Algorithmique et optimisation discrète

DM 4MMAOD6 2021-2022

Maxime NEMO

08 novembre 2021

Contents

Analyse des défauts de cache

Le comportement d'un algorithme dépend très fortement de son implémentation. En effet, chaque choix d'implémentation fera que le programme a un coût d'exécution différent. Le coût d'un programme peut être mesuré grâce au nombre d'opérations (le travail), ou bien grâce à l'espace mémoie qu'il utilise, ou bien encore par sa localité (le nombre de défauts de cache). En effet, les données utiles pour son exécutions sont rapide à accéder lorsqu'elles sont présentes dans le cache, mais bien moins rapide lorsque celles-ci sont absentes.
Le modèle qu'on va retenir pour le cache est le modèle LRU (Least Recently Used). Cette politique a au plus 2 fois plus de défauts de cache qu'une politique optimale. On va utiliser le modèle CO pour l'analyse des défauts de cache. Celui ci consiste à supposer qu'on a un cache de taille Z, utilisant des blocs de taille L et avec une politique LRU.
Le but sera alors de trouver le nombre de défauts de cache, de concevoir un algorithme qui utilise les valeurs de Z et de L pour faire moins de défauts, et puis enfin de construire un programme qui ne fait pas trop de défauts, quelles que soient les valeurs Z et L, qui sera alors bon sur toutes les machines.

Exemple 1:
Prenons l'algorithme naïf suivant, que nous allons analyser et améliorer petit à petit.

Algorithm 1 Example 1

Input: A and B two matrix of size $n \times n$
Output: C a matrix of size $n \times n$

1: **for** $i = 0; i < n; i++$ **do**
2: **for** $j = 0; j < n; j++$ **do**
3: $c_{i,j} \leftarrow a_{i,j} \times b_{j,i}$
4: **end for**
5: **end for**

Calculons le nombre de défauts de cache dans le cadre de notre modèle CO.
Pour cela, on va faire plusieurs hypothèses :

- Si Z est très grand ($Z = \infty$) :
  Alors on va faire $\frac{n \times n}{L}$ défauts sur A, $\frac{n \times n}{L}$ défauts sur B, $\frac{n \times n}{L}$ défauts sur C
  On fait alors $\frac{3n^2}{L}$ défauts de cache
- Si Z est très petit, alors, en notant Q le nombre de défauts de cache,
  $Q(n, L, Z) = \underbrace{\frac{n^2}{L}}_{SurA} + \underbrace{\frac{n^2}{L}}_{SurC} + \underbrace{n^2}_{surB} \approx n^2$.

On remarque alors que sous les deux hypothèses précédentes, le résultat n'est pas le même, cela signifie que notre algorithme n'est pas optimisé.

L'organisation des données a un effet sur les performances. En effet, l'algorithme précédent lit les éléments de A de manière contiguë et écrit les éléments de C de manière contiguë, c'est-à-dire selon le sens du stockage, mais ce n'est pas le cas pour la lecture des éléments de B. Il s'agit ici d'un problème de la localité spatiale. Celle-ci n'étant pas bonne pour B, il va falloir régler cela. Il existe aussi des problèmes de localité temporelle, c'est-à-dire que l'on accède aux données proches dans la mémoire de façon espacée dans le temps. La solution est de faire un parcours selon le sens de stockage autant que possible.
Une première optimisation serait de transformer notre algorithme naïf en algorithme cache-aware. C'est-à-dire un programme qui est optimal en terme de défauts de cache mais qui utilise les valeurs de Z et de L pour cela.
On va utiliser des techniques de blocking. Cela consiste en l'amélioration de la localité en effectuant un parcours des matrices par blocs qui tiennent dans le cache.
On se limite volontairement au blocs rectangulaire. On pourrait très bien immaginer des blocs de formes plus complexes.

On va alors travailler par bloc de taille $\alpha \times \beta$
Méthode générale : cache-aware
Supposons α et β entiers tels que les calculs effectués dans le bloc se fassent sans que le cache soit plein si il était initialement vide.
Pour les calculs suivant, on suppose que le cache est toujours alligné pour simplifier.
On va calculer le nombre de défaut de cache par bloc, puis le multiplié par le nombre de bloc que l'on va devoir explorer. Si cette valeur est assymptofiquement la même que l'optimal qu'on avait trouvé avec l'algorithme naïf, alors on a produit un algorithme cache-aware.

Application : On a alors #défaut de cache par bloc = $\underbrace{\beta\lceil\frac{\alpha}{L}\rceil}_{surA} + \underbrace{\beta\lceil\frac{\alpha}{L}\rceil}_{surC} + \underbrace{\alpha\lceil\frac{\beta}{L}\rceil}_{surB}$
On a donc finalement : $Q(n, L, Z) \approx \frac{n}{\alpha} \times \frac{n}{\beta} \times \frac{3\alpha\beta}{L} = \Theta(\frac{n^2}{L})$ qui est bien l'optimal que l'on avait trouvé (quand $Z = \infty$) On peut aussi en déduire que $\alpha = \beta \approx \sqrt{\frac{Z}{3}}$, avec $\alpha = \beta \leq \sqrt{\frac{Z}{3}}$
On peut alors construire un algorithme cache-aware qui va effectuer les calculs par blocs (boucle *for* sur les blocs). On peut alors créer *Algorithm 5* en annexe

On a trouvé un algorithme qui, asymptotiquement, est toujours optimal, mais celui-ci dépend de Z et de L, et donc un seul et même algorithme ne peux pas etre optimal sur toutes les machines puisque Z et L seront différents. La prochaine étape est alors de créer un algorithme cache-oblivious.
Méthode générale : cache-oblivious

On va effectuer du blocking récursif, c'est-à-dire découper récursivement en blocs plus petits jusqu'à ce que la résolution d'un sous-problème tient dans le cache.
Application : *Algorithm 6* en annexe

Cas des algorithmes récursifs

La programmation récursive peut amener à des **calculs redondants** qui les rendent très mauvais niveau performance. Il est alors possible d'éliminer les redondances en suivant la méthode suivante:

- Trouver les calculs redondants en analysant les dépendances entre instructions avec un graphe d'appel par exemple.

- Éliminer ces calculs grâce à :
  - Soit de la mémoïsation, c'est-à-dire mémoriser le résultat d'un appel pour pouvoir s'en servir plus tard si on a encore besoin du résultat. Pour cela, on utilise généralement des tables de hash ou des tableaux.
  - Soit passer à de la programmation itérative avec ordonnancement topologique, c'est-à-dire effectuer les calculs nécéssaires dans un sens tel que l'on ait déjà effectué (et mémorisé) les valeurs intermédiaires nécéssaires à chaque calcul avant de le faire.

Exemple 2 : Soit la fonction f définie ci-dessous

Algorithm 2 Example 2

Input: int x
Output: res the result

```
function f(x)
    if x = 0 then
        Return 1
    end if
    res = 0
    for i = 0; i < x; i++ do
        res = res + f(i)
    end for
    Return res
end function
```

On remarque qu'il y a beaucoup de calculs redondants (tous les $f(i)$ avec $i < x$).
La première amélioration est d'utiliser de la mémoïsation. On applique la méthode générale de mémoïsation, c'est-à-dire de regarder si le calcul n'aurait pas déjà été fait précédement.

Algorithm 3 Example 2 - memorization

Input: int x
Output: res the result

```
    T = [] a hashmap
    function f(x)
        if x = 0 then
            Return 1
    5:  end if
        res = 0
        for i = 0; i < x; i++ do
            if k theney i in T
                val = T[i]
    10:     else
                val = f(i)
            end if
            res = res + val
        end for
    15: Return res
    end function
```

En faisant une analyse des dépendances entre instructions avec un graphe d'appels de f, on se rend compte que $f(x)$ dépends seulement de $f(i), i < x$. On peut alors créer une version itérative de notre algorithme en calculant les instructions qui ne dépendent en premier de rien, puis les instructions qui dépendent de rien des des instructions qu'on vient de calculer etc…

On obtient alors l'algorithme suivant :

Algorithm 4 Example 2 - iterative

Input: int x

Output: res the result

```
function f(x)
    T = [x]
    T[0] = 1
    for i = 1; i <= x; i++ do
        s = 0
        for j = 0; j < i; j++ do
            s = s + T[j]
        end for
        T[i] = s
    end for
end function
```

On a appliqué simplement la méthode, il est évident que dans notre cas, l'algorithme peut encore être simplifié, mais pas au niveau de la redondance des calculs de $f(i)$ puisque, chaque $f(i)$ n'est calculé qu'une et une seule fois.

Programmation dynamique

Pour résoudre des problèmes plus complexes, il est parfois pratique d'utiliser de la programmation dynamique. On peut modéliser un problème par une équation de Bellman lorsque :

- Une solution optimale peut s'exprimer de façon récursive
- Et la formule récursive réduit le problème général en sous-problèmes dont on va chercher la solution optimale.

Une fois l'équation de Bellman trouvée, on peut créer un programme récursif utilisant cette équation. On peut alors ensuite utiliser les méthodes d'optimisation des programmes récursifs évoqués précédemment.

Exemple 3: (inspiré d'un énoncé de l'ENS Lyon)
On veut construire une tour la plus haute possible à partir de différentes briques. On dispose de n types de briques et d'un nombre illimité de briques de chaque type. Chaque brique de type i a une longueur x_i une largeur y_i et une hauteur z_i. Chaque brique doit être posée sur la base x, y Dans la construction de la tour, une brique ne peut être placée au dessus d'une autre que si les deux dimensions de la base de la brique du dessus sont strictement inférieures aux dimensions de la rangée de briques du dessous

Solution :
Entrées :
L_{base} la longueur de la base ; l_{base} la largeur de la base ; les types de briques
Equation de Bellman :

Si il existe une brique compatible :
$h_{max}(L_{base}, l_{base}) = \max_{\text{type } i; y_i < l_{base}; x_i < L_{base}} (z_i + h_{max}((L_{base}//x_i) \times x_i, y_i)$
$h_{max}(L_{base}, l_{base}) = 0$ si aucune brique ne respecte les conditions du max

On peut alors créer le *code fournis en annexe*.

Annexes

Algorithm 5 Example 1 - Cache-Aware

Input: A and B two matrix of size $n \times n$
Output: C a matrix of size $n \times n$

```
for I = 0; I < n; I += α do
    int i_max ← min(I + α, m)
    for J = 0; J < n; J += α do
        int j_max ← min(J + α, m)
        for i = I; i < i_max; i++ do
            for j = J; j < j_max; j++ do
                c_{i,j} ← a_{i,j} × b_{j,i}
            end for
        end for
    end for
end for
```

Algorithm 6 Example 1 - Cache-Oblivious

Input: A and B two matrix of size $n \times n$, T a threshold, a, b two indexes initially = 0, c, d two indexes initially = n

Output: C a matrix of size $n \times n$

function $f_{rec}(a, b, c, d)$
  $NbLine \leftarrow b - a$
  $NbCol \leftarrow d - c$
  if $NbLine < T$ and $NbCol < T$ **then**
    for $i = a; i < b; i++$ **do**
      for $j = c; j < d; j++$ **do**
        $c_{i,j} \leftarrow a_{i,j} \times b_{j,i}$
      end for
    end for
  else
    if $NbLine > NbCol$ **then**
      $f_{rec}(a + NbLine/2, b, c, d)$
      $f_{rec}(a, b + NbLine/2, c, d)$
    else
      $f_{rec}(a, b, c + NbCol/2, d)$
      $f_{rec}(a, b, c, d + NbCol/2)$
    end if
  end if
end function

Exemple 3 : Algorithme

```
#!/usr/bin/env python3

souvegarde = {} # Permettera de construire la solution optimale par la suite

def h_max(longueur_base, largeur_base):
```

```
    """
    longueur_base = longeur maximale que peut avoir le mur
    largeur_base = largeur maximale que peut avoir le mur
    """

    # Conditions aux limites:
    if longueur_base == 0:
        return 0
    if largeur_base == 0:
        return 0


    # Trouver les brique qui peuvent convenir:
    B = []
    for brique in briques:
        # Si on a un brique moins longue et moins large que la base, alors elle
        if brique[1] < largeur_base and brique[0] < longueur_base:
            B.append(brique)

    if len(B) == 0:
        return 0

    # Trouver la brique qui va maximiser l'equation de Bellman:
    i_max = 0
    val_max = 0
    for i, brique in enumerate(B):
        val = brique[2] + h_max((longueur_base//brique[0])*brique[0], brique[1])
        if val > val_max:
            val_max = val
            i_max = i

    # enregistrer le resultat dans souvegarde:
    if (longueur_base, largeur_base) in souvegarde:
        if souvegarde[(longueur_base, largeur_base)][1] < val_max:
            souvegarde[(longueur_base, largeur_base)] = (B[i_max], val_max)
        # Sinon laisser la solution precedente qui etait deja meilleure
    else:
        souvegarde[(longueur_base, largeur_base)] = (B[i_max], val_max)
```

```
    return val_max


if __name__ == "__main__":
    briques = ((10, 3, 5), (5, 6, 4), (1, 2, 3), (6, 4, 2), (7, 6, 5))
    n = len(briques)
    longueur_base = 100
    largeur_base = 10
    print("En prenant comme briques : ", briques)
    print("Et comme taille de fondations : ", (100, 10))
    print("h_max = ", h_max(longueur_base, largeur_base))
    print("En utilisant les briques suivantes (fondations en haut, ciel en bas)"
    long = longueur_base
    larg = largeur_base
    while (long, larg) in souvegarde:
        b = souvegarde[(long, larg)][0]
        print(b)
        long = (long//b[0])*b[0]
        larg = b[1]
```