Introduction au calcul parallèle avec la bibliothèque OpenMP

Jérémie Crenne

Mars 2021

1 Introduction

Dés la fabrication du tout premier processeur commercial au début des années 70 (Intel 4004), l'augmentation du nombre de transistors intégré et intégrable poursuit une seule et même voie : celle d'accélérer les temps de traitement. Limités par une augmentation en berne des fréquences d'horloges, les architectes se sont tournés vers d'autres techniques d'optimisation comme la multiplication des coeurs de calculs.

Depuis quelques années déjà, la disponibilité des processeurs dits "multi-coeur" à rendu possible l'exécution efficace de programmes en parallèle. Cependant, de nombreuses problématiques apparaissent dés que l'on raisonne de cette façon. En effet, un programme n'exploite pas de manière inhérente le parallélisme disponible dans les architectures à base de microprocesseur. Pour ce faire, il est nécessaire de les programmer en conséquence.

OpenMP est une bibliothèque fournissant un ensemble de fonctions et de directives compilateur (aussi appelées pragma) permettant la programmation des architectures possédant plusieurs coeurs de calculs et qui offre ainsi une méthode d'implémentation de haut-niveau de la programmation parallèle.

Cette séance de TP est une introduction à la programmation avec OpenMP. Nous allons effectuer quelques expérimentations autour des problématiques liées à la programmation parallèle en utilisant le C comme langage de programmation.

N.B : A chaque question posée (qui s'y prête) dans ce sujet, il vous ai demandé de faire le lien entre le logiciel et l'architecture matérielle sous-jacente. Soyez aussi précis que possible dans la rédaction de vos réponses. N'hésitez donc pas à utiliser des tableaux et des schémas pour bien expliquer les situations rencontrées. Une attention toute particulière sera portée au style de votre rapport sachant qu'un rapport rédigé en anglais est un plus !

2 Principes de la programmation parallèle avec OpenMP

Il existe deux raisons largement répandues pour lesquelles implémenter un programme en parallèle a du sens :

— Lorsque l'on travaille sur des données (data based) et qu'il existe un besoin d'effectuer des opérations mathématiques sur un grand ensemble de données. L'utilisation de la programmation parallèle permet de réduire le temps d'exécution

— Lorsque l'on travaille sur des tâches (task based) et qu'il existe un besoin d'exécuter une tâche en parallèle d'une autre. Comme dans un programme qui implémente une interface graphique par exemple. Si l'on imagine que celui-ci exécute également une série de calculs puis doit afficher des résultats dans l'interface, il faut considérer que l'utilisateur puisse toujours interagir avec celle-ci même si les calculs ne sont pas terminés. Là ou la programmation séquentielle ne peut résoudre ce problème, l'usage de la programmation parallèle, elle, le peut

Un programme séquentiel (simple-fil ou single-threaded) s'exécute comme montré en Figure 1, tandis qu'un programme qui exécute du code en parallèle (multi-fil ou multi-threaded) s'exécute comme décrit en Figure 2. Quelques concepts sont à relevés à partir de la Figure 2.

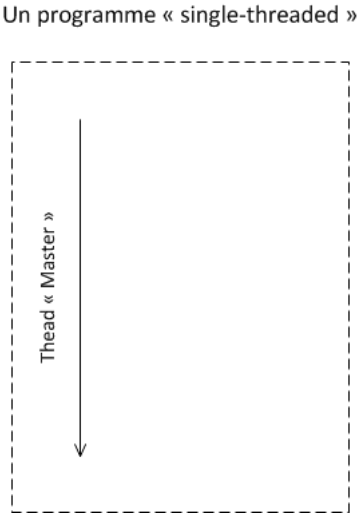

FIGURE 1 – Principe de l'exécution séquentielle d'un programme

OpenMP utilise le modèle *fork-and-join*. Cela signifie que :

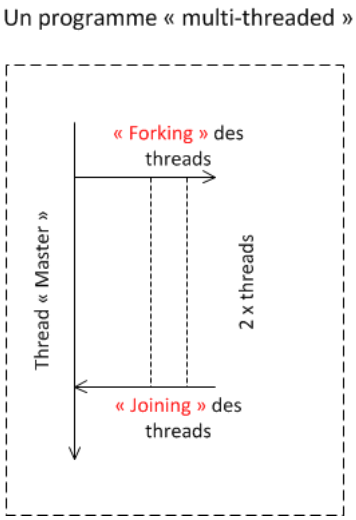

FIGURE 2 – Principe de l'exécution en parallèle d'un programme

— Le thread principal (ou master), créé (fork ou split) un certain nombre de threads esclaves (ou workers)

— Les threads exécutent le code d'une région parallèle en effectuant les calculs nécessaires et les mises à jour des variables

— Les threads rejoignent (join ou merge) le thread principal à la fin de la région parallèle

N.B : Il est recommandé en cas de doute concernant le nombre de threads esclaves à utiliser, de créer autant de threads qu'il y a de coeurs disponibles. Il est possible de choisir le nombre de threads qu'OpenMP va créer et de connaitre le nombre maximum de threads qu'il est possible d'exécuter un parallèle par un appel au fonctions suivantes :

```
omp_set_num_threads( # de threads a utiliser );
# de threads max = omp_get_max_threads();
```

Une problématique importante de la programmation parallèle concerne l'accès aux variables. En effet, que se passe t-il lorsque plusieurs threads accèdent à une même variable ? Les variables dans une région parallèle peuvent être de deux sortes :

— **Privées** : elles peuvent être explicitement déclarées lors de la définition d'une région parallèle à l'aide du mot clé *private(ma variable)*. Elles sont donc locales à chaque thread. Elles ne peuvent pas être accédées par

les autres threads esclaves. Chaque thread créé donc sa propre copie de la variable. Cela élimine complètement les problèmes de situation de compétition (race competition). Par défaut, les variables déclarées dans une région parallèle sont privées

— **Partagées** : elles peuvent également être déclarées lors de la définition d'une région parallèle avec le mot clé *shared(ma variable)*. Les threads esclaves partagerons alors la même variable en mémoire. Par défaut, toutes les variables définies en dehors d'un région parallèle sont partagées

OpenMP permet également l'utilisation des clauses de réductions dont la syntaxe est la suivante :

```
reduction( operateur: nom variable )
```

Il s'agit d'une implémentation des régions dites *atomiques* (*atomic regions*). Une telle région permet d'éviter que plusieurs threads provoquent des écritures dans des variables partagées et assure qu'un seul thread effectue une seule écriture à la fois. Cela fonctionne en deux temps :

— Au début d'une région parallèle, chaque thread esclave créé une copie locale d'une variable partagée. Il initialise également cette variable dépendamment de l'opérateur utilisé.

— A la fin de la région parallèle, la copie locale est fusionnée avec la variable partagée en utilisant l'opérateur qui est définit en clause de réduction. Cela permet de s'assurer qu'aucune donnée provenant des threads esclaves ne soit perdue à cause des situations de compétitions.

Un bon nombre d'opérateurs peut être utilisé comme l'addition, la soustraction, la multiplication ou encore les opérateurs logiques type AND, OR,... Voici un exemple d'utilisation avec une parallélisation d'une boucle for :

```
#pragma omp parallel for reduction( +: mavariablepartagee )
```

3 Exercices de programmation parallèle

3.1 Pré-requis

Travail :

— Q1 : expliquez la différence qu'il y a entre un coeur physique et un coeur logique de microprocesseur
— Q2 : donnez une définition de la notion de fil ou thread
— Q3 : expliquez qui est responsable de la répartition des threads sur les différents coeurs de calcul. Schématiquement, comment cela est il ef-

fectué ? (Utiliser un schéma représentant les différents coeurs de calcul et un chronogramme doit pouvoir vous aider)
— Q4 : théoriquement, pour l'architecture avec laquelle vous travaillez, combien de threads (au maximum) pourront être potentiellement exécutés en parallèle ?

3.2 Exercice 1

Travail :

— Q1 : étudiez le fichier *makefile* et expliquez comment sont liés les fichiers objets avec la bibliothèque OpenMP
— Q2 : quel fichier d'en-tête est utilisé pour permettre l'appel aux fonctions OpenMP dans un fichier source
— Q3 : lancez un terminal et utilisez la commande *make all*
— Q4 : exécutez le binaire avec l'option Test1 (./openmp Test1) et donnez une explication précise du fonctionnement de ce programme
— Q5 : vérifiez, en pratique, le nombre de threads maximum qu'il est possible d'exécuter en parallèle
— Q6 : complétez le programme du Listing 1 pour forcer la création de huit threads. Vérifier le bon fonctionnement du programme

```
void OpenmpTest1() {
        printf( "Master thread\n" );
        #pragma omp parallel
        {
                printf( "Worker thread\n" );
        }
        printf( "Master thread\n" );
}
```

Listing 1 – Un exemple d'utilisation d'OpenMP

3.3 Exercice 2

On souhaite écrire un programme qui permet de créer quatre threads qui exécutent chacun l'affichage de l'identifiant du thread en cours d'exécution. Lorsque l'identifiant de ce thread vaut 3 et uniquement 3, le thread concerné affiche le message suivant : "Bonjour je suis le thread 3".

Travail :

— Q1 : écrivez ce programme en vous inspirant du programme du Listing 1 précédent,

— Q2 : vérifier le bon fonctionnement de votre programme en indiquant quelle(s) précaution(s) est/sont à prendre pour respecter sa sémantique

N.B : Pour obtenir l'identifiant du thread en cours d'exécution, il faut appeler la fonction :

```
id du thread = omp_get_thread_num( );
```

3.4 Exercice 3

On souhaite écrire un programme qui affiche le résultat du calcul suivant :

$$\sum_{i=0}^{100000} \sum_{j=0}^{100000} sum = sum + i + j$$

Travail :

— Q1 : écrivez une première version séquentielle de ce programme et vérifiez son bon fonctionnement
— Q2 : écrivez maintenant ce même programme en version parallèle sachant qu'une boucle for peut se paralléliser avec la directive *#pragma omp parallel for*. Pensez à prendre toutes les précautions nécessaires pour respecter sa sémantique (bonne utilisation des variables privées et partagées, et des clauses de réduction).
— Q3 : évaluez les performances des deux versions en utilisant les fonctions de mesure de temps qui vous sont fournies. Concluez sur les résultats obtenues. Sont ils en accords avec ce que vous attendiez ?
— Q4 : exécutez la commande *gnome-system-monitor* dans un terminal pour visualiser la charge des différents coeurs de calcul au cours de l'exécution de votre programme. Les graphiques obtenues correspondent t-ils à ce que vous attendiez ?

3.5 Exercice 4

On souhaite maintenant effectuer une approximation de la valeur de π en utilisant la formule de Leibniz suivante :

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ... = \frac{\pi}{4}$$

Travail :

— Reprenez les questions Q1, Q2, Q3 et Q4 de l'exercice précédent