Apache Spark et MPI :
Une comparaison entre deux frameworks pour le calcul distribué

Aurélien Coet

Université de Genève

2016

Table des Matières

1. Introduction

On observe depuis quelques années le développement rapide du phénomène dit du *big data*, l'explosion de la taille et de la quantité de données numériques disponibles à travers le monde. La société *IBM* estime même aujourd'hui que 90 % des données actuellement répertoriées dans le monde ont été générées au cours des deux dernières années seulement [1]. Le *big data* est un phénomène relativement nouveau, ce qui s'explique en partie par la récente apparition de nombreuses nouvelles sources d'information, comme les senseurs présents dans les appareils connectés qui ont envahi notre quotidien, les importants flux d'informations générés sur le web et les réseaux sociaux, ou encore les diverses expériences et mesures scientifiques rendues possibles par les progrès technologiques effectués ces dernières années.

L'accès à de très grandes quantités de données permis par le *big data* ouvre la voie à de nouvelles perspectives intéressantes dans de nombreux domaines tels que la médecine, l'économie, le *data mining* ou encore l'intelligence artificielle. Dans ce contexte, l'informatique moderne se voit confrontée à de nouveaux défis de taille. En particulier, le traitement de quantités de données extrêmement grandes pose des problèmes computationnels considérables, malgré la puissance de calcul toujours croissante des ordinateurs actuels. Afin de faire face à ce problème, une solution de nos jours largement acceptée est celle de l'utilisation du parallélisme et du calcul distribué.

De nombreux *frameworks* et librairies spécialisés dans le calcul parallèle et distribué existent. Toutefois, deux d'entre-eux sont principalement utilisés et plus largement acceptés. Le premier est *MPI – Message Passing Interface* – une interface de programmation spécialisée dans le calcul sur les machines parallèles à mémoire distribuée, devenue aujourd'hui le standard *de facto* pour la programmation d'applications scientifiques à hautes performances. Le second est *Spark*, un *framework* de calcul distribué développé par la fondation *Apache*. Ce dernier est particulièrement adapté pour le traitement itératif de grandes quantités de données, et il est souvent utilisé pour des algorithmes de *machine learning* et de *data mining*.

Bien que *Spark* et *MPI* ne fonctionnent pas selon des modèles de programmation identiques et qu'ils n'aient à la base pas été créés avec les mêmes objectifs, tous deux proposent de nombreuses fonctionnalités comparables en termes de calcul parallèle sur des machines à mémoire distribuée. Il est aussi généralement accepté que *MPI* offre des performances supérieures à d'autres *frameworks* comme *Spark*. Le but de ce projet est donc d'effectuer une comparaison entre *Spark* et *MPI*, non seulement par rapport à leur simplicité d'utilisation et de déploiement sur un cluster, mais aussi en termes de performances pures. On tentera plus particulièrement de savoir s'il est possible d'obtenir des performances de calcul supérieures avec *MPI*, comme cela est attendu, et éventuellement de proposer un modèle basé sur cette interface qui offrirait la même simplicité que *Spark* mais bénéficierait de la puissance de *MPI*.

Ce document se divise en plusieurs parties : la première consiste en un état de l'art présentant les deux environnements considérés dans ce travail, ainsi que les comparaisons ayant déjà été effectuées dans la littérature sur ce sujet. Un algorithme de base pour les mesures comparatives entre *frameworks* est proposé dans la seconde partie, et son implémentation séquentielle est décrite. Le détail des implémentations parallèles avec *Spark* et *MPI* est ensuite donné, et l'accent est particulièrement mis sur les défis d'optimisation rencontrés lors de l'écriture de la version *MPI*. Après cela, les résultats des mesures de performances avec les deux versions distribuées de l'algorithme sont présentés et discutés. Finalement, une proposition est faite pour un *framework* basé sur *MPI* qui offrirait la même simplicité d'utilisation que *Spark*.

2. État de l'art

2.1 Apache Spark

Apache Spark [2] est un framework de calcul distribué adapté pour le traitement de grandes quantités de données. Il est inspiré du modèle *MapReduce* proposé par les employés de *Google* Jeffrey Dean et Sanjay Ghemawat en 2004 [3]. D'abord initié à l'université de Berkeley en 2009 [4], *Spark* a vu son développement transféré en 2013 à la fondation *Apache*, sous licence *open source*. C'est aujourd'hui l'un des plus gros projets de ce type dans le domaine du *big data*, avec plus de 200 entreprises qui contribuent activement à son développement.

Spark est prévu pour fonctionner sur des clusters de machines à mémoire distribuée au-dessus de *Hadoop [5]* (un autre framework de la fondation *Apache*), dont il utilise le système de fichiers distribué, le *HDFS*, ainsi que le gestionnaire de clusters, *YARN* (bien que *Spark* puisse aussi fonctionner sur d'autres gestionnaires de clusters ou même en mode *standalone*). Le concept à la base du modèle proposé par le *framework* est celui de *RDD* (*Resilient Distributed Dataset*) [6]. Les *RDD* sont des collections de données distribuées sur lesquelles un ensemble d'opérations de calcul et de réduction sont possibles, dont en particulier les deux opérations *map* et *reduce* héritées du modèle *MapReduce*. Le modèle des *RDD* tire ses origines dans le concept du parallélisme de données (c'est-à-dire l'idée qu'un programme est parallélisé en distribuant ses données et les tâches à effectuer dessus entre plusieurs processeurs), ainsi que dans celui de la programmation fonctionnelle. Le *framework* permet d'ailleurs très facilement d'utiliser les *lambda-fonctions* dans ses programmes.

Spark est écrit en *Scala*, mais il offre aussi des interfaces de programmation pour les langages *Java*, *Python* et *R* (en plus de l'interface *Scala)*. L'avantage de l'interface de programmation de *Spark* est qu'elle permet de développer des applications avec la même simplicité qu'un programme séquentiel : tout l'aspect de distribution des tâches de calcul entre processeurs et de réduction des résultats vers un seul d'entre eux est géré par *Spark*, et ce de façon totalement transparente. Un autre avantage est que les *RDD* sont « *fault-tolerant* », c'est-à-dire que toute erreur pouvant survenir dans les données lors de l'exécution

d'un programme est corrigée de façon automatique par le système, grâce au modèle de stockage de données redondant du *HDFS* ainsi qu'à la façon dont sont générés les *RDD*.

Un petit exemple de programme *Spark* écrit en *Python* est donné ci-dessous :

```
sc = SparkContext()
distribData = sc.parallelize([1,2,3,4,5,6])
result = distribData.map(lambda a : a**2).reduce(lambda a, b : a+b)
```

Dans ce code, un *RDD* contenant une liste d'entiers est créé sur tous les processeurs du programme, et le contenu de la liste entre les parenthèses de *parallelize* est distribué entre eux. Ensuite, la fonction *map* est appelée sur chaque processeur afin d'appliquer la fonction lambda entre les parenthèses à chaque élément du *RDD*. Finalement, la fonction *reduce* combine les données du *RDD* en leur appliquant la fonction lambda entre ses parenthèses et les réunit sur un seul processeur.

Les applications *Spark* sont exécutées dans une machine virtuelle *Java* (*JVM*) sur les processeurs qui participent au programme. Ceci implique que le programmeur d'une telle application ne possède pas réellement de contrôle sur la représentation en mémoire des données traitées par le *framework*, contrairement à ce qui est possible dans une application *MPI* programmée en *C* ou en *C++*, par exemple. On verra plus loin que l'utilisation de la *JVM* peut en partie expliquer les différences de performances souvent observées entre les deux environnements de programmation.

La gestion de la distribution des tâches sur un cluster dans *Spark* est faite selon un modèle *maître-esclave*, comme cela est expliqué de façon détaillée dans [7]. Plus de détails sont aussi donnés sur le modèle de programmation adopté par *Spark* sur le site officiel du *framework* [2]. On ne discutera pas plus ici de son fonctionnement, cela n'étant pas le sujet central de ce travail. Toutefois, les sources mentionnées ci-dessus représentent d'excellents compléments aux informations données dans cette section. L'article original des auteurs de *Spark* ([4]) représente aussi un bon point de départ pour mieux comprendre les motivations et les choix derrière cet environnement de programmation.

2.2 MPI

MPI est une interface de programmation standardisée pour les communications inter-processeurs dans les applications parallèles à mémoire distribuée [8]. Elle a été développée à partir de 1992, mais sa première version standardisée fut proposée en 1994. Elle a depuis connu un très grand succès, en particulier dans le domaine du calcul scientifique et des applications nécessitant des performances élevées. De nombreuses implémentations du standard existent, mais la plus largement acceptée aujourd'hui est la librairie *open source OpenMPI* [9].

MPI n'offre pas d'autres primitives de programmation que des fonctions permettant aux processeurs dans un cluster de communiquer à travers le réseau d'interconnexion qui les relie. Ainsi, tout l'aspect logique et calculatoire d'un programme écrit avec MPI est laissé au

programmeur, qui doit lui-même gérer les questions de parallélisation et de synchronisation entre les processeurs dans son application. Ceci implique une grande liberté pour les développeurs dans la façon dont ils distribuent les tâches de calculs entre processeurs dans leurs programmes, mais aussi potentiellement une très grande complexité pour l'écriture d'applications parallèles avec l'*API*. Tous les aspects de logique et d'optimisation liés au parallélisme étant laissés au programmeur, il peut être parfois long et difficile d'écrire un programme fonctionnel et efficace avec l'interface (il faut éviter les interblocages entre processus, déterminer soi-même comment distribuer les données à traiter entre les processeurs, comment synchroniser les calculs, etc...). De plus, la gestion de la mémoire est totalement contrôlée par l'utilisateur dans *MPI*, puisque l'interface est définie pour les langages *C* et *C++*. Ceci permet d'effectuer de nombreuse optimisations lors de la programmation, mais implique aussi qu'un certain nombre d'erreurs parfois difficiles à corriger sont rendues possibles.

Il est généralement accepté que *MPI* permet, lorsqu'il est correctement utilisé, d'obtenir des performances de calcul très élevées. La plupart des implémentations de l'interface proposent des communications entre processeurs très bien optimisées, qui permettent d'écrire des programmes parallèles efficaces dans lesquels les temps de communication représentent un *overhead* lié au parallélisme minime.

Un petit exemple de programme écrit avec *MPI* est donné ci-dessous :

```
#include <iostream>
#include « mpi.h »

int main(int argc, char** argv){
	int myRank;
	int nProc;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
	MPI_Comm_size(MPI_COMM_WORLD, &nProc);

	if (myRank>0)
		MPI_Send(&myRank, 1, MPI_INT, myRank-1, 0, MPI_COMM_WORLD);

	int recv = 0;
	if (myRank<nProc-1)
		MPI_Recv(&recv, 1, MPI_INT, myRank+1, 0, MPI_COMM_WORLD,
				MPI_STATUS_IGNORE);

	std::cout << « Message received on proc. » << myRank;
	std::cout << « from proc. » << recv << std::endl;

	MPI_Finalize();
	return 0;
}
```

Le code ci-dessus est exécuté par tous les processeurs participant au programme. Ils commencent par initialiser *MPI*, puis ils récupèrent dans les variables *myRank* et *nProc* leur rang et le nombre total de processeurs. Ensuite, tous les processeurs dont le rang est plus grand que zéro envoient la valeur de ce dernier à leur voisin de rang inférieur. Après cela, les

processeurs de rang plus petit que *nProc* reçoivent de leur voisin de droite la valeur de son rang, puis ils affichent tous un message contenant leur rang et celui qu'ils ont reçu. Dans le code présenté ici, aucune garantie ne peut être faite quant à l'ordre dans lequel les processeurs recevront les valeurs et les imprimeront sur la sortie standard.

De nombreuses informations sur le standard *MPI* sont disponibles sur [8]. Une liste des routines *MPI* ainsi que leur description est donnée dans [10]. Des tutoriels existent également sur [11] pour apprendre à utiliser l'API.

2.3 Comparaisons entre Spark et MPI

Tout d'abord, il est intéressant de noter que *Spark* et *MPI* ne sont pas utilisés et maintenus par les mêmes communautés d'utilisateurs. *Spark* est principalement utilisé dans les domaines de l'intelligence artificielle et du *data mining*, surtout par des entreprises, alors que *MPI* est principalement utilisé par la communauté scientifique pour les calculs nécessitant de très hautes performances. Il est en général assez rare de voir les deux communautés interagir entre elles et utiliser le framework de l'autre.

Les applications *Spark* sont souvent exécutées sur des plateformes basées sur le *cloud* (*Amazon Web Services, Google Cloud Platform*, etc...), alors que *MPI* est en général installé et utilisé sur des clusters spécialement dédiés à l'exécution d'applications scientifiques parallèles. Dans beaucoup de structures telles que les universités, tout un environnement de calcul parallèle et de clusters supportant *MPI* existe déjà, ce qui encourage d'avantage l'utilisation de ce dernier que de *Spark* dans ces milieux. En effet, dans un tel contexte, l'utilisation de *Spark* n'est pas toujours évidente : il n'est souvent pas possible de mettre en place de nouveaux clusters dédiés à *Spark* uniquement (généralement pour des raisons financières), et installer ce dernier sur un cluster adapté à *MPI* peut s'avérer problématique (pour des questions de stabilité du système, entre autres). Il existe malgré tout des exemples de tentatives de faire fonctionner les deux environnements ensemble sur un même *cluster*. C'est le cas de [12], qui propose une façon d'intégrer *Spark* à un environnement fonctionnant avec un système de queue de type *PBS* (comme *TORQUE*, par exemple).

Au-delà de questions purement matérielles, le fait que *Spark* et *MPI* ne soient pas utilisés par les mêmes communautés s'explique aussi par le fait que les deux environnement de programmation n'ont pas été pensés à la base avec les mêmes objectifs (*MPI* plutôt pour le calcul scientifique, *Spark* pour le traitement du *big data*). Toutefois, ils offrent dans certains cas les mêmes possibilités pour le calcul distribué, c'est pourquoi il est intéressant de comparer les performances pouvant être obtenues avec chacun d'entre eux.

Il existe dans la littérature un certain nombre de comparaisons entre *MPI* et *MapReduce*, « l'ancêtre » de *Spark* ([13] et [14] par exemple). En revanche, les comparaisons de performances entre *Spark* et *MPI* sont encore assez rares. Nous mentionnerons toutefois ici deux d'entre elles : [15] et [16]. On citera aussi [17], qui ne compare pas *Spark* et *MPI* mais

qui offre un point de vue intéressant sur la notion de *scalabilité* dans les systèmes distribués tels que *Spark* et *MapReduce*.

Dans [15], les auteurs de l'article ont tenté de déterminer le type de performances que pouvait offrir *Spark* dans des applications scientifiques. Pour ce faire, ils ont implémenté avec ce dernier un algorithme de classification dans le domaine de la physique des hautes énergies (ou physique des particules), puis ils ont mesuré la qualité des solutions qu'offrait l'implémentation ainsi que son temps d'exécution. Ils ont ensuite implémenté le même algorithme à l'aide de *MPI*, puis ils ont effectué les mêmes mesures dessus, afin de pouvoir comparer les résultats et ainsi déterminer le genre de performances qui pouvaient être attendues de la part de *Spark*. Ils sont arrivés à la conclusion que ce dernier avait l'avantage d'offrir un modèle de programmation simple, qui cachait les difficultés d'implémentation pouvant être rencontrées avec *MPI*, et qu'il offrait une solution très *scalable* par rapport à la taille du problème étudié. Toutefois, ils ont aussi remarqué que *MPI* bénéficiait de performances jusqu'à 500 fois plus élevées que *Spark*.

Dans [16], les auteurs ont comparé les implémentations avec *Spark* et *OpenMP/MPI* de deux algorithmes de *machine learning*, *KNN* et *SVM-Pegasos*. Ils ont exécuté leurs programmes sur des machines virtuelles déployées dans un cluster basé sur le cloud. Le programme *Spark* a été testé sur une machine faisant tourner une version de *Spark* sur *Hadoop*, et la version *OpenMP/MPI* sur une machine simulant un cluster de type *Beowulf*. Les conclusions auxquelles sont arrivés les auteurs de l'article sont que, bien que *Spark* puisse être préféré à *OpenMP* couplé avec *MPI* pour des raisons de facilité de programmation et de bonne *scalabilité*, il est encore très loin d'offrir les mêmes performances que *OpenMP/MPI*, qui est souvent jusqu'à plusieurs dizaines de fois plus rapide dans son exécution.

Dans [17], les auteurs ont comparé les performances qu'ils pouvaient obtenir avec des programmes séquentiels sur des algorithmes donnés par rapport à des implémentations avec *Spark* et d'autres *frameworks* distribués du même type. En particulier, ils ont tenté de déterminer s'il n'était pas dans certains cas préférable d'utiliser des programmes séquentiels plutôt que parallèles, à cause de l'*overhead* important dû à la distribution des tâches et des données dans les systèmes distribués. Ils sont arrivés à la conclusion que, souvent, un très grand nombre de processeurs devait être utilisé dans des applications distribuées pour pouvoir observer un gain de performances par rapport aux versions séquentielles des mêmes programmes. Les auteurs critiquent aussi dans l'article le fait que la grande *scalabilité* de nombreux systèmes distribués soit souvent en réalité due au fait qu'un important *overhead* lié au parallélisme empêche d'observer des gains de performances significatifs en-dessous d'un grand nombre de processeurs utilisés.

À la lumière des résultats observés dans les articles ci-dessus, on s'attend à ce que notre comparaison entre *Spark* et *MPI* soit favorable au second en termes de performances. Toutefois, il est aussi évident que *Spark* fournit une plus grande simplicité de programmation

que *MPI*, c'est pourquoi une solution qui bénéficierait des avantages des deux *frameworks* est une idée qui semble intéressante à explorer.

3. Algorithme de base et version séquentielle

3.1 Algorithme de base

L'algorithme de base que nous utiliserons dans ce travail pour comparer les performances de *Spark* et de *MPI* est celui du *wordcount*, c'est-à-dire le comptage de la fréquence d'apparition des mots dans un texte. C'est un algorithme qui est souvent utilisé pour introduire *Spark* aux nouveaux utilisateurs du *framework*. On a donc une implémentation type du *wordcount* avec *Spark* qui nous est donnée sur le site officiel du projet et qu'on pourra utiliser comme référence lors de nos futures mesures. De plus, l'algorithme choisi présente l'avantage d'être très simple, ce qui nous permettra de nous concentrer sur les aspects de parallélisation plus que d'algorithmique lors de notre implémentation avec *MPI*.

On peut résumer l'algorithme du *wordcount* par le pseudo-code suivant :

1) On déclare une collection de paires de clés-valeurs *wordCounts*, dont les clés seront des chaînes de caractères et les valeurs des entiers.

2) Le texte du fichier à traiter est chargé dans une chaîne de caractères *text*.

3) Pour chaque mot *word* dans *text :*

    a) Si *word* ϵ *wordCounts* :

        - Incrémenter la valeur associée à *word* dans *wordCounts.*

    b) Sinon :

        - Ajouter la paire clé-valeur (*word*, 1) à *wordCounts.*

4) Retourner *wordCounts*.

Si on sait qu'on a N mots au total dans le texte à traiter et M mots différents parmi eux ($M \leq N$), alors la complexité de l'algorithme ci-dessus peut être définie comme $O(N\ f(M))$, où $f(M)$ est une fonction qui correspond au nombre d'opérations nécessaires pour accéder à une clé donnée dans la collection *wordCounts* et modifier la valeur qui lui est associée. f est une fonction de M, car en général le temps pour accéder à un élément dans une collection dépend de la taille de cette dernière. Le temps pour mettre à jour une valeur dans une collection est souvent le même que celui pour simplement y accéder, c'est pourquoi on utilise ici $f(M)$ pour représenter le temps de modification d'un élément.

Afin de pouvoir déterminer plus tard les *speedups* qu'offrent les versions parallèles de l'algorithme avec *Spark* et *MPI*, une version séquentielle du *wordcount* a été implémentée en *C++*. Elle se trouve dans le fichier *wordcount_seq.cpp* de ce projet. Pour déterminer le *speedup* d'un programme parallèle, on le compare toujours à la meilleure version séquentielle

connue. C'est pour cette raison que le *C++* a été choisi : on s'attend à ce que les performances offertes par ce langage soient les meilleures possibles.

3.2 Implémentation en C++

Le code *C++* de *wordcount_seq.cpp* charge le contenu du fichier à traiter dans un objet de type *std::stringstream* (librairie standard *C++*), puis il fait appel à la fonction *count_words()* définie dans les fichiers *useful_functions.h* et *useful_functions.cpp* de ce projet. La fonction copie le contenu du *stringstream* dans un tableau de caractère (chaîne de caractères de type *C*), puis elle utilise la fonction *strtok()* de la librairie standard *C* afin de « tokeniser » la chaîne (la diviser en mots). Ensuite, les *tokens* obtenus sont parcourus et ajoutés comme clés à un *hashmap* de type *std::unordered_map*, avec comme valeurs associées leur nombre d'occurrences dans le texte.

Le code de la fonction *count_words()* est donné ci-dessous :

```
void count_words(std::unordered_map<std::string, int>& wordcount,
   std::stringstream& text, char const& delim){

       char* c_text = new char[text.str().size()+1];
       strcpy(c_text, text.str().c_str());
       c_text[text.str().size()] = '\0';

       // 'strtok()' is called on the string to tokenize it.
       char* c_word = strtok(c_text, " \n");
       while (c_word){
              std::string word(c_word);
              if (!word.empty())
                     wordcount[word]++;
              c_word = strtok(NULL, " \n");
       }
       delete [] c_text;
}
```

Le choix de l'utilisation de la fonction *strtok()* dans *count_words* s'explique par les excellentes performances qu'elle offre par rapport à d'autres fonctions du même type en *C++*. De même, des *unordered maps* ont été utilisé plutôt que des *maps* (*std::map*) pour le dictionnaire, car le temps d'accès à un élément dans un *unordered map* de taille *N* est de complexité *O(1)* amortie (complexité constante en moyenne, linéaire en *N* dans le pire des cas), contre *O(logN)* pour les *maps*. Si on reprend la formule de la complexité de l'algorithme donnée en 3.1, l'utilisation d'un *unordered map* implique qu'on aura une complexité moyenne pour l'algorithme en *O(N)*, contre *O(NlogM)* avec les *maps* (où *M* est le nombre de mots différents dans le texte). La différence principale entre les *maps* et les *unordered maps* est que les premiers sont ordonnés (ce sont en fait des structures de *treemap*), alors que ce n'est pas le cas des seconds (ce sont des *hashmap*). Comme l'ordre des paires ne nous intéressait pas ici, le choix s'est naturellement porté sur la structure offrant les meilleures performances.

Dans la pratique, des mesures du temps d'exécution de l'implémentation *C++* avec les deux types de *maps* ont confirmé l'avantage d'utiliser un *unordered map*, comme le montrent les résultats dans la *figure 1*. Les mesures qui y sont présentées ont été effectuées sur

4 fichiers différents, *big.txt*, *large.txt*, *verylarge.txt* et *large3.txt*, dont les tailles sont respectivement de 6.5 Mo, 15.5 Mo, 23 Mo et 311 Mo. Plus de détails sur ces fichiers sont donnés ultérieurement dans ce travail. Les temps en secondes indiqués dans le tableau sont des valeurs moyennes calculées sur 5 exécution du programme à chaque fois.

Fichier utilisé pour les mesures	**Temps d'exécution moyen de *wordcount_seq.cpp* avec des map (en secondes)**	**Temps d'exécution moyen avec des unordered_map (en secondes)**	**Speedup obtenu avec les *unordered_map* par rapport aux *map***
big.txt	1.3033	0.4268	3.0539
large.txt	3.3116	1.0833	3.0569
verylarge.txt	4.9099	1.5864	3.0949
large3.txt	70.5287	22.0336	3.2010

Figure 1 – Table des temps d'exécution de wordcount_seq.cpp

On remarque dans la *figure 1* que plus la taille du fichier traité par *wordcount_seq.cpp* est grande, plus il devient intéressant d'utiliser des *unordered maps* comme structures de données plutôt que des *maps*. En effet, plus un fichier traité a une taille importante, plus le dictionnaire des mots différents rencontrés dans le texte aura tendance à être grand. De plus, il y a généralement plus de mots dans les fichiers plus grands, ce qui signifie que la structure pour stocker les fréquences des mots est accédée et mise à jour plus souvent.

4. Implémentation de l'algorithme avec Spark

L'implémentation de la version distribuée du *wordcount* avec *Spark* se trouve dans le fichier *wordcount.py* joint à ce document. La base du code est celle proposée pour le langage *Python* sur le site officiel de *Spark*, [18], dans la partie d'introduction rapide du *framework*. On a rajouté quelques éléments à cette base pour effectuer des mesures du temps d'exécution du script et permettre d'entrer en paramètres au programme le nom d'un fichier à traiter, ainsi que le nom d'un fichier de sortie où écrire les résultats obtenus.

Le morceau de code qui implémente l'algorithme est le suivant :

```
sc = SparkContext()

# The file to work on is opened in parallel on the nodes in the cluster.
filename = sys.argv[1]
textfile = sc.textFile(filename)

# The words in the distributed file are counted and their count is saved in a map.
wordcount = textfile.flatMap(lambda line: line.split()).map(lambda word:
             (word,1)).reduceByKey(lambda a,b: a+b)

# The results on the nodes are combined and retrieved by the master node.
result = wordcount.collect()
```

On peut résumer le fonctionnement du *wordcount* avec *Spark* comme suit :

- Tout d'abord, un contexte *Spark* est créé avec la ligne `sc = SparkContext()`. On ouvre à partir de ce contexte un fichier texte de façon distribuée sur les processeurs participant au programme avec la commande `textfile = sc.textFile(filename)`, où *filename* est le nom du fichier à traiter.

- Ensuite, le programme divise chaque ligne du texte en une liste des mots qui étaient séparés par des espaces, ce qui retourne un *RDD* composé d'une liste de mots sur chaque processeur, pour sa partie de fichier. La fonction *map* associe ensuite à chaque mot du *RDD* la valeur 1 dans un tuple, ce qui donne un nouveau *RDD* de paires *(mot, 1)*. Enfin, la méthode *reduceByKey* récupère toutes les clés identiques dans le *RDD* précédent et additionne leurs valeurs les unes avec les autres, ce qui donne finalement un *RDD* contenant pour chaque mot du texte une paire dont la clé est le mot et la valeur son nombre d'occurrences.

- Le *RDD* final des mots et de leur fréquence est encore distribué sur plusieurs processeurs après l'opération *reduceByKey*. On appelle donc la méthode *collect* des *RDD* dessus, afin de réduire les résultats vers un seul processeur.

On voit bien ci-dessus que le code pour le *wordcount* avec *Spark* est très simple. Il tient sur moins d'une cinquantaine de lignes (si on compte aussi le code pour les mesures du temps d'exécution du script et l'écriture des résultats dans un fichier externe). À aucun endroit il n'est nécessaire de spécifier au programme comment distribuer les tâches de calcul entre les processeurs, ni comment récupérer les résultats à la fin de l'exécution. Ces éléments sont totalement gérés par *Spark*, sans que l'utilisateur n'ait à s'en soucier. *MPI* n'offre pas un tel confort de programmation, comme nous allons le voir plus loin.

5. Implémentation de l'algorithme avec MPI

L'implémentation du *wordcount* avec MPI présente deux principaux défis de parallélisation et d'optimisation qu'il a fallu adresser au cours ce travail. Le premier défi était de choisir la façon dont sont distribuées les tâches de calcul entre les processeurs du programme, c'est-à-dire comment lire le fichier dont les mots doivent être comptés et le distribuer entre les processeurs. Le second défi a été de définir comment les processeurs doivent se communiquer leurs résultats afin de les réduire et les réunir sur un seul d'entre eux. Les sections qui suivent décrivent la façon dont ces problèmes ont été résolus lors du développement de la version *MPI* du *wordcount*.

5.1 Lecture parallèle d'un fichier

Il a fallu définir dans cette partie comment distribuer le fichier à lire entre les processeurs. L'idée étant de paralléliser au maximum l'exécution du programme, on a voulu faire en sorte que tous les processeurs lisent en parallèle un morceau de fichier, indépendamment les uns des autres.

Ceci n'est pas possible avec les fonctions standard pour la lecture de fichiers qui sont proposées par le *C++* (en utilisant un *std::ifstream* par exemple). Ces dernières ne permettent d'effectuer que des lectures séquentielles sur un fichier, et si un ensemble de processeurs doit accéder à des morceaux de fichier en même temps, ils sont obligés de le faire les uns après les autres, même si les parties qu'ils ont à lire ne se chevauchent pas. De plus, si on tente de faire lire un même fichier à plusieurs processeurs en même temps avec ces fonctions sans mettre en place un système de queue, aucune garantie ne peut plus être faite quant aux lecture et écritures effectuées par les processeurs (les accès concurrents n'étant pas gérés).

Heureusement, il existe dans *MPI* une *API* particulière appelée *MPI_IO*, spécialement prévue pour les lectures et écritures parallèles sur le système de fichiers d'une machine. *MPI_IO* propose un ensemble de fonctions permettant d'effectuer des opérations de lecture/écriture selon le même modèle que les *send* et *receive* de *MPI*. En particulier, il est possible d'indiquer aux processeurs d'un programme s'ils doivent exécuter ces opérations de façon bloquante ou non, et il existe même des primitives collectives pour ces actions. Les opérations collectives de *MPI_IO* sont en général les mieux optimisées pour les lectures/écriture parallèles sur un même fichier, bien que l'efficacité des optimisations soit très dépendante de leur implémentation sous-jacente et du système de fichier sur lequel les primitives sont exécutées. Les opérations collectives doivent toujours être appelées par tous les processeurs sur un même communicateur, et chaque processeur doit indiquer quel morceau du fichier il va lire (ou écrire). On ne peut donc pas effectuer de lecture « locale » avec une opération collective (c'est-à-dire qu'un processeur seul ne peut pas faire de lecture ou d'écriture avec).

Comme notre idée est ici de lire en parallèle des morceaux de fichier sur tous les processeurs de notre programme en même temps, et ce de la façon la plus optimisée possible, les primitives collectives de *MPI_IO* sont particulièrement bien adaptées. On veut que nos processeurs lisent tous un morceau de fichier de taille à peu près égale, et qu'ils ne traitent que des mots entiers (on ne veut pas qu'un morceau lu par un processeur s'arrête au milieu d'un mot). On a imaginé pour cela trois versions différentes de la lecture parallèle.

5.1.1 Première version

L'implémentation de la première version de la lecture parallèle se trouve dans le fichier *par_distribution.cpp*, dans le dossier *Source/MPI/distribution/V1* joint à ce travail. L'idée dans cette implémentation est que chaque processeur calcule localement sa position de départ approximative dans le fichier en divisant la taille de ce dernier par le nombre de processeurs du programme, de la façon suivante :

```
MPI_Offset filesize;
MPI_File_get_size(textfile, &filesize);
MPI_Offset localsize = filesize/nProc;
MPI_Offset localStart = myRank*localsize; // 'myRank' est le rang d'un processeur
```

La taille du fichier est obtenue ci-dessus à l'aide de la fonction *MPI_File_get_size()* de *MPI_IO*. On n'a donc pas besoin de gérer les aspect de concurrence entre processeurs pour cette action. Ensuite, tous les processeurs (sauf le premier, de rang 0) se placent sur leur position de départ dans le fichier, puis ils le lisent caractère par caractère jusqu'à ce qu'ils rencontrent un espace. Lorsque c'est le cas, ils s'arrêtent et déterminent que la position suivant l'espace rencontré est celle à laquelle commencera réellement leur morceau de fichier. Ils envoient donc cette position à leur voisin gauche dans le communicateur (processeur dont le rang est inférieur au leur de 1), qui utilisera cette dernière comme position de fin de sa partie de fichier, pour qu'il n'y ait pas de chevauchement entre les morceaux lus.

```
MPI_File_seek(textfile, localStart, MPI_SEEK_SET); // placement sur 'localStart'
if (myRank > 0){
	char c;
	while (c != ' ')
		MPI_File_read(textfile, &c, 1, MPI_CHAR, MPI_STATUS_IGNORE);
	MPI_File_get_position(textfile, &localStart);
	MPI_Send(&localStart, 1, MPI_LONG_LONG, myRank-1, 0, MPI_COMM_WORLD);
}
```

Une fois cela fait, tous les processeurs lisent en parallèle leur morceau de fichier :

```
MPI_Offset totalSize = localEnd-localStart+1;
char* text = (char*)malloc((totalSize+1)*sizeof(char));
MPI_File_read_at_all(textfile, localStart, text, totalSize, MPI_CHAR,
	MPI_STATUS_IGNORE);
text[totalSize] = '\0';
```

Il y a plusieurs problèmes avec l'implémentation proposée ci-dessus. Tout d'abord, les opérations de lecture caractère par caractère exécutées par chaque processeur pour trouver sa position de départ sont bloquantes. À chaque fois que deux processeurs veulent exécuter cette action, il faut donc que l'un des deux attende que l'autre ait terminé, ce qui représente un goulot d'étranglement pour le parallélisme. L'autre problème observé est que les actions de lecture dans le fichier représentent de nombreuses opérations d'accès au disque de la machine. Or, on veut justement minimiser la quantité de ces dernières pour gagner en performances. Pour ces deux raisons, on a imaginé une autre solution pour la lecture parallèle.

5.1.2 Seconde version

Le code de la seconde version pour la lecture parallèle se trouve dans le fichier *par_distribution.cpp,* dans le dossier *Source/MPI/distribution/V2/* joint à ce document. L'implémentation proposée ici est très largement inspirée de la réponse de l'utilisateur Jonathan Dursi à une question sur le forum *StackOverflow* ([19]).

Dans cette implémentation, tous les processeurs commencent par calculer leur positions approximatives de début et de fin pour la lecture dans le fichier, de la même façon que dans la première version. Toutefois, ils ajoutent en plus un nombre fixe d'octets à leur position de fin, afin que les morceaux de fichier qu'ils lisent se chevauchent deux à deux. Seul

le processeur de rang maximum n'ajoute pas d'*overlap* après sa position de fin, puisqu'il lit déjà un morceau se terminant au dernier octet du fichier à traiter.

```
MPI_Offset start = myRank*localsize;
MPI_Offset end = start+localsize-1;
end += OVERLAP; // 'OVERLAP' est une valeur entière fixée.
if (myRank == nProc-1)
	end = filesize;

localsize = end-start+1;
```

Une fois cela fait, les processeurs lisent tous le morceau de fichier qui se trouve entre les positions qu'ils ont calculé dans une chaîne de caractères, et ils peuvent fermer le fichier dans lequel ils ont effectué la lecture.

```
char *chunk = (char*)malloc((localsize+1)*sizeof(char));
MPI_File_read_at_all(textfile, start, chunk, localsize, MPI_CHAR,
	MPI_STATUS_IGNORE);
chunk[localsize] = '\0';
MPI_File_close(&textfile);
```

Finalement, les processeurs parcourent leur chaîne de caractères depuis son départ, jusqu'à ce qu'ils rencontrent un espace. Une fois cela fait, il déterminent que le caractère suivant cet espace sera le premier de leur partition de fichier réelle, et ils suppriment tout ce qui se trouvait avant. De la même façon, ils parcourent leur partition depuis sa position de fin moins la valeur d'*overlap* qu'ils ont ajouté, et le premier espace rencontré devient le dernier caractère de leur partition. Tout le texte qui se trouve après est ignoré, puisqu'il sera traité par le processeur de rang directement supérieur.

```
int localStart = 0, localEnd = localsize;
if (myRank != 0){
	while (chunk[localStart] != ' ') localStart++;
	localStart++;
}
if (myRank != nProc-1){
	localEnd -= OVERLAP;
	while (chunk[localEnd] != ' ') localEnd++;
}
localsize = localEnd-localStart+1;
localContent.write(&(chunk[localStart]), localsize-1);
free(chunk);
```

La solution proposée ci-dessus présente plusieurs avantages par rapport à la précédente. Tout d'abord, les processeurs n'effectuent qu'un seul accès au disque pour la lecture de leur partition de fichier, lorsqu'ils récupèrent leurs morceaux de texte. Toutes les opérations qui suivent se font alors directement sur des chaînes de caractères qui se trouvent dans leur mémoire centrale, ce qui est beaucoup moins coûteux en termes de performances. De plus, les processeurs n'ont ici besoin d'effectuer aucune communication avec *MPI*, puisque chacun détermine localement quelle est sa partition réelle de fichier. Si l'*overlap* ajouté au départ est suffisamment grand, tous les processeurs liront bien des morceaux contigus de fichier, sans qu'il n'y ait de chevauchement entre-eux.

La solution présente toutefois aussi quelques problèmes qui justifient l'implémentation d'une troisième et dernière version de la lecture parallèle. Si l'*overlap* ajouté après les positions de fin des processeurs est trop petit, il se peut qu'un ou plusieurs d'entre-eux ne rencontrent pas d'espace dans la partie qui se chevauche avec le morceau de leur voisin de droite. Dans ce cas, ces processeurs termineront leur partition sur un mot incomplet, et comme leur voisin de rang supérieur lit son morceau depuis le début jusqu'à ce qu'il rencontre un espace, les partitions lues par les deux voisins ne seront pas contigües (il y aura un morceau de fichier non-lu entre les deux partitions). Si au contraire un *overlap* trop important est ajouté, les processeurs du programme liront tous des morceaux très grands de fichier, ce qui n'est pas optimal en termes d'usage de la mémoire et d'accès au disque.

Enfin, l'implémentation proposée ici présente un autre problème lié aux lectures parallèles avec *MPI_IO*. En effet, on a vu que la fonction *MPI_File_read_at_all* de *MPI_IO* était particulièrement bien optimisée pour les lectures simultanées dans un fichier par plusieurs processeurs. Toutefois, comme les processeurs lisent ici des morceaux de fichiers qui se chevauchent, ils entrent potentiellement en conflit lors de la lecture de leur partition, ce qui pourrait empêcher de bien paralléliser cette dernière. On ne peut donc plus être certains que les optimisations proposées par l'implémentation de *MPI_IO* auront encore un quelconque effet ici.

5.1.3 Troisième version

La troisième et dernière version de la lecture parallèle se trouve dans le fichier *par_distribution.cpp*, situé dans le dossier *Source/MPI/distribution/V3/* joint à ce document. L'implémentation est basée sur une idée proposée par le Dr. Jean-Luc Falcone de l'université de Genève.

Tout comme dans les deux versions précédentes, les processeurs calculent ici les positions approximatives de début et de fin de leurs morceaux dans le fichier à l'aide de la taille en octets de ce dernier, du nombre de processeurs participant au programme et de leur rang. Contrairement à la seconde version de la lecture parallèle, aucun *overlap* n'est ajouté ici. Une fois cela fait, les processeurs lisent la partition de fichier entre les positions calculées dans une chaîne de caractères.

```
MPI_Offset filesize;
MPI_File_get_size(textfile, &filesize);
MPI_Offset localsize = filesize/nProc;

MPI_Offset start = myRank*localsize;
MPI_Offset end = start+localsize-1;
if (myRank == nProc-1)
       end = filesize;
localsize = end-start+1;

char *chunk = (char*)malloc((localsize+1)*sizeof(char));
MPI_File_read_at_all(textfile, start, chunk, localsize, MPI_CHAR,
       MPI_STATUS_IGNORE);
chunk[localsize] = '\0';
```

Ensuite, chaque processeur lit tous les caractères depuis le début de sa partition de fichier jusqu'au premier espace rencontré. Il les retire de sa partition et les stocke dans une chaîne de caractères qu'il envoie à son voisin gauche (le processeur de rang directement inférieur au sien, qui lit la partition juste avant la sienne). Le voisin qui reçoit la chaîne la concatène à la fin de sa partition. De cette façon, si la partition d'un processeur commençait au milieu d'un mot, le morceau de mot est retiré de la partition, et il est recollé à son autre moitié à la fin de la partition du processeur de rang inférieur. Tout ceci est fait par l'intermédiaire d'objets de type *std::stringstream*.

```
std::stringstream tmpStream(chunk);
// Libération de la mémoire de la chaîne de caractère 'chunk', maintenant inutile.
free(chunk);
if (myRank > 0){
	std::string first;
	getline(tmpStream, first, ' ');
	send(first, myRank-1, 0, MPI_COMM_WORLD);
}
localContent << tmpStream.rdbuf();

if (myRank < nProc-1){
	std::string last;
	recv(last, myRank+1, 0, MPI_COMM_WORLD);
	localContent << last;
}
```

Par rapport aux deux versions précédentes, celle-ci possède l'avantage de ne lire qu'une seule fois dans le fichier sur le disque, et de le faire sans que les partitions lues par les processeurs ne se chevauchent. Ainsi, on a minimisé autant que possible le nombre d'accès au disque par les processeurs, et on s'attend en plus à ce que l'accès se fasse de façon aussi optimale que possible grâce aux primitives collectives de *MPI_IO*.

5.1.4 Comparaison des trois versions

On a effectué quelques mesures de performances sur les trois méthodes de lecture parallèle pour les comparer. Les mesures ont été effectuées sur la machine parallèle *Scylla* de l'*UNIGE*, pour les trois fichiers *large3.txt*, *large1.txt* et *verylarge1.txt*.

large3.txt et *large1.txt* sont deux fichiers qui ont été générés en concaténant plusieurs dizaines de fois un fichier appelé *large.txt* avec lui-même, afin obtenir des fichiers de tailles de 311 méga-octets et 622 méga-octets respectivement. l*arge.txt* contient une dizaine d'oeuvres littéraires en anglais récupérées sur le site du *Gutenberg Project* ([20]) et concaténées les unes avec les autres. v*erylarge1.txt* est un fichier d'environ 1.15 giga-octets qui a été généré en concaténant plusieurs fois le fichier *verylarge.txt* avec lui-même. *verylarge.txt* correspond au fichier *large.txt* présenté plus haut, mais auquel une dizaine d'oeuvres supplémentaires ont été ajoutées.

Les résultats obtenus pour les mesures sont présentés dans la *figure 2*. On a mesuré le temps moyen d'exécution de chaque version de la lecture parallèle sur 5 *runs*, pour 2, 4, 8, 24, 36 et 48 processeurs, puis la courbe des temps d'exécution de chaque méthode a été affichée.

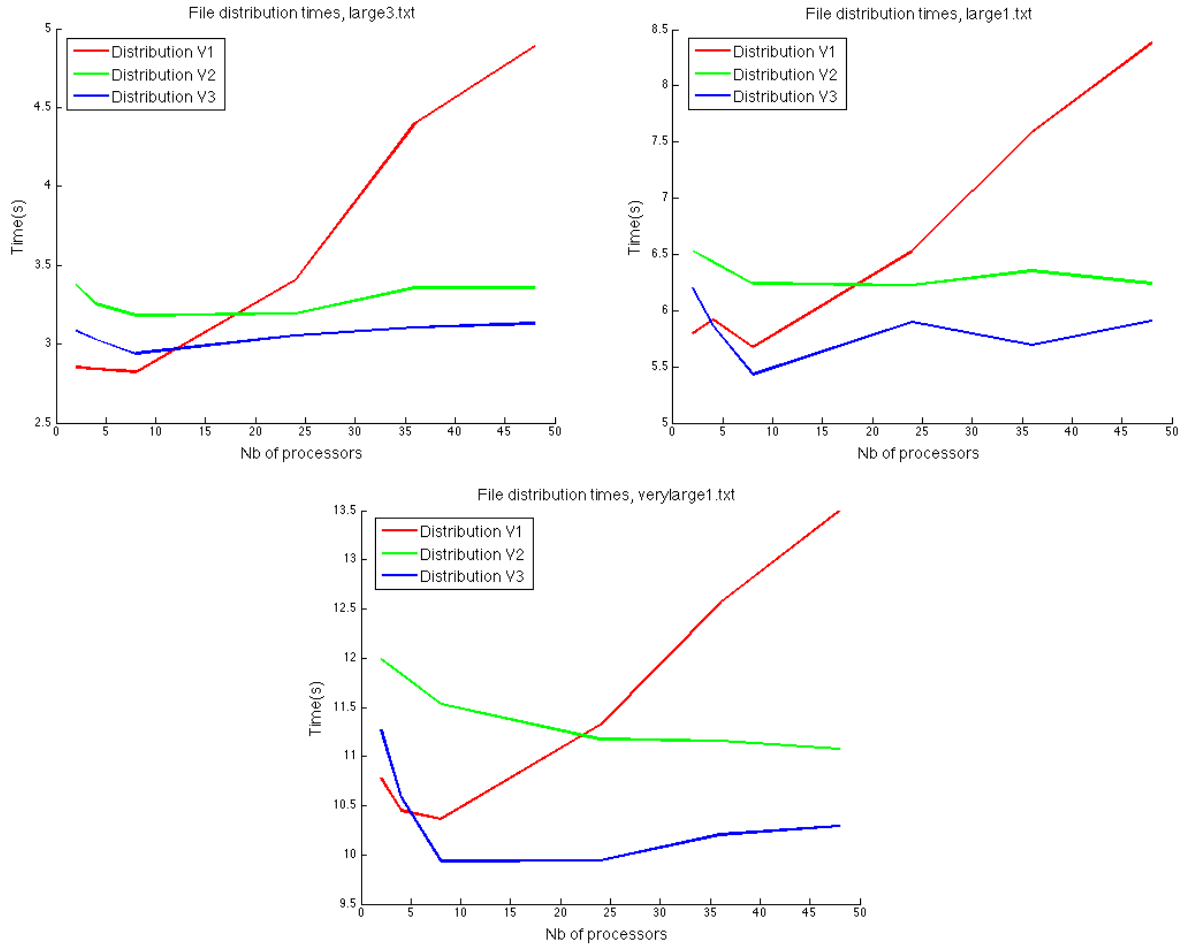

Figure 2 – Mesures du temps d'exécution des 3 méthodes de distribution d'un fichier

On voit sur la *figure 2* que la méthode la plus rapide est en général la dernière des trois versions proposées ci-dessus. Comme la première version effectue de nombreuses lectures bloquantes entre processeurs dans le fichier, on observe que cette dernière n'est pas du tout *scalable* et devient très inefficace lorsqu'un grand nombre de processeurs est utilisé. La seconde version n'est que légèrement moins bonne que la dernière, mais on observe malgré tout une différence de temps qui justifie le choix de la troisième pour les mesures de performances ultérieures dans ce travail.

On note aussi sur les graphiques que les deux dernières versions semblent prendre des temps à peu près constants pour la lecture parallèle. On aurait pu s'attendre à ce que la lecture soit toujours plus rapide au fur-et-à-mesure que le nombre de processeurs augmente, puisque le morceau de fichier que chacun d'entre-eux doit lire est alors plus petit. La raison pour laquelle ce n'est pas le cas ici est très probablement liée au système de fichier de la machine *Scylla* et à la compatibilité des optimisations de *MPI_IO* avec ce dernier. On verra plus loin

qu'avec un autre système de fichiers, les temps de lectures diminuent plus significativement lorsque beaucoup de processeurs sont utilisés.

5.2 Comptage des mots en parallèle

Cette partie de l'implémentation du *wordcount* avec *MPI* fonctionne exactement comme dans la version séquentielle du programme : chaque processeur appelle simplement la fonction *count_words* du fichier *useful_functions.cpp* sur sa partition de ficher. La seule différence avec la version séquentielle est que les processeurs traitent seulement ici une partie du fichier au lieu de son intégralité.

5.3 Réduction des résultats vers un seul processeur

Pour cette partie du programme, il a non seulement fallu définir comment effectuer la réduction des résultats vers un seul processeur avec *MPI*, mais aussi comment implémenter les envois de *maps* (ou *unordered maps*) entre processeurs, puisque *MPI* ne fournit pas de fonctions pré-définies pour l'envoi et la réception d'objets de haut-niveau de ce type.

5.3.1 Réduction des résultats

Pour la réduction des résultats du *wordcount*, on a utilisé un modèle dans lequel les processeurs participant au programme combinent deux à deux leurs résultats, jusqu'à ce qu'un seul d'entre-eux possède la combinaison de tous les résultats des autres dans sa mémoire. Ceci correspond au modèle de réduction dans un hypercube.

L'implémentation de la réduction se trouve directement dans le fichier *wordcount_parallel.cpp* de ce travail. Son fonctionnement est expliqué ci-dessous :

1) Après le comptage des mots dans leur partition de fichier, tous les processeurs possèdent en mémoire une collection de paires clé-valeur *wordCounts*, contenant la fréquence des mots dans leur morceau de fichier.

2) Au départ, tous les processeurs définissent le nombre de processeurs « actifs » dans la réduction comme le nombre total de processeurs du programme. Le nombre de « receveurs » est défini comme la moitié du nombre de processeurs « actifs ».

3) Tant que le nombre de processeurs « actifs » est plus grand que 1 :

- Si le rang d'un processeur est plus petit que le nombre de processeurs « actifs » ET plus grand ou égal à celui de « receveurs », il envoie sa collection *wordCounts* au processeur dont le rang est égal au sien moins le nombre de « receveurs ».

- Si le rang d'un processeur est plus petit que le nombre de « receveurs », il reçoit sa collection du processeur dont le rang est supérieur au sien de la valeur du nombre de « receveurs ». Il combine cette collection avec la sienne en y ajoutant les mots de la collection reçue qu'il n'avait pas vu, et en sommant les fréquences des mots qu'il avait déjà vu avec les siennes.

- Tous les processeurs divisent le nombre d'« actifs » par 2, et le nombre de « receveurs » est fixé comme la moitié du nombre d'« actifs ».

Avec la méthode décrite ci-dessus, la réduction des résultats se fait en $log_2(P)$ étapes, où P est le nombre total de processeurs dans le programme. À chaque étape, la moitié des processeurs qui n'ont pas encore partagé leur résultat envoient ce dernier aux autres, puis ils deviennent « inactifs » dans la réduction. Ceux qui reçoivent les collections des autres la combinent avec la leur, puis la moitié d'entre-eux enverra cette dernière à l'autre moitié lors de l'étape suivante. Tout ceci est répété jusqu'à ce qu'un seul processeur ait récupéré la combinaison de tous les résultats.

Le code *C++* pour la méthode définie ici est donné plus bas. Il gère la possibilité que le nombre total de processeurs du programme ne soit pas une puissance de deux. Dans ce cas, lors de certaines étapes de la réduction, il y aura un nombre impair de processeurs actifs, et il faudra que l'un d'entre-eux passe un tour sans partager son résultat avec un autre.

```
int activeProcs(nProc);
int receivers((nProc+1)/2);
int nb_recv_odd(nProc%2);

while (activeProcs>1){
	if (myRank >= receivers && myRank < activeProcs){
		send(wordCount, myRank-receivers, 0, MPI_COMM_WORLD);
	}
	else if (myRank < receivers){
		// Gestion du cas où le nb de processeurs actifs est impair.
		if (nb_recv_odd==0 || (nb_recv_odd!=0 && myRank < receivers-1)){
			unordered_map<string, int> recvMap;
			recv(recvMap, myRank+receivers, 0, MPI_COMM_WORLD);
			merge(wordCount, recvMap);
		}
	}

	activeProcs = (activeProcs+1)/2;
	receivers = (activeProcs+1)/2;
	nb_recv_odd = activeProcs%2;
}
```

La fonction *merge* utilisée dans le code ci-dessus combine simplement deux *std::unordered_map* comme cela a été expliqué dans la description de la méthode de réduction plus haut. Les fonctions *send* et *recv* ont été définies pour permettre d'envoyer des objets de type *std::map* ou *std::unordered_map* avec *MPI*, comme nous allons le voir dans la section suivante.

5.3.2 Envoi et réception de maps avec MPI

Afin de permettre l'envoi et la réception de *maps* et d'*unordered maps* avec *MPI*, on a implémenté des fonctions basées sur l'*API* de trois façons différentes. Elles implémentent toutes la même interface, définie dans le fichier *mpi_functions.h* joint à ce document. Les trois versions se trouvent respectivement dans les dossiers *V1/*, *V2/* et *V3/* du dossier *Source/MPI/send* de ce travail.

Dans la première version des fonctions *send* et *recv* pour les *maps* (ou les *unordered maps*), on parcourt simplement les entrées de la structure de données une à une, et chaque

paire est envoyée et reçue indépendamment des autres, avec les fonctions de base *MPI_Send* et *MPI_Recv* de *MPI*. C'est la façon la plus simple d'effectuer la communication d'un *map* entre deux processeurs, mais aussi la moins efficace. En effet, si le *map* envoyé est très grand, un nombre très important d'opérations d'envoi et de réception sont faites à la suite à travers le réseau d'interconnexion de la machine. On a alors un temps de latence accumulé pour toutes les communications qui devient très grand.

Afin d'éviter ce temps de latence, on a implémenté deux autres versions du *send* et du *receive*, dans lesquelles la communication du *map* entre les processeurs se fait avec une seule opération d'envoi et de réception. Pour rendre cela possible, on a d'abord appliqué une opération de sérialisation sur le *map*, pour le transformer en une simple chaîne d'octets (chaîne de caractères), qui est ensuite envoyée à travers le réseau en une seule étape. À la réception, la chaîne d'octet est dé-sérialisée en lui appliquant l'opération inverse de celle pour la sérialisation.

La seule différence entre la deuxième et la troisième version du *send/receive* de *maps* avec *MPI* réside dans la façon dont sont sérialisés les structures avant d'être envoyées (puis dé-sérialisés lorsqu'elles sont reçues). Dans la seconde version, on a écrit notre propre implémentation de la sérialisation, alors que dans la troisième, la bibliothèque *open source Cereal* ([21]) a été utilisée. *Cereal* offre un ensemble de classes et de méthodes pour sérialiser et dé-sérialiser facilement différents types de structures de données.

Afin de comparer les trois façons de faire, on a mesuré les temps pris par chaque méthode pour envoyer les *unordered maps* dans la phase de réduction de l'algorithme du *wordcount* parallèle. Les mesures ont été effectuées sur la machine *Scylla* de l'UNIGE, avec les fichiers *large3.txt*, *large1.txt* et *verylarge1.txt*. À chaque fois, les mesures ont été faites pour 2, 4, 8, 24, 36 et 48 processeurs. Les temps d'envoi/réception ont été mesurés en secondes, et on a calculé le temps moyen pris sur 5 exécutions pour chaque nombre de processeurs. Les résultats sont présentés dans la *figure 3* sous forme de courbes des temps d'exécution par rapport au nombre de processeurs.

On voit bien dans la *figure 3* qu'un gain considérable de temps est obtenu lorsque les deuxième et troisième versions du *send/recv* sont utilisées. Les temps de latence dans la première version représentent donc bien un *overhead* important à éviter. Pour les mesures de *speedup* dans la section suivante de ce travail, c'est la seconde version des envois et réceptions de *maps* qui a été utilisée, puisque c'est elle qui offre les meilleures performances, comme le montrent les résultats obtenus ici.

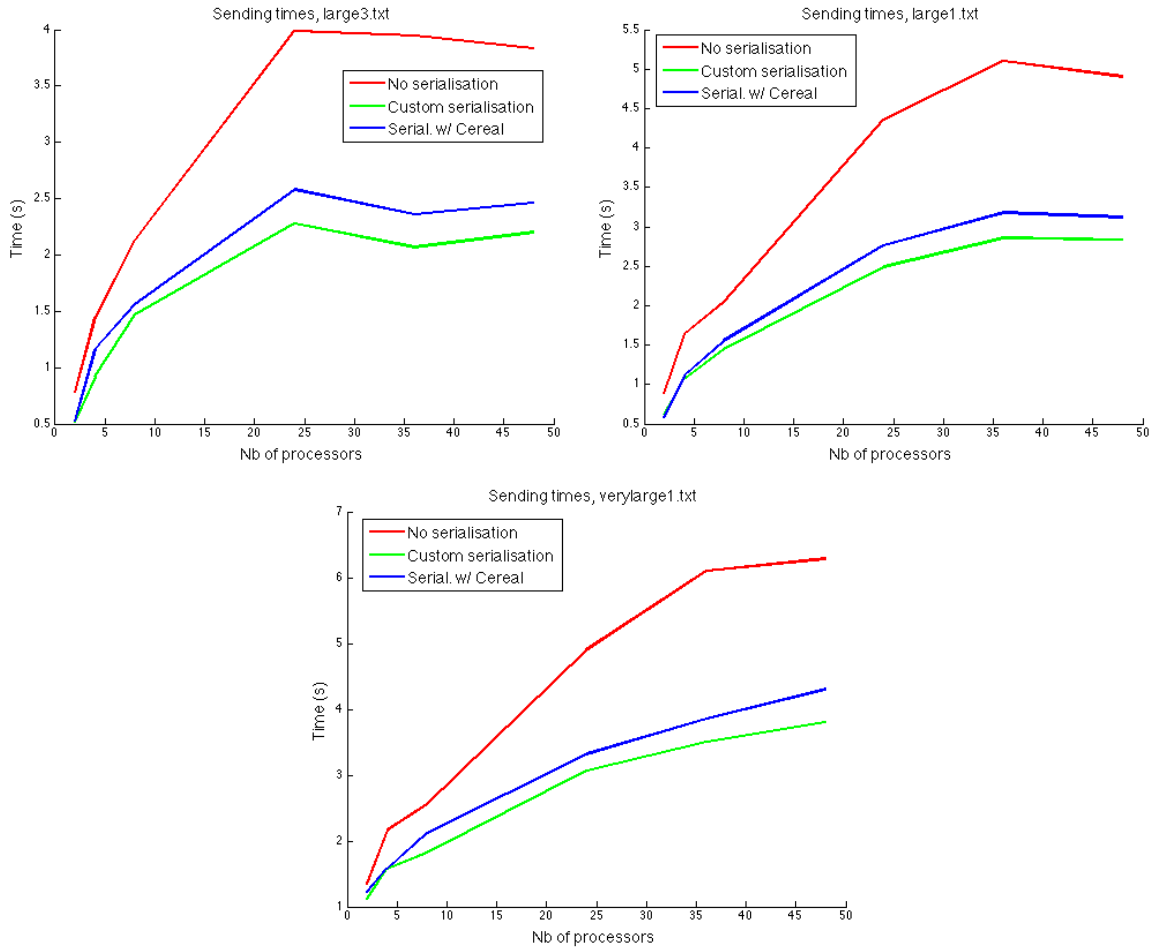

Figure 3 – Temps moyens d'exécution pour la communication de maps avec les 3 versions différentes du send/receive

5.3.3 Modèle de performances pour la réduction

Afin d'avoir une meilleure idée du modèle mathématique suivi par l'opération de réduction des résultats dans notre programme, on a tenté de définir un modèle de performances pour cette dernière. Pour cela, on a effectué le « *fitting* » d'une fonction mathématique sur les mesures du temps d'exécution de la réduction. Ces mesures ont été obtenues en utilisant la seconde version des *send/recv* présentée plus haut. Elles ont été faites sur le temps cumulé pour les $log_2(P)$ étapes de la réduction (avec P le nombre de processeurs), en comptant à chaque fois le temps pris par les processeurs pour la réception et pour la fusion des *maps.*

On a utilisé ici les résultats de mesures obtenues sur les trois fichiers *large3.txt*, *large1.txt* et *verylarge1.txt,* décrits précédemment dans la section 5.1.4. Il est intéressant de noter que, comme ces fichiers consistent tous en la concaténation d'un seul fichier avec lui-même plusieurs fois à la suite, tous les processeurs du programme traitent ici des partitions de fichier contenant un texte qui est semblable. Les ensembles de mots lus par tous les processeurs dans le programme sont donc les mêmes après le comptage des fréquences dans le texte. Ceci implique que les tailles des *maps* obtenus par tous les processeurs sont les

mêmes dès le départ de la réduction, et qu'elles ne grandissent pas à chaque étape, lorsque les *maps* sont fusionnés. La taille des *maps* est donc maximale dès le début, et on peut considérer que le modèle de performances obtenu ici avec le *fitting* correspond à une borne supérieure du temps pouvant être pris pour la réduction.

La fonction dont le *fitting* a été effectué sur nos résultats est celle représentant le temps pris pour la réduction, notée *T(P)*. Si on pose que la constante *a* est le temps pris à chaque étape de la réduction pour qu'un processeur reçoive le *map* d'un autre et le fusionne avec le sien, et *b* est un temps d'initialisation au début de la réduction, alors on peut exprimer notre fonction comme :

$$T(P) = a \log_2(P) + b.$$

a est multiplié par $\log_2(P)$ puisque c'est le nombre d'étapes de la réduction, et on peut aisément supposer que c'est une constante, puisque la taille des *maps* ne croît pas à chaque étape dans notre cas. Les résultats du *fitting* se trouvent dans la *figure 4*. On observe dans cette dernière que la fonction utilisée semble bien correspondre aux mesures réelles.

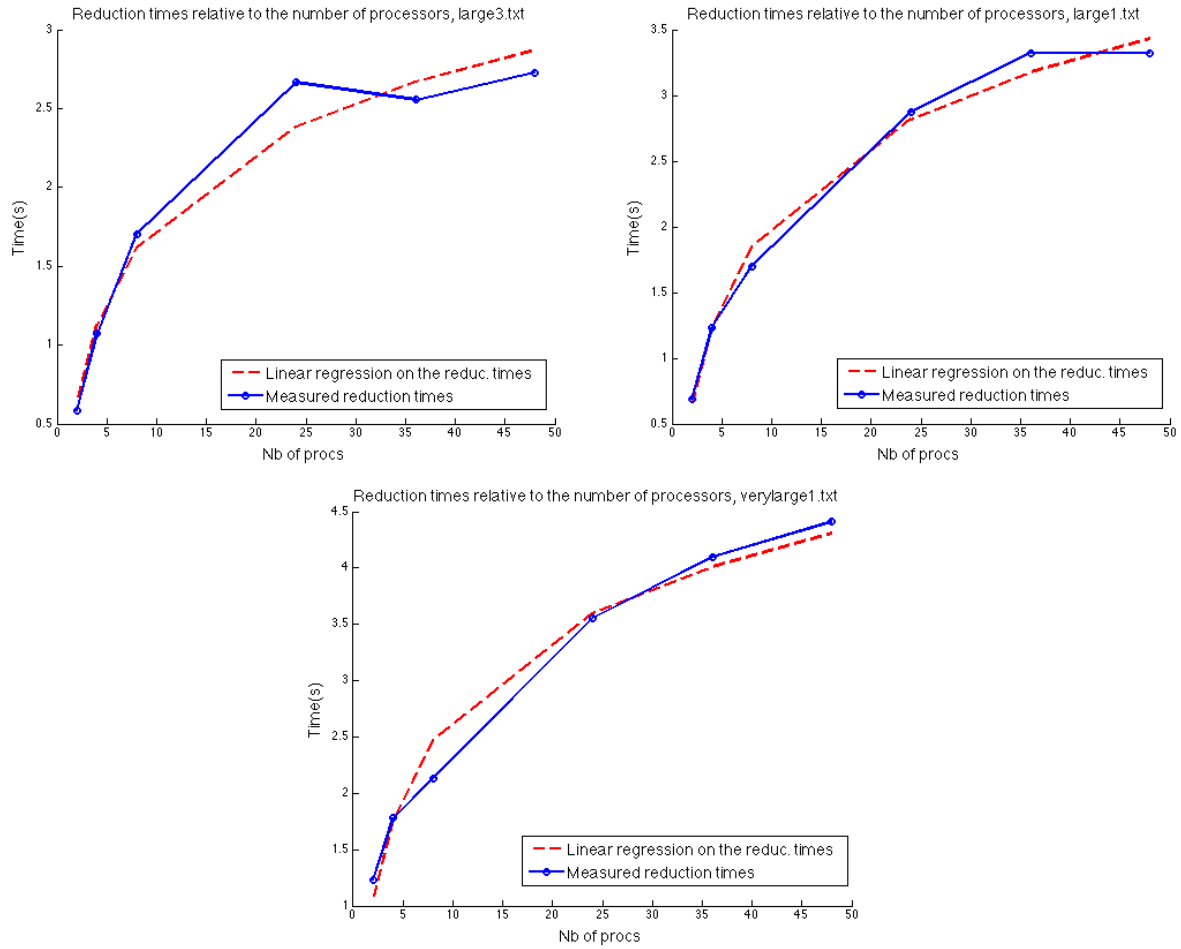

Figure 4 – Régression linéaire de T(P) sur les mesures du temps pris par la réduction, dans le programme du wordcount parallèle

Les valeurs des paramètres *a* et *b* trouvées pour chacun des graphiques de la *figure 4* sont les suivantes : *a = 0.4858* et *b = 0.1576* pour le fichier *large3.txt* (311 Mo), *a = 0.6121* et *b = 0.0133* pour le fichier *large1.txt* (622 Mo), et enfin *a = 0.7108* et *b = 0.3384* pour le fichier *verylarge1.txt* (1.15 Go). Comme indiqué plus haut, le paramètre *a* peut s'interpréter comme le temps nécessaire en secondes pour l'envoi d'un *map* d'un processeur à un autre lors d'une étape de la réduction, et *b* est le temps d'initialisation au début de la phase de réduction. On observe dans les valeurs données ici que *a* croît lorsque la taille du fichier traité devient plus grande. Or, on sait aussi que la taille du *map* que les processeurs doivent s'envoyer à chaque étape de la réduction est plus grande lorsque le fichier traité est plus grand, puisque ce dernier contient plus de mots différents (ce qui implique qu'il y a plus de clés différentes dans les *maps* que les processeurs s'envoient). Ainsi, on voit bien que le temps d'envoi des *maps* entre les processeurs lors de la réduction croît avec leur taille, comme on s'y attendait (on a en revanche trop peu de mesures pour déterminer si la croissance est linéaire ou si elle suit un autre modèle).

6. Mesures de performances et speedups

Toutes les mesures présentées dans cette section ont été effectuées sur la même machine, le cluster *Baobab* de l'UNIGE [22]. Les valeurs indiquées correspondent toujours au temps moyen mesuré en secondes sur 5 exécutions d'un même programme.

Les programmes ont été exécutés sur plusieurs fichiers texte de tailles différentes, afin de déterminer les performances pouvant être obtenues sur des tailles de problèmes variables. Les fichiers suivants ont été utilisés : *large1.txt* (622 Mo), *verylarge1.txt* (1.15 Go), *wikipedia_part.txt* (5.39 Go) et *wikipedia.txt* (environ 18 Go). Les deux derniers fichiers mentionnés ont été obtenus à partir du dernier « *dump* » en anglais de *Wikipedia*, disponible sur le site de l'encyclopédie au format *xml* [23]. Pour *wikipedia_part.txt*, seule une partie du fichier de *dump* a été sélectionnée, puis cette dernière a été pré-traitée pour supprimer toutes les balises *xml* et les hyperliens du document, ainsi que pour séparer tous les mots du texte de la ponctuation (par exemple, les occurrences de « *mot.* » ont été transformées en « *mot* . », pour éviter que deux mots identiques ne soient traités différemment lors du comptage parce que l'un d'entre-eux est collé à un symbole de ponctuation). Le fichier *wikipedia.txt* a été obtenu de la même façon, mais en conservant un plus gros morceau de l'archive *Wikipedia*.

Afin de déterminer quel niveau de « scalabilité » peuvent offrir les versions parallèles du *wordcount*, on a effectué les mesures de performances sur ces dernières en utilisant différents nombres de processeurs.

6.1 Performances de la version séquentielle

Pour la version séquentielle du *wordcount*, on a mesuré trois valeurs : le temps pris par le programme pour charger et lire le fichier, le temps pour compter les mots dans le texte et le temps total d'exécution. Les résultats obtenus sont présentés dans le tableau de la *figure 5*.

Comme on pouvait s'y attendre, plus un fichier est gros, plus les temps mesurés pour son traitement sont longs.

Fichier traité	***large1.txt* (622 Mo)**	***verylarge1.txt* (1.15 Go)**	***wikipedia_part.txt* (5.39 Go)**	***wikipedia.txt* (18 Go)**
Temps moyen de lecture du fichier	0.9431	1.84	11.39	38.05
Temps moyen pour le comptage de mots	30.79	59.58	294.93	974.06
Temps total moyen d'exécution	31.73	61.42	306.32	1012.1

Figure 5 – Table des temps moyens mesurés en secondes pour le wordcount séquentiel en C++

6.2 Performances avec Spark

Pour lancer les mesures de performances avec *Spark* sur le *cluster Baobab*, des scripts pour le déploiement d'un programme *master* et de *workers* sur les nœuds de la machine ont été utilisés. Ils se trouvent dans les fichiers *spark_master.sh* et *spark_slave.sh*, dans le dossier *Source/Spark/deployment_scripts* de ce travail. Le dossier contient aussi un script pour lancer une application *Spark* sur un nœud *master* (*spark_app.sh*).

Ici, seul le temps d'exécution total du *wordcount* a été mesuré, car *Spark* utilise le modèle de « *lazy evaluation* » sur ses fonctions. Ainsi, lorsqu'il est indiqué dans le code qu'un fichier doit être chargé et lu en mémoire, cette action n'est en réalité effectuée par le programme que lorsque les données du fichier sont utilisées dans un calcul devant retourner une valeur. Dans le *wordcount*, le chargement en mémoire des données du fichier est donc fait au même moment que le comptage des mots et la réduction des résultats vers un seul processeur. On ne peut donc pas différencier les temps pris par chacune des parties du programme, à moins d'introduire des opérations inutiles pour forcer la lecture du fichier à un moment donné, mais de telles opération prendraient du temps et fausseraient les mesures.

Il est aussi important de noter que les mesures effectuées dans ce travail pour *Spark* n'ont pas pu être faites dans les conditions idéales pour le bon fonctionnement du *framework*, puisque *Hadoop* n'est pas installé sur *Baobab* et que le système de fichier utilisé par le cluster n'est pas celui qui est privilégié par *Spark* (le *HDFS)*. Toutefois, les résultats obtenus permettent de se faire une idée du type de performances pouvant être observées lorsque *Spark* est utilisé sur un *cluster* utilisant *slurm* (le gestionnaire de tâches installé sur *Baobab*). De plus, on peut supposer que *Spark* a bénéficié du degré de parallélisme offert par le système de fichier de *Baobab* (le *BeeGFS* [24]), puisque ce dernier fonctionne selon un modèle distribué similaire à celui du *HDFS.*

Les temps d'exécution du *wordcount Spark* ainsi que ses valeurs de *speedup* par rapport à la version séquentielle sont donnés dans les tableaux de la *figure 6.* Le *speedup* en fonction du nombre de processeurs est ensuite représenté sous forme de graphiques dans la *figure 7.*

Fichier : *large1.txt* (622 Mo)						
Nb de proc.	2	4	8	24	48	64
Temps moyen d'exécution	114.73	69.02	47.83	37.88	36.24	35.83
Speedup	0.28	0.46	0.66	0.84	0.88	0.89

Fichier : very*large1.txt* (1.15 Go)						
Nb de proc.	2	4	8	24	48	64
Temps moyen d'exécution	199.66	115.18	75.27	43.11	35.05	31.15
Speedup	0.31	0.53	0.82	1.42	1.75	1.97

Fichier : *wikipedia_part.txt* (5.39 Go)						
Nb de proc.	4	8	24	48	64	128
Temps moyen d'exécution	432.29	261.82	161.59	122.05	99.08	86.61
Speedup	0.71	1.17	1.9	2.51	3.09	3.54

Fichier : *wikipedia.txt* (18 Go)					
Nb de proc.	16	32	64	128	256
Temps moyen d'exécution	493.45	418.02	389.16	198.08	200.62
Speedup	2.05	2.42	2.6	5.11	5.04

Figure 6 – Tables des temps moyens d'exécution en secondes pour le wordcount Spark et valeurs de speedup

Les résultats présentés ici semblent confirmer les conclusions faites par les auteurs de l'article « *Scalability ! But at what COST ?* » *[17]* : en dessous d'un certain nombre de processeurs ou d'une certaine taille de problème à traiter (ici, une certaine taille de fichier), il n'est pas véritablement intéressant d'utiliser *Spark*, car il offre des performances inférieures à la version séquentielle du *wordcount* en *C++*. Ceci est très probablement dû à un *overhead* important lié à la distribution de tâches entre les processeurs qui est faite par le *framework*, et sur laquelle le programmeur n'a pas de contrôle.

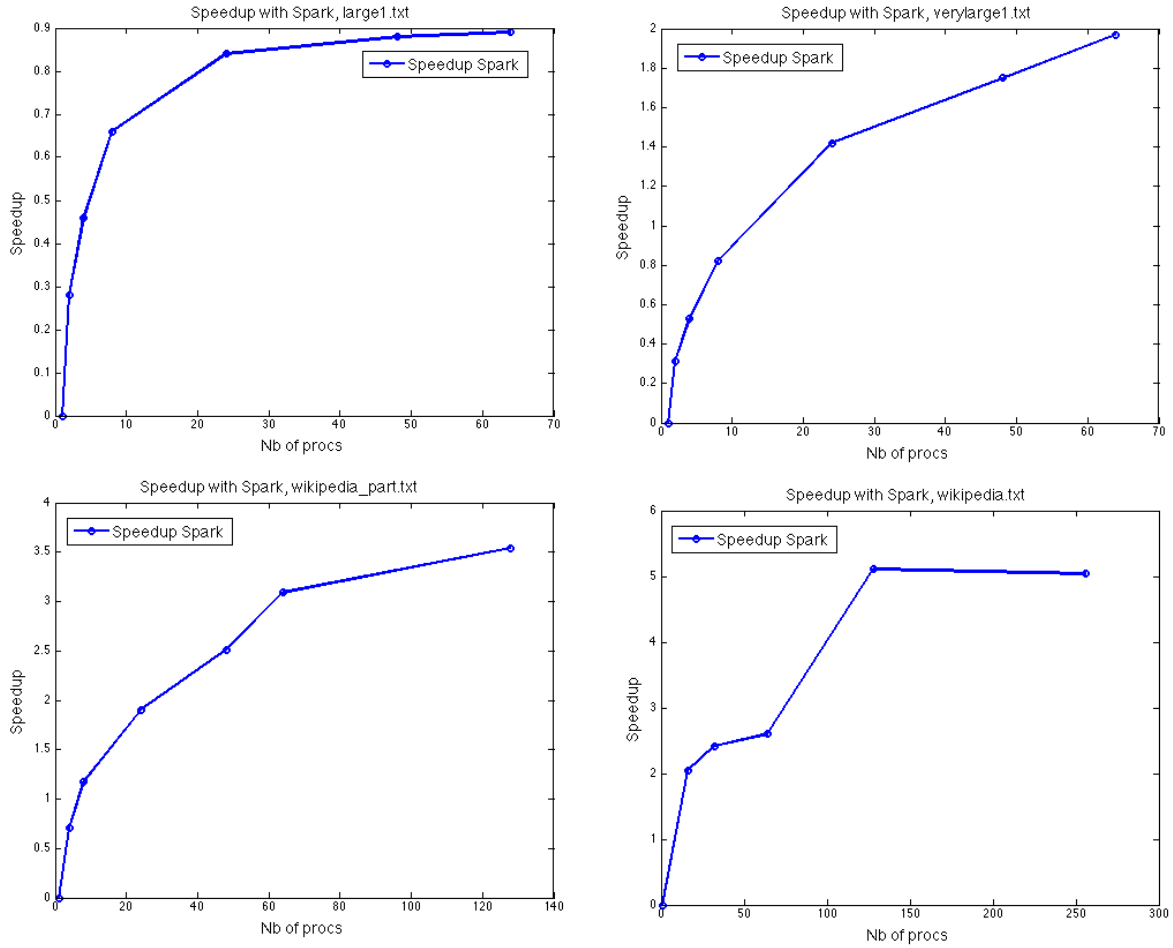

Figure 7 – Speedup du wordcount Spark par rapport à la version séquentielle

6.3. Performances avec MPI

Pour le *wordcount* avec *MPI*, on a mesuré 4 valeurs de temps différentes pour chaque fichier traité et chaque nombre de processeurs : le temps de chargement et de lecture du fichier, le temps pour le comptage des mots, le temps pris pour la réduction des résultats et enfin le temps total d'exécution du programme.

Les résultats des mesures de temps ainsi que les *speedups* obtenus par rapport à la version séquentielle sont présentés dans les tables de la *figure 8*. Dans la *figure 9*, les valeurs de *speedup* obtenues avec *MPI* par rapport à la version séquentielle du *wordcount* sont représentées sous forme de graphiques.

On a aussi effectué le « *fitting* » d'une loi d'Amdahl sur les courbes de *speedup* de la *figure 9*. Pour rappel, la loi d'Amdahl est une formule mathématique visant à associer un modèle de performances théorique au *speedup* obtenu pour un programme parallèle. Dans la loi d'Amdahl, l'hypothèse est faite qu'une portion α du travail à effectuer dans le programme n'est pas parallélisable. Le *speedup S* est alors exprimé selon la formule :

$$S = \frac{1}{\alpha + \frac{1+\alpha}{p}} \leq \frac{1}{\alpha}$$

où p est le nombre de processeurs utilisés. L'utilité de cette loi est qu'elle permet de déterminer la borne supérieure théorique du *speedup* d'un programme parallèle : on sait qu'il ne dépassera jamais la valeur 1/α.

Fichier : *large1.txt* (622 Mo)						
Nb de proc.	2	4	8	24	48	64
Temps de lecture	0.77	0.52	0.39	0.49	0.54	0.47
Speedup lecture	1.22	1.82	2.4	1.91	1.74	2.02
Temps pour le comptage	15.44	7.53	4.51	1.56	0.87	0.63
Speedup comptage	1.99	4.09	6.83	19.75	35.22	49.1
Temps réduc.	0.224	0.754	0.99	1.51	1.8	1.5
Temps total	16.45	8.83	5.95	3.65	3.33	2.67
Speedup total	1.93	3.59	5.34	8.7	9.54	11.87

Fichier : *verylarge1.txt* (1.15 Go)						
Nb de proc.	2	4	8	24	48	64
Temps de lecture	1.46	0.96	0.69	0.62	0.61	0.53
Speedup lecture	1.26	1.92	2.67	2.96	2.98	3.44
Temps pour le comptage	29.41	16.79	8.06	2.96	1.45	1.09
Speedup comptage	2.03	3.55	7.39	20.1	41.2	54.9
Temps réduc.	0.32	0.62	1.2	1.85	2.2	2.14
Temps total	31.2	18.41	10.01	5.53	4.39	3.88
Speedup total	1.97	3.34	6.14	11.09	13.98	15.84

Fichier : *wikipedia_part.txt* (5.39 Go)						
Nb de proc.	4	8	24	48	64	128
Temps de lecture	4.78	3.32	2.14	2.13	1.91	2.11
Speedup lecture	2.4	3.43	5.32	5.34	5.97	5.39
Temps pour le comptage	71.3	36.9	11.95	6.3	4.69	2.45
Speedup comptage	4.14	7.97	24.7	46.8	62.8	120.2
Temps réduc.	15.7	21.04	22	22.48	22.9	23.8
Temps total	92.5	62.2	37.3	32.18	30.58	29.47
Speedup total	3.31	4.92	8.21	9.52	10.02	10.39

Fichier : *wikipedia.txt* (18 Go)					
Nb de proc.	16	32	64	128	256
Temps de lecture	5.92	3.94	2.61	2.4	2.84
Speedup lecture	6.43	9.65	14.6	15.9	13.4
Temps pour le comptage	58.6	30.98	15.4	7.82	4
Speedup comptage	16.6	31.4	63.2	124.5	250.4
Temps réduc.	49.5	58.6	58.7	59.2	60.1
Temps total	116.3	92.24	79.5	72.4	69.7
Speedup total	8.7	10.5	12.7	14	14

Figure 8 – Tables des temps d'exécution en secondes pour le wordcount MPI et valeurs de speedup

On remarque dans les résultats de la *figure 8* que le temps pour le comptage des mots en parallèle correspond presque exactement au temps séquentiel divisé par le nombre de processeurs utilisés, comme on pouvait l'espérer. Cette partie du code est donc très bien parallélisée.

Les temps pour la lecture du fichier diminuent aussi avec le nombre de processeurs, mais de moins en moins au fur-et-à-mesure que ce nombre devient grand. En particulier, on voit que c'est généralement à partir de 64 processeurs utilisés qu'on a plus de gain de temps pour la lecture parallèle des fichiers. Une explication possible à cela est que, comme le *cluster Baobab* ne possède que trois nœuds de stockage, il n'est probablement pas possible d'accéder aux disques en lecture avec plus de 64 processeurs à la fois. Les résultats restent malgré tout meilleurs que ceux de la version séquentielle. Les valeurs obtenues sont aussi plus petites que celles mesurées sur *Scylla*, ce qui est probablement dû aux systèmes de fichiers installés sur

chaque machine : celui de *Baobab* est certainement mieux adapté pour les lectures parallèles que celui de *Scylla* (comme indiqué plus haut, on sait que *Baobab* possède trois nœuds dédiés spécifiquement au stockage, ce qui n'est pas le cas de *Scylla*). Il est aussi possible que l'implémentation de *MPI_IO* pour le système de fichiers de *Baobab* soit meilleure.

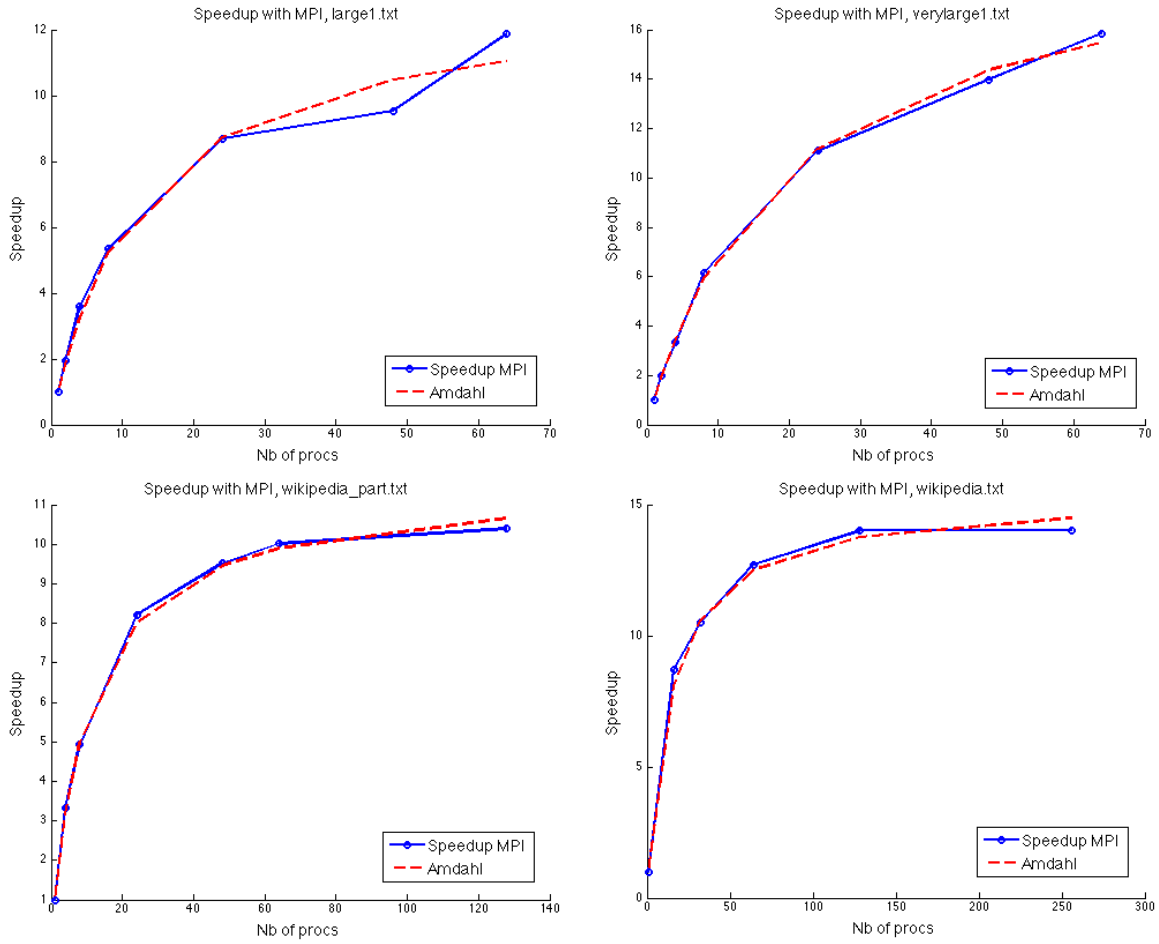

Figure 9 – Speedup du wordcount MPI par rapport à la version séquentielle et loi d'Amdahl

Le temps de réduction des résultats représente un *overhead* lié au parallélisme dans le code, puisqu'il correspond à du temps pris par le programme parallèle pour effectuer des opérations qui sont absentes de la version séquentielle du *wordcount*. La portion de code de cette partie correspond donc au paramètre α dans la loi d'Amdahl. On observe d'ailleurs dans la *figure 9* que les courbes de *speedup* de notre programme semblent bien suivre cette loi.

On remarque enfin dans les valeurs de la *figure 8* que le temps pris pour la réduction des résultats augmente légèrement avec le nombre de processeurs. En effet, plus le nombre de processeurs utilisé est important, plus le nombre d'étapes nécessaires pour l'opération devient grand. La réduction est donc une étape qui limite la « scalabilité » du programme *MPI*.

6.4 Comparaison entre Spark et MPI

En divisant le temps d'exécution du *wordcount Spark* par celui de la version *MPI* pour chaque fichier traité et chaque nombre de processeurs utilisés, on obtient le *speedup* de *MPI* par rapport à *Spark*. On a représenté ce *speedup* dans les graphiques de la *figure 10*.

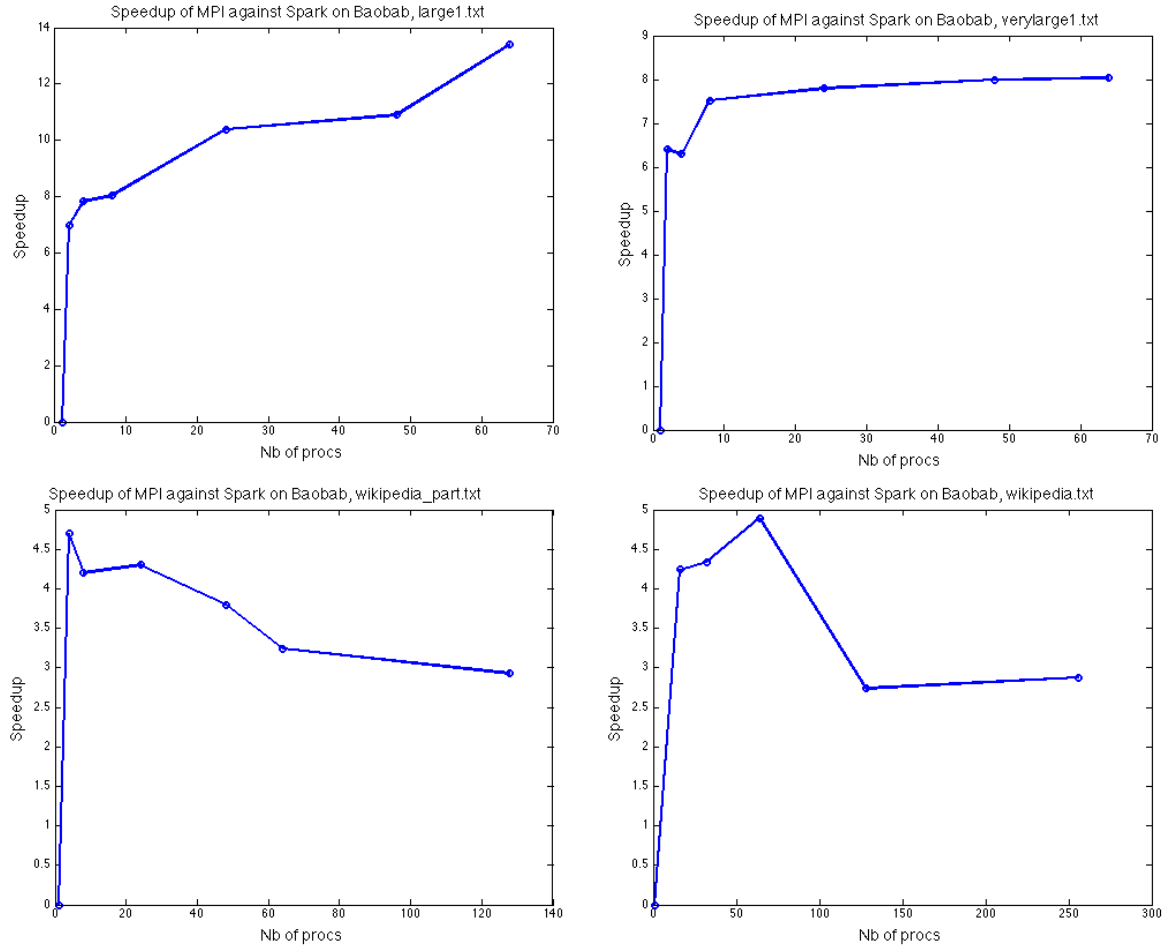

Figure 10 – Speedup de MPI par rapport à Spark

On voit bien que *MPI* est toujours beaucoup plus rapide que *Spark* (près de 14 fois plus rapide dans le meilleur des cas). Toutefois, on observe aussi que lorsque des fichiers de plus grande taille sont traités, le *speedup* de *MPI* par rapport à *Spark* a tendance à se réduire, en particulier lorsque beaucoup de processeurs sont utilisés. Pour expliquer ceci, on peut revenir sur la remarque faite précédemment quant à la « scalabilité » de *Spark* : il ne devient intéressant d'utiliser le *framework* que lorsque les problèmes à traiter sont très grands et que le nombre de processeurs participant au programme est important, car *Spark* introduit un *overhead* lié à la distribution des tâches entre processeurs qui est conséquent.

Il est difficile de comparer les performances de *Spark* et *MPI* pour la phase de réduction des résultats, qui constitue le principal goulot d'étranglement du parallélisme dans l'application étudiée ici. À cause du modèle de *lazy evaluation* des fonctions de *Spark*, on n'a pas pu mesurer le temps pris par ce dernier pour l'exécution de cette phase. De plus, le

modèle de réduction adopté ici avec *MPI* est fondamentalement différent de celui utilisé par *Spark* lorsqu'il doit fusionner des collections de paires de clés-valeurs. En effet, *Spark* utilise une opération appelée *shuffle* pour la réduction des *maps*. Cette opération consiste à réduire plusieurs *maps* en un seul, dans lequel toutes les valeurs associées à une même clé sont fusionnées, mais en conservant le résultat de la réduction distribué sur plusieurs processeurs. Pour ce faire, *Spark* redistribue les différentes clés du *map* entre les processeurs participant au programme selon une logique particulière (alphabétiquement, par exemple), puis il réduit toutes les valeurs associées à une même clé sur le processeur qui la possède après la redistribution. Une description plus détaillée de l'implémentation du *shuffle* dans *Spark* est donnée sur [25]. Le modèle adopté par *Spark* possède l'avantage de maintenir un *map* distribué sur plusieurs processeurs même après sa réduction, ce qui permet de lui appliquer d'autres opérations de façon distribuée ultérieurement. Toutefois, le *shuffle* est une opération complexe et très coûteuse en temps. Il est donc très probable que les performances de la réduction implémentée avec *MPI* au cours de ce travail soient meilleures que celles du *shuffle* de *Spark.*

D'une manière générale, on observe que *MPI* est bien meilleur en termes de performances, et il semble donc particulièrement intéressant d'utiliser ce dernier dans des applications où les temps d'exécution sont critiques. Cela dit, on a aussi vu que le développement d'une application avec *MPI* représentait un effort de programmation considérablement plus important que l'écriture d'un même programme avec *Spark.* C'est pourquoi on propose dans la section suivante un petit exemple de *framework* développé avec *MPI* qui offre des primitives pour la programmation similaires à *Spark.*

7. Proposition d'un framework MapReduce avec MPI

Un exemple de *framework* pour le calcul distribué programmé en *C++* et utilisant *MPI* est proposé dans le dossier *Source/MPI_Capsule* de ce travail. Le *framework* est implémenté sous la forme d'une librairie de *headers C++*. Il offre des abstractions de haut niveau permettant d'effectuer le même type d'opérations que celles qui ont été utilisées dans le *wordcount* programmé avec *Spark* (la lecture d'un fichier sur plusieurs processeurs de façon distribuée, par exemple, ou encore des fonctions *map* et *reduce* pouvant être appliquées sur des données distribuées, encapsulées dans des objets spécifiques). Toutes les classes et les méthodes proposées dans la librairie consistent en fait simplement en des versions encapsulées du code et des fonctions présentés précédemment dans le *wordcount* programmé avec *MPI.* C'est pour cette raison que le choix du nom « *MPI Capsule* » a été fait.

Pour pouvoir utiliser *MPI Capsule* dans un programme écrit en *C++*, il suffit d'y inclure le *header* <*mpi_capsule.hpp*>, avec la ligne suivante :

```
#include <mpi_capsule.hpp>
```

Pour pouvoir compiler un programme qui utilise la librairie, il faut utiliser un *wrapper* permettant la compilation d'un programme *MPI* écrit en *C++*, tel que *mpicc* ou *mpic++* par

exemple. Il est aussi nécessaire d'indiquer au compilateur le chemin vers les *headers* de *MPI Capsule* (avec l'option *-I* si le compilateur *gcc* est utilisé, par exemple). Enfin, pour les aspects de sérialisation dans le code, *MPI Capsule* utilise *Cereal*, une libraire de *headers* programmée pour le standard *C++11*. Pour cette raison, les programmes qui utilisent le *framework* doivent être compilés avec la norme 2011 du langage *C++* au moins (ceci peut être spécifié lors de la compilation avec l'option *-std=c++0x* si *gcc* est utilisé).

Deux exemples de programmes qui utilisent *MPI Capsule* sont fournis dans le dossier *Source/MPI_Capsule/examples* de ce travail. Le premier consiste simplement en une implémentation du *wordcount* avec la librairie. Le second est un petit programme qui lit un fichier contenant des données numériques et qui calcule la somme du carré de ces dernières. Chacun des exemples est fourni avec un *Makefile* permettant de le compiler et de l'exécuter facilement.

Un extrait du code du *wordcount* implémenté avec *MPI_Capsule* est donné ci-dessous :

```
MPI_Context context(argc, argv);

auto dText = context.textFile(argv[1], ' ');

auto counts = dText.map(count_words).reduce(merge);

orderedData.printData([](map<string,int>& map) { for (auto kv : map) cout <<
       kv.first << ":" << kv.second << endl; });
```

Dans ce code, un contexte *MPI_Context* est instancié, puis la méthode *textFile* de l'objet est appelée afin de charger dans une chaîne de caractères le contenu du fichier dont le chemin est indiqué en paramètre, le tout de façon distribuée sur plusieurs processeurs. Ensuite, lorsque la méthode *map* est appelée, chaque processeur applique la fonction *count_words* sur le morceau de fichier qu'il a lu (la fonction *count_words*, définie ailleurs, permet de compter les occurrences des mots présents dans une chaîne de caractère). Après cela, tous les processeurs du programme appellent la méthode *reduce*. Cette dernière effectue la réduction des résultats obtenus précédemment par chaque processeur vers un seul d'entre-eux. Elle applique pour la fusion des résultats la fonction *merge* définie ailleurs. Finalement, la méthode *printData* est appelée afin d'afficher le résultat final de la réduction. Il est intéressant de noter qu'il est possible de passer des lambda-fonctions en paramètre aux méthodes décrites ici, comme cela est illustré dans la dernière ligne du code présenté ci-dessus.

8. Conclusion

Au cours de ce travail, nous avons vu à quel point il pouvait être compliqué d'écrire des applications parallèles (même très simples) avec *MPI* en comparaison avec ce qu'il est possible de faire avec *Spark*. La section d'explications sur l'implémentation en *MPI* du *wordcount* en témoigne bien : on a vu que tous les aspects de distribution des tâches, de calculs parallèles et de réduction des résultats devaient être gérés par le développeur et

pouvaient parfois représenter des problèmes relativement difficiles à résoudre de façon efficace. Au contraire, on a vu à quel point il était facile de programmer le même genre d'applications à l'aide de *frameworks* pour le calcul distribué tels que *Spark.*

Toutefois, nous avons aussi observé lors de nos mesures de performances que la facilité de développement proposée par *Spark* avait un coût : *MPI* est beaucoup plus rapide que *Spark* pour la résolution d'un même problème dans les mêmes conditions (parfois jusqu'à plus de dix fois). On peut très probablement expliquer cette observation par le fait que *MPI* est une interface de plus bas niveau, et qu'elle introduit donc un *overhead* lié au parallélisme généralement plus faible dans ses programmes, puisque sa gestion de la mémoire et des communications inter-processeurs est mieux optimisée.

Afin de prendre le meilleur des deux mondes et de le combiner dans un même *framework*, on a proposé dans la dernière partie de ce travail une librairie programmée avec *MPI* offrant des abstractions de haut niveau pour faciliter le développement d'applications parallèles. Le modèle de programmation sur lequel nous nous sommes basés pour l'interface de cette librairie est celui de *Spark*, c'est-à-dire le modèle *Map/Reduce.* Cependant, la librairie développée dans le cadre de ce travail n'implémente qu'une toute petite partie de ce que *Spark* offre, et son but est plus de servir d'exemple de ce qui pourrait être fait à large échelle que de constituer une véritable alternative à *Spark*. Il pourrait donc être intéressant de développer un réel *framework* complet basé sur *MPI* qui implémente toutes les fonctionnalités de *Spark* lors d'un travail ultérieur.

Un autre aspect important du modèle proposé par *Spark* qui n'a pas été abordé en détail au cours de ce travail est l'opération de *shuffle*. Comme expliqué dans la section 6.4, le *shuffle* permet d'effectuer la réduction de paires de clés-valeurs tout en continuant de stocker ces dernières de façon distribuée sur plusieurs processeurs, afin de pouvoir leur appliquer d'autres traitements en parallèle par la suite. Le *shuffle* est une opération qui est difficile à mettre en place et à programmer de façon efficace dans un système distribué, et il pourrait être très intéressant d'étudier de quelle façon ceci pourrait être fait avec *MPI* dans un *framework Map/Reduce* basé sur ce dernier.

9. Bibliographie

[1] “IBM - Définition du Big Data” Disponible sur : https://www-01.ibm.com/software/fr/data/bigdata/ Accédé le : 07-Oct-2016.
[2] “Apache Spark™ - Lightning-Fast Cluster Computing.” Disponible sur : http://spark.apache.org/ Accédé le : 07-Oct-2016.
[3] J. Dean and S. Ghemawat, “MapReduce : Simplified Data Processing on Large Clusters” *OSDI ’04: 6th Symposium on Operating Systems Design and Implementation*, 2004.
[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Schenker, and I. Stoica, “Spark : Cluster Computing with Working Sets” *HotCloud 2010*, Juin 2010.
[5] “Welcome to Apache™ Hadoop®!” Disponible sur : http://hadoop.apache.org/ Accédé le : 07-Oct-2016.
[6] M. Zaharia *et al.*, “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing” *NSDI 2012*, Avril 2012.
[7] “JerryLead/SparkInternals” Disponible sur : https://github.com/JerryLead/SparkInternals Accédé le : 18-Jul-2016.
[8] “MPI Forum.” Disponible sur : http://mpi-forum.org/ Accédé le : 07-Oct-2016.
[9] “Open MPI: Open Source High Performance Computing.” Disponible sur : https://www.open-mpi.org/ Accédé le : 07-Oct-2016.
[10] “Web pages for MPI Routines.” Disponible sur : http://www.mpich.org/static/docs/v3.1/www3/ Accédé le : 07-Oct-2016.
[11] “Tutorials · MPI Tutorial.” Disponible sur : http://mpitutorial.com/tutorials/ Accédé le : 07-Oct-2016.
[12] T. Baer, P. Peltz, J. Yin, and E. Begoli, “Integrating Apache Spark into PBS-Based HPC Environments” in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, New York, NY, USA, 2015, pp. 34:1–34:7.
[13] S. J. Kang, S. Y. Lee, and K. M. Lee, “Performance Comparison of OpenMP, MPI, and MapReduce in Practical Problems” *Adv. Multimed.*, vol. 2015, p. e575687, Aug. 2015.
[14] X. Lu, B. Wang, L. Zha, and Z. Xu, “Can MPI Benefit Hadoop and MapReduce Applications?” in *2011 40th International Conference on Parallel Processing Workshops*, 2011, pp. 371–379.
[15] S. Sehrish, J. Kowalkowski, and M. Paterno, “Exploring the Performance of Spark for a Scientific Use Case ” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1653–1659.
[16] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, “Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf” *Procedia Comput. Sci.*, vol. 53, pp. 121–130, Jan. 2015.
[17] “Scalability! But at what COST?” Disponible sur : http://www.frankmcsherry.org/graph/scalability/cost/2015/01/15/COST.html Accédé le : 21-Oct-2016.
[18] “Quick Start - Spark 2.0.1 Documentation.” Disponible sur : http://spark.apache.org/docs/latest/quick-start.html Accédé le : 20-Oct-2016.
[19] “c - MPI io reading file by processes equally by line (not by chunk size) - Stack Overflow.” Disponible sur : http://stackoverflow.com/questions/13327127/mpi-io-reading-file-by-processes-equally-by-line-not-by-chunk-size Accédé le : 04-Nov-2016.
[20] “Project Gutenberg” Disponible sur : http://www.gutenberg.org/ Accédé le : 04-Nov-2016.
[21] “cereal Docs - Main.” Disponible sur : http://uscilab.github.io/cereal/ Accédé le : 05-Nov-2016.

[22] “The cluster — Unige HPC 0.6 documentation.” Disponible sur : http://baobabmaster.unige.ch/enduser/src/enduser/enduser.html Accédé le : 18-Nov-2016.
[23] “Index of /enwiki/.” Disponible sur : https://dumps.wikimedia.org/enwiki/ Accédé le : 18-Nov-2016.
[24] “BeeGFS - The Parallel Cluster File System” Disponible sur : http://www.beegfs.com/content/ Accédé le : 05-Dec-2016.
[25] S, y Ryza, and SaisaiShao, “Improving Sort Performance in Apache Spark: It’s a Double” *Cloudera Engineering Blog*, 14-Jan-2015. Disponible sur : http://blog.cloudera.com/blog/2015/01/improving-sort-performance-in-apache-spark-its-a-double/ Accédé le : 16-Jul-2016.