td-prog-parallele-7-fev

February 11, 2024

1 Rappel épisodes précédents

Au cours précédent vous avez vu comment paralléliser de manière efficace un calcul de somme préfixe, aussi appelé *scan*, (souvent pour indiquer qu'on calcule les "sommes" par rapport à une opération binaire qui n'est pas forcément l'addition de nombres).

En TD nous avons vu quelques exemples d'utilisation de scan.

Pour mémoire, cette fonction se décline en deux variantes, exclusive et inclusive, dont une implémentation séquentielle rudimentaire est la suivante.

[13]:
```
def prefix_sum_exclusive(A):
    """ returns the arrays of prefix sums of A, exclusive variant"""
    return [sum(A[:i]) for i in range(len(A))]

def prefix_sum_inclusive(A):
    """ returns the arrays of prefix sums of A, inclusive variant"""
    return [sum(A[:i+1]) for i in range(len(A))]

A0 = [1,2,0,4,3]
prefix_sum_exclusive(A0), prefix_sum_inclusive(A0)
```

[13]:
```
([0, 1, 3, 3, 7], [1, 3, 3, 7, 10])
```

[15]:
```
def reduce(A, add, zero):
    """reduce, "fold_left version" """
    acc = zero
    for a in A:
        acc = add(acc,a)
    return acc

def scan_exclusive(A, add, zero):
    """ returns the arrays of prefix sums of A, exclusive variant"""
    return [reduce(A[:i], add, zero) for i in range(len(A))]

def scan_inclusive(A, add, zero):
    """ returns the arrays of prefix sums of A, inclusive variant"""
    #return [reduce(A[:i+1], add, zero) for i in range(len(A))]
    # ou plus efficace
```

```
    acc = zero
    res = []
    for a in A:
        acc = add(acc,a)
        res.append(acc)
    return res

A1 = [3.1, 4.2, 8.3, -4.2, -1.6, 2.9, -3.5, 0.0, 1.1, -2.2]
```

[15]:
```
([inf, 3.1, 3.1, 3.1, -4.2, -4.2, -4.2, -4.2, -4.2, -4.2],
 [3.1, 4.2, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3])
```

[16]:
```
import math
scan_exclusive(A1, lambda x,y: min(x,y), math.inf)
```

[16]:
```
[inf, 3.1, 3.1, 3.1, -4.2, -4.2, -4.2, -4.2, -4.2, -4.2]
```

[17]:
```
scan_inclusive(A1, max, -math.inf)
```

[17]:
```
[3.1, 4.2, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3]
```

2 Quicksort

C'est un algorithme de tri bien connu basé sur une opération de partition que nous appelerons *seg_split* pour suivre la présentation de Blelloch partie 1.5. Il s'agit de choisir un pivot (**dans tout ce TD, le pivot est la première valeur du tableau à trier**), puis de positionner toutes les autres valeurs à gauche ou à droite de ce pivot.

[2]:
```
def seg_split(A, low, high):
    """splits in place the segment A[low:high] according to the pivot
    value A[low]; returns the indexes of the first and last
    occurrences of pivot after the split"""
    segment = A[low:high]
    pivot = segment[0]  # <- choix arbitraire
    smallies = [x for x in segment if x < pivot]
    equalies = [x for x in segment if x == pivot]
    greaties = [x for x in segment if x > pivot]
    # on réordonne le segment en mettant les petits d'abord, puis les pivots,␣
 ↪puis les grands
    A[low:high] = smallies + equalies + greaties
    first_pivot_occurrence = low + len(smallies)
    last_pivot_occurrence = first_pivot_occurrence + len(equalies) - 1
    return (first_pivot_occurrence, last_pivot_occurrence)

A0 = [3, 7, 4, 1, 3, 9, -2]
seg_split(A0,0, len(A0)), A0
```

[2]:
```
((2, 3), [1, -2, 3, 3, 7, 4, 9])
```

[3]:
```
def quick_sort(A, low=0, high=None):
    """sorts in place A[low:high]"""
    if high == None : high=len(A)
    if low < high:
        print(f"{low=} {high=} {A=}")
        (first, last) = seg_split(A, low, high)
        quick_sort(A, low, first)
        quick_sort(A, last + 1, high)

A0 = [3, 7, 4, 3, 10, 9, -2]
quick_sort(A0)
A0
```

```
low=0 high=7 A=[3, 7, 4, 3, 10, 9, -2]
low=0 high=1 A=[-2, 3, 3, 7, 4, 10, 9]
low=3 high=7 A=[-2, 3, 3, 7, 4, 10, 9]
low=3 high=4 A=[-2, 3, 3, 4, 7, 10, 9]
low=5 high=7 A=[-2, 3, 3, 4, 7, 10, 9]
low=5 high=6 A=[-2, 3, 3, 4, 7, 9, 10]
```

[3]:
```
[-2, 3, 3, 4, 7, 9, 10]
```

Le but dans la suite est d'utiliser les sommes préfixes avec segments pour donner une version parallèle de quicksort où:

1. l'opération `seg_split` est parallélisée.
2. les deux appels récursifs à `quick_sort` sont parallélisés

Remarque: l'implémentation de `seg_split` donnée ci-dessous fait beaucoup de recopies. Une version "en place" de `seg_split` peut être obtenue en adaptant l'algorithme du drapeau holandais.

3 Un split parallèle à l'aide des sommes préfixes

L'idée est calculer, pour chaque valeur plus petite que le pivot, de combien de cases vers la gauche il va falloir la déplacer pour obtenir le tableau après le seg split.

Par exemple, si avant le seg split, le tableau vaut

```
[3, 7, 1, 5, 2, 3, 6]
```

après le seg split parallèle il vaudra

```
[1, 2, 3, 3, 5, 7, 6]
```

et le 1 a été décalé de 2 cases vers la gauche, tandis que le 2 a été décalé de 3 cases vers la gauche. On peut calculer ce décalage facilement en comptant combien d'éléments plus grands ou égaux au pivot précèdent la valeur à décaler dans le tableau initial, par exemple 3, 7 et 5 précèdent 2 dans le tableau initial et sont supérieurs ou égaux au pivot (3),

On peut aussi calculer, de façon similaire, un décalage vers la droite valeurs plus grandes que le pivot (c'est ce qu'on fait dans la fonction `parallel_quick_sort` à la fin), ou bien, c'est plus simple ici, renverser l'ordre des grands et calculer leur position en partant de la fin du tableau en comptant le nombre de plus grand qui les précèdent.

[4]:
```
def parallel_seg_split(A):
    # REMARQUE: par soucis de simplicité, on diffère un peu du précédent␣
 ↪seg_split
    #   - on ne fait pas un split en place, mais on retourne un nouveau tableau
    #   - on ne renvoie pas les indices du premier et dernier pivot
    pivot = A[0]
    B = [1 if x >= pivot else 0 for x in A ]
    lshift = scan_exclusive(B)  # <- potentiel de parallelisation
    B = [1 if x > pivot else 0 for x in A ]
    neg_index = scan_exclusive(B)  # <- potentiel de parallelisation
    print(f"{A      = }\n{B      = }\n{lshift = }")
    res = [pivot] * len(A)
    for i in range(len(A)):  # <- potentiel de parallelisation
        if A[i] < pivot:
            res[i - lshift[i]] = A[i]
        elif A[i] > pivot:
            res[-1 - neg_index[i]] = A[i]
    return res

A = [3, 7, 1, 5, 2, 3, 6]
parallel_seg_split(A)
```

```
A      = [3, 7, 1, 5, 2, 3, 6]
B      = [0, 1, 0, 1, 0, 0, 1]
lshift = [0, 1, 2, 2, 3, 3, 4]
```

[4]:
```
[1, 2, 3, 3, 6, 5, 7]
```

4 Le scan segmenté

En plus du tableau `A` sur lequel on effectue le scan, on prend en compte un tableau `Seg` de même longueur que `A` contenant des 0s et des 1s. Chaque 1 indique le début d'un nouveau segment.

Exemple:

```
A   =  [7, 5, 12, 8, -1, 2]
Seg =  [1, 0,  0, 1,  1, 0]
```

les segments sont `[7, 5, 12]`, `[8]`, et `[-1, 2]`, et la somme préfixe (exclusive) avec segments vaut

```
[0, 7, 12, 0, 0, -1]
```

[5]:
```
def scan_exclusive_with_segments(A, Seg):
    res = []
    n = len(A) # == len(Seg)
```

```
    for i in range(n):
        if Seg[i] == 1:
            acc = 0
        res.append(acc)
        acc += A[i]
    return res

A1 = [7,5,12,8,-1,2]
Seg1 = [1,0,0,1,1,0]
scan_exclusive_with_segments(A1, Seg1)
```

[5]: `[0, 7, 12, 0, 0, -1]`

On peut paralléliser le scan segmenté de la même façon que le scan non segmenté. Une façon de le voir, c'est de remarquer que le scan segmenté se ramène à un cas particulier de scan non segmenté non pas sur les entiers, mais sur des couples d'entiers, avec une opération binaire bien choisie (l'algo parallèle de scan est paramétrique dans l'opération binaire d'"addition" considérée).

[6]:
```
def scan_exclusive_functional(A, add, zero):
    """ returns the arrays of prefix sums of A, exclusive variant
    parametric in a binary operator `add` and its neutral `zero`"""
    return [reduce(A[:i], add, zero) for i in range(len(A))]
    # ou plus efficace (on suppose len(A) > 0)
    # acc, prev = zero, A[0]
    # for i in range(1, len(A)):
    #     A[i], acc, prev = acc, add(acc, prev), A[i]

def scan_exclusive_functional_segment(A, Seg, add, zero):
    """returns the exclusive segmented scan of A and Seg
    parametric in a binary operator `add` and its neutral `zero`"""
    pairs = list(zip(A, Seg))
    def binop(c1, c2):
        (x1, _f1), (x2, f2) = c1, c2
        if f2==1: return c2
        else: return (x1 + x2, 0)
    scans  = scan_exclusive_functional(pairs, binop, (zero, 0))
    print(f"{pairs=}\n{scans=}")
    return [scans[i][0] if Seg[i]==0 else 0 for i in range(len(A))]

scan_exclusive_functional_segment(A1, Seg1, lambda x,y:x+y, 0)
```

```
pairs=[(7, 1), (5, 0), (12, 0), (8, 1), (-1, 1), (2, 0)]
scans=[(0, 0), (7, 1), (12, 0), (24, 0), (8, 1), (-1, 1)]
```

[6]: `[0, 7, 12, 0, 0, -1]`

[7]:
```
def scan_inclusive_functional_segment(A, Seg, add, zero):
    """returns the inclusive segmented scan of A and Seg
```

```
    parametric in a binary operator `add` and its neutral `zero`"""
    ## une implémentation séquentielle (pour pouvoir l'utiliser dans la suite)
    ## TODO: une implémentation parallèle
    acc = zero
    res = []
    for i in range(len(A)):
        if Seg[i] == 1:
            acc = zero
        acc = add(acc, A[i])
        res.append(acc)
    return res
```

5 Utilisation du scan segmenté dans quick sort

On travaille avec le tableau `A` à trier ainsi que le tableau `Seg` des débuts de segments. Au départ il y a un seul segment, i.e. `Seg = [1, 0, 0, ..., 0]`.

On répète les deux opérations suivantes en alternance

1. pour chaque segment en parallèle, on partitionne le segment par rapport à son pivot;
2. on met à jour `Seg`

Et ce jusqu'à ne plus avoir que des segments d'un seul élément (Seg= `[1, 1, ..., 1]`). Pour simplifier ci-dessous on partitionne en deux, et non en trois comme Blelloch et nos fonctions seg_split précédentes.

[8]:
```
def parallel_quick_sort(A):
    # illustre l'idée de la parallélisation à l'aide des sommes préfixes␣
 ↪segmentées

    # initialisation et gestion du cas particulier du tableau vide
    n = len(A)
    if n == 0: return []
    Seg = [1] + [0] * (n-1)

    # boucle principale
    while 0 in Seg:

        # calcul des pivots
        def binop(x,y):
            if x == None: return y
            else: return x
        Pivots = scan_inclusive_functional_segment(A, Seg, binop, None)

        # calcul des drapeaux
        F = [None] * n
        for i in range(n):
            if A[i] < Pivots[i]:
```

```
                F[i] = '<'
            elif A[i] > Pivots[i]:
                F[i] = '>'
            else:
                F[i] = '='

        # calcul des décalages
        B = [1 if x != '<' else 0 for x in F]
        lshift = scan_inclusive_functional_segment(B, Seg, lambda x,y:x+y, 0)
        print(f"{A      = }\n{Seg    = }\n{Pivots = }\n{F      = }\n{B      =␣
↪}\n{lshift = }\n")
        B = [1 if x != '>' else 0 for x in F]
        rshift = scan_inclusive_functional_segment(B[::-1], [1] + Seg[n-1:0:
↪-1], lambda x,y:x+y, 0)[::-1]
        A2 = Pivots.copy()

        # partitionnement
        for i in range(n):
            if A[i] < Pivots[i]:
                A2[i - lshift[i]] = A[i]
            elif A[i] > Pivots[i]:
                A2[i + rshift[i]] = A[i]

        # mise à jour de Seg (et recopie de A2 dans A)
        for i in range(n):
            A[i] = A2[i]
            if A[i] == Pivots[i]:
                Seg[i] = Seg[min(i+1,n-1)] = 1
```

[9]:
```
A = [3, 7, 1, 5, 2, 3, 6]
parallel_quick_sort(A)
```

```
A      = [3, 7, 1, 5, 2, 3, 6]
Seg    = [1, 0, 0, 0, 0, 0, 0]
Pivots = [3, 3, 3, 3, 3, 3, 3]
F      = ['=', '>', '<', '>', '<', '=', '>']
B      = [1, 1, 0, 1, 0, 1, 1]
lshift = [1, 2, 2, 3, 3, 4, 5]

A      = [1, 2, 3, 3, 7, 5, 6]
Seg    = [1, 0, 1, 1, 1, 0, 0]
Pivots = [1, 1, 3, 3, 7, 7, 7]
F      = ['=', '>', '=', '=', '=', '<', '<']
B      = [1, 1, 1, 1, 1, 0, 0]
lshift = [1, 2, 1, 1, 1, 1, 1]

A      = [1, 2, 3, 3, 5, 6, 7]
Seg    = [1, 1, 1, 1, 1, 0, 1]
```

```
Pivots = [1, 2, 3, 3, 5, 5, 7]
F      = ['=', '=', '=', '=', '=', '>', '=']
B      = [1, 1, 1, 1, 1, 1, 1]
lshift = [1, 1, 1, 1, 1, 2, 1]
```

[10]:
```
A
```

[10]:
```
[1, 2, 3, 3, 5, 6, 7]
```