Module 9: Multiprocesseur (suite)

1 Introduction

Nous avons vu dans la leçon précédente les différents modèles d'architectures parallèles. Nous avons présenté en détail le parallélisme d'instructions à savoir les aléas structurels, les aléas de données et les aléas de contrôle. Nous avons également présenté le mécanisme de prédiction des branchements avec ses deux types statique et dynamique.

Dans cette leçon, nous allons présenter d'autres types de parallélisme à savoir le parallélisme de tâches et le parallélisme de données. Nous terminerons cette leçon par présenter le concept de graphe de dépendance, les multiprocesseurs, et une brève description des cartes graphiques.

2 Parallélisme de tâches

Ce type de parallélisme se présente lorsque des tâches différentes exécutent des opérations différentes sur des données qui peuvent être différents ou non.

Par exemple, le code suivant montre des opérations différentes sur des données :

```
1 x = 5
2 y = 10
3 z = 15
4 a = (x * z) / (x + z)
5 b = (x + y) / 2
6 c = a - b
7 d = a * b
```

L'idée de base dans le parallélisme de tâches est qu'une tâche est divisée en plusieurs sous-tâches indépendantes qui sont exécutées sur des unités de calcul distinctes. Mais, en pratique, implémenter le parallélisme de tâches au sein d'un algorithme nécessitera des modifications, parfois substantielles et complexes au code.

2.1 Loi d'Amdahl

La loi d'Amdahl explique, selon une formule mathématique, que le gain maximal atteignable pour une parallélisation d'un algorithme, est limité par la portion non parallélisée de l'algorithme. Ceci dit que nous ne pourrons pas atteindre les performances maximales si l'algorithme n'est pas complètement parallélisable. La loi d'Amdahl est souvent liée à la scalabilité des applications. La scalabilité est une propriété d'une application à être exécutée efficacement sur un très grand nombre de processeurs. En d'autres mots, la partie non parallèle d'un programme limite les performances et fixe une borne supérieure à la scalabilité.

La formule mathématique de la loi d'Amdahl peut être présentée de la façon suivante. Soit :

— F la partie de l'algorithme (ou prbolème d'une manière générale) qui peut être parallélisée.
— A_{par} accélération (speedup) obtenue sur la partie parallélisable.
— P le nombre de processeurs.

La formule devient donc :

$$A_{total} = \frac{1}{(1-F) + \frac{F}{A_{par}}} \tag{1}$$

Nous pouvons également calculer l'efficacité du système selon la formule suivante :

$$Efficacité = \frac{A_{total}}{P} \tag{2}$$

À titre d'exemple, si nous avons 8 processeurs ($P = 8$, $A_{par} = 8$), et que le pourcentage de la partie parallélisable est de 70% ($F = 0.7$), nous avons :

$$A_{total} = \frac{1}{(1-0.7) + \frac{0.7}{8}} = 2.58 \tag{3}$$

Pareillement, l'efficacité égale à :

$$Efficacité = \frac{2.58}{8} = 0.32(32\%) \tag{4}$$

Maintenant, si pourcentage de la partie parallélisable augmente à 90 % (F = 0.9), nous aurons :

$$A_{total} = \frac{1}{(1-0.9) + \frac{0.9}{8}} = 5.33 \tag{5}$$

Pareillement, l'efficacité égale à :

$$Efficacité = \frac{5.33}{8} = 0.66(66\%) \tag{6}$$

Nous pouvons donc voir que l'efficacité est doublée malgré que l'augmentation dans le pourcentage de la partie parallélisable est de 20% (90 - 70 = 20).

3 Parallélisme de données

Le parallélisme de données se produit lorsque plusieurs tâches indépendantes appliquent la même opération sur différents éléments d'un ensemble de données.

Dans le parallélisme de données, les calculs sont associés aux données. Donc, pour pouvoir distribuer les calculs, il suffit de distribuer les données. De cette façon, les calculs peuvent se faire en parallèle.

3.1 Map/Reduce

Le Map/Reduce ou MapReduce est un modèle de programmation parallèle inventé par Google permettant de manipuler de grandes quantités de données en les distribuant dans un cluster de machines pour être traitées.

Le MapReduce est largement utilisé pour le traitement des données massives (**big data**). Ce modèle est implémenté dans plusieurs framworks. Le plus connu de ces framworks est le Hadoop [1]. La figure 1 présente un exemple de fonctionnement du modèle MapReduce pour le comptage de mots dans un texte [2].

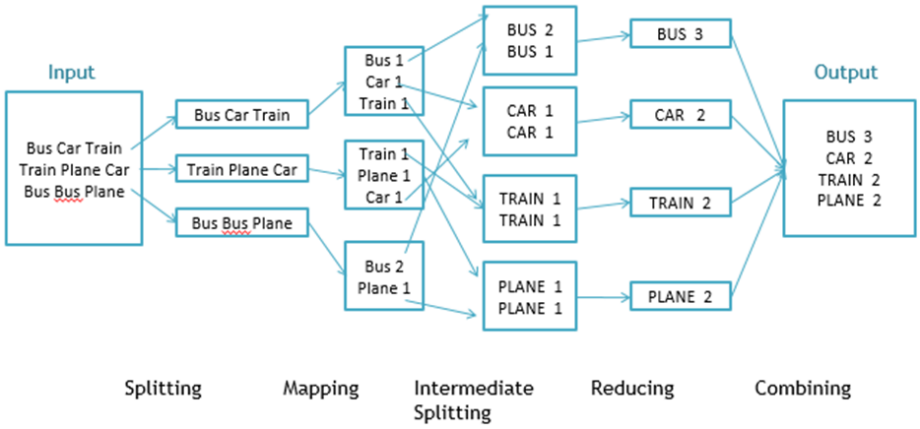

FIGURE 1 – Exemple de fonctionnement de MapReduce pour le comptage de mots.

Comme vous pouvez le constater à partir de la figure 1, il y a 5 étapes dans ce processus de calcul :

1. Splitting (ou division) : le paramètre de division permet de diviser les données en entrée pour pouvoir distribuer les calculs par la suite. Ce paramètre pourrait être un espace par exemple, une virgule, ou une nouvelle ligne par exemple comme c'est le cas dans la figure 1.

1. https ://fr.wikipedia.org/wiki/Hadoop
2. Figure tirée de `https://dzone.com/articles/word-count-hello-word-program-in-mapreduce`

2. Mapping : la mapping prend en entrée un ensemble de données et génère en sortie un autre ensemble de données de la forme clé-valeur (key-value).
3. Intermediate splitting (division intermédiaire) : permet de réorganiser les partictions de données de telle sorte que les les données avec la même clé devraient se retrouver sur le même cluster pour préparer la phase de reducing.
4. Reducing : permet de regrouper les données et additioner les données ayant la même clé.
5. Combining : permet de récupérer les résultats de la phase de reducing à partir des différents clusters et les retourner en sortie.

Ce qu'il faut retenir à partir de cet exemple, est que le modèle de MapReduce applique un traitement similaire à chaque élément.

Un exemple du code Java pour les deux fonctions Map et Reduce est présenté en bas. Cet exemple est tirée intégralement du site web `https://dzone.com/articles/word-count-hello-word-program-in-mapreduce`.

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import ↩
   org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import ↩
   org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
public class WordCount {
public static void main(String [] args) throws ↩
   Exception
{
Configuration c=new Configuration();
String[] files=new ↩
   GenericOptionsParser(c,args).getRemainingArgs();
Path input=new Path(files[0]);
Path output=new Path(files[1]);
Job j=new Job(c,"wordcount");
j.setJarByClass(WordCount.class);
j.setMapperClass(MapForWordCount.class);
j.setReducerClass(ReduceForWordCount.class);
j.setOutputKeyClass(Text.class);
j.setOutputValueClass(IntWritable.class);
```

```
26 FileInputFormat.addInputPath(j, input);
27 FileOutputFormat.setOutputPath(j, output);
28 System.exit(j.waitForCompletion(true)?0:1);
29 }
30 public static class MapForWordCount extends ↩
      Mapper<LongWritable, Text, Text, IntWritable>{
31 public void map(LongWritable key, Text value, ↩
      Context con) throws IOException, ↩
      InterruptedException
32 {
33 String line = value.toString();
34 String[] words=line.split(",");
35 for(String word: words )
36 {
37       Text outputKey = new ↩
            Text(word.toUpperCase().trim());
38   IntWritable outputValue = new IntWritable(1);
39   con.write(outputKey, outputValue);
40 }
41 }
42 }
43 public static class ReduceForWordCount extends ↩
      Reducer<Text, IntWritable, Text, IntWritable>
44 {
45 public void reduce(Text word, Iterable<IntWritable> ↩
      values, Context con) throws IOException, ↩
      InterruptedException
46 {
47 int sum = 0;
48    for(IntWritable value : values)
49    {
50    sum += value.get();
51    }
52    con.write(word, new IntWritable(sum));
53 }
54 }
55 }
```

4 Graphe de dépendance

La dépendance des opérations ou de tâches représente un point très important lors du passage au parallélisme.

Pour que deux opérations OP1 et OP2 d'un programme soient dépendantes, deux choses doivent être vérifiées :

— Collision : OP1 et OP2 sont en collision si toutes les deux accèdent au même emplacement mémoire avec au moins une opération d'écriture mémoire.
— Précédence : l'opération OP1 précède l'opération OP2 si OP1 s'exécute avant OP2 dans le programme.
— Donc, il existe une dépendance de OP1 vers OP2 si :
  — OP1 et OP2 sont en collision, **ET**
  — OP1 s'exécute avant OP2 (précédence).

L'ensemble de dépendances définit ce qu'on appelle le graphe de dépendance du programme.

Le graphe de dépendance Permet de représenter visuellement le parallélisme d'un programme ou d'un algorithme. Les sommets dans le graphe de dépendance représentent les tâches, et arcs représentent les relations de dépendance entre les différentes tâches. La figure 2 présente un exemple d'un graphe de dépendance d'un programme. Les tâches sont représentées par la lettre **T**.

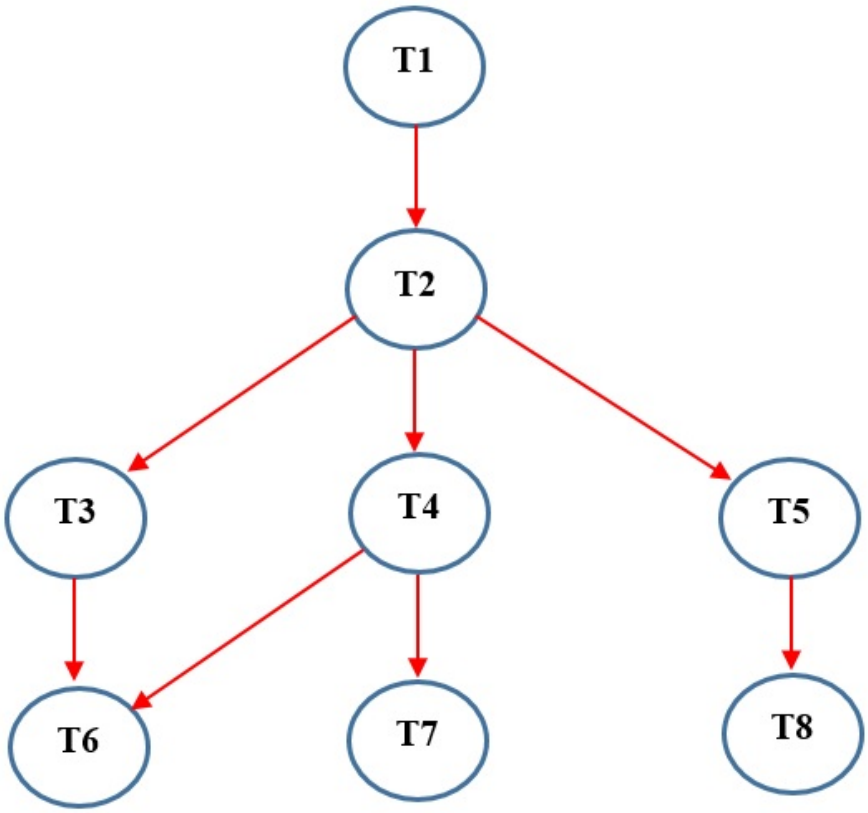

FIGURE 2 – Exemple de graphe de dépendance entre les tâches.

5 Multiprocesseur

Un multiprocesseur est un ordinateur comprenant plusieurs processeurs avec une seule mémoire principale partagée entre ces processeurs. Bien évidemment, chaque processeur possède sa propre mémoire cache. Les caches permettent de réduire considérablement la charge sur le bus et la mémoire principale.

Dans un ordinateur multiprocesseur, les processeurs communiquent par des lectures/écritures sur des variables en mémoire partagée (physiquement ou virtuellemen).

Nous pouvons trouver ces connexions dans les architectures SMP (symmetric memory multiprocessor) ou multiprocesseur symétrique à mémoire partagée comme le montre la figure 3 [3].

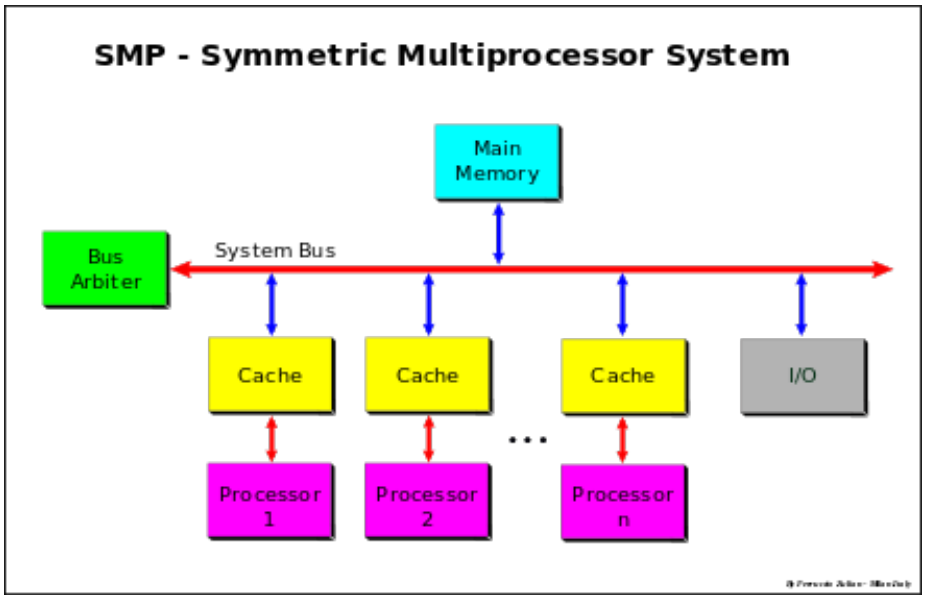

FIGURE 3 – Multiprocesseur symétrique à mémoire partagée.

Il existe principalement deux types de multiprocesseur :

— **Centralisé** : dans ce type de multiprocesseur, la mémoire principale est partagée physiquement entre les processeurs comme illustrée dans la figure 3.

— **Distribué** : dans ce type de multiprocesseur, la mémoire principale est partagée virtuellement entre les processeurs, c'est-à-dire, que chaque processeur se voit attribuer une mémoire principale (de façon virtuelle).

3. Figure tirée du Wikipédia `https://en.wikipedia.org/wiki/Symmetric_multiprocessing`

Il y a cependant des inconvénients avec ces types de partages comme la cohérence de cache et le problème de synchronisation. Bien que la duplication de certaines données dans les caches permet de minimiser les accès mémoire, le problème qui se pose c'est qu'est-ce qui se passe si une écriture est effectuée dans ces caches ?

Pour répondre à cette question, il existe ce qu'on appelle un protocole de cohérence de cache. Ce protocole représente un ensemble de règles assurant que les processeurs possèdent la même valeur d'un emplacement mémoire.

6 GPU (Graphical Processing Unit)

Dans le contexte du parallélisme, il est important de parler, même brièvement, des cartes graphiques qui prennent actuellement de l'ampleur.

Les cartes graphiques sont basées sur une architecture massivement parallèles. Ces cartes traditionnellement utilisées pour le calcul graphique, elles sont maintenant largement utilisées pour des calculs génériques comme l'exécution des algorithmes basés sur les réseaux de neurones.

Il existe plusieurs types de cartes graphiques puissantes à savoir :

— API CUDA de Nvidia
— API ATI Stream de ATI
— API générique de OpenCL

La figure 4 présente un exemple d'une carte graphique Geforce [4].

4. Image tirée de Google Image

FIGURE 4 – Carte graphique Geforce de Nvidia.

Les cartes graphiques peuvent maintenant avoir plus de 3500 coeurs (unités de calcul) comme dans le cas de la carte graphique Nvidia GeForce GTX 1080 Ti. Un processeurs composé d'au mois deux coeurs est appelé multi-coeur.