imbrication des régions parallèles et l'ajustement dynamique du nombre de threads dans les régions parallèles : `omp_set_nested()`, `omp_set_dynamic()`, `omp_get_nested()`, `omp_get_dynamic()`.
- Tester si le programme est actuellement dans une région parallèle : `omp_in_parallel()`.

5.2 Les fonctions de mesure du temps

`omp_get_wtime()`: Retourne le temps écoulé en secondes depuis un temps de référence.

`omp_get_wtick()`: Retourne le temps écoulé en secondes entre deux « tops » d'horloge (indique la précision de la mesure du temps).

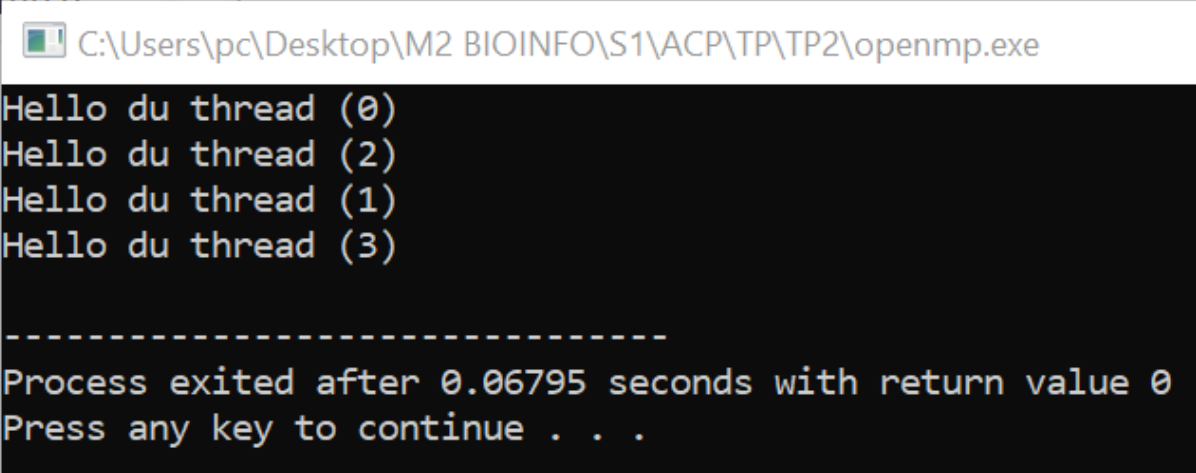
Chapter 6

Exercices

6.1 Exercice 1

Le programme est dans le fichier source "openmp.c".

6.1.1 Compilation et exécution



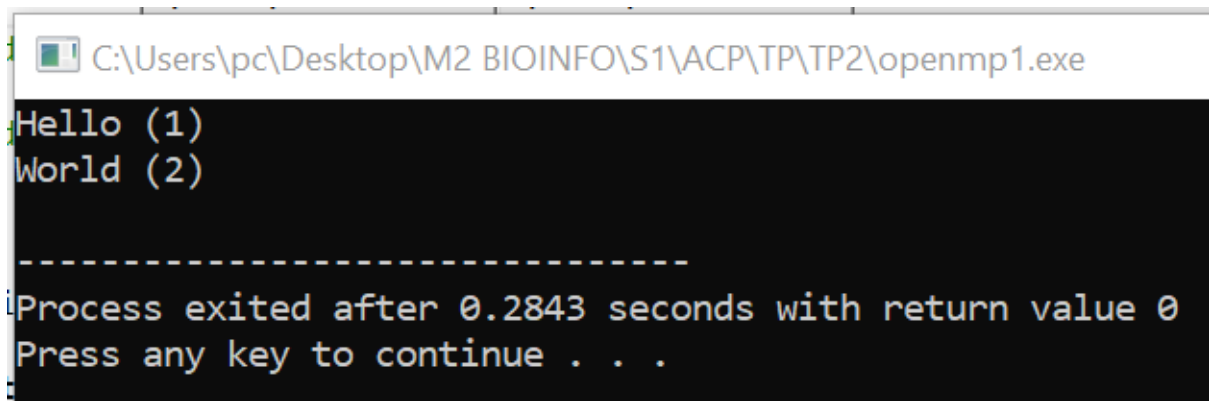
```
C:\Users\pc\Desktop\M2 BIOINFO\S1\ACP\TP\TP2\openmp.exe
Hello du thread (0)
Hello du thread (2)
Hello du thread (1)
Hello du thread (3)
-----
Process exited after 0.06795 seconds with return value 0
Press any key to continue . . .
```

Figure 6.1: Output

6.1.2 L'ajout des sections

Création de deux sections une pour un thread qui va afficher hello (num thread) et une qui affichera world (num thread).

Le programme est dans le fichier source "openmp1.c".

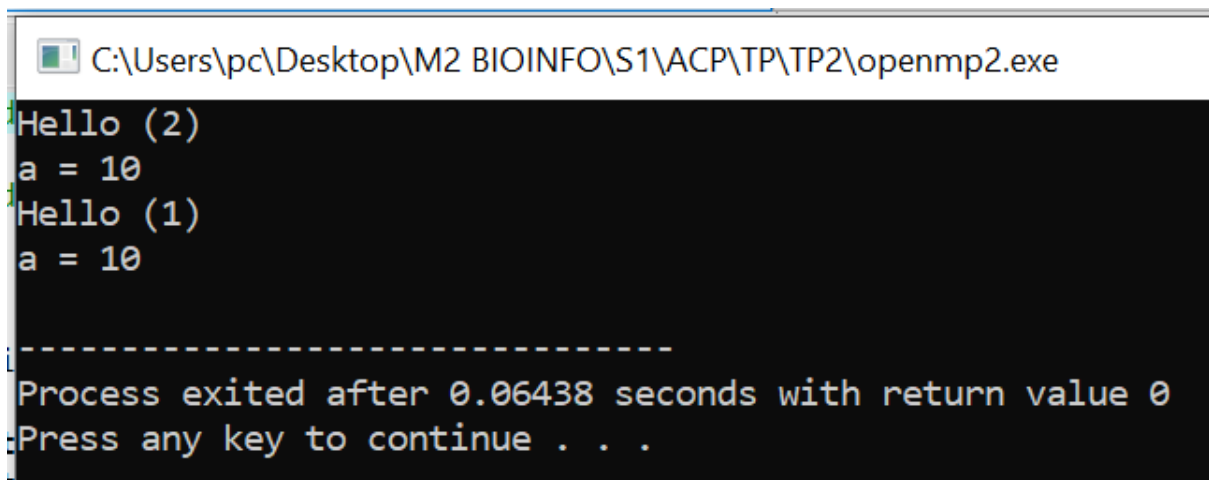


```
C:\Users\pc\Desktop\M2 BIOINFO\S1\ACP\TP\TP2\openmp1.exe
Hello (1)
World (2)

-----
Process exited after 0.2843 seconds with return value 0
Press any key to continue . . .
```

Figure 6.2: Output

Tester les variables partagées, privées. Déclarer une variable **a** avant la section parallèle, l'initialiser à 5 (par défaut elle sera partagée !). Dans la région parallèle, section 1, un des threads va modifier la valeur de **a** un autre thread qui exécute la deuxième section va juste l'afficher. Le programme est dans le fichier source "openmp2.c".

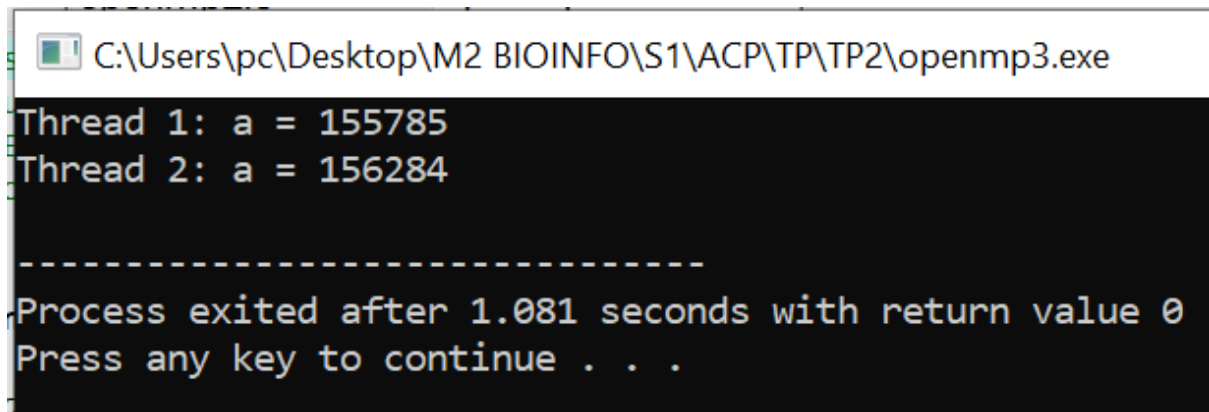


```
C:\Users\pc\Desktop\M2 BIOINFO\S1\ACP\TP\TP2\openmp2.exe
Hello (2)
a = 10
Hello (1)
a = 10

-----
Process exited after 0.06438 seconds with return value 0
Press any key to continue . . .
```

Figure 6.3: Output

Maintenant, toujours la variable **a**, on va la déclarer private pour chaque thread (clause `private(a)`). Afficher la valeur de **a** au début et à la fin de chaque section (ajouter `sleep` pour ralentir l'exécution).



```
C:\Users\pc\Desktop\M2 BIOINFO\S1\ACP\TP\TP2\openmp3.exe
Thread 1: a = 155785
Thread 2: a = 156284
-----
Process exited after 1.081 seconds with return value 0
Press any key to continue . . .
```

Figure 6.4: Output

6.2 Exercice 2 (la clause Schedule)

Le programme est dans le fichier source "exercice2.c".

1- Les instructions exécutées par :

- un seul thread : hors la région parallèle (avant `#pragma omp parallel private(tid)`).
- tous les threads : à l'intérieur de la région parallèle.

2- L'ordre d'exécution des instructions est aléatoire.

3- Redirigez la sortie de l'exécutable sur l'utilitaire sort (sur linux).

4- La répartition n'est pas stable.

5- La répartition est stable.

6- static vs dynamic:

static est une politique de planification OpenMP. Il distribue les morceaux d'itération aux threads disponibles selon une distribution circulaire : premier morceau au premier thread, deuxième morceau au deuxième thread et ainsi de suite.

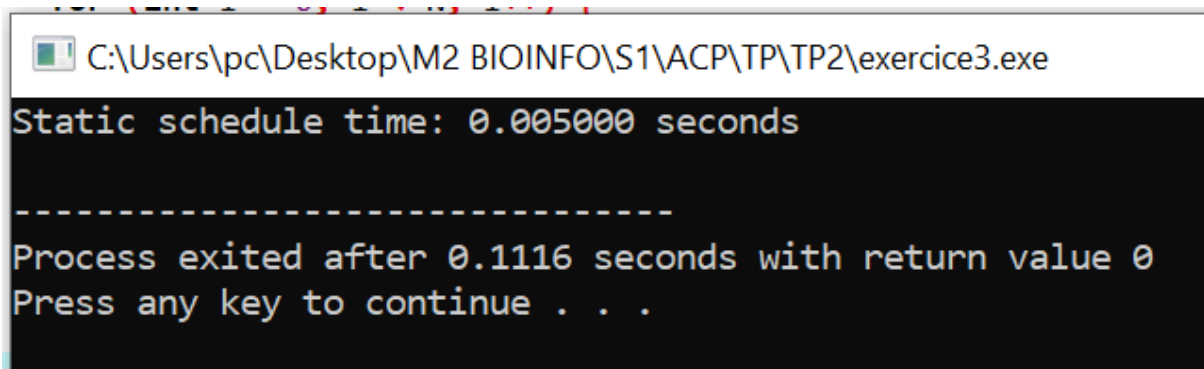
dynamic est une politique de planification OpenMP. La planification dynamique est préférable lorsque les itérations peuvent prendre des durées très différentes. Cependant, la planification dynamique entraîne une certaine surcharge. Après chaque itération, les threads doivent s'arrêter et recevoir

une nouvelle valeur de la variable de boucle à utiliser pour sa prochaine itération.

6.3 Exercice 3

Le programme est dans le fichier source "exercice3.c".

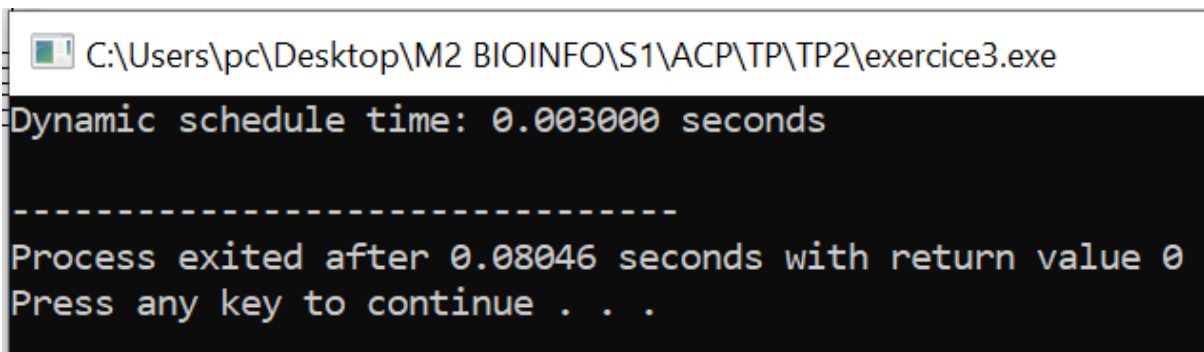
6.3.1 Static schedule



```
C:\Users\pc\Desktop\M2 BIOINFO\S1\ACP\TP\TP2\exercice3.exe
Static schedule time: 0.005000 seconds
-----
Process exited after 0.1116 seconds with return value 0
Press any key to continue . . .
```

Figure 6.5: Output

6.3.2 Dynamic schedule



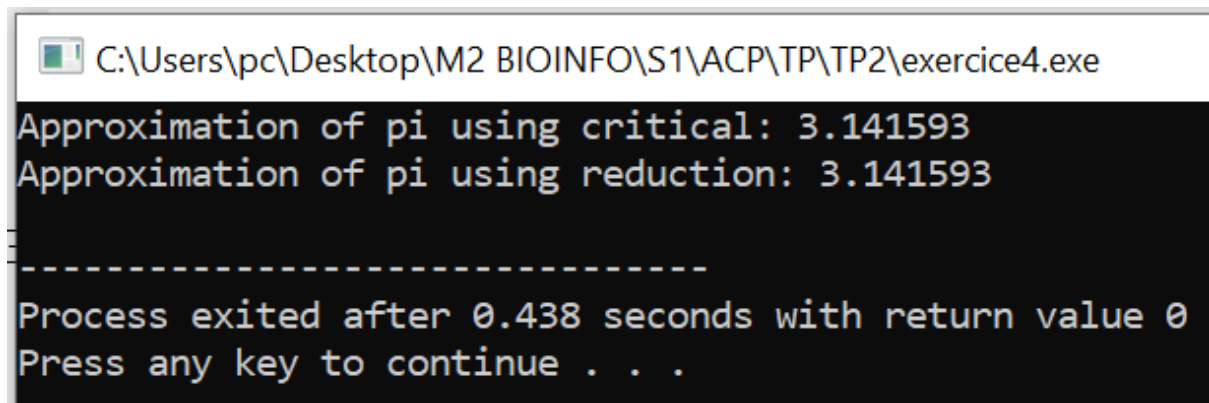
```
C:\Users\pc\Desktop\M2 BIOINFO\S1\ACP\TP\TP2\exercice3.exe
Dynamic schedule time: 0.003000 seconds
-----
Process exited after 0.08046 seconds with return value 0
Press any key to continue . . .
```

Figure 6.6: Output

Le scheduling dynamique est plus rapide que le scheduling statique dans ce cas.

6.4 Exercice 4 (calcul de pi)

Le programme est dans le fichier source "exercice4.c".



A screenshot of a Windows command prompt window. The title bar at the top shows the file path: C:\Users\pc\Desktop\M2 BIOINFO\S1\ACP\TP\TP2\exercice4.exe. The command prompt area has a black background with yellow text. The output of the program is as follows:

```
Approximation of pi using critical: 3.141593
Approximation of pi using reduction: 3.141593

-----
Process exited after 0.438 seconds with return value 0
Press any key to continue . . .
```

Figure 6.7: Output