HPC - TME 4/5 - OPENMP

Diez Marie

Table des matières

1 Introduction

Nous avons travaillé avec MPI qui permet d'executer des programmes indépendant ou un suite d'instruction indépendante sur différent coeurs ou noeuds de calcule. Si MPI est lancé sur différents noeuds de calcule, alors OpenMP va permettre d'utiliser les différents coeurs du noeud en lancant N threads. OpenMP tourne au sein d'un processus qui utilise les coeurs disponibles de la machine en lancant N threads.

Utilisation :

- Personnel :
  - MPI lance N processus associé à N threads avec donc 1 thread par processus.
  - OpenMP tourne au sein d'un processus avec N threads disponible.
  On peut avoir autant de threads qu'on veut mais avec 4 coeurs on a 4 threads en meme temps.
- Déployé :
  - MPI lance N processus associé à N noeuds avec donc 1 thread par noeud.
  - OpenMP tourne au sein d'un noeuds qui à N coeurs et donc N threads disponibles.
  simultanément. On peut assigné plus de Threads que de coeurs disponibles mais on verra les performances diminuée pour la gestion de ces threads par la machine.

2 Parrallélisation simple

Tous les testes de performance ont été effectué sur mon ordinateur personnel avec un processeurs $Ryzen53500U$.

2.1 Calcule de Pi

Lecalcule de π peut se faire complètement en parrallèle chaque thread executera un calcule indépendant des autres grâce à une instruction $parallel for$ de OpenMP. On met en place une réduction sur π de type $(+ : pi)$ pour obtenir la valeur final.

On peut mettre en place différent type de parrallélisme graĉe à l'option *schedule*, dynamique / static / taille des blocs...

En compilant avec $-fopenmp$ on passe de :

- static, taille de bloc : 1
  5.35sec à 1.3 sec
- static, taille de bloc : 100
  5.35sec à 1.15 sec
- dynamic, taille de bloc : 1
  5.35sec à 20 sec
- dynamic, taille de bloc : 100
  5.35sec à 1.3 sec

On remarque que le choix des ces paramètres est important et va dépendre du problème à résoudre. Ici comme nous avons un problème simple la version dynamique est plus longue en mettre en place pour la gestion. On peut alors prendre une version static, une taille de bloc trop petite entraine une perte de performance, il n'est pas non plus utile de choisir des blocs de grands, sinon on risque de perdre l'utilité d'avoir plusieurs processus.

2.2 Calcule matriciel

Il est plus avantageux de parralléliser la boucle la plus externe avec OpenMP pour éviter de lancer parrallélement encore plus de thread ce qui est lourd. Il faut pensait à mettre en privé les variables qui doivent être propre à chaque thread :
Dans la triple boucle du calcule d'indice respectif i, j, k il faut mettre en privé les variables j, k avec la clause $private(j, k)$.

- Avec uniquement un appel à *ompparallel* avec la clause *private*
  2sec à 0.6sec
- Avec un schedule *static* et une taille bloc : 1
  2sec à 0.5sec

Lorsque l'on augmente la taille des blocs les performances diminue.

3 Récursivité et parrallélisation

3.1 Fibonacci

Pour parralléliser avec OpenMP des programmes récursif on utilise les tâches. L'idée est de lancer le programme avec un seul thread grâce à l'instruction *ompsingle* :

```
int res;
/* Do computation:  */
#pragma omp parallel
{
    #pragma omp single
    res = fib(n);
}
```

Puis lancer un thread pour faire la première partie du calcule et un thread pour faire l'autre, il faut attendre que les 2 ont fini avant de faire le *return*. On peut mettre en place un grain qui permet de ne pas faire les appels aux instructions OpenMP pour lancer des tâches si il ne reste que peux de calcule à faire, ce qui permet d'éviter les sur-coût lié au parrallélisme (gestion des threads, attente...)

```
int grain = 10;
if (n < 2)
    return n;
else {
    int i, j;
    #pragma omp task shared(i) if(n >= grain)
    i = fib(n - 1);
    #pragma omp task shared(j) if(n >= grain)
    j = fib(n - 2);
    #pragma omp taskwait
    return i + j;
}
```

Les performances sont très mauvaises en parrallèle (je n'ai pas compris pourquoi) :

- n=45 11sec en sequentiel et 57s en parrallèle.

3.2 QuickSort

Pour le QuickSort l'idée est la même que précédemment, on dit à un des thread de lancer le premier appel au QuickSort avec *ompsingle* :

```
#pragma omp parallel
{
    #pragma omp single
    QuickSort(tableau, 0, taille - 1);
}
```

Puis on sépare le calcule en 2 parties : un thread s'occupera de la partie Gauche du tableau et l'autre de la partie Droite et ainsi de suite récursivement tant que la taille du tableau est supérieur au grain fixé :

```
void QuickSort(int tableau[], int debut, int fin)
{
    int grain = 50;
    int gauche = debut - 1;
    int droite = fin + 1;

    /* Si le tableau est de longueur nulle, il n'y a rien    faire. */
    if (debut >= fin)
        return;

    const int pivot = tableau[debut];    // premier element choisi comme pivot

    /* Sinon, on parcourt le tableau, une fois de droite    gauche, et une
       autre de gauche    droite,    la recherche d'  lments    mal plac s,
       que l'on permute. Si les deux parcours se croisent, on arr te. */
    while (1) {
        do
            droite--;
        while (tableau[droite] > pivot);
        do
            gauche++;
        while (gauche <= fin && tableau[gauche] <= pivot);

        if (gauche < droite)
            echanger(tableau, gauche, droite);
        else
            break;
    }

    /* On met le pivot    sa place: */
    echanger(tableau, debut, droite);

    /* Maintenant, tous les    lments    inf rieurs au pivot sont avant ceux
       sup rieurs au pivot. On a donc deux groupes de cases    trier. On utilise
       pour cela... la m thode quickSort elle-m me ! */
    #pragma omp task if ((fin-debut) >= grain)
    QuickSort(tableau, debut, droite - 1);
    #pragma omp task if ((fin-debut) >= grain)
    QuickSort(tableau, droite + 1, fin);

}
```

Performance :

- n = 26 et grain = 50
  17sec en sequentiel à 5sec.

4 Code

Le code est sur Github : https://github.com/MarieDiez/HPC_TME4_5