M1 / MIHPS / U13 / TP6
Techniques d'optimisation de la parallélisation
Problèmes de performances avec OpenMP

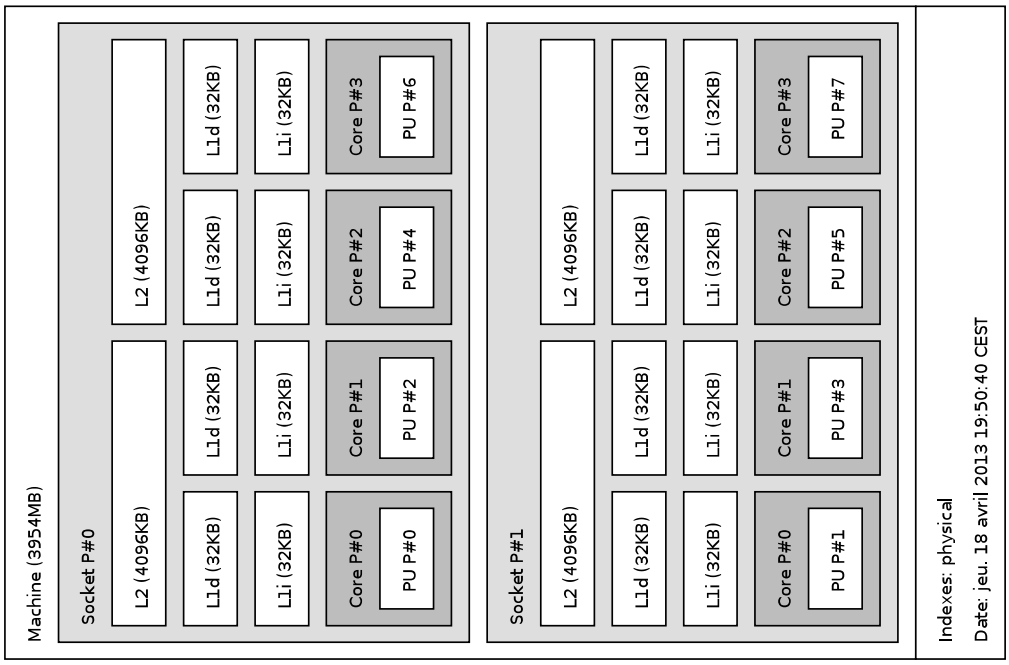
1) Objectif

Pour cette séance, nous allons principalement essayer de mettre en évidence certains problèmes de performances qui peuvent survenir lors de l'utilisation de code parallèle en mémoire partagée. Les différents problèmes seront étudiés ici sous la forme de microbenchmarks afin de mieux comprendre les phénomènes.

2) Topologie des stations utilisées

Les optimisations réalisées en parallèle doivent tenir compte de certains phénomènes liés à la topologie des stations utilisées.

Pour connaitre la topologie de votre machine, vous pouvez utiliser l'outil hwloc : http://www.open-mpi.org/projects/hwloc/. La commande « lstopo » vous donne la structure de vos processeurs, par exemple sur centre :



3) Synchronisation et false-sharing

3.a) Problème initial

Nous allons regarder la parallélisation d'une réduction de type somme. Le problème est simplifié à l'extrême par rapport à ce que vous avez pu faire en TP de programmation parallèle. Nous allons considérer le calcule d'une somme des éléments d'un vecteur d'entier.

```
int sum = 0;
int * values = malloc(.....);

for ( i = 0 ; i < size ; ++i)

//..
for ( i = 0 ; i < size ; ++i)
      sum += values[i];
//..
```

Implémenter ce microbenchmark et mesurez son temps avec la fonction getticks() fournit par le fichier cycle.h de la bibliothèque FFTW. Une version est copiée dans : */users/doinf-2011/20908769/cycle.h.*

Assuez-vous de bien afficher le résultat de la somme pour que le compilateur n'élimine pas la boucle.

Donnez un temps mesuré en cycles par éléments pour plus de lisibilité, on ne mesure que la boucle de somme.

3.b) Options de compilation

Réalisé la mesure en compilant avec l'option -O0 et -O3, que constatez-vous ?

3.c) Parallélisation OpenMP

Une fois que votre benchmark est implémenté et que vous avez obtenu un temps de référence, parallélisez votre programme à l'aide d'OpenMP. Pour ce TP, nous allons ignorer les primitives de réduction d'OpenMP et implémenter la solution à la main en utilisant uniquement les pragma parallèles et for.

Dans un premier temps, mesurez la performance obtenue en maintenant la variable *sum* globale avec l'utilisation des primitives de synchronisation :

- *Sans synchronisation*
- *#pragma omp critical*
- *#pragma omp atomic*

Quelle remarque pouvez-vous formuler ? Expliquez la source du phénomène.

3.d) Problème du « atomic » dans ce cas ?

Vous l'avez vu en TP de programmation parallèle, l'utilisation des *atomics* n'est pas une bonne solution dans notre cas, pourquoi ?

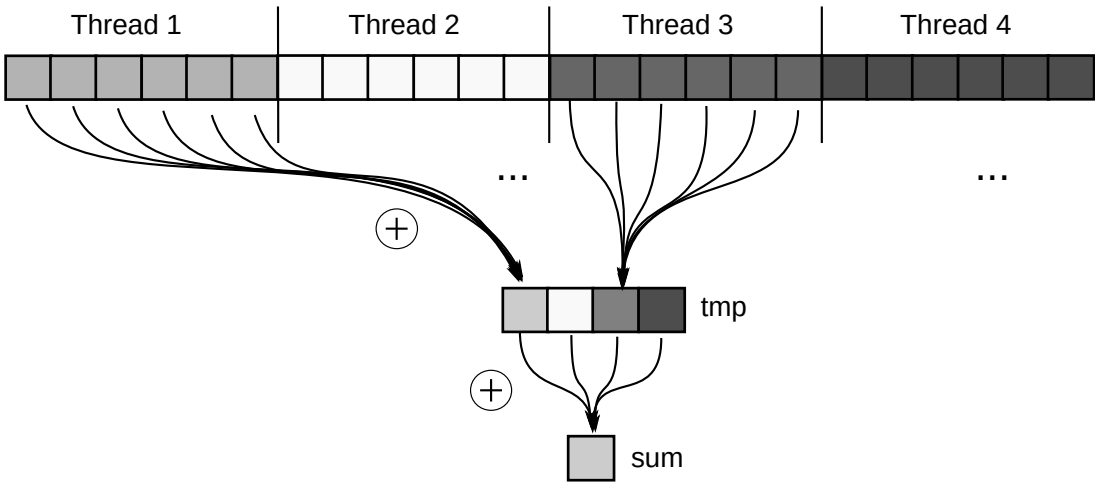
Expliquer le problème qui se pose ?

3.e) Conflits en lecture versus écriture

Modifier le benchmark utilisant la méthode atomique et inverser les termes de l'équation de sortie à lire 'sum' au lieu de l'écrire. Que remarquez-vous ?

3.f) Réduction par un tableau

Toujours sans utiliser les options d'OpenMP, implémenter la somme à l'aide d'un tableau intermédiaire indexé par l'ID du thread.



Mesurez à nouveau la performance. Que remarquez-vous ?

3.g) False-sharing

Ajoutez du padding entre les éléments du tableau temporaire de façon à ce que chaque élément utilisé soit sur sa propre ligne de cache. On rappelle qu'une ligne de caches occupe 64o sur les processeurs Intels et AMD.

Testez également pour des valeurs plus faibles de paddings. Que remarquez-vous ?

Comment peut-on expliquer ce phénomène ?

Que peut-on en conclure quand aux précautions à prendre dans la définition des structures manipulées dans un programme parallèle ?

3.h) Réduction OpenMP

Testez maintenant la réduction OpenMP. Comparez à vos performances et complétez le tableau suivant :

Méthode	Temps séquentiel	Temps parallèle
Séquentielle -O0	7.985049	-
Séquentielle -O3	0.870956	-
#pragma omp	2.098920	0.730559
#pragma omp critical	48.713608	360.131622
#pragma omp atomic	21.311146	90.611580
#pragma omp atomic (read)	19.770330	5.312098
Table intermédiaire	2.106674	1.127297
Table intermédiaire + padding	2.125977	0.729299

3.i) Bande passante

Comparez votre version parallèle à votre version séquentielle sans OpenMP, que pouvez-vous dire ? Introduisez des calcules type exponentiel, sin et/ou cos dans la boucle de calcule, que remarquez-vous ?

3.j) Limites

Quelques sont les limites de notre implémentation finale si l'on devait utiliser un grand nombre de cœurs ? Proposez (sans implémenter) des alternatives.

4) Malloc et false-sharing.

Dans un nouveau programme, effectuez deux appels à malloc() pour des tailles de l'ordre de 16 octets et affichez la différence des adresses obtenues. Quelle remarque pouvez-vous faire quand aux sources d'apparition des problèmes de false-sharing dans une application complexe ?

5) Partage de ressource

Essayer de lancer un des microbenchmarks précédents avec N+1 threads OpenMP, N étant le nombre de cœurs de votre station. Que constatez-vous, expliquez ?

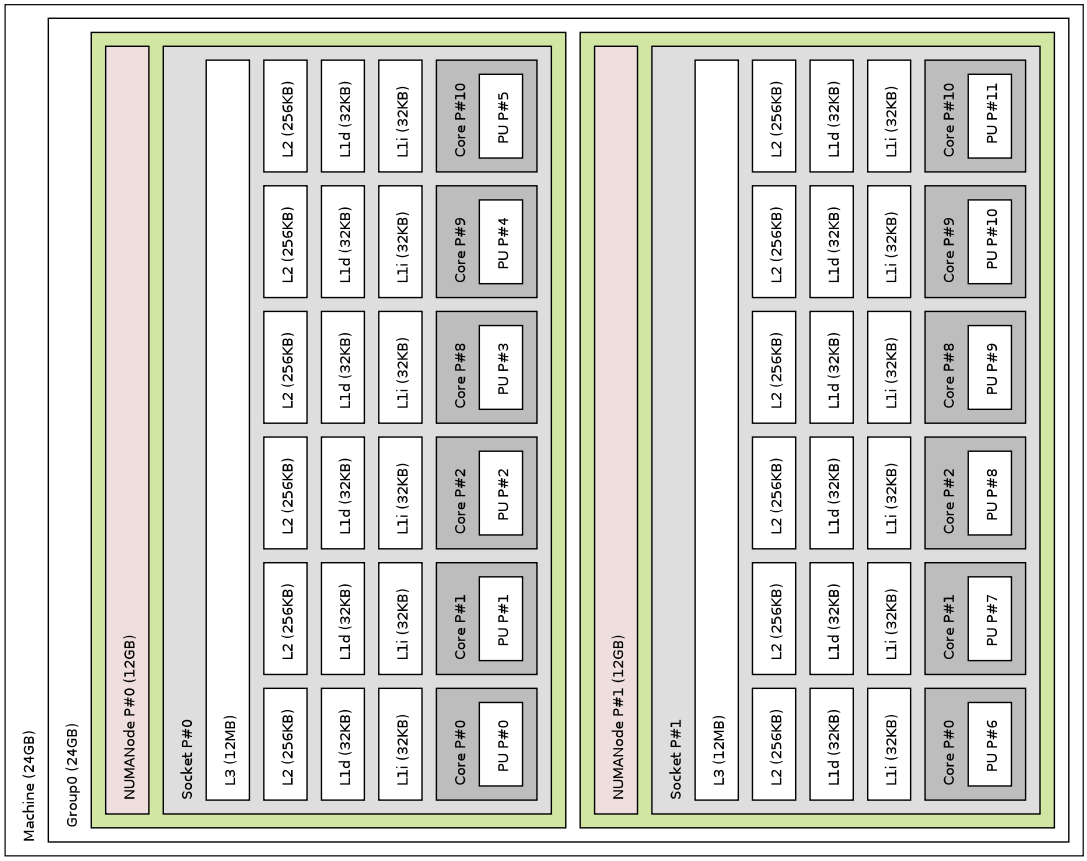
6) Initialisation mémoire en OpenMP en mode NUMA

On se propose d'exécuter le code suivant sur une architecture du type suivant. Quelle erreur a faite le developpeur de ce code :

```
int main(void)
{
      double * buffer = malloc(SIZE * sizeof(double));
      double sum;

      for (int i = 0 ; i < SIZE ; i++)
            buffer[i] = i;

      #pragma omp parallel reduce(+:sum)
      for (int i = 0 ; i < SIZE ; i++)
            sum += buffer[i];
}
```



7) Malloc, thread-safe ou multithread ?

7.a) Allocateur de Linux

On se propose de tester les performances de la fonction malloc() de Linux en effectuant des allocations intensives de manière parallèle. Pour cela, on implémentera un microbenchmark basé sur les mêmes principes que ce que l'on a fait jusqu'à maintenant.

- Déclarez un tableau de 64k-éléments de type char * (char * bloc[64*1024]).
- Initialisez-le à 0 avec un memset.
- On allouera pour chaque élément du tableau un segment mémoire de 32o.
- Rendez ces allocations parallèles grâce à OpenMP.
- Répétez l'opération 1000 fois. Si le tableau contient un pointeur déjà alloué, libérez-le avant de poursuivre.

On obtient ainsi une suite d'allocations/libérations qui nous permet de tester malloc/free. Un tel schéma d'allocation n'est pas très réaliste, mais suffira pour nos tests.

Mesurez les performances sur 1,2,4 ou 8 threads. Avec OpenMP, vous pouvez limiter le nombre de threads à l'exécution de la manière suivante :

```
OMP_NUM_THREADS=8 ./mon_program
```