THÈSE DE DOCTORAT DE
TÉLÉCOM PARISTECH

Spécialité Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Matthieu DURUT

Pour obtenir le grade de

DOCTEUR de TÉLÉCOM PARISTECH

Sujet de la thèse :

Algorithmes de classification répartis sur le cloud

soutenue le 28 septembre 2012 devant le jury composé de :

M. Ludovic Denoyer	Maître de conférences, Université Paris VI	Examinateur
M. Frédéric Magoulès	Professeur, École Centrale Paris	Rapporteur
M. Laurent Pautet	Professeur, Télécom ParisTech	Examinateur
M. Fabrice Rossi	Professeur, Université Paris I	Directeur de thèse
M. Michel Verleysen	Professeur, Université catholique de Louvain	Rapporteur
M. Joannès Vermorel	Fondateur de la société Lokad	Examinateur
M. Djamal Zeghlache	Professeur, Télécom SudParis	Examinateur

Remerciements

Mes premiers remerciements vont à Fabrice Rossi, mon directeur de thèse, et Joannès Vermorel, mon encadrant au sein de la société Lokad. Par ses très vastes connaissances en statistiques et en informatique, Fabrice est depuis le départ un interlocuteur indispensable qui m'a sans cesse apporté une vision claire des enjeux et des difficultés dans lesquels j'évoluais. Cette connaissance pluri-disciplinaire, rare, m'a permis d'avancer sur le sujet à cheval entre deux mondes qu'est le clustering réparti. Joannès quant à lui, par son dynamisme et son intelligence quasi-visionnaire des enjeux industriels et techniques, a su apporter à ces travaux la dimension applicative sans laquelle la statistique n'est rien à mes yeux.

Une troisième personne a joué un rôle crucial dans cette thèse, c'est Benoît Patra. Benoît, ami d'école avant d'être collègue, a été en thèse à mes côtés à Lokad. Benoît est l'ami sur lequel j'ai compté dans la difficulté. Son intransigeance intellectuelle et sa puissante volonté font de lui un collaborateur exceptionnel avec lequel j'ai pris beaucoup de plaisir à travailler.

Je remercie vivement la société Lokad de m'avoir proposé cette thèse CIFRE. Les thèses CIFRE ont souvent mauvaise réputation. Je n'aurais pas voulu la mienne autrement. Lokad m'a fourni un excellent cadre de travail. En plus d'un soutien matériel (notamment sur les amples ressources de cloud englouties lors de mes nombreux mois d'expérimentation), Lokad m'a laissé un temps précieux pour mes travaux plus académiques. Les échelles de temps en recherche académique et dans une startup sont difficilement conciliables. Fabrice et Joannès ont su trouver un juste milieu. Mes pensées vont également à mes autres collègues de bureau : Hamza, Benjamin, Estelle, Rinat et Christophe.

Mes remerciements vont ensuite à Michel Verleysen et Frédéric Magoulès pour m'avoir fait l'honneur de rapporter ce travail. Je remercie également Djamal Zeghlache, Ludovic Denoyer et Laurant Pautet pour leur participation à ce jury.

Je souhaite également saluer la bienveillance de Gérard Biau, qui m'a fourni des conseils éclairés tout au long de cette thèse.

En dehors de mes activités directes de recherche, j'ai pris beaucoup de plaisir

pendant ces trois années à discuter de sujets techniques —souvent plus que de raison autour de bières— avec mes amis Charles Binet, Jérémie Bonnefoy, Arnaud Brothier, Matthieu Cornec, Guillaume Delalleau, Oriane Denizot, Xavier Dupré, Rénald Goujard, Jérémie Jabuckowitz, Pierre Jacob, Julien Lecoeur, Gwénolé Lemenn, Fantine Mordelet, Adrien Saumard, Baptiste Rambaud et David Sibaï. J'espère les retrouver au pot de soutenance.

Enfin, je remercie tous mes proches, parents et amis, notamment ma tante Colette Millot et ma femme Elodie, qui a accompagné mes joies et mis en déroute les nombreux doutes que ne manque pas de soulever le travail de thèse.

Aknowledgement

In addition to all the the people thanked above, I would like to express a special thanks to Lokad and the Microsoft Azure team for the material support I have been provided with. During my PhD, I have been using for research purpose between 120.000 and 130.000 hours of CPU. This heavy consumption has been graciously undertaken by Lokad and the Microsoft Azure team. May they be thanked for this.

Résumé

Les thèmes de recherche abordés dans ce manuscrit sont inspirés et motivés de problèmes concrets rencontrés par la société Lokad. Ils ont trait principalement à la parallélisation d'algorithmes de classification non-supervisée (clustering) sur des plateformes de Cloud Computing. Le chapitre 2 propose un tour d'horizon de ces technologies. Nous y présentons d'une manière générale le Cloud Computing comme plateforme de calcul. Le chapitre 3 présente plus en avant l'offre cloud de Microsoft : Windows Azure. Le chapitre suivant analyse certains enjeux techniques de la conception d'applications cloud et propose certains éléments d'architecture logicielle pour de telles applications. Le chapitre 5 propose une analyse du premier algorithme de classification étudié : le Batch K-Means. En particulier, nous approfondissons comment les versions réparties de cet algorithme doivent être adaptées à une architecture cloud. Nous y montrons l'impact des coûts de communication sur l'efficacité de cet algorithme lorsque celui-ci est implémenté sur une plateforme cloud. Les chapitres 6 et 7 présentent un travail de parallélisation d'un autre algorithme de classification : l'algorithme de Vector Quantization (VQ). Dans le chapitre 6 nous explorons quels schémas de parallélisation sont susceptibles de fournir des résultats satisfaisants en terme d'accélération de la convergence. Le chapitre 7 présente une implémentation de ces schémas de parallélisation. Les détails pratiques de l'implémentation soulignent un résultat de première importance : c'est le caractère en ligne du VQ qui permet de proposer une implémentation asynchrone de l'algorithme réparti, supprimant ainsi une partie des problèmes de communication rencontrés lors de la parallélisation du Batch K-Means.

Mots clés : calcul réparti, méthodes de clustering, K-Means, quantification vectorielle, asynchronisme, algorithmes en ligne, Cloud Computing, Windows Azure, descente de gradient répartie.

Abstract

The subjects addressed in this thesis are inspired from research problems faced by the Lokad company. These problems are related to the challenge of designing efficient parallelization techniques of clustering algorithms on a Cloud Computing platform. Chapter 2 provides an introduction to the Cloud Computing technologies, especially the ones devoted to intensive computations. Chapter 3 details more specifically Microsoft Cloud Computing offer : Windows Azure. The following chapter details technical aspects of cloud application development and provides some cloud design patterns. Chapter 5 is dedicated to the parallelization of a well-known clustering algorithm: the Batch K-Means. It provides insights on the challenges of a cloud implementation of distributed Batch K-Means, especially the impact of communication costs on the implementation efficiency. Chapters 6 and 7 are devoted to the parallelization of another clustering algorithm, the Vector Quantization (VQ). Chapter 6 provides an analysis of different parallelization schemes of VQ and presents the various speedups to convergence provided by them. Chapter 7 provides a cloud implementation of these schemes. It highlights that it is the online nature of the VQ technique that enables an asynchronous cloud implementation, which drastically reduces the communication costs introduced in Chapter 5.

Keywords: distributed computing, clustering methods, K-Means, vector quantization, asynchronous, online algorithms, Cloud Computing, Windows Azure, parallel gradient descent.

Contents

Chapitre 1

Introduction

1.1 Contexte scientifique

Dans un article publié dans la revue américaine Nature en 1960 ([113]), Eugène Wigner s'intéresse à la capacité surprenante des mathématiques à formaliser, de manière souvent très concise, les lois de la nature. Un récent article intitulé « The unreasonable effectiveness of data » ([63]) en propose un éclairage nouveau. Reprenant avec ironie une partie du titre de l'article d'Eugène Wigner, ses auteurs défendent la thèse selon laquelle les comportements humains ne se modélisent pas comme les particules élémentaires, et que des formules ne peuvent donner de réponse satisfaisante qu'à peu de problèmes dans lesquels des facteurs humains ou économiques ont un rôle important. Les auteurs proposent plutôt d'adopter un point de vue plus centré sur les données que sur un modèle spécifique. Forts de leur expérience appliquée chez Google, ils décrivent comment des algorithmes simples appliqués à des bases de données gigantesques peuvent fournir de meilleurs résultats que des algorithmes plus fins, appliqués à des bases de données plus petites.

Cet article illustre un phénomène nouveau : alors que de nombreux champs du savoir sont limités par un manque de données pour confirmer ou infirmer des théories, d'autres champs sont désormais confrontés au problème inverse qui consiste à réussir à exploiter des masses de données très volumineuses, souvent désignées par le terme « Big Data ». Certains travaux avancent même que la manipulation et l'utilisation intelligente de ces immenses jeux de données pourrait devenir un nouveau pilier de la recherche scientifique au même titre que la théorie, l'expérimentation et la simulation ([68]). D'autres travaux encore, comme ceux de Lin et Dyer dans [81], soulèvent l'hypothèse que certains algorithmes, incapables de passer à l'échelle et de s'appliquer sur des gros volumes de données, risquent d'être

délaissés par les praticiens et de se retrouver réduits au statut d'algorithmes jouets.

En statistiques comme ailleurs, le lien avec l'informatique se resserre donc, ouvrant des enjeux inter-disciplinaires en partie mésestimés il y a encore quelques années. Sur un plan théorique, cette multi-disciplinarité est déjà représentée en statistiques par des branches comme l'apprentissage statistique (machine-learning) ou les statistiques bayésiennes. Sur un plan pratique, elle était jusqu'à récemment la chasse gardée de géants tels que Google (20 Pétaoctets de données analysées par jour en 2008 selon [48]), de Youtube (2 milliards de vidéos visionnées par jour dès 2010 selon [17]), ou encore le projet européen du Large Hadron Collider (15 Pétaoctets par an). Elle s'est démocratisée et touche aujourd'hui un public bien plus large et se retrouve même au coeur des enjeux technologiques de nombreuses startups.

Comment gérer ces quantités phénoménales de données et de calculs ? Une réponse possible est celle de répartir les tâches sur un ensemble d'unités de calcul et de stockage plutôt que de se restreindre à une seule machine. Comme expliqué dans l'ouvrage de Lin et Dyer ([81]), cette idée n'est pas nouvelle : en 1990, Leslie Valiant dans [107] faisait déjà le constat que l'avènement annoncé du calcul parallèle n'avait pas encore eu lieu. Bien que la démocratisation des algorithmes répartis soit annoncée partiellement en vain depuis des décennies, certains éléments portent à croire que s'entame actuellement ce phénomène. Tout d'abord, l'affaiblissement sensible des progrès dans la cadence des processeurs ne permet plus de résoudre les problèmes logiciels par l'attente de dispositifs matériels plus performants. Ensuite, la volonté de diminuer les consommations énergétiques des unités de calcul (pour améliorer l'autonomie mais aussi diminuer les coûts) tend à multiplier les coeurs des processeurs plutôt que leur cadence (on retrouvait déjà ce phénomène par exemple dans l'Amiga ou plus récemment dans de nombreux smartphones qui multiplient les processeurs dédiés). D'un point de vue pratique, la solution du calcul réparti sur de nombreuses machines est d'ailleurs celle retenue le plus souvent par les géants cités précédemment.

La recherche et l'enseignement en statistiques devraient donc offrir une place toujours plus grande à l'étude d'algorithmes répartis dans les années à venir. Cette thèse s'inscrit dans cette thématique et a pour objet l'étude de la parallélisation de certains algorithmes de classification non-supervisée (clustering) [1].

1. Par la suite, nous désignerons ces problèmes par le simple terme de classification.

1.2 Contexte de la thèse

Cette thèse est le fruit d'une collaboration entre la société Lokad et Télécom ParisTech.

Lokad est une jeune société éditrice de logiciels, spécialisée dans la prévision statistique de séries temporelles. Ces prévisions sont vendues à des entreprises pour gérer et optimiser leurs flux de clients, d'appels ou de stocks. Les clients principaux de Lokad sont les acteurs de la grande distribution, les e-commerces, etc. Pour rendre facilement exploitables ces prévisions, Lokad propose des logiciels en mode *Software as a Service* (SaaS), c'est-à-dire des applications hébergées par ses soins et accessibles directement depuis Internet, qui utilisent ces prévisions pour fournir des outils d'aide à la décision. Lokad fournit par exemple, via son application SalesCast, un système de gestion des stocks en déterminant les niveaux de stock à maintenir et les réapprovisionnements à effectuer. Lokad propose également, via son application ShelfCheck, un outil de détection d'indisponibilité d'un produit en rayonnage, reposant lui aussi sur des prévisions de ventes de produits.

Deux caractéristiques de Lokad inscrivent cette société dans les thèmes de calcul intensif et donc parallèle.

La première de ces caractéristiques est le caractère SaaS des logiciels développés par Lokad. Une utilisation classique des services de Lokad se déroule de la manière suivante : une application SaaS se charge d'accéder à la base de données du client ; cette application envoie les données via le réseau sur les serveurs de Lokad ; le moteur interne de prévision statistique s'applique ensuite aux données pour fournir des prévisions ; les prévisions sont enfin retournées au client via le réseau. Cette approche permet à Lokad d'avoir un contrôle fin sur la qualité des prévisions qu'elle délivre, mais elle implique que l'intégralité des tâches de calcul soit effectuée par les serveurs de Lokad.

La seconde de ces caractéristiques est la gestion automatisée des prévisions. Lokad a développé un moteur interne de prévision de séries temporelles qui ne requiert pas de travail direct d'un statisticien : pour chaque série temporelle à prévoir, ce moteur évalue la précision de chacun des modèles disponibles en production par validation croisée, et sélectionne automatiquement les prévisions du modèle le plus précis pour la série. L'automatisation des prévisions permet à Lokad de répondre à la demande de certaines entreprises dont la taille des données rendait très difficile des prévisions plus manuelles. C'est le cas par exemple de la grande distribution, où chaque enseigne possède des centaines de points de vente,

et où le nombre de produits par point de vente peut atteindre 50 000 références. Cette automatisation implique également que la quantité de calcul à fournir soit au moins proportionnelle au nombre de séries temporelles à prévoir, mais aussi au nombre de modèles et au nombre de points de validation croisée par modèle. Ainsi l'automatisation vient au prix de calculs bien plus intensifs.

Parallèlement, Télécom ParisTech est un acteur européen de référence dans les domaines des Sciences et Technologies de l'Information et de la Communication (STIC). Télécom ParisTech est impliqué dans de nombreux travaux ayant trait à des questions statistiques sur des jeux de données volumineux. Par exemple, Télécom ParisTech et EDF ont eu un partenariat de recherche, concrétisé par le laboratoire commun BILab, spécialisé dans les technologies de l'aide à la décision, égalements appelées Business Intelligence (BI). En particulier, le BILab s'est intéressé aux questions afférentes à la prévision statistique. En raison de la taille d'EDF, ces questions qui étaient à l'origine statistiques sont à présent aussi liées à l'informatique et à la nécessité de gérer d'importants volumes de données et des calculs coûteux.

Télécom ParisTech et Lokad se trouvent donc tous les deux à l'intersection entre informatique et statistiques, à l'endroit même où les deux disciplines se rejoignent sur les problèmes d'algorithmes statistiques répartis.

Le concept récent de Cloud Computing (francisé en « informatique dans les nuages ») est en train de modifier profondément le monde informatique. Le Cloud Computing propose une transition depuis le modèle économique dans lequel l'utilisateur possède les logiciels et les infrastructures matérielles vers un modèle dans lequel l'utilisateur est un simple locataire de services, qu'ils soient logiciels ou matériels. Le Cloud Computing abstrait et dématérialise donc l'espace physique dans lequel les calculs et les données sont gérés et ouvre des perspectives nouvelles pour les entreprises mais aussi pour la recherche académique.

Notre travail de thèse s'inscrit à l'intersection de ces différents sujets : statistiques, calcul réparti et Cloud Computing. Nos activités de recherche ont porté sur cette technologie naissante pour laquelle peu de travaux académiques ont à ce jour été réalisés. Notre travail a consisté à explorer les capacités et limites de ces technologies encore mal comprises, pour en appréhender les enjeux et proposer des manières pertinentes de paralléliser des algorithmes. Ce travail d'exploration et de synthèse donne naissance aux chapitres 2 et 3.

Les algorithmes que nous avons portés sur le cloud se répartissent en deux catégories. La première catégorie regroupe différents composants du moteur de

prévision interne de Lokad que nous avons adaptés pour le cloud. Ces travaux, moins académiques, sont d'autant plus difficiles à partager que le code qui en découle est propriétaire. Ils nous ont cependant permis par induction de dégager quelques considérations générales sur la conception d'applications sur le cloud. Nous en présentons certaines dans le chapitre 4.

La seconde catégorie d'algorithmes a trait aux méthodes de classification non-supervisée réparties, qui font l'objet principal des chapitres 5 à 7 du présent document. La question de la classification est un aspect très important de la prévision de séries temporelles chez Lokad. En effet, pour améliorer ses prévisions, Lokad utilise des méthodes dites « multi-séries ». L'objet de ces méthodes est d'utiliser l'information contenue dans un groupe de séries homogènes afin d'affiner la prévision de chacune d'entre elles. Un exemple d'utilisation est l'exploitation des saisonnalités : en dégageant des comportements saisonniers d'un ensemble de séries temporelles, certains modèles de Lokad sont capables d'affiner la composante saisonnière, plus bruitée sur une seule série temporelle. La classification est donc un outil essentiel pour Lokad, outil qu'il était nécessaire d'adapter aux plateformes de Cloud Computing.

Les algorithmes de classification répartis considérés dans ce manuscrit mettent en lumière certaines des contraintes imposées par les plateformes de Cloud Computing, notamment en terme de communication. Le chapitre 5 traite de la parallélisation d'un algorithme bien connu de classification : le Batch K-Means. La version répartie de cet algorithme nécessite des communications importantes entre les différentes machines sur lesquelles il est parallélisé, ce qui limite en partie sa rapidité. Les deux chapitres suivants traitent d'un autre algorithme, cousin germain du Batch K-Means : l'algorithme de Vector Quantization (VQ). Cet algorithme, qui peut être considéré comme la version en-ligne du Batch K-Means, permet de contourner certains problèmes de communication inter-machines par l'introduction d'asynchronisme.

Les travaux scientifiques portant sur des calculs répartis de grande ampleur sont appliqués à des jeux de données dont l'échelle de taille peut se révéler très variable. Pour chaque ordre de grandeur du problème, les outils logiciels et matériels adéquats sont très différents ; on conçoit aisément qu'à des problèmes aussi différents en taille que l'indexation du web par Google Search ou l'affluence de trafic sur un petit site web, des solutions très distinctes sont fournies. Il est ainsi primordial de préciser quels sont les ordres de grandeur pour lesquels un travail donné est conçu. Nos travaux de thèse portent sur des calculs dont l'ampleur correspondrait sur une seule machine (mono-coeur) à une centaine d'heures, avec à notre disposition 250 machines. Les jeux de données les plus volumineux sur

lesquels nous avons fait tourner nos algorithmes représentent environ 50 Giga-octets. Nous avons choisi de concentrer nos travaux sur ces ordres de grandeur qui reflètent la taille des données des clients les plus importants actuellement de Lokad.

La suite de cette partie introductive présente succinctement le contenu de chaque chapitre de ce manuscrit.

1.3 Présentation des travaux

1.3.1 Chapitre 2 - Introduction au Cloud Computing

Les progrès importants des mécanismes de collecte des données n'ont pas été accompagnés de progrès aussi rapides dans le développement des processeurs. Cette réalité a incité au développement de systèmes physiques et logiciels permettant de répartir des charges de travail sur de multiples unités de calcul. Ces calculs intensifs ont tout d'abord été portés sur des architectures physiques dédiées, communément appelées super-calculateurs. Ces super-calculateurs étaient des dispositifs physiques conçus en faible quantité, spécifiquement pour réaliser des calculs intensifs. L'explosion du marché des ordinateurs personnels dans les années 80, puis d'Internet à la fin des années 90 a ouvert des perspectives nouvelles quant à la manière de répartir des calculs. De nouveaux systèmes sont alors apparus, reposant sur la collaboration de plusieurs agents administrativement distincts et mettant en commun une partie de leurs ressources. Parmi ces systèmes, on trouve des infrastructures de Grid Computing, comme Condor ([105]), ou des systèmes plus récents, par exemple de Peer-To-Peer, comme Napster ou Folding@Home.

La parallélisation de calculs sur une vaste quantité de machines soulève de nombreuses difficultés, qu'elles aient trait à la communication entre les machines, à l'accès en écriture ou lecture à une mémoire partagée efficace, ou à la répartition de la charge de calcul sur les différentes unités disponibles. Les années 2000 ont vu l'émergence d'applications Internet consommant d'immenses ressources : c'est le cas par exemple de Google Search, de Bing, de Facebook, de Youtube, d'Amazon, etc. Les entreprises à la tête de ces applications ont développé des environnements logiciels (frameworks) mais aussi physiques (via la construction de centres de calcul spécifiques ou data centers) pour proposer des solutions aux difficultées susnommées.

Certaines de ces entreprises, rejointes par d'autres acteurs économiques, ont alors

proposé de mettre en location un accès à ces différentes solutions, physiques et logicielles. Ces sociétés sont alors devenues des fournisseurs de Cloud Computing.

Le Cloud Computing a de très nombreuses implications techniques, scientifiques ou commerciales. Les différentes offres se distinguent selon qu'elles offrent des solutions clef en main, ou au contraire qu'elles laissent plus de libertés à leurs utilisateurs au prix d'une complexité d'utilisation plus élevée. Parmi les solutions clef en main, on recense les offres de Software as a Service (SaaS), comme Google Search, Gmail, Deezer, Facebook, etc. On y regroupe également de nombreuses solutions spécifiques, développées pour chaque client par des sociétés tierces (notamment des sociétés de services en ingénierie informatique (SSII)). Ce chapitre présente principalement la catégorie d'offres de Cloud Computing les plus à même de permettre l'usage de calculs intensifs. Cette catégorie d'offres est souvent celles des acteurs clouds les plus importants. Plus standardisées et orientées performance, leurs offres s'adressent à un public de développeurs exclusivement. Parmi elles, nous approfondirons les différences entre les offres de Platform as a Service (PaaS) et celles d'Infrastructure as a Service (IaaS).

Les offres de Cloud Computing analysées proposent un ensemble de technologies, parfois pré-existantes au concept de Cloud Computing, réunies dans une même pile technologique. Le chapitre 2 traite de ces différentes technologies. En particulier, nous donnons un apercu des difficultés que soulèvent un système de stockage de données réparti sur un ensemble de machines distantes. Nous présentons également certains des environnements logiciels construits pour aider le développement d'applications réparties, comme le célèbre MapReduce, conçu originellement par Google et amplement repris.

1.3.2 Chapitre 3 - Introduction à Azure

Microsoft est une société dont les différentes filiales ont d'importants besoins en matériel informatique : c'est le cas par exemple pour une partie du moteur de recherche Bing ou pour le stockage de certaines données comme celles liées à la plateforme de jeu en ligne de la XBox 360. Ces larges besoins internes ont amené Microsoft à fournir des efforts de recherche et développement dans les infrastructures matérielles et logicielles pour disposer d'une plateforme de calcul et de stockage puissante. Ces efforts ont abouti par exemple sur des composants logiciels comme Cosmos, Scope ([38]), Dryad ([73]) ou DryadLINQ ([116]).

A la suite d'Amazon et de Google, Microsoft a utilisé les outils développés pour ses besoins internes pour devenir fournisseur de Cloud Computing. Son offre

cloud se répartit en deux grandes familles de produits. La première regroupe les applications classiques de Microsoft auparavant proposées en mode « desktop » et aujourd'hui disponibles en version cloud. Parmi elles, on peut citer Office 365, qui est la version SaaS du pack Office. Les offres de SQL Server ont également été portées sur le cloud pour prendre le nom de SQL Azure. Ces migrations sur le cloud ne se font pas sans certaines difficultés liées au contexte de l'activité de Microsoft : les offres SaaS de Microsoft font en quelque sorte concurrence à leur pendant desktop, mais elles modifient également le positionnement de Microsoft par rapport à certains de ses partenaires dont l'activité principale consistait à garantir la bonne installation et la maintenance de ces applications desktop chez les clients de Microsoft. Cependant Microsoft a affiché sa volonté de développer ces offres qui représenteront très probablement une part importante de son activité dans les prochaines années.

La deuxième famille de produits est une plateforme de Cloud Computing destinée aux éditeurs logiciels, appelée Azure. Cette plateforme permet à ses clients d'héberger des applications en mode SaaS, qui utilisent les ressources de CPU, de stockage et de bande-passante fournies par Azure. Microsoft a réutilisé une partie de la technologie développée dans Cosmos (comme expliqué dans [36]) pour construire un système de stockage dédié pour cette plateforme, appelé Azure Storage.

Le chapitre 3 présente les caractéristiques et les enjeux techniques d'Azure. En particulier, nous nous intéressons aux abstractions de stockage proposées, ainsi qu'à la manière dont elles se sont positionnées par rapport aux différents compromis nécessaires aux solutions de stockage réparties, comme introduit dans le chapitre 2. Nous présentons également quelques clefs du développement d'applications sur cette plateforme.

1.3.3 Chapitre 4 - Éléments de conception logicielle sur le cloud

Le chapitre 4 présente certaines techniques ou schémas récurrents (dits « design pattern ») rencontrés pendant le développement de nos applications sur le cloud. Ces techniques ont été bien souvent au coeur du développement des quatres applications sur lesquelles nous avons travaillé durant cette thèse. Les deux premières applications sont les implémentations cloud de l'algorithme réparti du Batch K-Means d'une part et de l'algorithme réparti de Vector Quantization d'autre part, respectivement étudiés dans les chapitres 5 et 7.

Parallèlement à ces algorithmes, nous avons porté sur le cloud deux applications qui correspondent au coeur de la technologie de prévision de Lokad. La première de ces deux applications est le moteur de prévision de séries temporelles de Lokad. Ce moteur, utilisé en production pour fournir via son API les prévisions utilisées ensuite dans nos applications-clients, profite de la fiabilité et de l'élasticité proposées par Azure. La seconde application, Lokad Benchmark, est un outil interne qui réplique le moteur de prévision. Cet outil est utilisé pour améliorer la précision statistique de nos modèles, pour augmenter le potentiel de passage à l'échelle de notre moteur sur de plus gros jeux de données, et pour surveiller et « profiler » l'exécution de prévisions.

Ces quatres applications s'inscrivent dans des contextes différents (prototype/en production) qui influencent fortement le positionnement de l'application quant à de nombreux compromis : robustesse/efficatité, passage à l'échelle/facilité de maintenance, rapidité/précision, etc. À la lumière de ces quatre applications, nous présentons comment les différentes abstractions et primitives fournies par Azure peuvent être combinées pour bâtir des applications sur Azure. Nous soulignons également l'absence de certaines de ces briques « élémentaires », et les conséquences sur le design induites par ces absences.

1.3.4 Chapitre 5 - Algorithmes de Batch K-Means répartis

Les algorithmes de classification (toujours au sens clustering) ont un rôle central en statistiques. D'un point de vue théorique, ils représentent un problème d'apprentissage non supervisé très étudié. D'un point de vue pratique, ils sont un outil souvent indispensable pour l'exploration des données. Ces algorithmes proposent de résumer un jeu de données en un jeu de données plus petit mais cependant représentatif des données initiales. Ce résumé est réalisé en constituant des sous-groupes (également appelés « clusters »), déduits du jeu initial par le regroupement des données les plus proches suivant un certain critère de similarité.

Etant donnés un critère de similarité et un nombre fixé de sous-groupes, le problème de trouver un regroupement optimal au sens du critère est un problème calculatoirement très difficile, souvent irréalisable dans la pratique dès que la taille du jeu de données dépasse quelques dizaines de points [2]. Des algorithmes d'approximation ont été proposés, fournissant une solution proche de la solution

2. À titre d'exemple, le problème du K-Means théorique comme exposé dans [31] est NP-complet, même dans le cas le plus simple de la classification dans le plan (voir [87])

optimale. Parmi ces algorithmes, le Batch K-Means est un algorithme populaire, connu notamment pour sa simplicité de mise en oeuvre.

Les algorithmes de classification comme le Batch K-Means sont utilisés dans des domaines très variés, qu'il s'agisse par exemple de biologie moléculaire, d'analyse de séries temporelles ou d'indexation du web pour les moteurs de recherche, etc. Chez Lokad, ces procédures sont utilisées pour former des sous-groupes au sein desquels les séries temporelles partagent un même comportement saisonnier. Il est alors plus aisé pour Lokad d'extraire une composante saisonnière de chaque sous-groupe plutôt que de chaque série temporelle, car la composante saisonnière, moyennée sur ce sous-groupe, se révèle souvent plus régulière que lorsqu'elle est extraite série par série.

Les algorithmes de classification connaissent depuis une quinzaine d'années le développement d'un nouveau défi : celui de la parallélisation. En effet, l'évolution plus rapide des moyens de collecte des données que des moyens de calcul a conduit les statisticiens à paralléliser leurs tâches de calcul sur de nombreuses machines, mêlant ainsi toujours plus statistiques et informatique.

Les travaux du chapitre 5 portent sur le problème du Batch K-Means réparti, en particulier son adaptation aux nouvelles architectures de calcul sur le cloud, développées au chapitre 2 et 3. Plus précisément, ils montrent comment le premier cadre théorique de Batch K-Means réparti, proposé par Dhillon et Modha ([51]) dans le cas d'une architecture DMM [3] disposant d'une implémentation MPI [4], ne s'applique que partiellement à Azure parce que ce dernier ne possède pour l'instant pas une telle implémentation. Ce chapitre propose une modification de l'algorithme qui permet de l'adapter à l'infrastructure offerte par Azure, et une modélisation du coût de notre implémentation.

1.3.5 Chapitre 6 - Considérations pratiques pour les algorithmes de Vector Quantization répartis

Ce chapitre ainsi que le suivant présentent des travaux réalisés et écrits en collaboration avec Benoit Patra, qui était également doctorant au sein de la société Lokad. Nous sommes partis du constat du chapitre 5 selon lequel les communications sur

3. Distributed Memory Multiprocessors : architecture parallèle ne disposant pas de mémoire physique partagée.

4. norme définissant une bibliothèque de fonctions pour fournir des moyens de communications entre machines.

une plateforme de Cloud Computing via le stockage réparti sont coûteuses. La parallélisation du calcul d'un Batch K-Means, bien qu'assez satisfaisante, y est donc ralentie par les nombreuses communications nécessaires ainsi que par les processus de synchronisation de toutes les machines. Nous avons donc travaillé sur la parallélisation d'un autre algorithme de classification, connu sous le nom d'algorithme de Vector Quantization (VQ) qui supprime ou limite ces problèmes et se montre donc, selon nous, plus adapté à une implémentation sur le cloud.

Tout comme le Batch K-Means, l'algorithme de VQ permet de calculer des sous-groupes pertinents en affinant des points de référence appelés centroïdes ou prototypes qui représentent les différents sous-groupes. Bottou et Bengio montrent dans [32] que l'algorithme de VQ peut être vu comme la version « en-ligne » du Batch K-Means : au fur et à mesure que l'algorithme de VQ traite des points tirés des données à résumer, il fait évoluer en conséquence les prototypes plutôt que de faire évoluer les points une fois qu'il a examiné toutes les données comme c'est le cas dans le Batch K-Means.

D'un point de vue statistique, le passage à un algorithme « en-ligne » a présenté des difficultés particulièrement intéressantes. En effet, ce type d'algorithme est par essence séquentiel, et la parallélisation du calcul est moins naturelle que dans le cas d'un Batch K-Means, pour lequel une même instruction doit être appliquée indépendamment à de nombreux points (on parle alors de tâches exhibant une propriété dite de « data-level parallelism »). Par certains aspects, l'algorithme de VQ appartient à la grande famille des algorithmes de descente de gradient stochastique. La parallélisation de ce genre d'algorithmes a déjà fait l'objet d'études (nous renvoyons par exemple aux articles [118], [49] et [84]). Dans le cas où le critère est convexe et suffisamment régulier, les stratégies classiques de parallélisation d'algorithmes de descente de gradient par moyennage des résultats des différentes machines mènent asymptotiquement à des accélérations de convergence optimales, comme démontré dans le récent article de Deker et al. ([49]), et obtiennent dans la pratique des résultats très satisfaisants, comme dans l'article de Langford et al. ([118]).

Dans le cas de l'algorithme de VQ, le critère n'est ni convexe ni suffisamment régulier, et le moyennage des résultats ne mène pas dans le cas général à de meilleures performances que l'algorithme séquentiel exécuté sur une seule machine. Ce résultat surprenant nous amène à reformuler l'algorithme proposé sous une forme qui souligne l'importance de la vitesse de décroissance du pas associé à la descente de gradient. Ce travail nous permet de donner un éclairage différent au précédent algorithme pour appréhender les mécanismes qui l'empêchent d'obtenir des accélérations de convergence satisfaisantes.

Enfin, nous proposons un nouvel algorithme de VQ réparti, qui évite les travers du précédent. En particulier, notre algorithme va additionner les termes de descente provenant des différentes machines, plutôt que d'en moyenner les valeurs. Cette modification de l'algorithme permet d'obtenir des accélérations de convergence satisfaisantes. Ces accélérations sont présentées en fin de chapitre.

1.3.6 Chapitre 7 - Implémentation cloud d'un algorithme de Vector Quantization réparti et asynchrone

Dans ce dernier chapitre, nous présentons le projet logiciel Cloud-DAVQ. Ce projet est la mise en oeuvre des algorithmes étudiés dans le chapitre précédent sur la plateforme de Cloud Computing de Microsoft : Windows Azure. Des implémentations ambitieuses d'algorithmes répartis de descente de gradient stochastique ont déjà été proposées, comme c'est le cas dans les articles de Louppe et al., Langford et al., ou Dekel et al. précédemment cités : [84], [118] et [49]. Cependant, ces travaux se sont à notre connaissance restreints au cas plus simple où l'infrastructure de calcul possède une mémoire partagée efficace et où les coûts de communication sont donc très faibles (dans [84] et [49]), ou à un cas où la convexité du critère permet de supprimer les communications inter-machines sauf à la fin de l'algorithme (dans [118]). Ainsi, aucun des travaux précédents ne propose d'exemples d'implémentation d'algorithmes de descente de gradient répartis dans un contexte où les communications sont à la fois nécessairement fréquentes et coûteuses.

D'un point de vue informatique, l'aspect « en-ligne » de l'algorithme de VQ nous a semblé particulièrement pertinent, puisque c'est la suppression du caractère batch qui permet de retirer la nécessité de la synchronisation des machines. Le passage vers un algorithme « en-ligne » parallélisé nous permet donc de construire un algorithme asynchrone, libérant nos machines de processus pénalisant les performances. Ce passage vers l'asynchrone permet dans les faits de supprimer deux mécanismes de synchronisation : le premier a trait aux organisations inter-machines, puisque chaque machine n'est plus bloquée dans ses communications et dans ses calculs par l'attente des résultats des autres unités de calcul. Le second mécanisme que nous supprimons est celui de la séquentialité lecture/calcul/écriture au sein de chaque machine. Par la suite, nous faisons référence à cet algorithme réparti asynchrone sous l'acronyme de Distributed Asynchronous Vector Quantization (DAVQ). Dans le cadre du DAVQ, chaque machine peut effectuer en parallèle (par un recouvrement à base de plusieurs threads) les tâches

de calcul et de communication pour exploiter au mieux les ressources de CPU et de bande-passante.

1.4 Résumé des contributions

Notre apport personnel s'articule autour de trois axes. Le premier a trait à la clarification des usages possibles des plateformes de Cloud Computing en général, et plus particulièrement dans le contexte d'une utilisation à but scientifique du PaaS. En effet, la littérature académique dans le domaine du cloud émerge seulement, et il était difficile de comprendre précisément les capacités, les objectifs, et les limitations techniques du cloud à la seule lumière des documents commerciaux proposés par les fournisseurs de Cloud Computing. Les chapitres 2 et 3 se veulent donc une introduction plus rigoureuse des plateformes cloud actuelles.

Le second axe porte sur l'architecture des programmes cloud imposée par les contraintes technologiques logicielles et matérielles du cloud. En particulier, nous montrons comment les notions d'idempotence ou l'abandon du système ACID amènent à repenser la conception d'algorithmes mathématiques mais aussi d'applications plus générales.

Le troisième axe se penche sur la parallélisation d'algorithmes de classification répartis. Il montre les limitations d'un algorithme connu de classification, le Batch K-Means, et étudie la parallélisation d'un autre algorithme : l'algorithme de VQ. Cet axe a donné lieu à 2 publications à propos de la parallélisation de l'algorithme de Batch K-Means en collaboration avec mon directeur de thèse Fabrice Rossi, et à une troisième publication à propos de la parallélisation de l'algorithme de VQ en collaboration avec Benoit Patra et Fabrice Rossi.

Chapter 2

Presentation of Cloud Computing

2.1 Introduction

For the last twenty years, the technological improvements in computing hardware have led to a situation where a gigantic amount of data can be gathered, stored and accessed. The amount of processing power required to explore and use these data largely exceeds the capacity of a single retail computer (referred to in the following as commodity hardware). Therefore, several computing systems have been designed to tackle this issue. In the 80's, more powerful systems known as supercomputers were created by improving custom hardware: this solution is referred to as scaling-up. The present trend is on the contrary to run these broad computations on a large set of commodity CPU working together, a solution referred to as scaling-out. In other words, processing more data nowadays often consists in throwing more commodity hardware at the problem.

The parallelization of computation on a large amount of machines gives rise to many problems, such as communications between computers, access to shared resources, or workload balance. The past two decades have been constantly providing engineered solutions to these issues, some of them known as Grid Computing and Peer-to-Peer architectures. These solutions have led to well-known software applications or results that have a deep impact on our everyday life, as is the case for Napster, Google Search, social networks like Facebook, etc.

While these considerations were challenging only few people several years ago, the number of software companies involved in large-scale computations is growing quickly. This situation has led to the creation of a new economic market of storage and computation facilities. Some very large software actors have decided to provide these facilities as a commercial service, allowing new players

to outsource their computing solution, making computation and storage a facility as electricity already is. These new commercial services are referred to as Cloud Computing.

The fast-growing interest in Cloud Computing over the past few years has led to a fuzzy and continuously evolving situation: many have heard of it, but few people actually agree on a specific definition. More importantly, even fewer understand how it can benefit them. From our point of view, the best definition of Cloud Computing has been provided by Armbrust et al. in [25]: Cloud Computing "*refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services*".

From a consumer's point of view, Cloud Computing allows any user to rent a large number of computing instances in several minutes to perform data/compute intensive jobs or build web applications. Since such instances can be dynamically provisioned up or down, it allows users to meet specific scale-up constraints, i.e. to be able to be enlarged to accommodate growing amount of work. For example, this scaling elasticity lets consumers face weekly data consumption peaks. The storage and computing capacities are provided as a service in a pay-as-you-go way. Cloud Computing therefore disburdens users from the hardware investment.

Many Cloud Computing solutions have already been developed. There are two categories of cloud solutions. The first category includes all the big Cloud Computing providers, such as Google, Microsoft or Amazon. They often provide the cheapest prices of Cloud Computing solutions, through well-designed but fixed frameworks. The second category gathers all the small Cloud Computing offers. Contrary to the former ones, the small actors provide their customers with custom Cloud Computing solutions. While the storage or CPU costs of these companies usually cannot compete with the big providers' prices, their economic advantage lies in all the specific services provided to their customers with. In particular, the small Cloud Computing providers sometimes embody the ability for companies' top management to challenge or replace some tasks previously reserved to their Information Technology (IT) department, for example in the case of a conflict between the top management and the IT. As a consequence, the Cloud Computing adoption may sometimes be the subject of internal political dealings. The small and custom Cloud Computing offers are therefore of prime interest for many potential customers as they can adapt to specific needs. However, the present chapter does not further detail these offers, as it aims to focus on the most appropriate cloud solutions for intensive computations.

A survey of Cloud Computing realized in 2011 by TNS ([8]) provides first feed-

backs of early cloud solution adopters. The survey was conducted on 3645 IT decision makers in eight different countries. The study states that 33% of companies adopted cloud primarily to access information from any device rather than to cut costs. Indeed, while a large part of these companies saved money thanks to cloud adoption (82%), the savings are on average small (more than half of the companies report less than $20,000 savings). Beyond the reduction of the costs, Cloud Computing can contribute to improve most of the quality of services. 93 % of all the companies interviewed have noted some improvements in their services since their cloud adoption. Among these improvements, the hardware administration and maintenance release is a major element.

Cloud Computing is also a subject of interest for academic researchers. The use of Cloud Computing can provide them with access to large-scale experiment platforms without getting hardware access grant or hardware investments from their administration. All the scientific intensive computing tasks do not fit the Cloud Computing constraints, especially because of the communication throughput and latency constraints. Yet, provided the tasks fit into the scope of Cloud Computing, even scientists with limited experience with parallel systems are provided with the resources of a large distributed system.

One can foresee that the importance of Cloud Computing in the coming years will keep growing, as many companies have expressed their will to resort to these cloud platform solutions. This dynamic is strengthened by the large research and development investments of the major cloud providers. These providers keep investing a lot of work and money to improve their services and tackle economical, ecological and performance challenges. For example, in 2011 Microsoft planned to spend 90 percent of its $9.6 billion research and development budget on cloud strategy (according to an interview given by the Microsoft International President Jean-Philippe Courtois to Bloomberg ([13])).

This chapter is a short overview of the current state of Cloud Computing (in early 2012). It is organized as follows. The second and following section gives a short introduction to the origins of Cloud Computing. The third section is devoted to a more in-depth definition of Cloud Computing. In particular, we compare Cloud Computing and Grid Computing, and give some insights of the different abstraction levels it targets. Cloud storage design is investigated in a fourth section, while the fifth section describes the main cloud execution frameworks.

2.2 Origins of Cloud Computing

2.2.1 HPC and commodity hardware computing

A large part of High Performance Computing (HPC) researches and experiments is implemented on custom HPC hardware, sometimes called supercomputers. While the exact beginning of custom HPC platforms is rather uncertain, it goes back at least to the 1960s and the work of Seymour Cray. These computing platforms have been successfully used in many research fields such as weather forecasting, aerodynamic research, nuclear explosion simulations, molecular dynamics simulation, drug discovery, etc.

During the 80's and 90's, the amount of personal computers all around the world has increased very significantly. The resulting scale economies have led to a situation where it is nowadays often more cost-effective to stack cheap retail CPU than buying expensive dedicated hardware. In parallel with HPC development on supercomputers, research has therefore been made to distribute computations on commodity hardware.

This situation is summarized by Drepper in [52]: *"It is important to understand commodity hardware because specialized hardware is in retreat. Scaling these days is most often achieved horizontally instead of vertically, meaning today it is more cost-effective to use many smaller, connected commodity computers instead of a few really large and exceptionally fast (and expensive) systems"*.

To distribute computations on commodity hardware, initial solutions have been provided to network together distinct commodity computers. For example, the Parallel Virtual Machine (PVM) framework originally written in 1989 is a software system that enables a collection of potentially heterogeneous computers to simulate a single large computer. A second major framework is the Message Passing Interface (MPI) framework, whose version 1.0 was released in 1994. This second framework is more explicitly described in Chapter 5. The hardware systems composed of multiple commodity computers and networked through software as PVM or MPI are often called Beowulf clusters.

A halfway solution between supercomputers and Beowulf clusters was introduced several years ago. This solution mixes the cost efficiency of commodity CPU with the throughput of dedicated hardware such as custom communication bus and ethernet cards. This solution is therefore composed of commodity CPU and RAM memory, along with custom and dedicated power supply devices, motherships, and inter-machines connections. All the different components are fastened on

racks and/or blades, then gathered in rack cabinets.

Grid Computing is a general term referring to hardware and software configurations to distribute computations on commodity CPU, whether in Beowulf clusters or in custom racks and cabinets. The following subsection describes more precisely the Grid Computing frameworks.

2.2.2 Grid Computing

Grid Computing (or grid for short) is a distributed computing system composed of many distinct networked computers (sometimes referred to as nodes). These computers are usually commercial standard hardware gathered into a grid platform. The different components are by design loosely coupled, which means each computer has little knowledge of the existence and identity of the others. A dedicated low-level software layer called middleware is used to monitor and load-balance tasks between the nodes.

Contrary to supercomputers, the network that connects the different devices of a Grid can be rather slow. In the case of supercomputers, all the processors are geographically very close and communication is endorsed through an efficient local computer bus or a dedicated High Performance Network. In the Grid Computing case, communication goes through a Local Area Network (LAN) or a Wide Area Network (WAN), which cannot compete with the throughput of a local bus. This communication constraint on Grid Computing means that Grid is very well-suited to large batch jobs where there are intensive computational requirements and small communication needs between machines.

Grid Computing architectures have been implemented in many different ways, some of them being simple gathering of physical devices where administrators and end users are one and the same people. In other cases, the system is a more complex entity composed of distinct collaborative organizations, referred to as administrative domains. In the latter case, the middleware manages the whole system administration, grants users access to the resources, and monitors all the participants to prevent malfunctioning or malicious code from affecting the results. Depending on the context, the term Grid Computing might refer only to this latter case. One of the first middleware handling of the multiple administrative domains is Condor ([105]).

Grids have been successfully set-up and used in many production fields. For example, it is used by Pratt & Whitney, by American Express, in banks for Value

At Risk estimation, pricing or hedging. It is also used by the National Aeronautics and Space Administration (NASA) in various projects such as the Information Power Grid (IPG) project. Grids have also been provided for academic researches, as it is the case for the Large Hadron Collider (LHC) in the form of the LHC Computing Grid, or for Grid 5000 (see [37]).

One of the most well-known forms of Grid Computing appeared in the 2000s, resulting from two major events of the 90s: the personal computer democratization and the Internet explosion. This new form of Grid Computing is known as the Peer-to-peer architecture, which is a Grid Computing platform composed of a collaborative set of machines dispatched worldwide and called peers. This architecture removes the traditional client/server model: there is no central administration and peers are equally privileged. They share a portion of their own resources (hard-drive, network bandwidth, CPU, etc.) to achieve the tasks required. Peer-to-peer model has been popularized by very large-scale distributed storage system, resulting in peer-to-peer file-sharing applications like Napster, Kazaa or Freenet ([41]). Since then, Peer-to-peer architectures have also been successfully applied on very intensive computational tasks. Some of the largest projects are listed by the Berkeley Open Infrastructure for Network Computing (see [24]) like Seti@Home or Folding@Home (see [78]).

2.2.3 Emergence of Cloud Computing

During the 2000s, the Internet boom urged some behemoth like Google, Facebook, Microsoft, Amazon, etc. to experience computation and storage challenges on data sets of a scale hardly ever met. In addition to the scaling-out of their systems, they faced two additional challenges. Firstly, they needed to create abstractions on storage and distributed computations to ease the use of these systems and allow non-distributed computing experts to run large-scale computations without getting involved in the current parallelization process. Secondly, they were concerned with minimizing the operating cost per machine in their data centers.

Such constraints meant partially dropping the previous mainstream data storage system (this situation is detailed in Section 2.4) along with developing new software and hardware frameworks to easily express distributed computation requests. These companies continued the work on commodity hardware and merged various innovative technologies into coherent technology stacks, on which a short overview is given in Subsection 2.3.3. They built large data centers hosting tens of thousands of commodity machines and used the technology stacks they improved

to meet their internal computations and storage needs.

Initially, the infrastructures and data centers as well as the new storage systems and the software frameworks for distributed computations were designed to be tools exclusively dedicated to the companies internal needs. Progressively, some of these tools have been transformed into public commercial products, remotely accessed through Application Programming Interface (API) as paying services. The companies that made these commercial offers became Cloud Computing providers.

2.3 Cloud design and performance targets

2.3.1 Differences between Cloud Computing and Grid Computing

A classical distinguishing criterion between Cloud Computing and Grid Computing is the uniqueness or multiplicity of administrative domains. Clouds are supposed to be rather homogeneous servers located in data centers controlled by a single organization, while grids are supposed to be a federation of distinct collaborative organizations sharing (potentially heterogeneous) resources. A consequence of the multiple administrative domains in Grid Computing is that grids need to handle the additional constraints of a federated environment, e.g. checking the credentials and the agents' identities, but also monitoring the system to detect malicious uses. These additional constraints, together with the potential lower bandwidth between two distinct administrative domains increase the communication burdens and therefore tend to require coarser granularity for the tasks distributed on the Grid than those distributed on the cloud.

A second aspect of Cloud Computing is the intensive use of Virtual Machines (VM). A VM is the software implementation of a computer that executes programs as if it was a physical machine. VM implementations were already available several years ago, but they were uniquely software based and therefore suffered from a significant impact on their overall performance. Nowadays the VM implementations are both hardware and software based, reducing the loss due to virtualization in a significant way. While the original motivation for virtual machines was the desire to run multiple operating systems in parallel, Cloud Computing is intensively using VM to build a higher degree of abstraction of the software environment from its hardware. The use of VM provides Cloud Computing with desirable features such as application provisioning and maintenance

tools, high availability and easier disaster recovery.

A third difference between grid and cloud is ownership. In most Grid Computing examples, consumers as a group own the resources they use. In the cloud case, resources are totally outsourced to the cloud provider and do not need delegation of authentication or authority: cloud consumers are paying for resources in a pay-as-you-go manner. Rather than accessing resources directly, Cloud Computing consumers access them through a kind of service they pay for. By adopting cloud, computing consumers do not need to invest in and own hardware and infrastructure (data center, power supply, cooling system, etc.) as they would probably need in the Grid Computing case.

A side effect of this ownership shift is multi-tenancy and elasticity. Elasticity is the ability for the cloud consumer to provision up or down the number of computing instances as well as the size of the data storage system. Since different resources are gathered into a specific cloud company actor instead of being held by each customer, the resources are mutualized. Such a mutualization of resources allows elasticity, in the same way the mutualization of the risks allows insurance mechanisms. This dynamic reconfiguration of the resources allows each consumer to adjust its consumption to its variable load, therefore enabling an optimum resource utilization. For instance, in most Cloud Computing systems, the number of computing instances can be resized in few minutes.

One last significant difference between cloud and grid is hardware administration and maintenance. Cloud consumers do not care about how things are running at the system end. They merely express their requests for resources and the Cloud Computing service then maps these requests to real physical resources. It therefore disburdens customers of a part of system administration. In the same way electricity became available to consumers in the 1930's when a national electricity grid was implemented, Cloud Computing is an attempt to commoditize IT. The administration part left to the cloud consumer depends on the type of cloud offer which will be chosen (Infractruture as a Service, Platform as a Service, etc.) as detailed in the following subsection.

From a computing consumer's point of view, Cloud Computing is therefore a Grid Computing with a single administrative domain, where the management of physical resources has a clear outsourced centralized ownership and hardware is accessed indirectly through abstractions, services and virtual machines. There are much fewer commitments for the consumer since no initial investments are

required anymore [1].

2.3.2 Everything-as-a-Service (XAAS)

A Cloud Computing environment is a three-actor-world. It is composed of the cloud provider which supplies hardware ownership and administration, the cloud consumer using this abstracted hardware to run applications (the cloud consumer often being a software company), and the application consumer, which is using the applications run on the cloud. The three actors are related to each other as follows: the cloud consumer is the customer of the cloud provider, and the application consumer is the customer of the cloud consumer.

It is up to the cloud consumer to choose which part of the application building he wants to handle by himself and which part will be built using the cloud provider pre-made libraries and services. This decision mostly depends on the cloud consumer's will to get involved in low-level software considerations. More specifically, there is a tradeoff for the cloud consumer between a higher control on the hardware and the systems he consumes, and a higher level of programming abstractions where developing applications is made easier.

To address the different choices of the cloud consumers, each cloud system has developed distinct strategies and API, as reported in [43]. A mainstream classification of Cloud Computing offers is therefore based on the programming abstraction level proposed by the cloud providers. Let us provide a short description of these different abstraction levels.

– *Infrastructure as a Service* (IaaS) is the lowest cloud abstraction level. Resources (computation, storage, and network) are exposed as a capability. Cloud consumers are relieved of the burden caused by owning, managing or controlling the underlying hardware. Therefore they do not access directly the physical machines, but indirectly through VM. IaaS consumers are given almost full control on the VM they rent: they can choose a pre-configured Operating System (OS) image or a custom machine image containing their own applications, libraries, and configuration settings. IaaS consumers can also choose the different Internet ports through which the VM can be accessed, etc. A VM in IaaS can run any application built by the cloud consumer or anyone else. The main example of IaaS offers is Amazon Elastic Cloud Compute (EC2).

1. With the notable exception of the cost of porting the existing applications on the cloud.

– *Platform as a Service* (PaaS) is a higher cloud abstraction level than IaaS, designed to ease applications building and hosting. Contrary to IaaS on which the OS and the configuration settings can be chosen and any application can be run, PaaS provides the cloud consumer with much less freedom. This consumer is given a fixed and constrained execution environment run on a specialized OS, and he is only in charge of defining the code to be run within this framework. The PaaS environment manages code deployment, hosting and execution by itself. PaaS offers may also provide additional development facilities, such as application versioning and instrumentation, application testing, system monitoring, etc. In the case of PaaS, storage is mostly provided in the form of a remote abstracted shared storage service. Prime examples of PaaS are Amazon MapReduce with Simple Storage Service (S3), Microsoft Azure, BungeeLabs, etc.

– *Software as a Service* (SaaS) is the highest cloud abstraction level where applications are exposed as a service running on a cloud infrastructure. Contrary to the customer of IaaS or PaaS, the SaaS consumer is not involved in any software development. This customer is only an application consumer communicating through the Internet with the application which is run on the cloud. Some examples are GMail, Salesforce.com, Zoho, etc.

These distinct abstraction level technologies have different positioning on some tradeoffs:

– *abstraction/control tradeoff*: PaaS provides higher abstractions to develop applications than IaaS, but the applications built on top of PaaS need to conform and fit with the framework provided. As outlined in Subsection 2.5.1, some processes hardly fit with a rigid framework like MapReduce. On the other hand, IaaS will provide cloud consumers with a higher control on the machines but with fewer off-the-shelf abstractions in return: the same distributed computation challenges will probably need to be repeatedly solved.

– *scalability/development cost tradeoff*: Applications run on a PaaS environment prove to be easily scalable. Yet, this scalability comes as a side-product of a strong design of the application so that it fits in with the framework provided. On the contrary, applications run on an IaaS environment provides more flexibility in the way the application can be built, but the scalability remains the customer's responsibility.

2.3.3 Technology stacks

Cloud Computing frameworks are different hardware and software technologies merged together into technology stacks to handle thousands of machines. This subsection highlights some of these hardware and software technology stacks developed by first row cloud actors.

Hardware technologies

In this subsection we describe how some of the cloud challenges are taken up, namely the minimization of the operating cost per machine and the improvement of bandwidth throughput between machines. The cost issue is managed by lowering energy consumption and improving automation (we refer the reader to [72] for an illuminating introduction to automation challenges in the cloud). The bandwidth throughputs are optimized with specialized hardware improvement and network topology refinements.

According to Jon Koomey (Consulting Professor at Stanford) on his blog, *"Cloud Computing is (with few exceptions) significantly more energy efficient than using in-house data centers"*. Yet, more than 40% of the amortized cost of a data center is still energy consumption, according to [59]. This energy consumption is optimized through special care of the cooling system and the minimization of power distribution loss (which respectively account for 33% and 8% of the total energy consumption of data centers, according to [59]). While early days data centers were built without paying much attention to those constraints, the most recently built data centers are located in places where power and cooling are cheap (for example on a river in Oregon for a Google data center, or in the Swedish town of Lulea for a Facebook data center).

Since automation is a mandatory requirement of scale, a lot of work has been done to improve it in data centers. According to [59], a typical ratio of IT staff members to servers is 1:100 in an efficient firm and 1:1000 in an efficient data center. This is partially due to the technologies developed by hosting companies (like Rackspace or OVH) or backup companies (like Mozy, funded in 2005). The virtualization techniques of machines —e.g. VM-ware or Microsoft Hyper-V— increase administrators' productivity by letting them handle much more machines. In addition, the number of CPU and the number of cores per CPU has been recently increased. This led to a situation in which the administrators are handling much more machines and in which each machine holds more cores. Machines are replaced when the physical failing machines in a rack or in a container hit a

given ratio. Other techniques of automation can be found in [72] or [79].

A key factor in cloud performance bandwidth (whether intra or inter data centers bandwidth) is the spatial disposition of the hardware in the data center, as well as the network topology between the different nodes dedicated to data storage and computing. While a lot of research has been made on this subject, major cloud companies have not been disclosing much about this major technological challenge and their own implementation. According to [59], Microsoft plans to upgrade some of its data centers following the networking architecture studied in [18].

Software technologies

Several software technology stacks have been developed to make the access to CPU and storage easier. They split into three abstraction level : the storage level, the execution level, and the DSL (domain-specific language) level.

– The storage level is responsible for preserving the data into the physical storage. It handles all the issues of distributed storage management, of data contention and provides the data with a structured access. It can be SQL or No-SQL storage, as described in Section 2.4. This storage can also be used as a means of communication between machines. Other types of storage are available in IaaS solutions such as Amazon Elastic Block Store that provides storage volumes that can be attached as a device to a running Amazon EC2 instance and that persist independently from the life of this instance.

– The execution level defines the general execution framework. It specifies how the machines are supposed to organize themselves, it monitors the workers and restarts the failing machines. Depending on the type of Cloud Computing offers (IaaS, PaaS, etc.), the execution level can provide each computing device with anything a standard OS could achieve, or it can give a much more constrainted execution framework where the execution flow must be designed in a specific way (the most famous example of such a constrained execution framework is MapReduce, described in Subsection 2.5.1).

– The DSL level is a higher level execution framework than the execution level. It often provides a declarative language instead of the procedural approach provided by the execution level, therefore saving the users from specifying how the work is supposed to be parallelized. It provides automated and under the hood parallelization, scheduling and inter-machines communications. Therefore, the DSL level helps users focus only on defining the result they want,

and not on how they want it to be computed. DSL implementations are often built in a SQL-like manner, allowing the people who are familiar with this syntax to run those requests in a similar way on an underlying No-SQL storage. Besides, since it provides a run-time framework to execute those requests, it spares the users from compiling their software each time they modify or add a request. Historically, they have been added years after the introduction of the execution level to provide non developing specialists with a means to easily use the execution level. PigLatin, Sawzall, DryadLinq are well-known examples of DSL implementations.

The storage level can be used independently from the two other levels as an outsourced storage service. The execution level can also be used independently from the storage level and the DSL level, provided no persistence is required. However, most cloud applications are built using both storage and execution level environments. The DSL level is an optional layer provided to ease the usage of the storage and execution levels. Theoretically, each project at a given level could be substitutable to any other project at the same level so we could for example plug and use any storage framework with any execution framework. Yet, in practice, some components are designed to work only with other components (as it is the case for the original MapReduce framework which requires some knowledge about where the data are stored for example). Therefore, there are four main frameworks that provide a full technology stack holding the three abstraction levels.

The first framework has been presented by Google. It is the oldest one. On the storage level, it is composed of Google File System ([56]) and BigTable ([39]) (the latter being built on top of Google File System). The execution level is represented by the original MapReduce version (described in Subsection 2.5.1): Google MapReduce ([47]). After the release of MapReduce, Google engineered a dedicated language, on the DSL level, built on top of MapReduce that provides the storage and the computation framework a higher level access: Sawzall ([97]).

The second technology stack is the Apache Hadoop project ([112]). This is one of the most famous open-source frameworks for distributed computing. It is composed of several widespread subprojects. The storage level includes HDFS, Cassandra, etc. The execution level includes Hadoop MapReduce, but can also be hand-tailored with the help of distributed coordination tools such as Hadoop Zookeeper. The DSL level includes Hive and Pig. Pig is a compiler that processes Pig Latin ([93]) code to produce sequences of MapReduce programs that can be run for example on Hadoop MapReduce. Pig Latin is a data processing language that is halfway between the high-level declarative style of SQL and the lower

procedural style of MapReduce. Pig Latin is used for example at Yahoo! (see for example [93]) to reduce development and execution time of data analysis requests.

The third technology stack is composed of Amazon Web Services (AWS) and AWS-compatible open-source components. Its most popular storage components are Amazon Simple Storage Service (Amazon S3), Amazon Simple Queue Service (Amazon SQS), Amazon SimpleDB, Amazon ElastiCache or Amazon Relational Database Service (Amazon RDS). AWS execution level is mainly composed of Amazon Elastic Cloud Compute (Amazon EC2) and Amazon Elastic MapReduce (which uses a hosted Hadoop framework running on top of Amazon EC2 and Amazon S3). AWS is one of the first large commercial Cloud Computing offers: it was initially launched in July 2002, and was valued at as a 1 billion dollar business in 2011 (see [1]). In parallel, Eucalyptus provides an open-source stack that exports a user-facing interface that is compatible with the Amazon EC2 and S3 services.

The fourth framework has been developed by Microsoft. The storage level is embodied by distinct independent projects: Azure Storage, Cosmos or SQL-Server. Cosmos is a Microsoft internal project that has been used by Search teams and BING. Azure Storage is a different project, initially built using a part of the Cosmos system (according to [36]). Azure Storage is meant to be the storage level standard layer for applications run on Azure. The execution level being held by Dryad ([73]) or by Azure Compute (or small frameworks like Lokad-Cloud ([11]) (on top of Azure Compute or Lokad-CQRS ([12])). The DSL level is represented through two different projects: DryadLINQ ([116]) and Scope ([38]).

Figure 2.1 (on the next page) is taken from a blog post of Mihai Budiu. It summarizes the technology stacks of Google, Hadoop and Microsoft.

2.4 Cloud Storage level

Persistence is a characteristic that ensures data outlive the process that has created them. A data storage is a system that provides data persistence and is a fundamental element in most application developments. In the case of distributed computing, data storage can also be used for a different purpose as an inter-machines means of communication.

While working on a cloud platform, no hardware shared memory is available since the machines are physically distant. In such a situation, inter-machines communications can be achieved through direct IP communications or through a

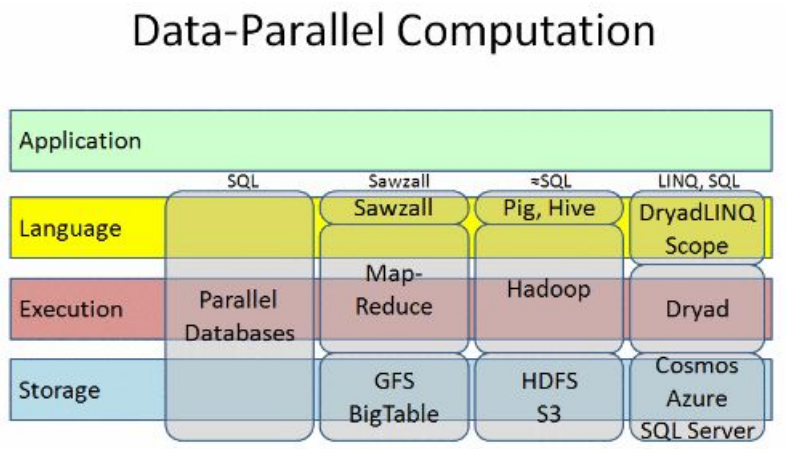

Figure 2.1: Illustration of the Google, Hadoop and Microsoft technology stacks for cloud applications building. For each stack, the DSL Level, the Execution Level and the Storage Level are detailed. In this scheme, SQL is also described as a stack in which the three levels are merged together and cannot be used independently.

distributed data storage system. While the direct IP communications have been much favored in several systems such as most Peer-to-Peer systems and are still largely used in Amazon EC2, the use of a data storage system as a means of communication is widely used in numerous Cloud Computing platforms. Since the cloud storage is abstracted and accessed “as a service”, it is easier to communicate through a data storage than to physically access a virtualized machine. The cloud storage has also been designed to easily scale and address satisfactory communication throughput.

In this section we describe the challenges involved by a large-scale distributed data storage system. We introduce in Subsection 2.4.1 the Relational DataBase Management System (RDBMS), which was the undisputed data storage system several years ago. Then we explain (in Subsection 2.4.2) why some of the guarantees of RDBMS impose constraints which are hard to meet in a distributed world. Finally, we introduce the reader in Subsection 2.4.3 to some new data storage systems, known as No-SQL storages, which relax some of the guarantees of the relational database systems to achieve better scaling performance.

2.4.1 Relational storage and ACID properties

Relational databases are structured through the relations between data entities. They are inspired by relational algebra and tuple calculus theories, while they do not respect all of their constraints. Relational databases are well understood and offer great guarantees for reliably storing and retrieving data in a robust manner. In addition to these guarantees, the relational model proved to be well-suited to express real-world business data interdependencies. RDBMS are therefore widely used in business applications.

A Structured Query Language (SQL) is provided together with the RDBMS to make the use of data easier. SQL is a declarative programming language (with some procedural elements) intended to manage data queries. Declarative languages express *what* computations should return rather than explicitly defining *how* the computations should be done. In the case of relational storages, queries are expressed in SQL to define requests to the desired data, while the underlying RDBMS performs the actual operations to answer the requests. Listing 2.1 gives an example of a SQL query.

Listing 2.1: a SQL query sample

```
SELECT T1.variable1 , T1.variable2
FROM Table1 T1, Table2 T2
WHERE T1.variable1 = T2.variable3
ORDER BY T1.name
```

A single "business" operation applied on data stored in a database is called a transaction. A single transaction can affect several pieces of data at once. The guarantees provided by RDBMS can be understood by observing how transactions are applied on the database. The following paragraphs introduce the main relational storage guarantees by describing how they are translated in terms of transactions management.

– *Consistency* is a property related to the way data are modified during transactions. There are multiple definitions of consistency, resulting in different consistency levels and depending on whether the storage is distributed or not. Initially, consistency was a property that guaranteed that some user-specified invariants on data were never broken. For example, such an invariant could consist in having the sum of some variables equal the sum of other variables. With the rise of distributed storages, the definitions of consistency have been multiplied, and consistent distributed storage (also called strongly consistent storages) often refers to the fact that each update is applied to all the relevant

nodes at the same logical time. A consequence of this definition is that if a piece of data is replicated on different nodes, the data piece will always be the same on each node, for each given instant. A more in-depth definition of consistency can be found in [57].

– *Atomicity* guarantees each transaction is applied as if it were instantaneous. More specifically, a transaction composed of n database elementary operations is atomic if all the operations occur or none occur (if one or several operations fail) and if all the processing units either see the database in the state it was before the beginning of the transaction or after the end of the transaction. A storage has the atomicity property if any transaction can be guaranteed to be atomic. This property guarantees that any transaction applied on a database does not corrupt it.

– *Isolation* guarantees that each transaction should happen independently from other transactions that occur at the same time.

– *Durability* enforces that data are persisted permanently and that transactions remain permanent, even in the presence of a system failure.

These four important guarantees are gathered under the acronym ACID, which stands for Atomicity, Consistency, Isolation and Durability. These properties are guaranteed by every relational storage. In the case of a distributed relational storage, some other very mandatory properties can be desired, such as Partition tolerance and Availability.

– A network is *partitioned* when all the messages sent from nodes in one component of the partition to nodes in another component are lost. Distributed storage is said to be partition tolerant when it continues to run in the presence of a partition failure.

– A distributed storage guarantees *Availability* if every request received by a non-failing node in the system returns a response.

2.4.2 CAP Theorem and the No-SQL positioning

The CAP theorem was first expressed by Eric Brewer at the 2000 Symposium on Principles of Distributed Computing (PODC). The CAP theorem was then proved in 2002 by Seth Gilbert and Nancy Lynch in [57]. This theorem is a theoretical impossibility result, which states that a distributed storage cannot achieve both

Availability, Consistency, and Partition Tolerance.

More specifically, this theorem is a theoretical translation of the well-known practical challenges of applying ACID transactions on data split on multiple machines. The ACID constraints require a lot of inter-machines communications to complete a single transaction (see for example [100]) and lead to serious lock contentions that deeply decrease the transactional throughput.

Moreover, guaranteeing that the data stored in cheap and unreliable hardware are highly available implies multiple replications of each single piece of data. Since data servers are expected to fail from time to time, the data are duplicated several times (Azure storage is reported to hold for each data 6 replicas: 3 replicas in a first data center and 3 others in a backup data center ([36])), so that a data server can die without any data loss. To guarantee strong consistency while keeping multiple versions of any data pieces in different locations causes large overheads, as reported in [19].

Scaling traditional ACID-compliant relational databases to distributed shared-nothing architectures with cheap machines therefore turned out to be very difficult. Data storage systems developed in the 2000s known as NoSQL storage (for Not Only SQL) such as Cassandra, HBase, BigTable, SimpleDB, etc. therefore dropped some of the ACID guarantees to achieve better scaling and lower latency. As stated by Daniel Abadi and Alexander Thomson in their blog, *"NoSQL really means NoACID"* (see [15]).

Some of these systems (like BigTable or Azure BlobStorage) decided to provide weakened atomicity and isolation guarantees for transactions. Indeed, transactions that modify more than one piece of data are not guaranteed to be either all done or none. Therefore, no logical transaction implying several pieces of data can be guaranteed to be atomic at the storage level. It is up to the storage user to ensure logical transactionality, and that user-specified invariants are not modified.

Other cloud storage systems like Cassandra have been dropping strong consistency between replicas to provide only a weakened consistency guarantee called eventual consistency. We refer the reader to [108] for an introduction to eventual consistency, while an introduction to BASE (Basically Available, Soft state, Eventually consistent) can be found in [99]. While the eventual consistency is satisfactory enough for a given number of well-known web applications such as Facebook, the loss of strong consistency is a very difficult issue for many other real world applications, which makes application development much harder. This is a key aspect of No-SQL storage: the responsibilities are partly shifted and the

scalability comes at the expense of a stronger developer burden. The consistency of various cloud storage systems has been investigated in [109].

2.4.3 Cloud Storage Taxonomy

Along with the calling into question of ACID properties, the relational data model which is related to them has also been challenged. No-SQL storages therefore provide different data models, often noted to provide a more fuzzy division between the database model and the application. In this paragraph we describe the most used database models. A more comprehensive list of these models can be found in [9].

Document-oriented databases assume data are organized around the concept of documents. Each document holds data and is persisted through a standard data format such as XML, YAML, JSON, etc. or even through binary or .pdf files. Contrary to RDBMS where the data scheme is fixed, document-oriented databases do not impose that each document should own the same fields as the others. Thus, new information can be added on a document without having to fill in the same information for all the other available documents. Each document can be accessed through a unique key, which is often a string. CouchDB, MongoDB, SimpleDB are well-known document-oriented databases.

Graph databases are structured on the standard graph theory concepts of nodes and edges. The information is stored on nodes, each node representing a different entity. Each entity is entitled to a set of properties describing the entity. Edges connect nodes to nodes or nodes to properties and describe the relationships between the nodes and the properties. Because of this graph structure, data and more specifically nodes are given an adjacency notion which is index-free and only driven by the nature of the data. Horton, FlockDB, AllegroGraph are graph databases.

Key-value pairs databases are schema-free databases. The data is stored on a material device in the form of a serialized programming language datatype, typically a class object. Any type can be persisted and data are accessed through a key, which can be a string or a hash of the data content. Apache Cassandra, Dynamo, Velocity, BigTable, Azure BlobStorage are key-value pairs storages.

2.5 Cloud Execution Level

We examine in this section some of the most widespread execution frameworks. While these frameworks are not only cloud-based, they are often an essential part of scientific cloud applications. The section begins with the most well-known framework: MapReduce.

2.5.1 MapReduce

General context

MapReduce has two different meanings. It first describes a general framework concept explained in this section. It also defines the original implementation of this abstract framework, developed by Google ([47]). Other implementations of the concept have then been set-up, like Hadoop ([112]), or CloudMapReduce ([82]). Unless explicitly stated otherwise, the term MapReduce refers in this section to the framework abstraction concept.

MapReduce is a framework that eases distributed computation runs and deployments. It was born from the realization that *"The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues"*, as stated by the original MapReduce paper ([47]). MapReduce is a scale-agnostic programming abstraction (see [64]) originally designed to prevent programmers from solving these issues repeatedly and to provide access to a huge amount of computation resources to programmers without any experience with parallel and distributed systems. The resort to distributed computations and to these computation resources lead to significant speedup, i.e. the time to run distributed computations is significantly reduced compared to the sequential computation.

MapReduce is a gathering of ideas which have already been discussed in the computer-science literature for a long time (even for decades for some ideas) and it is not the first framework to have implemented some of these ideas. MapReduce is still a well-known abstraction concept, thanks to the large success of its first implementation: Google MapReduce. This framework implementation has proved to be very easy to use for "programmers without any experience with parallel and distributed systems" —referred to in the following as non-experts— and achieved very good overall scalability as well as a strong resilience to failures.

In the following paragraphs, we provide a very short overview of this framework.

We refer the reader to the Hadoop definitive guide ([112]) for in-depth details of the most famous open-source implementation and to [81] and [40] for examples of machine-learning problems that can be addressed through a MapReduce framework.

Programming Model

MapReduce provides an execution environment to process a two-stage execution (divided into a Map stage and a Reduce stage). A typical job in MapReduce is composed of many Map tasks and Reduce tasks. In the Map stage, a user-specified computation —referred to as Map— is applied independently and in parallel on many data chunks. Each Map task points to a specific data chunk to which the Map computation will be applied on. Once a Map task is completed, intermediate results are produced and stored on hard-drives. Once all the Map tasks have been successfully completed, an aggregation operation is started, referred to as Reduce. Depending on the size and complexity of all the intermediate results produced by the Map tasks, the reduce stage is divided into one or many reduce tasks.

The Map and the Reduce logic are defined by the user. This user submits the MapReduce job through the MapReduce interface, specifying the actual Map and Reduce computations, in addition to the specification of the data that must be processed. The framework provides an environment that handles all the execution orchestration and monitoring. In particular, MapReduce automatically maps the job into a set of sub-tasks, sends the Map and Reduce instructions to the right processing units, makes sure all the computations are successfully run, and restarts or reschedules tasks that have failed.

Technical considerations

The primary targets of MapReduce are heavy computations or computations on large data sets. The leading idea of MapReduce is that it is not an experimental framework but a production tool. As a direct consequence, MapReduce provides a simple and robust abstraction that can easily be handled by a wide range of non-experts.

The simplicity of MapReduce comes from two different design choices. Firstly, by making a clear distinction between *what* needs to be computed (user-defined) and *how* the computations are actually made (handled automatically by the framework), MapReduce disburdens the users from difficult parallel computations

considerations. More specifically, the users need not dive into general parallel applications design, neither do they need to have any knowledge about the hardware actually running the computations: MapReduce indeed provides a scale-agnostic ([64]) interface.

Secondly, the framework helps the users to figure out what the parallel algorithm is actually running. As stated in [81], *"concurrent programs are notoriously difficult to reason about"*. Because of its elementary design, a MapReduce execution has Leslie Lamport's sequential consistency property [2], which means the MapReduce result is the same as the one that would have been produced by some reordered sequential execution of the program. This property helps to understand the MapReduce execution as it can therefore be rethought as a sequential run speeded-up.

As previously stated, the MapReduce framework also provides a strong resilience to failures. A reliable commodity hardware usually has a Mean-Time Between Failures (MTBF) of 3 years. In a typical 10,000 server cluster, this implies that ten servers are failing every day. Thus, failures in large-scale data centers are a frequent event and a framework like MapReduce is designed to be resilient to single point of failures [3]. Such a resilience is hard to achieve since the computing units are dying silently, i.e. without notifying of their failure. MapReduce holds a complex monitoring system to ping all the processing units and to ensure all the tasks will be successfully completed, as detailed in [47].

The initial problems MapReduce has been devised to deal with are related to text mining. These problems involve gigantic amounts of data that do not fit in the memory of the processing units dedicated to the computation, as explained in [48]. In such situations, providing the processing units with a continuous flow of data without starving these units out would require an aggregated bandwidth very hard to achieve. MapReduce has provided a new solution to this problem by co-locating the data storage and the processing system. Instead of pushing the data chunks to the processing units in charge of them, MapReduce pushes the tasks to the processing units whose storage holds the data chunks to be processed. Such a design requires a distributed file system that can locate where the data are stored, but removes a lot of stress on communication devices.

2. Provided that the map and reduce operators are deterministic functions of their input values.
3. With the noticeable exception of the single orchestration machine referred to as master.

MapReduce performance

Since its first release, MapReduce has proved to be very well-suited to multiple large computation problems on machine clusters. An embarrassingly parallel problem is one for which little effort is required to separate the problem into a number of parallel (and often independent) tasks. These problems require few or zero inter-machines communications and therefore allow to achieve speedup close to the optimal. On such problems, MapReduce is performing very well.

A MapReduce mechanism that can significantly impact MapReduce performance is the synchronization process: the Reduce stage cannot begin before all the Map tasks have been completed. This design is very sensitive to stragglers. As defined in [47], a straggler is a *"machine that takes an unusually long time to complete one of the last few map or reduce tasks"*. Because of the synchronization process in the end of the Map stage, the overall computation is significantly slowed down by the slowest machine in the pool. In the case of a very large execution involving thousands of processing units, the worst straggler will behave much worse than the median behaviour, and significant delays can be noted. MapReduce provides a backup execution logic that duplicates the remaining in-progress tasks when almost all the map tasks have already been completed. Such a duplication mechanism is reported to significantly reduce the time to complete large MapReduce operations in [47].

MapReduce has proved to be very efficient in the case of embarrassingly parallel algorithms. For example, the Hadoop implementation has been successfully used by Yahoo! to win the terasort contest (see [92]). The Google implementation is also reported to be widely used by Google's internal teams (see e.g. [48]).

Recent criticisms of the MapReduce framework for machine-learning

In addition to the quality of the implementations of Google's and Hapache's MapReduce, a key factor in MapReduce success and adoption by the community is how easy it is to use. Because MapReduce only supports tasks that can be expressed in a Map/Reduce logic, it is rather straightforward to parallelize a task using this framework: either the task is intrinsically in a Map/Reduce form and the expression of the task within the framework is obvious, or the task cannot be expressed within MapReduce.

The simplicity to use it and to figure out how to adapt a given sequential algorithm into the framework has driven the interest of statistical and machine-learning

researchers and practitioners that had no knowledge in distributed computations. Such a success has paved the way for numerous research works to modify and adapt algorithms originally not suited to the MapReduce framework and to suggest MapReduce implementations of the modified algorithms. Yet, since the MapReduce is a rigid framework, numerous cases have proved to be difficult or inefficient to adapt to the framework:

– The MapReduce framework is in itself related to functional programming. The Map procedure is applied to each data chunk independently. Therefore, the MapReduce framework is not suited to algorithms where the application of the Map procedure to some data chunks need the results of the same procedure to other data chunks as a prerequisite. In other words, the MapReduce framework is not suited when the computations between the different pieces of data are not independent and impose a specific chronology.

– MapReduce is designed to provide a single execution of the map and of the reduce steps and does not directly provide iterative calls. It is therefore not directly suited for the numerous machine-learning problems implying iterative processing (Expectation-Maximisation (EM), Belief Propagation, etc.). The implementation of these algorithms in a MapReduce framework means the user has to engineer a solution that organizes results retrieval and scheduling of the multiple iterations so that each map iteration is launched after the reduce phase of the previous iteration is completed and so each map iteration is fed with results provided by the reduce phase of the previous iteration.

– Both the thinking of the abstract framework and of google's practical implementation have been originally designed to address production needs and robustness. As a result, the primary concern of the framework is to handle failures and to guarantee the computation results. The MapReduce efficiency is therefore partly lowered by these reliability constraints. For example, the serialization on hard-disks of computation results turns out to be rather costly in some cases.

– MapReduce is not suited to asynchronous algorithms.

MapReduce is the reference framework for distributed computations. Its parallelization scheme is the natural choice because of its simplicity. But because of its limitations, some have started to question the relevance of MapReduce in the case of fine-granularity tasks in machine-learning. As outlined in the alternative GraphLab framework introduction paper ([85]) : *"The overhead associated with the fault-tolerant, disk-centric approach is unnecessarily costly when applied to*

the typical cluster and multi-core settings encountered in ML research". This position was also shared by numerous researchers (including John Langford) during the NIPS workshop LCCC debate in December 2010.

The questioning of the MapReduce framework has led to richer distributed frameworks where more control and freedom are left to the framework user, at the price of more complexity for this user. Among these frameworks, GraphLab and Dryad are introduced in the following paragraphs.

2.5.2 GraphLab

GraphLab is a new framework directly designed to address parallelization challenges of machine-learning algorithms. GraphLab was initially developed after it has been observed that MapReduce was insufficiently expressive for some machine-learning algorithms, while low-level APIs (like MPI) usage led to too much development overhead.

The GraphLab representation model is much more complex and richer than the MapReduce one. It consists in a directed graph representing the data and computational dependencies and in a set of update functions specifying the local computations to be run. By adopting a graph model, GraphLab supports structured data dependencies and is therefore suited to computation schemes where update scopes overlap and some computations depends on other computation results. More specifically, the original GraphLab introduction paper states that it is targeted to *"compactly express asynchronous iterative algorithms with sparse computational dependencies"*.

Like MapReduce, GraphLab provides a higher abstraction level than MPI and prevents users from managing synchronization, data races and deadlocks challenges by themselves. It also provides ad-hoc tasks schedulers.

GraphLab is therefore a machine-learning parallelization framework which provides more expressiveness than MapReduce, at the expense of a more complex usage. To our knowledge, the GraphLab framework is only currently available on shared-memory multiprocessors setting (only tested on single machines up to 16 cores), making GraphLab unavailable or ineffective for distributed cloud computations.

2.5.3 Dryad and DryadLINQ

Dryad ([73]) is a general-purpose distributed computation framework designed to scale on any single administrative domain, be it a multi-core single computer or a whole data center with thousands of computing instances. Conversely, Dryad is not targeted to multiple administrative domain environments and does not handle all the authentication and access grants required for most Grid Computing environments.

Like GraphLab, Dryad exposes a much more expressive computing framework than MapReduce in the form of a Direct Acyclic Graph (DAG). In this graph, each vertex is a computation task and edges are data and intermediate results channels. Dryad infrastructure maps the logical DAG of computation tasks onto actual physical resources without requiring the developer to get involved in this mapping process. In addition, Dryad handles scheduling across resources, failure recoveries, communication concurrencies, etc.

Whereas with MapReduce the developer is given few implementation choices except on how to fit its computation task into the rigid framework of MapReduce, Dryad provides the developer with much more control through the expressiveness of the DAG. This control gain can be explained by a different approach of the frameworks: while MapReduce was focused on accessibility to provide the widest class of developers with computation power, Dryad is more concerned about expressiveness and performance.

In addition to a richer expressiveness of the execution flow, Dryad provides a higher control than MapReduce on communication flow and tasks scheduling. This optional higher control endows the programmers with a better understandings of the underlying hardware performance and topology or of the computation tasks to improve the overall performance. For example, Dryad provides different communication media between computing nodes: direct communication through TCP pipes, or indirect communication through files written to hard-drive or shared-memory buffers. Whereas the default Dryad settings is to use files, the developer can choose to resort to a different medium, e.g. in the case of a single multi-core computer.

As a side result of this higher control and expressiveness, Dryad is reported to be more demanding on the developer's knowledge and requires several weeks to be mastered. To make Dryad more user-friendly, Microsoft has also been developing an additional framework built on top of Dryad and mimicking the LINQ library interface: this additional framework is called DryadLINQ ([116]). Both Dryad

and DryadLINQ have been made available on the Microsoft research website.

Chapter 3

Presentation of Azure

3.1 Introduction

With the rise of advanced Internet technologies, an alternative mode of software consumption has been proposed. This alternative mode, referred to as Software as a Service (SaaS), provides the use of applications that are not hosted on the users' hardware but on a distant server. The user is accessing this distant server through the Internet, and most of the resources required by the application are provided on the server side. This server/client approach is a kind of renewal of the old server/terminals design. This SaaS approach has a lot of advantages and drawbacks. To name but a few, SaaS lowers maintenance labor costs through multi-tenancy, it requires a smaller customer commitment since the customer is renting the usage of a software instead of buying it, it partially outsources IT for companies for whom it is not the core business, etc. The SaaS approach has also its detractors. For example, Richard Stallman has incarnated the voice of those for whom SaaS raises many concerns, e.g. data privacy.

No matter how it turns out, SaaS market will probably experience significant growth in the following years and Cloud Computing will be a competitive hosting platform for these software applications. This partial software paradigm shift has a lot of impact on software development companies. Among them, Microsoft is an illuminating example. According to its annual report for the fiscal year ended June 30, 2011, more than 83% of Microsoft revenues are split among three of its divisions: Windows & Windows Live Division ($ 18,778 millions), Server and Tools Division ($ 17,107 millions), and Microsoft Business Division ($ 21,986 millions). These 3 divisions all correspond to standard desktop software or operating system: Windows & Windows Live Division is mainly composed of Microsoft Windows, the Server and Tools Division mainly deals with Win-

dows Server and SQL-Server, and the Business Division is related to desktop software such as Office, SharePoint, Exchange, Lync, Dynamics, etc. While one can therefore see that Microsoft revenues were and still are mainly driven by the licence selling of their desktop software applications and of their operating system Windows, an important migration has been initiated toward Cloud Computing solutions to provide these applications as SaaS. This situation has turned Microsoft into both a cloud provider and a cloud consumer. This introduction gives a brief overview of the different Microsoft Cloud Computing solution layers.

A first key aspect of the Microsoft Cloud Computing solutions is this effective shift operated by Microsoft from desktop software to Software as a Service (SaaS). An increasing part of the software applications presently sold by Microsoft has indeed been migrated to the cloud and is offered alternatively as a SaaS solution. Microsoft Office 365 for example, is a commercial software service offering a set of Microsoft products. It has been publicly available since 28 June 2011 and includes the Microsoft Office suite, Exchange Server, SharePoint Server, etc. Another important Microsoft cloud SaaS solution is the search engine Bing, which is run (at least partially) on their Cloud Computing platform (see [36]). The cloud SaaS version of these applications are targeted to any customer of the desktop version of the same applications and the use of these cloud applications does not require any software development knowledge. The consumer just uses these applications as if everything was processed locally whereas the computations are actually run on the cloud.

A second aspect of the Microsoft Cloud Computing system is Microsoft cloud High Performance Computing (HPC) solution. This solution is mainly based on Dryad and DryadLINQ (see [73] and [116]). A new Azure component available since november 2011, called Windows Azure HPC Scheduler includes modules and features that enable to launch and manage high-performance computing (HPC) applications and other parallel workloads within a Windows Azure service. The scheduler supports parallel computational tasks such as parametric sweeps, Message Passing Interface (MPI) processes, and service-oriented architecture (SOA) requests across the computing resources in Windows Azure. The Azure HPC solution is targeted to be the corresponding cloud version of the Microsoft HPC solution on Windows Server.

The purpose of this chapter is to describe the Microsoft cloud components that are the elementary building blocks upon which cloud applications are build. These elementary building blocks are used by Microsoft to run its SaaS applications like Office 365 or its HPC solutions (Azure HPC), but are also used by external companies to build other cloud applications. For example, Lokad has migrated

its forecasting engine on the cloud to benefit from the scalability and elasticity provided by Azure. The Azure elementary building blocks are gathered in the form of a Platform as a Service (PaaS) solution (see Subsection 2.3.2). This solution, referred to as Windows Azure Platform, is marketed for software developers to store data and to host applications on Microsoft's cloud. Azure lets developers deploy and run robust and scalable applications. Built upon geographically distributed data centers all over the world, Azure also provides the hosted web applications with low-latency responsiveness guarantees and a solid framework for CPU-intensive processing tasks. Contrary to the SaaS part of their Cloud Computing solution, only developers are targeted by Windows Azure.

The Windows Azure Platform is composed of a persistent storage (Azure Storage) and of a cloud operating system (Azure Compute) that provides a computing environment for applications. In addition to these two core blocks, many tools are available to help developers during the application building, deployment, monitoring and maintenance. This chapter provides a short overview of the Azure Platform. It is organized as follows. The first section below presents Azure Compute. We give there a definition of the main components of a Windows Azure application. Section 3.3 defines the different parts of Azure's persistent storage system and the architecture proposed to build a scalable storage system. Section 3.4 is dedicated to the raw performances of the storage and of the computing instances. We report for example on the bandwidth and Flops we have managed to obtain. These results have a crucial impact on algorithms since they determine their design and potential achievements. Section 3.5 introduces some key figures to estimate the total price of running algorithms on Azure.

3.2 Windows Azure Compute

Windows Azure Compute is exposed through hosted services which are deployable to an Azure data center. Each hosted service corresponds to a specific web application and is composed of roles, each of this role corresponding to a logical part of the service. The different roles that can be part of a hosted service are: the *web roles*, the *worker roles* and the *Virtual Machines (VM) roles*. A single hosted service is composed of at least one web role and one worker role. Each role is run on at least one virtual machine, referred to as a role instance or a worker, so a single hosted service requires at least two role instances. We now briefly describe the different roles:

– Web roles are designed for web application programming. Web Roles allow public computers to be connected to the hosted service over standard HTTP

and HTTPS ports. VM running on a given web role are pre-configured with IIS7 (Internet Information Services) and specifically designed to run Microsoft web-programming technologies as ASP.NET or Windows Communication Foundation (WCF), but they also support native codes such as PHP or Java to build web applications.

– Worker roles are designed to run general background processes. These processes can be dependent on a web role (handling the computation required by the web role) or independent. One of the differences between web and worker roles is that worker roles don't come with a pre-installed IIS. Worker roles execution code can be defined using the .NET framework.

– VM roles are designed to provide developers with a much wider scope of possibilities and especially to control the operating system image. VM roles should not be used unless worker and web roles do not fit the developer's purpose, as it is the case for example when one has long and complicated installations in the operating system or a setup procedure that cannot be automated. In VM roles, the developer will upload his own virtual hard drive (VHD) that holds a custom operating system (more specifically a Windows Server 2008 R2 image) that will be run on the different VM running the VM role. This role will not be used in our cloud algorithm implementations.

The role is the scalability unit of a given hosted service. For each role, the number of role instances that are run is a user-defined quantity that can be dynamically and elastically modified by the Azure account owner through the Monitoring API or the Azure account portal. Azure Compute manages the lifecycle of role instances. More specifically, Azure by itself ensures that all the role instances are alive and available. In the event of a failure, the failing role instance is automatically restarted on a different virtual machine.

As a consequence, developers are not expected to handle the individual virtual machines by themselves: they just implement the logic for each role and upload the code to Azure through a package from the Azure administration portal, or directly from Visual Studio. An Azure-side engine called Azure Fabric handles the package deployment on as many virtual machines as requested. Similarly, the role update requires the developer to only update the role and not each of role instances, which is automatically handled by Azure.

While the role instances have not been designed to be manually controlled, it is still possible to directly access them. More specifically, an internal endpoint is exposed by each role instance so that the other role instances that are related to

the same role could access it. These internal endpoints are not visible outside the hosted service.

3.3 Windows Azure Storage

The Windows Azure Storage (WAS) is the storage component of the Windows Azure Cloud Platform. It is a public cloud service available since November 2008 and which presently (in January 2012) holds 70PBytes of data. It is used as an independent storage service but also as the persistence storage for the applications run on Windows Azure Cloud Platform. According to [36], the WAS is used internally by Microsoft for applications such as social networking search, serving video music and game content but also outside Microsoft by thousands of customers.

The WAS has been designed to be highly scalable, so that a single piece of data can be simultaneously accessed by multiple computing instances and so that a single application can persist terabytes of data. For example, the ingestion engine of Bing used to gather and index all the Facebook and Twitter content is reported to store around 350 TBytes of data in Azure (see [36]).

The WAS provides various forms of permanent storage components with differing purposes and capabilities. The following subsection describes these components.

3.3.1 WAS components

The WAS is composed of four different elements: Windows Azure BlobStorage, Windows Azure TableStorage, Windows Azure QueueStorage (respectively referred to in the following as BlobStorage, TableStorage and QueueStorage), and Windows Azure SQL.

– The BlobStorage is a simple scalable Key-Value pairs storage system. It is designed to store serialized data items referred to as blobs.

– The TableStorage is also an alternative Key-Value pairs storage system. It provides atomicity guarantees in a constrained scope, as detailed below.

– The QueueStorage is a scalable queues system designed to handle very small objects. It is used as a delivery message mechanism across the computing

instances.

– Azure SQL is a relational cloud database built on SQL-Server technologies. It is designed to be an on-demand RDBMS and requires no setup, installation and management.

A common usage pattern for the different storage elements is as follows: I/O data transfer is held through the BlobStorage, overall workflow instructions and job messages are held by the QueueStorage and describe how the blobs need to be processed, and intermediate results or service states are kept in the TableStorage or the BlobStorage.

Azure BlobStorage

Azure BlobStorage is a large-scale key-value pairs storage system. It is designed to be the main storage component of most azure applications.

The objects stored in the BlobStorage are stored in Binary Large OBjects (referred to as blobs). The size of a blob is bounded to 50GBytes. The blobs are stored on multiple nodes, and a complex load-balancing system (partially described in Subsection 3.3.2) ensures the system is at the same time scalable and strongly consistent. The strong consistency is an important aspect of the BlobStorage since it has an impact on overall latency and scalability of the BlobStorage, as explained in 2.4.2. This property is a key characteristic of the WAS, as many other No-SQL storage (such as Cassandra) do not provide such a guarantee. The strong consistency implies that each blob is immediately accessible once it has been added or modified, and that any subsequent read from any machine will immediately see the changes made by the previous write operations.

Each object stored in the BlobStorage is accessed through its key. The key is a simple two-level hierarchy of strings referred to as containerName and blobName. ContainerName and blobName are scoped by the account, so two different accounts can have the same containerNames or blobNames. The BlobStorage API provides methods to retrieve and push blobs for any given key, and a method to list all the blobs whose key shares a given prefix.

A key feature of data storages is the way multiple data can be accessed and modified simultaneously. As detailed in Subsection 2.4.1, a single transaction may indeed involve multiple pieces of data at once. In the context of RDBMS, transactions are guaranteed to be atomic, so each piece of data concerned by

a transaction is updated accordingly or none is (if an update fails). The Azure BlobStorage does not give any primitive to atomically modify *multiple* blobs. Because of the internal design of the WAS, the different blobs are stored on multiple machines distributed on separate storage stamps (we refer the reader to more in depth details in Subsection 3.3.2). This physical distance between the storage nodes of different blobs involved in a transaction precludes low latency atomic multi-blob transactions. This is why there are no primitives on Azure side to run transactions over multiple blobs atomically.

Azure TableStorage

Azure TableStorage is the second key-value pair storage component of the WAS. It is designed to provide structured data used in Azure applications with a more adapted storage. The structured storage is provided in the form of tables. A single application may create one or multiple tables, following a data-driven design. Each table contains multiple entities. To each entity, two keys are associated in the form of strings: the first key is called the PartitionKey and can be shared with some other entities of the same table, while the second key is specific to each entity and is called the RowKey. Any given entity is therefore uniquely referenced as the combination of its corresponding PartitionKey and RawKey. Each entity holds a set of $< name, typed\ value >$ pairs named Properties. A single table can store multiple entities gathered into multiple different PartitionKeys.

In a given table, each entity is analogous to a "row" of a RDBMS and is a traditional representation of a basic data item. This item is composed of several values, filled in the entity's properties. Each table is schema-free insofar as two different entities of a given table can have very different properties. Yet, a common design pattern consists in adopting a fixed schema within a given table so all the entities have exactly the same set of properties.

Because of the absence of a mandatory fixed schema, even in the case of the previous design pattern where the schema is *de facto* fixed, the TableStorage does not provide any solution to represent the relationships between the entities' properties contrary to other structured storage such as RDBMS. Another consequence of this flexible schema is the lack of "secondary" indices in a given table: the only way to enumerate entities in a table relies on the entity key, namely the RowKey. Therefore, selecting entities according to a criterion based on a given property will lead to execution time linear in the number of entities in the table (contrary to RDBMS, where secondary indices may lead to requests time proportional to

the logarithm of the number of entities).

An additional feature of the TableStorage is to allow reads, creations or updates of multiple entities in a single command. These commands —referred to as entity group transactions— support executing up to 100 entities in a single batch, provided the entities are in the same table and share the same PartitionKey. In addition to the atomicity guarantees described in Subsection 3.3.1, the entity group transactions are of prime interest since they provide a way to significantly lower (up to a factor 100) the price of storage I/O in the case of small objects being stored.

Azure QueueStorage

Azure Queues provide a reliable asynchronous message delivery mechanism through distributed queues to connect the different components of a cloud application. Queues are designed to store a large amount of small messages (with maximal individual size of 8 KBytes, see for example [7]). Using queues to communicate helps to build loosely coupled components and mitigates the impact of any individual component failure.

Azure queues are not supposed to respect FIFO logic (First In First Out) as standard queues. At a small scale, the Azure queue will behave like a FIFO queue, but if it is more loaded, it will adopt a different logic so that it can better scale. Therefore, the FIFO assumption cannot be assumed when designing a cloud application that uses queues.

Messages stored in a queue are guaranteed to be returned at least once, but possibly several times: this requires one to design *idempotent* tasks. When a message is dequeued by a worker, this message is not deleted but it becomes invisible for the other workers. If a worker fails to complete the corresponding task (because it throws some exception or because the worker dies), the invisibility timer happens to time-out and the message becomes available again. If the worker processes the message entirely, it notifies the queue that the processing has been completed and that the message can be safely deleted. Through this process, one can make sure no task is lost because of a hardware failure for instance.

Synchronization Primitives

Item updates mechanism for BlobStorage and TableStorage is achieved through an optimistic nonblocking atomic read-modify-write (RMW) mechanism. A nonblocking update algorithm consists in updates executed speculatively, assuming

no concurrent machine is updating the data at the same time. The nonblocking update algorithms do not imply synchronization or locking mechanisms when an update is executed. However, a check is performed at the end of the update to make sure that no conflicts have occurred. In the case of a conflict, the update is aborted and needs to be restarted. Nonblocking update algorithms are often called optimistic because they bet on the fact conflicts are statistically unlikely to happen. If concurrent writes happen, then the update mechanism is likely to take more time than a simple locking update process. Yet on average, the optimistic update algorithms are much more efficient than locking algorithms.

The optimistic RMW mechanism is allowed in Azure through timestamp referred to as etag, following the Azure terminology. This timestamp indicates the exact date of the last successful write operation applied to the item inside the BlobStorage or the TableStorage. The Azure item update is performed as follows: the item to be updated is downloaded by the computer in charge of the item update in addition to the corresponding etag of the item. The actual value of the item is locally updated by the computer, then pushed back to Azure BlobStorage or TableStorage with the previous etag. If the returned etag matches the present etag of the item version stored in the WAS, then the item is actually updated. If the two etags do not match, then a distinct machine has been concurrently updating the same item, and the updating process is relaunched.

The RMW mechanism is a classical synchronization primitive (see for example [67]). Implementing the timestamp system and an efficient conditionnal write mechanism is a difficult task on architectures without hardware shared-memory like Azure since a single piece of data is replicated multiple times on different machines. More specifically, special attention on Azure side is paid to lower the overall write latency. A part of the adopted design is described in Subsection 3.3.2.

Partitions, Load-Balancing and multi-items transactionality

To load-balance the objects access, the WAS is using a partitioning system. This partitioning system is built upon a partition key in the form of a string. For the BlobStorage, the partition key of a blob is the concatenation of its containerName and of its blobName strings. In the case of the BlobStorage, each partition is therefore only holding a single blob, and it is impossible to gather several blobs into a single partition. For the TableStorage, the partition key is the PartitionKey string already introduced in Subsection 3.3.1. In this case, multiple entities can be gathered in a single partition provided they are in the same table and share the

same partition key.

The load-balancing system is separated into two load-balancing levels. The first level is achieved by Azure itself and provides an automatic inter-partitions load-balancing. Nodes dedicated to handle data requests are called partition servers. The inter-partitions load-balancing system maps partitions to partition servers. A single partition server can handle multiple partitions. But for a given partition, a single partition server is in charge of addressing all the requests to this partition, while the effective data of the partition can be stored on multiple nodes (we refer the reader to Subsection 3.3.2 for more in depth explanations). The WAS monitors the usage pattern of the request on the partitions and can dynamically and automatically reconfigure the map between partitions and partition servers.

Since all the requests related to a single partition are addressed by a single partition server, the partition is therefore the smallest unit of data controlled by Azure for load-balancing. The second load-balancing system is handled by the user. This second system refers to the user's ability to adjust the partition granularity so that a given partition handles more or fewer data. More specifically, the WAS consumer needs to tune the partition granularity with respect to an atomicity/ scalability tradeoff. The bigger a partition is, the more transactions on multiple data pieces can be done atomically. Yet, since all the requests to this partition are addressed by a single partition server, the bigger a partition, the busier the partition server.

For the BlobStorage, since each object is stored in a different partition key, access to the different blobs (even with the same containerName) can be load-balanced across as many servers as needed in order to scale access, but no atomicity for transactions is possible on Azure level. On the contrary, the TableStorage provides the transactions run on a group of entities sharing the same partition key with atomicity.

3.3.2 Elements of internal architecture

The internal implementation of the WAS is rather complex and leverages a multi-layer design. This design lets the storage be at the same time strongly consistent, highly available and partition tolerant. Providing at the same time these three properties is a difficult challenge, at least theoretically, due to the CAP theorem. This subsection provides a short insight of the underlying infrastructure and shows how the previous guarantees are achieved despite the difficulty mentioned

above. We refer the reader to [36] for a more in-depth presentation of the WAS underlying architecture.

Storage stamps

Following the terminology of [36], the storage instances in a data center are divided into storage stamps. Each storage stamp is a cluster of about 10-20 racks, each rack holding about 18 disk-heavy storage nodes. While the first generation of storage stamps holds about 2PBytes of raw data in each stamp, the next generation holds up to 30PBytes of raw storage in each stamp [1]. To lower the cost of cloud storage, Azure team keeps the storage stamps highly utilized. But to keep sufficient throughput and high availability even in the presence of a rack failure, each stamp is not used above 70% of its capacity. When a storage stamp is filled up to this bound, the location service migrates some accounts to different stamps, to keep the storage stamp on a capacity usage ratio of 70%.

Each rack within a storage stamp is supported on isolated hardware: each rack is supplied in bandwidth and energy with independent and redundant networking and power, to be a separate fault domain.

The data centers hosting WAS services are spread in several regions of North America, Europe and Asia. In each location, a data center holds multiple storage stamps. When creating a new cloud application on Azure, customers are asked to choose a location affinity among these three continents. While the cloud consumer cannot choose the exact data center location to be primarily used by its application, this location affinity choice allows the user to lower the application latency and improve its responsiveness.

Front End Layer

The Front-End layer is composed of multiple servers processing the client's incoming requests and routing them to the partition layer. Each front-end server holds in cache the mapping between partition names (as defined in Subsection 3.3.1) and partition servers of the partition layer. When receiving a client's

1. These numbers may seem to be surprisingly high in view of the total amount of data stored in Azure presented in Section 3.3. All these numbers yet come from the same source: [36]. A first response comes from all the data replicas that are created for each data piece. A second key aspect may be the Content Delivery Network (CDN), a special Azure service devoted to serve content to end users with high availability and high performance. Finally, the quantity of machines dedicated to the storage system may be small in comparison to the total amount of machines required by computations.

request, the Front-End layer server authenticates and authorizes the request, then routes it to the partition server that primarily manages the partition name related to the request. The most frequently accessed data are cached inside the Front-End servers and are directly returned to corresponding requests to partially unload the partition layer servers.

Partition Layer

The Partition layer is composed of multiple servers managing the different partitions. As already stated in 3.3.1, one partition server can manage multiple partitions, but a given partition is managed by only one partition server. For the BlobStorage, this is obvious since the partition key is down to the blobName. A partition master monitors each partition server's load and tunes the load-balancing process.

Stream Layer

The Stream layer is responsible for persisting data on disk and replicating the data across multiple servers within a storage stamp. The stream layer is a kind of distributed file system within a storage stamp.

Following the terminology of [36], the minimum unit of data for writing and reading is called a "block". A typical block is bounded to 4 MBytes. An extent is a set of consecutive and concatenated blocks. Each file stored by the stream layer is referred to as a stream. Each stream is a list of pointers referring to extent locations.

The target extent size used by the partition layer is 1 GBytes. Very large objects are split by the partition layer into as many extents as needed. In the case of small objects, the partition layer appends many of them to the same extent and even in the same block. Because the checksum validation is performed at the block level, the minimum read size for a read operation is a block. It does not mean that when a client requests a 1 kilobyte object the entire block is sent back to the customer, but the stream layer reads the entire block holding the object before returning the actual small object.

Because a stream is only an ordered collection of pointers to extents, a new stream can be very quickly created as a new collection of pointers of existing extents.

A key feature of the stream layer is that all writes are append-only. When a write occurs, the last extent is appended to one or multiple blocks. This append operation is atomic: either all the entire blocks are appended, or none are. The atomicity is guaranteed by a two-step protocol. Blocks are firstly written into the disk and appended after the last block of the last extent of the stream. Once these write operations are completed, the stream block and extent pointers list are updated accordingly.

An extent has a target size, defined by the partition layer. When an append enlarges an extent to the point it exceeds this target size, the extent is sealed and a new extent is added to the end of the stream. Therefore, all the extents but the last one are immutable in a stream and only the last one can be appended.

Partition servers and stream servers are co-located on the same storage nodes.

Extent replication and strong consistency

A key feature of data storage is data persistence, durability and availability. Since data servers are expected to fail from time to time, the data are duplicated several times, so a data server can die without any data loss. Yet, in the case of a networking partitioning, availability and consistency are two properties that are hard to meet at the same time, as stated by the CAP theorem (see 2.4.2).

The default azure policy consists in keeping three replicas for each extent within the same storage stamp (according to [36]). This policy is referred to as the intra-stamp replication policy and is intended to guarantee no data loss, even in the event of several disks, nodes or rack failures. A second policy designed to prevent data loss in the rare event of a geographical disaster is called inter-stamp replication policy. This replication process ensures that each data extent is stored in two different data centers, geographically distant. This subsection details how the inter and intra-stamp replication policies fit into the CAP theorem constraints.

The stream layer is responsible for the intra-stamp replication process. When an extent is created, the Stream Manager promotes one of the three extent nodes (EN) into a primary EN. The partition layer is told of the primary EN, and all the write requests of the partition layer are sent to this primary EN. The primary EN is in charge of applying the write request to its extent but also to make sure the two other replicas are updated in the same way. It is only once all the three extents have been updated that the partition layer is notified by the primary EN of the success of the write request. Therefore, this update process is carried out as quickly as possible since it impacts the customer's request latency as no write

success is returned before all the extents are updated. As the three extents are located in the same storage stamp, this update process can be quick. This update mechanism ensures strong consistency among the three extent replicas, and each read request can be addressed by any of the EN since the three replicas are always guaranteed to be the same.

During a writing event, if an extent node is unreachable or the write operation fails on this node, the partition layer is notified the write operation has failed. The Stream Manager then seals the extent in the state it was before the write failure, adds a new unsealed extent at the end of the stream and promotes one of the three extent nodes as a primary node for this extent. This operation is reported in [36] to be completed within 20ms on average. The partition layer can therefore resume the write operation in a short time without waiting for the failing nodes to become available or operational again.

The partition layer is responsible for the inter-stamp replication process. It is a low priority *asynchronous* process scheduled so that it does not impact the customer's requests performance. Contrary to the intra-stamp replication process run by the stream layer, the inter-stamp replication process is not designed to keep data durable in the event of hardware failures. Rather, it is designed to store each data chunk in two distinct data centers to obviate the rare event of a geographical disaster happening in one place. It is also used to migrate accounts between stamps to preserve the 70% usage ratio of each storage stamp. This replication process does not guarantee strong consistency.

3.3.3 BlobStorage or TableStorage

BlobStorage and TableStorage are both No-SQL storages, designed to store and persist data through a key-value pairs format. Both of them are well integrated into the Azure Platform solution. We now investigate some elements that need to be taken into account while choosing one storage or the other:

– When dealing with really large objects, BlobStorage requests are easier to express, since each blob can be arbitrarily large (up to 50 GBytes per blob). We do not need to break an object into a lot of entities, making sure each entity conforms to the size constraint on entities.

– When manipulating objects that can be stored both in TableStorage and BlobStorage, BlobStorage is reported by [69] to be more effective in insert opera-

tions on blobs larger than 4KBytes.

– As already expressed in Subsection 3.3.1, data locality management and atomicity concerns can also impact the choice between BlobStorage and TableStorage. While it is still possible to design custom primitives for multi-blobs atomic transactions (see Subsection 4.5.3), these primitives are not native and less efficient.

– Price is another factor that might affect our choice. Using Azure Storage, three things are charged : the storage used size, the I/O communications, and the requests to the storage. A single query on the TableStorage can load up to 100 elements at a time and costs as much as a single query on the BlobStorage (see [3]). When dealing with many small objects, using TableStorage will thus allow to run requests on objects by groups of 100 entities, so it would divide up to a factor 100 the bill of storage requests in comparison with running the corresponding requests one by one on the BlobStorage. If we do not need to store the objects on different places, we could store them in one bigger blob, bounding the request to one. Such a design choice would lead to significant improvements when all the items thus grouped are all read or updated at the same time. In the case of random read and write accesses to a pool of items, grouping them into a single item leads to significant I/O communication increase. The usage of the TableStorage while dealing with small objects therefore seems to be more adequate but special attention is required.

All the cloud experiments run on Chapter 5 and Chapter 7 have been implemented using the BlobStorage. Atomic updates on multiple objects at once will not be mandatory for most of our storage requests. Besides, the BlobStorage is easier to manipulate and due to its complexity, the TableStorage suffered some bugs when we started our experiments (in June 2010).

3.4 Azure Performances

3.4.1 Performance Tradeoffs, Azure Positionning

As outlined in [43], no one cloud system can be said to be the best for all the application requirements, and different storage systems have made different choices while facing some tradeoffs. Before providing some raw numbers of Azure performances, we try to position the Azure framework in comparison with these tradeoffs:

Read/Write tradeoff: There is a natural tradeoff between read throughput and write throughput. A piece of data cannot be at the same time highly available to both read and write operations: having a highly available item to read operations requires the item to be duplicated multiple times; write operations must then change all the replicas, which slows down the write operations. Azure lets the traffic arbitrate this tradeoff: the more requested a blob is per time unit, the more duplicated the blob is and the more available it becomes. In return, the more duplicated the blob is, the longer it takes to run write operations. Azure also implements an in-memory mechanism that mitigates this tradeoff: the most requested blobs (the hot blobs) are kept in cache rather than on hard-drive to lower read and write latencies. Such a mechanism is adopted in many other storage systems. The impact of both these mechanisms must however be tempered by the fact they rely on requests which are spread over time: to detect that a blob is "hot" and that it needs to be both duplicated and put into cache, Azure Storage first needs to notice a significant amount of requests before launching these mechanisms. This design makes the mechanism ineffective in the case of a blob which is first read by multiple workers at the same time, but then never read again. Such a scenario is frequent in the algorithms presented in Chapters 5 and 7.

Consistency/Persistence/Latency tradeoff: Data persistence is a mandatory feature for a storage service such as Azure Storage. To prevent client data loss, Azure stores each single piece of data multiple times in different places to guarantee data persistence [2]. As explained in Subsection 3.3.2, such a replication mechanism induces longer write operations when strong consistency is also guaranteed. To lower the overall write latency, Azure implements a two-level replication mechanism. The first one, referred to in Subsection 3.3.2 as intra-stamp mechanism, is a synchronous process: the write operations returns succeed only when all the replicas have been updated. Since all those replicas are stored in a single storage-stamp, all the writes are likely to be completed in a small amount of time. This mechanism guarantees strong consistency among all these replicas. The second replication process, referred to as the extra-stamp mechanism, is an asynchronous mechanism: when the storage is not stressed, the data piece is replicated in another geographically distant storage-stamp. Such a mechanism

2. Such guarantees need to be viewed with caution. Despite Microsoft commitments and their Service Level Agreement (SLA), the duplication mechanism can only protect data from hardware failures, not from software bugs or human manipulation errors (as it is the case for all the technologies). This has been evidenced several times, for example by Amazon EC2 outage started April 21th 2011, or Azure outage started February 29th 2012 (for which Lokad has suffered from web-site downtime for several days and temporary loss of a part of its data).

does not guarantee strong consistency, but is only used as back-up in the very unlikely event of a complete loss of the first whole storage-stamp.

CAP guarantees / latency tradeoff: This tradeoff has already been partly discussed in the Consistency/Persistence/Latency tradeoff. As stated by the CAP theorem, a distributed storage cannot be 100% strongly consistent, 100% available and 100% partition tolerant. Azure has chosen to guarantee none of these properties at 100% rate. The strong consistency is guaranteed provided that the whole primary storage stamp of a given data chunk is not made completely unavailable (which is an event very unlikely to occur and which didn't occur to our knowledge during all our months of experiments). The system is also neither 100% available nor 100% partition tolerant. Yet, all in all Azure framework is performing well enough for this lack of guarantees to be hardly noticed.

Synchronous actions / Responsiveness tradeoff : Many asynchronous methods are designed to improve responsiveness. In the special case of an API, it is advised that methods that are internally implemented synchronously should be exposed only synchronously and methods that are internally implemented asynchronously should be exposed only asynchronously (see for example [16]). To our knowledge, this is the design adopted by Azure Storage. More specifically, all the List, Get, Push and Create methods are all exposed synchronously. Only one method is exposed asynchronously: it is the container deletion method. The main priority performance targets of the WAS are low latencies for read and write operations in addition to throughput. The container deletion method, unlikely to be used very often, is not designed to be efficient. This method is therefore made asynchronous and cloud consumers should not recreate a freshly deleted container.

3.4.2 Benchmarks

Benchmarking a storage system is a challenging task involving a lot of experiments and measurements in multiple contexts. Some previous works have already been made to evaluate some cloud storage solutions. For example, the Yahoo! Cloud Serving Benchmark ([43]) provides an open-source framework to evaluate storage services. Other measurement works have been made on specific cloud storage solutions, such as [111] or [110] for Amazon and [69] or the (now removed) AzureScope website ([6]) for Azure.

In the context of our work, we do not aim to produce such a rigorous and comprehensive evaluation of the Azure services performances. Yet we have observed that the performances of Azure depend on the context in which the requests are

applied (in particular, the size of the data that is uploaded or downloaded, the API used to communicate with the BlobStorage, the serializer/deserializer that is used, etc. are impacting performances). As a consequence, even the previously mentioned papers that have been run in a rigorous context might be inaccurate in our specific situation. Because of this, we provide in this subsection naive measurements made in the specific context of our clustering experiments (see Chapter 5 and Chapter 7). In particular, the storage bandwidth as well as the CPU performance are investigated as they are of prime interest for the design of our algorithms.

Bandwidth between the storage and the role instances

Contrary to [69], we investigate the read and write bandwidth for small data chunks (several MBytes). To do so, we pushed 8 blobs of 8 MBytes into the storage, and we measured the time spent to retrieve them. For a single worker using a single thread, we retrieved the blobs in 7.79 seconds on average, implying a 8.21 MBytes/sec read bandwidth. We tried to use multiple threads on a single worker to speedup the download process, and we found out the best read bandwidth was obtained using 5 threads: we retrieved the 8 blobs in 6.13 seconds on average, implying a 10.44 MBytes/sec read bandwidth. The multi-threads read bandwidth we observed differs from what is achieved in AzureScope. This may be explained by two potential factors: we observed an average bandwidth whereas AzureScope observed peak performances and we did not have the same experimental environment. In the same manner, similar experiments have been made to measure the write throughput. We obtained average throughput of 3.35 MBytes/sec write bandwidth.

This benchmark is very optimistic compared to the downloading context of our clustering experiments run in Chapter 5 because of a phenomenon referred to as the aggregated bandwidth bounds discussed in Subsection 5.5.2. In addition to this aggregated bandwidth bound phenomenon, multiple concurrent I/O operations run in parallel can sometimes overload a specific partition layer server, resulting in slower request responses: when running experiments with a large number of workers reading and writing in parallel, we sometimes experienced blobs already pushed into the storage becoming available only after 1 or 2 minutes.

Workers Flops performances

In the same way as for the storage bandwidth, the workers' CPU cadency highly depends on the kind of tasks the workers are assigned to. All our experiments were run on Azure small VM that are guaranteed to run on 1.6 GHz CPU. Yet because of virtualization we do not have any warranty in terms of effective Floating point operation per second (Flops). On any architecture, this Flops performance highly depends on the nature of the computation that are run and especially of the code implementation (see for example Drepper in [52]). Therefore, to fit our predictive speedup model developed in Chapter 5, we ran some intensive L2 distance computations to determine how fast our algorithm could be run on these VM. As a control experiment, the code was first run on a desktop Intel Core 2 Duo T7250 2*2GHz using only one core. We noticed that our distance calculations were performed with a performance of 750 MFlops on average. We then ran the same experiment on Azure small VM and we got for the same code a performance of 669 MFlops.

It is worth mentioning that differences in processing time between workers have been observed. These observations are developed in Subsection 5.5.6 of Chapter 5.

Personal notes

During all the experiments that we have been running (from September 2010 to August 2011), we have noticed various Azure oddities. The following remarks do not come from rigorous experiments designed to prove these oddities, but as side observations made while running our clustering algorithms.

– *Inter-day Variability of Azure performances*: during the months of our experiments, we have noticed several days for which the Azure storage performances were significantly below average performances (from 30 to 50% below). [69] reports that *"the variation in performance is small and the average bandwidth is quite stable across different times during the day, or across different days"*. This different result may come from the different environments where the measurements took place. More specifically, our experiments have been run on the Azure account of Lokad, account for which other hosted services were sometimes run concurrently to execute heavy computations. It is also noteworthy to mention the date of those experiments. The experiments run by [69] were run from October 2009 to February 2010 (during the Community Technology Preview and before the commercial availability of Azure), while our experi-

ments were run from September 2010 to August 2011. Such variations make performance measurements of distributed algorithms harder. Especially, our experiments therefore need to be performed several times before conclusions can be drawn.

– *Sensitivity to the serialization mechanism*: different serializers (to name but a few of them available in .NET: Binary, DataContract, XML, zipped-XML, JSON, ProtocolBuffers, etc.) are available to serialize an in-memory object into a stream then stored on a disk or sent through the network. The choice of the serializer has an impact on the overall communication latency and with storage duration.

– *Temporary unavailability of a blob*: we have been experiencing several times the temporary (around 15 minutes) total unavailability of a given blob when this blob was read by about 200 machines simultaneously. We have not managed to isolate the exact reason to account for this phenomenon.

– *Temporary unavailability of the storage*: we have been experiencing several times the temporary (around 15 minutes) total unavailability of the BlobStorage. We have not managed to isolate the exact reason for this phenomenon.

– *Lower throughput for smaller blobs*: while the experiments in [69] have been run on a blob of 1GBytes, a lot of interactions with the storage are made on much smaller blobs. The throughput for these smaller blobs is sometimes reduced.

– *Code redeployment, VM pool resizing*: During our experimenting year, we noticed impressive improvements in code redeployment and dynamic worker resizing. More specifically, our first code redeployments (December 2009) were taking hours (with redeployments failing after more than four hours). In August 2011, code redeployment was systematically taking less than 10 minutes. This redeployment mechanism is still improvable: for example, Lokad.Cloud (see Subsection 4.5.4) provides a code redeployment within several seconds through a clever trick of separate AppDomain usage (see [2]). Over the same period, dynamic worker resizing was also significantly improved, with waiting time reduced from approximately one hour to several minutes (at least when requiring less than 200 small role instances).

– *Queues pinging frequency*: Workers are pinging queues on a regular basis to detect if there are messages to process. To limit the number of network pings, a standard schedule policy consists in pinging the different queues only once

per second. This policy introduces a small additional latency.

3.5 Prices

The pricing of Microsoft Azure is driven by the pay-as-you-go philosophy of Cloud Computing. It requires neither upfront costs nor commitments [3]. Cloud consumers are charged for the compute time, for data stored in the Azure Storage, for data queries, etc. This section gives a short overview of the Azure pricing system. We refer the reader to [4] for an accurate pricing calculator and to [5] for more in-depth pricing details.

CPU consumption is measured in hours of usage of VM. Figure 3.1 presents the Azure pricing for the different VM sizes. The use of Azure Storage is charged following three criteria: the quantity of data stored by Azure storage($ 0.125 per GBytes stored per month), the quantity of data which is transferred from Azure storage to other machines ($ 0.12 per GBytes for North America and Europe regions) (all the inbound data transfers or between 2 Azure instances are free), and the number of requests addressed to the storage ($1 per 1,000,000 storage transactions).

This pricing can be lowered through subscriptions and are expected to be cut in a near future: indeed, Azure pricing is in general close to Amazon pricing that has been cut nineteen times in just six years.

Instance	CPU Cores	Memory	Disk	I/O Perf.	Cost Per Hour
Extra Small	Shared	768 MB	20 GB	Low	$0.02
Small	1 x 1.6 GHz	1.75 GB	225 GB	Moderate	$0.12
Medium	2 x 1.6 GHz	3.5 GB	490 GB	High	$0.24
Large	4 x 1.6 GHz	7 GB	1,000 GB	High	$0.48
Extra Large	8 x 1.6 GHz	14 GB	2,040 GB	High	$0.96

Table 3.1: Compute instance price details provided by Microsoft on April 2012. http://www.windowsazure.com/en-us/pricing/details/.

The costs described above seem to traduce the fact that storing data is much cheaper than using computing units. For many applications hosted by Azure, most of the costs will therefore come from the VM renting.

3. No commitments are required, but customers can receive significant volume discounts when signing for commitment offers for several months. In addition, the development cost to migrate an application on a Cloud Computing Platform is often significant, which is a form of strong commitment.

Chapter 4

Elements of cloud architectural design pattern

4.1 Introduction

The SaaS cloud solutions presented in Chapter 2 have already proved to be a successful economic and technological model, as demonstrated for example by Amazon or Gmail. In parallel, the IaaS cloud solutions have also proved to be good candidates for many customers, as suggested by the long list of some of Amazon Web Service customers [1] or by their business volume (see [1]). In contrast, the PaaS solutions are younger and address a different market.

The PaaS cloud solutions are deemed to be easy to manipulate. Indeed, many tools are provided to improve the developer efficiency. In addition to these tools, many abstractions and primitives are already available to prevent the PaaS customers from re-implementing the same engineered solutions multiple times. For example, the Azure Queues provide an asynchronous mechanism that helps to build loosely coupled components at a low development cost. However, the development of cloud applications requires specific considerations when the application is expected to have a certain level of complexity or to achieve a certain scale-up. The recent interest for PaaS solutions has therefore driven some research about design patterns for improved cloud application development (see e.g. [64] or [44]).

In addition to the general Cloud Computing design patterns, the question of design patterns in the context of CPU-intensive cloud applications arises. To our knowledge, the Azure platform has not been originally designed specifically to host intensive computations. While the overall system has been designed to

1. http://aws.amazon.com/solutions/case-studies/

provide very satisfactory scale-up (in terms of the number of available computing units or of the aggregated bandwidth bounds) for many cloud applications, the Azure PaaS system does not provide all the specific software primitives that CPU-intensive applications would require. In this chapter, we investigate the design of large-scale applications on Azure and provide some cloud design patterns that have appeared as mandatory when working on these applications.

Since Azure has not been initially designed to be a scientific computing platform, no computation framework (with the exception of Windows Azure HPC presented in the introduction of Chapter 3) is presently available for Azure: Azure provides neither a "low-level" framework such as MPI, nor a "high-level" framework such as MapReduce. While the Azure team has reported that they are working on implementing the MapReduce framework for Azure, such a framework is not available yet. In this situation, one can only resort to the abstractions and the primitives which are already available (see for example Subsection 3.3.1) to build CPU-intensive applications. These abstractions and primitives can turn out to be inadequate for CPU-intensive application development.

During our Ph.D. work, we have been involved in four CPU-intensive applications. Two of them are the cloud Batch K-Means prototype and the cloud Vector Quantization prototype [2] that are described in detail in Chapters 5 and 7. In addition to these two prototypes, we have worked on two applications run by Lokad that are not open-source. The first one is the Lokad time series forecasting engine: it is the scientific core of Lokad applications which is hosted on Azure and provides an API that is in charge of building and returning the actual forecasts. This engine is used in production and benefits from the scale-up elasticity and the robustness provided by Azure. The second application is Lokad benchmarking engine, which replicates the Lokad forecasting engine but which is used as an internal tool to monitor, profile and evaluate our forecasting engine in order to improve the accuracy of the forecasts, increase the scale-up potential and reduce the computation wall time (see definition in section 5.3.1). This application does not require the same robustness as the Lokad forecasting engine, but faces many challenges, such as a complex reporting process for which the parallelization is both mandatory and difficult.

In the light of the experience obtained while working on these four applications, this chapter describes engineering considerations about the software development of cloud applications with a specific focus on CPU-intensive issues when it is relevant. It is organized as follows. Section 4.2 describes how the communications

2. Both of these prototypes are available at http://code.google.com/p/clouddalvq/

can be implemented between the different processing units of a given worker role. Section 4.3 presents recurring elements of cloud application architecture design. Section 4.4 presents some elements of the design required to provide the Azure applications with scalability. Section 4.5 provides additional elements, especially the idempotence constraint. Section 4.6 presents a synchronization primitive that happened to be necessary in many situations we encountered.

4.2 Communications

4.2.1 Lack of MPI

The Message Passing Interface (MPI) is an API standard that defines the syntax and semantics of libraries that provide a wide range of routines to automatically harness direct inter-machine communications in a distributed architecture. More specifically, the various MPI implementations provide an abstraction layer that disburdens the application developer from manual management of inter-machine communications. In Section 5.3, we highlight the importance of MPI for the distributed Batch K-Means on Distributed Memory Multiprocessors (DMM) architectures. We show how significant the impact of the actual implementation of MPI primitives is on the speedup that can be achieved by the distributed Batch K-Means on DMM architectures: this is the tree-like topology of the communication patterns of MPI primitives that results in a $O(log(M))$ cost for averaging the prototypes versions where M is the number of processing units.

Direct inter-machine communications are also available in Azure. As explained in Section Azure Compute (Section 3.2), an internal endpoint is exposed by each processing unit of Azure so that each of the processing units could talk directly to the others using a low-latency, high-bandwidth TCP/IP port. The bandwidth of these direct communications is very sensitive to multiple factors such as the number of units that are communicating at the same time (because of aggregated bandwidth bounds), the fact the resource manager may allocate other VM instances from other deployments (applications) on the same physical hardware, the fact these other VM may also be I/O intensive, etc. Yet, the average behavior of such direct communications is very good: [69] and [6] report direct inter-machine communication bandwidth from 10 MBytes/sec to 120 MBytes/sec with median measurement bandwidth of 90 MBytes/sec.

While these direct inter-machine communications are available on Azure, no framework has already been released on Azure to provide higher level primitives

such as the merging or the broadcasting primitives provided by MPI.

A key feature of the Azure VM system is that the cloud client cannot get any direct topology information about the VM he temporarily owns. On the contrary, the VM are totally abstracted to disburden the cloud client from these considerations. Such a design does not prevent from building a MPI-like API but makes it more difficult. Several works have ported MPI on Cloud Computing platform such as EC2 (we refer the reader for example to [88] or [14]) but no MPI implementation is available on Azure.

Because of the importance of bandwidth in the speedup performance of many distributed machine-learning algorithms such as Batch K-Means, and because the direct inter-machine bandwidth is much higher than the bandwidth between the storage and the processing units (see Subsection 3.4.2), MPI would be a very useful framework for Azure, as far as intensive computing is concerned.

4.2.2 Azure Storage and the shared memory abstraction

The Azure Storage service provides a shared memory abstraction in the form of the BlobStorage (and of the TableStorage, but because of their similarity we only mention BlobStorage in this subsection). The different processors of a Symmetric Multi-Processors (SMP) system (see Section 5.3) can access the shared memory implemented in RAM to read or write data and therefore use this shared memory as a means of communication between the different processors. In the same way, all the different computing units of an Azure application can access any of the blobs stored into the BlobStorage.

There is however a critical difference between the use of a shared memory in a SMP system and the use of the BlobStorage as a shared memory in a cloud environment: it is the latency and the bandwidth. Indeed, in the SMP case, each processor is very close to the shared memory and efficient means of communication such as buses are used to convey the data between the shared memory and the processors. In contrast, the BlobStorage data are stored on different physical machines from those hosting the processing VM. Therefore, the communication between the computing units and the BlobStorage are conveyed through TCP/IP connections between the distant machines. The resulting bandwidth is much lower than the bandwidth of a real shared memory: around 10MBytes/sec for reads and 3 MBytes/sec for writes (see Subsection 3.4.2). In return, the communication through the BlobStorage provides persistency: if a worker fails and a task is restarted, many partial results can be retrieved through the storage, therefore

mitigating the delay for job completion induced by this failure.

4.2.3 Workers Communication

The previous subsections have presented two approaches for the communications: the first one —a direct inter-machine mechanism through IP communications— is efficient but made difficult by the lack of any abstraction primitive as MPI would provide: re-implementing MPI-like primitives in an environment where the computing units (which are VM) are likely to fail, to be rebooted or to be moved from a physical device to another one, would be a demanding task that would be similar to implementing a peer-to-peer network. The second approach, which consists in communications through the BlobStorage, is designed to be both simple and scalable, but is inefficient (because of the low bandwidth between the workers and the storage).

Both approaches seem valid but lead to different designs, challenges and performances. We have made the choice of resorting to the BlobStorage and to the QueueStorage as communication means between workers because of the two following reasons. Firstly, beyond the pros and cons of each approach, the communication through storage seems to be more suited to the Azure development philosophy. Indeed, this is the way communications are made in every Microsoft code sample provided in tutorials, and it is also suggested by the Azure QueueStorage, which has been specifically designed for this approach. Secondly, because of its simplicity over the "peer-to-peer" communication model, the communication through the storage model is the one that has been chosen by Lokad for both the forecasting engine and the benchmarking engine. Resorting to the same mechanism for our clustering prototypes was therefore an industrial constraint.

4.2.4 Azure AppFabric Caching service

Caching is a strategy that consists in storing data in the RAM of a machine instead of storing it on a hard-drive. Thanks to this strategy, the data can be accessed more quickly, because one does not need to pay for the cost of hard-drive accesses. The caching mechanism is therefore known to improve applications performance.

Some cloud providers provide a caching service, such as Google App Engine, AppScale and Amazon Web Services (through Amazon Elasticache). In the same way, the Azure AppFabric Caching service (Azure Caching for short) provides

a caching solution for the Azure applications. This system became available only after a large part of our clustering algorithm implementations had been developed and benchmarked. As a consequence, it has not been used in the cloud algorithms presented in Chapters 5 and 7. We believe the Caching service may yet lead to significant throughput increase, especially during the prototypes versions communication between the processing units (see e.g. Chapter 5).

4.3 Applications Architecture

4.3.1 Jobs are split into tasks stored in queues

Following the terminology of the original MapReduce research paper ([47]), the total execution of an algorithm is referred to as a job. A given job is divided in multiple tasks that stand for the elementary blocks of logic that are executed. Each task is run by a single worker, and a single worker can only process one task at a time. During the duration of an entire job, each processing unit is expected to run successively one or several tasks.

If the number of tasks of a given job is lower than the number of processing units, or if one or several workers are temporarily isolated from the rest of the network, some workers may process no task during a given job execution.

The multiple tasks are described by messages stored in queues. More specifically, to each task is associated a message queued in the QueueStorage. When a processing unit is available, it pings a queue and dequeues a message that describes a specific task. The message is kept in the queue but becomes invisible to all the other workers for a specific fixed "invisibility timespan". Once the task is completed, the corresponding processing unit notifies the queue that the task has been successfully completed and that the corresponding message could be safely deleted, then the processing unit pings the same or another queue to get a new message and then processes the new corresponding task. If a worker fails before notifying the queue that the message has been successfully processed, the message becomes visible again to other machines after the invisibility timespan. The number of tasks does not need to be defined when the job is started. On the contrary, a common Azure application design is that many tasks, once completed, produce one or several new tasks in return.

Among all the tasks that are processed during a given job, many of them refer to the same logical operation applied on multiple distinct data chunks of the same

type, a situation often described in the literature as a data-level parallelism. To reflect this data parallelism, a frequently chosen design consists in gathering in the same queue only the messages whose corresponding tasks refer to the same logical operation applied on distinct data chunks. In such a design, to each queue is associated a QueueService which holds the logical operations to be applied on each item of the corresponding queue.

4.3.2 Azure does not provide affinity between workers and storage

As outlined in the original MapReduce research paper ([47]), *"Network bandwidth is a relatively scarce resource in [our] computing environment"*. The MapReduce framework has therefore adopted a design where the data are not sent to the processing units, but the code to be executed is sent by the framework environment on the machines actually storing the data to avoid network contention (see also Subsection 2.5.1). This strategy is often referred to as data/CPU affinity.

Presently, there are no mechanisms in Azure to provide such an affinity between the workers and the storage. Indeed, data are accessed through the BlobStorage and the TableStorage but the machines that actually store the data (and run the stream layer described in Subsection 3.3.2) are totally abstracted. As a result, we cannot run any tasks on the machines where the corresponding piece of data are stored. Each worker processing data will therefore need to first download the data it needs to process, contrary to Google's MapReduce, with which the tasks are scheduled and assigned on each worker by the framework in such a way that data downloading is minimized.

4.3.3 Workers are at first task agnostic and stateless

When a worker role (see Subsection 3.2) is deployed, a certain number of processing units (the role instances) which are controlled by the customer are assigned to the worker role to run the underlying code instructions. In addition to the event of a role instance failure, the Windows Azure Fabric can arbitrarily and silently move a VM from a physical machine to another one. Because of these events, any role instance needs to be resilient to reboot or displace, and therefore must be stateless, i.e. it must not store, between two tasks, data (in RAM or in the local hard-drive) that are essential to successfully complete the worker role logical operations.

Let us describe three properties shared by all the role instances of a given worker role. Firstly —as just explained— the role instances are stateless. Secondly, no instance role stores locally a part of the data to be processed (see Subsection 4.3.2). Finally, all the role instances have the same size (small instances/medium instances/big instances, etc.). As a consequence of these three properties, all the role instances are exchangeable and the mapping between the tasks and the role instances can be done arbitrarily without undergoing any performance drop.

This behavior has been observed in the development of Lokad forecasting engine or Lokad benchmarking engine. However, we have found out that in the context of our clustering algorithms a significant reduction of role instances I/O could be obtained when slightly relaxing the stateless hypothesis and partially dropping the previous commutability characteristic.

Indeed, many distributed machine-learning algorithms require that the total data set should be split among the multiple processing units so that each processing unit loads into its own local memory a part of the total data set, and that multiple computations are made on this partial data set. In this situation, reloading the partial data set for each computation to be done leads to a very large I/O overhead cost with the storage. This overhead cost can be significantly reduced by making the processing units keep the partial data set within their local memory until the algorithm is completed. When adopting this design, we reintroduce a data/CPU locality by assigning to each processing unit all the computations related to a specific partial data set. This design, close to the memoization technique, is used in the algorithms presented in Chapters 5 and 7. More specifically, each processing unit is assigned to the processing tasks of a specific partial data set. When the partial data set is needed, the processing unit checks if the data set is already available in the local memory. If the data set is not available, it is downloaded from the storage and kept into memory for the subsequent uses.

4.4 Scaling and performance

4.4.1 Scaling up or down is a developer initiative

Through the management API or the Azure account portal, the Azure account manager can modify the number of available workers. Contrary to many other hardware architectures, the application scaling-up-or-down is therefore left to the developer's choice.

In some cases, the estimation of an adequate number of processing units may be deduced by the initial data set to be processed. As an example, Chapter 5 provides an algorithm for which the knowledge of the data set size and of the framework performance (CPU+bandwidth) is sufficient to determine an adapted number of processing units. This is also the case of the Lokad forecasting engine in which the knowledge of the data set size (number of time series and length of each time series) is sufficient to determine how many workers are necessary to return the forecasts in one hour.

In other cases, the application scaling requirement may be more difficult to anticipate. In these cases, the auto-scaling system may rely on the approximative estimation of the queue size described in Subsection 4.5.2. More specifically, an adaptive resizing strategy may be adopted in which the number of processing units is increased as long as the number of the processing messages in the queues keeps growing, and in which the number of processing units is decreased when the number of queued messages is decreasing. Because of the friction cost of worker resizing (see Subsection 4.4.2), this resizing strategy is adapted only on the applications that are run during hours on end.

Depending on the granularity level set by the developer, the downsizing of the workers pool may be delayed until the algorithm is completed. Indeed, the Azure Management API does not provide any way to choose the workers to be shut down when downsizing the workers pool. If the granularity level is too coarse, the chance of shutting down a worker comes at a cost that may deter the user from downsizing the workers pool before all the computations are completed.

4.4.2 The exact number of available workers is uncertain

Several reasons may affect the number of available processing units. Firstly, following a resizing request, the re-dimensioning of role instances follows a general pattern in which a large majority of workers are allocated within 5 to 10 minutes (see for example [69]), but the last workers to be instantiated may take as much as 30 minutes before becoming available (in the worst encountered case). Secondly, a worker can be shut down by the Azure Fabric, the hosting VM can be moved on a different physical machine, or the physical machine can die because of a hardware failure. Finally, each processing unit may become temporarily unavailable (for example because of a temporary connectivity loss). Therefore, the number of available processing units may vary over time.

As a consequence, no algorithm should rely on an exact number of available processing units. If this number exceeds the quantity of workers expected by the algorithm, this excess of computing power often results in a waste: the additional workers are not of any help (this is the case for example in the algorithm presented in Chapter 5, in which the initial data set is split into M data chunks processed by M workers, and additional workers have no impact on the algorithm speedup). In contrast, the lack of any expected processing unit often results in dramatic performance reduction: in the event of a synchronization barrier, the lack of a single processing unit may result in a completion taking twice the expected time or no completion at all depending on the implementation.

Because of this cost asymmetry, a frequent design is to request a number of processing units slightly higher than the number of processing units that will be used by the distributed algorithm.

4.4.3 The choice of Synchronism versus Asynchronism is about simplicity over performance

While migrating Lokad forecasting engine and Lokad benchmarking engine to the cloud, we have observed a recurring pattern in which a part of the application could be schematized as a sequential set of instructions —referred to as $I_1, I_2, I_3, etc.$— applied independently on multiple data chunks, referred to as $D_1, D_2, D_3, etc.$ A specific task is associated to each pair of instruction and data chunk.

In this situation, no logical constraint requires the instruction I_j (for $j > 1$) to wait for the instruction I_{j-1} to be applied on every data chunk before I_j is applied on any chunk. Yet, we have observed that in this situation, adding artificial synchronization barriers to ensure that the requirement described above is guaranteed vastly contributes to simplify the application design and debugging. Provided the number of tasks for each instruction significantly exceed the number of processing units, the relative overhead of these artificial synchronization barriers is kept small.

On the contrary, when the tasks that can be processed in parallel are tailored to match the number of available processing units, the overhead of synchronization induced by the stragglers (see Subsection 2.5.1 or [23]) may be high, as described in our cloud-distributed Batch K-Means chapter (see Chapter 5). In this situation, the overhead is actually so big that it leads us to redesign the algorithm to remove

this synchronization barrier (see Chapters 6 and 7).

4.4.4 Task granularity balances I/O costs with scalability

The choice of task granularity is left to the developer. It results in a tradeoff between scalability and efficiency because of overhead costs. Indeed, the coarser the granularity, the less the processing units will pay for I/O with the storage and for task acquisition through the queue-pinging process. But the coarser the granularity, the less additional processing units may contribute to the speedup improvement and the more sensitive the overall algorithm will be to stragglers.

The design of the clustering algorithms of Chapters 5 and 7 has made the choice of very coarse-grained granularity: each of the processing unit is expected to process only one task for the total algorithm duration, to minimize the heavy storage I/O related to the data set download (see Subsection 4.3.3). In contrast, the Lokad forecasting engine and the Lokad benchmarking engine have been tuned to use a much finer granularity. In both cases, a general design pattern has been implicitly followed: the granularity is minimized under the condition that the I/O overhead induced by the granularity choice does not exceed a few percents of the total algorithm duration.

4.5 Additional design patterns

4.5.1 Idempotence

Multiple definitions of idempotence have been stated, but we refer the reader to the definition of Helland in [64]: *"idempotence guarantees that the processing of retried messages is harmless"*. The idempotence question arises in many cloud application situations when the messages queued can be processed multiple times.

Let us consider the cloud context of message processing: a processing unit consumes a message from a queue and then updates some data stored in a durable way (for example in the BlobStorage). The message is first consumed by the worker, communication is then made between the worker and the BlobStorage, and data are updated in the BlobStorage. After the update of the data, the worker may die before it tells the queue that the message has been successfully processed and should be deleted (see section 4.3.1). Queues have been designed to react to machine failures in an "at-least-once" way: in the event of a machine failure, the

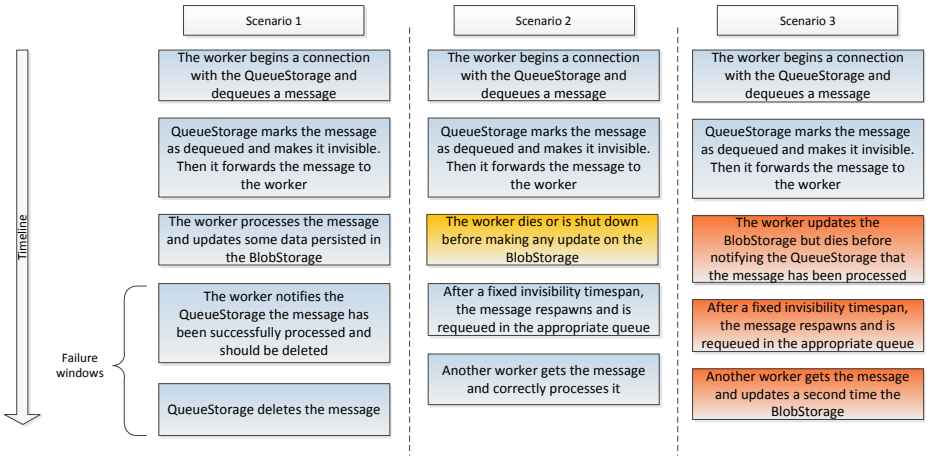

Figure 4.1: Multiple scenarii of message processing that impact the BlobStorage

message is re-queued after some timespan and another worker will process the same message, resulting in a second updating operation being performed on the same data stored in the BlobStorage.

Technically, this issue derives from the fact there is no direct mechanism to bind the message consumption and deletion with the message processing made by the worker so that one of these two events cannot happen without the other. The absence of this coupling leads to failure windows between the end of the data update and the message deletion in which the message is delivered and processed more than once. This scenario is illustrated in Figure 4.1.

Because of this “at-least-once” design, what Helland refers to as the recipient entity (which is the BlobStorage entity that may be subject to multiple updates) must be designed so that the repeated processing of a single message does not lead to a state for this entity which would be different from the one which could have been expected with a single processing. In practice, lots of message processing of our cloud applications are intrinsically idempotent. In particular, provided the processing is a deterministic function of the input message and that neither the entities that may be used nor the internal worker state may be modified, the message processing is idempotent. Among the tasks that suit this pattern, the most widespread case is the design of a work-flow in which each message processing is a deterministic function of its input and each entity stored in the permanent storage is *read-only*, i.e. it is not modified once it has been created and written

into the storage for the first time. In this case, when re-processing a message, the output is necessarily the same as the one obtained the first time, and the same blob is rewritten with the same value at the same place.

An important case that does not suit this pattern is the message processing that consists in a blob update, and in particular counter increments and decrements. In this case, a solution consists in the design of the recipient entity (i.e. the counter) so that the counter remembers the previous increments and decrements already made and that the update primitives are idempotent. The particular case of counter increments and decrements is explored in Section 4.6.

4.5.2 Queues size usage

The Azure QueueStorage API provides a method to get a rough estimation of the number of messages actually stored in a given queue, but no exact count is provided by the API. As a result, the API cannot be used as a synchronization barrier to start a process when x messages have been processed in a specific queue. In contrast, the approximative estimation of a queue size may be used to estimate the load of the corresponding QueueService.

4.5.3 Atomicity in the BlobStorage

As already stated in Subsection 3.3.1, the Azure BlobStorage does not provide any atomicity for transactions implying multiple blobs. Indeed, as outlined for example in [64], *"a scale-agnostic programming abstraction must have the notion of entity as the boundary of atomicity"*.

Despite this framework impossibility, it is still possible to design applications so that the transactions implying multiple objects (for example class objects) can be done atomically. The first solution consists in storing the multiple objects in a single entity (i.e. in a single blob, as far as the BlobStorage is concerned). This solution leads to some overhead cost: the read (resp. the update) of any of the objects stored in an entity requires the whole entity to be downloaded (resp. downloaded then re-uploaded). When the design of the application results in a situation in which the read/update of one of the objects stored in an entity and the reads/updates of the other objects stored in the same entity always happen simultaneously, this overhead is negligible.

A second solution consists in resorting to an indirection through a kind of "blob pointer". Let us examine the case of two objects stored in two different entities and that may be both involved in a single update operation requiring atomicity. In this context, the storage locations of the two entities may be unknown by the processing units and only accessed through a third entity that stores location of the two previous entities; this third entity being referred to in the following as the pointer entity. When a given processing unit needs to update one entity (resp. the two entities), it accesses the pointer entity, downloads the entity (resp. the two entities), locally updates it (resp. them), and re-uploads it (resp. them) into the storage in a new location (resp. two new locations). Then it updates the pointer entity so that it stores the updated location (resp. locations). In the event of a concurrent update of one or the two entities through the same mechanism, the Read-Modify-Write (RMW) primitive described in Chapter 3 guarantees that one update will succeed while the other will fail and will be restarted on the updated pointer entity and on the updated entity (resp. entities). Due to the indirection cost introduced by the pointer entity, this second solution use cases appear to be rare.

In the algorithms presented in Chapters 5 and 7, the prototypes versions and their corresponding weights are stored together in a single entity, following the first solution. On the contrary, sometimes the atomicity is not mandatory, as shown through the example detailed in Subsection 4.6.3.

4.5.4 Lokad-Cloud

Lokad-Cloud [3] is an open-source framework developed by Lokad that adds a small abstraction layer on top of the Azure APIs to ease the Azure workers startup and life cycle management, and the storage accesses. For example, in the same manner than TCP/IP provides a reliable (but "leaky", see [10]) abstraction using unreliable IP layers, Lokad.Cloud abstracts the storage read and write operations to handle most of issues that could be encountered while dealing with the WAS through exponential back-off retries. Lokad.Cloud has been awarded the 2010 Microsoft Windows Award. A slightly customized Lokad.Cloud framework has been used to build our cloud prototypes described in Chapters 5 and 7.

3. `http://code.google.com/p/lokad-cloud/`

4.6 The counter primitive

4.6.1 Motivation

While Azure is deemed to provide many primitives and abstractions that ease the application development, it does not presently provide a synchronization barrier primitive that is lifted once a given set of tasks has been completed. To implement this synchronization barrier we choose to resort to a concurrent counter stored in the BlobStorage [4] . This counter is initially set to the number of tasks that need to be processed and is decremented every time a task is completed. When this counter hits 0, all the tasks have been completed and the synchronization barrier can be lifted.

The design of synchronization barriers and counters in a distributed world has been studied in many contexts. In particular, special attention is paid to avoid memory contention in order to achieve overall slow latency and high throughput. To alleviate the memory contention, a classical pattern is to resort to a combining tree structure of counters that distributes the data requests on multiple pieces of data instead of a single one. This pattern, referred to as *software combining* in [67] has been used for more than a quarter of a century (see e.g. [115], [58], [66] or [67]).

4.6.2 Sharded Counters

Before presenting the adopted solution, let us describe a well-known type of distributed counters used in many cloud applications: the sharding counters ([60]). A sharding counter consists in a very simple set of distributed counters designed to limit the memory contention and therefore to achieve a much higher update rate. Instead of using a single counter that is incremented or decremented, the sharding counter owns L different counters referred to as shards. To increment or decrement the counter, a shard is chosen at random and this shard is incremented or decremented accordingly.

Sharding counters are designed to address the use-case in which the actual value of the counter may be slightly outdated or non-consistent when it is read, and the write operations (increment or decrement) are much more frequent than the read operations. Indeed, since the counter is split into L shards, the read operation

4. As explained in Subsection 4.2.4, this counter would probably vastly benefit from being stored in the Azure Caching service using the same implementation strategy than the one described below, but this service was released too late for our implementations.

requires to read the actual L values and to add them up. A well-known example of sharding counters use-case is the monitoring of online users for a given application to adapt the quantity of servers with the traffic-load. In this scenario, the traffic-load may be approximated with several percent of error margin without damaging the servers dimensioning logic.

In the context described in the previous subsection, the counter is read after each decrement to check if the value 0 has been hit. The sharding counters abstraction would therefore lead in our situation to unsatisfactory read latencies.

4.6.3 BitTreeCounter

Based on the observation that sharded counters were not suitable to our purpose and that no other cloud counter abstraction was available, we have developed a new implementation of a combining tree counter, referred to in the following as BitTreeCounter. The BitTreeCounter is designed to be a synchronization barrier primitive: the counter returns a boolean that is true only when the tasks of a given set are all completed.

Our BitTreeCounter is designed to lead to efficient use of optimistic read-modify-write concurrent operations (see Subsection 3.3.1) by implementing a set of dependent counters to avoid collisions and remove the contention issues.

In addition to the large throughput and the low latency required for many distributed counters, we also require our BitTreeCounter to be adapted to the idempotence requirement described in Subsection 4.5.1: the number of times a given task is processed should not impact our counter behaviour. This additional condition requires that tasks should be given an id (to fix ideas, the number of tasks is set to T and task ids are set in the range 0 to $T - 1$), and that the counter should own the list of the tasks already completed at least once (and therefore for which the counter has already been decremented).

Let us now describe the actual implementation of our BitTreeCounter. It is designed so a single method can be used externally: it is the decrement method that takes in input the task id that is completed, and returns a boolean to tell if all the tasks have been completed and therefore if the synchronization barrier can be lifted. Internally, the BitTreeCounter is implemented as a tree of nodes, for which each node is stored inside the BlobStorage in a distinct entity. The topology of the BitTreeCounter is organized as follows: each node, with the exception of the leaves, has exactly C children. The tree depth H is set such as $H = \lceil \frac{log(T)}{log(C)} \rceil$.

At depth h (with $1 \leq h \leq H$), our BitTreeCounter has therefore C^{h-1} nodes, referred to in the following as the nodes $\{N(p,h)\}_{p=0}^{C^{h-1}-1}$. Figure 4.2 presents the BitTreeCounter topology when $C = 3$ and $H = 3$.

Each node holds a bit array of length C: for $1 \leq h \leq H$ and $0 \leq p \leq C^{h-1} - 1$, the bits of node $N(p,h)$ are referred to as $\{b(i,p,h)\}_{i=0}^{C-1}$. There is a bijection between tasks ids (from 0 to $T-1$) and all the bits of the leaves. More specifically, the task id i completion information is stored in the bit $b(i/C^{H-1}, i\%C^{H-1}, H)$ of the leave $N(i\%C^{H-1}, H)$ [5]. For a non-leave node, each bit in its bit array corresponds to the state of one of its children. More specifically, for $1 \leq h \leq H-1$, $0 \leq p \leq C^{h-1} - 1$ and $0 \leq i \leq C-1$, $b(i,p,h) = 0$ if and only if all the bits of node $N(iC+p, h+1)$ are set to 0, that is Equation (4.1).

$$b(i,p,h) = 0 \Longleftrightarrow b(u, iC+p, h+1) = 0 \; for \; 0 \leq u \leq C-1 \tag{4.1}$$

Let us describe how the BitTreeCounter is updated. At the beginning, all the bit arrays are filled with 1 values. When the task i is completed, the bit $b(i/C^{H-1}, i\%C^{H-1}, H)$ of the leave $N(i\%C^{H-1}, H)$ is set to 0. If all the bits of this leave are also equal to 0, then the leave's parent (i.e. the node $N(i\%C^{H-2}, H-1)$) is also updated: $b(i\%C^{H-1}/C^{H-2}, i\%C^{H-2}, H-1)$ is then set to 0. In the same manner, if the other bits of the node $N(i\%C^{H-2}, H-1)$ are equal to 0, then the bit $b(i\%C^{H-2}/C^{H-3}, i\%C^{H-3}, H-2)$ of node $N(i\%C^{H-3}, H-2)$ is also updated, etc. When all the bits of the root node have been set to 0, then all the tasks have been completed at least once, and the synchronization barrier is lifted.

Notice that our BitTreeCounter may happen to be in an inconsistent state. Indeed, since the multiple nodes are stored in distinct entities and since they are accessed directly and not through an indirection (the "blob pointer" described in Subsection 4.5.3), a given node may be filled with 0 while its parent has failed to be updated. The choice not to resort to a blob pointer is about efficiency over consistence. In our situation, this potential inconsistent state does not result in any problematic behavior: since each task is not marked as completed before the BitTreeCounter is updated accordingly, the failure to complete the BitTreeCounter update of a leave (and if necessary of its parent and of the parent of its parent, etc.) results in a task failure; the task will be re-processed after it re-appears in the queue and the BitTreeCounter will in the end be updated to recover its correct state. Such a failure happens very infrequently.

5. Notice that two successive tasks are by design not related to the same leave. Since in most of our use cases the tasks are processed in approximatively the same order than they are created, this design prevents most of concurrent write operations.

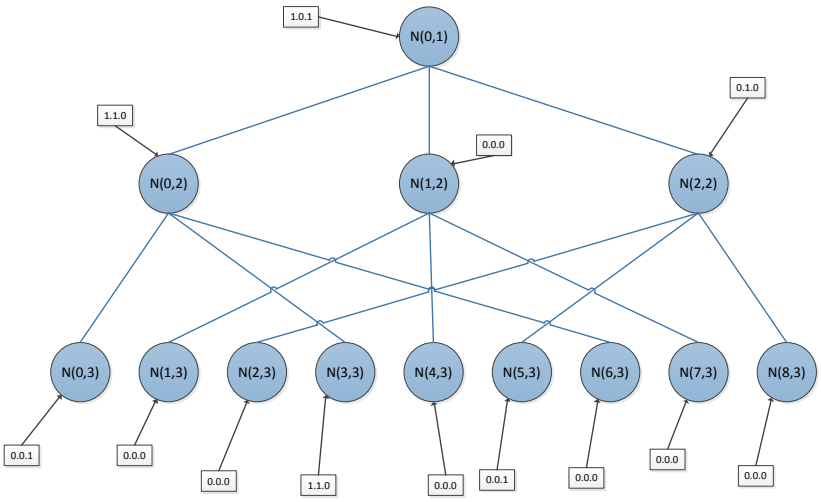

Figure 4.2: Distribution scheme of our BitTreeCounter. In the current situation, the tasks $\{3, 12, 18, 23\}$ are not yet completed. The bit arrays of the corresponding leaves ($N(0, 3)$, $N(3, 3)$ and$N(5, 3)$) are therefore not yet filled with 0. When the task 23 will be completed, the node $N(5, 3)$ will be updated accordingly. Since its bit array will then become filled with 0, the node $N(2, 2)$ will then be updated, which will in turn lead to the update of node $N(0, 1)$. Once the tasks $\{3, 12, 18, 23\}$ are completed, the root node $N(0, 1)$ will be filled with 0 and the BitTreeCounter then returns true to lift the synchronization barrier.

Chapter 5

Distributed Batch K-Means

5.1 Introduction to clustering and distributed Batch K-Means

Clustering is the task of unsupervised classification that assigns a set of objects into groups so that each group is composed of similar objects. Non-hierarchical clustering has been widely used for more than half a century as a summarizing technique to build a simplified data representation. It is one of the primary tools of unsupervised learning. It plays an outstanding role in several pattern analyses, in decision making, proximity exploration or machine-learning situations including text mining, pattern or speech recognition, web analysis and recommendation, marketing, computational biology, etc.

Two main clustering questions have been addressed in many contexts, reflecting the clustering's broad utility. The first question deals with the notion of similarity between objects. In general, pattern proximity is based on some similarity measure defined for any pair of data elements. Such a similarity measure is of prime interest since it totally defines the proximity notion and therefore needs to reflect what the user considers as close or similar. When the data set is contained in a vectorial space, a simple euclidean distance is often used. In many other situations, a more specific similarity measure is used, such as in the context of DNA microarrays exploration. In other cases, clustering output is used as the training input of a set of supervised learning functions. This latter case, often referred to as clusterwise linear regression (see for example [65]), is of prime interest for Lokad since the time series clustering is used to produce groups of time series that are processed group by group. From a computation point of view, the complexity of the similarity measure has a significant impact on the computational cost of clustering.

The second clustering aspect that has been widely addressed is the algorithm choice. For a given similarity measure, one can define a performance criterion (often referred to in the literature as a loss function) for a clustering result to quantify how close to its assigned cluster each point is. The problem of returning the optimal clustering solution in regard to this loss function is known to be in many cases computationally untractable (see for example [87] in the case of the K-Means). Many algorithms are devised to return approximations of the optimal solution, often by converging to a local minimum of the loss function. We refer the reader to [74] for an in-depth clustering algorithm review. Some well-known techniques of non-hierarchical clustering are Batch K-Means, Vector Quantization (VQ), Neural Gas, Kohonen Maps, etc.

This chapter focuses on the Batch K-Means clustering algorithm. The choice of Batch K-Means has been motivated by several reasons. Firstly, while better clustering methods are available, Batch K-Means remains a useful tool, especially in the context of data summarization where a very large data set is reduced to a smaller set of prototypes. As such, Batch K-Means is at least a standard pre-processing tool. Secondly, Batch K-Means has a low processing cost, proportional to the data size and the number of clusters. Thus, it is a good candidate for processing very large data sets. Finally, apart from the number of iterations to convergence, the processing time of Batch K-Means depends only on the data dimensions and on K, rather than on the actual data values: timing results obtained on simulated data apply to any data set with the same dimensions.

Distributed computing in machine-learning or data-mining arises when the computation time to run some sequential algorithm is too long or when the data volume is too big to fit into the memory of a single computing device. Such algorithms have already been successfully investigated for fifteen years (see for example [22], [51] or [102]) and applications built on top of these distributed algorithms are presently used in a wide range of areas, including scientific computing or simulations, web indexing applications such as Google Search, social network exploration applications such as Facebook, sparse regression in computer vision, etc. Parallelization is today one of the most promising ways to harness greater computing resources, whereas the volume of data sets keeps growing at a much faster rate than the sequential processing power. In addition, most of recent CPU are now multi-cores, implying parallelization to benefit from these supplementary resources.

In this chapter, we investigate the parallelization of Batch K-Means over different computing platforms. Batch K-Means is known to be easy to parallelize on shared

memory computers and on local clusters of workstations: numerous publications (see e.g. [51]) report linear speedup up to at least 16 processing units (which can be CPU cores or workstations). There are two reasons why the Batch K-Means algorithm is suited for parallelization. The first reason is that distributed Batch K-Means produces exactly the same result as sequential Batch K-Means. For algorithms where the sequential and the distributed versions produce different results, it is necessary to confront the two algorithm versions on both speedup and accuracy criteria. In the case of Batch K-Means, the comparison of the sequential and distributed versions is limited to the speedup criterion. In addition, the exact matching of the results of the two algorithm version provides an easy mechanism to guarantee that the distributed algorithm version is correctly implemented. From an engineering point of view, this property makes the development process much easier.

The second reason why distributed K-Means is easy to parallelize is that it has been claimed to be an embarrassingly parallel algorithm (we refer the reader to Subsection 2.5.1 for a definition of embarrassingly parallel problems). Most of the computation time is spent on evaluating distance between data points and prototypes. These evaluations can easily be distributed on multiple processing units. Distributed K-Means can then be viewed as an iterated MapReduce algorithm: firstly, the data set is split into M subsets of equal size (where M is the number of processing units), each processor (mapper) being responsible for computing distances between its data points and the prototypes. In the end, all the processors compute a version of local prototypes, then forward this version to a unique processor responsible for gathering all the versions of local prototypes versions, computing the prototypes' shared version (reduce step) and sending it back to all the units. Forwarding the versions of local prototypes can be done in several ways, depending on the hardware/software framework: broadcasting can be done using MPI (Message Passing Interface) implementation or web services for example.

Batch K-Means has already been successfully parallelized on DMM architectures using MPI (see for example [51] or [75]). This algorithm has also already been implemented on shared-nothing platforms, for example using Hadoop, but to our knowledge no theoretical study of the behavior of this algorithm on such a platform has been done yet. This chapter therefore investigates parallelization techniques of Batch K-Means over a platform of Cloud Computing (in the present case Azure). The main difficulty of the cloud algorithm consists in implementing synchronization and communication between the processing units, using the facilities provided by Windows Azure cloud operating system. We detail this technical challenge, provide theoretical analyses of speedup that can be achieved

on such a platform and the corresponding experimental results.

Let us now briefly outline the chapter. In Section 5.2, we present the sequential Batch K-Means algorithm and its computational cost. In Section 5.3, we describe how the Batch K-Means is often distributed on several processors and examine the new corresponding computational cost. We also build some model of the real cost of a distributed Batch K-Means on DMM and develop a bandwidth condition inequality. Section 5.4 is devoted to some specificities of the cloud that prevent previous developments of DMM Batch K-Means cost from being applied on the cloud. We provide a new parallelization implementation to adapt the distributed Batch K-Means to the cloud specificities. Section 5.5 presents the experimental results of our distributed Batch K-Means on the cloud.

5.2 Sequential K-Means

5.2.1 Batch K-Means algorithm

Let us consider the following problem: given a data set of N points $\{\mathbf{z}_t\}_{t=1}^N$ of a d dimensional space, we want to construct K points $\{w_k\}_{k=1}^K$, referred to in the following as prototypes or centroids, as a summary of the data set using the euclidean distance as a similarity measure. Through this similarity measure, one can define the empirical distortion:

$$C_N(w) = \sum_{t=1}^{N} \min_{\ell=1,\dots,K} \|\mathbf{z}_t - w_\ell\|^2, \quad w \in \left(\mathbb{R}^d\right)^K.$$

C_N is guaranteed to have at least one minimizer, as it is both continuous and coercive. As already stated, the computation of an exact minimizer is often untractable, and this problem is proved to be NP-Hard, even in the easier case of $d = 2$ (see [87]). Among the many algorithms that compute an approximation of the optimal minimizer, the Batch K-Means is a well known algorithm, widely used because it is easy to implement and provides overall satisfactory results.

The Batch K-Means algorithm belongs to the class of alternating optimization algorithms. Indeed, it alternates two optimization phases iteratively. The first phase, called the assignment phase, takes as input a given set of prototypes, and assigns each point in the data set to its nearest prototype. The second phase, referred to as the recalculation phase, is run after the assignment phase has been completed. During this second phase, each prototype is recomputed as the average of all the points in the data set that has been assigned to him. Once the second phase has been completed, phase 1 is run again, and phase 1 and 2 are run iteratively until a

stopping criterion is met. Algorithm 1 logical code describes the alternation of the two phases.

Algorithm 1 Sequential Batch K-Means

Select K initial prototypes $(w_k)_{k=1}^K$
repeat
  for $t = 1$ to N **do**
    for $k = 1$ to K **do**
      compute $||\mathbf{z}_t - w_k||_2^2$
    end for
    find the closest centroid $w_{k^*(t)}$ from $\mathbf{z}_t$;
  end for
  for $k = 1$ to K **do**
    $w_k = \frac{1}{\#\{t,k^*(t)=k\}} \sum_{\{t,k^*(t)=k\}} \mathbf{z}_t$
  end for
until the stopping criterion is met

Batch K-Means is an algorithm that produces, by construction, improved prototypes (in regards to the objective function (6.1)) for each iteration and that stabilizes on a local minimum of this objective function. In many cases, the prototypes w and the corresponding empirical loss $C_N(w)$ are deeply modified in the first iterations of the Batch K-Means then they move much more slowly in the latter iterations, and after several dozens iterations the prototypes and the corresponding empirical loss are totally fixed. Yet, such a behavior is not systematic and Batch K-Means may need more iterations before stabilizing (see e.g. [26]). The classical stopping criteria are: wait until the algorithm is completed (prototypes remain unmodified between two consecutive iterations), or run an a-priori fixed number of iterations or run the algorithm until the empirical loss gain between two iterations is below a given threshold.

Batch K-Means can be rewritten as a gradient-descent algorithm (see [32]). As one can notice in many other gradient-descent algorithms, Batch K-Means is very sensitive to initialization. More specifically, both the time to convergence and the quality of the clustering are strongly impacted by the prototypes initialization. We refer the reader to [35] for an in-depth review of different K-Means initialization techniques. In the following, the K prototypes are initialized with the values of the first K points of our data set and in the case of a distributed algorithm, all the computing units will be initialized with the same prototypes.

5.2.2 Complexity cost

The Batch K-Means cost per iteration does not depend on the actual data values but only on the data size. It is therefore possible to provide a precise cost for a Batch K-Means for a given data size. This cost has already been studied by Dhillon and Modha in [51]. Let us briefly list the different operations required to complete the algorithm. For a given number of iterations I, Batch K-Means requires: $I(K+N)d+IKd$ read operations, $IKNd$ subtractions, $IKNd$ square operations, $IKN(d-1)+I(N-K)d$ additions, IKd divisions, $2IN+IKd$ write operations, IKd double comparisons and an enumeration of K sets whose cumulated size is N.

If the data set is small enough to fit into the RAM memory and if specific care is made to avoid most of cache miss (see for example [52]), the read and write operation costs can be neglected for the sequential Batch K-Means version. Making the reasonable approximation that additions, subtractions and square operations are all made in a single CPU clock cycle and that $N >> K$ and $N >> d$, one can model the time to run I iterations of Batch K-Means by:

$$SequentialBatch^{Walltime} = (3KNd + KN + Nd)IT^{flop},$$

where T^{flop} denotes the time for a floating point operation to be evaluated.

5.3 Distributed K-Means Algorithm on SMP and DMM architectures

5.3.1 Distribution scheme

As already observed in [51], the Batch K-Means algorithm is inherently data-parallel: the assignment phase which is the CPU-intensive phase of the sequential algorithm version, consists in the same computation (distance calculations) applied on all the points of the data set. The distance calculations are intrinsically parallel, both over the data points and the prototypes. It is therefore natural to split the computational load by allocating disjoint subsets of points to the different processing units. This property makes the Batch K-Means algorithm suitable for many distributed architectures: the assignment phase is shortened by distributing point assignments tasks over the different processing units. Overall, the distributed Batch K-Means wall time —i.e. the human perception of the passage of time from the start to the completion of the algorithm, as opposed to the CPU time— is reduced compared to its sequential counterpart because of the

shortening of this assignment phase duration.

Let us assume that we own M computing units and that we want to run a Batch K-Means over a data set composed of N data points $\{\mathbf{z}_t\}_{t=1}^{N}$. The initial data set is split into M parts of homogeneous size $S^i = \{\mathbf{z}_t^i\}_{t=1}^{n}$ for $1 \leq i \leq M$ with $n = N/M$. The computing units are assigned an Id from 1 to M and the computing unit m processes the data set S^m. Each computing unit is given the same computation load, and the different processors complete their respective tasks in similar amount of time. Once all the computing units have completed their task, one or multiple units can proceed to the recalculation phase, before the reassignment phase is run again. We detail this distributed algorithm version in the following logical code (Algorithm 2).

As previously stated, this distributed algorithm produces after each iteration the very same result as the sequential Batch K-Means. The distributed Batch K-Means is therefore only evaluated on a speedup criterion: how much does this distributed algorithm version reduce the total time of execution? In many implementations, the overall time of execution is significantly reduced by the parallelization of the assignments and slightly increased by the communication between the different processors that is induced by the parallelization. We can therefore model the total time of execution by the equation (5.1).

$$Distributed\ Batch^{WallTime} = T_M^{comp} + T_M^{comm}, \tag{5.1}$$

where T_M^{comp} refers to the wall time of the assignment phase and T_M^{comm} refers to the wall time of the recalculation phase (mostly spent in communications).

The wall time of the assignment phase (T_M^{comp}) does not depend on the computing platform. This wall time roughly equals the assignment phase time of the sequential algorithm divided by the number of processing units M. Indeed, for this phase the translation of the algorithm toward the distributed version does neither introduce nor remove any computation cost. It only distributes this computation load on M different processing units. Therefore, this wall time is modeled by the following equation (5.2).

$$T_M^{comp} = \frac{(3KNd + KN + Nd)IT^{flop}}{M}. \tag{5.2}$$

As previously stated, the wall time of the reassignment phase is mostly spent in communications. During this phase, the different M prototypes versions computed by the M processing units are merged together to produce an aggregated

Algorithm 2 Distributed Batch K-Means on SMP architectures

Code run by the computing unit m
Get same initial prototypes $(w_k)_{k=1}^K$ as other units
repeat
  for $t = 1$ to n **do**
    for $k = 1$ to K **do**
      compute $||z_t^m - w_k||_2^2$
    end for
    find the closest prototype $w_{k^*(t,m)}$ to $\mathbf{z}_t^m$
  end for
  for $k = 1$ to K **do**
    Set $p_k^m = \#\{t,\ \mathbf{z}_t^m \in S^m\ \&\ k^*(t,m) = k\}$
  end for
  for $k = 1$ to K **do**
    $$w_k^m = \frac{1}{p_k^m} \sum_{\{t,\ \mathbf{z}_t^m \in S^m\ \&\ k^*(t,m)=k\}} \mathbf{z}_t^m$$
  end for
  Wait for other processors to finish the for loops
  if i==0 **then**
    for $k = 1$ to K **do**
      $$\text{Set } w_k = \frac{1}{\sum_{m=1}^{M} p_k^m} \sum_{m=1}^{M} p_k^m w_k^m$$
    end for
    Write w into the shared memory so it is available for the other processing units
  end if
until the stopping criterion is met

prototypes version (which is exactly the prototypes version that would have been produced by the sequential algorithm), and then this prototypes version (referred to as shared version in the following) is made available to each computing unit before the reassignment phase is restarted. In the following subsections, we investigate how the shared prototypes version is made available depending on the hardware architecture and how this communication process impacts the wall time of the recalculation phase on Symmetric Multi-Processors (SMP) and DMM architectures.

5.3.2 Communication costs in SMP architectures

SMP refers to a multiprocessor computer architecture where several identical processors are connected to a shared memory and controlled by a single OS instance. A typical SMP configuration is a multi-core processor, where each core is treated as a separate processor. In SMP architectures, the different processing units, the main memory and the hard-disks are connected through dedicated hardware such as buses, switches, etc. In this context, the communication is very fast.

In such an architecture, the communication costs of the recalculation phase are therefore very small compared to the processing costs of the reassignment phase. This brings us to neglect the communication costs in the case of a SMP architecture ($T_M^{comm,SMP} = 0$). One can then simplify the Batch K-Means cost on SMP architecture by the simplified equation (5.3).

$$Distributed\ Batch_{SMP}^{WallTime} = T_M^{comp} = \frac{(3KNd + KN + Nd)IT^{flop}}{M}. \quad (5.3)$$

This model of Batch K-Means on SMP architectures provides a perfect theoretical model where the neglect of SMP communication costs leads to a theoretical perfect speedup of a factor M for M processing units.

5.3.3 Communication costs in DMM architectures

In contrast with SMP architectures, a DMM system refers to a multi-processor in which each processor has a private memory and no shared memory is available. In such a system, the prototypes version computed by the processing unit i is accessible by the processing unit j (with $i \neq j$), if and only if the processing unit i explicitly sends its result to the processing unit j. Since all the communications in DMM architectures need to be explicit, several frameworks have been devised

to provide communication and synchronization primitives. These frameworks disburden the application developer from specific communication handling by providing a higher level communication layer. Among such frameworks, the most well-known ones are Message Passing Interface (MPI) (see e.g. [103]) or PVM (see e.g. [104]).

The wall time of the synchronous distributed Batch K-Means algorithm on DMM has been studied by Dhillon and Modha in [51] then by Joshi in [75]. In both cases, Batch K-Means is distributed on DMM architecture using a MPI framework. In [51], this wall time is modeled by the following equation (5.4).

$$Distributed\ Batch_{DMM}^{WallTime} = T_M^{comp} + T_M^{comm,DMM} \tag{5.4}$$

$$= \frac{(3KNd + KN + Nd)IT^{flop}}{M} + O(dKIT_M^{reduce}), \tag{5.5}$$

where T_M^{reduce}, following the notation of [51], denotes the time required to perform a sum or an average operation of M doubles distributed on M processing units.

To determine the wall time of the recalculation phase on DMM architectures using MPI, one therefore needs to determine the time to perform a sum or an average operation distributed on M processing units, referred to as T_M^{reduce}. This quantity is determined by the design of MPI: the MPI framework can perform communication and can broadcast data between the processing units using a tree-like topology that is described in the following subsection (5.3.4). For such a tree-like topology, the communication and broadcasting primitives are reported to be performed in $O(log(M))$. As a consequence, T_M^{reduce} is also reported to be performed in $O(log(M))$ (see e.g. [70]).

In many cases, the hardware architecture provides enough bandwidth for MPI to be very efficient. In the experiments made in [51], the communication costs are very acceptable and have little impact on the distributed Batch K-Means speedup (see Subsection 5.3.7). Yet, the communication latency and the bandwidth between processing units should not be neglected in many other cases. The following subsections provide a more detailed evaluation of T_M^{reduce} and compute some bandwidth condition to prevent communications from becoming a bottleneck of the algorithm.

5.3.4 Modeling of real communication costs

A lot of work has been done to provide precise model of the MPI primitives performances over the last two decades. In [70], numerical values of many MPI execution performances are provided. As outlined in [62], achieving such numerical measurements is a very difficult task that highly depends on the underlying hardware topology, on the synchronization methods, on the network congestion, on the different MPI implementations, etc. In this subsection we do not aim to provide accurate performance evaluation of the MPI behavior in our clustering context but rather to explicit qualitative patterns of the constraints brought by communications.

Let us consider the communication scheme described in Figure 5.1. This overly-simplified communication scheme highlights the tree-like nature of the communication patterns adopted by many MPI implementations. More specifically, the figure highlights the logarithmic scaling of the communication mechanism: to sum or average values across computing units, many MPI implementations have developed a structure where data chunks are sent in multiple steps. Each step of this communication pattern consists in two actions: a processing unit sends its data to a second one, the receiving unit merges the two data chunks into a data chunk of the same size. After each step, the number of data chunks that are sent in parallel is divided by 2. Such a pattern induces $\lceil log_2(M) \rceil$ communication steps.

In the following equations, we neglect communication latencies as well as the time to merge the data chunk (which is small in comparison with the data communication between the two processing units) and model communication costs as the ratio of the quantity of data to be sent divided by the bandwidth. We therefore provide a theoretic model of $T_M^{comm,DMM}$ by:

$$T_M^{comm,DMM} = \sum_{m=1}^{\lceil log_2(M) \rceil} \frac{IKdS}{B_{\frac{M}{2^m}}^{DMM,MPI}},$$

where S refers to the size of a double in memory (8 bytes in the following) and $B_x^{DMM,MPI}$ refers to the communication bandwidth per machine while x processing units are communicating at the same time. The number of processing units that are communicating at the same time x has a strong impact on $B_x^{DMM,MPI}$ because of a phenomenon referred to as aggregated bandwidth bounds.

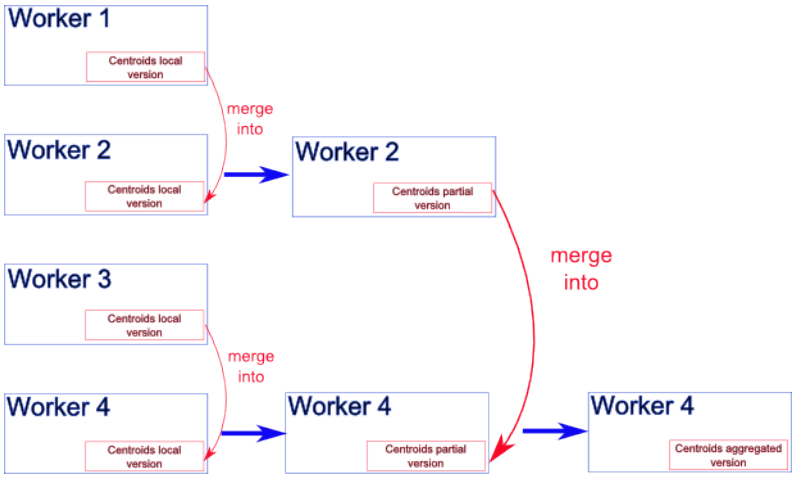

Figure 5.1: This non-realistic scheme of the merging prototype logic highlights the tree structure of many MPI primitives. To begin with, the first worker sends a data chunk to the second worker, while the third worker sends simultaneously its data chunk to the fourth worker. A second step follows the first one in which the merging result of the first and second workers are sent to the fourth worker that already owns the merging result of the third and fourth workers. In two steps, the data chunks of the four workers are merged.

The aggregated bandwidth refers to the maximum number of bytes the actual network can transfer per time unit. We refer the reader to Subsection 5.5.2 for a more precise explanation. As regards our model, let us make the strong simplification that the bandwidth per worker only depends on the hardware context and on the total number of processing units, and not on the actual number of processing units communicating at the same time. We therefore note this bandwidth $B_M^{DMM,MPI}$.

The time spent in communication is then written:

$$\begin{aligned} T_M^{comm} &= \sum_{m=1}^{\lceil log_2(M) \rceil} \left(\frac{IKdS}{B_M^{DMM,MPI}} \right) \\ &= \lceil log_2(M) \rceil \frac{IKdS}{B_M^{DMM,MPI}}. \end{aligned}$$

We can then deduce an estimation of the speedup rate on DMM architectures more detailed than in [51], where T_M^{comm} is specified :

$$Speedup(M,N) = \frac{SequentialBatch^{Walltime}}{Distributed\ Batch^{WallTime}_{DMM}} \tag{5.6}$$

$$= \frac{T_1^{comp}}{T_M^{comp} + T_M^{comm}} \tag{5.7}$$

$$= \frac{3NKdIT^{flop}}{\frac{3NKdIT^{flop}}{M} + \frac{IKdS}{B_M^{DMM,MPI}}\lceil log_2(M)\rceil} \tag{5.8}$$

$$= \frac{3NT^{flop}}{\frac{3NT^{flop}}{M} + \frac{S}{B_M^{DMM,MPI}}\lceil log_2(M)\rceil}. \tag{5.9}$$

Using the previous speedup model, one can deduce the optimal number of processing units for distributed Batch K-Means implemented on DMM architectures:

$$M^*_{DMM} = \frac{3NT^{flop}B_M^{DMM,MPI}}{S}. \tag{5.10}$$

5.3.5 Comments

The speedup model provided in the previous Subsection 5.3.4 is too simple to provide a satisfactory forecasting tool to anticipate with enough precision the actual speedup provided by a distributed Batch K-Means implementation on a given architecture. Yet, this naive model allows us to draw some qualitative conclusions shared by many distributed Batch K-Means implementations on shared-nothing architectures:

– The speedup depends neither on the number of prototypes (K), nor on the data dimension (d), neither does it depend on the data distribution. It only depends on the number of points in the data set (N) and on architecture characteristics such as bandwidth or CPU frequency.

– The speedup is a monotonically increasing function of N. In particular, the previous equation leads to $\lim_{N\to+\infty} SpeedUp(M,N) = M$. From this theoretical result, a conclusion can be drawn: the more loaded the RAM of workers is, the more efficient the workers are. Provided the processing units are given enough RAM, it is possible to get an efficiency per worker arbitrarily close to 100%.

– The actual speedup of the distributed implementation might even be higher than the theoretical speedup model since we have not taken into account the

fact that the RAM is bounded. Indeed, the accumulated RAM from each of the machines may enable the data set to move from disk into RAM thereby drastically reducing the time access for read and write operations. This phenomenon is often referred to as superlinear speedup.

– The previous equation shows that speedup is a monotonically decreasing function of T^{flop} and a monotonically increasing function of $B_M^{DMM,MPI}$. This result shows that the parallelization is best suited on slow machines with high communication bandwidth.

– Because of the actual values of bandwidths and CPU frequencies in present real world DMM architectures, it is always efficient to use all the available processing units of a DMM architecture to perform a distributed Batch K-Means.

5.3.6 Bandwidth Condition

In this subsection we briefly determine some theoretical conditions that guarantee that the recalculation phase is small in comparison with the reassignment phase. This condition turns into:

$$T_M^{comm} << T_M^{comp}.$$

Such a condition translates into:

$$\frac{IKdS}{B_M^{DMM,MPI}}\lceil log_2(M)\rceil << \frac{(3NKd + NK + Nd)IT^{flop}}{M}. \tag{5.11}$$

A sufficient condition for equation (5.11) to be verified is that the following equation (5.12) is verified:

$$\frac{IKdS}{B_M^{DMM,MPI}}\lceil log_2(M)\rceil << \frac{3NKdIT^{flop}}{M}. \tag{5.12}$$

Finally we get the following condition:

$$\frac{N}{M\lceil log_2(M)\rceil} >> \frac{S}{3T^{flop}B_M^{DMM,MPI}}.$$

For example, in the context of a DMM architecture with 50 computing units composed of a single mono-core retail processor (2Ghz) and a network interface controller with a 1 Gbit/sec bandwidth, the condition turns into:

$$N >> 12,000.$$

Assuming that the dimension $d = 1000$, it is therefore necessary to have a data set of more than 10 GBytes (i.e. 200 MBytes per machine) on such an architecture, to ensure that communications will cost less than 1% of the computation duration.

5.3.7 Dhillon and Modha case study

The experiments of [51] have been run on an IBM SP2 platform with a maximum of 16 nodes. Each node is an IBM POWER2 processor running at 160MHz with 256 MBytes of main memory. Processors communicate through the High-Performance Switch (HPS) with HPS-2 adapters. Performance of the HPS has been discussed in [117] and [20]. In [20] for example, point-to-point bandwidth in a SP2 with HPS and HPS-2 adapters using MPI is reported to be about 35 MBytes/sec.

It is difficult to estimate the bandwidth actually obtained during the experiments of [51], as it is also difficult to estimate effective Flops. In [51], no comments are made about bandwidth, yet Flops are reported to be very fluctuent: 1.2 GFlops in some experiments, 1.8 GFlops in others (the maximum would be 16 * 160MFlops = 2,5GFlops). Yet, on smaller data sets, Flops might be very much lower than the reported numbers. Since we cannot guess the actual bandwidth and Flops during these experiments, theoretical bandwidth and Flops are used in the following to provide approximative results.

Dhillon and Modha report that they have obtained a speedup factor of 6.22 on 16 processors when using 2^{11} points. Using condition (5.6), we can estimate speedup when using 2^{11} points :

$$\begin{aligned} EstimatedSpeedup &= \frac{3NT^{flop}}{\frac{3NT^{flop}}{M} + \left(2\frac{S}{B_M^{DMM,MPI}} + 5T^{flop}\right)\lceil log_2(M)\rceil} \\ &\simeq 11. \end{aligned}$$

We can see the estimated speedup is not accurate, but we are in the range where communication is important enough to prevent a perfect speedup and small enough though for a significant speedup to be observed.

When using 2^{21} points, we get :

$$EstimatedSpeedup = \frac{3NT^{flop}}{\frac{3NT^{flop}}{M} + \left(2\frac{S}{B_M^{DMM,MPI}} + 5T^{flop}\right)\lceil log_2(M)\rceil} \simeq 15.996.$$

[51] reports that for 2^{21} points, a 15.62 speedup is observed. Again, anticipated speedup indicates there will be little issue to parallelize, and observed speedup is indeed excellent.

5.4 Implementing Distributed Batch K-Means on Azure

5.4.1 Recall of some Azure specificities

The architecture of most Azure hosted applications is based on two components: *web roles* and *worker roles*. Web roles are designed for web application programming. In the context of our cloud-distributed Batch K-Means prototype, a single web role is used for monitoring the algorithm behaviour, observing the queues load, noticing the BlobStorage short outages or unavailability, profiling our algorithm and redeploying quickly our application using the Appdomain trick (see Subsection 3.4.2 and [2]). Worker roles are designed to run general background processing. Each worker role typically gathers several *cloud services* and uses many *workers* (Azure's processing units) to execute them. Our cloud-distributed Batch K-Means prototype uses only one worker role, several services and tens or hundreds of workers. This worker role is used to run the actual Batch K-Means algorithm.

The computing power is provided by the workers, while the Azure storage system is used to implement synchronization and communication between workers. It must be noted indeed that Azure does not currently offer any standard API for distributed computation, neither a low-level one such as MPI, nor a higher-level one such as MapReduce ([47]) or Dryad ([73]). MapReduce could be implemented using Azure components (following the strategy of [82]), yet, as pointed out in e.g. [85], those high-level API might be inappropriate for iterative machine-learning algorithms such as Batch K-Means. Therefore we rely directly on the Azure queues and the BlobStorage.

Our prototype uses *Lokad-Cloud* [1] (see Subsection 4.5.4), an open-source framework that adds a small abstraction layer to ease Azure workers startup and life cycle management, and storage access.

5.4.2 The cloud Batch K-Means algorithm

As already outlined, the distributed Batch K-Means can be reformulated as an iterated MapReduce where the assignment phases are run by the mappers and the recalculation phases are performed by the reducers. The design of our distributed cloud Batch K-Means is therefore vastly inspired by the MapReduce abstraction and follows the considerations introduced in Chapter 4. This section presents our cloud implementation. Figure 5.2 provides a scheme of the cloud implementation of our distributed Batch K-Means. Algorithms 3 (resp. 4 and 5) reproduce the logical code run by the mappers (resp. the partial reducers and the final reducer).

Following the MapReduce terminology, we split our algorithm into three cloud services (setup, map and reduce services), each one matching a specific need. A queue is associated to each service; it stores messages specifying the storage location of the data needed for the tasks. The processing units regularly ping the queues to acquire a message. Once it has acquired a message, a worker starts running the corresponding service, and the message becomes invisible till the task is completed or timeouts. Overall, we use $M + \sqrt{M} + 1$ processing units in the services described below (we suppose in the following that $\sqrt{M}$ is an integer).

The *SetUpService* generates M split data sets of $n = N/M$ points in each and puts them into the BlobStorage. It is also generating the original shared prototypes which are also stored in the BlobStorage. Once the processing units in charge of the set-up have completed the data generation, they push M messages in the queue corresponding to the "Map Service". Each message contains a taskId (from 1 to M) related to a split data set to be processed and a groupId (from 1 to $\sqrt{M}$), which is described above. The same processing unit also pushes $\sqrt{M}$ messages in the queue corresponding to the "Reduce Service". In the current implementation [2], the SetUp service is executed by M processing units to speedup the generation process.

Once the set-up has been completed, the Map queue is filled with M messages (corresponding to the M map tasks) and the Reduce queue is filled with $\sqrt{M}$

1. `http://code.google.com/p/lokad-cloud/`
2. available at http://code.google.com/p/clouddalvq/

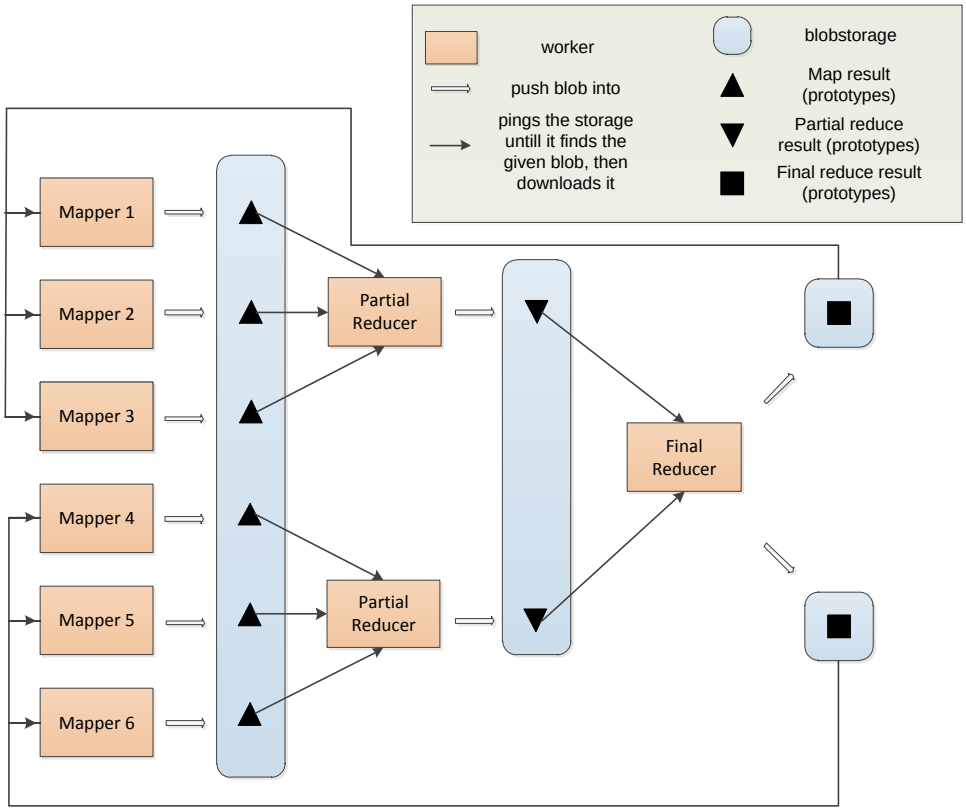

Figure 5.2: Distribution scheme of our cloud-distributed Batch K-Means. The communications between workers are conveyed through the BlobStorage. The recalculation phase is a two-step process run by the partial reducers and the final reducer to reduce I/O contention.

messages (corresponding to the $\sqrt{M}$ reduce tasks). Each processing unit pings the different queues to acquire a Map task or a Reduce task.

When executing a *MapService* task, a processing unit is referred to as a mapper. Each mapper first downloads the corresponding partial data set it is in charge of (once for all). Then the mapper loads the initial shared prototypes and starts the distance computations that form the assignment phase. Once the assignment phase has been completed, the mapper builds a local version of the prototypes according to the locally run assignment phase. This prototypes version is sent to the BlobStorage, and the mapper waits for the Reduce service to produce a shared version of the prototypes. When the shared version of the prototypes is made available, the mapper downloads it from the storage and restarts the assignment phase using the new prototypes version thus obtained. The instructions performed

Algorithm 3 Distributed Cloud Batch K-Means : Mapper

Dequeue a message from the Map Queue.
Get taskId, groupId and IterationMax from the message.
Set m=taskId
Retrieve the partial data set $S^m = \{\mathbf{z}_i^m\}_{i=1}^N$ from the storage
Retrieve the initial prototypes shared version $\{w_k^{srd}\}_{k=1}^K$
Initialize w^m as $w^m = w^{srd}$
for It=0; It < IterationMax ; It++ **do**
  for $\mathbf{z}_i^m \in S^m$ **do**
    for $k = 1$ to K **do**
      Compute $||\mathbf{z}_i^m - w_k||_2^2$
    end for
    Find the closest prototype $w_{k^*(i,m)}$ to $\mathbf{z}_i^m$
  end for
  for $k = 1$ to K **do**
    Set $p_k^m = \#\{t,\ \mathbf{z}_t^m \in S^m\ \&\ k^*(t,m) = k\}$
  end for
  for $k = 1$ to K **do**
    Set $w_k^m = \dfrac{1}{p_k^m} \displaystyle\sum_{\{t,\ \mathbf{z}_t^m \in S^m\ \&\ k^*(t,m)=k\}} \mathbf{z}_t^m$
  end for
  Send w^m into the storage in a location depending on the iteration It
  Send p^m into the storage in a location depending on the iteration It
  Ping the storage every second to check if w^{srd} is available for iteration It
  Download it when it becomes available
  Replace w^m by the new downloaded shared version
end for

by the mappers are detailed in Algorithm 3.

When executing a *ReduceService* task, a processing unit is referred to as a partial reducer. Each partial reducer downloads from the storage multiple prototypes versions that come from different Map tasks, and merges them into an average prototypes version. More specifically, each Reduce task consists in merging $\sqrt{M}$ prototypes versions. The Reduce task message holds an Id called groupId that refers to the group of prototypes versions that this task needs to collect. When the $\sqrt{M}$ prototypes versions it is in charge of are retrieved, the partial reducer merges the prototypes versions using weighted averages and pushes the merged result into the storage. Once all the $\sqrt{M}$ Reduce tasks have been completed, a last reducer, referred to as the final reducer, downloads the $\sqrt{M}$ merged results thus

Algorithm 4 Distributed Cloud Batch K-Means : Partial Reducer

Dequeue a message from the Partial Reduce Queue.
Get groupId and IterationMax from the message.
Set g= groudId
for It=0; It < IterationMax ; It++ **do**
  Retrieve the prototypes version $\{w^m\}_{m=g\sqrt{M}}^{(g+1)\sqrt{M}}$ corresponding to iteration It
  Retrieve the corresponding weights $\{p^m\}_{m=g\sqrt{M}}^{(g+1)\sqrt{M}}$ corresponding to iteration It
  for $k = 1$ to K **do**
    Set $p_k^g = \sum_{m=g\sqrt{M}}^{(g+1)\sqrt{M}} p_k^m$
    Set $w_k^g = \frac{1}{p_k^g} \sum_{m=g\sqrt{M}}^{(g+1)\sqrt{M}} p_k^m w_k^m$
  end for
  Send w^g into the storage in a location depending on the iteration It
  Send p^g into the storage in a location depending on the iteration It
end for

Algorithm 5 Distributed Cloud Batch K-Means : Final Reducer

Dequeue the single message from the Final Reduce Queue.
Get IterationMax from the message.
for It=0; It < IterationMax ; It++ **do**
  Retrieve the prototypes version $\{w^g\}_{g=1..\sqrt{M}}$ corresponding to iteration It
  Retrieve the corresponding weights $\{p^g\}_{g=1..\sqrt{M}}$ corresponding to iteration It
  for $k = 1$ to K **do**
    Set $p_k^{srd} = \sum_{g=1}^{\sqrt{M}} p_k^g$
    Set $w_k^{srd} = \frac{1}{p_k^{srd}} \sum_{g=1}^{\sqrt{M}} p_k^g w_k^g$
  end for
  Send w^{srd} into the storage in a location depending on the iteration It
end for

produced and merges all of them into a single prototypes version called shared version. This shared version is pushed into the storage to be made available for the mappers [3]. When this shared version is read by the mappers, a new iteration of the algorithm is started and the assignment phase is re-run by the mappers. The instructions performed by the partial reducers (resp. the final reducer) are detailed in Algorithm 4 (resp. Algorithm 5).

The two-step design of the ReduceService (partial reducers and the final reducer) is of paramount importance. This design ensures no processing unit needs to download more than $\sqrt{M}$ prototypes versions per iteration. In the same way, $\sqrt{M}$ copies of the shared prototypes version are made available in the storage instead of one, to ensure no blob is requested in parallel by more than $\sqrt{M}$ processing units per iteration.

For each iteration of the algorithm, there are three synchronization barriers. Firstly, each partial reducer needs to get the results of the $\sqrt{M}$ mappers it is related to before merging them. Secondly, the final reducer needs to get the results of all the $\sqrt{M}$ partial reducers before merging them into the shared prototypes version. Finally, each mapper needs to wait for the final reducer to push the shared version into the BlobStorage before restarting the reassignment phase. As outlined in Chapter 4, the synchronization primitive is one of the few primitives that is not directly provided by Azure. Let's consider two different designs to implement this synchronization process.

The first solution consists in using a counter to keep track of the number of tasks that need to be completed before the synchronization barrier can be lifted. This counter needs to be idempotent and designed so that the contention should be limited. Such a counter has been already described in the BitTreeCounter section of Chapter 4. For example, all the mappers that process tasks sharing the same groupId could own such a counter initially set to $\sqrt{M}$; once a mapper is done, it decrements the counter, and when a mapper decrements the counter from 1 to 0, it knows all the other mappers of the same groupId are also done, so that the corresponding partial reduce task can be started.

The second solution comes from the ability for each processing unit to ping a specific blob storage location to determine if there is a blob in there. Let us re-examine the previous example. Instead of using a specific counter and starting

3. More specifically, the final reducer pushes $\sqrt{M}$ copies of the shared prototypes version in the storage instead of one. Each of these copies is read by all the mappers sharing the same groupId.

the reducer only when the counter hits 0, the partial reducer could be started from the beginning, together with the mappers. This partial reducer would then regularly query the storage to detect whether the prototypes versions produced by the mappers have already been made available. When all the $\sqrt{M}$ versions have been retrieved, then the partial reducer can start the merging operation. This second solution requires that each blob put into the storage should have a corresponding blobName that is pre-defined using a fixed rule; in the case of prototypes versions made by mappers, the corresponding blobNames are built using the following addressing rule: prefix/iteration/groupId/jobId.

Both solutions provide idempotence and avoid contention. The second solution has been chosen because it helps to reduce stragglers (see Subsection 5.5.6) by overlapping the latest computations of mappers with the retrieval of first available results by the partial reducer. This second solution comes with the drawback of running more workers ($M + \sqrt{M} + 1$) than the first solution (M). This drawback is not that significant since —as this is highlighted in the following subsections— the scale-up of our algorithm will be more limited by the bandwidth/CPU power of our machines than by the actual cost of these machines.

Let us remark that the two-step design of our clustering prototype has been inspired by previous design choices that we adopted unsuccessfully. Indeed, our first design was a single-step reduce run by a single worker, with a synchronization barrier between the assignment phase and the recalculation phase in the form of a single naive blob counter. The contention on this naive blob counter, just as the overload on the single reducer motivated us to the design presented above.

5.4.3 Comments on the algorithm

In the previous proposed algorithm as well as in the DMM implementation of [51], the assignment phase of Batch K-Means is perfectly parallelized. As already outlined in the model of speedup in the case of DMM architectures, the global performances are hindered by the wall time duration of the reduction phase. In the case of a SMP architecture, the recalculation phase is approximately instantaneous and the parallelization in such architectures is very efficient. On DMM architectures, we have already shown how the MPI framework provides very efficient primitives to perform the reduction phase in amounts of time that are in most cases negligible (with a $O(log(M))$ cost). In such cases, the DMM implementations lead to a linear speedup nearly as optimal as in the SMP case, as confirmed in the large data set experiments of [51].

In the case of our Azure implementation that relies on the storage instead of on direct inter-machine communications, the recalculation phase is proportionally much longer than in the DMM case. The two-step design of our algorithm produces a recalculation phase cost of $O(\sqrt{M})$ because the partial reducers as well as the final reducer need to download $\sqrt{M}$ prototypes versions per iteration. Such a cost is asymptotically much higher than the asymptotical cost of the MPI framework ($O(log(M))$). While it would have been possible for our reduction architecture to complete in $O(log(M))$ by implementing a $log(M)$-step architecture, such a design would have led to higher recalculation costs because of frictions. Indeed, the latency between the time a task to being completed and the time the consumer of the former task manages to acknowledge that the result is available, is rather high. This phenomenon highlights a previously stated remark (see Chapter 3): the cloud platforms are probably not suited to process very fine-grained tasks.

We also draw the reader's attention on the importance of the data set location and downloading. The previous algorithm is inspired by an iterated MapReduce with the noticeable exception that no Map task corresponds to a single iteration of K-Means but to all the iterations. Since Azure does not provide any mechanism to run computations on the physical machines where the data are stored through the BlobStorage, each processing unit need to download the data set from the storage (i.e. from distant machines). To prevent each of the workers from re-loading a data set at each iteration, we have chosen that each processing unit should be processing the same data chunk for each iteration.

5.4.4 Optimizing the number of processing units

A very important aspect of our cloud implementation of Batch K-Means is the elasticity provided by the cloud. On SMP or DMM architectures, the quantity of CPU facilities is bounded by the actual hardware, and the communications are fast enough. As a result, the user has better run the algorithm on all the available hardware. The situation is significantly different for our cloud implementation. Indeed, the cloud capabilities can be redimensionned on-demand to better suit our algorithm requirements. Besides, the communication costs which are induced show that oversizing the number of workers would lead to increasing the overall algorithm wall time because of these higher communication costs. Running Batch K-Means on Cloud Computing therefore introduces a new interesting question: what is the (optimal) amount of workers that minimize our algorithm wall time?

To answer this question, let us re-tailor the speedup formula detailed in Subsection 5.3.4. As already explained in Subsection 5.4.2, the symbol M in the following will not refer to the total number of processing units but to the number of processing units that will run the Map tasks. The total amount of processing units could be lowered to this quantity, but as outlined in the same subsection, it would come at the cost of a slightly slower algorithm. As in the DMM case, the wall time of distance calculations remains unmodified:

$$T_M^{comp} = \frac{(3NKd + NK + Nd)IT^{flop}}{M},$$

where N, as in the DMM architecture case, stands for the total number of points in all the data sets gathered. In addition to the distance calculations, a second cost is introduced to model the wall time of the reassignment phase: the time to load the data set from the storage. Let us introduce T_{Blob}^{read} (resp. T_{Blob}^{write}) that refers to the time needed by a given processing unit to download (resp. upload) a blob from (resp. to) the storage per memory unit. The cost to load the data set from the storage is then modeled by the following equation:

$$T_M^{Load} = \frac{NdST_{Blob}^{read}}{M}.$$

Let us keep in mind that by design this loading operation needs to be performed only once, even when $I > 1$. This loading operation can be neglected in speedup model if $T_M^{Load} << T_M^{Comp}$. A sufficient condition for this to be true is:

$$\frac{ST_{Blob}^{read}}{3IKT^{flop}} << 1.$$

This condition turns out to be true in almost all the cases. Let us provide a new wall time model of the recalculation phase performed by the two-step reduce architecture provided in Subsection 5.4.2. This recalculation phase is composed of multiple communications between workers and storage as well as average computation of the different prototypes versions. More specifically, each iteration requires:

– M mappers to write their version "simultaneously" in the storage;
– each of the partial reducers to retrieve $\sqrt{M}$ prototypes versions;
– each of the partial reducers to compute an average of the versions thus retrieved;

– each of the partial reducers to write "simultaneously" its result in the storage;
– the final reducer to retrieve the $\sqrt{M}$ partial reducer versions;
– the final reducer to compute the shared version accordingly;
– the final reducer to write the $\sqrt{M}$ shared versions in the storage;
– all the mappers to retrieve the shared version.
The recalculation phase per iteration can therefore be modeled by the following equation:

$$\begin{aligned} T_M^{comm,periteration} &= KdST_{Blob}^{write} + \sqrt{M}KdST_{Blob}^{read} + 5(\sqrt{M})KdT^{flop} \\ &+ KdsT_{Blob}^{read} + KdST_{Blob}^{write} + \sqrt{M}KdST_{Blob}^{read} \\ &+ 5(\sqrt{M})KdT^{flop} + \sqrt{M}KdST_{Blob}^{write}. \end{aligned}$$

Keeping only the most significant terms, we get:

$$T_M^{comm} \simeq I\sqrt{M}KdS(2T_{Blob}^{read} + T_{Blob}^{write}).$$

More generally, we can show that a p-step reduce process leads to communication cost of the following form:

$$T_M^{comm,\ p-step} \simeq I\ \sqrt[1/p]{M}KdS(pT_{Blob}^{read} + (p-1)T_{Blob}^{write}).$$

In the context of our 2-step reduce process, we can deduce an approximation of the speedup factor :

$$SpeedUp = \frac{T_1^{comp}}{T_M^{comp} + T_M^{comm}} \tag{5.13}$$

$$\simeq \frac{3IKNdT^{flop}}{\frac{3IKNdT^{flop}}{M} + I\sqrt{M}KdS(2T_{Blob}^{read} + T_{Blob}^{write})} \tag{5.14}$$

$$\simeq \frac{3NT^{flop}}{\frac{3NT^{flop}}{M} + \sqrt{M}S(2T_{Blob}^{read} + T_{Blob}^{write})}. \tag{5.15}$$

Using this model, which is kept simple, one can see there is an optimal number of workers to use. This number (M^*) can be expressed as:

$$M^* = \sqrt[2/3]{\frac{6NT^{flop}}{S(2T^{read}_{Blob} + T^{write}_{Blob})}}. \tag{5.16}$$

This quantity must be compared to the optimal number of processing units in the DMM case expressed in equation (5.10). Let us remark that contrary to the DMM model, the best number of processing units in our cloud model does not scale linearly with the number of data points N. It directly follows that our cloud implementation suffers from a theoretical impossibility to provide an infinite scale-up with the two-step reduce architecture. Subsection 5.5.5 investigates how our cloud-distributed Batch K-Means actually performs in terms of practical scale-up.

One can verify that for the previous value of M^*, $T^{comp}_{M^*} = \frac{1}{2}T^{comm}_{M^*}$, which means that when running the optimal number of processing units, the reassignment phase duration is half of the recalculation phase duration. In such a situation the efficiency of our implementation is rather low: with $M^* + \sqrt{M^*} + 1$ processing units used in the algorithm, we only get a speedup of $M^*/3$.

5.5 Experimental results

5.5.1 Azure base performances

In order to use our cost model as a predictive tool to determine the optimal number of workers M^* and our implementation performance, one needs to evaluate the performances of Azure services. These performances have already been briefly analyzed in Subsection 3.4.2. We refer the reader to this subsection for more explanations and just recall here the recorded performances.

– *BlobStorage Read Bandwidth:* 8MBytes/sec
– *BlobStorage Write Bandwidth:* 3MBytes/sec
– *CPU performance while performing distance calculations:* 670 MFlops

5.5.2 The two-step reduce architecture benchmark

The aggregated bandwidth refers to the maximum number of bytes the actual network can transfer per time unit. In most cases, such a quantity does not equal the product of maximal bandwidth per processing unit by the number of processing units, since the network may not sustain such maximal bandwidths for each worker when all of them are communicating at the same time. The aggregated bandwidth measurement is a difficult task that is hardware and task dependent. In this subsection we focus on a specific case raised by our clustering algorithm.

In the case of the two-step recalculation phase implementation developed in Subsection 5.4.2, it is both difficult and inaccurate to theoretically determine the recalculation phase duration, partly because of the previously mentioned aggregated bandwidth bounds. We therefore produce a custom benchmark to evaluate the time spent in the recalculation phase as follows: the prototypes are designed as a data chunk of 8MBytes and the recalculation phase is run 10 times ($I = 10$). For the different values of M, the clustering implementation is run but the processing part is replaced by waiting for a fixed period of time (15 seconds), so that communication time can be recorded without being affected by straggler issues, as reported in Subsection 5.5.6. The following table provides the wall time of this benchmark (for 10 iterations), and the amount of time spent in the recalculation phase (Wall Time - 10*15 seconds).

M	5	10	20	30	40	50	60	70	80
Wall Time (in sec)	287	300	335	359	392	421	434	468	479
Communication (in sec)	137	150	185	209	242	271	284	318	329
$2T_{Blob}^{read} + T_{Blob}^{write}, (in\ 10^{-7} sec/Byte)$	7.64	5.92	5.16	4.76	4.78	4.78	4.59	4.73	4.58

M	90	100	110	120	130
Wall Time (in sec)	509	533	697	591	620
Communication (in sec)	359	383	547	441	470
$2T_{Blob}^{read} + T_{Blob}^{write}, (in\ 10^{-7} sec/Byte)$	4.71	4.77	6.51	5.02	5.13

Table 5.1: Evaluation of the communication throughput per machine and of the time spent in the recalculation phase for different number of communicating units M.

First of all, one can notice that the quantity $2T_{Blob}^{read} + T_{Blob}^{write}$ does not grow with M (at least for $M < 100$), which proves that we do not suffer in our experiment from aggregated bandwidth bounds before using 100 workers. Secondly, we can note that the obtained value is smaller than the value that would be obtained using the values provided in Subsection 5.5.1 (which corresponds to $5.83 * 10^{-7}$ sec/Byte): indeed, the parallelization of downloads and uploads in each machine (through multiple threads) reduces enough the communication to compensate

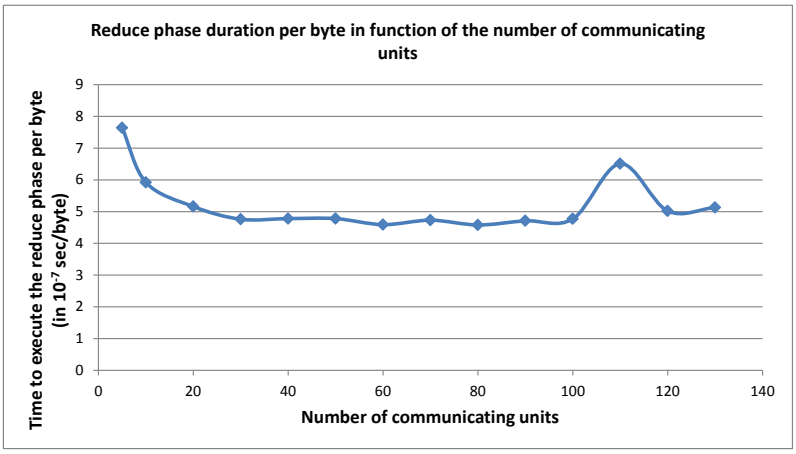

Figure 5.3: Time to execute the Reduce phase per unit of memory ($2T_{Blob}^{read}+T_{Blob}^{write}$) in 10^{-7}sec/Byte in function of the number of communicating units.

the frictions introduced by our two-step design. Finally, the whole process is sometimes behaving much worse than expected (see the outlier $M = 110$). The case $M = 110$ has been re-launched 1 hour later, obtaining the value 4.85. Figure 5.3 sums up the table and shows that aggregated bandwidth bounds are not hit before 100 hundred processing units are used. Before this threshold, the quantity $2T_{Blob}^{read} + T_{Blob}^{write}$ remains constant.

5.5.3 Experimental settings

In the following experiments, Batch K-Means are run on synthetic data. As explained in the introduction to this chapter, the Batch K-Means wall time depends on the data size but not on the actual data values, except for the number of iterations to convergence. Thus, the synthetic nature of the data has no impact on the conclusion that we draw. Besides, the synthetic nature of our data has allowed us to easily modify parameters such as the dimension d to highlight some results. The synthetic data are generated uniformly in the unit hypercube using the following settings: the dimension d is set to 1000 and the number of clusters K is set to 1000. The number of points in the total data set depends on the experiment. For speed-up experiments, the total data set is composed of $500,000$ data points (for a total size of 4 GBytes) that are evenly split among the multiple processing units. For scale-up experiments, the data set total number of points grows with the number of processing units in the scale-up experiments.

In all our experiments, the algorithm is run for 10 iterations to get stable timing estimates.

Theoretically, K and d should have the same impact on map and reduce phases since Map and Reduce costs are supposed to be proportional to Kd. As a consequence, the speedup is theoretically agnostic to K or d. Yet, since our model does not take latency into account, having very small values of K and d would lead to underestimate communication costs by neglecting latency. Provided K and d are kept big enough, our speedup/scale-up results do not depend on K or d.

As explained in the introduction to the chapter, the distributed Batch K-Means produces for each iteration the very same results as what a sequential Batch K-Means would return. Thus, the following experiments only focus on the speedup provided by the parallelization and not the function loss improvements.

5.5.4 Speedup

In this subsection we report on the performances of the cloud implementation proposed in Subsection 5.4.2 in terms of speedup. In other words, we investigate whether the proposed implementation allows us to reduce the Batch K-Means execution wall time for a given data set. In addition, we compare the observed optimal speedup to the theoretical optimal speedup obtained by the equation (5.13). We report on the results of one out of multiple experiments that we have run, the other experiments having shown the same general patterns and qualitative conclusions.

The proposed implementation is tested using the settings described in Subsection 5.5.3. The algorithm is run for 10 iterations to get stable timing estimates. Neglecting loading time and memory issues (a small instance has only 1.75 GBytes of memory), a sequential Batch K-Means implementation would use approximately 6 hours and 13 minutes to run the 10 iterations.

The following table reports on the total running time in seconds (including data loading) of the proposed implementation for different numbers of mappers (M). We also report on the speedup over the theoretical total running time, and the efficiency (speedup divided by the total number of processing units $M + \sqrt{M} + 1$). The speedup as a function of M is plotted in Figure 5.4 (it is the curve with $N = 500,000$).

M	10	50	60	70	80	90	95	100	110
Time	2223	657	574	551	560	525	521	539	544
SpeedUp	10.0	34.1	39.0	40.6	40.0	42.6	43.0	41.5	41.2
Efficiency	0.67	0.58	0.57	0.51	0.44	0.42	0.41	0.37	0.34

M	120	130	140	150	160
Time	544	574	603	605	674
SpeedUp	41.2	39.0	37.1	37.0	33.2
Efficiency	0.31	0.27	0.24	0.23	0.19

Table 5.2: Evaluation of our distributed K-Means speedup and efficiency for different number of processing units M.

As expected, the total processing time is minimal for a specific value of M referred to as M^*_{eff} (here 95), for which the speedup (43.0) is comparable to $M^*_{eff}/3$ (as predicted by the model). With the values of T^{flop} reported in Subsection 5.5.1 and the values of $2T^{read}_{Blob} + T^{write}_{Blob}$ evaluated in Subsection 5.5.2, the theoretical value of M^* is 112.While the theoretical optimal number of workers M^* slightly overestimates the actual optimal number of workers M^*_{eff}, the equation (5.16) provides a good first estimate of this number before running the experiments. More specifically, once the values $2T^{read}_{Blob} + T^{write}_{Blob}$ and T^{flop} have been evaluated, our equations provide an a-priori tool to estimate what the optimal number of machines to use would be.

5.5.5 Optimal number of processing units and scale-up

Let us bear in mind that the scale-up is the ability to cope with more data in the same amount of time, provided that the number of processing units is increased accordingly. In our cloud Batch K-Means, the theoretical optimal number of mappers M^* is not proportional to N (see equation (5.16)), contrary to the DM-M/MPI model cost. As a consequence, our cloud version cannot hope to achieve linear scale-up.

As a consequence, the scale-up challenge is turned into minimizing growth of wall time as N grows, using $M^*(N)$ mappers. Theoretically, our model gives a processing cost (T^{comp}_M) proportional to N/M and a communication cost (T^{comm}_M) proportional to $\sqrt{M}$. As a consequence, the algorithm execution total time ($T^{comp}_M + T^{comm}_M$) is proportional to $\sqrt[1/3]{N}$ and the optimal number of workers M^* is proportional to $\sqrt[2/3]{N}$.

In the following experiment, the values of K and d are kept constant ($K = 1000$ and $d = 1000$). For various values of N, our implementation is run on different

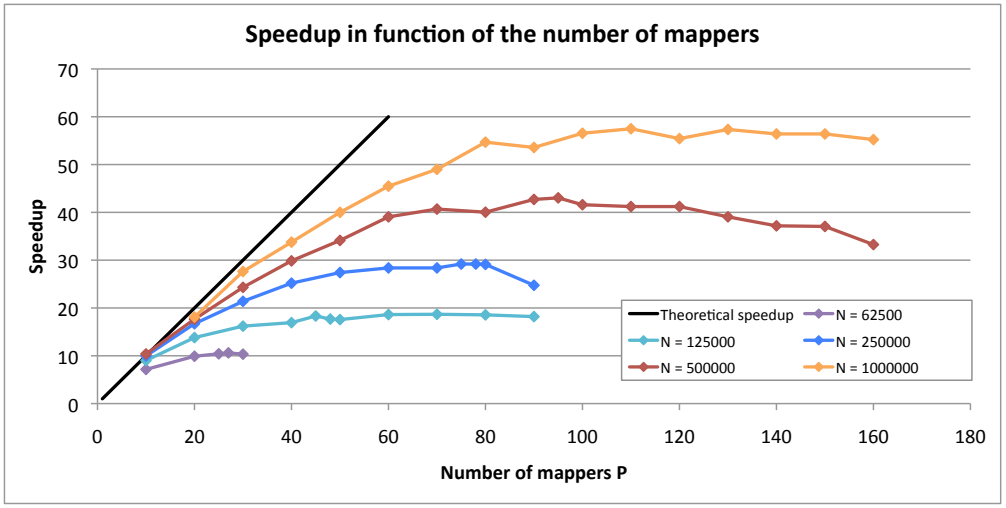

Figure 5.4: Charts of speedup performance curves for our cloud Batch K-Means implementation with different data set size. For a given size N, the speedup grows with the number of processing units until M^*, then the speedup slowly decreases.

values of M to determine the effective optimal values M^*_{eff} for a given N.

	N	M^*_{eff}	M^*	Wall Time	Sequential theoretic time	Effective Speedup	Theoretical Speedup ($= \frac{M^*}{3}$)
Exp. 1	62500	27	28	264	2798	10.6	9.34
Exp. 2	125000	45	45	306	5597	18.29	14.84
Exp. 3	250000	78	71	384	11194	29.15	23.55
Exp. 4	500000	95	112	521	22388	43.0	37.40

Table 5.3: Comparison between the effective optimal number of processing units M^*_{eff} and the theoretical optimal number of processing units M^* for different data set size.

As expected, one can see that $M^*_{eff}(N)$ and $T_M^{comp} + T_M^{comm}$ do not grow as fast as N. Between the experiment 1 and the experiment 4, N is multiplied by 8. Our theoretical model anticipates that M^*_{eff} should grow accordingly by $8^{2/3} = 4$. Indeed, M^*_{eff} grows from 27 to 95 (that is a 3.51 ratio). In the same way, our model anticipates that the execution wall time should grow by $8^{1/3} = 2$. Indeed, the execution wall time grows from 264 seconds to 521 seconds. Figure 5.4 provides the detailed experiment results of the speedup obtained for multiple values of N and M.

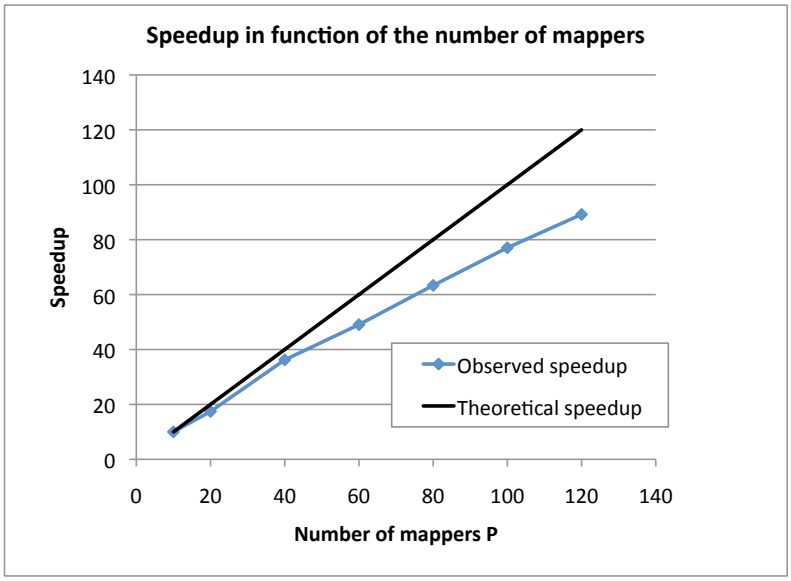

Figure 5.5: Charts of speedup performance curves for our cloud Batch K-Means implementation with different number of processing units. For each value of M, the value of N is set accordingly so that the processing units are heavy loaded with data and computations. When the number of processing units grows, the communication costs increase and the spread between the obtained speedup and the theoretical optimal speedup increases.

For our last experiment, we aim to achieve the nominal highest value possible for speedup. As explained in Subsection 5.3.5, for a fixed number of mappers M, the best achievable speedup is obtained by filling the RAM of each machine with data so each machine is in charge of a heavy computation load. While the previous table and Figure 5.4 show how the speedup grows with N (using $M^*(N)$ mappers), Figure 5.5 shows how the speedup grows with M (using the highest value of N that do not oversize the RAM of our VM). For this experiment, we set $K = 1000$, $d = 1000$, and set N in such a way that each mappers is given 50,000 data points $N = M * 50,000$ [4]. The results are reported in Figure 5.5.

Overall, the obtained performances are satisfactory and the predictive model provides reasonable estimates of the execution wall time and of the optimal number of processing units that need to be used. While there is room for im-

4. The value of $n = 50,000$ corresponds to 400 MBytes in RAM, while the RAM of a small role instance is supposed to be 1.75GBytes. In theory, we could have therefore loaded much more our instances. In practice, when we run this experiment in 2010, the VM crashed when we used higher values for n

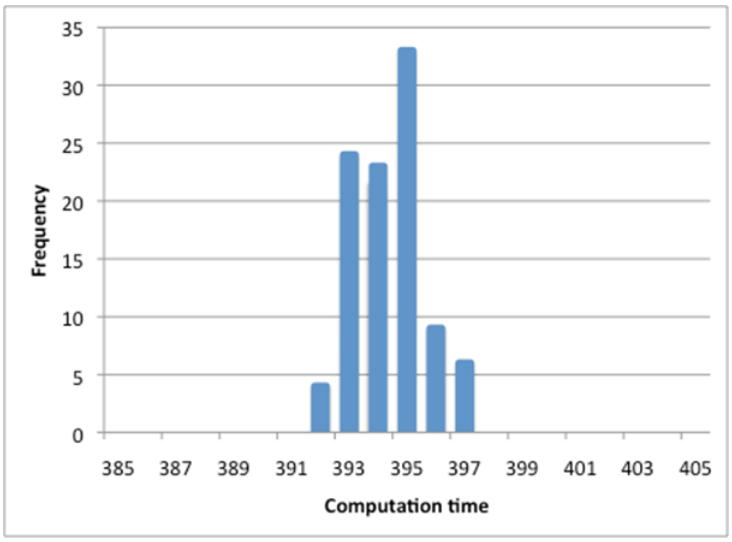

Figure 5.6: Distribution of the processing time (in second) for multiple runs of the same computation task for a single VM. As expected, the distribution is concentrated around a specific value.

proving our implementation, the latency issues might prevent resorting on a tree like $O(log(M))$ reducer as available in MPI without using direct inter-machines communications. Without native high performances API, communication aspects will probably remain a major concern in Azure implementations.

5.5.6 Straggler issues

In this subsection we investigate the variability of CPU performances of the multiple processing units and show that some tasks are processed in amounts of time significantly higher than expected, a phenomenon referred to as stragglers (see e.g. [47] or [80]).

In the first experiment, we run the same task —that consists in a heavy distance calculations load (each task corresponds to a reassignment phase)— 100 times. The task is expected to be run in 7 minutes. The results are provided in Figure 5.6.

In the second experiment, the same task is run 10 times in a row by 85 workers. Each of the 850 records therefore consists in the same computation load, performed on processing units supposedly of the same performance (Azure small role instances). The same experiment has been run on 2 different days, on 2

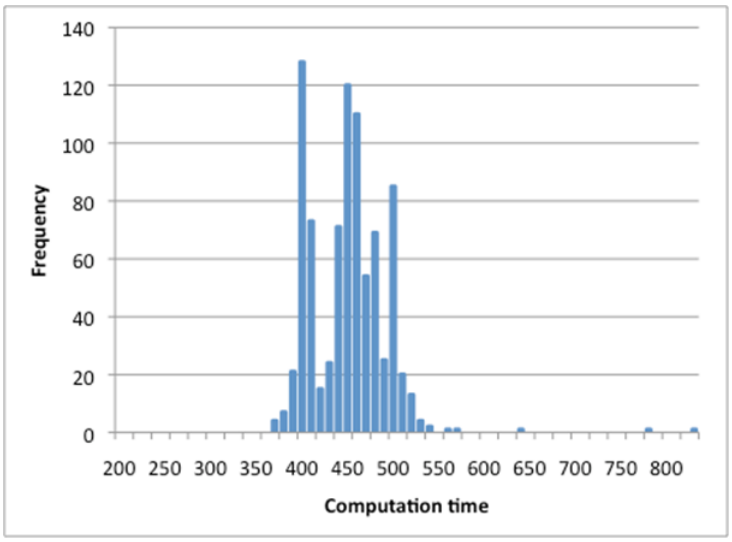

Figure 5.7: Distribution of the processing time (in second) for multiple runs of the same computation task for multiple VM. One can note the "3 modes" distribution and outliers (tasks run in much more time).

different hours, on different workers, but the same following patterns have been observed. Figure 5.7 provides the empirical distribution of these computation durations.

From this experiment, we can deduce that:

– *The 3-mode distribution* : 90% of the tasks are completed between 390 and 500 seconds. For the tasks which have been completed in this interval, we observe 3 different modes of our empirical distribution.The 3 modes may be due to hardware heterogeneity or multiple VM hosted on the same physical machine.

– *A worker can be affected by temporary slowness* : The three longest runs (823 seconds, 632 seconds, 778 seconds) have been performed by the same VM, which has also performed very well on other iterations: (360 seconds, 364 seconds, ...). This could be explained by very different reasons, such as the fact the physical machine hosting our VM has been hosting temporarily another VM, a temporary downsizing of the size of cache memory for our VM, etc.

Straggler issues have already been pointed out, for example in the original MapReduce article ([47]) by Google, yet they were observed while running thousands of machines. We show that straggler issues are also observed on a pool of workers as small as 100 VM. [47] describes a monitoring framework to detect tasks tak-

ing too much time, and uses backup workers to re-launch tasks that have been detected to be too long. Yet, this approach leads to wait for the standard duration of the task before detecting straggler tasks and launching again the corresponding tasks. This situation severely limits the potential speedup that can be achieved.

As pointed out by Graphlab in [85], the easy access of MapReduce frameworks and the great help it provides to design scalable applications has driven part of the machine-learning community to think their algorithms to fit in a MapReduce framework. However, in many cases the combination of stragglers and of a synchronous framework like MapReduce prevents users from obtaining good overall speedups.

Chapter 6

Practical implementations of distributed asynchronous vector quantization algorithms

6.1 Introduction

The distributed Batch K-Means is a synchronous distributed algorithm: the reassignment phase, which is the CPU-intensive phase, consists in distance computations that can be distributed over a pool of processing units thanks to its data-level parallelism property. Chapter 5 has highlighted how this algorithm can achieve good overall speedup and scale-up on a Cloud Computing platform but has also shown that its performance still suffers from the stragglers and the high costs of cloud communications. In the two following chapters, we develop a distributed asynchronous clustering algorithm so that the asynchronism will eventually reduce the performance loss incurred by the stragglers and the communications.

To do so, let us investigate the parallelization techniques of an algorithm which is a close relative to Batch K-Means. This algorithm is the Vector Quantization (VQ) algorithm, also referred to in the literature as online K-Means. The VQ algorithm is indeed online: its data input is drawn and processed piece-by-piece in a serial-fashion. After the examination of each data piece, a resume of the data —the prototypes— is refined accordingly to better reflect the data set. A reminder of the sequential VQ algorithm is given in Section 6.2.

Because of the online property of the VQ algorithm, the parallelization techniques of the VQ algorithm presented in Section 6.4 do not introduce unavoidable syn-

chronization barriers that would lead to overheads which would be comparable to the distributed Batch K-Means implementation ones. As a consequence, the VQ algorithm may seem to be more adapted to Cloud Computing platforms. Our work on the distributed VQ (DVQ) algorithm is divided into two chapters: the present chapter investigates various parallelization schemes of the VQ algorithm using simple programs simulating a distributed architecture, while Chapter 7 presents the actual behavior of the chosen DVQ scheme on Azure.

As outlined by Pagès in [94], the VQ algorithm belongs to the class of stochastic gradient descent algorithms (for more information on stochastic gradient descent procedures we refer the reader to Benveniste et al. in [29]). The DVQ algorithm is based upon the VQ technique: it executes several VQ procedures on different (and possibly distant) processing units while the results are broadcasted and merged through the network. As a consequence, the DVQ algorithm falls within the general framework of parallel gradient descent algorithms.

The distributed gradient descent algorithms have been vastly experimented and analyzed (see e.g. Chapter 14 of [28]). Distributed and asynchronous stochastic gradient descent procedures for supervised machine-learning problems have been studied theoretically and experimentally by Langford et al. in [118] and by Louppe and Geurts in [84]. In these papers the computer programs are based on shared memory architectures. However, our work focuses on the unsupervised learning problem of clustering without any efficient distributed shared memory. The lack of such a shared memory introduces significant time penalties when accessing data, therefore slowing down the exchange of information between the computing entities.

In order to avoid this problem, Zinkevich et al. propose in [119] a parallelized stochastic gradient descent scheme with no communication between processors until the end of the local executions. As mentioned by the authors, this perfectly suits the popular distributed computing framework MapReduce (see, for instance, Dean and Ghemawat in [47]). However, in our quantization context this approach would not work because the loss function, namely the empirical distortion, is not convex [1]. Therefore a final average of completely independent VQ executions would not lead to satisfactory prototypes, as shown by the results of Section 6.4 below when the delays get large.

The non-convexity of the loss function recalled in equation (6.1) seems to be

1. Even in a convex context, this solution may turn out to be not optimal, as outlined in the so-called "no-communication" solution study in [49].

a very important point, if not the most important, of the nature of our problem. Indeed, in the convex and smooth case, recent results have presented parallelization schemes that have proved to be asymptotically optimal (see for instance the recent [49]). We show in this chapter that comparable distribution schemes do not provide speedup in our non-convex case but we provide improvements through other distribution schemes for which sensitive speedups are obtained.

We will not be concerned with practical optimizations of the several parameters of VQ implementations such as the initialization procedures or the choice of the sequence of steps $\{\varepsilon_t\}_{t=1}^{\infty}$ (introduced in Section 6.2). In the present chapter, we assume that a satisfactory sequential VQ implementation has been found. Therefore, our goal is to make this algorithm "scalable" (i.e. able to cope with larger data sets) using multiple computing resources without modifying those parameters. Consequently, in all our experiments, the initialization procedure is always the same: each initial prototype is an average of 20 arbitrary data vectors. For a review of standard initialization procedures, the reader is referred to Peterson et al. in [96] where a satisfactory method is also provided for the usual Batch K-Means algorithm. The techniques proposed by Milligan and Isaac in [89], Bradley and Fayyad in [35], or Mirkin in [90] also seem to perform well. As for the learning rate, similarly to Langford et al. in [118] or [28], we set $\varepsilon_t = 1/\sqrt{t}$ for all $t \geq 1$. A thorough analysis of the theoretical properties of the learning rates can be found in Bottou et al. in [33, 34].

The rest of this chapter is organized as follows. Section 6.2 provides a short introduction to the sequential VQ algorithm. Section 6.3 describes the synthetic functional data used in the experiments of this and the next chapter. Section 6.4 is devoted to analyses and discussions about simple distributed schemes implementing VQ procedures. In particular, we show in this section that the first natural implementation of VQ cannot bring satisfactory results. However, we describe alternative parallelization schemes that, as we had expected, give substantial speedup, i.e. gain of execution time brought by more computing resources compared to a sequential execution.

6.2 The sequential Vector Quantization algorithm

Let us recall the exact clustering problem dealt with in Chapter 5 and detailed in equation (6.1): given a data set of N points $\{\mathbf{z}_t\}_{t=1}^{N}$ of a d dimensional space, we want to construct κ points $\{w_k\}_{k=1}^{\kappa}$, referred to in the following as prototypes or centroids, as a resume of the data set using the euclidean distance as a similarity

measure. Through this similarity measure, one can define the empirical distortion:

$$C_N(w) = \sum_{t=1}^{N} \min_{\ell=1,\ldots,\kappa} \|\mathbf{z}_t - w_\ell\|^2, \quad w \in \left(\mathbb{R}^d\right)^\kappa. \tag{6.1}$$

Like Batch K-Means, the popular VQ algorithm (see Gersho and Gray in [55]) introduces a technique to build prototypes $\{w_k\}_{k=1}^{\kappa}$ that provides a satisfactory result as regards the previous criterion.

The VQ algorithm defined by equation (6.2) consists in incremental updates of the $\left(\mathbb{R}^d\right)^\kappa$-valued prototypes $\{w(t)\}_{t=0}^{\infty}$. Starting from a random initial $w(0) \in \left(\mathbb{R}^d\right)^\kappa$ and given a series of positive *steps* $(\varepsilon_t)_{t>0}$, the VQ algorithm produces a series of $w(t)$ by updating w at each step with a "descent term". Let us first introduce $H(\mathbf{z}, w)$ defined by

$$H(\mathbf{z}, w) = \left((w_\ell - \mathbf{z}) \, \mathbb{1}_{\left\{l = \operatorname{argmin}_{i=1,\ldots,\kappa} \|\mathbf{z} - w_i\|^2\right\}} \right)_{1 \leq \ell \leq \kappa}.$$

Then, the VQ iterations can be written:

$$w(t+1) = w(t) - \varepsilon_{t+1} H\left(\mathbf{z}_{\{t+1 \bmod n\}}, w(t)\right), \quad t \geq 0, \tag{6.2}$$

where the mod operator stands for the remainder of an integer division operation. The entire VQ algorithm is described in the logical code of Algorithm 6. The VQ iterations make passes (i.e. cycles) over the data set until a stopping criterion is met. The comparison with the theoretical description, where the data set is supposed to be infinite and consequently where there is no cycle, has been studied by Bermejo and Cabestany in [30].

As pointed out by Bottou and Bengio in [32], the VQ algorithm defined by the iterations (6.2) can also be viewed as the online version of the widespread Batch K-Means presented in Chapter 5 (which is also referred as Lloyd's method in [83] for the definition). The VQ algorithm is also known as the Competitive Learning Vector Quantization (especially in the data analysis community), the Kohonen Self Organizing Map algorithm with 0 neighbor (see for instance Kohonen in [76]) or the online K-Means procedure (see MacQueen in [86] and Bottou and Bengio in [32]) in various fields related to statistics.

As already explained in the introduction, it is well known that the VQ algorithm belongs to the class of stochastic gradient descent algorithms. However, the almost sure convergence of the VQ algorithm cannot be obtained by general tools of gradient descent such as the Robbins-Monro method for instance (see Patra in [95]). Indeed, the main difficulty essentially arises from the lack of convexity

and smoothness of the distortion. We refer the reader to Pagès in [94] for a proof of the almost sure convergence of the VQ procedure.

Algorithm 6 Sequential VQ algorithm

Select κ initial prototypes $(w_k)_{k=1}^{\kappa}$
Set t=0
repeat
  for $k = 1$ to κ **do**
    compute $||\mathbf{z}_{\{t+1 \bmod n\}} - w_k||_2^2$
  end for
  Deduce $H(\mathbf{z}_{\{t+1 \bmod n\}}, w)$
  Set $w(t+1) = w(t) - \varepsilon_{t+1} H\left(\mathbf{z}_{\{t+1 \bmod n\}}, w(t)\right)$
  increment t
until the stopping criterion is met

6.3 Synthetic functional data

The various parallelization schemes investigated in this chapter have been tested using simple programs simulating a distributed architecture, and synthetic vector data [2]. Contrary to Batch K-Means, the parallelization schemes of the VQ algorithm and the sequential VQ implementation do not provide the very same results. As a consequence, the actual data values have an influence on the algorithm duration and performance; contrary to Batch K-Means, we will need to evaluate each parallelization scheme both on a speedup criterion and on an accuracy criterion (using the empirical distortion defined by equation (6.1) above). The use of synthetic data serves three purposes: firstly, through this synthetic generation, the data set size can be tuned to resize our clustering problem so that it embodies a challenging problem as far as computations and communications are concerned. Secondly, for a given data set size, we can tune the "clustering difficulty" by modifying the complexity of our data, i.e. by modifying the actual data set dimensionality (tuned by G in the following). Finally, the ability to re-generate any part of the data set on any processing unit without downloading it from a storage is of great help when evaluating any quantization results on the cloud as it removes a lot of communication and therefore eases the evaluation process architecture.

2. Source code is available at the address `http://code.google.com/p/clouddalvq/`

Many real-world applications produce high dimensional data. In numerous situations, such data can be considered as sampled functions. This is the case in the popular context of time series, meteorological data, spectrometric data etc. We therefore have chosen to resort to B-splines to analyze our clustering algorithm implementations. In the present section we explain the construction of the B-splines mixtures random generators used in our experiments. Note that the recent researches by Abraham et al. in [21] and Rossi et al. in [101] focus on clustering algorithms for B-splines functional data.

6.3.1 B-spline functions

The B-spline functions have been thoroughly studied for many decades. They have been analyzed both for their surprising mathematical properties and their ability to create nice shapes. Consequently they are frequently used in industries related to design such as automotive industry, architecture, graphics editor software, etc. In this subsection we only provide a computational definition. For more information on B-splines history and mathematical properties, we refer the reader to de Boor in [46].

The term "B-spline" —an abbreviation of basis spline—, is ambiguous as its meaning varies depending on the context: in some situations, a B-spline refers to one of the $b_{i,n}(.)$ functions defined below. In other situations, a B-spline refers to a linear combination of the $b_{i,n}(.)$ functions. In the following, the $b_{i,n}(.)$ functions are referred to as basic B-splines, and a B-spline is defined as a linear combination of these basic B-splines.

The family of basic B-spline functions of degree n with χ knots $x_0 \leq x_1 \leq \ldots \leq x_{\chi-1}$ is composed of $\chi - n - 1$ piecewise polynomial functions. The symbols n and χ denote both integers satisfying $\chi \geq n + 2$. They can be defined recursively by the Cox-de Boor formula below (see for instance de Boor in [45]). For all $x \in [x_0, x_{\chi-1}]$,

$$b_{i,0}(x) = \begin{cases} 1 & \text{if } x_i \leq x < x_{i+1} \\ 0 & \text{otherwise} \end{cases}, \quad i = 0, \ldots, \chi - 2,$$

and

$$b_{i,n}(x) = \frac{x - x_i}{x_{i+n} - x_i} b_{i,n-1}(x) + \frac{x_{i+n+1} - x}{x_{i+n+1} - x_{i+1}} b_{i+1,n-1}(x), \quad i = 0, \ldots, \chi - n - 2.$$

Figure 6.1 is a plot of six cubic basic B-splines where $\chi = 10$ and $n = 3$.

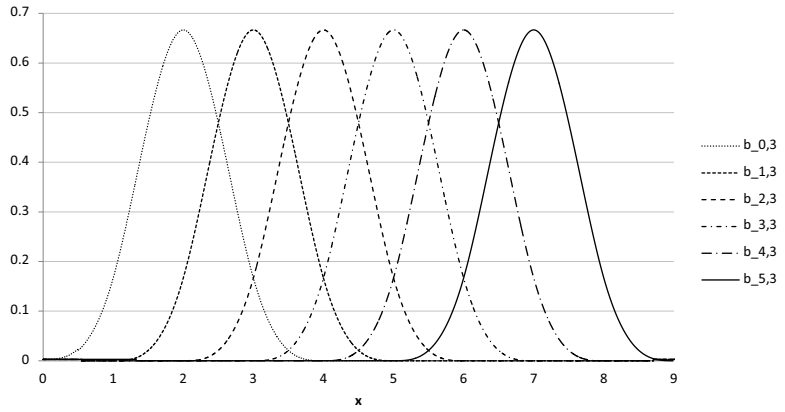

Figure 6.1: Plots of the six basic cubic B-spline functions with ten uniform knots: $x_0 = 0, \ldots, x_9 = 9$ ($n = 3$, $\chi = 10$).

As mentioned earlier, a B-spline is a vector of the linear span of the basic B-splines. Thus, a B-spline takes the form, for any $x \in [x_0, x_{\chi-1}]$,

$$b(x) = \sum_{i=0}^{\chi-n-2} p_i b_i(x), \tag{6.3}$$

where $\{p_i\}_{i=0}^{\chi-n-2} \in \mathbb{R}^{\chi-n-1}$ are referred to as the control points or de Boor points. Figure 6.2 shows a cubic (i.e., $n = 3$) B-spline. The graph illustrates the natural smoothness of such functions. In the sequel, we consider only cubic B-splines and uniform knots: $x_0 = 0, \ldots, x_{\chi-1} = \chi - 1$ with $\chi \geq 6$.

6.3.2 B-splines mixtures random generators

Let us now describe the random generators used throughout the experiments described in Chapters 6 and 7. Each of these vector-valued generators is based on G ($G \geq 1$) B-spline functions which, from now on, will be referred to as the centers of the mixture. We also want these centers "not to be too close to each other". To do so, we have chosen our centers with (nearly) orthogonal coefficients p_i's appearing in equation (6.3). In the next paragraph, we explain the construction of such coefficients.

For $g = 0, \ldots, G-1$ and $i = 0, \ldots, \chi - 5$, let the $u_{g,i}$'s be reals, drawn independently from the standard uniform distribution on the open interval $(0, 1)$. The vectors $\{\{p_{g,i}\}_{i=0}^{\chi-5}\}_{g=0}^{G-1}$ are defined as block-wise orthogonal vectors computed

Figure 6.2: Plot of a cubic B-spline, a linear combination of the basic B-splines plotted in Figure 6.1

by applying a revised Gram-Schmidt procedure on the G vectors $\{u_{g,i}\}_{i=0}^{\chi-5}$ (see for instance Greub in [61]). Let us keep in mind that the Gram-Schmidt algorithm is well defined if the number of vectors, here G, is lower than or equal to the dimension of the vector space which is equal to $\chi-4$. In all our experiment settings, we will have $G > \chi-4$. Therefore, we divide our G vectors $\{\{u_{g,i}\}_{i=0}^{\chi-4}\}_{g=0}^{G-1}$ into subgroups of $\chi-4$ vectors. To each group, we apply a block-wise Gram-Schmidt procedure. All the G vectors thus obtained are then normalized to a common value $sc > 0$ and are then referred to the G vectors $\{\{p_{g,i}\}_{i=0}^{\chi-5}\}_{g=0}^{G-1}$.

For any $g \in \{1, \ldots, G-1\}$, let B_g be the B-spline defined by equation (6.3) with the control points $\{p_{g,i}\}_{i=0}^{\chi-5}$ chosen as explained in the paragraph above. Let us not forget that in our context, the data cannot be functions but only sampled functions. Consequently, the centers of the distribution are the functions B_g's, only observed through the d-dimensional vector ($d \geq 1$) $\overline{B}_g$, where

$$\overline{B}_g = \{B_g\left(i(\chi-1)/d\right)\}_{i=0}^{d-1}.$$

The vectors $\overline{B}_0, \ldots, \overline{B}_{G-1}$ define the centers of the $\mathbb{R}^d$-valued law of probability defined below. As shown in Figure 6.3, the orthogonal property of the coefficients makes them "not too close to each other", as requested.

We are in a position to define the distribution simulated by our $\mathbb{R}^d$-valued random generator. Let **N** be a uniform random variable over the set of integers $\{0, \ldots, G-1\}$ and ε a $\mathbb{R}^d$-valued Gaussian random variable with 0 mean and $\sigma^2 I_d$ for covariance matrix, where $\sigma > 0$ and I_d stands for the $d \times d$ identity matrix. Our random generators simulate the law of the d-dimensional random

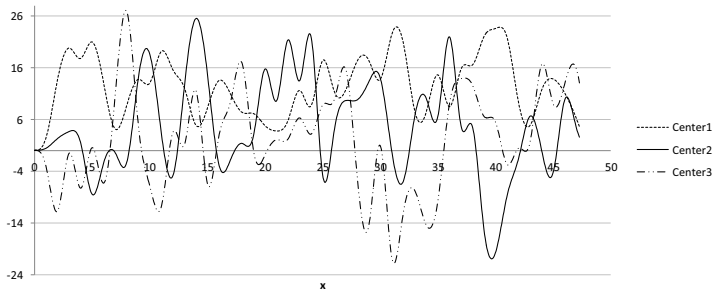

Figure 6.3: Plot of four splines centers with orthogonal coefficients: $\overline{B}_1, \ldots, \overline{B}_4$ where $G = 1500$, $d = 1500$, $\chi = 50$, $sc = 10$.

variable **Z**, defined by

$$\mathbf{Z} = \overline{B}_{\mathbf{N}} + \varepsilon. \tag{6.4}$$

Figure 6.4 shows two independent realizations of the random variable **Z** defined by equation (6.4), the sampled functional data used in our experiments.

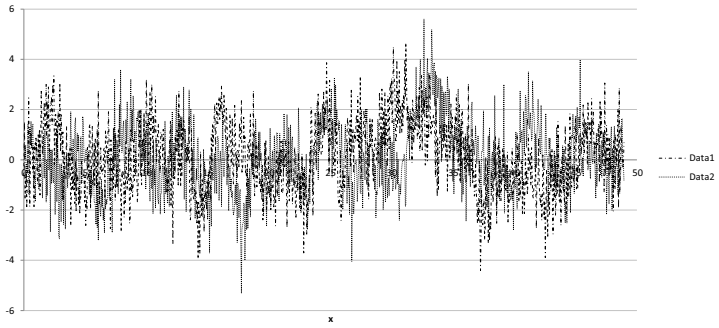

Figure 6.4: Plot of two independent realizations of the random variable **Z** defined by equation (6.4): $\overline{B_1}, \ldots, \overline{B_4}$ where $G = 1500$, $d = 1500$, $\chi = 50$, $sc = 10$.

6.4 Discussing parallelization schemes of the VQ algorithm

This section is devoted to the analysis of some practical parallelization schemes for the DVQ algorithm. Indeed, the practical parallelization of the VQ procedure is not straightforward. For example, let us remark that, if two different prototypes

versions w^1 and w^2 have been computed by two concurrent VQ executions, there is no guarantee that the quantization scheme provided by the convex combination $\alpha w_1 + (1-\alpha)w_2$ ($\alpha \in (0,1)$) has good performance regarding the non-convex empirical distortion criterion. Therefore, careful attention should be paid to the parallelization scheme of any attempt of DVQ implementation.

Following the notation of Subsection 5.3.1, we suppose that we own a (potentially) large data set split among the local memory of M computing units. This split data set is represented by the sequences $S^i = \{\mathbf{z}_t^i\}_{t=1}^n$, $i \in \{1, \ldots, M\}$, the global data set being composed of $N = nM$ vectors. In addition to its part of the split data set S^i, the processing unit i owns a specific version of the prototypes w^i.

From the M prototypes versions w^i, $i \in \{1, \ldots, M\}$, a shared version w^{srd} of the prototypes is built, following a merging logic which varies over the different parallelization schemes described below. This shared version corresponds to the result that the algorithm would return to the user if it was asked to. It is strongly related to the "consensus version" described for example in [95]. The shared version is the prototypes version used to evaluate the actual accuracy of our various implementations. More specifically, we use a normalized empirical distortion to evaluate the vector w^{srd}. This empirical distortion is defined for all $w \in \left(\mathbb{R}^d\right)^\kappa$ by

$$C_{n,M}(w) = \frac{1}{nM} \sum_{i=1}^{M} \sum_{t=1}^{n} \min_{\ell=1,\ldots,\kappa} \left\|\mathbf{z}_t^i - w_\ell\right\|^2. \tag{6.5}$$

We have already highlighted in Chapter 5 the importance for distributed Batch K-Means of making sure that the machines RAM are heavily loaded. In the same manner, the RAM of each processing unit is heavily loaded in our distributed VQ experiments. As a consequence, the number of points in the total data set (nM) is made dependant of the number of computing unit M. Strictly speaking, the values of $C_{n,M}(w)$ obtained for different values of M and fixed values of n and w are therefore not equal. Yet, they are all very close to the generalization error and can be compared without trouble.

All the following parallelization schemes are evaluated with a simulated distributed architecture (actually performed sequentially on a single-core CPU). All the experiments in this section are carried out with the following parameters: $n = 1000$, $d = 100$, $\kappa = 100$ and with the synthetic data sampled by the random

generators introduced above. The generators are tuned with the following settings: $G = 150$, $\chi = 100$, $sc = 100$ and $\sigma = 5.0$. As it is detailed in Subsection 7.2.2, the evaluation of a prototypes version is often as long as the entire quantization algorithm. As a consequence, reporting the evolution of the quantization error over time or over the iterations is much more costly than running the algorithm (as for a given VQ execution, the evaluation of the quantization error is run multiple times to plot its evolution). The chosen settings reflect this fact and are kept small to lower the evaluation time performed on our single-core CPU. Our evaluation settings are modified in Chapter 7 to induce much larger computational challenges, as we will leverage on hundreds of processing units for the quantization algorithms and for the evaluation of these algorithms.

In order to bring substantial speedup, three schemes for VQ parallelization are proposed below. We start by the investigation of the most natural parallelization scheme, where the versions resulting of the distributed VQ executions are averaged on a regular basis. This updating rule is used for example in [106], [95] or [49] [3]. We have tracked the evolution of the empirical distortion with the iterations. This evolution shows that no speedup can be brought using this scheme. Thus, a new model is proposed to overcome this unsatisfactory situation. The main idea beneath this new scheme is the following one: the agreement between the computing units is not performed using an average anymore, but it uses instead the cumulated sum of the so-called "displacement terms". Finally, the last model to be considered is an improvement of the second model, where delays are introduced to prepare the analysis of asynchronism used in Chapter 7.

6.4.1 A first parallel implementation

Our investigation therefore starts with the following simple parallelization scheme of the VQ technique. All the versions are set equal at time $t = 0$, $w^1(0) = \ldots = w^M(0)$. For all $i \in \{1, \ldots, M\}$ and all $t \geq 0$, we have the following iterations:

$$\begin{cases} w^i_{temp} = w^i(t) - \varepsilon_{t+1} H\left(\mathbf{z}^i_{\{t+1 \bmod n\}}, w^i(t)\right) & \\ w^i(t+1) = w^i_{temp} & \text{if } t \bmod \tau \neq 0 \text{ or } t = 0, \\ \begin{cases} w^{srd} = \frac{1}{M} \sum_{j=1}^{M} w^j_{temp} \\ w^i(t+1) = w^{srd} \end{cases} & \text{if } t \bmod \tau = 0 \text{ and } t \geq \tau. \end{cases} \tag{6.6}$$

3. In [49], the gradients are averaged but the prototypes versions. However, this strategy results in the very same prototypes version update rule

This algorithm is composed of two phases. The processing phase is given by the first equation of (6.6). It corresponds to a local execution of the VQ procedure. The result of these computations is referenced by the variables w_{temp}^i. The averaging phase, defined by the braced inner equations, describes the merging strategy chosen for this parallelization scheme. Note that the averaging phase is executed only whenever τ points have been processed by each concurrent processor. The larger the integer τ is, the more independent the concurrent processors are left for their execution. During the averaging phase a common shared version w^{srd} is computed as the average of all the w^i, $i \in \{1, \ldots, M\}$. Once w^{srd} has been computed, all the local versions, namely the w^i, are set equal to w^{srd}. If the averaging phase does not occur, then the local versions are naturally set equal to the result of the local computations. See Figure 6.5 for a graphical illustration of the iterations (6.6).

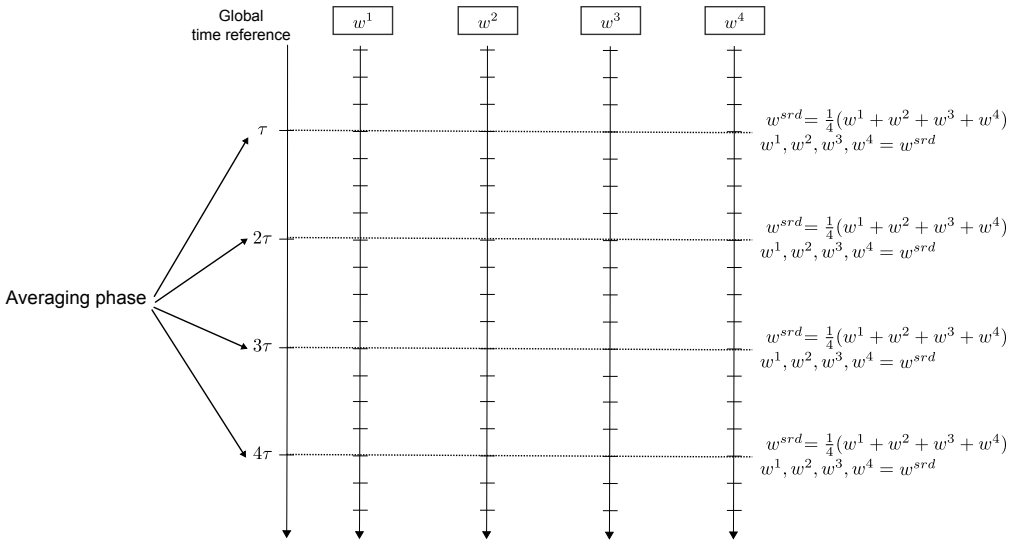

Figure 6.5: Illustration of the parallelization scheme of VQ procedures described by equations (6.6). The prototypes versions are synchronized every τ points.

This parallelization scheme fits the theoretical Distributed Asynchronous Learning Vector Quantization (DALVQ) model presented by Patra in [95]. In particular, the iterations (6.6) define a DALVQ procedure without asynchronism and for which the averaging phases are performed on a regular basis. A straightforward examination shows that both sets of assumptions (AsY_1) and (AsY_2) presented in [95] are satisfied. As a consequence, asymptotic consensus and convergence

towards critical points are guaranteed.

The results are gathered with different values of τ ($\tau \in \{1, 10, 100\}$) in the charts displayed in Figure 6.6. For each chart, we plot the performance curves when the distributed architecture has a different number of computing instances: $M \in \{1, 2, 10\}$. The curve of reference is the sequential execution of the VQ, that is $M = 1$. We can notice that for every value of $\tau \in \{1, 10, 100\}$, multiple resources do not bring speedup for convergence. Even if more data are processed, there is no increase in the convergence speed. To conclude, no gain in terms of wall time can be brought using this parallel scheme. In the next subsection we investigate the cause of these non-satisfactory results and propose a solution to overcome this problem.

6.4.2 Towards a better parallelization scheme

Let us start the investigation of the parallel scheme given by iterations (6.6) by rewriting the sequential VQ iterations (6.2) using the notation introduced in Section 6.1. For $t \geq \tau$ it holds

$$w(t+1) = w(t-\tau+1) - \sum_{t'=t-\tau+1}^{t} \varepsilon_{t'+1} H\left(\mathbf{z}_{\{t'+1 \bmod n\}}, w(t')\right). \qquad (6.7)$$

For iterations (6.6), consider a time slot $t \geq 0$ where an averaging phase occurs, that is, t mod $\tau = 0$ and $t > 0$. Then, for all $i \in \{1, \ldots, M\}$,

$$w^i(t+1) = w^i(t-\tau+1) - \sum_{t'=t-\tau+1}^{t} \varepsilon_{t'+1} \left(\frac{1}{M} \sum_{j=1}^{M} H\left(\mathbf{z}^j_{t'+1}, w^j(t')\right) \right). \qquad (6.8)$$

To the empirical distortion defined by equation (6.5) is associated its theoretical counterpart the distortion function C (see for example [98] or [31]). In the first situation of iterations (6.7), for a sample $\mathbf{z}$ and $t' > 0$, $H(\mathbf{z}, w(t'))$ is an observation and an estimator of the gradient $\nabla C(w(t'))$ (see e.g. [95]). In the second situation of iterations (6.8), let us assume that the multiple versions are close to each other. This means that $w^j(t') \approx w^i(t')$, for all $(i, j) \in \{1, \ldots, M\}^2$. Thus, the average

$$\frac{1}{M} \sum_{j=1}^{M} H\left(\mathbf{z}^j_{\{t'+1 \bmod n\}}, w^j(t')\right)$$

can also be viewed as an estimation of the gradient $\nabla C(w^i(t'))$, where $i \in \{1, \ldots, M\}$.

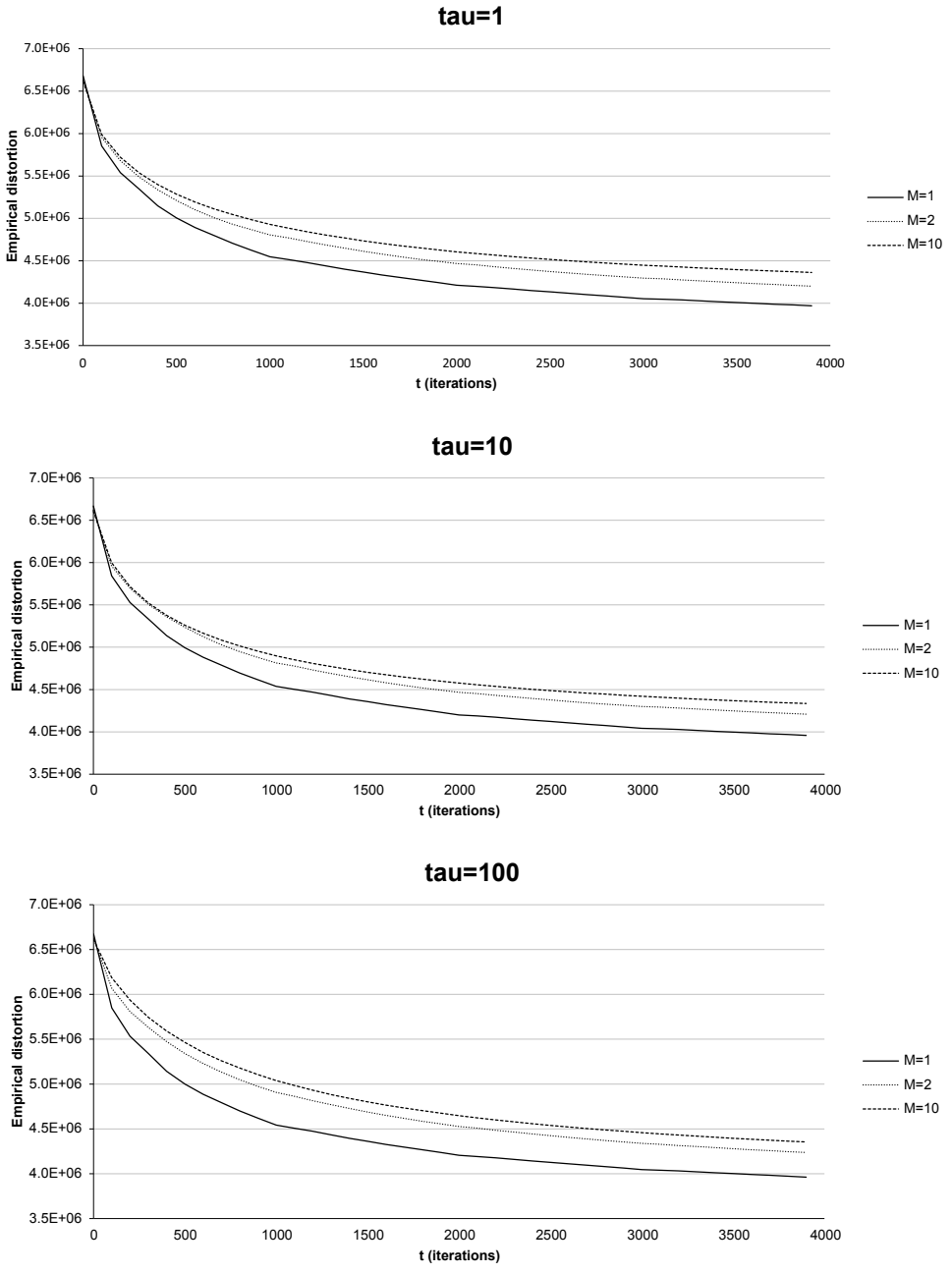

Figure 6.6: Charts of performance curves for iterations (6.6) with different numbers of computing entities: $M = 1, 2, 10$. The three charts correspond to different values of τ which is the integer that characterizes the frequency of the averaging phase ($\tau = 1, 10, 100$).

In both situations, the descent terms following the step $\varepsilon_{t'+1}$ in equations (6.7) and (6.8) are estimators of gradients. Consequently, the two algorithms can be thought of as stochastic gradient descent procedures with different estimators. However, they are driven by the same learning rate which is given by the sequence of steps $\{\varepsilon_t\}_{t=1}^{\infty}$. The convergence speed of a non-fixed step gradient descent procedure is essentially driven by the decreasing speed of the sequence of steps (see for instance Kushner et al. in [77] or Benveniste et al. in [29]). The choice of this sequence is subject to an exploration/convergence trade-off. In the case of the sequential VQ iterations (6.2) the time slot t and the current count of samples processed are equal. Therefore, for the same time slot t, the distributed scheme of iterations (6.6) has visited much more data than the sequential VQ algorithm to build its descent terms. Yet, since the two algorithms share the same learning rate, they are expected to get comparable convergence speeds, which is confirmed by the charts in Figure 6.6.

We now introduce for all $j \in \{1, \ldots, M\}$ and $t_2 \geq t_1 \geq 0$ the term $\Delta^j_{t_1 \to t_2}$ —referred to in the following as the "displacement term"— which corresponds to the variation of the prototypes computed by processor j during the time interval (t_1, t_2),

$$\Delta^j_{t_1 \to t_2} = \sum_{t'=t_1+1}^{t_2} \varepsilon_{t'+1} H\left(\mathbf{z}^j_{\{t'+1 \bmod n\}}, w^j(t')\right).$$

Using the notation above, the parallel scheme given by iterations (6.6) writes, for all $t > 0$ where $t \bmod \tau = 0$ and all $i \in \{1, \ldots, M\}$,

$$w^i(t+1) = w^i(t-\tau+1) - \frac{1}{M} \sum_{j=1}^{M} \Delta^j_{t-\tau \to t}. \tag{6.9}$$

In the previous paragraphs we have explained that the sequential VQ and the distributed scheme above share the same convergence speed with respect to the iterations $t \geq 0$. Therefore, if one considers the evolution of the learning rate with respect to the number of samples processed, then the distributed scheme has a much lower learning rate, thereby favoring exploration to the detriment of convergence. As explained in the introduction to the chapter, we assume that the learning rate provided by the sequence $\{\varepsilon_t\}_{t=1}^{\infty}$ is supposed to be satisfactory for the sequential VQ. In this work, we seek for a parallel scheme that has, in comparison to a sequential VQ, the same learning rate evolution in terms of processed samples and whose convergence speed with respect to iterations is accelerated. Notice that these iterations correspond to the true wall time measured by an exterior observer. With this aim in view, we propose the new system of

equations (6.10).

At time $t = 0$ all versions are equal, $w^1(0) = \ldots = w^M(0) = w^{srd}$. For all $i \in \{1, \ldots, M\}$ and all $t \geq 0$, the new parallel scheme is given by

$$\begin{cases} w^i_{temp} = w^i(t) - \varepsilon_{t+1} H\left(\mathbf{z}^i_{\{t+1 \bmod n\}}, w^i(t)\right) & \\ w^i(t+1) = w^i_{temp} & \text{if } t \bmod \tau \neq 0 \text{ or } t = 0, \\ \begin{cases} w^{srd} = w^{srd} - \sum_{j=1}^{M} \Delta^j_{t-\tau \to t} \\ w^i(t+1) = w^{srd} \end{cases} & \text{if } t \bmod \tau = 0 \text{ and } t \geq \tau. \end{cases} \tag{6.10}$$

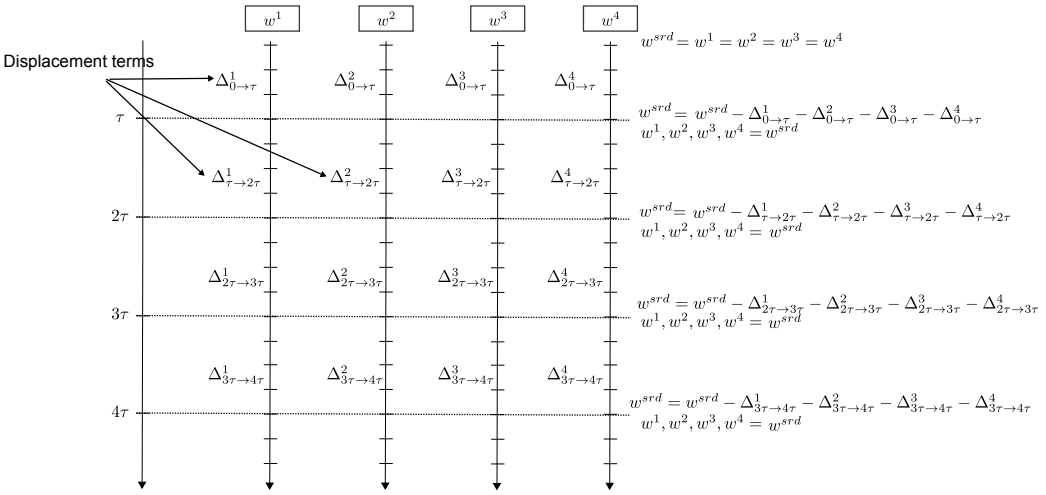

Figure 6.7: Illustration of the parallelization scheme of VQ procedures described by equations (6.10).

Roughly speaking, these iterations add up the displacement terms $\Delta^j_{\cdot\to\cdot}$ rather than use the average as shown in equation (6.9). The computation of w^{srd} described in equations (6.10) is now called "reducing phase" instead of "averaging phase". It can be easily implemented, but at the price of keeping supplementary information in the local memory of the computing units. More precisely, iterations (6.10) require to have, for processor j at any time $t \geq 0$, the displacement term from $\lfloor t/\tau \rfloor \tau + 1$ (last reducing phase) to the current time t. With the notation above, this displacement term writes $\Delta^j_{\lfloor t/\tau \rfloor \tau \to t}$. Figure 6.7 provides a synthetic representation of these iterations.

Contrary to the previous parallelization scheme, the iterations defined in equation (6.10) do not fit into the theoretical Distributed Asynchronous Learning Vector

Quantization (DALVQ) model presented by Patra in [95]. To our knowledge, no theoretical results provide such iterations with the convergence towards critical points. However, the results of the simulations in the charts displayed in Figure 6.8 show effective convergence and that substantial speedups are obtained with more distributed resources. The acceleration is greater when the reducing phase is frequent. Indeed, if τ is large then more autonomy has been granted to the concurrent executions and they could be attracted to different regions that would slow down consensus and convergence.

6.4.3 A parallelization scheme with communication delays.

In the previous subsections we have focused on the speedup that can be brought by parallel computing resources. However, the previous parallelization schemes did not deal with the computation of shared versions. In our context, no efficient shared-memory is available. The shared version will be visible for other processors only through network connections. The resulting communication costs need to be taken into account in our model. With this aim in view, we introduce delays.

The effect of delays for parallel stochastic gradient descent has already been studied for instance by Langford et al. in [118] or by Dekel et al. in [49]. However, these authors consider only the case where the computing architecture is endowed with an efficient shared memory. In this context, as outlined in [118], computing the gradient is much more time-consuming than computing the update (which includes communication latencies). On the contrary, in the Cloud Computing environment, the gradient computation in a single point is much faster than the actual update, because of the communications. In this subsection, we improve the model of iterations (6.10) with non negligible communication costs, resulting in the following more realistic iterations (6.11).

For each time $t \geq 0$, let $\tau^i(t)$ be the latest time before t when the unit i finished to send its updates and received the shared version. At time $t = 0$ we have $w^1(0) = \ldots = w^M(0) = w^{srd}$, and for all $i \in \{1, \ldots, M\}$ and all $t \geq 0$,

$$\begin{cases} w^i_{temp} = w^i(t) - \varepsilon_{t+1} H\left(\mathbf{z}^i_{\{t+1 \bmod n\}}, w^i(t)\right) & \\ w^i(t+1) = w^i_{temp} & \text{if } t \bmod \tau \neq 0 \text{ or } t = 0, \\ \begin{cases} w^{srd} = w^{srd} - \sum_{j=1}^{M} \Delta^j_{t-2\tau \to t-\tau} & \text{if } t \bmod \tau = 0 \text{ and } t \geq 2\tau, \\ w^i(t+1) = w^{srd} - \Delta^i_{t-\tau \to t} & \text{if } t \bmod \tau = 0 \text{ and } t \geq \tau. \end{cases} & \end{cases} \tag{6.11}$$

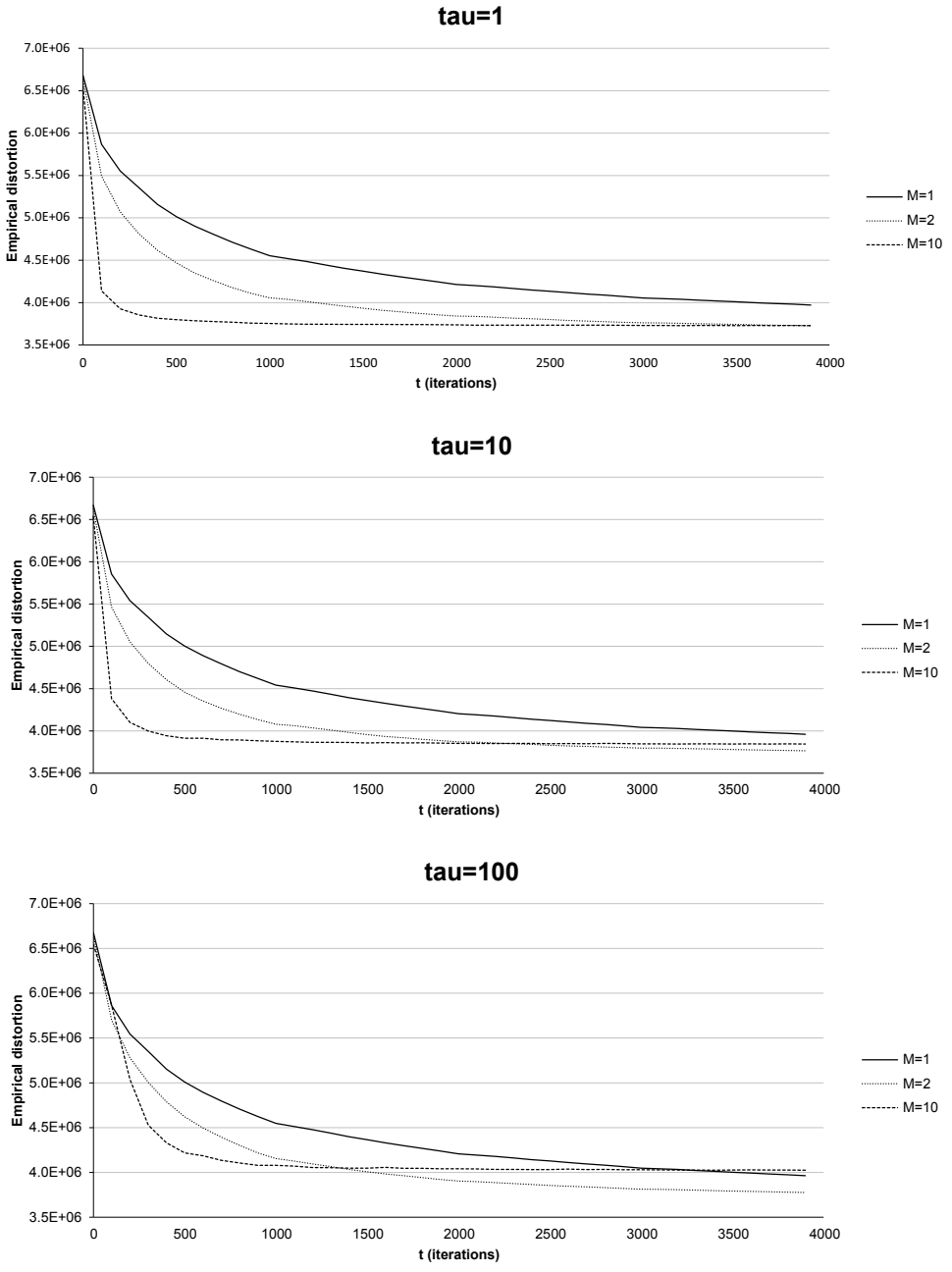

Figure 6.8: Charts of performance curves for iterations (6.10) with different numbers of computing entities, $M = 1, 2, 10$. The three charts correspond to different values of τ ($\tau = 1, 10, 100$).

The main difference with iterations (6.10) is the introduction of stale information: a dedicated process permanently modifies the shared version with the latest updates received from the processing units. This shared version is made available for every processing unit as a kind of consensus that emerges from the aggregation of the prototypes versions. Because of the communication delays now modeled, each processing unit can access to all the information of its own VQ procedure, but can only access stale gradient results from the other processing units through the shared version. This model easily extends to asynchronous schemes, as the aggregation process does not require the displacement terms from all of the processing units to proceed and can provide a partially updated shared version. This model is therefore close to the cloud asynchronous version developed in Chapter 7. Figure 6.9 gives a synthetic view of the iterations (6.11).

The performance curves resulting from the simulations of this distributed scheme are displayed in Figure 6.10. Remark that in our experiments small delays ($\tau = 1$ or 10) do not have any severe impact, since the results are similar to the non-delayed model reported in Figure 6.8. However, large delays $\tau = 100$ can have a significant impact and prevent the distributed scheme from bringing speedup to convergence with more computing resources.

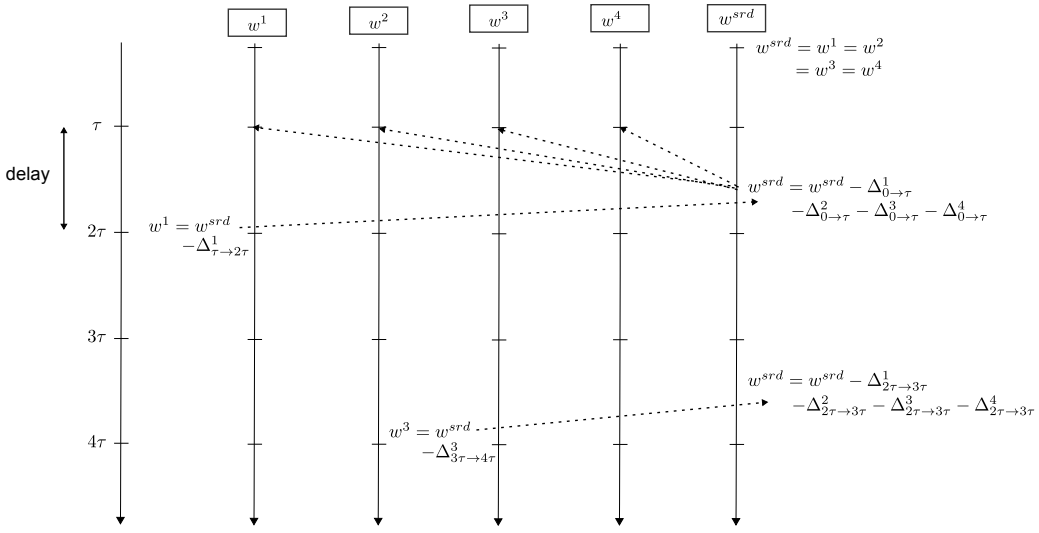

Figure 6.9: Illustration of the parallelization scheme described by equations (6.11). The reducing phase is only drawn for processor 1 where $t = 2\tau$ and processor 4 where $t = 4\tau$.

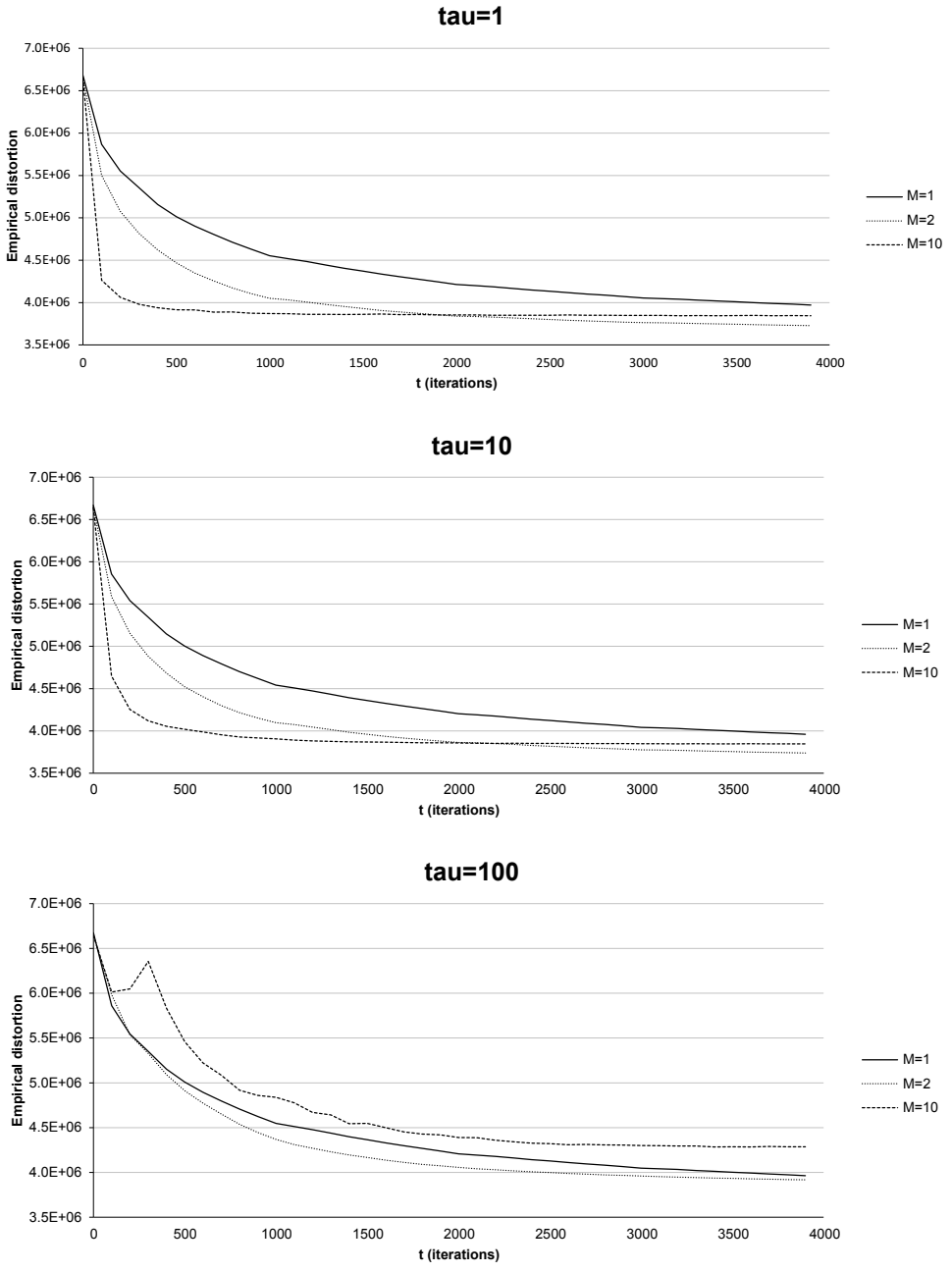

Figure 6.10: Charts of performance curves for iterations (6.11) with different numbers of computing entities, $M = 1, 2, 10$. The three charts correspond to different values of τ ($\tau = 1, 10, 100$).

6.4.4 Comments

The results presented in this chapter are rather insensitive to the experiment settings that we have chosen. In particular, the main conclusion is that averaging the results of multiple prototypes versions has not led to any speedup in our context —probably because of the non-convexity of our empirical distortion— but that the alternative parallelization scheme introduced in Subsection 6.4.2 provides such a speedup. This conclusion has proved to be verified in numerous experiment settings on which we have not reported. More specifically, this conclusion is true for multiple choices of the decreasing learning rate $(\varepsilon_t)_{t>0}$, for multiple choices of G, d, χ, κ, sc. In the same way, the introduction of small randomness and asynchronism between the communication windows in model (6.11) does not affect the conclusion.

Conversely, the choice of τ is of prime importance. While in the first two parallelization schemes, this value shows little impact on the speedup achieved, it becomes a decisive factor in the scheme presented by equations (6.11). Indeed, the use of stale information (or information with delays) leads to less effective schemes: such a conclusion has already been stated, for example in [28] or in [49]. More specifically, the degree of delay that is endurable without suffering noticeable performance loss must be set with regards to both the time to compute a gradient on a single point and the number of points to be processed for the scheme to approach convergence. As an example, the delays that will be observed in the following chapter are much higher than $\tau = 100$ but are compensated with much larger computation requirements before convergence.

Chapter 7

A cloud implementation of distributed asynchronous vector quantization algorithms

7.1 Introduction

The problem of the choice between synchronism or asynchronism has already been presented in Subsection 4.4.3 as an expression of the natural tradeoff between simplicity and efficiency (because of the stragglers mitigation). Resorting to asynchronism for cloud machine-learning algorithms therefore seems to be an important mechanism to improve the overall algorithms efficiency when tasks are coarse-grained. In the context of Cloud Computing platforms, the communication costs prompt us to design coarse-grained tasks, and therefore to investigate whether the asynchronism may be appropriate.

Beyond the stragglers mitigation, choosing asynchronism seems to improve the efficiency of some algorithms as this strategy allows the full usage of all the resources which are available. One can easily notice that distributed algorithms benefit from bandwidth throughput improvement, CPU cores load increase or apparent latency reductions. As a result, asynchronism may improve the usage ratio of all the available hardware components and as a consequence the algorithm behaviors by allowing overlaps of communication and computation.

The interest of asynchronism in scientific computations has already been investigated on multiple platforms: asynchronous algorithms have been analyzed on GPU hardware (see e.g. [42]), on heterogeneous clusters (see e.g. [27]) or on a multi-cores machine (see e.g. [71]). Many results on the theoretical conver-

gence of asynchronous stochastic gradient optimization algorithms have also been proved, especially by Tsitsiklis (see e.g. [106]) or more recently by Patra in [95]. To our knowledge, we provide in this chapter the first experimental results of asynchronous gradient-descent algorithms implemented on a Cloud Computing platform.

Two main difficulties arise from distributed asynchronous clustering algorithms. The first one is intrinsic to asynchronous algorithms, and mainly comes from the randomness thus introduced: a distributed asynchronous algorithm is not only producing different results from its sequential counterpart, but its returned result is also non-determinist, making the experiments hardly reproducible, contrary to Batch K-Means as outlined for example in [53]. In addition to this first difficulty, a specific challenge of our clustering problem consists in the non-convexity of our loss function (see equation (6.1)). This non-convexity leads to an additional difficulty: the natural update mechanism, suggested for example by [106] or [95] and tested in the fist parallelization scheme of Chapter 6 does not provide any speedup.

The algorithm used in this chapter therefore relies on an asynchronous version of the latter parallelization schemes of online gradient descent developed in Chapter 6. The adoption of an online paradigm allows the introduction of asynchronism. In addition, online gradient descent algorithms are deemed to provide faster rates of convergence than their batch counterparts (see e.g. [114]). This *a-priori* is confirmed at the end of this chapter: for the clustering context described in Section 7.4, our distributed asynchronous Vector Quantization (DAVQ) algorithm significantly outperforms Batch K-Means.

This good performance must however be assessed in terms of the development cost of our DAVQ prototype: as presented in the introduction to this document, the DAVQ project [1] is a shared project with Benoit Patra. While we were already familiar with the Azure application development, the development of this project took us months; moreover, while the distributed Batch K-Means experiments have required hardly more than 10,000 hours of CPU, the DAVQ project experiments took us more than 110,000 hours. In addition to this very costly development, the DAVQ algorithm proved to be bound in speedup: beyond a certain limit defined by the clustering context, an excess of workers led to unstable results (see the case $M = 16$ in Figure 7.4 or the case $M = 64$ in Figure 7.5).

Let us now briefly describe our DAVQ prototype. Like our distributed Batch K-Means algorithm, it is built on top of Azure. As already explained, our cloud

1. `http://code.google.com/p/clouddalvq/`

DAVQ implementation makes a great use of the results obtained in Chapter 6 where parallel implementations of DAVQ have been tested and discussed in general (i.e. independently of the distributed architecture). More particularly, the update mechanism based on the communication of the displacement terms is implemented. The DVQ models presented in Chapter 6 introduced non negligible delays that will picture essential aspects of the communication on the Cloud Computing platforms. Our DAVQ prototype is an open-source project released under the new BSD license and written in C#/.NET. The project is also built using the open-source project Lokad.Cloud [2] which provides an O/C (Object to Cloud) mapper and an execution framework that abstracts away low level technicalities of Windows Azure.

This chapter is organized as follows. Section 7.2 focuses on the software architecture of our DAVQ algorithm and many connections are drawn with Chapter 6. The study of the scalability of the DAVQ technique is presented in Section 7.3. The experiments are similar in spirit to those made in Section 6.4. More precisely, we investigate the scalable gain of time execution brought by DAVQ algorithms compared to a single execution of the VQ method. Finally, Section 7.4 is a benchmark between our DAVQ implementation and the distributed implementation of Batch K-Means developed in Chapter 5.

7.2 The implementation of cloud DAVQ algorithm

7.2.1 Design of the algorithm

The cloud implementation of our DAVQ algorithm is similar to the cloud implementation of Batch K-Means depicted in Chapter 5. In particular, it is based on two main QueueServices: the ProcessService and the ReduceService. These services are close to the MapService and ReduceService described in Chapter 5, but the MapService has been renamed ProcessService to highlight that this new design does not fit in an iterative MapReduce perspective anymore. The two services are based on the main idea introduced in Chapter 6 and according to which the averaging of concurrent local versions w^j's does not provide the convergence with any speedup. A better approach consists in computing the sum of all the displacement terms as shown in Subsection 6.4.2.

The core logic of our DAVQ implementation is embedded in the implementation of the ProcessService. Similarly to the cloud Batch K-Means implementation, we have multiple role instances running the ProcessService in parallel —referred to as

2. `http://lokad.github.com/lokad-cloud/`

M instances following the terminology used in Chapters 5 and 6— each instance performing a VQ execution. In parallel, the displacement terms introduced in Subsection 6.4.2, namely the $\Delta^j_{\cdot\rightarrow\cdot}$'s, are sent to the BlobStorage. The ReduceService is hosted by a dedicated worker (a single instance for now). Its task consists in retrieving the multiple displacement terms and computing the shared version of the prototypes, w^{srd}. This shared version is the common reference version retrieved by the multiple instances of the ProcessService which use it to update their local version. Figure 7.2 provides a synthetic overview of the "reducing task" assigned to the ReduceService, while Figure 7.1 gives a representation of the communication of the ProcessService instances and the ReduceService instance through the BlobStorage. The implementation of the ProcessService is more complicated than that of the ReduceService and is discussed in the next paragraph.

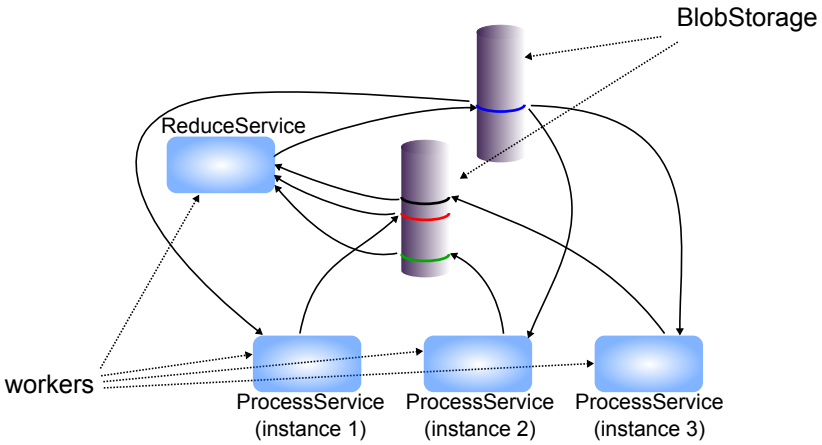

Figure 7.1: Overview of the interaction between ProcessService instances and the instance of ReduceService. All the communications are made through the BlobStorage: ReduceService gets the blobs put by the ProcessService instances while they retrieve the computation of the ReduceService.

The ProcessService is designed to benefit from the suppression of the synchronization process. As explained in the introduction to this chapter, in addition to the mitigation of stragglers which is an inter-machine mechanism, asynchronism allows us to adopt an intra-machine optimization: the overlap of communication and computation inside each processing unit. Indeed, each worker running a process task will compute local VQ iterations while it downloads the latest shared version from the BlobStorage and uploads a displacement term. The design of the communication and computation overlap of our ProcessService addresses two difficulties we have encountered. The first difficulty is related to a problem

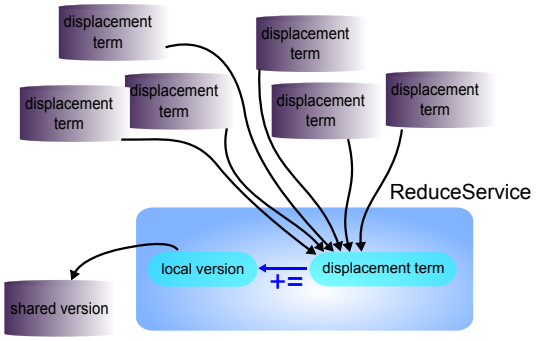

Figure 7.2: Overview of the ReduceService instance processing the "reducing task". This QueueService builds the shared version by computing the sum of all the available displacement terms sent by the ProcessService instances in the BlobStorage.

of compatibility between our prototype, the .NET framework, Azure and the Lokad.Cloud framework. During the first months of the development of our prototype, the Lokad.Cloud framework was built on top of the .NET framework 3.5; because of compatibility issues, we were not able to resort to the asynchronous I/O primitives of the .NET framework 4.0.

The second difficulty we encountered was related to the thread-handling mechanism of Azure. To our surprise, we noticed after many attempts that we could not manage to obtain both satisfactory I/O bandwidth and satisfactory Flops with an automatic thread management. In many cases, the Flops obtained were close to the Flops obtained without handling parallel communications but the I/O bandwidth was 10 times lower than if we had only been performing an I/O operation. The exact reason of this disappointing result has not been clearly isolated but probably comes from an unsatisfactory behavior of the thread scheduler inside a VM. The solution to circumvent this difficulty is presented below. The following paragraph describes how each process task is organized in multiple threads run in parallel.

The ProcessService performs simultaneously three functions using the following threads: a process thread, a push thread, and a pull thread. The process thread is a CPU-intensive thread while the push thread and the pull thread are dedicated to the communications with the BlobStorage. Let us lay the emphasis on the fact that this parallelization is made inside the VM, at CPU level. The multi-threaded execution of the ProcessService (in its latest version) is based upon the Task

Parallel Library provided in the .NET framework 4.0 (see for instance Freeman [54]). More precisely, the pull thread is responsible for providing the latest shared version to the process thread. The pull thread uses the BlobStorage's timestamps (etags) to detect new shared versions sent by the ReduceService and made available. The pull thread and the process thread are communicating through a classical producer/consumer design pattern, implemented through a read buffer built upon the .NET 4.0 BlockingCollection class which is of great help to the read buffer to manage concurrency issues. The push thread is responsible for pushing to distant blobs the displacement term updates computed by the process thread. The interaction of the push thread and the process thread also uses a Producer/Consumer design pattern with a communication buffer (referred to in the following as write buffer). However, in this context, the process thread is the producer and the push thread is the consumer.

As described above, the automatic scheduling of these three threads led to poor I/O bandwidth that severely deteriorated the communication behavior and therefore the consensus agreement between the multiple processing unit running the ProcessService. To avoid this, we forced the process thread to sleep during very short period so that it could return control to the thread scheduler thus allowing the two I/O threads to gain more CPU time. This method required a fine tuning of the sleep parameters that depends both on the time to communicate a displacement term and on the time to compute a single gradient estimation. We did not manage to find a cleaner method to regulate the threads balance.

The blobs pushed by the push thread are then consumed by the ReduceService: this QueueService is notified by messages in its queue that new displacement terms can be used for the shared version updates. Figure 7.3 illustrates the multithreaded logic of the ProcessService.

7.2.2 Design of the evaluation process

In the experiments depicted in Section 7.3 and Section 7.4, the performance of clustering algorithms could have been measured during their executions. However, the algorithms should not be slowed down by their performance measurements. Consequently, our DAVQ algorithm uses the fact that the BlobStorage has been designed to cope with multiple reading actions. A new QueueService is implemented: the SnapshotService and a dedicated worker is deployed to host it. The SnapshotService keeps making deep copies of the shared version blob —the snapshots— and stores them in other persistent blobs. Let us keep in mind that the blob containing the shared version is constantly modified by the ReduceService.

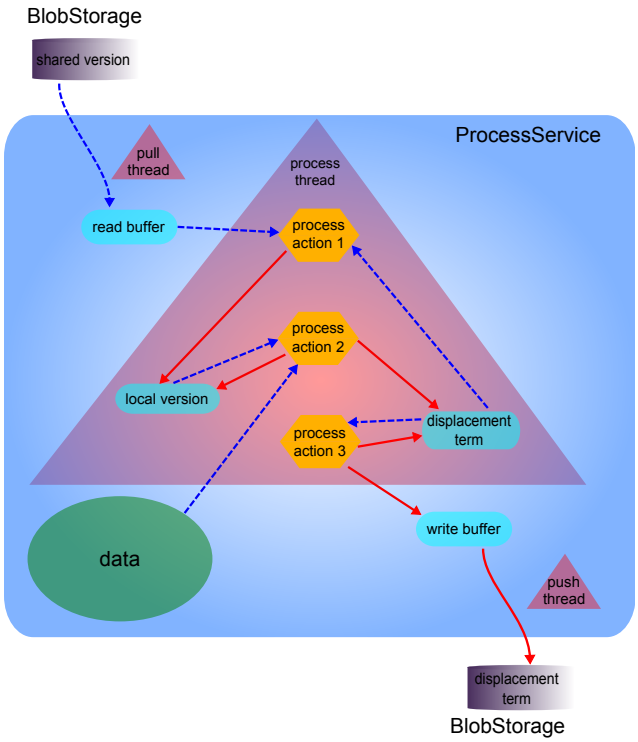

Figure 7.3: Overview of the ProcessService. Each triangle stands for a specific thread (process thread, push thread and pull thread). The arrows describe read-/write actions: the tail of a blue dashed arrow is read by the entity at its head and the entity at the tail of a red solid arrow makes update on the entity that lies at its head. The push thread and the pull thread enable communications between the process thread and the BlobStorage. The process thread alternatively performs three actions (process action $1, 2, 3$). Process action 1 replaces the local version of the prototypes by the sum of the latest shared version (kept in the read buffer) and a displacement term. Process action 2 uses data to execute VQ iterations and updates both the local version and the displacement term. Process action 3 moves the displacement term to a dedicated buffer (write buffer) and pushes its content to the BlobStorage.

Therefore it is necessary to make copies to track back its evolution. Once the multiple ProcessService instances stop processing VQ iterations, i.e. when the stopping criterion is met, then the evaluation processus starts: all the computing instance that previously ran the ProcessService or the ReduceService now run the EvaluationService (another QueueService); for each snapshot, one of the computing units compute the corresponding empirical distortion given by equation

(6.1) (or its normalized version (6.5)). In order to perform the evaluations of the snapshots the same synthetic data are generated again by using new instances of random generators. Therefore, no data is broadcast through the network that would have been a massive bottleneck for the evaluations of the algorithms. This task is still highly CPU consuming: indeed, the evaluation of a single snapshot requires delays that are comparable to a whole iteration of Batch K-Means or to all the VQ iterations performed by a ProcessService instance. However, thanks to the elasticity of the VM allocations, it can be completed within reasonable delays.

7.3 Scalability of our cloud DAVQ algorithm

7.3.1 Speedup with a 1-layer Reduce

In this subsection our objective is to prove that our DAVQ algorithm brings a substantial speedup which corresponds to the gain of wall time execution brought by more computing resources compared to a sequential execution. Consequently the experiments presented in this subsection are similar to those provided in Chapter 6. In the previous chapter, we proved that a careful attention should be paid to the parallelization scheme, but without any specifications on the architecture. We should bear in mind that the experiments presented in Chapter 6 simulated a generic distributed architecture. In this subsection we present some experiments made with our true cloud implementation.

The following settings will attempt to create a synthetic but realistic situation for our algorithms. We load the local memory of the workers with data generated using the splines mixtures random generators defined in Section 6.3. Using the notation introduced in this section, each worker is endowed with $n = 10000$ vector data that are sampled spline functions with $d = 1000$. The benefit of using synthetic data for our implementations has been discussed in Section 6.3. The number of clusters is set to $\kappa = 1000$ while there are $G = 1500$ centers in the mixture and the number of knots for each spline χ is set to 100. The model made in Chapter 5 has shown that it is not possible to tune the relative amount of time taken by a gradient computation compared to the amount of time taken by prototypes communication. Indeed, both of them linearly depend on κ and d. The chosen settings ensure that the complexity of the clustering job would lead to approximately an hour execution on a sequential machine when using the same learning rate as in the previous chapter: $\varepsilon_t = \frac{1}{\sqrt{t}}$ for $t > 0$.

Similarly to Section 6.4, we investigate the speedup ability of our cloud DAVQ algorithm. Let us keep in mind that the number of samples in the total data set

equals to nM. Thus, this number is varying with M (the number of ProcessService instances). Consequently, the evaluations are performed with the normalized quantization criterion given by equation (6.5).

Figure 7.4 shows the normalized quantization curves for $M = 1, 2, 4, 8, 16$. We can notice that our cloud DAVQ algorithm has a good scalability property for $M = 1, 2, 4, 8$: the algorithm converges more rapidly because it has processed much more points as shown at the end of the section. Therefore, in terms of wall time, the DAVQ algorithm with $M = 2, 4, 8$ clearly outperforms the single execution of the VQ algorithm ($M = 1$). However we can remark the bad behavior of our algorithm with $M = 16$. This behavior is explained in the subsequent subsection and a solution to overcome the problem is presented and evaluated.

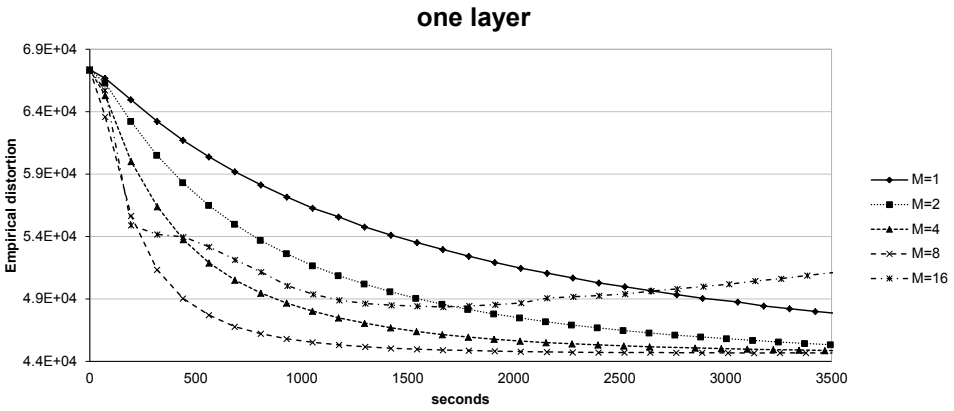

Figure 7.4: Normalized quantization curves with $M = 1, 2, 4, 8, 16$. Our cloud DAVQ algorithm has good scalability properties up to $M = 8$ instances of the ProcessService. Troubles appear with $M = 16$ because the ReduceService is overloaded.

7.3.2 Speedup with a 2-layer Reduce

The performance curve with $M = 16$ of Figure 7.4, shows an unsatisfactory situation where the algorithm implementation does not provide better quantization results than the sequential VQ algorithm. Actually, the troubles arise from the fact that the ReduceService is overloaded [3]. Indeed, too many ProcessService

3. The Lokad.Cloud framework provides a web application that provides monitoring tools, including a rough profiling system and real time analysis of queues length, which helped us to

instances communicate their results (the displacements terms) in parallel and fill the queue associated to the ReduceService with new tasks resulting in a ReduceService overload. Once again the elasticity of the cloud enables us to allocate more resources and to create a better scheme by distributing the reducing tasks workload on multiple workers. As we did for Batch K-Means, we now set an intermediate layer in the reducing process by replacing the ReduceService by two new QueueServices: the PartialReduceService and the FinalReduceService. We deploy $\sqrt{M}$ instances of PartialReduceService that will be responsible for computing the sum of the displacement terms, but only for a certain fraction of ProcessService instances: each PartialReduceService instance is gathering the displacement terms of $\sqrt{M}$ ProcessService instances in the same way as for Batch K-Means. Then, the unique instance of FinalReduceService retrieves all the intermediate sums of displacement terms made by the PartialReduceService instances and computes the new shared version. Figure 7.6 is a synthetic representation of the new layers, which are introduced to remove the difficulty brought by the overload of the initial ReduceService. In Figure 7.5 we plot the performance curves with PartialReduceService instances. We can remark that a good speedup is obtained up to $M = 32$ ProcessService workers. However, the algorithm does not scale-up to $M = 64$.

Contrary to the phenomenon observed when using a single-layer Reduce, the queues were not overloaded in the case $M = 64$. This unsatisfactory result highlights again the sensitivity of VQ algorithms to the learning rate $\{\varepsilon_t\}_{t>0}$ and to the first iterations: we have observed in some of these unsatisfactory experiments that the aggregated displacement term was too strong when M was exceeding a threshold value. Beyond this threshold, the aggregated displacement term leads indeed to an actual shared version of the prototypes that oscillate around some value, while moving away from it. These issues may be solved by adapting the learning rate to the actual value of M for the first iterations, when M exceeds a specific threshold.

7.3.3 Scalability

Let us recall that the scalability is the ability of the distributed algorithm to process growing volumes of data gracefully. The experiments reported in Figure 7.7 investigate this processing ability with more accuracy, by counting the samples that have been processed for the computation of the shared version. The graph shows the results with one or two layers for the reducing task and with different

note that the ReduceService became overloaded when $M = 16$

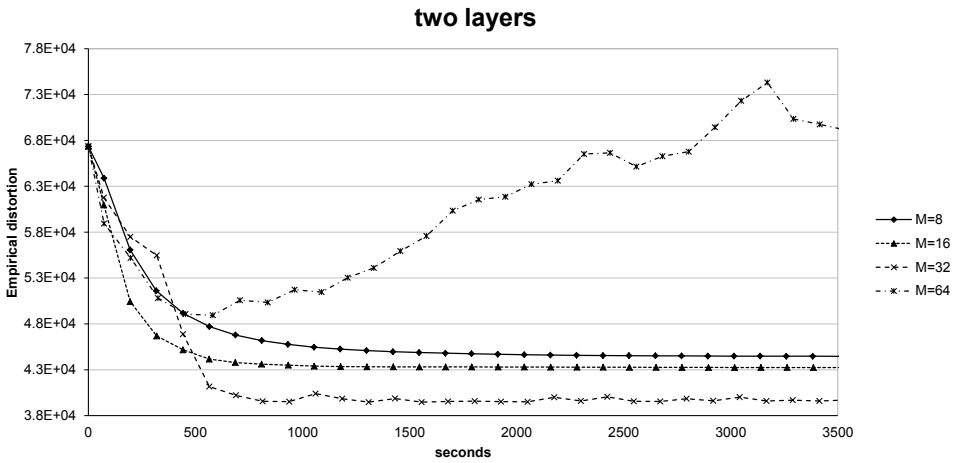

Figure 7.5: Normalized quantization curves with $M = 8, 16, 32, 64$ instances of ProcessService and with an extra layer for the so called "reducing task". Our cloud DAVQ algorithm has good scalability properties for the quantization performance up to $M = 32$. However, the algorithm behaves badly when the number of computing instances is raised to $M = 64$.

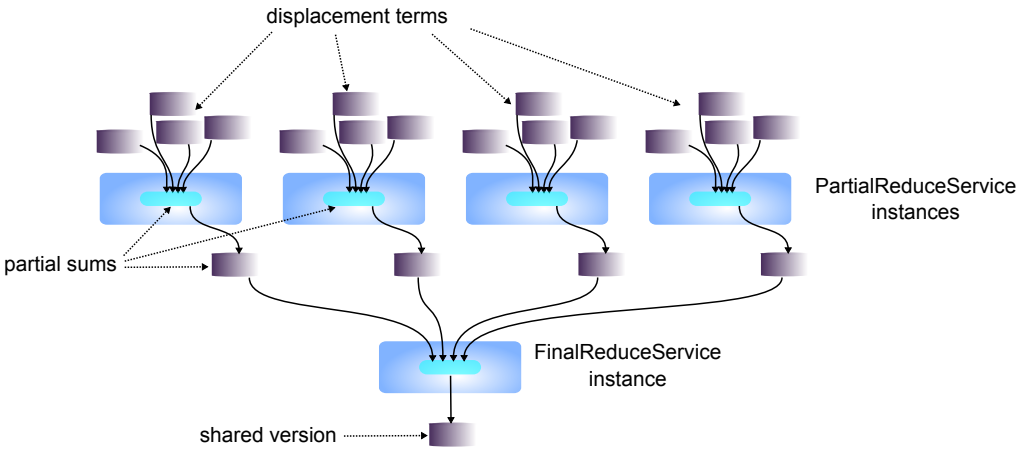

Figure 7.6: Overview of the reducing procedures with two layers: the PartialReduceService and the FinalReduceService.

values of M, namely $M = 1, 2, 4, 8$ for the first case and $M = 8, 16, 32, 64$ for the second one. The curves appear linear, which proves that the algorithms behave well. The slopes of the multiple curves, characteristic of the processing ability

of the algorithm, exhibit a good property: the slope is multiplied by 2 when the number of processing instances M is multiplied by 2. This proves that the algorithm has a good processing scalability as we had hoped for. This is confirmed by Figure 7.8, showing a linear curve for the number of points processed in a given time span (3600 seconds) using different computing instances M. We can remark that the scalability in terms of data processed is still good up to $M = 64$ while the algorithm does not work well in terms of accuracy (i.e. quantization level). Furthermore, the introduction of the second layer does not slow down the execution even for intermediate values of M ($M = 8$) as proved by the charts displayed in Figure 7.7.

7.4 Competition with Batch K-Means

The experiments provided in the previous section prove the benefit of using our cloud DAVQ algorithm for large scale quantization jobs compared to the usual VQ algorithm. However, such experiments do not prove that our cloud DAVQ algorithm is a good clustering method in-itself. In this section we present a comparison between our cloud DAVQ algorithm and distributed Batch K-Means, which is a standard clustering method. More precisely, we compare our cloud DAVQ algorithm with the Azure implementation of the distributed Batch K-Means developed in Chapter 5. While we do not aim to compare the performances of the two algorithms in a general context (the difficult task of comparing the sequential VQ algorithm with the sequential Batch K-Means algorithm has already been vastly investigated, for example in [114] or in [53]), we provide in this section a context in which cloud DAVQ seems to be a good clustering method as it clearly outperforms Batch K-Means. We work with a fixed large synthetic data set and our goal is to provide a summary with a reduced set of prototypes. The two algorithms are compared with the evolution of the quantization error using intermediate values of the prototypes that are observed through the SnapshotService.

The number of mappers in the cloud Batch K-Means is the number of processing units. Therefore it is natural to compare our cloud DAVQ and Batch K-Means with the same data set and the same number of ProcessService instances and mappers M. The local memory of a worker is loaded with $n = 500,000$ vector data whose dimension d is set to 100. The number of clusters is set to $\kappa = 5,000$ while the number of centers in the mixture G equals $7,500$ and the number of knots for each spline χ equals 100. We set the learning rate ε_t (see Section 6.2) to a constant value $\varepsilon = 10^{-2}$. In all our previous experiments, we had supposed that a satisfactory value of the learning rate had already been chosen in the form of the decreasing sequence $\varepsilon_t = \frac{1}{\sqrt{t}}$ with $t > 0$ and compared the DVQ or DAVQ im-

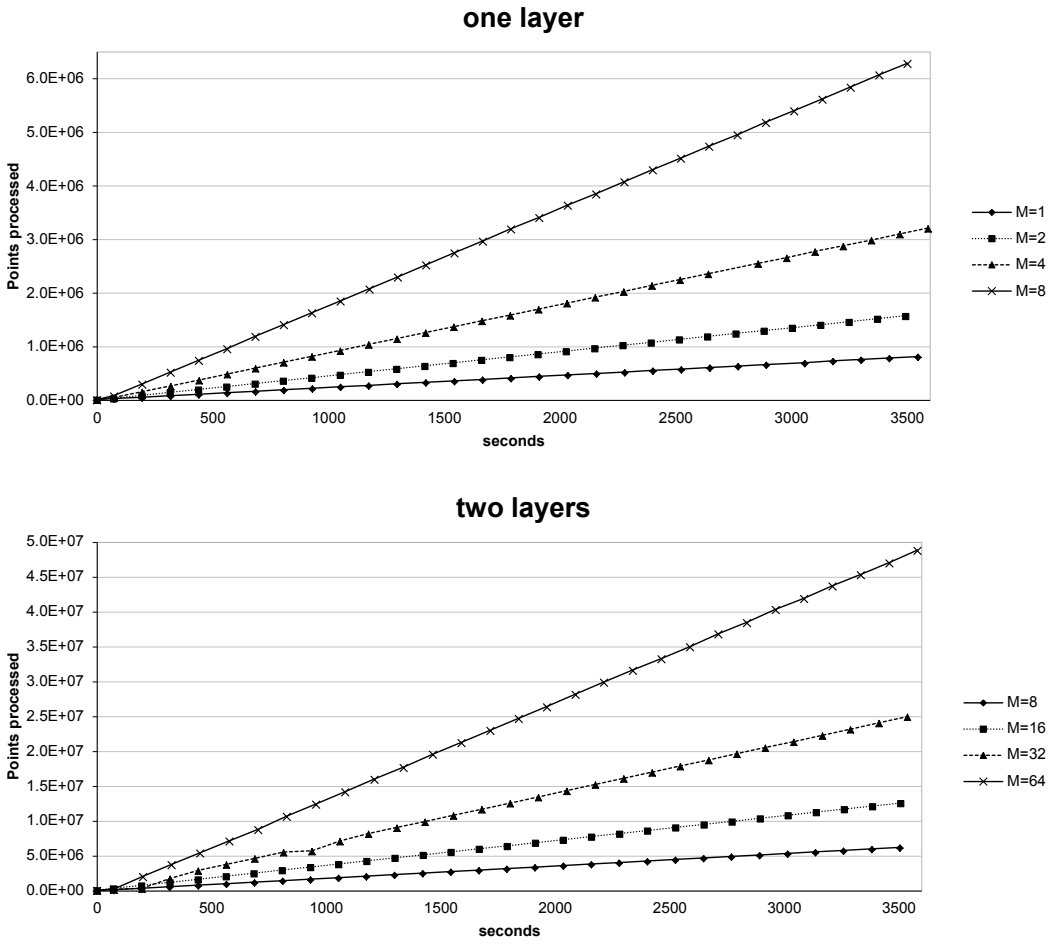

Figure 7.7: These charts plot the number of points processed as time goes by. The top chart shows the curves associated to $M = 1, 2, 4, 8$ with one layer for the reducing task whereas at the bottom a second layer is added and $M = 8, 16, 32, 64$.

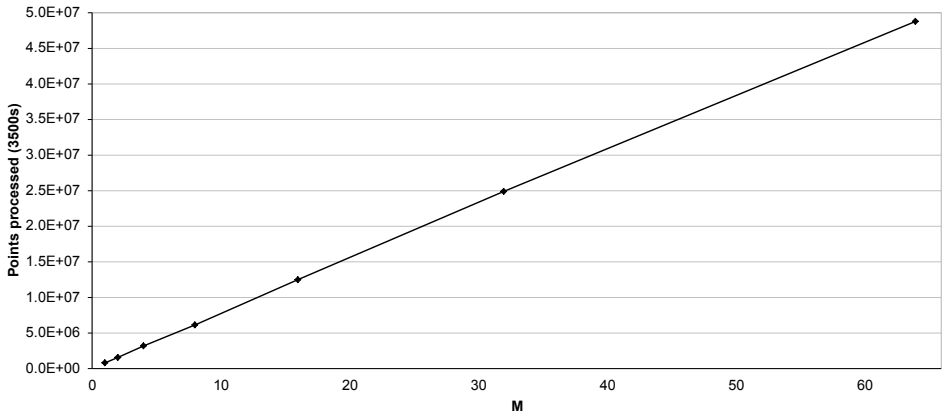

Figure 7.8: This chart plots the number of points processed in a given time span (3600 seconds) for different values of M ($M = 1, 2, 4, 8, 16, 32, 64$). The reducing task is composed of one layer for $M = 1, 2, 4$ and two layers for $M = 8, 16, 32, 64$. We can observe a linear scalability in terms of point processed by the system up to $M = 64$ computing instances.

plementations with the sequential VQ iterations. In this experiment, the constant value of the learning rate $\varepsilon = 10^{-2}$ is made comparable to the corresponding learning rate of Batch K-Means when seen as a gradient descent algorithm (see Bottou and Bengio in [32]). We have not made any optimization of the learning rate, which is therefore probably not optimal.

Figure 7.9 shows the curves of the clustering performance for the two procedures which are in competition. We report there on three experiments with various sizes of data sets nM ($M = 8, 16, 32$). The quantization curve of the cloud Batch K-Means corresponds to a step function. Indeed, the reference version used for the Batch K-Means evaluations is the version built during the synchronization phase and is modified only there. We can see that our cloud DAVQ algorithm clearly outperforms the cloud Batch K-Means. In all cases ($M = 8, 16, 32$) the VQ algorithm takes less time to reach a similar quantization level. Therefore, this implementation seems to be a competitor to existing solutions for large scale clustering jobs, thanks to its scalability and its mathematical performance.

The following tables reformulate the results provided in Figure 7.9. For each value of M, we provide the amount of time spent by the DAVQ procedure to obtain the same quantization loss levels than the ones obtained for the four first iterations of Batch K-Means. One can notice that in these experiments, for a given

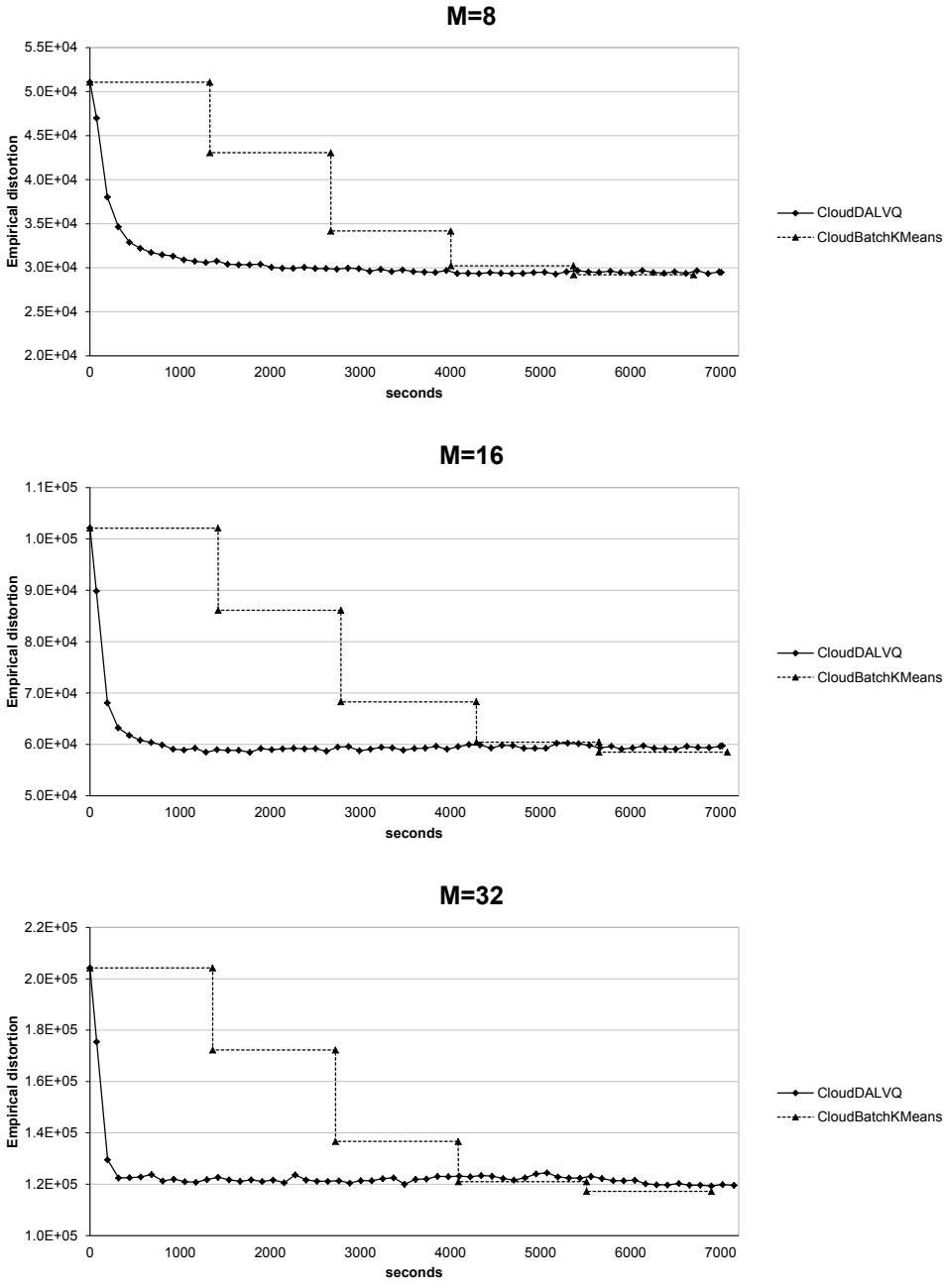

Figure 7.9: These charts report on the competition between our cloud DAVQ algorithm and the cloud Batch K-Means. The graphs show the empirical distortion of the algorithm over the time. The empirical distortion is computed using the shared version for our cloud DAVQ algorithm while it is computed during the synchronization phase (all mappers receiving the same prototypes) for the cloud Batch K-Means. In these experiments the cloud DAVQ algorithm outperforms the cloud Batch K-Means: the same quantization level is obtained within a shorter period.

value of quantization, the DAVQ procedure may return the results as quickly as 14 times faster than the Batch procedure would.

Iteration	0	1	2	3	4
Batch duration (in sec.) (M=8)	0	1383	2679	4019	5401
DAVQ duration (in sec.) (M=8)	0	132	346	2031	5256

Iteration	0	1	2	3	4
Batch duration (in sec.) (M=16)	0	1426	2765	4278	5617
DAVQ duration (in sec.) (M=16)	0	130	198	665	2981

Iteration	0	1	2	3	4
Batch duration (in sec.) (M=32)	0	1400	2722	4087	5505
DAVQ duration (in sec.) (M=32)	0	86	181	803	?

Table 7.1: Evolution of the empirical distortion for distributed Batch K-Means and our VQ algorithm. The tables report how much time Batch K-Means requires to complete a specific number of iteration, and how much time it takes to our DAVQ implementation to achieve the same level of empirical distortion.

Conclusion

Nous avons montré dans les premiers chapitres de cette thèse que le Cloud Computing est une offre protéiforme qui répond à un besoin grandissant du monde économique en ressources informatiques, qu'elles soient logicielles ou matérielles. Durant nos trois années de thèse, nous nous sommes appliqués à bâtir des projets logiciels sur l'environnement matériel et logiciel fournit par une plateforme PaaS spécifique, celle d'Azure. A la lumière de ces projets, nous nous proposons maintenant de donner notre vision d'Azure en tant que plateforme de calcul, puis de rappeler nos conclusions présentées dans les chapitres 5, 6 et 7 quant à la parallélisation d'algorithmes de clustering.

De l'utilisation du PaaS comme plateforme de calcul intensif

Constat technique

Le premier constat que nous pouvons porter sur Azure est celui d'une plateforme qui tient ses promesses en termes de garantie de performance, au moins jusqu'à 200 machines. En effet, les résultats que nous obtenons sont comparables aux valeurs nominales présentées par Microsoft sur le site AzureScope, site aujourd'hui supprimé. Ainsi, nous avons bien retrouvé les puissances de calcul que nous nous attendions à obtenir par rapport à la taille des VM que nous avions louées, et les performances du BlobStorage que nous avons observées concordent également avec celles qui étaient déclarées par Microsoft. Enfin, la contrainte sur la bande passante agrégée est bien d'environ 850 Mégaoctets/sec, comme déclaré par Microsoft.

Cependant, ces performances sont loin d'égaler celles que nous pourrions obtenir avec un nombre équivalent de CPU ou de GPU sur un cluster de machines en local. Toutes choses égales par ailleurs, le cloud semble peu efficace dans le sens

où il ne tire pas un profit optimal des ressources matérielles sous-jacentes : la virtualisation [4], les communications moins efficaces que dans un réseau local, et le manque actuel de contrôle dans la fabrique d'Azure sur la topologie des machines qui nous sont allouées, apportent des limites physiques sur la quantité de calcul que nous pouvons effectuer sur cette plateforme, à nombre de machines fixé. Ce constat est particulièrement vérifié quand le nombre de processeurs impliqués est faible : lorsque nous souhaitons implémenter un algorithme sur 16 unités de calcul, il est bien plus simple et performant d'utiliser une seule machine pourvue de 16 coeurs, dont le coût est très accessible. Lorsque nous augmentons le nombre de processeurs que nous souhaitons utiliser, les solutions disponibles se raréfient, et le cloud devient alors d'autant plus compétitif. Nous pouvons donc raisonnablement prendre le pari que les cas d'applications du Cloud Computing comme plateforme de calcul intensif impliqueront des projets nécessitant au moins une centaine d'unités de calcul.

Un autre aspect intéressant des limitations actuelles d'Azure réside dans le manque d'environnements logiciels adaptés, qu'ils aient trait à l'exécution générale ou à la communication. Le PaaS est réputé plus simple d'utilisation que le IaaS en raison des primitives supplémentaires qu'il propose. En effet, les "briques élémentaires" du PaaS sont de plus haut niveau que celles proposées dans les solutions IaaS. Cependant les solutions IaaS proposent aujourd'hui des implémentations éprouvées d'environnements logiciels comme MapReduce ou MPI. Ces environnements, qui simplifient bien plus la tâche de développement que les primitives de plus haut niveau fournies par le PaaS, sont encore en cours de développement pour les solutions PaaS. Ainsi, plusieurs projets d'implémentation de MapReduce pour Azure ont été proposés, notamment celui de Microsoft Research —appelé projet Daytona— dont la première version a été mise à disposition du public en juillet 2011. Ce projet, qui est à l'heure où nous écrivons ces lignes, la version la plus aboutie des implémentations de MapReduce sur Azure, est encore à l'état de prototype et n'a pas été intégré à l'offre commerciale d'Azure. Paradoxalement, ce sont donc les offres IaaS qui semblent fournir à l'heure actuelle les solutions les plus simples pour effectuer des calculs intensifs.

Nous identifions cependant quatre cas d'utilisation (qui ne s'excluent pas mutuellement) dans lesquels le Cloud Computing est déjà compétitif techniquement pour réaliser des calculs intensifs. Le premier de ces quatre cas a trait à un besoin en calcul volatile : puisqu'Azure est à la fois élastisque et agnostique à la charge en calcul, le nombre de machines disponibles peut être sous quelques minutes redimensionné pour s'adapter à un pic de demande, ou réciproquement à une

4. Que ce soit via la machine virtuelle .NET ou la virtualisation de l'OS hébergée sur le cloud.

baisse temporaire de celle-ci. Un second cas d'utilisation d'Azure est incarné par les tâches de calcul requérant plusieurs centaines de machines ; dans ce cas, moins de solutions sont facilement accessibles, et Azure devient donc par contraste plus compétitif. Une troisième catégorie de tâches sont celles ne requérant pas des communications importantes ou des latences faibles entre les machines, deux des points faibles d'Azure par rapport à un cluster local de machine. Enfin, et c'est à nos yeux une catégorie très importante de tâches, de nombreux calculs intensifs sont réalisés dans un contexte de production, c'est à dire dans un cadre où la robustesse est primordiale. Azure fournit un environnement de calcul dans lequel le remplacement d'une machine morte est réalisé automatiquement dans un délai très bref, et dans lequel la perte d'une telle machine affecte peu le temps total d'exécution d'une tâche de calcul.

Les quatres contextes qui viennent d'être évoqués sont souvent tous réunis dans le cas d'enjeux industriels. C'est le cas par exemple pour le coeur technologique de Lokad, la société dans laquelle j'ai réalisé cette thèse. En effet, ce coeur technologique fournit une API qui permet d'exécuter des prévisions sur un ensemble de séries temporelles. Dans ce contexte, les garanties de robustesse et d'élasticité du cloud sont primordiales. Par ailleurs, une partie importante des calculs présentant un cas de parallélisme sur les données (dit de « data-level parallelism »), ces tâches de prévision demandent peu de communication entre les différentes unités de calcul. Enfin, les volumes de données, ainsi que la complexité algorithmique de ce coeur technologique requièrent d'allouer chaque jour plusieurs fois entre 100 et 200 machines, mais ce coeur technologique n'est pas utilisé une majorité du temps, et utilise donc dans ces circonstances une seule VM (c'est à dire le minimum). Pour Lokad, Azure est donc une excellente solution technologique à ses besoins de performance, de robustesse et d'élasticité.

Constat économique

Au-delà des divers éléments techniques qui viennent d'être présentés, nous sommes convaincus que l'adoption du cloud en général et du PaaS en particulier se joueront probablement plus sur des considérations économiques et stratégiques que techniques. A ce titre, trois éléments doivent être pris en compte : le coût de développement de l'application d'une part (c'est à dire le coût du software), et le coût du hardware d'autre part (achat amorti ou location, consommation énergétique, administration, entrepôt, etc.).

En ce qui concerne le développement d'applications très intensives en calcul, Azure n'est pas encore compétitif en raison du manque actuel d'environnement

logiciel disponible. Nous avons déjà constaté que ce manque fait d'Azure une plateforme sur laquelle le développement de ces applications est rendu long et délicat. Ce coût de développement peut aujourd'hui se révéler trop important, puisque le coût horaire d'un ingénieur compétent peut représenter l'équivalent du coût horaire de plus d'un millier de machines sur des plateformes comme Azure ou Amazon. Microsoft a cependant annoncé qu'une implémentation commerciale de MapReduce pour Azure est en cours de développement. Alors qu'une première version d'une adaptation de Dryad sur Azure a été développée et lancée, Microsoft a abandonné le projet pour se concentrer sur une implémentation officielle de MapReduce sur Azure, plus simple à manipuler et donc plus accessible pour ses clients que Dryad.

L'estimation du coût moyen d'usage du matériel physique ne fait quant à elle pas l'objet d'un consensus. En effet, certaines incertitudes demeurent sur les tarifs à long terme que proposeront Amazon ou Azure. En effet, ces tarifs ont été maintes fois diminués au cours des dernières années (ainsi certains des prix d'Amazon ont fait l'objet de 19 réductions successives sous différentes formes depuis 6 ans), et devraient probablement encore l'être à l'avenir.

Au delà de ces incertitudes à la fois sur les coûts de développement et sur les coûts d'utilisation, Azure peut-il se révéler une solution économiquement compétitive pour réaliser du calcul intensif ? En d'autres termes sera-t-il plus rentable d'utiliser une plateforme cloud comme Azure par rapport à un cluster local de machines, des GPU, du matériel reprogrammable comme des Field-Programmable Gate Array (FPGA) ou des Application Specific Integrated Circuit (ASIC) ? En ce qui concerne les FPGA et les ASIC, leur marché est en sensible croissance, mais l'utilisation de ce type de solutions entraine souvent des coûts de développement qui rendent leur utilisation si ce n'est situationnelle, du moins peu adaptée dans de nombreux cas. Puisque de récentes offres de cloud (comme celles d'Amazon) proposent maintenant des GPU, la vraie question de la compétitivité économique du cloud est donc celle de la performance relative du cloud vis-à-vis d'un cluster local de machines, contenant des CPU et/ou des GPU.

Sur ce dernier point, l'analogie entre Cloud Computing et Assurance peut apporter des éléments de réponse. De la même manière que les mécanismes d'assurance fournissent une mutualisation des risques pour lisser en espérance les coûts liés à des accidents, les mécanismes de cloud fournissent une mutualisation des ressources pour lisser dans le temps les besoins en consommation de multiples clients, et ainsi transformer l'informatique en un bien de production courant comme l'électricité. De nombreuses organisations ont des besoins fluctuants en calcul mais n'ont pas la taille critique au delà de laquelle les besoins internes

sont suffisants pour que la mutualisation s'opère sans recourir à une organisation extérieure. Gageons alors qu'au moins pour ces organisations, les solutions cloud sont ou pourront être pertinentes.

Implémentation d'algorithmes de clustering répartis

Cloud Batch K-Means

La parallélisation du Batch K-Means est un problème bien maitrisé dans le cas d'architectures Symmetric Multi-Processors (SMP) ou Distributed Memory Multi-processors (DMM), à tel point qu'il fait parfois figure de cas d'école pour le développement d'algorithmes de machine-learning parallèle. En effet, son schéma de parallélisation est évident et son efficacité est excellente à condition que les coûts de communication soient faibles. Sur des architectures SMP ou sur des architectures DMM pourvues de MPI, ces coûts sont effectivement très faibles. À l'inverse, les coûts de communication sur Azure rendent cette parallélisation moins efficace. Même si nous avons obtenu des accélérations de convergence qui avaient été atteintes précédemment dans peu de travaux académiques [5], nous avons aussi montré que l'utilisation des ressources était peu efficace.

Plus précisément, le synchronisme de l'algorithme du Batch K-Means réparti met en lumière deux phénomènes connus en calcul réparti, mais qui semblaient peu illustrés du côté de la communauté machine-learning. Le premier de ces phénomènes est l'aspect crucial que peuvent revêtir des ralentissements non-anticipés sur certaines machines, un phénomène dit de « stragglers » [6]. L'aspect synchrone de l'algorithme du Batch K-Means réparti implique en effet que le comportement général de l'algorithme est déduit non pas du comportement moyen des machines mais du comportement de la "pire" de celles-ci.

Le second phénomène est celui de l'importance des débits dans les communications. Puisque le synchronisme ne permet pas dans le cas du Batch K-Means de recouvrir les temps de communication et ceux de calcul, le coût des premiers doit être ajouté à celui des seconds. En l'absence d'un environnement logiciel efficace pour prendre en charge ces communications, nous avons utilisé le sto-

5. avec certaines exécutions 58 fois plus rapides que leur pendant séquentiel exécuté sur une machine avec la même puissance mais une RAM infinie

6. phénomène qui n'est sensible que sur des architectures matérielles peu intégrées comme celles d'un cloud

ckage d'Azure comme moyen de transférer des données entre les machines. Dans ce contexte, les coûts de communication ne peuvent plus être négligés. Nous pouvons montrer qu'un mécanisme d'aggrégation des résultats des M différentes unités de calcul en p étapes conduit à un coût de communication en $O(p\sqrt[p]{M})$. Cependant, les latences internes à Azure et les fréquences de ping des queues empêchent de concevoir des mécanismes efficaces d'aggrégation en $p = log(M)$ étapes comme c'est le cas dans MPI. Nous avons analysé le modèle correspondant au mécanisme d'aggrégation en deux étapes que nous avons retenu pour notre implémentation. Ce modèle de coût permet notamment d'anticiper l'ordre de grandeur du nombre optimal de machines à utiliser pour minimiser le temps d'exécution de notre Batch K-Means réparti sur Azure.

Cloud DAVQ

Les ralentissements non-anticipés (les « stragglers ») et les coûts de communication qui viennent d'être rappelés nous ont donc incités à orienter nos travaux vers des algorithmes asynchrones de clustering. Pour le Batch K-Means, c'est le caractère « Batch » de l'algorithme qui entraine le synchronisme : l'étape de recalcul des prototypes nécessite que les communications soient réalisées exactement une fois par itération. Le passage vers un algorithme « en-ligne »—l'algorithme de Vector Quantization (VQ)— permet donc de supprimer ce point de synchronisation évidente.

La parallélisation de l'algorithme de VQ sur le cloud présente deux difficultés majeures : la non-convexité de la fonction de perte et la latence des écritures d'un objet (dans notre cas situé dans le BlobStorage). De nombreux travaux ont été proposés pour répondre à la question de la descente de gradient parallèle en présence d'une de ces difficultés mais aucun à notre connaissance ne proposait de réponse lorsque les deux difficultés sont réunies.

Lorsque les latences des écritures sont faibles par rapport au coût de calcul du gradient, il est possible de ne posséder qu'une seule version du paramètre à optimiser (dans notre cas les prototypes qui résument les données). Ainsi, certains travaux comme [84] ou [50] proposent des schémas de parallélisation dans lesquels différents threads executés sur une seule machine multi-coeurs accèdent en écriture chacun leur tour au paramètre pour le modifier (on parle alors d'entrelacement des écritures). Comme souligné dans les conclusions de [50] ou dans [28], ce mécanisme d'entrelacement n'est possible que sur une architecture matérielle avec mémoire partagée et pour un nombre faible de coeurs.

Sur le cloud, les temps d'écriture dans le BlobStorage sont trop importants par rapport au temps de calcul du gradient pour utiliser de telles techniques. Chaque unité de calcul, ici des machines virtuelles potentiellement distantes, possède donc sa propre version du paramètre (i.e. des prototypes) qu'elle modifie localement au fur et à mesure qu'elle examine des points issus du jeu de données sur lequel elle travaille. La difficulté de la parallélisation réside alors dans le choix d'une stratégie d'utilisation de ces différentes versions pour obtenir une version « meilleure ».

La présentation dans les chapitres 6 et 7 de notre travail de parallélisation de l'algorithme de VQ, désigné dans la suite par Distributed Asynchronous Vector Quantization (DAVQ), reprend à rebours la chronologie de notre travail. En effet, inspirés par les résultats présentés dans [106], [95] ou [49], nous avons tout d'abord implémenté sur Azure une version répartie asynchrone de l'algorithme de VQ dans laquelle les différentes versions des prototypes sont moyennées de manière asynchrone. À notre surprise les premiers résultats ne fournissaient pas d'amélioration par rapport à l'algorithme séquentiel. Nous avons pendant longtemps cherché à comprendre ce résultat que nous mettions initialement sur le compte de spécificités d'Azure ou de réglages inhérents aux algorithmes stochastiques, notamment le réglage de la décroissance du pas $\{\varepsilon_t\}_{t=1}^{\infty}$.

Ces premiers résultats négatifs nous ont amenés à une analyse plus fine des schémas de parallélisation, analyse présentée dans le chapitre 6. En particulier, nous avons montré que dans notre problème le moyennage de différentes versions des prototypes ne mène pas à une meilleure version. Ce résultat vient très probablement de l'absence de convexité de notre fonction de perte, puisque des résultats théoriques d'accélération sont fournis dans des cadres très proches lorsque l'hypothèse de convexité est ajoutée (nous renvoyons par exemple à [49]). Les modélisations (6.10) et (6.11) présentent des schémas alternatifs de parallélisation dont la simulation a présenté une accélération de la convergence par rapport à la version séquentielle de l'algorithme de VQ.

Le chapitre 7 présente l'implémentation cloud des travaux précédents. En particulier, nous décrivons en détail l'ordonnancement de notre algorithme en différents services et les mécanismes classiques de recouvrement de calcul et de communication utilisés au sein de chaque unité de calcul. Nous montrons que la version cloud de notre DAVQ permet d'obtenir des gains conséquents en terme d'accélération de la convergence : l'utilisation de machines supplémentaires permet jusqu'à un certain point d'augmenter cette vitesse de convergence vers un niveau de quantification équivalent.

La dernière section du chapitre 7 présente un exemple dans lequel notre implémentation cloud du DAVQ converge plus rapidement que notre implémentation cloud du Batch K-Means vers des niveaux de quantification comparables. Nous retrouvons ainsi comme dans le cas séquentiel certaines des conclusions qui avaient été tirées de comparaisons réalisées entre Batch K-Means et Online K-Means (c'est à dire VQ) dans lesquelles l'algorithme en-ligne permettait d'obtenir plus rapidement un même niveau de quantification. Ces bonnes performances statistiques de notre version stochastique doivent être mises en regard avec le coût important de développement de l'algorithme, ainsi que l'instabilité et la difficulté de paramétrage inhérente à de nombreux algorithmes stochastiques.

Perspectives

Les ressources logicielles et matérielles sur le cloud, tout comme les enjeux technologiques de Lokad, sont en constante évolution. À ce titre, il est crucial de ne pas dissocier les résultats présentés dans cette thèse, ainsi que les questions qui les ont engendrés, du contexte historique où ces questions et réponses ont été exprimées. Par exemple, le choix d'utiliser le système de stockage persistent d'Azure comme moyen de communication est le fruit du manque d'environnements logiciels comme MPI ou MapReduce lorsque ce travail de thèse a été réalisé. Ces environnements logiciels devraient cependant sans doute être disponibles dans un futur proche pour Azure. Lorsqu'ils le seront, de nouvelles possibilités s'ouvriront pour paralléliser nos algorithmes.

Au delà de cette adaptation à l'évolution technologique, il serait également intéressant de réfléchir à une adaptation de nos travaux dans le cas d'une métrique différente de classification. Comme nous l'avons déjà évoqué, l'utilisation du critère présenté dans les chapitres 5 et 6 présente en effet deux lacunes. La première est d'ordre numérique : sa non-convexité entraine des difficultés supplémentaires importantes lorsque nous cherchons un minimum à ce critère. En second lieu, le critère précédent n'est peut être pas le plus pertinent eu égard à l'utilisation qui en est faite ensuite par Lokad : il faudrait effectivement choisir un critère qui tienne compte de la technique de régression utilisée à la suite de la classification. Ces questions sont abordées dans la littérature sous le terme de « clusterwise regression », et pourraient permettre de proposer des critères de classification plus adéquats dans notre cas.

Les techniques de minimisation de notre critère présentées dans les chapitres 6 et 7 sont appliquées à un problème de classification mais peuvent s'adapter sans difficulté majeure à de nombreux autres problèmes de descente de gradient parallèle. Il serait intéressant de les comparer en pratique, dans le cas d'autres fonctions à minimiser, aux techniques de moyennage des résultats proposées par exemple dans [49], pour lesquelles des vitesses optimales de convergence sont démontrées. Dans le cadre de notre algorithme de VQ réparti, nous pourrions également tester

des techniques d'accélération de descente de gradient, par exemple la méthode de Nesterov (nous renvoyons le lecteur à [91]).

De manière plus générale, nous observons à la lumière des expériences que nous avons réalisées que la parallélisation d'algorithmes d'apprentissage statistique ou de fouille de données n'est pas encore un domaine mûr. En effet, l'offre actuelle des environnements logiciels disponibles propose deux alternatives souvent insatisfaisantes : l'alternative MapReduce d'une part, simple d'utilisation mais limitée dans l'expressivité et peu adaptée à des algorithmes asynchrones, ou avec de nombreuses communications, ou encore à des algorithmes itératifs. La seconde alternative est actuellement représentée par des environnements logiciels plus riches, comme Dryad ou Graphlab, qui ne sont pas encore réellement disponibles sur le cloud, et dont la complexité nous semble restreindre très fortement leur public respectif. Des environnements logiciels à la fois simples et puissants dans leur expressivité restent donc à imaginer et concevoir.

List of Figures

List of Tables

Bibliography

[1] Amazon cloud to break the 1 billion dollars barrier? http://www.crn.com/news/cloud/231002515/amazon-cloud-to-break-the-1-billion-barrier.htm read the 29/03/2012.

[2] Appdomain trick. http://code.google.com/p/lokad-cloud/wiki/ExceptionHandling read the 26/04/2012.

[3] Azure pricing. http://www.microsoft.com/windowsazure/pricing/.

[4] Azure pricing calculator. http://www.windowsazure.com/en-us/pricing/calculator/advanced/ read the 25/04/2012.

[5] Azure pricing details. http://www.windowsazure.com/en-us/pricing/details/ read the 25/04/2012.

[6] Azure Scope. http://azurescope.cloudapp.net/.

[7] Azure storage resources. http://blogs.msdn.com/b/windowsazurestorage/archive/2010/03/28/windows-azure-storage-resources.aspx.

[8] Cloud survey. http://assets1.csc.com/newsroom/downloads/CSC_Cloud_Usage_Index_Report.pdf read the 28/03/2012.

[9] http://nosql-database.org/. http://nosql-database.org/censoredNoSQLdatabases.

[10] Leaky abstractions. http://www.joelonsoftware.com/articles/LeakyAbstractions.html.

[11] Lokad-cloud. http://code.google.com/p/lokad-cloud/.

[12] Lokad-cqrs. http://lokad.github.com/lokad-cqrs/.

[13] Microsoft cloud investment. http://www.bloomberg.com/news/2011-04-06/microsoft-s-courtois-says-to-spend-90-of-r-d-on-cloud-strate html read the 28/03/2012.

[14] MPI cluster on EC2. http://datawrangling.s3.amazonaws.com/elasticwulf_pycon_talk.pdf read the 03/05/2012.

[15] Problems with ACID and how to fix them. http://dbmsmusings.blogspot.com/2010/08/problems-with-acid-and-how-to-fix-them.html read the 28/03/2012.

[16] Should I expose synchronous wrappers for asynchronous methods? http://blogs.msdn.com/b/pfxteam/archive/2012/04/13/10293638.aspx read the 26/04/2012.

[17] Websitemonitoring. http://www.website-monitoring.com/.

[18] *VL2: A Scalable and Flexible Data Center Network*, 2011.

[19] D. Abadi. Problems with CAP, and Yahoo's little known NoSQL system. http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html read the 28/03/2012.

[20] G. A Abandah and E. S. Davidson. Modeling the communication and computation performance of the IBM SP2. *Proceedings of the 10th International Parallel Processing Symposium*, (April), 1996.

[21] C. Abraham, P. A. Cornillon, E. Matzner-Løber, and N. Molinari. Unsupervised curve clustering using B-Splines. *Scandinavian Journal of Statistics*, 30:581–595, 2003.

[22] R. Agrawal and J. C. Shafer. Parallel mining of association rules: Design, implementation, and experience. *IEEE Trans. Knowledge and Data Eng., 8(6):962-969*, 1996.

[23] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Loose synchronization for large-scale networked systems. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 28–28, Berkeley, CA, USA, 2006. USENIX Association.

[24] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004.

[25] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[26] D. Arthur and S. Vassilvitskii. How slow is the k-means method? *Construction*, 2006.

[27] J. Bahi, S. Contassot-Vivier, and R. Couturier. Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. *Parallel Computing*, 31(5):439–461, 2005.

[28] R. Bekkerman, M. Bilenko, and J. Langford. *Scaling up Machine Learning*. Cambridge University Press, 2012.

[29] A. Benveniste, M. Métivier, and P. Priouret. *Adaptive algorithms and stochastic approximations*. Springer-Verlag, 1990.

[30] S. Bermejo and J. Cabestany. The effect of finite sample size on on-line k-means. *Neurocomputing*, 48:511–539, 2002.

[31] Gérard Biau, Luc Devroye, and Gábor Lugosi. On the performance of clustering in hilbert spaces. *IEEE Transactions on Information Theory*, 54(2):781–790, 2008.

[32] L. Bottou and Y. Bengio. Convergence properties of the k-means algorithms. In *Advances in Neural Information Processing Systems 7*, pages 585–592. MIT Press, 1995.

[33] L. Bottou and Y. LeCun. Large scale online learning. In *Advances in Neural Information Processing Systems 16*. MIT Press, 2004.

[34] L. Bottou and Y. LeCun. On-line learning for very large datasets. *Applied Stochastic Models in Business and Industry*, 21:137–151, 2005.

[35] P. S. Bradley and U. M. Fayyad. *Refining Initial Points for K-Means Clustering*, volume 727, pages 91–99. Morgan Kaufmann, San Francisco, CA, 1998.

[36] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. Haq, M. I. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[37] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 99–106, Washington, DC, USA, 2005. IEEE Computer Society.

[38] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, August 2008.

[39] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *in proceedings of the 7th conference on usenix symposium on operating systems design and implementation - volume 7*, pages 205–218, 2006.

[40] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for Machine Learning on Multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[41] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *International workshop on designing privacy enhancing technologies: design issues in anonymity and unobservability*, pages 46–66. Springer-Verlag New York, Inc., 2001.

[42] S. Contassot-Vivier, T. Jost, and S. Vialle. Impact of asynchronism on GPU accelerated parallel iterative computations. In *PARA 2010: State of the Art in Scientific and Parallel Computing*, LNCS. Springer, Heidelberg, 2011.

[43] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[44] J. Dai and B. Huang. *Design Patterns for Cloud Services*, volume 74, pages 31–56. Springer Berlin Heidelberg, 2011.

[45] C. de Boor. On calculating with b-splines. *Journal of Approximation Theory*, 6:50–62, 1972.

[46] C. de Boor. *A practical guide to splines*. Springer-Verlag, 1978.

[47] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[48] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Communications of the ACM, vol. 51, no. 1*, pages 107–113, 2008.

[49] O. Dekel, R. Gilad-Bachrach, O Shamir, and L. Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13:165–202, March 2012.

[50] O. Delalleau and Y. Bengio. Parallel stochastic gradient descent. *Proceedings of SPIE*, 6711:67110F–67110F–14, 2007.

[51] I. S. Dhillon and D. S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD*, pages 245–260, London, UK, 2000. Springer-Verlag.

[52] U. Drepper. What every programmer should know about memory, 2007.

[53] J.C. Fort, M. Cottrell, and P. Letremy. Stochastic on-line algorithm versus batch algorithm for quantization and self organizing maps. *Neural Networks for Signal Processing XI Proceedings of the 2001 IEEE Signal Processing Society Workshop IEEE Piscataway NJ USA*, 00(C):43–52, 2001.

[54] A. Freeman. *Pro .NET 4 Parallel Programming in C#.* Apress, 2010.

[55] A. Gersho and R. M. Gray. *Vector quantization and signal compression*. Kluwer, 1992.

[56] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[57] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[58] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. *SIGARCH Comput. Archit. News*, 17(2):64–75, April 1989.

[59] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, December 2008.

[60] J. Gregorio. Sharding counters. `https://developers.google.com/appengine/articles/sharding_counters` read the 17/05/2012.

[61] W. H. Greub. *Linear algebra*. Springer-Verlag, 4th edition, 1975.

[62] W. Gropp and E. Lusk. Reproducible measurements of MPI performance characteristics. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 11–18, London, UK, UK, 1999. Springer-Verlag.

[63] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24:8–12, 2009.

[64] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 132–141.

[65] C. Hennig. Models and methods for clusterwise linear regression. In *Proceedings in Computational Statistics*, pages 3–0. Springer, 1999.

[66] M. Herlihy, B. H. Lim, and N. Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13:343–364, 1995.

[67] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[68] T. Hey, S. Tansley, and K. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington, 2009.

[69] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey. Early observations on the performance of Windows Azure. *Sci. Program.*, 19(2-3):121–132, April 2011.

[70] T. Hoefler, W. Gropp, R. Thakur, and J. L. Träff. Toward performance models of MPI implementations for understanding application scaling issues. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, pages 21–30, Berlin, Heidelberg, 2010. Springer-Verlag.

[71] B. Hong and Z. He. An asynchronous multithreaded algorithm for the maximum network flow problem with nonblocking global relabeling heuristic. *IEEE Transactions on Parallel and Distributed Systems*, 22:1025–1033, 2011.

[72] M. Isard. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, April 2007.

[73] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[74] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. In *ACM computing surveys, Vol.31, no.3, September*, 1999.

[75] M. N. Joshi. Parallel K-means algorithm on Distributed Memory Multiprocessors. Technical report, Computer Science Department University of Minnesota, Twin Cities, 2003.

[76] T. Kohonen. Analysis of a simple self-organizing process. *Biological Cybernetics*, 44:135–140, 1982.

[77] H. J. Kushner and D. S. Clark. *Stochastic approximation for constrained and unconstrained systems*. Springer-Verlag, 1978.

[78] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. *Security*, 2009.

[79] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm. What's inside the cloud? an architectural map of the cloud landscape. In *ICSE Workshop on Software Engineering Challenges of Cloud Computing, 2009. CLOUD 09.* IEEE Press, Mai 2009.

[80] J. Lin. The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce. In *LSDS-IR workshop*, 2009.

[81] J. Lin and C. Dyer. *Data-intensive text processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[82] H. Liu and D. Orban. Cloud mapreduce: a mapreduce implementation on top of a cloud operating system. Technical report, Accenture Technology Labs, 2009. `http://code.google.com/p/cloudmapreduce/`.

[83] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[84] G. Louppe and P. Geurts. A zealous parallel gradient descent algorithm. In *NIPS 2010 Workshop on Learning on Cores, Clusters and Clouds*, 2010.

[85] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[86] J. B. MacQueen. Some methods of classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[87] M. Mahajan, P. Nimbhorkar, and K. Varadarajan. The planar k-means problem is np-hard. In *Proceedings of the 3rd International Workshop on Algorithms and Computation*, WALCOM '09, pages 274–285, Berlin, Heidelberg, 2009. Springer-Verlag.

[88] D. C. Marinescu. *Cloud Computing: Theory and Practice*. 2012.

[89] G. W. Milligan and P. D. Isaac. The validation of four ultrametric clustering algorithms. *Pattern Recognition*, 12:41–50, 1980.

[90] B. Mirkin. *Clustering for data mining: a data recovery approach*. Chapman & Hall/CRC, 2005.

[91] Y. Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Kluwer Academic Publishers, 2004.

[92] O. O Malley. Terabyte sort on apache hadoop. `http://www.hpl.hp.com/hosted/sortbenchmark/YahooHadoop.pdf`.

[93] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[94] G. Pagès. A space vector quantization for numerical integration. *Journal of Applied and Computational Mathematics*, 89:1–38, 1997.

[95] B. Patra. Convergence of distributed asynchronous learning vector quantization algorithms. *Journal of Machine Learning Research*, 12:3431–3466, 2010.

[96] A. D. Peterson, A. P. Ghosh, and R. Maitra. A systematic evaluation of different methods for initializing the k-means clustering algorithm. Technical report, 2010.

[97] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, October 2005.

[98] D. Pollard. Strong consistency of k-means clustering. *The Annals of Statistics*, January 1981.

[99] D. Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[100] Y. Raz. The dynamic two phase commitment (d2pc) protocol. In *Database Theory - ICDT '95, Lecture Notes in Computer Science, Volume 893 Springer, ISBN 978-3-540-58907-5*, pages 162–176, 1995.

[101] F. Rossi, B. Conan-Guez, and A. El Golli. Clustering functional data with the SOM algorithm. In *Proceedings of ESANN 2004*, pages 305–312, Bruges, Belgium, April 2004. `http://apiacoa.org/publications/2004/som-esann04.pdf`.

[102] J. C. Shafer, R. Agrawal, and M. Mehta. A scalable parallel classifier for data mining. *Proc. 22nd International Conference on VLDB, Mumbai, India*, 1996.

[103] M. Snir, S. Otto, S. Huss-Lederman, W. David, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Boston, 1996.

[104] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2:315–339, 1990.

[105] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 17:2–4, 2005.

[106] J. Tsitsiklis, D. Bertsekas, and M. Athans. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Transactions on Automatic Control*, 31:803–812, 1986.

[107] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[108] W. Vogels. Eventually consistent. *Queue*, 6(6):14–19, October 2008.

[109] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storages : the consumers'perspective. *Reading*, pages 134–143, 2011.

[110] E. Walker. Benchmarking Amazon EC2 for high-performance scientific computing. *LOGIN*, 33(5):18–23, October 2008.

[111] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon ec2 data center. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, pages 1163–1171, Piscataway, NJ, USA, 2010. IEEE Press.

[112] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[113] E. Wigner. The unreasonable effectiveness of mathematics in the natural sciences. In *Communications in Pure and Applied Mathematics vol. 13, No. I February*, 1960.

[114] D.Randall Wilson and Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429 – 1451, 2003.

[115] P.C. Yew, N.F. Tzeng, and D.H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, pages 388–395, April 1987.

[116] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Kumar, and G. J. Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[117] X. Zhiwei and K. Hwang. Modeling communication overhead: MPI and MPL performance on the IBM SP2. *Parallel Distributed Technology Systems Applications IEEE*, 4(1):9–23, 1996.

[118] M. Zinkevich, A. Smola, and J. Langford. Slow learners are fast. In *Advances in Neural Information Processing Systems 22*, pages 2331–2339, 2009.

[119] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems 23*, 2010.