

TP OpenMP – Calcul Matriciel

0. Serveur « turing »

Nous allons utiliser les serveurs pédagogiques de l'ISIMA pour les TP de « Calcul Parallèle », en particulier le serveur **turing**.

Le serveur **turing** est composé de 4 NUMA (Non Uniform Memory Access) nodes, de 24 coeurs physiques chacun (Intel Xeon Platinum). L'environnement logiciel pour ce cours est le suivant :

- Fedora release 29, gcc 8.3, OpenMP 4.5 (201511), MPI 3.1 – OpenMPI 2.1

1. OpenMP

OpenMP est une bibliothèque qui facilite l'écriture des programmes parallèles multi-threading pour des architectures parallèles à mémoire partagée. Elle fournit des fonctions de gestion de threads (ex. `omp_get_num_threads`) et des directives de compilation (`#pragma`) qui permet la création et la gestion des régions parallèles à partir d'un programme séquentiel.

L'exécution d'un programme OpenMP entraîne l'exécution simultanée de plusieurs threads. Les threads d'un même programme peuvent avoir des variables partagées (i.e. une seule copie mémoire par variable) ou des variables privées (le même nom, mais une copie mémoire par variable et par thread). La gestion des variables partagées / privées est primordiale pour la justesse des résultats du programme et de sa performance. La performance du programme est aussi grandement affectée par la gestion de régions parallèles.

Les versions récentes de **gcc** intègrent la bibliothèque OpenMP. La version d'OpenMP supportée dépendant de la version de **gcc**. L'utilisation de l'option **-fopenmp** permet à **gcc** de compiler vos programmes OpenMP.

2. Prise en main : éditer le programme `hello_omp.c` suivant

```
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <unistd.h>
#include "omp.h"

int main()
{
    int tid=-1;  char hostname[1024];

    gethostname(hostname, 1024);
    printf("Before PARALLEL REGION TID %d: There are %d threads on CPU %d of %s\n\n",
        omp_get_thread_num(), omp_get_num_threads(), sched_getcpu(), hostname) ;

    #pragma omp parallel num_threads(4) firstprivate(tid)
    {
        tid=omp_get_thread_num();
        if (!tid)
            printf("In the PARALLEL REGION TID %d: There are %d threads in process\n",
                omp_get_thread_num(), omp_get_num_threads());
        printf("Hello World from TID %d / %d on CPU %d of %s!\n\n",
            tid, omp_get_num_threads(), sched_getcpu(), hostname);
    }

    printf("After PARALLEL REGION TID %d: There are %d threads\n\n" ,
        tid, omp_get_num_threads()) ;

    return EXIT_SUCCESS ;
}
```

- Compiler le programme avec : **gcc -fopenmp hello_omp.c -o hello_omp** ; Vous pouvez ajouter les options habituelles de compilation.
- Exécution directe du programme **hello_omp** :
`$./ hello_omp`
- Interpréter le résultat d'exécution : qui (thread) exécute quoi (instruction) ?
- Modifier le nombre de threads en utilisant l'une des méthodes suivantes :
 - o la clause `num_threads` de `#pragma omp parallel`
 - o la variable d'environnement `OMP_NUM_THREADS`
 - o la fonction `void omp_set_num_threads(int num_threads) ;`
- Si on remplace la clause `firstprivate` par `private`, que peut-il se passer ?
 Pour constater le changement, commenter la ligne `tid=omp_get_thread_num()` ; et re-exécuter plusieurs fois le programme.
- Supprimer la clause `private`, refaire les exécutions, que constatez vous ?

3. Calcul matriciel

Soient A , B deux matrices carrées d'ordre n , on demande d'écrire un programme qui calcule $C=A \times B$.

- 3.1. Implémenter un programme séquentiel, la taille du problème sera passé en paramètre sur la ligne de commande. Vérifier les résultats obtenus.
- 3.2. Identifier les parties du programme adaptées à la parallélisation, puis utiliser les directives OpenMP pour réaliser cela.
- 3.3. Pour chaque variable concernée par la parallélisation, indiquer elle doit être privée ou partagée.
- 3.4. Vérifier que le programme parallèle donne des résultats corrects en comparant ses résultats avec ceux du programme séquentiel, et en faisant varier le nombre de threads et la taille de matrices.
- 3.5. Vérifier la répartition du calcul entre les threads en utilisant le numéro de thread employé pour chaque itération, ceci pour différent type d'ordonnancement `schedule(dynamic/static)`.
- 3.6. Mesurer le temps d'exécution du programme parallèle (en variant le nombre de threads et la taille des matrices), puis calculer la performance de la parallélisation en utilisant le facteur d'accélération du programme (voir l'annexe).
- 3.7. Analyser et comparer les résultats des différentes parallélisations.

P.S. : Utiliser éventuellement le processus itératif $A_{n+1}=A_n \times B$ afin que le volume de calcul soit plus conséquent.

Annexe : Evaluation de performance d'un programme parallèle

La performance d'un programme parallèle s'évalue en comparant le temps d'exécution parallèle au temps d'exécution séquentielle. L'évaluation commence par la mesure du temps écoulé entre le début du 1^{er} thread (ou processus) et la fin du dernier thread (ou processus) du programme.

On mesure le temps d'exécution en variant le nombre de threads (ou processus) utilisés et la taille du problème. Plusieurs mesures sont à réalisées pour chaque couple (nombre de threads, taille du problème). La moyenne de ces mesures sera considérée comme le temps d'exécution de ce cas.

L'ensemble de mesures est alors stocké dans un tableau. A partir de ces mesures, on peut tracer les courbes du temps d'exécution selon la taille et/ou selon le nombre threads. On peut aussi calculer le facteur d'accélération (speedup), l'efficacité du programme parallèle, etc.

Attention : les mesures doivent être effectuées sur le même nœud, sous les mêmes conditions. Elles peuvent être polluées si plusieurs threads (ou processus) s'exécutent simultanément sur un même core (ou socket / node). L'hyper-threading des cores est une des causes qui peut fausser la mesure du temps d'exécution. Il est désactivé sur le cluster. Bref, la condition de **turing** n'est pas idéale pour la mesure de performance d'un programme parallèle.

Il est important d'utiliser le contrôle de l'affinité des threads pour avoir des mesures stables. En OpenMP, vous pouvez utiliser les variables d'environnement OMP_PROC_BIND (=true) et OMP_PLACES ("{0},{4},{8},{12},{16},{20},{24},{28}") pour avoir 1 thread par core sur 8 cores du NUMA node 0 de **turing**. Cela évite qu'un thread soit affecté à des cores différents au cours d'une exécution. Vous pouvez ainsi manuellement éviter le hyperthreading.

Exemple :

Tableau 1: Le temps d'exécution d'un programme parallèle en fonction de la taille du problème et du nombre de threads employés

Nb_threads Taille	1	2	4	8	16
256	0,22	0,16	0,13	0,06	0,03
512	1,88	1,19	0,77	0,36	0,18
768	7,09	3,71	2,64	1,02	0,57
1024	16,29	8,24	6,35	2,36	1,27
1280	31,92	17,33	10,60	4,72	2,58
1536	60,50	36,28	22,07	9,52	5,01
1792	105,13	67,82	33,59	17,83	9,33
2048	158,88	105,99	64,40	28,90	15,17

Tableau 2 : Le speedup du programme en fonction du nombre de threads et de la taille du problème

Nb_threads Taille	1	2	4	8	16
512	1	1,58	2,43	5,15	10,21
1024	1	1,98	2,57	6,89	12,83
1536	1	1,67	2,74	6,36	12,08
2048	1	1,50	2,47	5,50	10,47

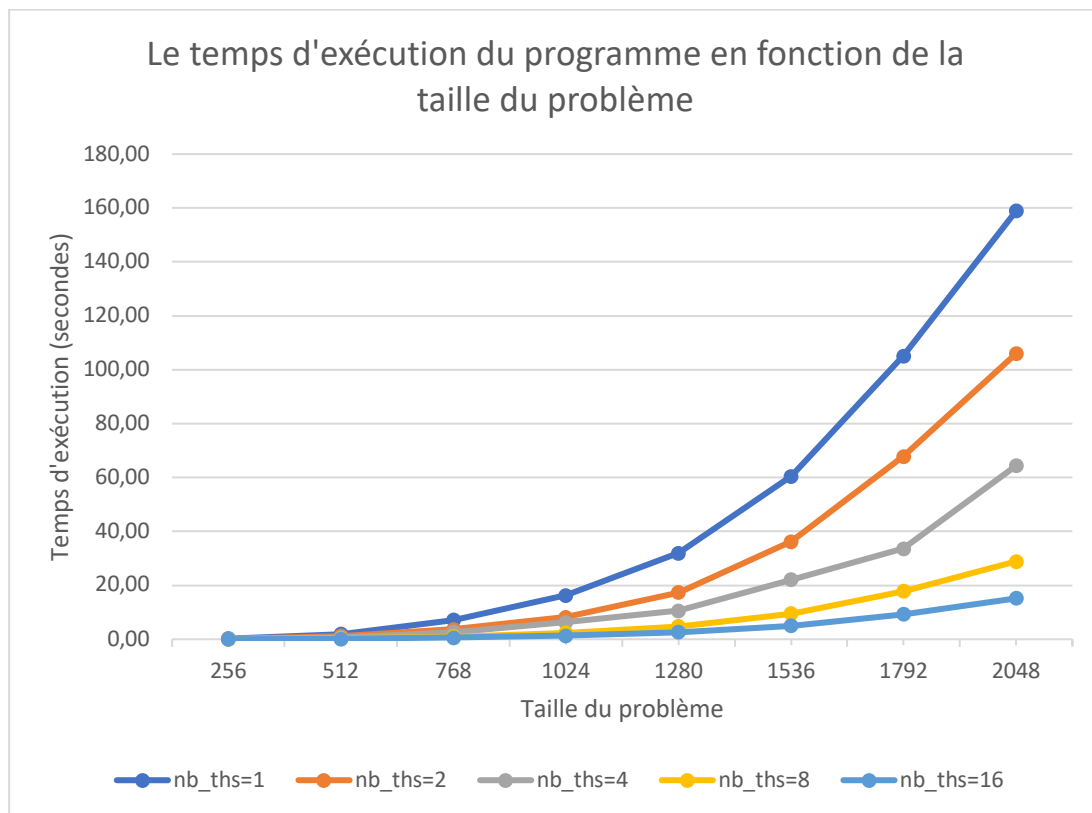


Figure 1 : L'évolution du temps d'exécution d'un programme parallèle en fonction de la taille du problème

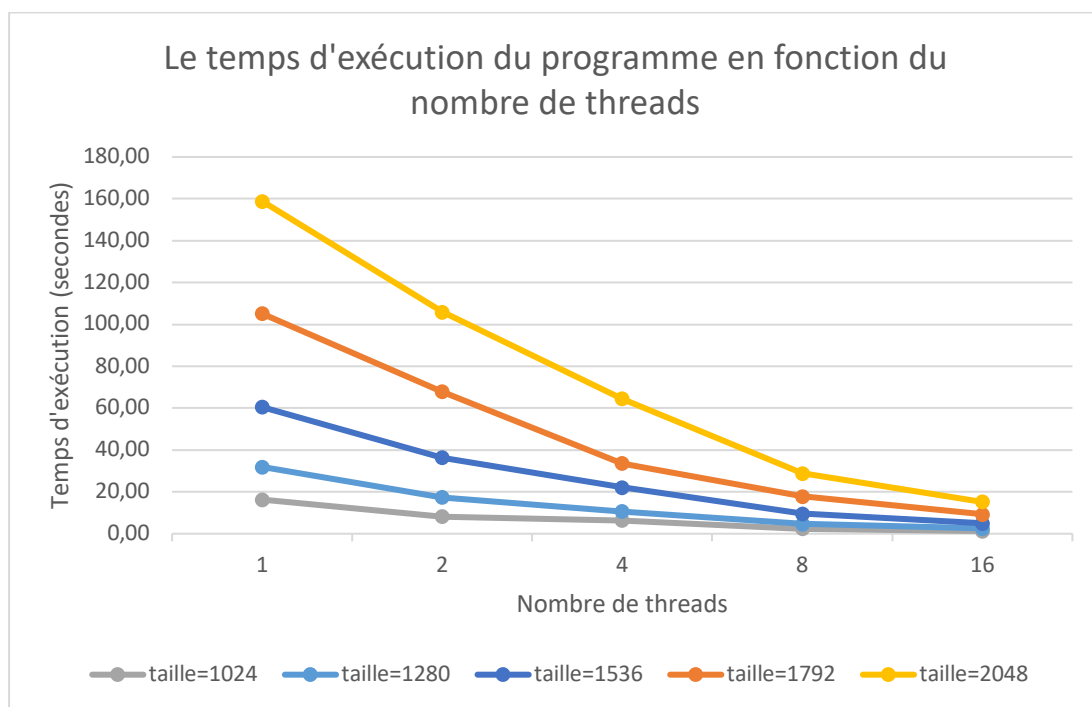


Figure 2 : L'évolution du temps d'exécution d'un programme parallèle en fonction du nombre de threads utilisés

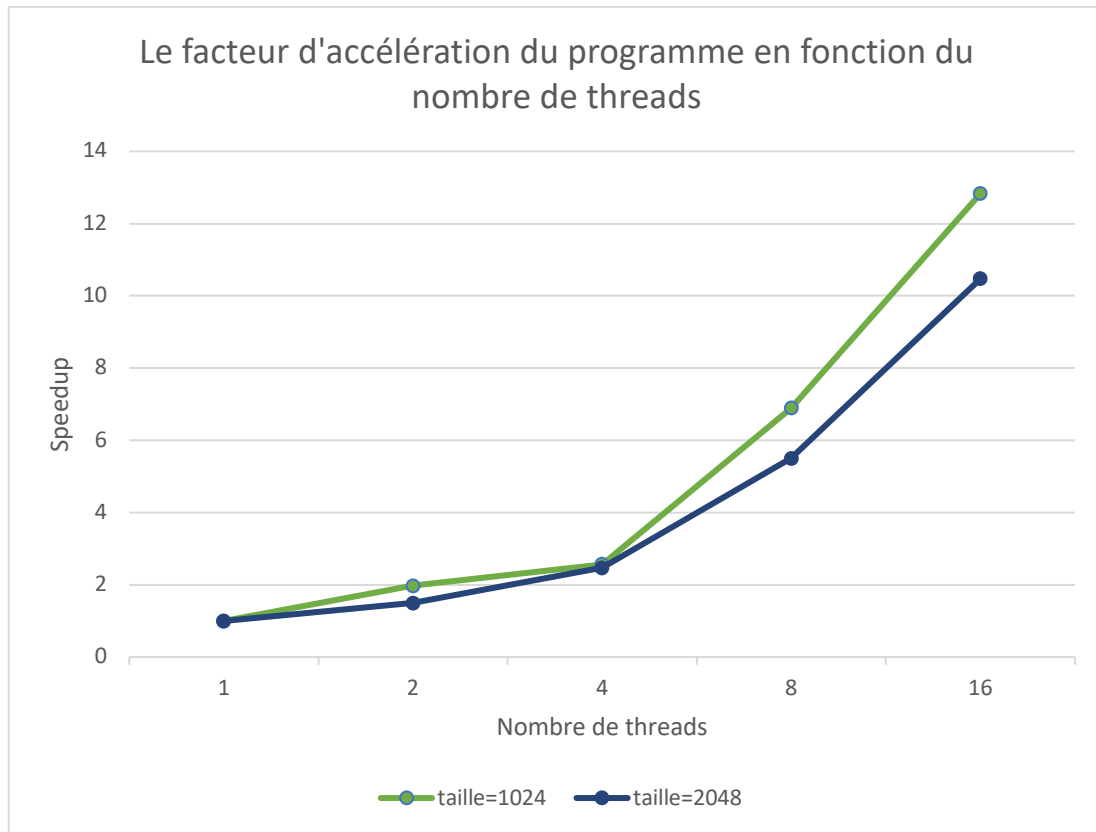


Figure 3 : Le facteur d'accélération du programme parallèle en fonction du nombre de threads utilisés pour la taille de matrice 1024x1024 et 2048x2048