École Nationale Supérieure
d'Ingénieurs de Caen

6, Boulevard Maréchal Juin
F-14050 Caen Cedex, France

Multi-threading en Threading Building Blocks (C++)

Niveau	3ème **année**
Parcours	**Électronique et Physique Appliquée**
Unité d'enseignement	2E1AC2 **- Architectures pour le calcul** [**Parallélisme**]
Création	9 **Novembre** 2020
Responsables	**Emmanuel Cagniot** **emmanuel.cagniot@ensicaen.fr** **Hugo Descoubes** **Hugo.Descoubes@ensicaen.fr**

Table des matières

1 Multi-threading en Threading Building Blocks (C++)

Threading Building Blocks (TBB) est une bibliothèque *open source* C++ initiée par Intel en 2006 pour simplifier le développement d'applications multi-threadées sur architectures multicoeurs. Dans cette dernière, le programmeur définit son application en termes de tâches à accomplir ainsi que leur ordonnancement les unes par rapport aux autres.

L'attribution des tâches aux différents threads disponibles sur la plateforme est réalisée par la bibliothèque elle-même, et plus précisément son *scheduler* qui implémente le modèle *work-stealing* emprunté au langage Cilk (extension du langage C permettant la programmation multi-thread).

Dans ce dernier, une file de tâches à accomplir est initialement constituée sur chaque cœur, le nombre de tâches de ces files étant quasiment identique d'un cœur à l'autre. Lorsqu'un cœur a épuisé sa file, le *scheduler* la regarnit avec des tâches initialement attribuées à d'autres cœurs et qui n'ont pas encore été réalisées. Par conséquent, ce type de *scheduling* garantit un équilibrage de charge correct entre les différents cœurs de l'architecture sans que le programmeur n'ait à s'en préoccuper.

TBB utilise abondamment la notion de *template programming* afin de tirer avantage du polymorphisme de compilation (*low-overhead polymorphism*) ; elle propose :

1. des algorithmes et des partitionneurs d'espaces d'itération permettant de sélectionner le bon grain de parallélisme pour certains de ces algorithmes ;
2. la notion de pipeline logiciel ;
3. la notion de graphe de flot de contrôle ;
4. la possibilité de contrôler finement le *scheduler* ;
5. les mécanismes de synchronisation classiques ;
6. des conteneurs *thread-safe* et des allocateurs de mémoire en contexte multithreadé.

1.1 Algorithmes et partitionneurs d'espace d'itération

Les algorithmes constituent un moyen simple d'aborder la programmation en TBB : il s'agit de repérer une zone de code parallélisable (par exemple une boucle de type *forall*) et de la remplacer par une instanciation de la fonction générique correspondante.

TBB propose les algorithmes suivants :
— `parallel_for` qui répartit les itérations d'une boucle de type `for` ;
— `parallel_do` qui tente de prendre en charge les autres types de boucles ;
— `parallel_for_each`, une version parallèle de l'algorithme `for_each` de la bibliothèque standard ;
— `parallel_reduce` qui implémente une réduction ;
— `parallel_scan` qui calcule les préfixes d'une opération binaire donnée ;
— `parallel_sort`, une version itérative de l'algorithme du *quicksort* ;
— `parallel_invoke` qui invoque plusieurs méthodes et/ou fonctions en parallèle.

Les algorithmes `parallel_for`, `parallel_reduce` et `parallel_scan` sont couplés à un partitionneur d'espace d'itération. Cet objet permet de sélectionner le grain de parallélisme, c'est à dire le nombre d'instructions moyen à faire exécuter par chaque thread.

Le partitionneur doit tenter de concilier deux objectifs contradictoires. D'une part, il doit maximiser le nombre de threads utilisés. D'autre part, il doit minimiser l'*overhead* lié au *scheduler* en maximisant le volume d'instructions moyen à faire traiter par chaque thread, c'est à dire, au final, minimiser le nombre de ces derniers.

TBB propose trois types de partitionneurs.

`simple_partitioner :` la classe `blocked_range` possède un attribut `my_grainsize` qui représente le nombre minimum d'itérations consécutives à faire traiter par un thread. Cet argument est, par défaut, initialisé à la valeur 1. La classe `blocked_range` propose également une méthode `bool is_divisible() const` qui indique si l'espace d'itération correspondant peut être subdivisé en deux sous-espaces dont les tailles sont sensiblement identiques. Un espace est considéré comme divisible si le nombre de ses itérations est supérieur à la valeur de `my_grainsize`. Le rôle d'un `simple_partitioner` est de subdiviser l'espace d'itérations afin que le nombre d'itérations consécutives de chaque sous-espace soit, au plus, égal à la valeur de l'attribut `my_grainsize`.

`auto_partitioner :` la valeur de l'attribut `my_grainsize` est déterminée de manière automatique. Plus précisément, l'espace d'itération affecté à un thread est subdivisé et sa moitié affectée à un autre thread uniquement si ce dernier devient disponible (inactif). Ce type de partitionnement, qui permet d'équilibrer au mieux la charge de travail, est le partitionnement par défaut des algorithmes `parallel_for` et `parallel_reduce`.

`affinity_partitioner :` un `auto_partitioner` qui, de plus, garantit qu'un sous-espace d'itération d'une boucle sera toujours attribué au même thread. Ce type de partitionneur exploite la propriété de localité des données dans le cache privé de niveau 1 d'un cœur.

Les figures 1 et 2 illustrent l'utilisation des algorithmes `parallel_for` et `parallel_reduce` couplés à un `affinity_partioner`.

Dans cet exemple, une boucle parallèle sur une instance `vecteur` de classe `vector` est encagée dans une boucle séquentielle. Par conséquent, il est judicieux de demander à ce que chaque sous-espace d'itération de la boucle parallèle soit toujours traité par le même thread au sein de la boucle séquentielle. C'est ce que nous faisons en couplant l'algorithme `parallel_for` à une instance `ap` de classe `affinity_partioner`.

Le travail à réaliser sur chaque sous-espace d'itération est, quant à lui, implémenté sous forme d'une *lambda* capturant l'indice de boucle `iter` et le tableau `vecteur` par référence. Cette *lambda* représente un objet fonction, c'est à dire une classe surchargeant l'opérateur fonction (`operator()`).

L'argument fourni à cet opérateur est un intervalle de classe `blocked_range`. Cette dernière est une classe générique paramétrée par le type de l'indice de boucle et dont les deux arguments fournis à son constructeur logique sont respectivement la plus petite (inclue) et la plus grande (exclue) valeur de cet indice.

Une première boucle de réduction permettant de calculer la somme des éléments du tableau `vecteur` est parallélisée via l'algorithme `parallel_reduce` toujours couplé au partitionneur `ap`.

Contrairement à l'algorithme `parallel_for`, nous choisissons de représenter le travail à réaliser via une classe et non pas des *lambdas*.

Dans ce cas, TBB impose de définir une surcharge de l'opérateur fonction mais également une méthode `join` et un constructeur par scission (*split constructor*). Nous définissons donc une classe `PartialReduce` telle que :

1. ses attributs sont une référence constante `_vecteur` vers le vecteur partagé par les threads et une somme partielle `_partialSum` ;
2. un constructeur logique permettant de se brancher sur le vecteur à partager et d'initialiser la somme partielle à zéro ;
3. un constructeur par scission. Son paramètre `split` n'a d'autre intérêt que de distinguer ce constructeur particulier du constructeur par recopie. Le constructeur par scission est utilisé par le *scheduler*

de TBB lorsqu'il repère un thread inactif. Dans ce cas, il scinde le plus grand intervalle d'itération affecté à un thread actif avant d'en confier la seconde partie au thread inactif ;

4. un accesseur en lecture retournant la valeur de la somme partielle ;
5. une surcharge de l'opérateur fonction qui calcule la somme partielle des éléments du tableau correspondant au sous-espace d'itération ;
6. une méthode `join` permettant de cumuler les résultats locaux ;
7. une neutralisation du constructeur par recopie. Notons que l'opérateur d'affectation est implicitement neutralisé du fait de l'utilisation d'un attribut de type référence.

Une seconde boucle de réduction présente la manière d'utiliser l'algorithme `parallel_reduce` avec des *lambdas* pour également calculer la somme des éléments du tableau `vecteur`.

TBB propose également les classes `blocked_range2d` et `blocked_range3d` représentant respectivement des espaces d'itération bi et tridimensionnels. Dans ces classes, chaque dimension est représentée par une instance de classe `blocked_range`. Par conséquent, des espaces d'itération multi-dimensionnels peuvent être partitionnés selon chacun de leurs dimensions.

1.1.1 `parallel_for_each`

Le prototype de cet algorithme est défini dans le module `tbb/parallel_for_each.h` :

```
1 template<typename InputIterator, typename Func>
2 void parallel_for_each(InputIterator first, InputIterator last, Func f);
```

Les paramètres `first` et `last` sont instanciés à partir des itérateurs proposés par un conteneur séquentiel. La fonction unaire `f` est appliquée simultanément à chaque élément de l'intervalle `[first, last[` si les itérateurs sont de type tableau (`RandomAccessIterator`) et séquentiellement dans le cas contraire.

1.1.2 `parallel_for`

Les différentes surcharges de cet algorithme sont définies dans le module `tbb/parallel_for.h` :

```
1 template<typename Index, typename Function>
2 Function parallel_for(Index first, Index_type last, Function f);
3
4 template<typename Index, typename Function>
5 Function parallel_for(Index first, Index_type last, Index step, Function f);
6
7 template<typename Range, typename Body>
8 void parallel_for(const Range& range, const Body& body, [, partitioner]);
```

La seconde surcharge implémente `for (auto i = first; i < last; i += step) { f(i); }`, l'argument `step` valant implicitement 1 dans la première. Ses paramètres `first`, `last` et `step` sont obligatoirement d'un type entier (ou convertible implcitement en entier) et le pas `step` doit en outre être positif. La fonction `f` ne doit pas provoquer d'effet de bord et la boucle ciblée doit être de type `forall`.

Les autres surcharges sont beaucoup plus générales. Le paramètre `Body` doit être instancié par une classe proposant un constructeur par recopie, un destructeur, ainsi qu'une surcharge de l'opérateur fonction telle que `void operator()(const Range& range) const`. Le constructeur par recopie et le destructeur pouvant être fournis par défaut, le paramètre `Body` peut également est instancié par une *lambda* (exemple de la figure 1).

FIGURE 1 – Algorithmes `parallel_for` et `parallel_reduce` couplés à un `affinity_partioner` (première partie).

22/04/2012 parforeduce.cpp 3

```
  // Initialisation du scheduler de TBB avec le nombre de threads par defaut,
  // c'est a dire le nombre de coeurs.
  tbb::task_scheduler_init init;

  // Vecteur de valeurs entieres.
  std::vector< unsigned long > vecteur(1024, 0);

  // Renommage de type.
  typedef PartialReduce::size_type size_type;

  // Affinity partitionner pour gerer efficacement une boucle parallele a
  // l'interieur d'une boucle sequentielle.
  tbb::affinity_partitioner ap;

  // Boucle sequentielle exterieure.
  for (size_type iter = 0; iter < vecteur.size(); iter ++) {

    // Boucle parallele interieure.
    tbb::parallel_for(tbb::blocked_range< size_type >(0, vecteur.size()),
                      [&vecteur, &iter](const tbb::blocked_range< size_type >& r) {
                        for (size_type i = r.begin(); i != r.end(); i ++) {
                          vecteur[i] += iter;
                        }
                      },
                      ap);
  }

  // Somme des elements du vecteur.
  unsigned long somme;

  // Boucle de reduction (somme) parallele avec utilisation d'une classe.
  {
    PartialReduce pr(vecteur);
    tbb::parallel_reduce(tbb::blocked_range< size_type >(0, vecteur.size()),
                         pr,
                         ap);
    somme = pr.getPartialSum();
  }

  // Affichage du resultat.
  std::cout << somme << std::endl;

  // Boucle de reduction (somme) parallele avec utilisation de lambdas.
  somme =
    tbb::parallel_reduce(tbb::blocked_range< size_type >(0, vecteur.size()),
                         0,
                         [&vecteur](const tbb::blocked_range< size_type >& r,
                                    const unsigned long& init) -> unsigned long {
                           return std::accumulate(&vecteur[r.begin()],
                                                  &vecteur[r.end()],
                                                  init);
                         },
                         std::plus< unsigned long >(),
                         ap);

  // Affichage du resultat.
  std::cout << somme << std::endl;
```

22/04/2012 parforeduce.cpp 4

```
  // Tout s'est bien passe.
  return EXIT_SUCCESS;

}
```

FIGURE 2 – Algorithmes `parallel_for` et `parallel_reduce` couplés à un `affinity_partioner` (dernière partie).

1.1.3 `parallel_do`

Le prototype de cet algorithme est défini dans le module `tbb/parallel_do.h` :

```
template<typename InputIterator, typename Body>
void parallel_do(InputIterator first, InputIterator last, Body body);
```

Cet algorithme est similaire à `parallel_for_each` puisque l'objet fonction `body` est appliqué en parallèle ou en séquentiel sur les éléments de l'intervalle `[first, last[` selon que les itérateurs fournis en arguments soient ou non de type tableau.

Les éléments du conteneur auquel sont rattachés les itérateurs `first` et `last` peuvent être d'un type fondamental ou classe. Dans ce dernier cas, la classe doit proposer un constructeur par recopie et un destructeur, les deux pouvant être fournis par défaut.

La classe `Body`, quant à elle, doit proposer une surcharge de l'opérateur fonction. La première forme de cette surcharge est `void operator()(`*`cv-qualifiers`* `T& item) const` dans laquelle le paramètre `item` est une reférence variable ou constant pour l'élément du conteneur courant. Utiliser cette surcharge revient à définir une boucle `for` généralisée de la forme `for(Item item& : conteneur) {... }`.

La classe `Body` peut également proposer la surcharge `void operator()(`*`cv-qualifiers`* `T& item, parallel_do_feeder< T >& feeder) const`. Pour illustrer sa finalité, considérons l'exemple suivant dans lequel tous les nœuds d'un arbre binaire doivent faire l'objet d'un même traitement indépendant. Nous pouvons effectuer ce traitement en parcourant l'arbre en largeur d'abord, c'est à dire traiter un nœud tout en ajoutant ses fils dans une liste de nœuds à traiter.

```
struct Noeud {
  int donnee;
  Noeud* gauche;
  Noeud* droite;
};

parallel_do(&arbre,
            &arbre + 1,
             [](Noeud& noeud, parallel_do_feeder< Noeud >& feeder) {
               if (noeud.gauche != nullptr) {
                 feeder.add(*noeud.gauche);
               }
               if (noeud.droite != nullptr) {
                 feeder.add(*noeud.droite);
               }
               noeud.donnee ++;
             });
```

Dans un premier temps, une liste de tâches à accomplir est constituée à partir des éléments de l'intervalle `[first, last[`. Ces derniers sont susceptibles de générer de nouveaux éléments à traiter : par conséquent, ceux-ci seront stockés dans le paramètre `feeder` dont la classe générique (et abstraite) `parallel_do_feeder` propose une unique méthode `void add(const T& item)`.

Lorsque des nouveaux éléments sont ajoutés, l'algorithme peut tirer pleinement partie des threads disponibles, chose qui n'était pas possible si les itérateurs `first` et `last` n'étaient pas de type tableau. C'est pourquoi la documentation de TBB mentionne que cet algorithme peut être utilisé avec des listes pourvu que le volume d'instructions traité par l'opérateur fonction soit d'environ 10000.

1.1.4 `parallel_reduce`

Les différentes surcharges de cet algorithme sont définies dans le module `tbb/parallel_reduce.h` :

```
template<typename Range, typename Value, typename Func, typename Reduction>
Value parallel_reduce(const Range& range, const Value& identity,
                      const Func& func, const Reduction& reduction,
                      [, partitioner]);

template<typename Range, typename Body>
void parallel_reduce(const Range& range, Body& body [, partitioner]);
```

La première surcharge est dédiée aux objets fonctions et aux *lambdas*.

Le paramètre `identity` représente l'élément neutre pour l'opération binaire `reduction`.

Le paramètre `func` doit être instancié par une classe ou une *lambda* proposant une surcharge de l'opérateur fonction telle que `Value operator()(const Range& range, const Value& x)`. Cette surcharge effectue une réduction sur son sous-espace d'itération en prenant `x` comme valeur initiale.

Enfin, le paramètre `reduction` doit être instancié par une classe ou un *lambda* proposant une surcharge de l'opérateur fonction telle que `Value operator()(const Value& x, const Value& y) const`.

Dans l'exemple de la figure 1, nous utilisons directement la classe générique `plus` de la bibliothèque standard instanciée par le type entier `unsigned long`.

La seconde surcharge est dédiée aux classes.

Le paramètre `Body` doit être instancié par une classe proposant (classe `PartialReduce`, Figure 1) :

1. un constructeur par scission ;
2. un destructeur (pouvant être fourni par défaut) ;
3. une surcharge de l'opérateur fonction telle que `void operator()(const Range& range)` qui réalise l'accumulation sur le sous-espace d'itération `range` ;
4. une méthode `void join(const Body& rhs)` permettant de fusionner deux résultats locaux.

1.1.5 `parallel_sort`

Les surcharges de cet algorithmes sont définies dans le module `tbb/parallel_sort.h` :

```
template<typename RandomAccessIterator>
void parallel_sort(RandomAccessIterator begin, RandomAccessIterator end);

template<typename RandomAccessIterator, typename Compare>
void parallel_sort(RandomAccessIterator begin, RandomAccessIterator end,
                   const Compare& comp);
```

Cet algorithme, applicable uniquement aux conteneurs de type tableau (accès en temps constant aux éléments), implémente une version dé-récursivée de l'algorithme du *quick-sort*.

La documentation de TBB mentionne que l'implémentation proposée est instable dans le sens que deux éléments e_1 et e_2 de même valeur (clé) peuvent être classés comme e_1 suivi de e_2 dans la version séquentielle et comme e_2 suivi de e_1 dans la version parallèle. La première surcharge utilise par défaut le comparateur `less` de la bibliothèque standard.

1.1.6 `parallel_scan`

Considérons la suite de valeurs $\{x_1, x_2, \cdots, x_n\}$ et une opération binaire associative notée $\oplus$. Les préfixes associés à cette opération constituent la suite de valeurs $\{y_1, y_2, \cdots, y_n\}$ telle que :

$$y_1 = x_1, \tag{1.1}$$
$$y_k = x_1 \oplus x_2 \oplus \cdots \oplus x_k, \quad k > 1. \tag{1.2}$$
$$\tag{1.3}$$

Ces préfixes peuvent être calculés en parallèle et l'algorithme correspondant porte le nom de parallel prefix ou parallel scan. Les surcharges de cet algorithme sont définies dans le module `tbb/parallel_scan` :

```
1  template<typename Range, typename Body>
2  void parallel_scan(const Range& range, Body& body);
3
4  template<typename Range, typename Body>
5  void parallel_scan(const Range& range, Body& body,
6                     const auto_partitioner& partioner);
7
8  template<typename Range, typename Body>
9  void parallel_scan(const Range& range, Body& body,
10                    const simple_partitioner& partitioner);
```

Les figures 3 et 4 présentent le calcul des sommes préfixes d'un vecteur. Afin d'expliciter cet exemple, la documentation de TBB définit la notion de résumé (*summary*) comme l'ensemble des informations nécessaires tel que pour deux sous-intervalles d'indiçage consécutifs r et s :

1. si r ne possède pas de prédécesseur alors le résultat du scan pour s peut être calculé à partir de s et du résumé de r ;
2. un résumé pour la concaténation de r suivi de s est obtenu par la concaténation du résumé de r suivi de celui de s.

Dans le cas de sommes préfixes, le résumé du sous-intervalle r représente tout simplement la somme des éléments du vecteur dont le rang appartient à r.

L'implémentation parallèle du *scan* comporte deux phases.

Dans un premier temps, l'intervalle d'indicage du vecteur dont il faut calculer les sommes préfixes (notre exemple), est subdivisé en sous-intervalles et une somme préfixe locale est calculée pour chaque sous-intervalle. Cette phase descente est appelée « *pré-scan* ».

Dans un second temps, les sommes préfixes locales sont fusionnées pour produire le vecteur contenant les sommes préfixes. Cette phase de remontée est appelée « *post-scan* ».

La classe `Body` doit proposer les méthodes suivantes :

1. `void operator()(const Range& r, pre_scan_tag)` qui représente l'opérateur de *pré-scan*. Son rôle consiste à calculer le résumé associé au sous-intervalle d'indiçage r. Le second paramètre muet permet de distinguer l'opérateur de *pré-scan* de l'opérateur de *post-scan* ;
2. `void operator()(const Range& r, final_scan_tag)` qui calcule le *scan* (met à jour le vecteur de préfixes) et le résumé du sous-intervalle d'indiçage r. Le second paramètre muet permet de distinguer l'opérateur de *post-scan* de l'opérateur de *pré-scan* ;
3. `Body(const Body& autre, split)`, le constructeur par scission qui scinde `autre` de telle manière que `this` et `autre` puissent calculer indépendamment un résumé pour leurs sous-intervalles respectifs ;

```
28/04/2012                    parallel_scan.cpp                    1

/**
 * @mainpage
 *
 * Algorithme parallel_scan en TBB.
 *
 * @author Emmanuel CAGNIOT - Emmanuel.Cagniot@ensicaen.fr
 * @date 28.4.2012
 */

#include <tbb/task_scheduler_init.h>
#include <tbb/parallel_scan.h>
#include <tbb/blocked_range.h>
#include <vector>
#include <iostream>
#include <cstdlib>

/**
 * @class BodyPlus parallel_scan.cpp
 * @brief Classe permettant de calculer une somme prefixe.
 * @author Emmanuel.Cagniot - @c Emmanuel.Cagniot@ensicaen.fr
 * @date 28.4.2012
 *
 * Definition inline de la classe generique BodyPlus utilisee par l'algorithme
 * parallel_scan pour calculer des sommes prefixes.
 *
 * @note les instances de cette classe ne peuvent pas etre dupliquees.
 */
template< typename Type, typename RankType >
class BodyPlus {
public:
  /**
   * Constructeur logique.
   *
   * @param[in] entree - la valeur de @ref _entree.
   * @param[in] sortie - la valeur de @ref _ptrSortie.
   */
  BodyPlus(const std::vector< Type >& entree, std::vector< Type >& sortie):
    _entree(entree),
    _ptrSortie(&sortie),
    _somme(0) {
  }

  /**
   * Constructeur par scission.
   *
   * @param[in] autre - l'instance a scinder.
   * @param[in] split - un argument fantoche permettant de distinguer ce
   *   constructeur du constructeur par recopie.
   */
  BodyPlus(const BodyPlus& autre, tbb::split):
    _entree(autre._entree),
    _ptrSortie(autre._ptrSortie),
    _somme(0) {
  }

public:
  const Type& lireSomme() const {
    return _somme;
  }
```

```
28/04/2012                    parallel_scan.cpp                    2

  }

public:

  /**
   * Operateur de pre-scan permettant de calculer la sommes des elements de
   * chaque sous-intervalle.
   *
   * @param[in]  r - le sous-espace d'iteration.
   * @param[in] pre_scan_tag - un argument fantoche permettant de distinguer cet
   *   operateur de l'operateur de post-scan.
   */
  void operator()(const tbb::blocked_range< RankType >& r,
                  tbb::pre_scan_tag) {
    Type acc = _somme;
    for (RankType i = r.begin(); i != r.end(); i ++) {
      acc += _entree[i];
    }
    _somme = acc;
  }

  /**
   * Operateur de post-scan permettant de calculer les prefixes d'un
   * sous-intervalle.
   *
   * @param[in]  r - le sous-espace d'iteration.
   * @param[in] final_scan_tag - un argument fantoche permettant de distinguer
   *   cet operateur de l'operateur de pre-scan.
   */
  void operator()(const tbb::blocked_range< RankType >& r,
                  tbb::final_scan_tag) {
    Type acc = _somme;
    for (RankType i = r.begin(); i != r.end(); i ++) {
      acc += _entree[i];
      _ptrSortie->at(i) = acc;
    }
    _somme = acc;
  }

  /**
   * Reduit au niveau k deux sous-resultats du niveau k - 1.
   *
   * @param[in] autre - l'instance avec laquelle fusionner le resulat local.
   */
  void reverse_join(const BodyPlus& autre) {
    _somme += autre._somme;
  }

  /**
   * Affecte au niveau k-1 le resultat de la reduction de deux sous-resultat
   * locaux du niveau k - 1.
   *
   * @param[in] autre - l'instance dont il faut recopier le resultat.
   */
  void assign(const BodyPlus& autre) {
    _somme = autre._somme;
  }

private:
```

FIGURE 3 – Calcul de sommes préfixes avec l'algorithme **parallel_scan** (première partie).

28/04/2012 parallel_scan.cpp 3

```
  /**
   * Constructeur par recopie.
   *
   * @param[in] autre - l'instance a recopier.
   */
  BodyPlus(const BodyPlus& autre) = delete;

protected:
  /**
   * Vecteur dont il faut calculer la somme prefixe.
   */
  const std::vector< Type >& _entree;
  /**
   * Vecteur de prefixes.
   */
  std::vector< Type >* const _ptrSortie;

  /**
   * Somme locale.
   */
  Type _somme;
};
/**
 * Programme principal.
 */
int
main() {
  // Initialisation du scheduler de TBB avec le nombre de threads par defaut,
  // c'est a dire le nombre de coeurs.
  tbb::task_scheduler_init init;

  // Renommage de type.
  typedef std::vector< int >::size_type size_type;

  // Vecteur d'entiers dont il faut calculer les sommes prefixes.
  std::vector< int > entree(4);
  for (size_type i = 0; i < entree.size(); i ++) {
    entree[i] = i;
  }

  // Vecteur d'entier destine a accueillir les sommes prefixes.
  std::vector< int > sortie(entree.size());

  // Instanciation de la classe BodyPlus.
  BodyPlus< int, size_type > body(entree, sortie);

  // Calcul des sommes prefixes.
  tbb::parallel_scan(tbb::blocked_range< size_type >(0, entree.size()), body);

  // Sommes de tous les elements du vecteur.
  std::cout << body.lireSomme() << std::endl;

  // Tout s'est bien passe.
  return EXIT_SUCCESS;
```

28/04/2012 parallel_scan.cpp 4

```
}
```

FIGURE 4 – Calcul de sommes préfixes avec l'algorithme `parallel_scan` (dernière partie).

4. `void reverse_join(const Body& autre)` qui concatène dans `this` les résumés de `this` et `autre`. Contrairement à la méthode `join` de l'algorithme `parallel_reduce`, l'instance `this` qui invoque cette méthode représente l'opérande de droite et non l'opérande de gauche de l'opération $\oplus$. Par conséquent, l'instanciation de `this` via le constructeur par scission est antérieure à celle de `autre` ;
5. `void assign(const Body& autre)` qui affecte à `this` le résumé de `autre`.

1.1.7 `parallel_invoke`

Cet algorithme permet d'invoquer simultanément de deux à dix fonctions sans argument. Les résultats éventuels retournés par ces dernières sont ignorés. Ses surcharges sont définies dans le module `tbb/parallel_invoque.h` :

```
1  template<typename Func0, typename Func1>
2  void parallel_invoke(Func0 f0, Func1 f1);
3
4  template<typename Func0, typename Func1, typename Func2>
5  void parallel_invoke(Func0 f0, Func1 f1, Func2 f2);
6
7  ...
8
9  template<typename Func0, typename Func1 ... typename Func9>
10 void parallel_invoke(Func0 f0, Func1 f1 ... Func9 f9);
```

Les fonctions, sans paramètre, peuvent être :

1. des fonctions du programme ;
2. des méthodes de classe ;
3. des objets fonction dont la classe propose une surcharge sans paramètre de l'opérateur fonction ;
4. des lambdas, par exemple `[]{ f(5, 2); }`.

1.2 Pipeline logiciel

Certaines opérations telles que les multiplications et les additions, qu'elles soient entières ou flottantes, sont très présentes dans les corps de boucles. Ces instructions présentent la particularité d'être décomposables en étapes, les sorties de l'étape i constituant les entrées de l'étape $i + 1$. Il est alors possible d'accélérer l'exécution des boucles correspondantes en réalisant un opérateur câblé implémentant le principe du pipeline.

Dans un premier temps, une ressource matérielle appelée « étage » est associée à chaque étape de l'instruction, cette ressource étant dotée de registres d'entrée et de sortie. Dans un second temps, les étages de l'opérateur sont synchronisés par une horloge. À chaque top de cette horloge, les instructions en cours d'exécution dans le pipeline avancent d'un étage.

Prenons l'exemple d'une addition flottante : celle-ci peut être segmentée en trois étapes consécutives :

1. comparaison des exposants et alignement de la mantisse du plus petit nombre sur celle du plus grand (dénormalisation) ;
2. addition des mantisses en virgule fixe ;
3. normalisation du résultat (IEEE 754).

La figure 5 présente l'opérateur pipeliné correspondant.

Lorsque les temps de traversée des étages sont homogènes, le pipeline se trouve alors dans le cas idéal. En ajustant la période de l'horloge sur le temps de traversée des étages, nous pouvons alors accélérer l'exécution des longues boucles d'un facteur proche du nombre d'étages du pipeline.

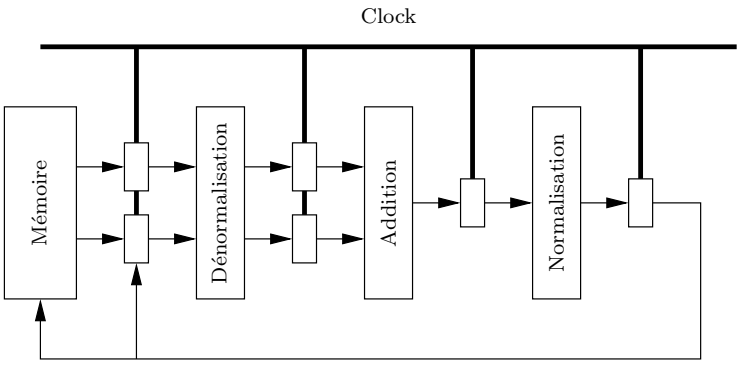


FIGURE 5 – Additionneur en virgule flottante.

Un pipeline logiciel peut être vu comme une implémentation du *behavioral design pattern* `Chain Of Responsability` (Figure 6) puisque les étages, symbolisés par des classes dérivant d'une classe de base abstraite, sont organisés en liste simplement chaînée et qu'une requête est propagée de proche en proche. Cependant, à la différence de ce *pattern*, la requête n'est pas émise par un client mais le premier étage du pipeline et cette dernière subit des modifications au fil des étages.

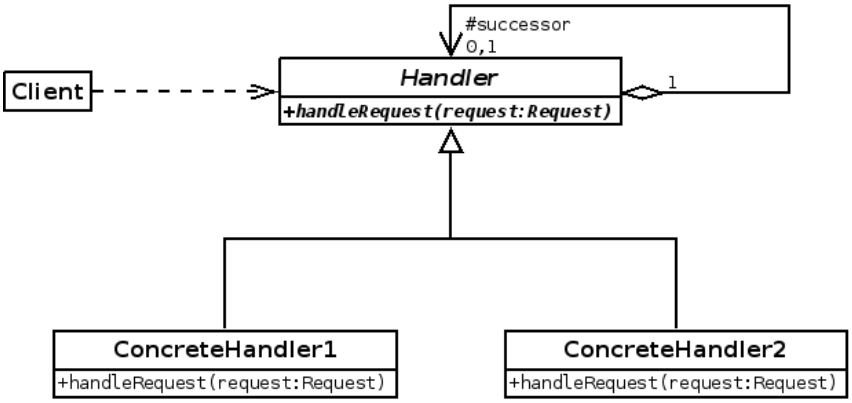


FIGURE 6 – Diagramme de classes du *behavioral design pattern* `Chain Of Responsability`.

Considérons un cas d'école dans lequel un fichier de données au format texte est composé de plusieurs lignes, chaque ligne contenant trois valeurs flottantes a, x et b. Nous souhaitons effectuer un traitement qui consiste à lire ce fichier de données ligne par ligne, à calculer pour chaque ligne le résultat $y = a \times x + b$ puis à générer dans un fichier de sortie au format texte une nouvelle ligne composée des données a, x, b et y.

Une façon naïve de procéder consisterait à lire chaque ligne, effectuer le calcul puis générer la ligne de sortie. Cette solution est typiquement une implémentation du modèle de VON NEUMAN puisque nous chargeons des opérandes, effectuons le calcul puis stockons le résultat. Par conséquent, nous nous bornons à présenter des opérandes et attendons la fin du calcul avant de présenter de nouveaux opérandes et ainsi de suite.

Une manière beaucoup plus efficace de procéder consiste à pipeliner l'ensemble, c'est à dire charger une nouvelle ligne pendant que nous effectuons un calcul et générons un résultat pour des lignes antérieures différentes.

Nous allons donc réaliser un pipeline logiciel composé de quatre étages :

1. le premier étage chargera une ligne du fichier d'entrée puis allouera dynamiquement l'opération à réaliser avant de la passer à l'étage suivant ;
2. le deuxième étage réalisera la partie multiplicative de l'opération puis passera cette dernière, encore inachevée, au troisième étage ;
3. le troisième étage réalisera la partie additive de l'opération et produira de ce fait son résultat final. L'opération, maintenant achevée, sera transmise au quatrième et dernier étage ;
4. le quatrième étage générera une ligne de données à partir de l'opération reçue avant de la désallouer dynamiquement.

La première chose à faire est de définir une classe `Operation` (ou toute autre structure de données) représentant l'opération à réaliser. La figure 7 présente sa définition *inline*.

TBB permet de réaliser un pipeline logiciel via ses classes `pipeline` et `filter`, toutes deux définies dans le module `tbb/pipeline.h`.

Nous allons donc commencer par définir une classe `EtageEntree` représentant l'étage d'entrée de notre pipeline. Cette dernière aura pour seul et unique attribut le flot d'entrée sur lequel lire les opérations à effectuer ligne par ligne. Ce flot aura été préalablement ouvert par le client du pipeline et sera supposé ne contenir aucune erreur. La figure 8 présente la définition *inline* de cette classe.

La classe `EtageEntree` dérive de la classe de base abstraite `filter`. Le constructeur logique de cette dernière prend comme argument une valeur indiquant le mode de fonctionnement de l'étage (sous-type énuméré `filter::mode`). Trois valeurs sont possibles.

`serial_in_order :` qui désigne un étage séquentiel c'est à dire non réplicable (n'existe qu'en un seul exemplaire et ne peut donc traiter qu'un seul jeton à la fois). Cette valeur impose un ordre dans le traitement des jetons. Si k désigne le rang de cet étage dans le pipeline alors tout étage séquentiel de rang $k' > k$ instancié avec la même valeur respectera cet ordre. De fait, si l'un de ces étages reçoit un jeton qui ne le respecte pas, alors ce dernier sera mis en attente et ne sera transmis à l'étage suivant que lorsque l'ordre sera rétabli.

`serial_out_of_order :` qui désigne également un étage séquentiel mais à qui l'on n'impose pas de respecter un ordre particulier dans le traitement des jetons. Par conséquent, tout jeton reçu est immédiatement traité puis transmis à l'étage suivant.

`parallel :` qui désigne un étage parallèle c'est à dire réplicable (selon le nombre de threads disponibles). Contrairement à un étage séquentiel, un étage parallèle peut traiter simultanément plusieurs jetons ; il est donc susceptible de faire parvenir à l'étage suivant des jetons ne respectant pas un ordre de traitement pré-établi.

Dans notre cas, nous utilisons la valeur `serial_in_order` car nous voulons que les jetons soient traités dans l'ordre de leur apparition dans le fichier de données. Notons que nous utilisons ici un attribut de type pointeur pour le flot d'entrée de caractères ; comme ce dernier n'est pas susceptible d'être modifié par le biais d'un mutateur, nous aurions également pu utiliser un attribut de type référence variable pour ce flot.

La classe de base `filter` impose de redéfinir la méthode abstraite `void* operator()(void* item)`.

Le paramètre de cette méthode représente le jeton transmis par l'étage précédent dans le pipeline. Son résultat représente le jeton à transmettre à l'étage suivant. La valeur spéciale `NULL` traduit l'épuisement du flot de jetons.

FIGURE 7 – Classe `Operation` représentant l'opération $y = a \times x + b$.

```
30/04/2012                    EtageEntree.hh                    1

#ifndef _EtageEntree_hh
#define _EtageEntree_hh

#include "Operation.hh"
#include <tbb/pipeline.h>
#include <iostream>
#include <sstream>

/**
 * @class EtageEntree EtageEntree.hh
 * @brief Etage d'entree du pipeline.
 * @author Emmanuel.Cagniot - @c Emmanuel.Cagniot@ensicaen.fr
 * @date 29.4.2012
 *
 * Definition inline de la classe EtatEntree representant l'etage d'entree
 * du pipeline, c'est a dire l'etage charge de lire les operations a effectuer
 * sur un flot d'entree, allouer dynamique une operation et l'envoyer a l'etage
 * charge de la multiplication.
 */
class EtageEntree: public tbb::filter {
public:
  /**
   * Constructeur logique.
   *
   * @param[in,out] entree - la valeur de @ref _ptrEntree.
   */
  EtageEntree(std::istream& entree):
    tbb::filter(tbb::filter::serial_in_order),
    _ptrEntree(&entree) {
  }

public:
  /**
   * Accesseur.
   *
   * @return la valeur de @ref _ptrEntree.
   */
  const std::istream& lireEntree() const {
    return *_ptrEntree;
  }

public:
  void* operator()(void*) {

    // Lecture d'une nouvelle ligne du fichier d'entree.
    std::string buffer;
    if (! std::getline(*_ptrEntree, buffer)) {
      return NULL;
    }

    // Extraction des informations.
    std::istringstream istr(buffer);
    double a, x, b;
    istr >> a >> x >> b;

    // Instanciation de l'operation a traiter.
    return new Operation(a, x, b);
```



```
  }

protected:
  /**
   * Flot d'entree.
   */
  std::istream* const _ptrEntree;
};

#endif
```

FIGURE 8 – Classe **EtageEntree** représentant l'étage d'entrée.

Dans notre cas, le paramètre `item` est inutilisé puisque la classe `EtageEntree` représente le premier étage de notre pipeline. Une fois la prochaine ligne extraite du fichier de données, la redéfinition de l'opérateur fonction alloue dynamiquement une instance de classe `Operation` puis la transmet à l'étage suivant.

La figure 9 présente les définitions *inline* des classes `EtageMult` et `EtageAdd` représentant respectivement les étages multiplicatif et additif de notre pipeline.

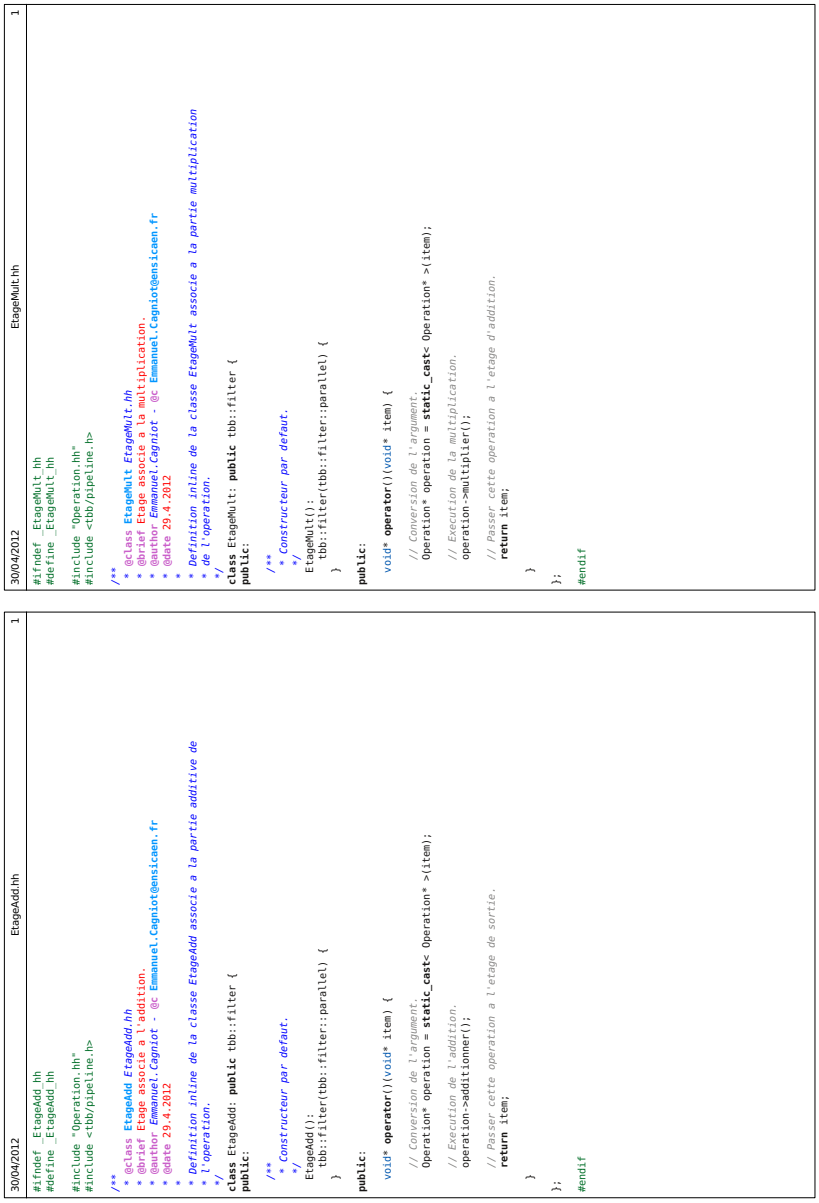


FIGURE 9 – Classes `EtageMult` et `EtageAdd` représentant respectivement les étages multiplicatif et additif.

Les classes `EtageMult` et `EtageAdd` sont relativement semblables, la première invoquant la méthode `multiplier` de son jeton et l'autre la méthode `additionner`. Comme il n'existe aucune dépendance entre instructions successives, ces classes sont instanciées avec la valeur `parallel`, ce qui leur permet de pouvoir traiter plusieurs opérations simultanément.

La figure 10 présente la définition *inline* de la classe `EtageSortie` qui représente l'étage de sortie de notre pipeline.

L'unique attribut de cette classe représente le flot de sortie sur lequel écrire les résultats. Ce flot est supposé ouvert par le client du pipeline.

Comme la classe `EtageEntree`, la classe `EtageSortie` est instanciée avec la valeur `serial_in_order`, ce qui fait que l'ordre d'écriture des résultats dans le fichier de sortie est l'ordre de lecture du fichier d'entrée. Une fois l'écriture effectuée, la redéfinition de l'opérateur fonction désalloue dynamiquement le jeton alloué par l'étage d'entrée.

Finalement, la figure 11 présente le code du client du pipeline, c'est à dire le programme principal dans notre cas.

La création d'un pipeline vide est effectuée en instanciant la classe `pipeline`.

Les étages sont ensuite instanciés puis ajoutés en queue via la méthode `add_filter`. Cette dernière prend un étage en argument mais utilise son adresse en interne. Par conséquent, l'instance concernée ne peut être une variable temporaire anonyme.

Une fois le pipeline assemblé, celui-ci est mis en marche via sa méthode `run`. L'argument fourni à cette méthode est le nombre maximum de jetons pouvant circuler simultanément dans le pipeline.

Le choix de cette valeur est crucial pour la performance du pipeline. Une valeur trop faible conduit à une sous-utilisation du pipeline tandis qu'une valeur trop importante peut conduire à un « embouteillage ». Dans notre cas, nous fixons arbitrairement cette valeur au nombre d'étages constituant notre pipeline.

Lorsque le pipeline a terminé son exécution, il faut invoquer sa méthode `clear` avant que les étages qui le composent invoquent leur destructeur. Cette philosophie est celle de la bibliothèque standard qui demande à ce qu'un élément ne soit pas détruit tant qu'il appartient à un conteneur.

L'exemple présenté ci-dessus est un cas d'école simpliste et la documentation de TBB indique qu'il ne faut pas espérer un facteur d'accélération supérieur à deux car le coût d'une lecture dans le fichier de données est largement supérieur à celui nécessaire pour effectuer une multiplication suivie d'une addition. La documentation de TBB recommande d'utiliser les pipelines logiciels pour des applications destinées à manipuler d'importants volumes de données et dans lesquelles les calculs à effectuer sur ces données sont relativement lourds.

1.3 Pipeline logiciel exprimé en graphe de flot de contrôle

Le pipeline logiciel est un cas particulier des graphes de flot de contrôle (*Control Graph Flow*). Depuis sa version 4.0, TBB propose cette notion très pointue par le biais d'une multitude de types de nœuds regroupés dans le module `tbb/flow.h`.

La figure 12 présente ces différents types organisés en quatre familles :
— `functional` : reçoivent des messages, exécutent du code fourni par l'utilisateur pour traiter ces messages et produisent des messages ;
— `buffering` : bufferisent les messages selon une organisation spécifique ;
— `split/join` : concentrent ou répartissent les messages ;
— `other` : tout ce qui n'entre pas dans l'un des trois familles précédentes.

Il n'est pas question d'inventorier ici chaque type de nœud. Aussi allons nous donner un exemple tiré de la littérature TBB : la mise en œuvre d'un pipeline logiciel.

```
30/04/2012                    EtageSortie.hh                    1

#ifndef _EtageSortie_hh
#define _EtageSortie_hh

#include "Operation.hh"
#include <tbb/pipeline.h>
#include <iostream>

/**
 * @class EtageSortie EtageSortie.hh
 * @brief Etage de sortie.
 * @author Emmanuel.Cagniot - @c Emmanuel.Cagniot@ensicaen.fr
 * @date 29.4.2012
 *
 * Definition inline de la classe EtatSortie representant l'etage de sortie du
 * pipeline, c'est a dire l'etage charge d'ecrire l'operation et son resultat
 * sur un flot de sortie avant de la desallouer dynamiquement.
 */
class EtageSortie: public tbb::filter {
public:
  /**
   * Constructeur logique.
   *
   * @param[in,out] sortie - la valeur de @ref _sortie.
   */
  EtageSortie(std::ostream& sortie):
    tbb::filter(tbb::filter::serial_in_order),
    _ptrSortie(&sortie) {
  }

public:
  /**
   * Accesseur.
   *
   * @return la valeur de @ref _sortie.
   */
  const std::ostream& lireSortie() const {
    return *_ptrSortie;
  }

public:
  void* operator()(void* item) {

    // Conversion de l'argument.
    Operation* operation = static_cast< Operation* >(item);

    // Ecriture de l'information sur le flot de sortie.
    *_ptrSortie << operation->lireA()
                << '\t'
                << operation->lireX()
                << '\t'
                << operation->lireB()
                << '\t'
                << operation->resultat()
                << std::endl;

    // Desallocation de l'operation allouee par l'etage d'entree.
    delete operation;
```

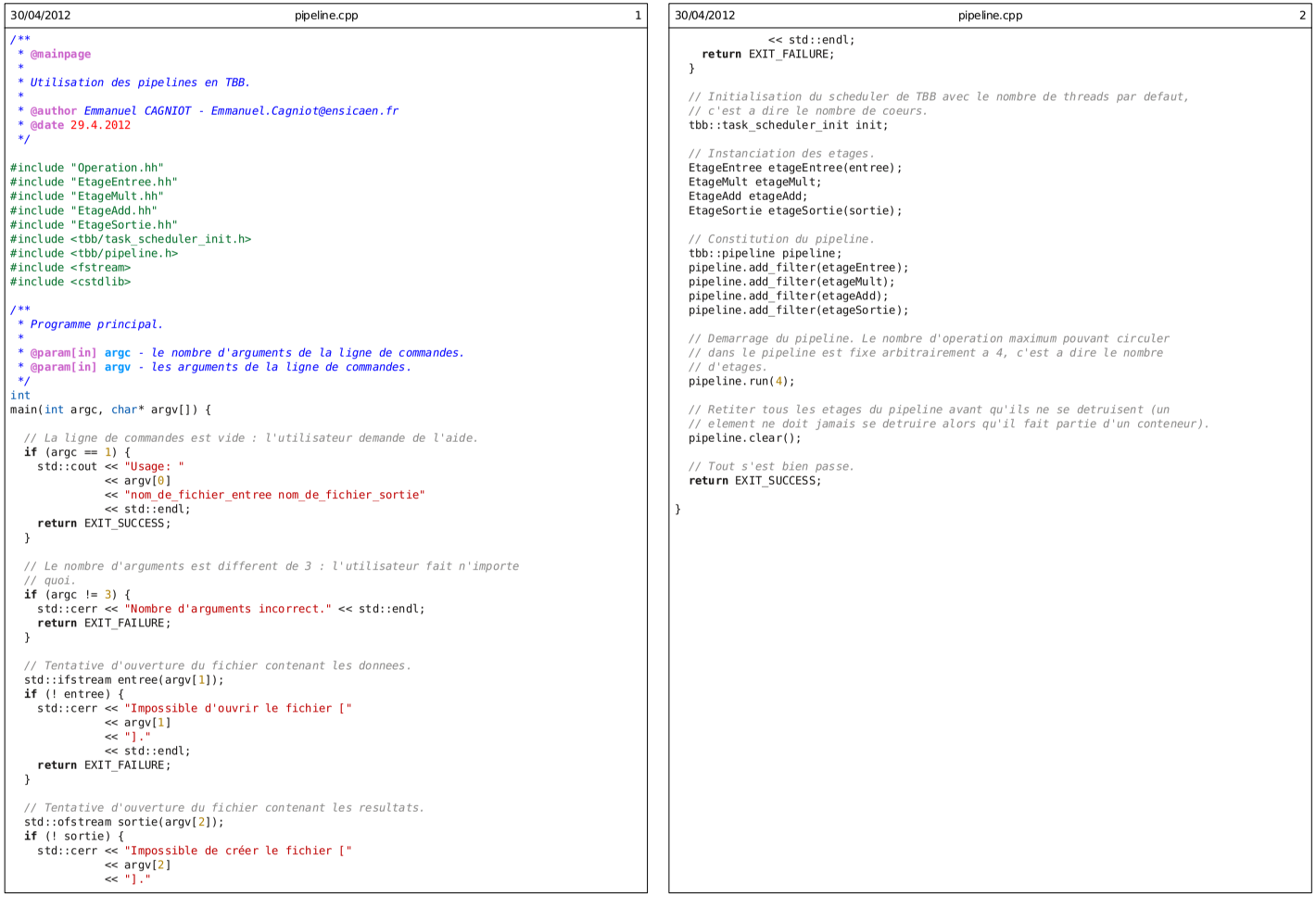
```
30/04/2012                    EtageSortie.hh                    2

    // Rien a retourner.
    return NULL;
  }

protected:
  /**
   * Flot de sortie.
   */
  std::ostream* const _ptrSortie;
};

#endif
```

FIGURE 10 – Classe `EtageSortie` représentant l'étage de sortie.



30/04/2012 pipeline.cpp 1

```cpp
/**
 * @mainpage
 *
 * Utilisation des pipelines en TBB.
 *
 * @author Emmanuel CAGNIOT - Emmanuel.Cagniot@ensicaen.fr
 * @date 29.4.2012
 */

#include "Operation.hh"
#include "EtageEntree.hh"
#include "EtageMult.hh"
#include "EtageAdd.hh"
#include "EtageSortie.hh"
#include <tbb/task_scheduler_init.h>
#include <tbb/pipeline.h>
#include <fstream>
#include <cstdlib>

/**
 * Programme principal.
 *
 * @param[in] argc - le nombre d'arguments de la ligne de commandes.
 * @param[in] argv - les arguments de la ligne de commandes.
 */
int
main(int argc, char* argv[]) {

  // La ligne de commandes est vide : l'utilisateur demande de l'aide.
  if (argc == 1) {
    std::cout << "Usage: "
              << argv[0]
              << "nom_de_fichier_entree nom_de_fichier_sortie"
              << std::endl;
    return EXIT_SUCCESS;
  }

  // Le nombre d'arguments est different de 3 : l'utilisateur fait n'importe
  // quoi.
  if (argc != 3) {
    std::cerr << "Nombre d'arguments incorrect." << std::endl;
    return EXIT_FAILURE;
  }

  // Tentative d'ouverture du fichier contenant les donnees.
  std::ifstream entree(argv[1]);
  if (! entree) {
    std::cerr << "Impossible d'ouvrir le fichier ["
              << argv[1]
              << "]."
              << std::endl;
    return EXIT_FAILURE;
  }

  // Tentative d'ouverture du fichier contenant les resultats.
  std::ofstream sortie(argv[2]);
  if (! sortie) {
    std::cerr << "Impossible de créer le fichier ["
              << argv[2]
              << "]."
```

30/04/2012 pipeline.cpp 2

```cpp
              << std::endl;
    return EXIT_FAILURE;
  }

  // Initialisation du scheduler de TBB avec le nombre de threads par defaut,
  // c'est a dire le nombre de coeurs.
  tbb::task_scheduler_init init;

  // Instanciation des etages.
  EtageEntree etageEntree(entree);
  EtageMult etageMult;
  EtageAdd etageAdd;
  EtageSortie etageSortie(sortie);

  // Constitution du pipeline.
  tbb::pipeline pipeline;
  pipeline.add_filter(etageEntree);
  pipeline.add_filter(etageMult);
  pipeline.add_filter(etageAdd);
  pipeline.add_filter(etageSortie);

  // Demarrage du pipeline. Le nombre d'operation maximum pouvant circuler
  // dans le pipeline est fixe arbitrairement a 4, c'est a dire le nombre
  // d'etages.
  pipeline.run(4);

  // Retiter tous les etages du pipeline avant qu'ils ne se detruisent (un
  // element ne doit jamais se detruire alors qu'il fait partie d'un conteneur).
  pipeline.clear();

  // Tout s'est bien passe.
  return EXIT_SUCCESS;

}
```

FIGURE 11 – Constitution puis exploitation de notre pipeline.

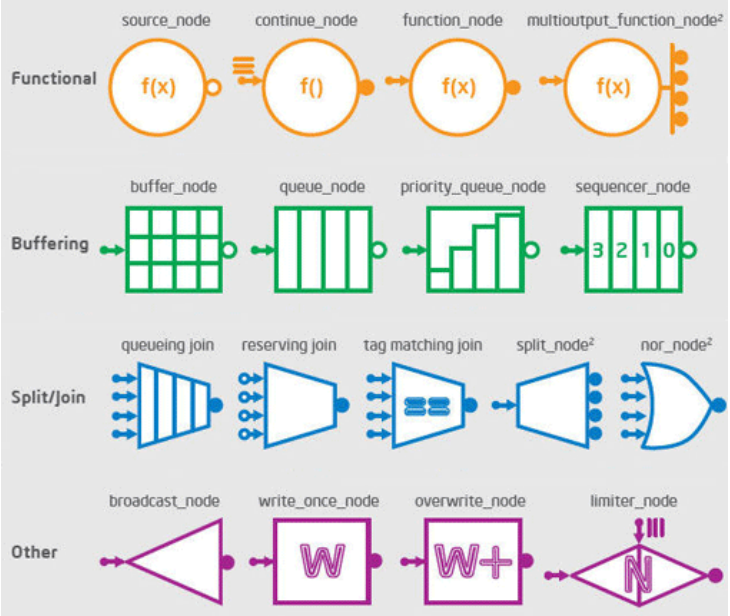


FIGURE 12 – Les différents types de nœuds dans un graphe de flot de contrôle.

Dans notre cas, nous allons recréer le pipeline logiciel de la section précédente. La figure 13 présente la modélisation générale d'un pipeline logiciel TBB dans le formalisme des graphes de flot de contrôle.

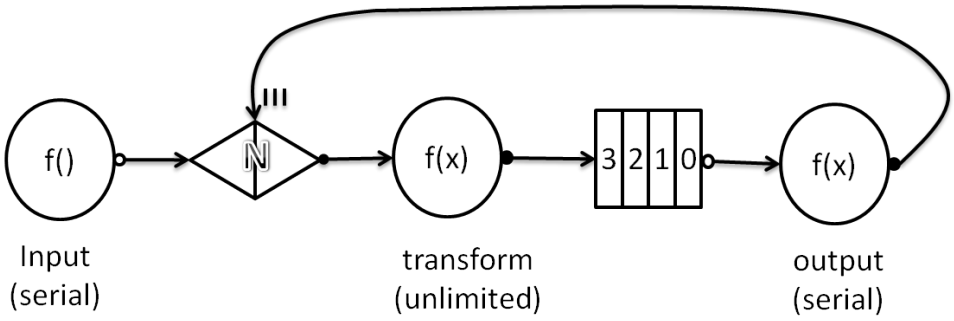


FIGURE 13 – Modélisation d'un pipeline logiciel TBB dans le formalisme des graphes de contrôle de flot.

1.3.1 Jeton

Comme pour le pipeline logiciel de la section précédente, les jetons qui circulent d'étage en étage sont les opérations allouées dynamiquement.

Cependant, la difficulté consiste à reproduire le mode de fonctionnement `serial_in_order`. Pour ce faire, chaque opération produite par l'étage d'entrée doit se voir attribuer son rang (un entier naturel) dans la chaîne de production. Aussi définissons nous une classe `OperationAvecId` (non présentée ici) dérivant de la classe `Operation` de la section précédente, celle-ci se voyant dotée d'un nouvel attribut `id_` de type `size_t`.

La classe `OperationAvecId` possède un constructeur logique prenant les trois arguments de sa classe de base plus cet entier et l'accesseur correspondant. Nous faisons ici le choix d'utiliser des pointeurs

intelligents de type `shared_ptr` en définissant le type synonyme :

```
typedef shared_ptr< OperationAvecId > GcPtrOperationAvecId;
```

À partir de maintenant, nous n'avons plus qu'à nous soucier des allocations mais pas des dés-allocations.

1.3.2 Étage d'entrée

L'étage d'entrée du pipeline est représenté par une instance de classe `source_node`. Cette classe générique est paramétrée par le type abstrait `Output` représentant les jetons produits par l'étage.

Une instance de type `source_node` est un nœud fonctionnel ne prenant aucun argument. Son mode de fonctionnement est implicitement `serial`, c'est à dire qu'il ne peut bufferiser qu'un seul jeton : tant que ce dernier n'a pas été envoyés à tous les nœuds successeurs dans le graphe, le code fourni par l'utilisateur n'est pas exécuté.

Le constructeur logique de la classe `source_node` est lui même générique et paramétré par le type abstrait additionnel `Body` représentant le code de l'utilisateur.

Les arguments attendus sont le graphe propriétaire du nœud (référence variable), le code de l'utilisateur transmis par copie et enfin un *flag* booléen indiquant si le nœud doit produire immédiatement.

Ce dernier paramètre possède une valeur par défaut valant `true`. Si l'utilisateur fournit la valeur `false` alors il devra explicitement invoquer la méthode `activate` du nœud afin de lancer la production (et donc le pipeline).

Les caractéristiques du code (`body`) fourni par l'utilisateur sont les suivantes :

1. un constructeur par recopie pouvant être implicite ;
2. un destructeur pouvant être implicite ;
3. une surcharge de l'opérateur d'affectation ne retournant rien (`void`). Si la surcharge est implicite, le résultat retourné (`Body&`) est tout simplement ignoré ;
4. une surcharge de l'opération fonctionnel `bool operator()(Output& v)` qui initialise le nouveau jeton `v`. La valeur retournée est `true` si un nouveau jeton a pu être initialisé ou `false` dans le cas contraire (fin de la production).

Dans notre cas, comme pour l'étage d'entrée de la section précédente, nous avons besoin d'un attribut représentant le flot d'entrée de caractères. Nous avons également besoin d'un compteur entier permettant de « *tagger* » chaque jeton produit.

Aussi faisons nous le choix de définir notre code sous la forme d'une classe `FnEntree` dont la définition *inline* est présentée en Figure 14.

1.3.3 Limiteur de jetons

L'une des caractéristiques d'un pipeline logiciel TBB est de pouvoir fixer le nombre de jeton en circulation à un instant donné.

Dans un graphe de contrôle de flot, la classe générique `limiter_node` est dédiée à cette tâche. Celle-ci est paramétrée par le type abstrait `T` représentant les jetons bufferisés.

Son constructeur logique prend comme argument le graphe propriétaire du nœud (référence variable), le nombre de jetons maximum pouvant être diffusés à tous les successeurs et un troisième paramètre prenant une valeur par défaut (nous ne détaillerons par ce dernier à la fonction particulière).

```
29/04/2014                FnEntree.hpp                1

#ifndef FnEntree_hpp
#define FnEntree_hpp

#include "GcPtrOperationAvecId.hpp"
#include <sstream>
#include <iostream>

/**
 * @class FnEntree FnEntree.hpp
 *
 * Définition inline de la classe FnEntree associé à au noeud
 * d'entrée du graphe. Ce dernier est un noeud source dont la
 * fonction consiste à lire le fichier source auquel il est
 * connecté ligne par ligne, instancier des opérations et les
 * passer à son successeur dans le graphe.
 */
class FnEntree {
public:

  /**
   * Constructeur logique.
   *
   * @param[in, out] source - la valeur de @ref source_.
   */
  FnEntree(std::istream& source):
      source_(source),
      compteur_(0) {
  }

public:

  /**
   * Accesseur.
   *
   * @return la valeur de @ref source_.
   */
  const std::istream& lireSource() const {
    return source_;
  }

  /**
   * Accesseur.
   *
   * @return la valeur de @ref compteur_.
   */
  const size_t& lireCompteur() const {
    return compteur_;
  }

public:

  /**
   * Opérateur () prenant en argument un pointeur intelligent vers
   * l'opération à allouer.
   *
   * @param[in,out] v - le pointeur vers l'opération à allouer.
   * @return @c true si une opération a pu être allouée sinon
   * @c false.
   */
```

```
29/04/2014                FnEntree.hpp                2

  bool operator()(GcPtrOperationAvecId& v) {

    // Tentative de lecture d'un nouveau jeu de donnée dans le
    // fichier source.
    std::string buffer;
    if (!std::getline(source_, buffer)) {
      return false;
    }

    // Une ligne a pu être lu : nous créons l'opération
    // correspondante.
    std::istringstream istr(buffer);
    double a, x, b;
    istr >> a >> x >> b;
    v.reset(new OperationAvecId(a, x, b, compteur_ ++));

    // Et nous indiquons qu'une nouvelle opération a été produite.
    return true;
  }

protected:

  /**
   * Flot d'entrée connecté au fichier source.
   */
  std::istream& source_;

  /**
   * Compteur des opérations déjà produites.
   */
  size_t compteur_;
};
#endif
```

FIGURE 14 – Classe **FnEntree** représentant le code exécuté par l'étage d'entrée du pipeline.

Le limiteur accepte de recevoir des jetons des nœuds prédécesseurs tant que le nombre maximum de jetons qu'il est autorisé à diffuser à ses successeurs n'est pas atteint. Lorsque c'est le cas, un objet interne appelé « décrémenteur » doit recevoir un message de classe `continue_msg` pour que le compteur des jetons diffusés sont décrémenté d'une unité (le limiteur est alors autorisé à diffuser un nouveau jeton).

Dans notre cas, comme pour le pipeline de la section précédente, nous limitons à quatre le nombre de jetons en circulation à un instant donné. La figure 15 présente la construction de notre limiteur de jetons.

1.3.4 Étages multiplicatif et additif

Les étages de multiplication et d'addition sont représentés par des instances de classe `function_node` représentant des nœuds capables d'exécuter le code fourni par l'utilisateur de façon concurrente (équivalent du mode de fonctionnement `parallel` dans les pipelines logiciels TBB).

Cette classe générique est paramétrée par quatre types abstraits, les trois derniers prenant des valeurs par défaut. Nous ne présentons ici que les trois premiers, le dernier étant dédié à la politique d'allocation dans les lignes de cache :
- `Input` : les jetons reçus ;
- `Output = continue_msg` : les jetons produits (par défaut un message blanc) ;
- `graph_buffer_policy = queueing` : la politique de gestion des jetons en attente de traitement (par défaut une file).

Comme celui de la classe `source_node`, le constructeur logique de la classe `function_node` est générique et paramétré par le type abstrait additionnel `Body` représentant le code fourni par l'utilisateur.

Ses arguments sont le graphe propriétaire du nœud (référence variable), une paramètre `concurrency` de type `size_t` représentant le nombre de jetons pouvant être traités simultanément et enfin, le code de l'utilisateur transmis par copie.

Le paramètre `concurrency` possède deux valeurs particulières : `serial` qui force un fonctionnement séquentiel et `unlimited` qui, au contraire, permet de traiter simultanément autant de jetons que possible.

Les caractéristiques du code (`body`) fourni par l'utilisateur sont les suivantes :

1. un constructeur par recopie pouvant être implicite ;
2. un destructeur pouvant être implicite ;
3. une surcharge de l'opérateur d'affectation ne retournant rien (`void`). Si la surcharge est implicite, le résultat retourné (`Body&`) est tout simplement ignoré ;
4. une surcharge de l'opération fonctionnel `Output operator()(const Input& v) const` qui traite le jeton en entrée `v` et retourne le résultat de ce traitement.

Dans notre cas, les traitement étant relativement simples, nous choisissons de les définir via des *lambdas*. Nous étages reçoivent des opérations en entrées et produisent la même opération en sortie. La figure 15 présente leur construction.

1.3.5 Ordonnanceur de jetons

Les jetons produits par l'étage d'entrée doivent être traités dans le même ordre par l'étage de sortie. Dans un pipeline logiciel TBB, il suffit que le mode de fonctionnement de l'étage de sortie soit identique à celui de l'étage d'entrée c'est à dire `serial_in_order`.

Dans le cas d'un graphe de contrôle de flot, cet ordre est rétabli par le biais d'une instance de la classe générique `sequencer_node`.

Cette classe est paramétrée par un type abstrait `T` représentant les jetons bufferisés et un autre, possédant une valeur par défaut et relatif à la politique d'allocation dans les lignes de cache.

Comme les classes `source_node` et `function_node`, son constructeur logique est lui même générique et paramétré par le type abstrait additionnel `Body` représentant le code d'ordonnancement fourni par l'utilisateur. Ses paramètres sont respectivement le graphe propriétaire du nœud (référence variable) et ce code transmis par copie. Ce dernier doit présenté les caractéristiques suivantes :

1. un constructeur par recopie pouvant être implicite ;
2. un destructeur pouvant être implicite ;
3. une surcharge de l'opérateur d'affectation ne retournant rien (`void`). Si la surcharge est implicite, le résultat retourné (`Body&`) est tout simplement ignoré ;
4. une surcharge de l'opération fonctionnel `size_t operator()(const & v)` qui retourne le rang du jeton `v` dans le processus de production.

Dans notre cas, le code d'ordonnancement étant trivial, nous choisissons de le définir sous la forme d'une *lambda*. La figure 15 présente la construction de notre ordonnanceur de jetons.

1.3.6 Étage de sortie

L'étage de sortie possède comme attribut le flot de sortie auquel il est connecté. Son rôle consiste à recevoir une opération et à écrire ses opérandes ainsi que son résultat sur le flot.

Contrairement à l'étage de sortie du pipeline logiciel de la section précédente, la désallocation de l'opération est implicitement assurée par le mécanisme de garbage collector intégré aux `shared_ptr`.

Lorsqu'il a traité une opération, l'étage de sortie doit envoyer un `continue_msg` au limiteur de jeton afin de le débloquer si besoin est.

Par conséquent, comme les étages de multiplication et d'addition, nous modélisons l'étage de sortie comme une instance de classe `function_node`, en mode de fonctionnement `serial`, prenant un jeton en entrée et produisant une instance de classe `continue_msg` en sortie. Comme pour les étages multiplicatif et additif, nous fournissons le code à exécuter via une lambda. La figure 15 présente la construction de notre étage de sortie.

1.3.7 Création des arcs et démarrage

La création d'un arc entre un nœud source et un nœud cible s'effectue via la fonction `make_edge` dont le premier argument représente la source et le second la cible.

Dans notre cas, nous utilisons cette fonction pour traduire les connexions de la figure 13. La figure 15 présente cette dernière étape du processus. Nous constatons que le démarrage du pipeline est simplement assuré par l'invocation de la méthode `wait_for_all` du graphe.

1.4 Tâche et *scheduler*

L'unité de programmation en TBB n'est pas le thread mais la tâche, c'est à dire une instance dont la classe dérive de la classe abstraite `task` définie dans le module `tbb/task.h`.

Les tâches à exécuter sont confiées aux différents threads disponibles par le *scheduler*.

La classe `task` impose à ses classes dérivées de redéfinir la méthode `task* execute()` dont le résultat est soit la prochaine tâche à exécuter, soit la valeur spéciale `NULL`.

29/04/2014 graphe.cpp 1

```
// Création du graphe:
tbb::flow::graph graphe;

// Type synonyme pour un pointeur intelligent d'opération.
typedef GcPtrOperationAvecId Jeton;

// Création du noeud source via une classe.
tbb::flow::source_node< Jeton > ndEntree(graphe,
                                         FnEntree(entree));

// Création du noeud limiteur (4 opérations maxi à un
// instant donné).
tbb::flow::limiter_node< Jeton > ndLimiteur(graphe, 4);

// Création des noeuds multiplicatif et additif via des
// lambdas.
tbb::flow::function_node< Jeton, Jeton >
    ndMult(graphe,
           tbb::flow::unlimited,
           [](const Jeton& jeton) -> Jeton {
               jeton->multiplier();
               return jeton;
           });
tbb::flow::function_node< Jeton, Jeton >
    ndAdd(graphe,
          tbb::flow::unlimited,
          [](const Jeton& jeton) -> Jeton {
              jeton->additionner();
              return jeton;
          });

// Création du noeud ordonnanceur via une lambda.
tbb::flow::sequencer_node< Jeton >
    ndOrdonnanceur(graphe,
                   [](const Jeton& jeton) -> size_t {
                       return jeton->lireId();
                   });

// Création du noeud de sortie via une lambda.
tbb::flow::function_node< Jeton >
    ndSortie(graphe,
             tbb::flow::serial,
             [&sortie](const Jeton& v) -> tbb::flow::continue_msg {
                 sortie << v->lireA()
                        << "\t"
                        << v->lireX()
                        << "\t"
                        << v->lireB()
                        << "\t"
                        << v->resultat()
                        << std::endl;
                 return tbb::flow::continue_msg();
             });

// Création des arcs
tbb::flow::make_edge(ndEntree, ndLimiteur);
tbb::flow::make_edge(ndLimiteur, ndMult);
tbb::flow::make_edge(ndMult, ndAdd);
tbb::flow::make_edge(ndAdd, ndOrdonnanceur);
```

29/04/2014 graphe.cpp 2

```
tbb::flow::make_edge(ndOrdonnanceur, ndSortie);
tbb::flow::make_edge(ndSortie, ndLimiteur.decrement);

// C'est parti.
graphe.wait_for_all();
```

FIGURE 15 – Partie du code relatif à la constitution du pipeline et son démarrage.

Une tâche étant susceptible de créer d'autres tâches, nous voyons apparaître une structure d'arbre dans lequel les nœuds représentent des tâches et les arêtes, des dépendances entre ces tâches, une tâches mère ne pouvant commencer son exécution que lorsque toutes ses tâches filles ont terminé la leur.

Deux informations sont implicitement associées à chaque tâche :

1. sa profondeur dans l'arbre ;
2. sa connectivité, c'est à dire le nombre de ses tâches filles n'ayant pas encore achevé leur exécution.

Une tâche est considérée comme prête à être exécutée lorsque sa connectivité devient nulle. Par conséquent, les tâches prêtes à démarrer sont situées sur les feuilles de l'arbre. La connectivité d'une tâche (et donc l'arbre) est continuellement mise à jour pendant son exécution ou celle de ses filles.

Le *scheduler* de TBB implémente un algorithme de *work-stealing* aléatoire (emprunté au langage Cilk).

Dans un premier temps, cet algorithme constitue un pool de threads logiques en instanciant la classe `task_scheduler_init` définie dans le module `task_scheduler_init.h`. Plusieurs threads logiques peuvent être associés à un même cœur.

Un ordonnancement de type temps partagé est alors mis en place sur le cœur, cet ordonnancement présentant de nombreux inconvénients en termes de performances :

1. le cœur doit sauvegarder l'état des registres du thread sortant puis restaurer ceux du thread entrant ;
2. le thread sortant doit restaurer la cohérence des données qui ont été modifiées entre le cache partagé de son cœur et la mémoire ;
3. le thread sortant doit lever chaque verrou qu'il a posé.

Toutes ces opérations génèrent un *overhead* important. Par conséquent, il est préférable d'utiliser la configuration optimale affectant un thread logique par cœur.

Une pile et un « *ready deque* » (*Double Ended Queue*) sont associés à chaque thread logique. Le rôle de la pile est de stocker l'environnement des tâches en cours d'exécution ou en attente. Le *ready deque* stocke, quant à lui, les tâches prêtes à démarrer. Les plus anciennes (faible profondeur) sont situées en queue tandis que les plus récentes (forte profondeur) sont situées en tête.

Un thread logique obtient du travail en retirant de son *ready deque* la tâche placée en tête. Soit $\mathcal{T}_m$ cette tâche. Le thread continue d'exécuter $\mathcal{T}_m$ jusqu'à ce que cette dernière effectue l'une des opérations suivante :

1. $\mathcal{T}_m$ lance une nouvelle tâche $\mathcal{T}_f$. La tâche mère $\mathcal{T}_m$ est insérée en tête du *ready deque* tandis que le thread commence l'exécution de la tâche fille $\mathcal{T}_f$;
2. $\mathcal{T}_m$ se bloque (en attente d'une opération ou d'un résultat). Le thread inspecte son *ready deque*. Si ce dernier n'est pas vide, il retire la tâche située en tête et commence son exécution. Dans le cas contraire, il choisit aléatoirement le *ready deque* d'un autre thread et recommence éventuellement jusqu'à en trouver un qui ne soit pas vide et dont la profondeur de sa tâche la plus ancienne (située en queue) soit supérieure à celle de $\mathcal{T}_m$. Il ponctionne alors cette tâche et commence son exécution. Cette technique de chapardage est appelée « *randomized work-stealing* » ;
3. $\mathcal{T}_m$ termine son exécution. Le thread applique la règle précédente ;
4. $\mathcal{T}_m$ active une tâche bloquée. Le thread insère cette dernière en tête de son *ready deque*.

Une tâche peut activer une autre tâche puis immédiatement se bloquer ou interrompre son exécution. Dans ce cas, la dernière règle est appliquée puis la seconde ou la troisième selon le cas. Lorsque l'application est lancée, le *ready deque* de chaque thread logique est vide. La première tâche créée est alors placée dans l'un d'entre eux tandis que les autres threads commencent le *work-stealing*.

L'analyse de cet algorithme montre deux choses :

1. le *ready deque* de chaque thread autorise une exploration de l'arbre des tâches de l'application en profondeur d'abord. Ce type d'exploration permet de limiter la consommation de mémoire aux seules tâches de la branche actuellement explorée tout en exploitant la localité des données dans les caches du cœur ;
2. le chapardage de tâches revient à explorer l'arbre des tâches de l'application en largeur d'abord, ce qui permet de toujours maintenir les cœurs en activité.

Par conséquent, cet algorithme hybride combinant parcours en largeur et en profondeur d'abord, conduit à un bon équilibrage de charge doublé d'une bonne exploitation de la localité des données dans les caches.

1.4.1 Attente explicite

Comme première illustration, nous allons nous inspirer de l'exemple donné dans la documentation de TBB, exemple consistant à calculer de façon naïve (récursive), le nième terme de la suite de FIBONACCI définie par :

$$\forall n \in [0,1], \quad f(n) = n, \tag{1.4}$$
$$\forall n > 1, \quad f(n) = f(n-1) + f(n-2). \tag{1.5}$$
$$\tag{1.6}$$

Afin de pouvoir introduire ultérieurement les optimisations liées au *scheduler* de TBB sans remettre en cause la majorité des classes écrites pour cet exemple, nous allons utiliser un schéma un peu plus compliqué faisant intervenir le *creational design pattern* `Abstract Factory`.

Dans un premier temps, nous définissons donc la classe abstraite `FabriqueAbstraite` et la classe `FabriqueTacheFibonacci` permettant d'instancier une tâche de calcul sans parent (root) pour le calcul d'un terme de la suite.

La figure 16 présente les définitions *inline* de ces deux classes.

Les arguments de la méthode `parent` représentent respectivement le rang du terme à calculer, le seuil (rang) sous lequel le calcul se poursuit de façon séquentielle et enfin, le lieu de stockage du résultat final.

La tâche est allouée dynamiquement via une surcharge de l'opérateur `new` faisant appel à la méthode de classe `task::allocate_root`. L'effet de cet appel est d'allouer un espace mémoire recyclable par la suite pour une tâche sans parent.

La méthode `terme` de la classe `Fibonacci` (Figure 17) lance le calcul du terme demandé en instanciant, via sa fabrique, une tâche sans parent de classe `TacheFibonacci` et en la lançant via la méthode de classe `task::spawn_root_and_wait`. Cette méthode, comme son nom l'indique, attend la fin de l'exécution de la tâche lancée avant de la désallouer.

La classe `TacheFibonacci` (Figure 18) représente le calcul parallèle d'un terme de la suite.

La redéfinition de la méthode `execute` commute en mode séquentiel lorsque le rang du terme demandé passe sous le seuil. Si tel n'est pas le cas, elle alloue dynamiquement deux tâches filles via la surcharge de l'opérateur `new` et la méthode de classe `task::allocate_child`.

Les deux tâches filles sont instanciées avec des références sur des données locales de la tâche mère (ici les variables `x`, `y`) et une autre sur l'attribut `_seuil`. Ce détail est important car pour pouvoir s'exécuter, les tâches filles ont besoin que l'environnement de la tâche mère soit maintenu dans la pile du thread.

FABRIQUEABSTRAITE.HH

```
03/05/2012                    FabriqueAbstraite.hh                    1

#ifndef _FabriqueAbstraite_hh
#define _FabriqueAbstraite_hh

#include <ttb/task.h>

/**
 * @class FabriqueAbstraite FabriqueAbstraite.hh
 * @brief Fabrique abstraite de taches de calcul pour la suite de Fibonacci.
 * @author Emmanuel.Cagniot - @c Emmanuel.Cagniot@ensicaen.fr
 * @date 3.5.2012
 *
 * Definition inline de la classe abstraite FabriqueAbstraite representant une
 * fabrique de taches de calcul pour la suite de Fibonacci.
 */
class FabriqueAbstraite {
public:
  /**
   * Destructeur.
   */
  virtual ~FabriqueAbstraite() {
  };

public:

  /**
   * Fabrique une tache parent.
   *
   * @param[in] rang - le rang du terme dont il faut calculer la valeur.
   * @param[in] seuil - le rang sous lequel le calcul se fait en sequentiel.
   * @param[in,out] - la variable dans laquelle recopier le resultat.
   */
  virtual ttb::task& tache(const unsigned long& rang,
                           const unsigned long& seuil,
                           unsigned long& resultat) const = 0;

};

#endif
```

FABRIQUETACHEFIBONACCI.HH

```
03/05/2012                  FabriqueTacheFibonacci.hh                 1

#ifndef _FabriqueTacheFibonacci_hh
#define _FabriqueTacheFibonacci_hh

#include "TacheFibonacci.hh"
#include "FabriqueAbstraite.hh"

/**
 * @class FabriqueTacheFibonacci FabriqueTacheFibonacci.hh
 * @brief Fabrique d'instances de classe TacheFibonacci.
 * @author Emmanuel.Cagniot - @c Emmanuel.Cagniot@ensicaen.fr
 * @date 3.5.2012
 *
 * Definition inline de la classe FabriqueTacheFibonacci representant une
 * fabrique concrete d'instances de classe TacheFibonacci.
 */
class FabriqueTacheFibonacci: public FabriqueAbstraite {
public:

  ttb::task& tache(const unsigned long& rang,
                   const unsigned long& seuil,
                   unsigned long& resultat) const {
    return *new(ttb::task::allocate_cont()) 
      TacheFibonacci(rang, seuil, resultat);
  }

};

#endif
```

FIGURE 16 – Fabrique de tâches de calcul pour le $n$ième terme de la suite de FIBONACCI.

05/05/2012 Fibonacci.hh 1

```
#ifndef _Fibonacci_hh
#define _Fibonacci_hh

#include "FabriqueAbstraite.hh"

/**
 * @class Fibonacci Fibonacci.hh
 * @brief Calcul d'un terme de la suite de Fibonacci par une methode recursive
 *   naive.
 * @author Emmanuel.Cagniot - @c Emmanuel.Cagniot@ensicaen.fr
 * @date 3.5.2012
 *
 * Definition inline de la classe Fibonacci permettant de calculer recursivement
 * (methode naive) un terme de la suite de Fibonacci.
 */
class Fibonacci {
public:

  /**
   * Constructeur logique.
   *
   * @param[in] fabrique - la valeur de @ref _fabrique.
   * @param[in] seuil -
   */
  Fibonacci(const FabriqueAbstraite& fabrique, const unsigned long& seuil):
    _fabrique(fabrique),
    _seuil(seuil) {
  }

public:

  /**
   * Calcule le terme dont le rang est fourni en argument.
   *
   * @param[in] rang - le rang du terme.
   * @return la valeur du terme correspondant.
   */
  unsigned long terme(const unsigned long& rang) const {
    unsigned long terme;
    tbb::task& root = _fabrique.tache(rang, _seuil, terme);
    tbb::task::spawn_root_and_wait(root);
    return terme;
  }

protected:

  /**
   * Fabrique de taches de calcul.
   */
  const FabriqueAbstraite& _fabrique;

  /**
   * Seuil (rang) a partir duquel le calcul se fait en sequentiel.
   */
  const unsigned long _seuil;

};

#endif
```

FIGURE 17 – Classe `Fibonacci` lançant la tâche de calcul initiale.

08/05/2012 TacheFibonacci.hh 1

```
#ifndef _TacheFibonacci_hh
#define _TacheFibonacci_hh

#include <tbb/task.h>

/**
 * @class TacheFibonacci TacheFibonacci.hh
 * @brief Tache de calcul d'un terme de la suite de Fibonacci en utilisant une
 *        methode recursive naive.
 * @author Emmanuel.Cagniot - @c Emmanuel.Cagniot@ensicaen.fr
 * @date 3.5.2012
 *
 * Definition inline de la classe TacheFibonacci representant une tache dediee
 * dediee au calcul d'un terme de la suite de Fibonacci. La methode recursive
 * utilisee est une methode naive.
 */
class TacheFibonacci: public tbb::task {
public:
  /**
   * Constructeur logique.
   *
   * @param[in] rang - la valeur de @ref _rang.
   * @param[in] seuil - la valeur de @ref _seuil.
   * @param[in] resultat - la valeur de @ref _resulat.
   */
  TacheFibonacci(const unsigned long& rang,
                 const unsigned long& seuil,
                 unsigned long& resultat):
    _rang(rang),
    _seuil(seuil),
    _ptrResultat(&resultat) {
  }

public:

  tbb::task* execute() {

    // Nous sommes passes sous le seuil : nous poursuivons en sequentiel.
    if (_rang < _seuil) {
      *_ptrResultat = sequentiel(_rang);
      return NULL;
    }

    // Nous sommes au dessus du seuil : nous creons deux taches filles.
    unsigned long x;
    TacheFibonacci& gauche =
      *new(allocate_child()) TacheFibonacci(_rang - 1, _seuil, x);
    unsigned long y;
    TacheFibonacci& droite =
      *new(allocate_child()) TacheFibonacci(_rang - 2, _seuil, y);

    // Nous fixons le compteur de reference de cette tache a la valeur trois
    // (et non deux) afin que le thread ne l'insere pas en tete de son ready
    // deque lors de l'appel de la methode spawn_and_wait_for_all. Il s'agit
    // ici d'une attente explicite.
    set_ref_count(3);
    spawn(droite);
    spawn_and_wait_for_all(gauche);
```

08/05/2012 TacheFibonacci.hh 2

```
    // Le resultat de la tache mere est la somme des resultats de ses taches
    // filles.
    *_ptrResultat = x + y;

    // Nous laissons au thread le soin de decider de la prochaine tache a
    // executer.
    return NULL;
  }

protected:

  /**
   * Calcule le terme dont le rang est fourni en argument de facon sequentielle.
   *
   * @param[in] rang - le rang du terme.
   * @return la valeur du terme.
   */
  static unsigned long sequentiel(const unsigned long& rang) {
    return rang < 2 ? rang : sequentiel(rang - 1) + sequentiel(rang - 2);
  }

protected:

  /**
   * Rang du terme a calculer. Cet attribut n'est pas declare constant pour
   * autorise le recyclage.
   */
  unsigned long _rang;

  /**
   * Rang a partir duquel le calcul se fait de facon sequentiel. Attention,
   * il s'agit ici d'un alias et non pas d'une copie : il faut donc que la
   * l'etat de la tache qui possede l'original soit maintenu dans la pile du
   * thread.
   */
  const unsigned long& _seuil;

  /**
   * Variable dans laquelle recopier le resultat. Meme remarque que pour
   * l'attribut precedent puisque nous avons ici un pointeur vers une donnee
   * locale à une autre tache. Cet attribut n'est pas declare constant pour
   * autoriser le recyclage.
   */
  unsigned long* _ptrResultat;

};

#endif
```

FIGURE 18 – Classe `TacheFibonacci` représentant le calcul parallèle d'un terme de la suite.

Les deux tâches filles sont respectivement lancées via les méthodes `spawn` et `spawn_and_wait_for_all`.

L'effet de cette dernière méthode est de lancer la tâche en argument puis d'attendre que la connectivité de la tâche mère retombe à la valeur 1. Cette connectivité est alors remise à zéro et la tâche mère peut reprendre son exécution.

Les instructions de lancement de tâches n'incrémentent pas la connectivité de la tâche mère, la philosophie de TBB étant de laisser au programmeur le soin de fixer explicitement cette valeur et donc, au final, de gérer lui-même la synchronisation de ses tâches.

Ainsi, lorsque la tâche mère lance ses deux tâches filles, sa connectivité devrait logiquement être fixée à 2 via la méthode `set_ref_count`. Cependant, à l'issue de l'exécution de l'une des tâches filles, cette connectivité serait alors automatiquement ramenée à la valeur 1, c'est à dire la valeur attendue par la méthode `spawn_and_wait_for_all` pour laisser la tâche mère reprendre son exécution.

Par conséquent, il est nécessaire d'augmenter la connectivité de la tâche mère pour qu'elle attende la terminaison de toutes ses tâches filles (et que ces dernières garde l'accès à l'environnement de leur tâche mère). C'est pour cette raison que nous fixons la connectivité de la tâche mère à la valeur 3 et non pas 2.

Lorsque la tâche mère reprend son exécution, il ne lui reste plus qu'à sommer les résultats de ses deux tâches filles. La méthode `execute` se termine en retournant la valeur spéciale `NULL` qui laisse au thread le soin de décider de sa prochaine tâche à exécuter.

Notons enfin que les attributs `_rang` et `_ptrResultat` sont volontairement laissés variables et non constants afin de permettre la mise en œuvre de la technique du recyclage de tâches pour les instances de classe `TacheFibonacci` ou de ses classes dérivées.

1.4.2 Tâche de continuation

Lorsque le thread qui exécute une tâche atteint la méthode `spaw_and_wait_for_all` ou la méthode `wait_for_all`, l'algorithme du *work-stealing* l'autorise à commencer l'exécution d'une autre tâche avant de reprendre celle en attente.

Pour ce faire, l'environnement de la tâche en attente doit être maintenu dans sa pile. Par conséquent, l'environnement de la nouvelle tâche qu'il exécute est placé au dessus de celui de la tâche en attente. Le risque est alors de voir la pile du thread déborder.

Pour limiter ce risque, le *scheduler* autorise le thread à voler une tâche à un autre thread uniquement si la profondeur de cette dernière est supérieure à celle de la tâche en attente. Cependant, cette restriction qui ne peut garantir à elle seule une absence de débordement de la pile, limite le nombre de tâches que le thread peut voler et par conséquent le parallélisme potentiel de l'application.

Pour ne pas brider ce parallélisme tout en limitant l'augmentation de la taille des piles, TBB propose la notion de tâche de continuation.

Lorsqu'une tâche mère doit lancer des tâches filles et attendre leurs résultats, elle instancie une tâche de continuation, lui fait lancer les tâches filles puis termine son exécution. Par conséquent, l'environnement de la tâche mère disparaît immédiatement de la pile du thread.

Pour que les tâches filles puissent continuer d'exploiter certaines données de leur tâche mère, ces données sont transférées dans la tâche de continuation. Cette dernière ne peut être lancée explicitement. Au contraire, son lancement est implicitement provoqué par la terminaison de sa dernière tâche fille. Par conséquent, son environnement, plus léger que celui de la tâche mère, n'est installé dans la pile du thread que le temps de son exécution. Cette dernière est obligatoirement brève puisque le rôle d'une tâche de continuation est de finaliser la calcul entrepris par la tâche mère qu'il l'a armée.

La figure 19 présente un exemple de mise en œuvre de la technique des tâches de continuation pour le calcul parallèle d'un terme de la suite de Fibonacci.

Dans cet exemple, nous commençons par définir une sous-classe `Continuation` qui représentera la tâche de continuation associée à la tâche mère de classe `TacheFibonacciII`.

Les attributs de cette sous-classe sont les emplacements de stockage des résultats produits respectivement par la tâche fille associée au sous-arbre gauche, celle associée au sous-arbre droit et enfin, le résultat final.

La sous-classe `Continuation`, qui dérive de la classe de base abstraite `tbb::task`, redéfinit sa méthode `execute` en y faisant sommer les résultats produits par les deux tâches filles.

La méthode `execute` de la tâche mère commence par instancier dynamiquement une tâche de continuation via une surcharge de l'opérateur `new` utilisant la méthode de classe `task::allocate_continuation`.

L'argument fourni au constructeur logique de la classe `Continuation` est l'emplacement de stockage du résultat final. Deux tâches filles sont alors dynamiquement instanciées par le biais de la tâche de continuation. Nous remarquons que les emplacement de stockage indiqués aux deux tâches filles sont tout simplement les attributs `_x` et `_y` de la tâche de continuation.

La connectivité de la tâche de continuation est ensuite fixée à la valeur 2 puis les deux tâches filles sont lancées via sa méthode `spawn`. À cet instant, la tâche mère termine son exécution tandis que la tâche de continuation attend que toutes ses filles aient terminé la leur pour commencer la sienne.

Le fait de fixer la connectivité de la tâche de continuation à la valeur 2 garantit que celle-ci tombera à zéro une fois la dernière tâche fille achevée. Par conséquent, la tâche de continuation sera implicitement insérée en tête du *ready deque* du thread qui a exécuté la dernière tache fille. C'est pour cette raison que nous parlons de lancement implicite pour les tâches de continuation.

1.4.3 *Scheduler bypass* et recyclage de tâche

Une tâche fille allouée dynamiquement via la surcharge de l'opérateur `new` et la méthode de classe `task::allocate_child`, est automatiquement désallouée à l'issue de l'exécution de la méthode `execute`.

Pour éviter l'*overhead* lié à l'allocation et la désallocation de tâches filles, TBB propose la possibilité de recycler une tâche, c'est à dire de ne pas la désallouer pour lui faire exécuter un nouveau calcul.

Une tâche peut être recyclée en tant que tâche fille en invoquant sa méthode `recycle_as_child_of` dont le seul et unique argument est la tâche mère. De même, une tâche peut être recyclée en tant que tâche de continuation en invoquant sa méthode `recycle_as_continuation`, cette dernière ne prenant pas d'argument. Cependant, dans tous les cas, une tâche ne peut être recyclée que si elle ne possède pas de tâches filles, c'est à dire que sa connectivité est nulle.

TBB propose également la possibilité de contourner le *scheduler* en désignant au thread qui exécute une tâche sa prochaine tâche à exécuter. Cette technique, appelée « *scheduler bypass* », permet d'éviter l'*overhead* lié à l'algorithme du *work-stealing*. Elle est mise en œuvre en faisant retourner par la méthode `execute` la prochaine tâche à exécuter et non plus la valeur spéciale `NULL`. Cependant, dans ce cas de figure, le programmeur prend en main l'équilibrage de charge de son application.

La classe `TacheFibonacciIII`, présentée en Figure 20, illustre la mise en œuvre conjointe des tâches de continuation, du recyclage de tâches et du *scheduler bypass*.

Dans cet exemple, la tâche mère qui arme une tâche de continuation, est recyclée en tant que tâche fille

```
06/05/2012                    TacheFibonacciII.hh                    1

#ifndef _TacheFibonacciII_hh
#define _TacheFibonacciII_hh

#include "TacheFibonacci.hh"

/**
 * @class TacheFibonacciII TacheFibonacciII.hh
 * @brief Tache de calcul d'un terme de la suite de Fibonacci en utilisant une
 *        methode recursive naive et le principe des taches de continuation.
 * @author Emmanuel.Cagniot - @c Emmanuel.Cagniot@ensicaen.fr
 * @date 3.5.2012
 *
 * Definition inline de la classe TacheFibonacciII representant une tache avec
 * continuation dediee au calcul d'un terme de la suite de Fibonacci.
 */
class TacheFibonacciII: public TacheFibonacci {
public:

  /**
   * Constructeur logique.
   *
   * @param[in] rang - la valeur de @ref TacheFibonacci::_rang.
   * @param[in] seuil - la valeur de @ref TacheFibonacci::_seuil.
   * @param[in] resultat - la valeur de @ref TacheFibonacci::_resulat.
   */
  TacheFibonacciII(const unsigned long& rang,
                   const unsigned long& seuil,
                   unsigned long& resultat):
    TacheFibonacci(rang, seuil, resultat) {
  }

public:

  tbb::task* execute() {

    // Nous sommes passes sous le seuil : nous poursuivons en sequentiel.
    if (_rang < _seuil) {
      *_ptrResultat = sequentiel(_rang);
      return NULL;
    }

    // Nous sommes au dessus du seuil : nous commencons par instancier une
    // tache de continuation puis poursuivons en lui faisant instancier deux
    // taches filles.
    Continuation& continuation =
      *new(allocate_continuation()) Continuation(_ptrResultat);
    TacheFibonacciII& gauche =
      *new(continuation.allocate_child()) TacheFibonacciII(_rang - 1,
                                                           _seuil,
                                                           continuation._x);
    TacheFibonacciII& droite =
      *new(continuation.allocate_child()) TacheFibonacciII(_rang - 2,
                                                           _seuil,
                                                           continuation._y);

    // Nous fixons la connectivite de la tache de continuation a la valeur deux
    // et nous lui faisons lancer ses deux taches filles.
    continuation.set_ref_count(2);
    continuation.spawn(gauche);
    continuation.spawn(droite);
```

```
06/05/2012                    TacheFibonacciII.hh                    2

    // Le thread decide de la prochaine tache a executer.
    return NULL;
  }
protected:
  /**
   * @class Continuation TacheFibonacciII.hh
   * @brief Tache de continuation pour le calcul parallele d'un terme de la
   *        suite de Fibonacci.
   * @author Emmanuel.Cagniot - @c Emmanuel.Cagniot@ensicaen.fr
   * @date 6.5.2012
   *
   * Definition inline de la sous-classe Continuation representant une tache de
   * continuation pour le calcul parallele d'un terme de la suite de Fibonacci.
   */
  class Continuation: public tbb::task {
  public:

    /**
     * Constructeur logique.
     *
     * @param[in,out] ptrResultat - la valeur de @ref ptrResultat.
     */
    Continuation(unsigned long* const ptrResultat):
      _ptrResultat(ptrResultat) {
    }

  public:

    tbb::task* execute() {
      *_ptrResultat = _x + _y;
      return NULL;
    }

  public:

    /**
     * Resultat du sous-arbre gauche.
     */
    unsigned long _x;

    /**
     * Resultat du sous-arbre droit.
     */
    unsigned long _y;

    /**
     * Emplacement de stockage du resultat.
     */
    unsigned long* const _ptrResultat;

  };
};

#endif
```

FIGURE 19 – Classe `TacheFibonacciII` représentant le calcul parallèle d'un terme de la suite avec mise en œuvre de la technique des tâches de continuation.

de cette dernière pour effectuer le calcul lié au sous-arbre gauche. La tâche de continuation lance alors explicitement l'une de ses tâches filles via sa méthode `spawn` tandis que la tâche recyclée est implicitement lancée puisque retournée en tant que résultat de la méthode `execute`.

1.4.4 Groupes de tâches

Les groupes de tâches (module `tbb/task_group.h`) permettent de simplifier l'utilisation des tâches lorsqu'il s'agit d'invoquer des fonctions. La figure 21 présente leur utilisation dans le cas d'une fonction récursive.

1.5 Instruction atomique

TBB propose les primitives de synchronisation classiques que sont les sémaphores. Plus précisément, ces mécanismes sont déclinés en plusieurs familles, par exemple celle des sémaphores récursifs.

L'utilisation des sémaphores est toujours délicate, le problème pour le programmeur étant de garantir l'absence de *deadlock* dans son application.

Lorsqu'une donnée d'un processus est susceptible d'être écrite par plusieurs threads, la stratégie courante consiste à faire protéger cette donnée par un sémaphore. L'utilisation de ces mécanismes à différents endroits dans une application importante est une source potentielle de *deadlock*.

Pour limiter ce risque, TBB propose la notion d'atomicité pour certaines instructions élémentaires telles que l'affectation. Une instruction élémentaire est dite atomique si son exécution ne peut être interrompue une fois commencée. Par conséquent, l'atomicité permet de réduire le nombre de sémaphores en circulation dans une application et donc les risques de *deadlock* inhérents.

La mise en œuvre de l'atomicité est réalisée via la classe générique `atomic` définie dans le module `tbb/atomic.h`. Cette classe est paramétrée par un type abstrait ne pouvant être instancié que par un type fondamental du langage. Les opérations qu'il est possible de réaliser sur une instance `x` de cette classe sont :

1. `... = x` qui retourne la valeur de `x` ;
2. `x = ...` qui modifie la valeur de `x` ;
3. `x.fetch_and_store(y)` qui effectue `x = y;` puis retourne la valeur initiale de `x` ;
4. `x.fetch_and_add(y)` qui effectue `x += y` puis retourne la valeur initiale de `x` ;
5. `x.compare_and_swap(y, z)` qui effectue `x = y` si la condition `x == z` est vérifiée. Cette méthode retourne systématiquement la valeur initiale de `x`.

1.6 Conteneur *thread-safe*

Les conteneurs de la bibliothèque standard ne sont pas *thread-safe*.

Par exemple, permettre à plusieurs threads de modifier simultanément plusieurs éléments différents d'une instance de classe `vector` ne pose aucun problème. À l'inverse, le problème se pose lorsque plusieurs threads sont susceptibles de modifier un même élément de ce cette instance. Pire, le conteneur est susceptible d'être corrompu lorsque l'on modifie sa structure, par exemple en y ajoutant des éléments.

Pour répondre à ce problème, TBB met à disposition du programmeur des conteneurs *thread-safe* tels que, par exemple, les `concurent_vector`. Ce type de conteneur présente un surcoût important par rapport à son homologue de bibliothèque standard. Par conséquent, son utilisation doit être justifiée et ne pas être systématique.

```
08/05/2012                    TacheFibonacciIII.hh                    1

#ifndef _TacheFibonacciIII_hh
#define _TacheFibonacciIII_hh

#include "TacheFibonacciII.hh"

/**
 * @class TacheFibonacciIII TacheFibonacciIII.hh
 * @brief Tache de calcul d'un terme de la suite de Fibonacci en utilisant une
 *    methode recursive naive, le principe des taches de continuation, celui
 *    du scheduler bypass et enfin, celui du recyclage de taches.
 * @author Emmanuel.Cagniot - @c Emmanuel.Cagniot@ensicaen.fr
 * @date 8.5.2012
 *
 * Definition inline de la classe TacheFibonacciIII representant une tache avec
 * continuation dediee au calcul d'un terme de la suite de Fibonacci.
 */
class TacheFibonacciIII: public TacheFibonacciII {
public:
  /**
   * Constructeur logique.
   *
   * @param[in] rang - la valeur de @ref TacheFibonacci::_rang.
   * @param[in] seuil - la valeur de @ref TacheFibonacci::_seuil.
   * @param[in] resultat - la valeur de @ref TacheFibonacci::_resulat.
   */
  TacheFibonacciIII(const unsigned long& rang,
                    const unsigned long& seuil,
                    unsigned long& resultat):
    TacheFibonacciII(rang, seuil, resultat) {
  }

public:

  tbb::task* execute() {

    // Nous sommes passes sous le seuil : nous poursuivons en sequentiel.
    if (_rang < _seuil) {
      *_ptrResultat = sequentiel(_rang);
      return NULL;
    }

    // Nous sommes au dessus du seuil : nous commencons par instancier une
    // tache de continuation puis poursuivons en lui faisant instancier une
    // seule et unique tache fille.
    Continuation& continuation =
      *new(allocate_continuation()) Continuation(_ptrResultat);
    TacheFibonacciII& gauche =
      *new(continuation.allocate_child()) TacheFibonacciII(_rang - 1,
                                                           _seuil,
                                                           continuation._x);


    // Nous recyclons cette tache (this) en tant que fille de la tache de
    // continuation pour le sous-arbre droit.
    recycle_as_child_of(continuation);
    _rang -= 2;
    _ptrResultat = &continuation._y;

    // Nous fixons la connectivite de la tache de continuation a la valeur deux
    // et nous lui faisons lancer une seule tache fille.
```

```
08/05/2012                    TacheFibonacciII.hh                     2

    continuation.set_ref_count(2);
    continuation.spawn(gauche);

    // Nous indiquons a scheduler la prochaine tache a executer (principe du
    // scheduler bypass).
    return this;

  }

};

#endif
```

FIGURE 20 – Classe `TacheFibonacciIII` illustrant la mise en œuvre conjointe des tâches de continuation, du recyclage de tâches et du scheduler bypass.

```
long
fibonacci(int n) {
  if (n < 2) {
    return n;
  }
  long a, b;
  tbb::task_group g;
  g.run([&]{ a = fibonacci(n - 1); });
  g.run([&]{ b = fibonacci(n - 2); });
  g.wait();
  return a + b;
}
```

FIGURE 21 – Parallélisation d'une fonction récursive à l'aide des groupes de tâches.

Pour limiter le surcoût dû à l'utilisation de sémaphores, TBB permet de verrouiller un groupe d'élément contigus et non pas d'associer un verrou à chacun de ces éléments.

À tout conteneur de la bibliothèque standard est associé un allocateur de mémoire. En contexte multithreadé, l'allocation dynamique de mémoire peut rapidement devenir un goulet d'étranglement si l'on utilise les allocateurs classiques fournis par le système d'exploitation. Par conséquent, TBB met à disposition deux allocateurs de mémoire génériques dédiés aux applications multi-threadées.

Le premier allocateur, `scalable_allocator`, permet à plusieurs threads d'allouer/désallouer simultanément des blocs de mémoire.

Le second, `cache_aligned_allocator` permet d'allouer des blocs dont la taille est un multiple de la taille d'une ligne du cache partagé de niveau 2. Par conséquent, deux objets alloués avec cet allocateur sont assurés de ne pas partager la même ligne de cache. Cet allocateur supprime le phénomène du *false sharing* mais induit un gaspillage de mémoire pouvant conduire à une augmentation du taux de défauts de cache. Par conséquent, il doit être utilisé avec précaution.