

## **TP5**

### **SI4 – mapReduce**

Ce TP permet de mettre en place un traitement de données basé sur le principe de « map » et « reduce ».

Ce travail va être mené en langage C. La mise en œuvre peut se faire dans tous les environnements.

L'opération de mappage consiste à mettre en place une structure de données qui va contenir des couples (clés, valeurs).

L'opération de mappage sera répartie sur deux processus au minimum (voir maximum).

La dernière opération « reduce », permet de reprendre le résultat des processus et de faire une synthèse des résultats. Cette synthèse se fait encore une fois avec une structure de données qui travaille toujours avec des couples (clés, valeurs).


Attention : Dans ce travail, le programmeur à la charge de répartir les travaux sur les processus, ce qui n'est le cas avec un framework.

Les différentes étapes de travail sont les suivantes :

1. Elaborer un fichier .txt qui sera analysé. Vous pouvez d'ailleurs prévoir plusieurs jeux d'essais, afin de mettre au point l'algorithme. L'objectif est de pouvoir traiter un fichier avec plus de 10 000 entrées, pour commencer à faire du traitement plus massif.
2. Charger le fichier dans une structure en mémoire. Il ne faut pas oublier que la mémoire centrale permet les meilleures performances. Dans le cas du Big Data, la plupart des données sont présentes en mémoire centrale. Les machines actuelles offrent suffisamment de mémoire pour travailler facilement.
3. Une fois le fichier chargé, il faut commencer avec un traitement mono-thread. Dans ce cas, le premier tableau est lu, puis un tableau de sortie est constitué avec des opérations de « Shuffling » (battage). Plus simplement il faut mêler les éléments identiques afin de faire des cumuls sur la rubrique valeur de chaque clé.
4. Dans le cas d'un mono-thread, il n'y a pas d'opération de « reduce » (réduction) ou de mappage dissociées. Le simple battage, qui permet de faire le cumul des valeurs, permet d'obtenir le résultat final. Le mapReduce se limite donc à une simple fonction baptisée mapReduce.

## 1. Fonctionnement mono-thread

Le fichier d'entrée est le suivant (format .txt) :



```
Toto
Aaron
Lucie
Aaron
Tibaud
Tibaud
Lucie
Stephane
Jean
Stephane
David
```

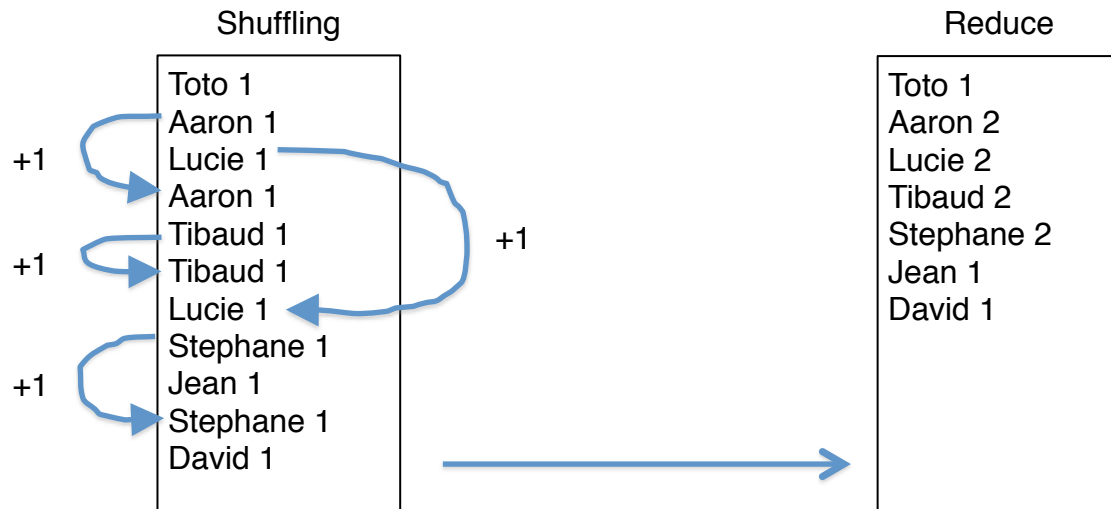
Le fichier est placé dans un tableau.

Le contenu du tableau de départ est le suivant (on fabrique la map):

map

Toto 1
Aaron 1
Lucie 1
Aaron 1
Tibaud 1
Tibaud 1
Lucie 1
Stephane 1
Jean 1
Stephane 1
David 1

L'opération de « shuffling », va permettre de regrouper les éléments identiques afin de faire un comptage :



Le résultat est simplement affiché, mais peut ensuite être trié, et être enregistré dans un fichier.

Le battage peut nécessiter un tableau intermédiaire si nécessaire.

```
input = loadFileInMemory(nameFile, countItem, sizeofItem);

customAreaOutput * output;
output = malloc(countItem * sizeof(customAreaOutput));

output=mapReduce(input, countItem, sizeofItem);
```

## 2. Fonctionnement Multithreads

L'objectif est ensuite de pouvoir répartir le traitement sur plusieurs processus (threads).

Il y a plusieurs possibilités (POSIX) pour créer des processus. Le problème réside simplement dans le partage des données (contexte mémoire).

Dans ce cas, seules les API contenues dans le fichier `#include <pthread.h>`, seront utilisées.

Pour obtenir des applications POSIX, il faut travailler avec Xcode, code::blocks, voir devcpp. Pour Visual Studio, il faut voir pour installer les fichiers nécessaires.

Une procédure pour VS : <http://lingtoling.wordpress.com/2012/04/24/setting-up-pthreads-in-windows-under-visual-studio/>

Pour créer des processus, nous allons travailler avec les API POSIX :

- Pthread\_create : création d'un processus associé à une routine, le passage de paramètres se fait à l'aide d'une structure.
- Pthread\_join : attente de la fin du processus léger pour terminer le processus principal. Join peut retourner des résultats du processus, cependant, il est

préférable de placer cela dans la structure utilisée à l'appel. La récupération de ces informations se fait après le join.

```
pthread_create(&fils[0], NULL, map,valeur[0]);
pthread_create(&fils[1], NULL, map,valeur[1]);

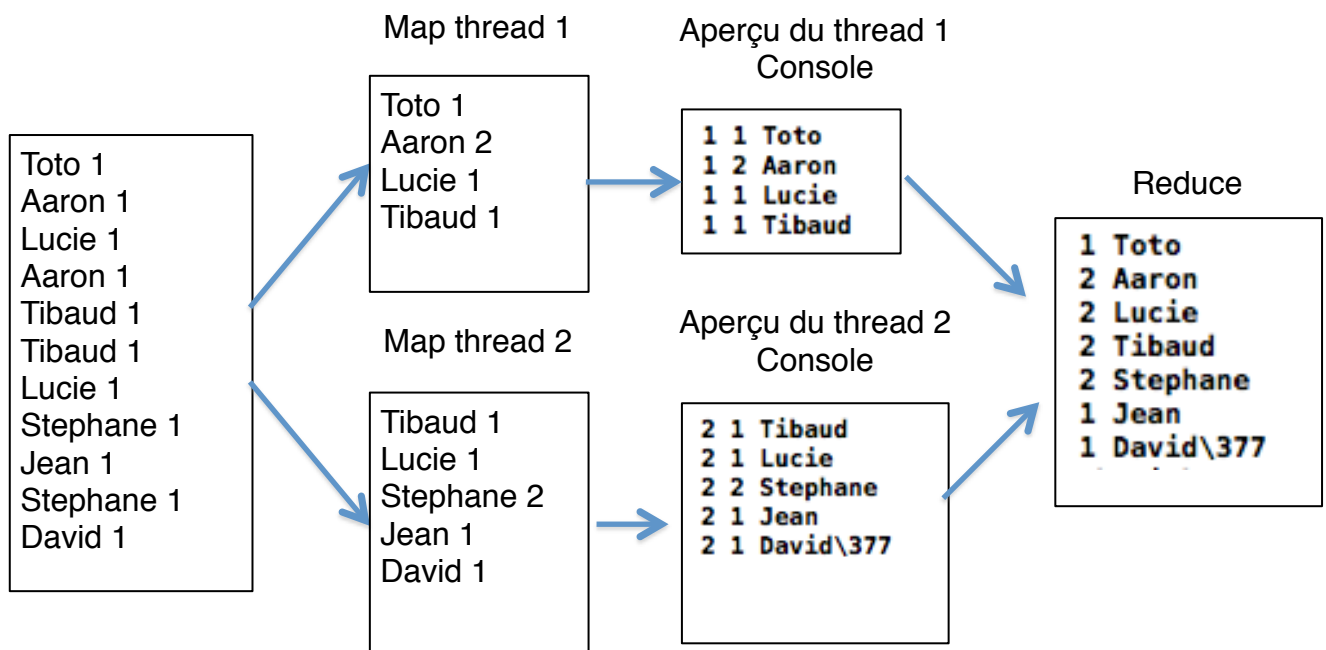
pthread_join(fils[0], NULL);
pthread_join(fils[1], NULL);

output[0] = valeur[0]->text;
output[1] = valeur[1]->text;

reduce(output[0], output[1],countItem/2, sizeofItem);
```

Il faut prévoir de créer au moins deux processus afin de faire du calcul parallèle. Vous pouvez créer des tableaux de processus.

A partir de là, l'approche est différente de celle utilisée avec le mono-thread. Il faut prendre le tableau du départ et le répartir suivant le nombre de processus. Avec deux, il suffit de couper en deux le tableau !



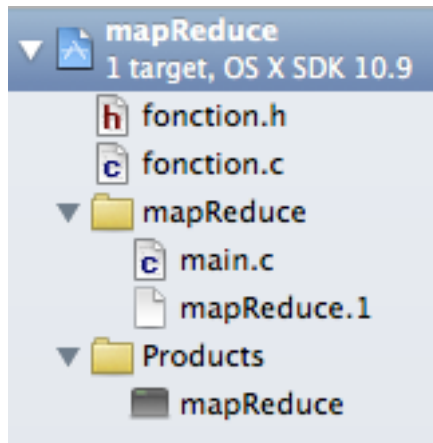
Normalement ce n'est pas le mappage de départ qui doit être coupé en deux mais le fichier de départ.

Couper le mappage en deux est assez bénéfique, puisque les threads interviennent sur deux portions du tableau qui sont différentes. Il n'y a pas dans ce cas de problème de partage de données (concurrence).

Attention : le nombre de processus détermine combien de morceaux vous faites au démarrage. Ici, le nombre de morceaux vaut 2, parce le nombre de processus vaut deux. Il faut donc veiller à ce que le nombre d'éléments à traiter soit supérieur au nombre de processus utilisés.

Attention : Dans ce travail, il est nécessaire de pouvoir utiliser un debugger, sous peine de longue(s) soirée(s) à trouver les erreurs.

### 3. Arborescence du projet



#include nécessaires :

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <time.h>

#include <sys/types.h>
#include <unistd.h>

#include <pthread.h>
```