

École Nationale Supérieure d'Ingénieurs de CAEN

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

$TP n^o 4$

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

Problème

Le bubble sort est un algorithme de tri difficile à paralléliser car l'élément e_j doit être comparé avec son prédécesseur e_{j-1} qui, lui-même, dépend de son prédécesseur e_{j-2} et ainsi de suite.

L'algorithme odd-even sort, présenté ci-dessous, est une variante du bubble sort basée sur une alternance d'étapes « paires » et « impaires » :

- aux étapes paires, les éléments de rang pair e_{2j} sont comparés avec leurs successeurs de rang impair e_{2j+1} ;
- aux étapes impaires, les éléments de rangs impairs e_{2j+1} sont comparés avec leurs successeurs de rang pair e_{2j+2} .

Procedure 1 odd-even sort

```
Input: vect(1:n)
Ouput : vect(1 : n)
  continue(0:1) := (true, true)
  shift := 0
  while continue(0) or continue(1) do
    continue(shift) := false
    for i := 1 + shift to n - 1 step 2 do
      if Vect(i) >= Vect(i + 1) then
         tmp := Vect(i)
         Vect(i) := Vect(i+1)
         Vect(i + 1) := tmp
         continue(shift) := true
      end if
    end for
    shift := 1 - shift
  end while
```

La complexité du odd-even sort est strictement identique à celle du bubble sort : $\mathcal{O}(n^2)$ si n représente le nombre d'éléments à trier. En effet, nous avons $\lceil \frac{n}{2} \rceil$ comparaisons par étapes (paires ou impaires) et 2n étapes sont nécessaires dans le pire des cas, c'est à dire lorsque les éléments sont ordonnés selon la relation d'ordre inverse. Cependant, l'intérêt du odd-even sort par rapport au bubble sort réside dans la possibilité d'effectuer simultanément toutes les comparaisons paire-impaire (e_{2j}, e_{2j+1}) puis impaire-paire (e_{2j+1}, e_{2j+2}) , ce qui ouvre la voie à une implémentation parallèle.

La figure 1 présente la définition de la fonction générique oddEvenSort implémentant la forme séquentielle de notre algorithme. Le premier paramètre de son 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 à utiliser. Il s'agit de n'importe quelle instance dont la classe surcharge l'opérateur fonction sous l'une des deux formes suivantes :

bool operator()(const T left, const T right) const

ou bien:

bool operator()(const T& left, const T& right) const

le type T étant le type des éléments du conteneur.

L'objectif de ce TP est une implémentation OpenMP efficace de la fonction générique oddEvenSort de la figure 1. L'archive tp4.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/oddEvenSort.hpp : fichier en-tête contenant la fonction générique oddEvenSort de la figure 1;
- src/include/ompOddEvenSortV1.hpp : fichier en-tête contenant le squelette incomplet de la première version parallèle de notre algorithme;
- $src/include/merge_n.hpp$: fichier en-tête contenant la fonction générique $merge_n$ permettant de fusionner les éléments de deux conteneurs triés selon la même relation d'ordre tout en n'en conservant que les n premiers;
- src/include/ompOddEvenSortV2.hpp : fichier en-tête contenant le squelette incomplet de la seconde version parallèle de notre algorithme;
- src/Metrics.cpp : définition de la classe Metrics;
- src/testOddEvenSort.cpp : programme de test permettant de comparer les versions séquentielle et
 parallèles de notre algorithme;
- 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.

1 Exercice

Nous commençons par paralléliser directement la fonction oddEvenSort de la figure 1 à l'aide des mécanismes proposés par OpenMP sans rien changer à son algorithme. L'effort porte ici sur la boucle interne for (lignes 29-44 de la figure 1), celle-ci étant évidemment parallélisable puisqu'elle ne présente aucune dépendance d'une itération à une autre. Par conséquent, nous pouvons répartir ces itérations sur les différents threads disponibles. La principale difficulté consiste à choisir la clause de répartition des

```
template < typename Random AccessIterator, typename Compare >
   void oddEvenSort(const RandomAccessIterator& first,
2
                     const RandomAccessIterator& last,
3
                     const Compare& comp) {
4
5
     // Si le nombre d'éléments à trier est inférieur à 1 alors nous avons
6
     // terminé.
     if (last - first < 1) {
8
       return;
9
     } // if
10
11
     // Flags indiquant si un échange entre deux éléments consécutifs s'est
12
     // produit lors de la dernière itération de la boucle extérieure.
13
     bool echange [2] = { true, true };
14
15
     // Décalage permettant d'alterner les périodes pair-impair puis
16
     // impair-pair.
17
     int shift = 0;
18
19
     // Boucle extérieure qui ne s'arrête que lorsque plus aucun échange
20
     // d'éléments n'est effectué à l'itération précédente.
21
     while (echange [0] || echange [1]) {
22
23
       // Pour l'instant, aucun échange entre deux éléments n'a encore été
24
       // effectué pour cette nouvelle itération de la boucle.
25
       echange [shift] = false;
26
27
       // Balayage des éléments de rang pair ou impair selon la période.
28
       for (RandomAccessIterator it = first + shift; it < last - 1; it += 2) {
29
         // Obtention de l'élément courant et de son successeur. Le mot-clé
31
         // auto, introduit par la norme 2011, permet de ne pas mentionner
32
         // le type des éléments.
33
         auto& courant = *it;
34
         auto\& suivant = *(it + 1);
35
36
         // Les éléments ne sont pas correctement ordonnés : il faut procéder
37
         // à un échange.
         if (! comp(courant, suivant)) {
39
           std::swap(courant, suivant);
40
           echange[shift] = true;
41
         } // if
43
       } // for
44
45
         // Mise à jour du décalage pour la période suivante.
46
       shift = 1 - shift;
47
48
     } // while
49
50
    // oddEvenSort
51
```

FIGURE 1 – Définition de la fonction générique oddEvenSort.

itérations la plus adéquate.

À priori, le corps de la boucle for présente des temps de calcul hétérogènes puisque :

- les éléments et leur successeurs qui ne respectent pas la relation d'ordre doivent être permutés via l'algorithme swap de la bibliothèque standard;
- les éléments déjà ordonnés ne font l'objet d'aucun traitement particulier.

La figure 2 présente la définition de la fonction générique swap de la bibliothèque standard dans sa norme 1998 (dans sa nouvelle norme 2011, elle fait usage de la *move semantic*, caractéristique sans importance dans notre cas). Comme son nom l'indique, cet algorithme procède à l'échange de deux éléments via une variable temporaire du même type.

```
template < typename T >
void swap(T& a, T& b) {
    T c(a);
    a = b;
    b = c;
}
```

FIGURE 2 – Algorithme swap de la bibliothèque standard (norme 1998).

1.1 Question

En observant le code de l'algorithme swap, peut-on affirmer dès maintenant que le corps de notre boucle for présente des temps de calcul fortement hétérogènes ou bien au contraire faiblement hétérogènes (quasi homogènes)?

1.2 Question

Déduisez de la question précédente la seule et unique clause de répartition des itérations qu'il soit possible d'utiliser dans notre cas.

1.3 Question

Écrivez une première version parallèle ompOddEventSortV1 en ne parallélisant que la boucle interne for qui travaille sur la variable partagée echange (les directives de synchronisation critical ou atomic ne sont pas autorisées car inefficaces dans notre cas).

Placer une boucle parallèle for à l'intérieur d'une boucle séquentielle (ici while) est déconseillé car vous êtes alors à la merci de votre compilateur (OpenMP n'est pas une bibliothèque mais un standard) :

- certains compilateurs ne créeront qu'une seule fois les threads et les utiliserons pendant toute la durée d'exécution de la boucle séquentielle;
- d'autres les créeront puis les détruiront à chaque itération de cette boucle d'où un *overhead* catastrophique.

La solution à ce problème consiste à maximiser les régions parallèles, c'est à dire leur faire englober la boucle séquentielle afin de garantir que les threads ne seront pas créés puis détruits à chaque itération de cette dernière. La contrepartie est une écriture beaucoup plus délicate : en effet, dans notre cas, le programmeur doit garantir qu'un thread n'est pas en train de modifier la variable partagée echange alors que les autres sont seulement en train d'évaluer la condition de continuation de leur boucle while.

1.4 Question

Écrivez une seconde version parallèle de la fonction ompOddEvenSortV1 tenant compte de la remarque précédente.

1.5 Question

Expliquez les performances médiocres de cette première implémentation de l'algorithme odd-even sort alors qu'à priori ce dernier est hautement parallèle.

2 Exercice

Le parallel block odd-even transposition sort est un algorithme conçu pour les architectures parallèles à mémoire distribué et leur modèle de programmation à parallélisme de processus communicants (message passing). Son principe consiste à :

- 1. découper le conteneur à trier en blocs d'éléments contigus si possible de même taille (équilibrage de charge); il y a autant de blocs que de processus disponibles;
- 2. affecter un bloc à chaque processus;
- 3. faire en sorte que chaque processus trie son bloc via la version séquentielle du odd-even sort;
- 4. alterner des étapes « paires » et « impaires » telles que :
 - (a) aux étapes paires, les processus de rang pair p_{2j} échangent leur bloc avec leurs successeurs de rang impair p_{2j+1} (s'ils existent). Chaque processus fusionne le bloc reçu avec le sien : les processus de rang pair conservent les éléments les plus petits tandis que les processus de rang impair conservent les plus grands (au sens de la relation d'ordre utilisée);
 - (b) aux étapes impaires, les processus de rang impair p_{2j+1} échangent leur bloc avec leurs successeurs de rang pair p_{2j+2} (s'ils existent) et tous accomplissent le travail décrit ci-dessus.

Il faut donc 2p étapes (p représentant le nombre de processus utilisés) pour trier le conteneur dans le pire des cas.

L'intérêt d'un tel algorithme dans le cas d'une architecture parallèle à mémoire partagée et son modèle de programmation à parallélisme de processus légers (ou threads) n'est pas évident à priori. Pour le saisir, il suffit de comparer les performances obtenues via la parallélisation directe décrite en première partie de cet énoncé, en faisant varier à la fois la taille du conteneur à trier et celle de ses éléments (entiers, flottants, structures, etc.) : ces performances peuvent varier fortement. Les raisons à cela peuvent être multiples :

- le conteneur est trop important et doit être maintenu dans le cache L2;
- le compilateur ne parvient pas à déterminer la bonne clause de répartition des itérations de la boucle for parallèle;
- il parvient à déterminer la bonne clause de répartition de la boucle for parallèle mais pas le bon nombre d'itérations consécutives à faire exécuter par chaque thread (apparition du false sharing);
- etc.

Si le compilateur ne peut pas tout inférer à partir du seul code source alors c'est au programmeur de prendre la main en gérant lui-même la localité des données. L'idée est ici de forcer les threads à calculer dans leur mémoire privée (généralement dans leur cache L1 si la taille d'un bloc le permet) et non pas en mémoire partagée. Par conséquent, le conteneur à trier doit être fractionné en blocs indépendants alloués dans la mémoire locale des threads, ce qui explique le recours à un algorithme dédié aux architectures à mémoire distribuée. Une fois l'algorithme terminé, ces blocs sont recopiés dans le conteneur d'origine qui lui, se trouve en mémoire partagée.

Le problème auquel nous sommes confrontés est simple : ce type de programmation n'est à priori pas prévu par OpenMP; il existe bien une directive copyin permettant à un thread de diffuser le contenu d'une donnée locale aux autres thread mais :

- le thread qui diffuse est obligatoirement le maître;
- la donnée ne peut être que scalaire (et pas tableau);
- elle doit être définie en tant qu'attribut de classe;
- etc.

Cependant, ce n'est pas vraiment une difficulté. Pour illustrer notre propos, supposons que chaque thread dispose d'une donnée locale a (par exemple un entier) et que cette donnée doit pouvoir être accédée en lecture et en écriture par l'ensemble des autres threads. Il suffit de :

- 1. définir un tableau de pointeurs d'entiers en mémoire partagée;
- 2. définir la donnée locale a dans la région parallèle;
- 3. renseigner l'emplacement correspondant dans le tableau de pointeurs en y recopiant l'adresse de a.

La figure 3 illustre cette petite « gymnastique ». Notons que cette dernière n'est pas sans danger car le contenu du tableau de pointeurs locales n'est valide qu'à l'intérieur de la région parallèle. D'autre part, le programmeur doit être extrêmement rigoureux dans la gestion des accès à la variable locale a afin de garantir, par exemple, que le thread propriétaire n'est pas en train de la modifier pendant que les autres sont en train de la lire.

```
// Nombre de threads disponibles.
     int threads;
2
   #pragma omp parallel
   #pragma omp single
     threads = omp_get_num_threads();
5
     // Tableau de pointeurs vers la donnée locale de chaque thread.
7
     int* locales[threads];
9
     // Région parallèle exécutée par l'ensemble des threads disponibles.
10
   #pragma omp parallel
11
12
       // Donnée locale à ce thread.
14
       int a;
15
16
       // Tid (thread id) de ce thread.
       const int tid = omp get thread num();
18
19
       // Ce thread autorise l'accès à sa donnée locale.
20
       locales [tid] = &a;
21
22
23
24
```

FIGURE 3 – Autoriser l'accès aux données locales d'un thread.

Notons enfin qu'il n'est pas question de définir les données locales d'un thread dans un tableau partagé (en dehors d'une région parallèle) car tout accès en écriture sur l'un de ces éléments déclencherait immanquablement le phénomène du *false sharing* (dans l'exemple de la figure 3, il y a *false sharing* juste au moment où chaque thread recopie l'adresse de sa variable locale dans le tableau partagé et c'est

tout).

L'algorithme merge de la bibliothèque standard n'est pas adapté pour fusionner deux conteneurs tout en ne conservant qu'un nombre précis d'éléments, ce qui est justement notre cas. Par conséquent, comme il n'est pas question de fusionner l'intégralité des deux conteneurs pour ne garder que le nombre d'éléments indiqué, nous devons écrire une fonction générique merge_n qui réponde à nos besoins. Sa définition est présentée en figure 4.

Dans le cas général, chaque thread dispose de trois tableaux locaux :

- bloc qui contient les éléments attribués à ce thread;
- gauche qui est destiné à recevoir les éléments du bloc du thread qui précède (tid immédiatement inférieur). Comme le thread dont le tid vaut zéro ne possède pas de prédécesseur, son tableau gauche est inutile;
- droite qui est destiné à recevoir les éléments du bloc du thread qui succède (tid immédiatement supérieur). Comme le thread dont le tid est maximum ne possède pas de successeur, son tableau droite est inutile.

Notre fonction générique merge_n nécessite la présence d'un quatrième tableau local fusion (le conteneur tiers result), de même taille que bloc et qui accueille, selon les étapes, soit :

- le résultat de la fusion entre bloc et gauche;
- celui de la fusion entre bloc et droite.

À l'issue de la fusion, le contenu du tableau fusion doit être recopié dans le tableau bloc.

La figure 5 présente l'implémentation du mécanisme des échanges-fusions que vous allez compléter. Dans ce morceau de code, chaque thread fusionne son tableau bloc avec soit son tableau droite, soit son tableau gauche puis recopie le contenu de son tableau fusion dans son tableau bloc.

2.1 Question

Écrivez le code permettant aux threads actifs d'aller chercher les éléments de leur voisin de droite (s'il existe) puis de les fusionner avec leur bloc.

2.2 Question

Écrivez le code permettant aux threads passifs d'aller chercher les éléments de leur voisin de gauche (s'il existe) puis de les fusionner avec leur bloc. Pour réaliser cette dernière opération il vous faudra parcourir le tableau gauche de droite à gauche tout en utilisant la relation d'ordre inverse (que vous pouvez créer via la fonction générique binary_negate de la bibliothèque standard).

2.3 Question

Écrivez le code permettant à chaque thread de recopier le contenu de son tableau fusion dans son tableau bloc. Vous prendez garde au fait que :

- cette recopie ne peut intervenir qu'une fois tous les échanges-fusions effectués;
- l'itération suivante de la boucle principale for ne peut commencer son exécution qu'une fois toutes les recopies effectuées.

```
template < typename InputIterator1,
             typename InputIterator2,
2
             typename OutputIterator,
3
             typename Size,
4
             typename Compare >
   OutputIterator merge n(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            OutputIterator result,
                            Size n,
                            Compare comp) {
10
11
12
      // Nous bouclons tant que le compte n'y est pas.
13
      while (n > 0) {
14
15
        // Nous sommes arrivés à la fin du premier conteneur : ne restent que
16
        // les éléments du second.
        if (first1 = last1) {
18
          return std::copy n(first2,
19
                               std::min < Size > (n, last2 - first2),
20
                               result);
21
        }
22
23
        // Nous sommes arrivés à la fin du second conteneur : ne restent que
24
        // les éléments du premier.
        if (first2 = last2) {
26
          return std::copy n(first1,
27
                               std::min < Size > (n, last1 - first1),
                               result);
29
        }
30
31
        // Nous choisissons le plus petit des deux éléments (au sens de la
32
        // relation d'ordre total utilisée) et nous avonçons conjointement
        // dans le conteneur correspondant et le conteneur cible de la fusion.
34
        *result ++ = comp(*first1, *first2) ? *first1 ++ : *first2 ++;
35
        // Un élément de moins à fusionner.
37
        n ---;
38
39
      }
41
      // Retourner l'itérateur balayant le conteneur cible.
42
      return result;
43
45
```

FIGURE 4 – Définition de l'algorithme merge_n permettant de fusionner les éléments correspondant à l'intervalle semi-ouvert [first1, last1[d'un premier conteneur avec ceux correspondant à l'intervalle semi-ouvert [first2, last2[d'un second conteneur. Les n éléments demandés sont écrit dans un conteneur tiers dans l'élément courant est repéré par l'itérateur result.

```
#pragma omp parallel
3
4
       // Boucle sur le nombre maximum d'étapes.
5
       for (int i = 0; i != 2 * threads; i ++) {
6
         // Modulo permettant de savoir si ce sont les threads de tid pair ou
         // impair qui échangent au cours de cette itération de la boucle. Les
9
         // threads dont le tid vérifie cette condition sont dits actifs tandis
10
         // que les autres sont dits passifs.
11
         const int modulo = i \% 2;
12
13
         // Les threads actifs vont chercher les éléments de leurs voisin de
         // droite (s'il existe) puis fusionnent.
15
         if (tid \% 2 = modulo) {
16
           if (! estDernier) {
17
             // -- Question 2.1 --
           } // if
19
         }
20
         // tandis que les threads passifs vont chercher ceux de leur voisin de
21
         // gauche (s'il existe) puis fusionnent.
22
         else {
23
           if (! estPremier) {
24
            // — Question 2.2 —
25
           } // if
         } // if
27
28
         // In fine, tous les threads recopient dans leur bloc local.
         // - Question 2.3 -
30
31
       } // for
32
```

FIGURE 5 – Implémentation du mécanisme des échanges-fusions.