Chapitre 1

PRAM : algorithmes abstraits

Dans ce chapitre nous présentons le modèle PRAM (Parallel Random Access Machine) qui est la norme de présentation théorique des algorithmes parallèles. La première section est un rappel de notions sur la complexité des algorithmes. Les sections suivantes expliquent le modèle PRAM et son utilité pour estimer le degré de parallélisme des algorithmes, en rapport au nombre d'unités de calcul. On donne des algorithmes PRAM pour les opérations parallèles les plus universelles que sont la diffusion, la réduction et le calcul parallèle des préfixes. Pour chacun on pose et résoud plusieurs exercices théoriques. Toutes ces notions seront reprises et améliorées ensuite avec le modèle BSP qui ajoute des informations pour la programmation parallèle réelle : mémoire distribuée, communications et synchronisations globales.

1.1 Prérequis

- Connaître les bases d'un langage de programmation algorithmique (C, Java, Ada, etc.).
- Connaître des algorithmes de base comme la recherche dichotomique, un algorithme de tri etc.
- Savoir calculer la limite à l'infini de fonctions mathématiques simples.
- Avoir des notions élémentaires d'architecture des ordinateurs.

1.2 Révision : complexité des algorithmes

Dans cette section nous rappelons brièvement les bases de la complexité des algorithmes. Elles servent à estimer le temps de calcul dont la réduction est la seule raison d'être des algorithmes parallèles. Les notions définies et expliquées sont les suivantes.

- Algorithmes
- Mesures de complexité
- Notations O, Ω et Θ.

– Complexité pire cas et cas moyen

Commençons par rappeler qu'un algorithme est une procédure effective, par exemple un programme informatique, qui se termine *en temps fini* pour toute donnée d'entrée.

1.2.1 Algorithmes

Un *problème de calcul*

$$\mathcal{P} : A \to B$$

est une fonction entre des données d'entrée A et des données de sortie B. Par exemple le problème informatique du tri peut être défini comme un problème de calcul

$$\text{Sort} : \text{float array} \to \text{float array}$$

tel que $\text{Sort}(T)$ est un tableau qui contient les mêmes données que T mais qui est en ordre croissant (et on peut ajouter des précisions comme un type de données variable, leur taille exacte en octets, les répétitions possibles des données etc.).

Un *algorithme* pour le problème de calcul

$$\mathcal{P} : A \to B$$

est :

– une procédure effective

– qui $\forall x \in A$ calcule $\mathcal{P}x$ en temps fini

L'ensemble A représente toutes les données d'entrée possibles. L'ensemble B représente les données de sortie. Par exemple un algorithme de tri peut être de type

$$\text{sort} : \text{float array} \to \text{float array}$$

s'il s'applique à des tableaux de nombres en virgule flottante. Un algorithme qui recherche une valeur dans un tableau pourrait être de type

$$\text{find} : \text{float} * (\text{float array}) \to \text{int}$$

si la réponse désirée est l'indice du tableau où la valeur est, éventuellement, trouvée. Par contre un programme qui transforme un flux infini de messages en un autre flux infini n'est pas un algorithme. Par exemple un processus qui exécute une boucle sans fin requête-réponse n'est pas un algorithme. En effet on ne peut lui attribuer un temps *fini* de calcul.

1.2.2 Mesures de complexité

On mesure le coût d'exécution d'un algorithme par la notion de *complexité* des calculs. La complexité de l'algorithme $f : A \to B$ est une statistique sur

$$\{\text{Coût}(f, x) \mid x \in A\}$$

où le coût est soit le nombre de :

- mots-mémoire utilisés (espace) ou
- d'instructions-machine effectuées (temps)

par le **calcul** de f sur entrée x. La mesure la plus importante et la plus courante est celle de **complexité en temps.**

Remarque : la complexité en espace est majorée par la complexité en temps, puisqu'il faut une instruction pour chaque allocation d'un mot.

Dans la pratique, la complexité en temps d'un algorithme est une estimation de la fonction qui donne le temps de calcul de l'algorithme en fonction de ses données d'entrée. Comme il arrive très souvent que ce temps ne dépende que de la taille des données d'entrée, la fonction prend siplement en entrée la taille n des données. Par exemple la complexité d'un algorithme de tri prendra comme entrée la taille du tableau à trier. L'unité de mesure de cette valeur n est, pour les discussions théoriques, un simple décompte du nombre de scalaires (par exemple n vaudrait la longueur du tableau `float array`). Pour une étude pratique il faudrait bien sûr préciser la taille de ces scalaires en octets. De même, la valeur de la fonction de complexité peut être une valeur théorique en unités de temps ou concrètement des secondes mesurées. Par convention les spécialistes font l'hypothèse que tout algorithme est compilé en opérations machines dont le temps d'exécution est fixe. Cette hypothèse n'est pas rigoureusement vraie mais il existe bien un facteur multiplicatif qui borne les variations de temps d'exécution des opérations machine réelles. Si par exemple une addition est réalisée en 0,5ns sur une architecture donnée, il se peut qu'une opération de chargement `LOAD` prenne 3 fois plus de temps. C'est en ce sens que la complexité est une estimation du temps de calcul : elle sert à donner un taux de croissance qui sert de formule analytique pour extrapoler ou prévoir des temps de calcul réels à partir de données réelles.

Motivons cela par un petit exemple d'algorithme :

```
s:= 0;
for i=1 to n do
  s:= s+x[i];
done
```

Question : Quelle est la complexité en temps de cet algorithme ?
Pour y répondre imaginons qu'on puisse compiler le corps de la boucle en une séquence d'instructions comme :

```
LOAD R1 (i)
ADD  R1 R1 (x)
LOAD R2 (R1)
LOAD R3 (s)
ADD  R2 R2 R3
STORE R2 (s)
```

qui sera exécutée n fois. La complexité devrait être $T(n) = 6n + k$ où k est le nombre d'instructions exécutées à l'initialisation et à la sortie de boucle. Or on suppose :

1. que le matériel exact n'importe pas pour la fonction de complexité
2. LOAD, STORE, ADD etc. en temps constant (indépendant de n).

Dans les analyses théoriques, on suppose toujours un *accès direct* à la mémoire : le modèle standard s'appelle RAM ou *random-access machine* [1]. En d'autres mots, toute instruction élémentaire prend un temps constant unitaire, comme si toutes les données résidaient toujours en mémoire interne et comme si toute opération arithmétique, logique, de contrôle pouvait être réalisée en un nombre invariable de cyles d'horloge. **La complexité en temps se résume ici à** an **où** a **est indépendante de** n (le terme additif constant est absorbé).

1.2.3 Notations asymptotiques

On a vu que la fonction de complexité d'un algorithme est celle qui associe à l'entier positif n le nombre $T(n)$ d'instructions effectuées sur une entrée de taille n. C'est une estimation de son temps de calcul.

Pour éliminer les constantes de la fonction de complexité, on la résume à une classe d'équivalence de fonctions, appelée son **ordre de complexité.** Ainsi $an + k$, an et n sont toutes du même ordre de complexité qu'on appelle *linéaire* (en n). Cette simplification est nécessaire car les constantes comme a, k dépendent de la machine cible et du système logiciel plutôt que de l'algorithme. Il n'est pas nécessaire de calculer la fonction de complexité *exacte.*

1.2.4 Majorant, minorant, équivalence asymptotique

Lorsqu'une complexité $T(n)$ est telle que

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} \leq k$$

où $k > 0$ est une constante indépendante de n, on dit que $T(n) \in O(f(n))$ [1]. Lorsqu'une complexité $T(n)$ est telle que

$$\lim_{n\to\infty} \frac{f(n)}{T(n)} \leq k$$

où $k > 0$ est une constante indépendante de n, on dit que $T(n) \in \Omega(f(n))$. Lorsqu'une complexité $T(n)$ est telle que $T(n) \in O(f(n))$ et $T(n) \in \Omega(f(n))$, on dit que l'ordre exact de complexité $T(n)$ est $f(n)$, et on écrit $T(n) \in \Theta(f(n))$.

1. On peut aussi écrire $T(n) = O(f(n))$ par abus de notation.

1.2.5 Exemples

Ici a et k sont des constantes indépendantes de la taille des données n. Les logarithmes sont toujours binaires.

- $an + k \in \Theta(n)$
- $n \log n \in \Omega(n)$
- $n \log n \in O(n^2)$
- $6n^3 + 2n + k \in \Theta(n^3)$.

1.2.6 Pire cas et cas moyen

Il peut arriver que le temps de calcul dépende de la valeur d'entrée et pas seulement de sa taille n. Pensons par exemple à un algorithme de recherche d'une valeur dans un tableau. Quelle que soit la méthode, il peut arriver que la valeur recherchée ne se trouve pas parmi celles du tableau. L'algorithme devrait la comparer à toutes les n valeurs présentes, et son temps de calcul ne pourrait alors par être inférieur à n. Si par contre la valeur recherchée est, par hasard, trouvée dès le début de la recherche alors le temps de calcul sera très faible, par exemple 2 ou 3 opérations, donc constant par rapport à n.

Autre exemple plus significatif : le célèbre algorithme de tri de Hoare ou quicksort [23]. Celui-ci sert à trier un tableau de n valeurs mais son temps de calcul varie en fonction des valeurs qu'il y trouve.

Soit un algorithme $f : A \to B$, $A_n = \{x \in A \mid x \text{ de taille } n\}$ et $T(f, x)$ est le temps de calcul de f sur x. Il faut distinguer

1. la complexité pire cas

$$\max_{x \in A_n} T(f, x) \qquad \text{et}$$

2. la complexité moyenne

$$\text{moy}_{x \in A_n} T(f, x)$$

La seconde peut dépendre de la distribution de probabilité des $x \in A_n$.

Exemple : recherche linéaire dans un tableau non-trié : complexité moyenne $= n/2$ *si on a certitude de trouver la valeur.*

1.2.7 Optimalité

On dit qu'un algorithme est *optimal* si sa complexité (pire cas en temps) $T(n)$ minore asymptotiquement celle de tous les algorithmes *possibles* pour le même problème.

Par exemple : la recherche linéaire dans un tableau non-trié, de complexité $\Theta(n)$ est optimale (pourquoi ?).

Par abus de langage, on dit parfois qu'un algorithme est optimal lorsque sa complexité est celle du meilleur algorithme *connu* pour le problème en question.

1.2.8 Complexité vs temps de calcul

Soient f_1, f_2 deux algorithmes pour le même problème tels que la complexité (asymptotique) de f_2 est la meilleure des deux.

Il arrive souvent que f_1 soit plus simple et donc que sa **constante** de complexité soit inférieure à celle de f_2.

Pour des valeurs de n relativement basses disons $n < n_0$, le temps de calcul de f_1 sera le meilleur alors que pour $n > n_0$ ce sera le contraire. Si le seuil n_0 est connu, on pourra construire un algorithme hybride qui applique f_1 ou f_2 selon la valeur de n afin de minimiser le temps de calcul.

1.3 Exercices : complexité des algorithmes

1. Montrer que si $f(n) = O(g(n))$ alors pour $a > 0$ on a $a * f(n) = O(g(n))$.
2. Montrer que si $f_1(n) = O(g_1(n))$ et $f_2(n) = O(g_2(n))$ alors $f_1 + f_2$ est $O(g_1 + g_2)$.
3. Montrer que si $f_1(n) = O(g_1(n))$ et $f_2(n) = O(g_2(n))$ alors $f_1 + f_2$ est $O(\max(g_1, g_2))$.
4. Expliquer précisément ce que définissent les fonctions suivantes :

```
function R1(n:integer) return integer is
begin
    if (n <= 1) then return 1
    else return 2 * R1(n-1)      end if
end;

function R2(n:integer) return integer is
begin
    if (n <= 1) then return 1
    else return R2(n-1) + R2(n-1)  end if
end;
```

5. Comparer la complexité en nombre d'appels récursifs de R1 et R2.
6. Calculer la complexité du programme ci-dessous en nombre d'instructions inst comme fonction de la valeur n.

```
for i=1 to n-1 do
    for j= i+1 to n do
        for k= 1 to j do
            X[i,j,k] := inst(X[i,j,k] )
        end do
    end do
end do;
```

Comment pourrait-on savoir si la complexité *moyenne* est la même que la complexité pire-cas ?

7. Dire précisément ce que définissent les fonctions suivantes sur les entiers naturels :

```
function f1(a,n:integer) return integer is
begin
    if (n=0) then return 1
    else return a * f1(a,n-1)
    end if
end

function f2(a,n:integer) return integer is
x: integer
begin
    x:= 1;
    for i= 1 to n do x:= x*a end for;
    return x
end

function f3(a,n:integer) return integer is
    begin
        if (n=0) then return 1
        else if (n mod 2=0) then return f3(a*a, n/2)
        else return a* f3(a*a, n/2)
        end if
    end
```

8. Calculer et comparer la complexité des fonctions f1, f2 et f3. Pour f1 et f3 estimer la complexité en nombre d'appels récursifs et pour f2 en nombre de passages dans la boucle.

1.4 Corrigé : complexité des algorithmes

1. $f = O(g)$ signifie que que la limite de $\frac{f(n)}{g(n)}$ quand n tend vers l'infini est finie, disons majorée par une constante k. Alors $\frac{a*f(n)}{g(n)}$ sera majoré par $a * k$ et donc $a * f = O(g)$.
2. $f_1(n) = O(g_1(n))$ et $f_2(n) = O(g_2(n))$ signifient que $\frac{f_1(n)}{g_1(n)}$ est majoré par k_1 et $\frac{f_2(n)}{g_2(n)}$ est majoré par k_2.
   Alors

$$\frac{(f_1(n) + f_2(n))}{(g_1(n) + g_2(n))} = \frac{f_1(n)}{(g_1(n) + g_2(n))} + \frac{f_2(n)}{(g_1(n) + g_2(n))}$$

ce qui est majoré par

$$\frac{f_1(n)}{g_1(n)} + \frac{f_2(n)}{g_2(n)} \leq k_1 + k_2$$

puisque toutes ces fonctions sont à valeurs positives. On a donc montré que $f_1 + f_2 = O(g_1 + g_2)$ avec la limite à l'infini majorée par $k_1 + k_2$.

3. $f_1(n) = O(g_1(n))$ et $f_2(n) = O(g_2(n))$ signifient que $\frac{f_1(n)}{g_1(n)}$ est majoré par k_1 et $\frac{f_2(n)}{g_2(n)}$ est majoré par k_2.
   Alors, puisque

$$(f_1(n) + f_2(n)) \leq 2 * \max(f_1, f_2)(n)$$

qui est majoré par $2*\max(g_1, g_2)$ on a $(f_1+f_2) \in O(2*\max(g_1, g_2)$. Cela signifie qu'il existe une constante k qui majore

$$\frac{(f_1(n) + f_2(n))}{(2 * \max(g_1(n), g_2(n))} \leq \frac{(f_1(n) + f_2(n))}{(\max(g_1(n), g_2(n))}$$

puisque toutes les valeurs sont positives.
On a ainsi montré que $f_1 + f_2 = O(\max(g_1 + g_2))$.

4. Les deux fonctions calculent la même valeur numérique soit 2^{n-1}.
5. Lorsqu'on appelle R1(n), celle-ci s'appelle elle-même sur n-1, puis n-2 etc, donc n fois en tout. Cela représente une complexité en temps $O(n)$ car chaque appel de R1 utilise un nombre fixe d'opérations.
   Lorsqu'on appelle R2(n), celle-ci s'appelle elle-même *deux fois* sur n-1, puis *quatre fois* sur n-2, etc. Cela représente une complexité en temps $T(n) = 2*T(n-1)$ c'est-à-dire que la valeur de R1 est aussi la complexité de R2 ! La fonction R2 est donc une version excessivement inefficace de R1. Elle revient à compter une valeur en autant d'opérations que la valeur elle-même (ce que l'espèce humaine évite de faire depuis les temps très anciens où elle écrivait les nombres en notation unaire). Le fait que son écriture soit très proche de celle de R1 doit nous mettre en garde sur la facilité de produire des algorithmes très inefficaces par inadvertance.
6. Le programme traverse sa boucle la plus interne un nombre de fois égal à $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=1}^{j} 1$ et à chaque fois effectue un seul appel à la fonctions inst. Le nombre d'instructions `inst` est donc

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} j \leq \sum_{i=1}^{n-1} \sum_{j=1}^{n} j$$

ce qu'on peut majorer par

$$\sum_{i=1}^{n} \sum_{j=1}^{n} j = \sum_{i=1}^{n} \frac{n(n+1)}{2} = n * \frac{n(n+1)}{2} = O(n^3).$$