École Nationale Supérieure
d'Ingénieurs de Caen

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

TP $n^o 3$

Niveau	**$2^{\text{ème}}$ année**
Parcours	**Informatique**
Unité d'enseignement	**2I2AC3 - Architectures parallèles**
Responsables	**Emmanuel Cagniot** **Emmanuel.Cagniot@ensicaen.fr** **Hugo Descoubes** **Hugo.Descoubes@ensicaen.fr**

1 Threading Building Blocks (Intel)

Threading Building Blocks (TBB) est une bibliothèque template 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 de ces tâches aux différents threads disponibles sur la plateforme est réalisée par la bibliothèque elle-même, et plus précisément son scheduler qui implémente le mécanisme du work-stealing emprunté au langage de programmation parallèle Cilk. 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.

Comme la bibliothèque standard C++, TBB utilise abondamment la notion de template afin de tirer avantage du low-overhead polymorphism. Cette bibliothèque 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 classique ainsi qu'une mécanisme garantissant l'atomicité d'une instruction ;
6. des conteneurs thread-safe et des allocateurs de mémoire en contexte multithreadé.

Intel propose sa bibliothèque en version commerciale et Open Source. Les deux versions ne diffèrent que par le support apporté au programmeur dans la version commerciale. La dernière version de TBB, la 4.0, est sortie courant Septembre 2011.

Tâches 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 `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()`.

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.

Les tâches peuvent être manipulées de deux façons dans TBB :

1. explicitement, en écrivant des classes concrètes dérivant de la classe abstraite `task` ;
2. implicitement, en exploitant les algorithmes (`parallel_for`, `parallel_invoke`, etc.), les pipelines logiciels ou bien encore les CFG (Control Flow Graph).

Le scheduler de TBB implémente un algorithme appelé randomized work-stealing. 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 (configuration par défaut) 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 d'ordonnancement 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.

2 Pipeline logiciel

L'objectif de ce TP est l'implémentation d'un parallel design pattern relativement méconnu ou négligé : `Pipeline`. Conséquence software du principe de fonctionnement des processeurs vectoriels, ce pattern présente la particularité de pouvoir s'appliquer à de nombreux problèmes tout en s'implémentant de façon simple. Nous allons donc utiliser ce pattern dans le cadre d'un algorithme de tri de mauvaise réputation : le bubble sort. Cette implémentation sera basée sur la bibliothèque C++ Threading Building Blocks (TBB) développée par Intel et qui met à disposition des programmeurs des briques de base (algorithmes, conteneurs thread-safe, etc.) permettant de réaliser simplement et rapidement des applications multithreadées efficaces. Au passage, nous allons continuer à nous familiariser avec les techniques du template programming au travers de la réalisation d'algorithmes semblables à ceux de la bibliothèque standard C++.

Pipeline matériel synchrone

Certaines instructions telles que les additions et les multiplications entières et/ou pseudo réelles sont présentes en grandes quantités dans les codes. De fait, elles sont traitées par des opérateurs câblés dédiés directement intégrés au processeur. Sous réserve d'accès simultanés à la mémoire (mémoires entrelacées composées de plusieurs modules), ces opérateurs utilisent un principe permettant d'accélérer l'exécution d'une séquence $\{i_n\}$ contenant n occurrences de la même instruction : le pipeline.

Le principe du pipeline consiste à :
— segmenter l'instruction en étapes si possible de même durée ;
— connecter les sorties de l'étape n aux entrées de l'étape $n+1$;
— synchroniser le passage des données entre étapes successives.

Sa mise en œuvre matérielle est réalisée comme suit :
— à chaque étape est associé un étage disposant de registres d'entrée et de sortie ;
— les étages sont synchronisés par une horloge ;
— l'ensemble des étages constitue le pipeline.

Considérons le cas concret d'une addition pseudo réelle. Cette instruction peut être segmentée en trois étapes consécutives :
— comparaison des exposants et alignement de la mantisse du plus petit nombre sur celle du plus grand (dénormalisation) ;
— addition des mantisses en virgule fixe ;
— normalisation du résultat (norme IEEE 754).

La Figure 1 présente l'opérateur pipeliné correspondant.

Supposons que notre additionneur soit dans une configuration idéale, c'est à dire que chacun de ses étages réclame le même temps (durée) de passage τ. Nous pouvons alors régler la période γ de l'horloge en choisissant $\gamma = \tau$.

Le temps d'exécution scalaire, noté $\tau_{scalaire}$, représente le temps d'exécution d'une séquence de n additions si notre opérateur se comporte selon le modèle d'exécution scalaire de Von Neuman, c'est à dire qu'il attend le résultat de l'addition i_p avant de lancer l'addition suivante i_{p+1}. Dans ce cas, nous avons tout simplement le résultat (désespérant) :

$$\tau_{scalaire} = n \times 3 \times \tau. \tag{2.1}$$

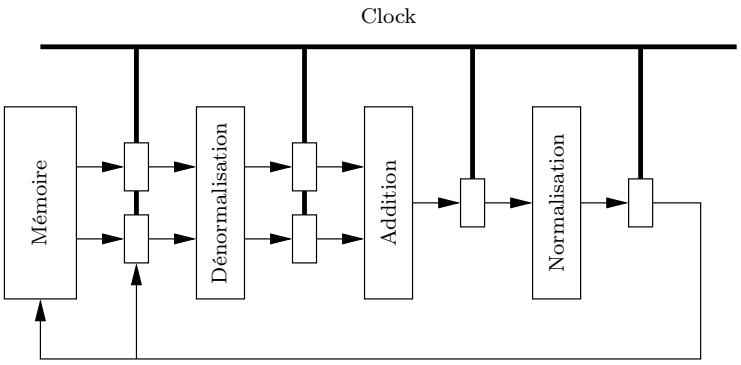

FIGURE 1 – Additionneur en virgule flottante.

Le temps de latence, noté $\tau_{latence}$, représente le temps nécessaire à l'obtention du résultat de la première addition i_0 si notre opérateur est exploité en mode pipeline, c'est à dire qu'il exécute simultanément (régime de croisière) :

— une dénormalisation sur l'addition i_{p+2} ;
— une addition en virgule fixe sur l'addition i_{p+1} ;
— une normalisation sur l'addition i_p.

Dans notre cas, ce temps de latence représente le temps nécessaire à la traversée des trois étages de l'additionneur, c'est à dire :

$$\tau_{latence} = 3 \times \tau. \tag{2.2}$$

Notre additionneur fonctionnant à présent en mode pipeline, le premier résultat est obtenu trois tops d'horloge plus tard que l'injection de l'addition i_0. Cependant, contrairement au mode de fonctionnement scalaire, les résultats suivants arrivent juste derrière à raison d'un nouveau résultat par top d'horloge, le pipeline ayant atteint son régime de croisière. Par conséquent, le temps d'exécution d'une séquence de n additions est maintenant donné par :

$$\tau_{pipeline} = \tau_{latence} + (n-1) \times \tau = (3+n-1) \times \tau. \tag{2.3}$$

Afin de mesurer le gain de rapidité procuré par le mode de fonctionnement pipeline par rapport au mode scalaire, nous définissons le facteur d'accélération (speed up) comme suit :

$$s_3^n = \frac{\tau_{scalaire}}{\tau_{pipeline}} = \frac{3 \times n}{3+n-1}. \tag{2.4}$$

Faisons à présent tendre le nombre d'additions n vers l'infini, c'est à dire que la séquence d'additions devient très longue ; il vient le résultat (magique cette fois) :

$$s_3^\infty = lim_{n\to\infty}\Big(\frac{3}{\frac{3}{n}+1-\frac{1}{n}}\Big) = 3, \tag{2.5}$$

qui signifie tout simplement que dans le cas idéal, le mode de fonctionnement pipeline de notre additionneur est trois fois plus rapide que le mode de fonctionnement scalaire de VON NEUMAN. Par conséquent, un pipeline sera d'autant plus rapide que son nombre d'étages est élevé et que les temps de traversée sont relativement homogènes (configuration proche du cas idéal).

Parallel design pattern `Pipeline`

De nombreux problèmes se caractérisent par une chaîne de traitements successifs au travers de laquelle passent des données. L'exemple le plus connu est la commande `pipe` (symbole `'|'`) du système d'exploitation UNIX. Les traitements pouvant être nombreux, une parallélisation basée sur une boucle de type `forall` sur les données (data parallelism, le premier type de parallélisation venant généralement à l'esprit) impliquerait l'écriture d'un code relativement complexe et donc difficilement maintenable. Pourvu

que chaque traitement puisse être réalisé indépendamment des autres et que ces traitements ne mettent à jour aucune structure de données partagée, alors le problème considéré peut être parallélisé beaucoup plus efficacement et simplement via le parallel design pattern `Pipeline`.

En règle générale, un parallel design pattern ne peut être complètement modélisé en UML (voire pas du tout) et ce pour plusieurs raisons :

— les compositions ou aggrégations sont parfois insuffisantes pour modéliser les dépendances entre classes ;
— les communications entre objets ne se résument pas à de simples invocations de méthodes ;
— etc.

Cependant, dans le cas du pattern `Pipeline`, il existe une forte proximité avec le behavioral design pattern `Chain of Responsability` dont la Figure 2 présente le diagramme de classes.

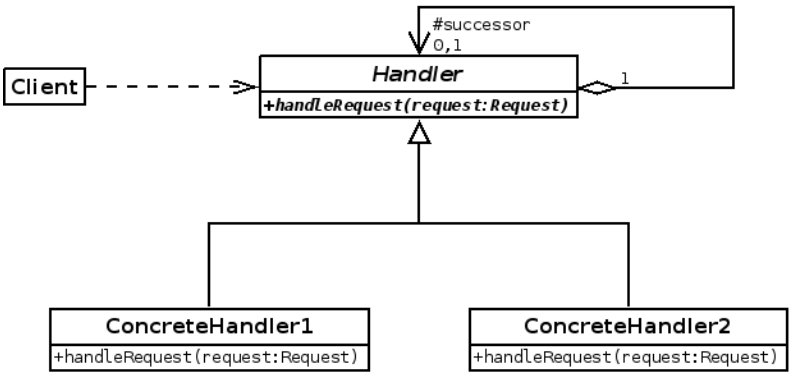

FIGURE 2 – Diagramme de classes du behavioral design pattern `Chain Of Responsability`.

En conception objet, le behavioral design pattern `Chain of Responsability` permet à un client d'envoyer une commande (requête) sans nommer explicitement l'objet qui la traitera. Plus précisément, le client émet une requête à l'élément placé en tête d'une liste constituée d'objets susceptibles de pouvoir la satisfaire. Si ce premier élément peut traiter la requête alors il le fait. Dans le cas contraire, il la transmet à l'élément suivant dans la liste et ainsi de suite. Ce pattern permet de réduire la dépendance entre le client et l'objet qui traitera la requête puisque ce dernier n'est jamais explicitement nommé.

Si nous transposons le pattern `Chain of Responsability` dans le monde du pipeline, les objets de la liste représentent les étages tandis que les requêtes représentent les données qui circulent au travers de ces étages. Cependant, la différence réside dans le fait qu'une requête, qui n'est pas émise par un client mais par le premier étage du pipeline, subit des modifications au fil des étages.

Toujours par analogie avec les design patterns exploités en conception objet, `Pipeline` serait classé dans la famille des structural design patterns puisqu'il permet d'obtenir un traitement complexe par composition de traitements élémentaires. Par conséquent, les bénéfices pour le programmeur sont évidents, ce dernier n'ayant besoin que de :

— décrire chaque traitement élémentaire indépendamment des autres (mise au point et debugging simplifiés à l'extrême) ;
— composer les traitements élémentaires pour obtenir le pipeline réalisant le traitement total ;
— laisser faire le pipeline pour l'aspect parallélisme.

En résumé, ce programmeur s'occupe de la partie séquentielle (le fonctionnement des étages) de son code et n'aborde que peu ou prou la partie parallèle inhérente au fonctionnement du pipeline.

Implémentation proposée par TBB

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

Le programmeur commence par définir une classe pour chaque type d'étage, cette classe concrète dérivant de la classe abstraite `tbb::filter`. Le constructeur logique de cette dernière, de droit d'accès

`public`, nécessite un argument de type `tbb::filter::mode`. Ce type énuméré (faiblement typé) propose trois valeurs décrivant le mode de fonctionnement de l'étage :

- `serial_in_order :` qui désigne un étage séquentiel, c'est à dire ne traitant qu'un jeton (une donnée) 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 susceptible de traiter plusieurs jetons simultanément. Ce type d'étage est donc susceptible de faire parvenir à l'étage suivant des jetons ne respectant pas un ordre de traitement pré-établi. Si ce dernier étage est de type `serial_in_order` alors il rétablira cet ordre.

La classe de base `filter` impose de redéfinir la méthode abstraite `void* operator()(void* item)`. L'argument de cette méthode représente l'adresse en mémoire du jeton transmis par l'étage précédent dans le pipeline. Son résultat représente l'adresse en mémoire du jeton à transmettre à l'étage suivant (généralement la même que celle de son argument). La valeur spéciale `NULL` traduit l'épuisement du flot de jetons. L'argument `item` n'est pas utilisé dans le cas d'un étage d'entrée. En règle générale, ce dernier alloue dynamiquement un jeton sur le tas tandis que l'étage de sortie le désalloue.

La création d'un pipeline vide se fait en instanciant la classe `pipeline`, celle-ci étant dotée d'un constructeur par défaut de droit d'accès `public`. Les étages sont ensuite instanciés puis ajoutés en queue via la méthode `add_filter` dont l'argument est de type `filter&` (alias variable pour un étage dont la classe dérive de `filter`).

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 »si un étage séquentiel, relativement lent, succède à un étage parallèle trop rapide. La configuration optimale pour une machine multicœurs est d'un seul et unique thread par cœur (configuration par défaut du scheduler de TBB). Pour un pipeline, la configuration optimale ne doit pas faire circuler plus de jetons que de threads disponibles et doit tenir compte des disparités en temps de calcul entre les étages séquentiels et parallèles.

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.

3 Exercice

Le principe de l'algorithme du bubble sort consiste à boucler tant qu'au moins un élément e_i du conteneur à trier a dû être permuté avec son successeur e_{i+1}, lors de l'itération précédente, afin de satisfaire à une relation d'ordre total. Son pseudo code est le suivant (pour la relation d'ordre total strictement inférieur à) :

La complexité au pire est $\mathcal{O}(n^2)$ si n représente le nombre d'éléments du conteneur à trier. Elle est atteinte lorsque les éléments du conteneur sont déjà triés selon la relation d'ordre total inverse de celle souhaitée. Cette complexité au pire cause la mauvaise réputation du bubble sort. Cependant, lorsque le nombre d'éléments à trier est très faible, cet algorithme est tout à faire pertinent du double fait de sa simplicité et de sa consommation mémoire nulle (tri in place).

L'archive `tp3.tar.gz`, accessible via la page web décrivant l'unité d'enseignement, contient un squelette incomplet de l'application à réaliser. Sa structure est la suivante :

- `src/include/Metrics.hpp :` fichier en-tête contenant la déclaration d'une classe `Metrics` permettant de calculer des facteurs d'accélération et d'efficacité ;
- `src/include/bubbleSort.hpp :` fichier en-tête contenant le squelette incomplet de la version séquentielle du bubble sort ;

Procedure 1 `bubbleSort`

```
Input : vect(1 : n)
Ouput : vect(1 : n)
  continue := true
  while continue do
    continue := false
    for i := 1 to n - 1 do
      if Vect(i) >= Vect(i + 1) then
        tmp := Vect(i)
        Vect(i) := Vect(i + 1)
        Vect(i + 1) := tmp
        continue := true
      end if
    end for
  end while
```

`src/include/pipelinedBubbleSort.hpp :` fichier en-tête contenant le squelette incomplet de la version pipelinée du bubble sort ;

`src/Metrics.cpp :` définition de la classe `Metrics` ;

`src/testPipelinedBubbleSort.cpp :` programme de test permettant de comparer les versions séquentielle et parallèles du bubble sort ;

`CMakeLists.txt :` script permettant de générer le makefile de l'application via l'utilitaire `cmake` ;

`Lisezmoi.txt :` fichier texte décrivant la procédure de génération du makefile et celle de la configuration du fichier `CMakeCache.txt` produit par `cmake`.

3.1 Question

Complétez la définition de l'algorithme `bubbleSort` qui implémente la version séquentielle du bubble sort et dont le patron est `template< typename RandomAccessIterator, typename Compare >`.

Le premier paramètre du patron représente le type d'itérateur attendu. Dans notre cas, il s'agit d'un `RandomAccessIterator`, c'est à dire un itérateur bidirectionnel permettant d'écrire des expressions telles que `it ++`, `-- it`, `it - 5`, `it + 3`, etc. La caractéristique principale de ces itérateurs est de garantir un accès en temps constant à n'importe quel élément du conteneur séquentiel qu'ils balaient dans les deux sens (typiquement des tableaux classiques, des tableaux génériques de classe `array` introduits par la norme 2011 ou bien encore des tableaux dynamiques de classe `vector`).

Le second paramètre du patron représente la relation d'ordre total à utiliser. Il s'agit d'une classe générique paramétrée par un type abstrait `T` et qui surcharge l'opérateur fonction sous la forme `bool operator()(const T left, const T right) const` ou bien encore `bool operator()(const T& left, const T& right) const`.

L'algorithme `swap`, défini dans le module `algorithm` de la bibliothèque standard, sera utilisé pour permuter deux éléments. Vous pourrez définir des variables ou des constantes (également des alias ou des pointeurs) du même type que ceux du conteneur à trier sans explicitement mentionner ce type via le mot-clé `auto` redéfinit par la norme C++ 2011 pour devenir le symbole de l'inférence de type (par exemple : `auto courant = *first` pour une variable ou bien encore `auto& courant = *first` pour un alias avec droit d'écriture).

3.2 Question

Le bubble sort possède de nombreuses variantes. Les plus intéressantes sont des versions parallèles implémentées sous forme matérielle : les réseaux de tris (réseaux de Batcher, réseaux bitoniques, etc.) qui constituent un pan entier du parallélisme. Dans notre cas, la parallélisation que nous nous proposons de réaliser est assez simple. Sachant que la complexité au pire de l'opération consistant à fusionner deux listes triées de taille $n/2$ est $\mathcal{O}(n)$ (complexité linéaire), nous allons tout simplement tronçonner le

conteneur à trier en fragments successifs de très petite taille (par exemple 8 éléments maximum), trier ces fragments à l'aide de la version séquentielle du bubble sort puis les fusionner in fine. Afin de limiter au maximum le risque du false sharing inhérent à toute application multithreadée, les fragments seront des copies physiques de portions d'éléments du conteneur à trier (notre tri ne sera pas in place, contrairement à la version séquentielle). En termes de pipeline, notre application nécessitera donc :

1. un étage d'entrée dont le rôle est de tronçonner le conteneur à trier. Les fragments seront de classe `vector`. Le mode de fonctionnement de cet étage sera `serial_out_of_order` puisqu'un seul et unique thread peut tronçonner le conteneur à trier mais aucun ordre particulier n'est requis lors de la fusion des fragments ;
2. un étage intermédiaire chargé de trier les fragments envoyés par l'étage d'entrée. Le mode de fonctionnement de cet étage sera `parallel` puisque plusieurs fragments peuvent être triés simultanément ;
3. un étage de sortie chargé de fusionner les fragments envoyés par l'étage de tri avec la partie déjà triée du conteneur d'origine. Comme l'étage d'entrée, le mode de fonctionnement de cet étage sera `serial_out_of_order` puisqu'un seul et unique thread peut écrire le conteneur à trier. La fusion sera réalisée via l'algorithme `inplace_merge` défini dans le module `algorithm` de la bibliothèque standard. L'étage d'entrée lit le conteneur à trier tandis que l'étage de sortie l'écrit. Cependant, il n'y a aucun risque de recouvrement puisque l'étage de sortie n'écrit que la partie déjà lue par l'étage d'entrée. Notons toutefois que comme deux threads vont travailler simultanément sur le même conteneur, l'un en lecture et l'autre en écriture, il existe un risque (faible mais il existe) de false sharing. Pour l'écarter définitivement, nous pourrions fusionner les fragments dans un conteneur annexe et local au dernier étage puis effectuer la recopie une fois le tri terminé. Cependant, le but de ce TP n'est pas d'écrire une version sur-optimisée mais simplement une version pipeline de l'algorithme. Nous allons donc nous en contenter.

Complétez la définition de l'algorithme `pipelinedBubbleSort` qui implémente cette architecture et dont le patron est `template< typename RandomAccessIterator, typename Compare >`.

La norme C++ 2011 autorise la définition inline de classes à l'intérieur de la définition d'une fonction ou d'une méthode (pourvu que ces classes ne soient pas génériques et qu'elles ne définissent aucun attribut de classe). Vous utiliserez donc cette commodité (aucune classe visible de fait pour l'utilisateur final).