Travaux dirigés n°4

Xavier JUVIGNY

November 26, 2021

Contents

1 Présentation de TBB

1.1 Introduction

La bibliothèque TBB (Threading Building Block library) est une biblithèque C++ qui remplit deux rôles :

1. Remplit des fonctionnalités manquantes au support pour le parallélisme dans le standard C++, là où le langage n'a pas suffisamment évolué ou bien où les nouvelles caractéristiques du C++ ne sont pas encore bien supportées par les compilateurs;
2. Propose une interface abstraite de haut niveau pour le parallélisme qui est au delà de ce que le standard C++ pourra proposer.

TBB contient un certain nombre de caractéristiques qu'on peut catégoriser en deux grands groupes :

- Des interfaces pour exprimer des algorithmes parallèles;
- Des interfaces qui sont indépendant du modèle d'exécution

Dans la première catégorie, on trouve :

- Des algorithmes parallèles génériques tels que la parallélisation de boucle, les réductions, etc.
- Des algorithmes de parallélisation par tâche (on définit un graphe de tâches qui seront exécutés en parallèle si possible par TBB)
- Une version parallèle de la STL comme proposé par la norme C++17/C++20

Dans la seconde catégorie, on trouve :

- Des conteneurs supportant la concurrence en multithreading (Tables de hashage, queues, tableaux dynamiques)
- Des allocateurs parallélisables ou bien qui peuvent s'aligner avec les lignes de cache,etc.
- Des primitives de synchronisation, etc.

Nous nous intéresserons ici surtout à l'utilisation des algorithmes parallèles génériques, et en particulier à la parallélisation de boucle

1.2 Installation de TBB

- Sous Windows (MSys 2) : `pacman -S mingw64mingw-w64-x86_64-intel-tbb`
- Sous Linux Ubuntu (WSL ou Linux natif) : `sudo apt install libtbb-dev`
- Sous Mac-OS X (avec homebrew) : `brew install tbb`

Allez dans le répertoire `TBBExamples` et essayer de compiler les exemples (en utilisant `make`).

1.3 Parallélisme de boucle

La parallélisation d'une boucle se fait à l'aide de la fonction `tbb::parallel_for` qui se décline en plusieurs versions :

- Une version simple : `tbb::parallel_for(start,end, step, lambda_function)`. Par exemple :

```
void print_hello( int i ) { std::cout << i << " : Hello world !" << std::endl; }

int main()
{   ...
    tbb::parallel_for(0, 100, 2, [](int i) { print_hello(i); });
```

- Une version utilisant des blocs en 1D, 2D ou 3D(voir plus loin)
- On peut également rajouter un paritioneur qui définira la stratégie de distribution des blocs et de leur répartion. Par défaut, le découpage essaye d'être "intelligent" et la distribution des blocs est dynamique.

1.3.1 Les blocs d'indices

TBB propose des classes permettant de répartir les indices dans plusieurs tâches. Il existe de base trois classes permettant de répartir en 1D, 2D ou 3D des données (la dimension correspond au nombre de boucles imbriquées). Il existe également une classe générique permettant de gérer des dimensions supérieures.

Pour chaque indice géré par la classe, il faudra donner à sa construction les indices de début et de fin de la boucle correspondante et, en option, une taille de grain. L'utilisation de cette taille de grain diffère selon le partitioneur employé (on verra cela dans la section suivante).

La syntaxe pour les trois classes de base (templatées) est la suivante :

- En une dimension :

```
tbb::blocked_range<type indice>(start, end,grainSize=1)
```

Exemple d'utilisation (voir `compute_pi.cpp` pour une utilisation concrète) :

```
tbb::blocked_range<int> r(0, 100, 2);
for (int i = r.begin(); i != r.end(); ++i )
{
    ...
}
```

- En deux dimensions :

```
tbb::blocked_range2d<type indice1, type indice2>(start i1, end i1, grainSize1,
                                                 start i2, end i2, grainSize2);
```

On accède aux premiers indices à l'aide de la méthode `rows` et aux seconds indices à l'aide de la méthode `cols`.
Exemple d'utilisation (voir `Mandelbrot_tbb.cpp` pour un exemple concrèt) :

```
auto r = tbb::blocked_range2d<int,unsigned>(-10,10,1,0, 40, 2);
for (auto i = r.rows().begin(); i != r.rows().end(); ++i)
  for (auto j = r.cols().begin(); j != r.cols().end(); ++j)
  {
    ....
  }
```

- En trois dimensions :

```
tbb::blocked_range3d<type indice1, type indice2, type indice3>(start i1, end i1, grainSize1,
                                                               start i2, end i2, grainSize2,
                                                               start i3, end i3, grainSize3);
```

On accède aux premiers indices à l'aide de la méthode `pages`, aux seconds indices à l'aide de la méthode `rows` et aux troisièmes indices à l'aide de la méthode `cols`. Exemple d'utilisation :

```
auto r = tbb::blocked_range3d<int,int,std::size_t>(-10,10,1,-10,10,1,0,40,2);
for (auto i = r.pages().begin(); i != r.pages().end(); ++i )
  for (auto j = r.rows().begin(); j != r.rows().end(); ++j )
    for (auto k = r.cols().begin(); k != r.cols().end(); ++k )
    {
      ...
    }
```

1.3.2 Les partitioneurs

Un partitioneur permet de spécifier comment un algorithme devra partionner sa boucle. La table suivante résume les caractéristiques des différents partitioneurs (le grain correspond au nombre d'itération que doit faire une boucle après avoir effectuer une partition de la boucle originale):

Partitioneur	**Description**	**Avec des blocs d'indices**
`simple_partitioner`	Taille des blocs limitée par le grain	$\frac{g}{2} \leq$ taille $\leq g$
`auto_partitioner`	Calcul automatiquement la taille du grain	$\frac{g}{2} \leq$ taille
`static_partitioner`	Détermine une taille de boucle fixe et une distribution uniforme des itérations sans faire d'équilibrage des charges. La distribution uniforme est créé sauf si la taille du grain empêche de créer P parties	$\max(\frac{g}{3}, \frac{N}{P}) \leq$ taille où • N est la taille du problème • P est le nombre de threads