Chapitre 4

Schémas de parallélisation fréquents du *HPC* et du *Big Data*

Les PC d'aujourd'hui contiennent souvent plusieurs processeurs qui sont tous multi-cœurs, et chaque cœur de calcul contient aussi des unités de calcul vectoriel. De plus, l'agrégation de plusieurs de ces PC au sein de *clusters* est nécessaire pour atteindre des capacités de calcul et de stockage importantes et extensibles (nécessaire pour *passer à l'échelle*). Mais exploiter ces architectures demande de développer des applications parallèles et distribuées, et donc de concevoir des algorithmiques plus complexes.

Ce chapitre introduit des schémas d'applications *multi-tâches*, visant à exploiter en parallèle plusieurs cœurs d'une même machine ou plusieurs machines. Il se focalise sur deux schémas de parallélisation classiques et de haut niveau : le schéma *SPMD* apparu dans le *HPC*, et le schéma *Map-Reduce* associé au *Big Data*.

4.1 Barrières de synchronisation

Les *barrières de synchronisation* sont typiques de la programmation parallèle visant à exécuter concurremment plusieurs tâches dans le but d'accélérer les traitements.

4.1.1 Principe des barrières sur fin de tâches

Considérons l'exemple (classique) d'une application composée d'une tâche principale qui crée des tâches filles (ou sous-tâches), puis qui doit attendre la fin de chacune de ses filles avant de terminer sa propre exécution. Dans les langages de programmation et les bibliothèques de synchronisation ces barrières sont souvent appelées *join(...)*. Quand une tâche crée une tâche fille (un processus ou un thread), elle récupère toujours un identificateur sur cette tâche, et peut ensuite se bloquer jusqu'à la fin de sa tâche fille en appelant une fonctionnalité de type *join(SubTaskId)*. Si la tâche fille visée est encore en vie, alors l'opération *join* bloquera et suspendra la tâche mère jusqu'à la mort de la tâche fille. Si la tâche fille est déjà morte, alors l'opération *join* sera sans effet et ne bloquera pas la tâche mère.

Donc, si une tâche mère souhaite attendre la fin de plusieurs tâches filles, elle pourra exécuter simplement une succession de *join* (*join(SubTaskId1) ; join(SubTaskId2) ;* ...), et l'ordre d'exécution des différents *join()* n'aura pas d'importance. Parfois il existe une opération *join(tab[])* qui prend en argument un tableau d'identificateurs de tâches, et qui attend la mort de toutes les tâches référencées dans le tableau.

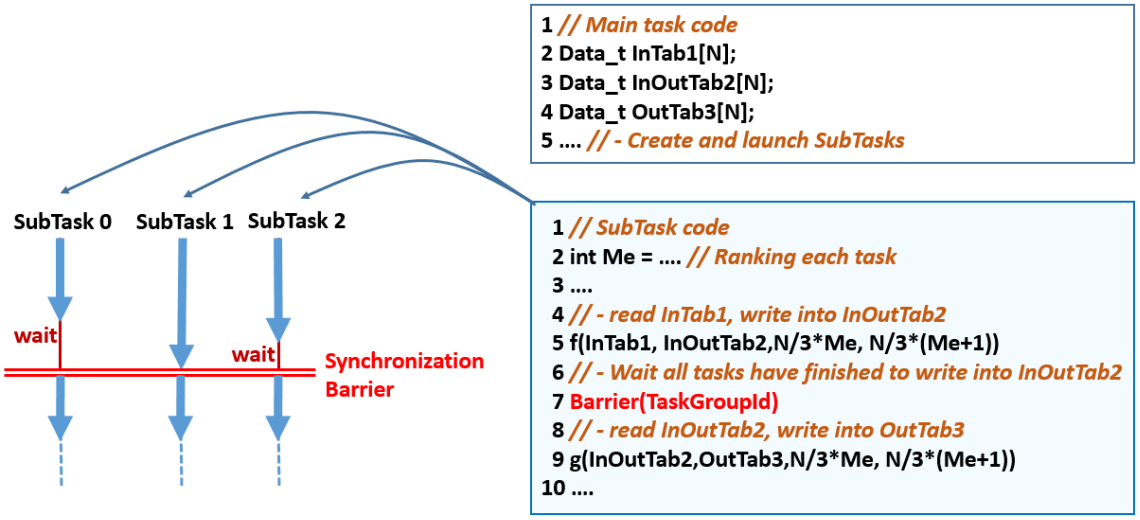

FIGURE 4.1 – Utilisation d'une barrière de synchronization au cours de calculs parallèles

4.1.2 Principe des barrières génériques

Une barrière de calcul générique désigne un outil permettant de resynchroniser un ensemble de tâches à une étape précise de leurs calculs (sans que ce soit leur fin). La figure 4.1 illustre cette démarche avec 3 sous-tâches (par exemple des threads créées au sein d'un processus) qui exécutent le même code, partagent des tableaux de données globaux et se répartissent chacun 1/3 des calculs.

Une solution classique est de commencer par numéroter les n sous-tâches de calcul de 0 à $n-1$ (ligne 2 des sous-tâches). Dans cet exemple on fait ensuite appel à des fonctions f et g qui calculent les valeurs d'un tableau de sortie entre deux indices fonctions du numéro de la sous-tâche, et en fonction des valeurs d'un tableau d'entrée :

$$f(InputTable, OutputTable, InfIndex, SupIndex)$$

On suppose que chaque valeur de sortie peut nécessiter d'accéder à tout ou partie du tableau d'entrée. En conséquence, une sous-tâche de calcul doit attendre la fin des calculs de la première étape (fonction $f(...)$, ligne 5) de toutes les sous-tâches avant d'entamer sa deuxième étape (fonction $g(...)$, ligne 9). La barrière de synchronisation installée ligne 7 du code des sous-tâches garantit donc que les calculs de la fonction g se feront bien à partir d'un tableau $InOutTab2$ totalement à jour. Sur la partie gauche de la figure 4.1 on voit que les sous-tâches 0 et 2 ont attendu sur la barrière : elles avaient terminé leurs calculs plus tôt, et ont du attendre que la sous-tâche 1 atteigne elle même la barrière. Dès que la sous-tâche la plus lente a atteint la barrière, l'ensemble des sous-tâches en attente ont été débloquées et ont poursuivi leurs calculs.

Evidemment, on espère équilibrer la charge de calcul des différentes sous-tâches, afin d'équilibrer les temps de calculs des différents cœurs ou processeurs utilisés pour exécuter les sous-tâches. On espère donc que les temps d'attente sur la barrière seront les plus courts possibles, ce qui est compliqué à réaliser quand les calculs des différentes sous-tâches ne sont pas identiques. Il est toujours plus simple de paralléliser efficacement des calculs réguliers (comme des calculs matriciels d'algèbre linéaire dense) plutôt que des calculs irréguliers dont la progression en chaque point dépend des valeurs calculées. L'analyse de données peut malheureusement déboucher sur des calculs irréguliers, et certains algorithmes de *clustering* sont ainsi compliqués à paralléliser efficacement.

4.1.3 Mise en œuvre de barrières génériques

Il existe des *barrières génériques* dans de nombreux langages de programmation et de nombreuses bibliothèques de synchronisation, qui sont souvent très simples à utiliser. Si ce n'est pas le cas, il est possible d'en implanter de très efficaces au sein d'une même machine à mémoire partagée à partir de quelques sémaphores : on peut facilement aboutir à une implantation qui suspende les tâches arrivant sur la barrière et qui n'utilise pas d'exclusion mutuelle (source de ralentissement). En revanche, implanter une barrière de synchronisation sur un cluster de machines sera beaucoup plus coûteux, car cela nécessitera des échanges de messages entre les machines (et donc à travers le réseau d'interconnexion). Afin de réduire ce coût, certains super-calculateurs possèdent même un *hardware* dédié à la réalisation de barrières rapides à grande échelle.

Mais dans tous les cas, traverser une barrière de synchronisation à un coût, même si toutes les tâches ont la même charge de travail et rencontre la barrière quasiment en même temps. Il est donc important de concevoir des algorithmes parallèles et distribués faisant le minimum d'appels à des barrières de synchronisation. Nous utiliserons ainsi des barrières de synchronisation *avec modération* quand nous étudierons la parallélisation de certains algorithmes d'analyse de données (voir les chapitre de la partie IV).

4.2 Schéma de parallélisation SPMD

4.2.1 Définition du SPMD

SPMD signifie *Single Program Multiple Data*, et correspond à un paradigme de programmation parallèle adapté aux architectures à base de CPU, que ce soit avec une mémoire partagée au sein d'un nœud de calcul ou avec une mémoire distribuée à travers un cluster de PC. Une programmation SMPD signifie que l'on déploie un ensemble de tâches et qu'elles exécutent toutes le même programme, avec de temps en temps des points de synchronisation. Mais tout en exécutant le même programme, deux tâches peuvent exécuter des instructions parfois différentes car certaines divergences sont tolérées. Considérons par exemple une structure *if (cond) then ... else ...*, une tâche peut passer dans le *then* et une autre dans le *else*, selon les valeurs de leurs variables testées dans la condition du *if*. De plus, contrairement au paradigme SIMD (*Single Instruction Multiple Data*), les tâches ne sont pas resynchronisées à chaque tic d'horloge, mais seulement à des points de synchronisation explicites que l'on espère les moins fréquents possibles (car une synchronisation est toujours coûteuse en temps d'exécution).

Le mode de parallélisation SPMD est bien adapté aux architectures modernes à base de plusieurs cœurs CPU, dont chacun possède sa propre horloge et son propre décodeur d'instructions. Deux cœurs CPU peuvent très bien exécuter en parallèles un *then* et un *else*. En fait ils pourraient très bien exécuter des codes complètement différents se re-synchronisant seulement de temps en temps, c'est-à-dire selon un mode MPMD (*Multiple Programs Multiple Data*). Néanmoins les programmes parallèles de type MPMD sont difficiles à mettre au point en dehors de schémas simples (comme le producteur-consommateur), alors que la plupart des algorithmes de calcul intensif sont naturellement de type SMPD. Ils distribuent les mêmes calculs sur des blocs de données différents, échangent des résultats intermédiaires ou font circuler des données initiales entre les tâches selon des schémas de communication souvent réguliers, et observent quelques divergences d'exécution selon les valeurs numériques rencontrées.

La programmation SPMD s'est ainsi imposée ces dernières années sur les architectures à base de CPU, bien qu'une programmation encore plus régulière avec encore moins de divergences, de type SIMD, revienne en force avec les unités vectorielles des cœurs CPU et avec les GPU. Des algorithmes de *Machine Learning* comme des réseaux de neurones se prêtent d'ailleurs bien à une parallélisation SIMD. Toutefois, il est fréquent de mixer les deux paradigmes : créer un

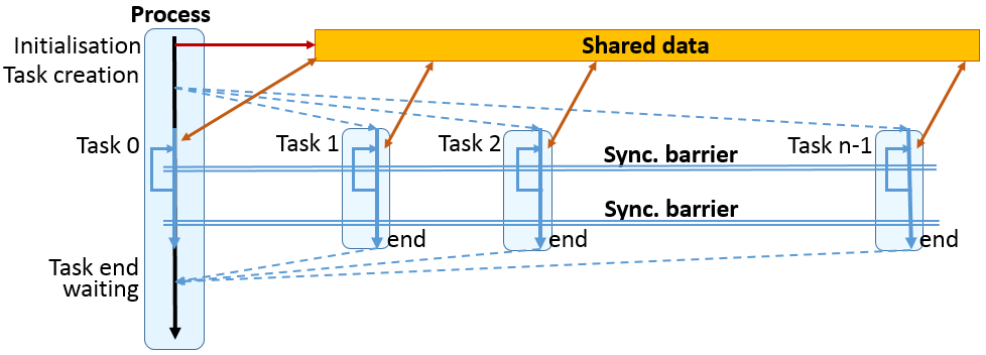

FIGURE 4.2 – Exemple d'architecture multithreadée selon un schéma SPMD

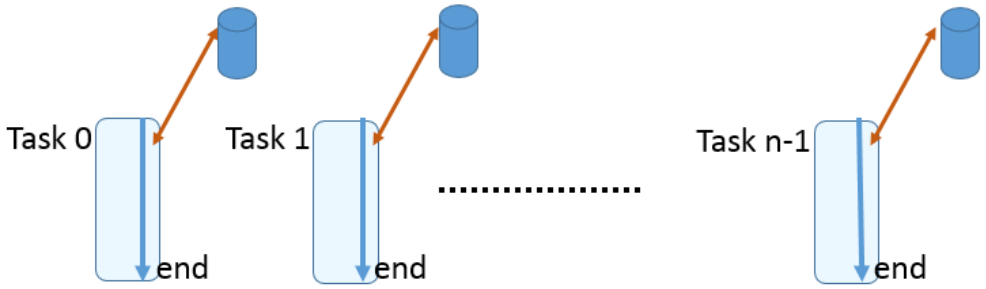

FIGURE 4.3 – Exemple d'architecture SPMD *"Embarrassingly Parallel"*

programme SPMD dont chaque tâche réalise essentiellement des calculs SIMD, afin d'exploiter un ensemble de cœurs CPU ayant chacun des unités vectorielles à sa disposition (architecture très classique aujourd'hui).

4.2.2 Exemple de schéma SPMD

La figure 4.2 montre un exemple de programme SPMD réalisé par multithreading au sein d'un même nœud de calcul (à mémoire partagée). Un processus (que l'on peut considérer comme le thread principal) alloue et initialise des structures de données dans son espace mémoire, puis créé n threads qui accèdent à ces structures de données et les partagent. Chaque thread exécute le même code, qui contient une boucle avec une barrière de synchronisation (voir section 4.1.2), puis termine sa boucle et exécute encore un peu de code avec une dernière barrière de synchronisation au milieu, et enfin meurt. Le thread principal attend la mort de tous ses threads fils (voir section 4.1.1) avant de poursuivre et de se terminer à son tour.

Chaque thread fils exécute le même code, mais peut s'autoriser quelques divergences. Toutefois, tous les threads doivent exécuter une barrière de synchronisation à chaque itération de leur boucle, et encore une après leur boucle. Si un thread fait une barrière de moins que les autres, alors les $n-1$ autres resteront bloqués sur leur barrière et ne termineront pas. En conséquence, le thread principal restera à son tour bloqué en attente de leur mort, et finalement l'ensemble du programme parallèle sera en état d'interblocage. Donc, si une barrière de synchronisation se trouve dans le corps d'un *then*, il doit y en avoir une aussi dans le corps du *else* correspondant.

Un mode de parallélisation SPMD est donc simple à mettre en œuvre et les différentes tâches peuvent assumer quelques divergences, mais doivent toutes respecter les étapes de synchronisation.

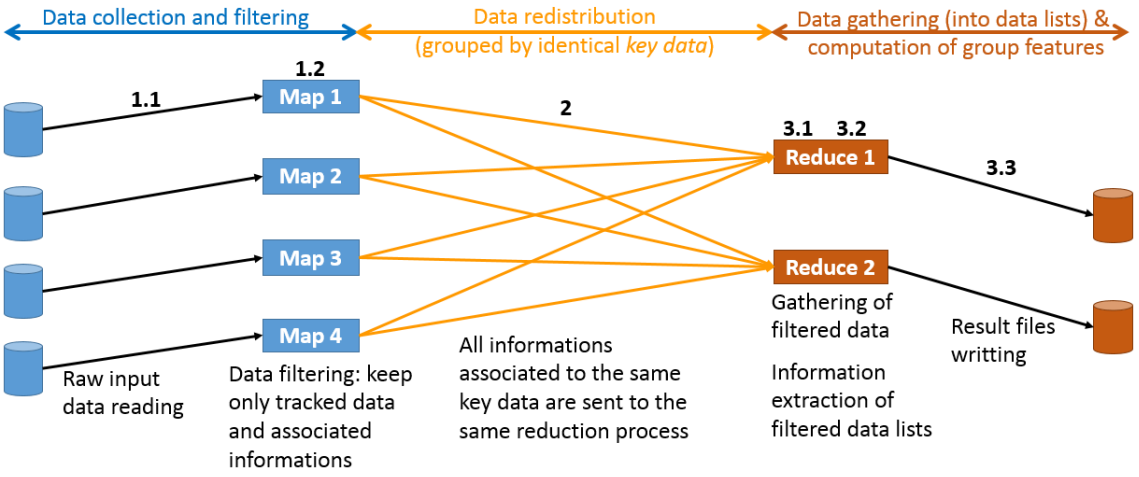

FIGURE 4.4 – Pipelining en 3 grandes étapes d'une chaîne de traitement *Map-Reduce*

4.2.3 Schéma SPMD *embarrassingly parallel*

Les problèmes *embarrassingly parallel*, ou *happily parallel*, sont souvent des schémas SPMD extrêmes. Il s'agit de cas où on lance n tâches en concurrence totale (voir figure 4.3), sans communication ni synchronisation, souvent pour exécuter le même programme sur des données différentes (correction de n images indépendantes) ou sur les mêmes données avec des jeux de paramètres différents (investigation multi-paramètres) pour identifier ensuite le jeu de paramètres ayant mené à la meilleure solution.

Dans tous les cas, ces problèmes suivent habituellement un schéma de parallélisation SPMD sans synchronisation, ou avec une simple barrière à la fin pour qu'une tâche mère sache que tout est terminé. Des investigations multi-paramètres peuvent s'utiliser en apprentissage numérique pour rechercher le meilleur paramétrage d'un algorithme de *Machine Learning*, ou le meilleur jeu de données d'apprentissage.

4.3 Schéma de parallélisation *Map-Reduce*

Une chaîne de traitement *Map-Reduce* transforme des ensembles de paires *clé-valeur*, et le fait en parallèle. On distingue deux grandes étapes : une étape de *Map* qui suit a priori un schéma *SPMD embarrassingly parallel*, puis une étape de *Reduce* qui semble suivre un schéma similaire. Ces deux étapes sont reliées par une redistribution des données (le *shuffle & sort*) qui route les paires clé-valeur de sortie de l'étape *Map* vers l'entrée de l'étape *Reduce*. L'étape de redistribution est complexe mais quasiment figée, et les deux étapes de *Map* et de *Reduce* sont *pipelinées*, ce qui rend le fonctionnement du *Map-Reduce* à la fois plus efficace et plus générique.

4.3.1 Principes de la chaîne de traitement *Map-Reduce*

La figure 4.4 présente une version très simplifiée de la chaîne *Map-Reduce* d'*Hadoop*, en 3 grandes parties : la collecte et le filtrage des données, le regroupement des données en ensembles cohérents, et le calcul de caractéristiques de groupes.

– *La collecte et le filtrage des données* : peut concerner des machines localisées sur différents sites (là où sont les données brutes) et réaliser de nombreuses lectures de fichiers. Chaque fichier est lu (sous-étape 1.1 de la figure 4.4) et analysé (1.2), et seules les données satisfaisant les processus *Map* seront conservées. Par exemple, dans d'énormes fichiers de logs on

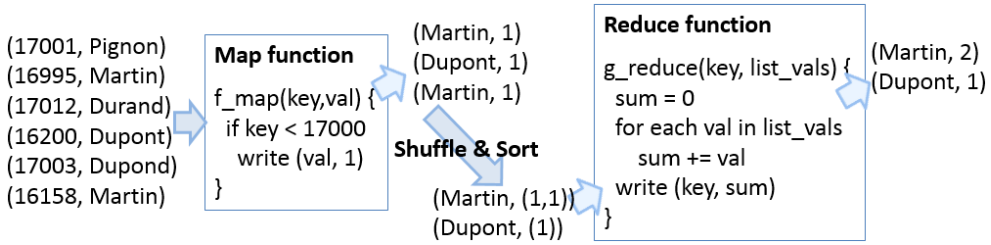

FIGURE 4.5 – Principe d'un *Map-Reduce* en *Hadoop* sur des paires clé-valeur

ne retient que les lignes concernant des adresses IP satisfaisant un certains masque réseau, et on ne retient finalement que les dates d'événements associés à ces IP. En fait, les processus *Map* produisent des couples clé-valeur dont la clé pourrait être une adresse IP (parmi celles retenues) et la valeur pourrait être une date d'événement associé à cette IP.

Dans ce genre d'opérations, ce sont en général les lectures de fichiers qui sont l'opération la plus coûteuse, à cause des bandes passantes limitées des disques. C'est particulièrement vrai si on lit des données stockées sur des disques bon marché.

– *La redistribution des données en ensembles cohérents* : chaque processus *Map* envoie ses données à des processus *Reduce* (voir ci-après), mais au final toutes les données avec la même clé doivent être envoyées vers le même processus *Reduce*. Il s'agit à la fois d'une étape de routage et d'une étape de redistribution des données filtrées, appelé *shuffle & sort*.

  L'implantation efficace de cette étape relève d'un haut niveau de technicité en terme d'informatique distribuée. En général elle est entièrement réalisée dans une bibliothèque ou un intergiciel, et n'est pas du tout à la charge du développeur applicatif. Toutefois *Hadoop* permet de contrôler la répartition des clés et le regroupement des données sur les processus *Reduce*, ce qui autorise certaines optimisations des applications *Map-Reduce* (voir section 6.5.3).

– *Le regroupement et le calcul de caractéristiques de groupes* : chaque processus *Reduce* regroupe dans une même liste toutes les valeurs reçues avec une même clé (3.1), puis applique une fonction de calcul sur cette liste des valeurs associées à une même clé (3.2). Par exemple, il identifie la date la plus ancienne et la date la plus récente concernant les événements d'une même IP. Enfin, il sauve une nouvelle paire clé-valeur (3.3) où la clé est toujours l'adresse IP et la valeur est l'intervalle de dates où des événements sont associés à cette IP.

  A nouveau, la sauvegarde sur disque peut se révéler une opération coûteuse lors de cette dernière étape.

4.3.2 Premier exemple de traitement *Map-Reduce*

La figure 4.5 illustre le fonctionnement d'une opération *Map-Reduce* indépendamment de son implantation exacte. Un ensemble de paires *clé1-valeur1* (numéro de contrat, nom du vendeur) est lu et traité par l'étage *Map* qui applique indépendamment sur chaque paire une fonction *f_map*. Celle-ci teste la clé de la paire et ne retient que les clés inférieure à 17000, pour lesquelles elle génère une nouvelle paire *clé2-valeur2* (*nom du vendeur*, *1*), avec *1 = un contrat vendu*. La clé de la deuxième paire est donc la valeur de la paire initiale, et sa valeur est imposée à *1*.

La phase *shuffle & sort* regroupe tout d'abord toutes les paires de même clé, et crée une nou-

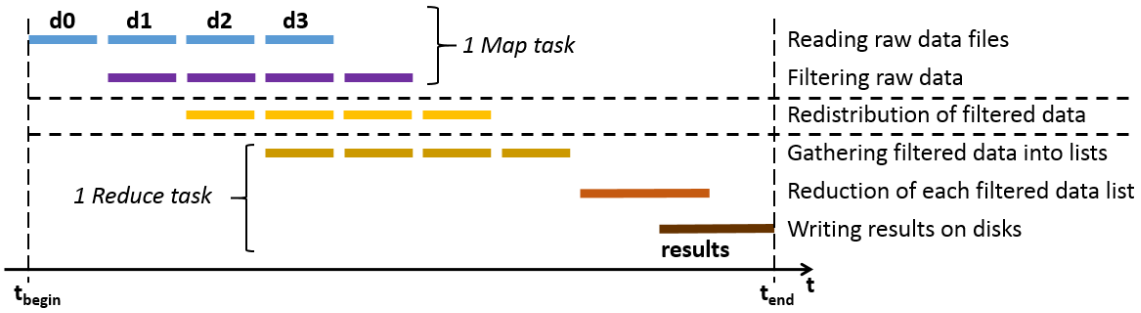

FIGURE 4.6 – Principe de recouvrement temporel des étapes d'un traitement *Map-Reduce*

velle paire *clé2 - liste de valeur2*. Deux paires de clé *Martin* et de valeur 1 vont donc donner naissance à une paire (*Martin*, (1,1)).

Puis on applique une fonction *g_reduce* à chaque paire *clé2 - liste de valeur2*, qui calcule la somme des *valeur2* de la liste, c'est à dire le nombre de contrats vendu par un même vendeur. La fonction *g_reduce* génère alors une nouvelle paire *clé3-valeur3*, avec toujours le nom du vendeur comme clé, et le nombre de contrats vendus par ce vendeur comme valeur.

Cet exemple est en fait le même que le traditionnel *Word Counter*, sorte de programme *Hello World* du *Map-Reduce*. Il peut être réalisé avec une tâche *Map* et une tâche *Reduce* sur une seule machine, mais il peut aussi l'être avec plusieurs tâches de chaque type, déployées sur un cluster.

4.3.3 Exécution parallèle de la chaîne *Map-Reduce*

Les différents processus *Map* de notre exemple peuvent s'exécuter en concurrence totale, ce sont des traitements indépendants. Il en est de même pour les processus *Reduce*, et pour les redistributions de données de la phase 2. Chaque phase de notre chaîne de traitement est fortement parallèle en interne, mais chaque étape dépend de la précédente pour ses entrées. D'autre part, ce type de traitement est appliqué sur de gros volumes de données, et il est souvent impossible de stocker en mémoire toutes les données d'entrée avant de les filtrer, ou même de garder en mémoire toutes les données retenues après filtrage.

L'environnement *Hadoop* s'appuies fortement sur des fichiers temporaires pour résoudre ce problème d'encombrement mémoire. Chaque tâche *Map* lit une partie des données, les filtre, et évacue rapidement les données retenues directement vers une tâche *Reduce* à travers le réseau d'interconnexion ou bien vers des fichiers temporaires.

Un raisonnement similaire s'applique aux tâches *Reduce*. Il peut toutefois être nécessaire qu'une tâche *Reduce* stocke en mémoire toutes les valeurs associées à une même clé avant de pouvoir réaliser ses calculs sur cette liste de valeurs. Si l'algorithme et le code de la tâche *Reduce* exigent un stockage fort en mémoire, cela peut limiter la taille des données traitées. Une solution consiste à implanter un algorithme *out-of-core*, qui travaille par morceaux puis stocke ses résultats intermédiaires sur disque avant de les relire pour terminer ses calculs. Mais ce genre d'algorithme peut être lent.

Il est donc préférable de déployer des tâches *Reduce* sur plusieurs machines, afin de disposer de toute la mémoire d'une machine pour chaque tâche *Reduce* et de charger chaque tâche *Reduce* de travailler sur moins de clés, donc sur moins de groupes de données, pour mieux répartir et accélérer les calculs.

4.3.4 *Pipelining* de la chaîne *Map-Reduce*

Une implantation efficace de la chaîne de traitement de la figure 4.4 à deux principaux objectifs : réaliser les traitements en limitant l'espace mémoire requis (pour ne pas limiter les volumes de données traitables), et réduire au maximum les temps d'exécution. Une solution consiste donc à *pipeliner* les différentes étapes de la figure 4.4 : traiter les données par morceaux et faire travailler chaque étape de la chaîne en parallèle sur des données successives.

La figure 4.6 illustre cette stratégie. Chaque tâche *Map* lit une partie de ses données sur disque, pendant qu'elle filtre la partie lue précédemment et qu'elle transmet les données retenues à la tâche de redistribution. Cette dernière évacuera alors ces données dès que possible vers les tâches *Reduce* pour libérer la mémoire occupée par la tâche *Map*. Chaque tâche *Reduce* regroupera les données reçues possédant la même clé en les ordonnant au fur et à mesure qu'elle les recevra, pour constituer des paires *clé-liste de valeurs*. En revanche, chaque tâche *Reduce* ne traitera ces nouvelles paires que lorsqu'elles seront complètes, c'est-à-dire lorsque toutes paires clé-valeur de sortie des tâches *Map* auront été produites et routées. Evidemment, chaque étape ou sous-étape ne durera pas autant que les autres, contrairement au chronogramme quasi-idéal de la 4.6. Mais une exécution en pipeline est quand même plus efficace qu'une exécution séquentielle des tâches *Map*, suivie de la redistribution des données, suivie d'une exécution séquetielle des tâches *Reduce*.

Remarque : *Ce schéma de* pipelining *peut être implanté par un mécanisme de producteur-consommateur au sein des tâches* Map *et* Reduce*. Dans une tâche* Map *les données lues sur disque sont alors stockées dans un buffer mémoire par un thread producteur, et retirées par un thread consommateur pour être filtrées. Les données retenues sont à leur tour stockées dans un buffer ou dans des fichiers pour être récupérées par la tâche de redistribution, qui les envoie à son tour vers les différentes tâches* Reduce*, etc. Pour être efficace, la tâche de redistribution et de routage doit également réaliser un maximum de communications en parallèle.*

4.4 Impact de la volumétrie des données

Le traitement de grosses volumétries de données impose d'exploiter plusieurs nœuds de stockage, éventuellement localisés sur plusieurs sites de production ou de stockage de données. Dès lors, certains choix ont été fait pour obtenir des traitements de données efficaces.

Amener les traitements aux données. Les processeurs étant capables de traiter des données beaucoup plus vite que les réseaux ne peuvent les transporter, et les données brutes (avant filtrage) étant souvent très volumineuses, il est préférable d'amener des codes de filtrage et de lancer des tâches *Map* sur les nœuds de données plutôt que de déplacer les données brutes vers des nœuds de calcul. On transforme donc des nœuds de stockage en nœuds de calcul le temps de filtrer les données brutes, et on doit gérer le déploiement d'une application distribuée sur un ensemble de nœuds de stockage/calcul.

Organiser des traitements par blocs. Traiter de grosses volumétries de données signifie souvent manipuler des données plus volumineuses que la mémoire d'une machine, et même que la somme des mémoires de tous les nœuds de stockage utilisés. Il est donc nécessaire de concevoir le plus possible des *traitements par morceaux* qui évacuent régulièrement leurs résultats vers des fichiers temporaires ou directement vers d'autres nœuds de traitement.

C'est ce que font les mécanismes internes d'*Hadoop* dans l'exécution des tâches *Map* sur des blocs de données d'entrée. Dans les tâches *Reduce* d'*Hadoop*, qui traitent d'importantes listes de valeurs, c'est au développeur qu'il incombe de ne pas saturer la mémoire des machines utilisées. Dans d'autres environnement *Map-Reduce* les traitements *Reduce* se font par parties sur chaque

liste de valeurs pour éviter les débordements mémoire et sans passer par des fichiers temporaires, mais imposent plus de contraintes algorithmiques aux développeurs.

Paralléliser efficacement les traitements. Afin de traiter de gros volumes de données en temps raisonnable, il est nécessaire de paralléliser et de distribuer les traitements sur un grand nombre de machines. Mais les machines étant des nœuds de données à base de PC standards reliés par un réseau standard, pas des nœuds de calculs interconnectés par des réseaux très performants, il est préférable d'adopter un schéma de parallélisation maximisant les traitements locaux, et recouvrant les calculs et les communications.

Dans le cas du *Map-Reduce* d'*Hadoop*, il n'y a pas de communications entre les tâches *Map*, ni entre les tâches *Reduce*, qui sont donc des tâches de calculs locaux. De plus l'étape de *Shuffle & Sort* entraîne beaucoup de communications mais en partie recouvertes par les tâches *Map* et par la réorganisation des données en entrée de chaque tâche *Reduce*, grace au *pipelining* d'*Hadoop*.

Pipeliner les traitements et les communications. Pour des raisons de performances il est souhaitable de recouvrir les communications avec les calculs des tâches amont et aval, quand cela est possible, afin de masquer les coûts de communication. Le mécanisme de *Map-Reduce* d'*Hadoop* le fait en pipelinant une partie des traitements et les communications.

Rendre simples les développements applicatifs. Au final, l'architecture logicielle distribuée obtenue (chaîne de *Map-Reduce*) est complexe, mais doit être exploitée par des *data scientists* et non pas par des spécialistes du calcul parallèle. Dans cette optique plusieurs environnements logiciels, comme *Hadoop*, *Spark*, *MongoDB*... permettent à des *data scientists* d'injecter facilement leurs codes *Map* et *Reduce* dans une architecture logicielle qui prend déjà en charge tous les aspects d'informatique distribuée.

4.5 Exercices

Bibliographie

[1] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[2] B. Azarmi. *Scalable Big Data Architecture*. Apress, 2016.

[3] R. Bruchez. *Les bases de données NoSQL et le Big Data*. Eyrolles, 2ème edition, 2016.

[4] R. Bruchez and M. Lutz. *Data science : fondamentaux et études de cas*. Eyrolles, 2015.

[5] B. Chapman, G. Jost, R. Van Der Pas, and D. Kuck. *Using OpenMP*. The MIT Press, 2008.

[6] K. Chodorow. *MongoDB, the Definitive Guide*. O'Reilly, 2ème edition, 2013.

[7] K. Dowd and Ch. Severance. *High Performance Computing*. O'Reilly, 2nd edition, 2008.

[8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1999.

[9] J.L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31 :532–533, 1988.

[10] H. Karau, A. Konwinski, P.Wendell, and M.Zaharia. *Learning Spark*. O'Reilly, 1st edition, 2015.

[11] H. Karau and R. Warren. *High Performance Spark*. O'Reilly, 1st edition, 2017.

[12] M. Kirk. *Thoughtful Machine Learning with Python*. O'Reilly, 2017.

[13] P. Lemberger, M. Batty, M. Morel, and J-L. Raffaelli. *Big Data et Machine Learning*. Dunod, 2015.

[14] D. Miner and A. Shook. *MapReduce Design Patterns*. O'Reilly, 2013.

[15] T. White. *Hadoop. The definitive Guide*. O'Reilly, 3rd edition, 2013.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets : A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.