Calcul parallèle : OpenMP vs. Cilk vs. Threading Building Blocks

Haifa Ben Hmida El Abri

Dorothée Huynh

Programmation Comparée

26 Février 2021

I. Introduction

Le parallélisme est une exécution simultanée de plusieurs calculs ou processus. Son but est de diminuer le temps de réalisation d'une grande tâche. La création des processeurs multi-coeurs au début des années 2000 et leur amélioration technique depuis lors a permis d'amener le parallélisme dans les ordinateurs de bureau et ainsi de démocratiser l'accès au parallélisme Les domaines d'utilisation sont variés : la dynamique des fluides, les prédictions météorologiques, le traitement d'information ou d'images, l'intelligence artificielle…

II. Modèles de parallélisation sur la mémoire partagée

Les modèles de parallélisation sur la mémoire partagée sont regroupés en trois catégories.

Les modèles basés sur les threads (ex. Pthread) fournissent des routines de bas niveau pour paralléliser une application, par exemple la gestion des threads et leur cycle de vie (création, terminaison et jointure d'un thread). Ils utilisent les verrous d'exclusion mutuelle et des variables conditionnelles pour établir des communications et une synchronisation entre les threads. Ces modèles sont adaptés aux applications utilisant de multiples données.

Les modèles basés sur les directives (ex. OpenMP) utilisent les directives haut niveau du compilateur pour paralléliser les applications. Ils sont une extension du modèle de parallélisme basé sur les threads. Ils prennent en charge les fonctionnalités bas niveau comme le partitionnement, la gestion des workers, la synchronisation et la communication entre les threads. Leur avantage est de grandement faciliter la programmation d'applications parallèles en épargnant aux programmeurs les casse-têtes liés à la parallélisation et la concurrence tels que les deadlocks, le false sharing et les data races.

Les modèles basés sur les tâches (ex. Cilk, TBB) sont basés sur le concept de spécifier des tâches au lieu de spécifier des threads. Les tâches ont une durée de vie plus courte que les threads et sont plus légères.

III. Bibliothèques de parallélisation

OpenMP

OpenMP est une bibliothèque qui prend en charge la programmation parallèle pour les langages C, C++ et Fortran. Sa spécification est décidée par le consortium OpenMP Architecture Review Board. Comme ce sont des spécifications, il n'y a pas de licence pour utiliser OpenMP. La 1ère version d'OpenMP pour le C/C++ paraît en octobre 1998 et cela fait d'elle la bibliothèque la plus ancienne des bibliothèques de parallélisation que nous comparons ici. Elle est multiplateforme (Mac OSX, Linux, Windows, ...) car elle est implémentée par des compilateurs (ex. Visual C++ pour Windows, gcc pour Linux, clang pour Max OSX) fonctionnant sur différentes plateformes. La prise en charge de OpenMP dépend donc du compilateur. Par exemple, Clang et Gcc prennent partiellement en charge la version 5 d'OpenMP sortie en 2018 et entièrement les versions antérieures tandis que Visual C++ a une prise en charge complète de OpenMP jusqu'à sa version 2 sortie en 2002.

OpenMP est non seulement une bibliothèque de fonctions (#include <omp.h>) (ex. omp_get/set_num/max_threads(), gestion verrous (init/set/unset/destroy)) et un ensemble de directives de compilation (commençant par #pragma omp), mais possède également des variables d'environnement spécifiques (commençant par OMP_, ex. OMP_NUM_THREADS à définir sinon la valeur indéterminée dépend du système et est souvent 1). De ce fait, il agit sur toutes les étapes post-écriture du code.

OpenMP cache à l'utilisateur et implémente les détails comme le partitionnement de la charge de travail, la gestion des workers, la communication et la synchronisation. L'utilisateur n'a plus qu'à spécifier les directives de parallélisation.

Threading Building Blocks

La bibliothèque Threading building blocks (TBB) est écrite en C++ et ne nécessite aucun compilateur spécial. En effet, elle fonctionne sur n'importe quel système ayant un compilateur C++ (compilateur Intel c++ ou gcc). Elle prend en charge la programmation parallèle, développée par Intel en 2006 (ce qui fait de TBB la bibliothèque la plus récente que nous comparons ici) et elle est distribuée sous une licence gratuite et une licence commerciale.

La bibliothèque TBB fonctionne sur différents systèmes comme Windows, OS X ou linux. Elle met la possibilité d'imbriquer des parties parallèles du code ce qui aide à augmenter les performances. Cependant, il y a certaines limitations aux traitements d'entrées - sorties et temps réel.

Pour utiliser TBB, il faut spécifier des tâches et pas des threads. Par conséquent, la bibliothèque mappe ces tâches sur des threads de manière efficace. Ce mappage sert à maximiser l'équilibrage de la charge autrement dit lier des threads particuliers à un matériel particulier sans garantir qu'aucun thread particulier ne soit lié à certains travaux pendant que les autres threads restent inactifs.

Cilk

Cilk est un langage de programmation parallèle multithread basé sur C++. Développé à l'origine dans les années 1990 au laboratoire (MIT), Cilk a ensuite été commercialisé sous le nom de Cilk ++ par une société dérivée, Cilk Arts. Cette société a ensuite été rachetée par Intel, qui a augmenté la compatibilité avec le code C et C ++ existant, appelant le résultat Cilk Plus.

Cilk aussi est basé sur les tâches comme tbb , adapté aux problèmes basés sur la stratégie de division.il offre des primitives pour l'expression explicite du parallélisme (créations de tâches) et les synchronisations. Il utilise l'ordonnancement par vol de travail (dès qu'un processeur est inactif, il demande du travail à un autre processeur) pour placer les tâches prêtes à être exécutées de manière dynamique sur des processeurs disponibles.

La description du parallélisme se fait à l'aide du mot clé **spawn** placé devant un appel de fonction. La sémantique de cet appel diffère de celle de l'appel classique d'une fonction : la procédure appelante peut continuer son exécution en parallèle de l'évaluation de la fonction appelée au lieu d'attendre son retour pour continuer. Cette exécution étant asynchrone, la procédure créatrice ne peut pas utiliser le résultat de la fonction appelée sans synchronisation. Cette synchronisation est explicite par utilisation de l'instruction **sync**. Le modèle de programmation de Cilk est de type série parallèle (la tâche mère s'exécute en concurrence avec ses filles, jusqu'à rencontrer l'instruction sync).

Comparaison TBB vs. Cilk

TBB et Cilk ont des fonctionnalités communes comme l'exécution parallèle qui est basée sur les tâches, les patterns parallèles complexes ainsi que l'équilibrage de la charge et les traitements d'entrées - sorties.
Mais ce qui différencie TBB à Cilk est :

- Le parallélisme des boucles imbriquées : TBB permet de paralléliser des boucles imbriquées à l'aide de la fonction parallel_for (fonction de parallélisation d'une boucle for) en lui passant comme premier paramètre blocked_range3d< (type des intervalles) > (où on définit les intervalles des boucles for imbriquées). Ce n'est pas le cas avec CILK où nous devons utiliser imbriquer des cilk_for.
- Les vecteurs et les mappages simultanés permettent une rédaction d'une structure de données sans verrouillage où les threads accèdent efficacement aux données partagées.
- TBB fournit la bibliothèque TBBmalloc qui permet l'allocation scalable de mémoire qui peut être utilisée seule ou conjointement avec la bibliothèque de fonctions générales TBB.

D'autre part, CILK et TBB appartiennent tous deux à Intel. Cependant, CILK est en état d'obsolescence. En effet, Intel a annoncé en septembre 2017 que Cilk Plus serait déprécié dans la sortie Intel Software Development Tools en 2018. Cilk n'est plus supporté par GCC depuis sa version 8.1 sortie en mai 2018.

IV. Benchmark

Nous comparons via l'exemple du produit matriciel le temps d'exécution de fonctions écrites de manière séquentielle, ou avec une parallélisation OpenMP ou Cilk ou TBB. L'idée est d'évaluer le temps d'exécution des fonctions de produit matriciel sur des paramètres (matrices à multiplier) communs.
Nous avons choisi plusieurs thématiques de test :

- Augmentation de la taille des deux matrices carrées

La forme carré permet d'enlever l'influence du nombre de lignes ou de colonnes.
Tailles des matrices (hauteur x largeur) : 100x100, 250x250, 500x500, 750x750, 1000x1000

- Différence de forme des deux matrices en gardant constant le nombre total d'élément

Tailles des matrices (hauteur x largeur) de 90 000 éléments : 100x900, 150x600, 200x450, 300x300, 450x200, 600x150, 900x100

- Augmentation du nombre de colonnes en gardant constant le nombre de lignes

Tailles des matrices (hauteur x largeur) de hauteur 100 : 100x1000, 100x2000, 100x3000, 100x4000, 100x5000, 100x6000, 100x7000, 100x8000, 100x9000, 100x10000

- Augmentation du nombre de lignes en gardant constant le nombre de colonnes

Tailles des matrices (hauteur x largeur) de largeur 100 : 1000x100, 2000x100, 3000x100, 4000x100, 5000x100, 6000x100, 7000x100

Les fonctions de produit matriciel ont une complexité temporelle cubique. Globalement, le temps d'exécution augmente quelque soit la fonction de produit matriciel quand le nombre d'éléments dans les matrices à multiplier augmente et la version séquentielle prend plus de temps que les versions parallèles. Pour les matrices carrées, l'augmentation du temps d'exécution est d'un facteur 9 par rapport à l'augmentation de la largeur et la hauteur des matrices et les versions parallèles sont deux

fois plus rapides que la version séquentielle. On remarque aussi que Cilk est plus rapide que TBB qui est plus rapide que OpenMP. Pour les matrices de 90 000 éléments, l'augmentation du temps d'exécution est linéaire avec l'augmentation du nombre de lignes, les versions parallèles sont 2,5 à 3 fois plus rapides que la version séquentielle et on a Cilk qui est plus rapide que OpenMP qui est plus rapide de 2% que TBB. Pour les matrices de 100 lignes, la version séquentielle est de 2,6 à 2,9 fois plus lente que les versions parallèles et on a Cilk plus rapide que TBB qui est 2% à 5% plus rapide que OpenMP et la différence entre TBB et OpenMP s'accroit quand le nombre de colonnes augmente. Pour toutes les versions, l'augmentation du temps d'exécution est 3 fois plus grande que l'augmentation du nombre de colonnes. Pour les matrices de 100 colonnes, la version séquentielle est 3 fois plus lente que les versions parallèles et on a Cilk plus rapide que Tbb qui est plus rapide que OpenMP. L'augmentation du temps d'exécution est 3 fois supérieure à l'augmentation du nombre de lignes.
En résumé, la version séquentielle est 3 fois plus lente que les versions parallèles et Cilk est plus rapide que TBB et OpenMP.

V. Conclusion

La parallélisation d'un calcul comporte bien des avantages, notamment une meilleure répartition de la charge de travail et une diminution du temps d'exécution du calcul. Cependant, il n'est pas forcément facile d'y parvenir. Il faut prendre en compte le niveau d'abstraction auquel on se place quand on implémente la parallélisation :

- Un bas niveau permet d'avoir plus de contrôle et d'optimisation manuelle mais on est confronté à gérer plus de détails soi-même.
- Un niveau plus haut d'abstraction facilite la programmation et démocratise l'implémentation de la parallélisation. Cependant, il peut être plus difficile de comprendre comment utiliser une bibliothèque de parallélisation ou de déboguer le programme.

Il peut être difficile de bien faire fonctionner la parallélisation afin d'obtenir un résultat de calcul correct et un temps d'exécution moindre qu'en séquentiel.

D'autre part, il faut aussi réfléchir à sa stratégie de parallélisation : bien cerner les endroits du code où il est bénéfique d'introduire de la parallélisation, sans tomber dans l'excès de mettre de la parallélisation partout car il ne faut pas perdre de vue que les threads et tâches parallèles ont un coût en terme de gestion et de ressources pour le système.
Un axe d'optimisation de programme parallèle serait d'utiliser le moins de threads possible à tout instant afin de ne pas trop alourdir le système avec la création, jointure et destruction des threads et/ou de minimiser le nombre de synchronisations entre threads (minimisation de l'attente entre threads et du temps mort) dans le cas d'une variable (ou plusieurs) partagée entre threads, tout en maximisant la diminution du temps d'exécution du calcul. On pourrait également songer à paramétrer le schedule des threads afin de laisser le moins de temps mort à un thread et lui donner d'autres instructions à exécuter, à condition que le résultat final soit correct. Cette réflexion est valable pour les tâches exécutées en parallèle (TBB, Cilk).

VI. Références

- Site officiel de OpenMP : https://www.openmp.org
- Documentation de TBB

https://software.intel.com/content/www/us/en/develop/documentation/tbb-documentation/top.html

- Comparatif de modèles de programmation parallèles dont Cilk, TBB et OpenMP

https://www.researchgate.net/publication/261424700_A_comparison_of_five_parallel_programming_models_for_C/link/5607df6808ae5e8e3f3a3a1d/download

- Dépréciation de Cilk

https://community.intel.com/t5/Software-Archive/Intel-Cilk-Plus-is-being-deprecated/td-p/1127776

VII. Annexe

Code de produit matriciel

Version séquentielle

```
matrix_t multiplySEQ(matrix_t matrixOne, matrix_t matrixTwo)
{
   int width1 = matrixOne[0].size();
   if (width1 != matrixTwo.size()) {  return matrix_t(0);   }
   int height = matrixOne.size();
   int width2 = matrixTwo[0].size();
   matrix_t result(height, std::vector<int>(width2, 0));

   for (int i = 0; i < height; i++)
   {
      for (int j = 0; j < width2; j++)
      {
         for (int k = 0; k < width1; k++) {
            result[i][j] += matrixOne[i][k] * matrixTwo[k][j];
         }
      }
   }
   return result;
}
```

La zone parallélisable est en ocre.

Version parallèle OpenMP

```
matrix_t multiplyOMP(matrix_t matrixOne, matrix_t matrixTwo)
{
   int width1 = matrixOne[0].size();
   if (width1 != matrixTwo.size())
   {
      return matrix_t(0);
   }
   int height = matrixOne.size();
   int width2 = matrixTwo[0].size();
   matrix_t result(height, std::vector<int>(width2, 0));
   #pragma omp parallel for shared(result)
   for (int i = 0; i < height; i++)
   {
      for (int j = 0; j < width2; j++)
```

```
    {
        int sum = 0;
        for (int k = 0; k < width1; k++)
        {
            sum += matrixOne[i][k] * matrixTwo[k][j];
        }
        result[i][j] += sum;
    }
  }
  return result;
}
```

La parallélisation avec OpenMP se fait avec une ligne (en ocre) indiquant une section parallèle où chaque thread exécute des itérations de la boucle for qui itère sur la hauteur de la matrice résultat. En vert, il s'agit d'une optimisation qui consiste à stocker dans une variable temporaire la somme accumulée destinée à être placée dans une case de la matrice résultat. Le fait d'attribuer à une case de la matrice résultat la valeur de la somme au lieu d'augmenter la valeur de la case à chaque ajout d'un terme de la somme permet de ne modifier qu'une seule fois chaque case de la matrice résultat et ainsi de limiter le nombre de synchronisations entre les threads afin que chaque thread accède à une variable résultat à jour, limitant ainsi le temps d'attente des threads en vue d'une synchronisation.

Version parallèle TBB

```
matrix_t multiplyTBB(matrix_t matrixOne, matrix_t matrixTwo)
{
   int width1 = matrixOne[0].size();
   if (width1 != matrixTwo.size())
   {
      return matrix_t(0);
   }

   int height = matrixOne.size();
   int width2 = matrixTwo[0].size();
   matrix_t result(height, std::vector<int>(width2, 0));
   size_t grain_size = height / 10;
   tbb::parallel_for(
   tbb::blocked_range<int>(0,height,grain_size), [&](const     tbb::blocked_range<int>&
range)
   {
            for(int i = range.begin(); i< range.end(); ++i)
            {
                  for (int j = 0; j < width2; j++)
                  {
                     int sum = 0;

                     for (int k = 0; k < width1; k++)
                     {
                        sum += matrixOne[i][k] * matrixTwo[k][j];
                     }
```

```
                    result[i][j] += sum;
                }
            }
    });

    return result;

}
```

La parallélisation de la boucle for avec TBB nécessite la fonction parallel_for avec comme arguments l'intervalle bloqué (minimum et maximum de l'intervalle et taille du grain (ici le nombre d'itérations) à attribuer à un thread) et une fonction contenant la boucle for à paralléliser.

Version parallèle Cilk

```
matrix_t multiplyCILK(matrix_t matrixOne, matrix_t matrixTwo)
{
    int width1 = matrixOne[0].size();
    if (width1 != matrixTwo.size())
    {
        return matrix_t(0);
    }
    int height = matrixOne.size();
    int width2 = matrixTwo[0].size();

    matrix_t result(height, std::vector<int>(width2, 0));

    cilk_for (int i = 0; i < height; i++)
    {
        for (int j = 0; j < width2; j++)
        {
            int sum = 0;
            for (int k = 0; k < width1; k++)
            {
                sum += matrixOne[i][k] * matrixTwo[k][j];
            }
            result[i][j] += sum;
        }
    }
    return result;
}
```

La parallélisation de la boucle for avec Cilk utilise le mot-clé cilk_for.