Optimisation de la gestion des ressources sur une plate-forme informatique du type Big Data basée sur le logiciel Hadoop

Aymen Jlassi

UNIVERSITÉ FRANÇOIS RABELAIS DE TOURS

ÉCOLE DOCTORALE MIPTIS

Laboratoire d'Informatique (EA 6300)

ÉQUIPE de RECHERCHE: Recherche Opérationnelle, Ordonnancement et Transport

THÈSE présentée par :

Aymen JLASSI

soutenue le : 11 Décembre 2017

pour obtenir le grade de : Docteur de l'Université François-Rabelais de Tours

Discipline/ Spécialité : INFORMATIQUE

Optimisation de la gestion des ressources sur une plate-forme informatique du type "Big Data" basée sur le logiciel Hadoop

THÈSE DIRIGÉE PAR :

M.	MARTINEAU, Patrick	Professeur, Université François Rabelais de Tours (Directeur de thèse)
M.	T'KINDT, Vincent	Professeur, Université François Rabelais de Tours

RAPPORTEURS :

MME.	NORRE, Sylvie	Professeur, Université Blaise Pascal Clermont II
M.	BEAUMONT, Olivier	DR, Inria, Bordeaux

EXAMINATEUR :

M.	LEBRE, Adrien	Chargé de recherche, Inria, Nantes
MME.	KEDAD-SIDHOUM, Safia	Maitre de Conférences (HDR), Université Pierre et Marie Curie, Paris.

Table des matières

Table des matières **1**

Liste des tableaux **5**

Table des figures **7**

Introduction **9**

1 Présentation du problème **13**
1.1 Présentation générale du modèle MapReduce 13
1.1.1 Présentation et principes 14
1.1.2 Caractéristiques des tâches de type Map et Reduce 16
1.1.3 Contraintes de précédence entre les tâches Map et Reduce 17
1.1.4 Différence entre ordonnancement hors ligne et ordonnancement en ligne . 18
1.2 Caractéristiques des grappes Hadoop 18
1.2.1 Architecture d'une grappe Hadoop 19
1.2.2 Relation entre tâches, ressources et conteneurs dans Hadoop 20
1.3 Problème d'ordonnancement de travaux MapReduce 21
1.3.1 Description du problème 22
1.3.2 Notations 23
1.3.3 Contraintes 26
1.3.4 Critères 27
1.4 Conclusion 27

2 Optimisation et ordonnancement **29**
2.1 Introduction 29
2.2 Les problèmes d'optimisation 29
2.3 Les problèmes d'ordonnancement 31
2.3.1 Les types d'ordonnancement 32
2.3.2 Définition du critère d'optimalité 33
2.3.3 Analyse de la complexité des algorithmes 33
2.4 Les méthodes de résolutions 34
2.4.1 Méthodes exactes 35

2.4.2 Les méthodes basées sur l'amélioration progressive de la solution 37
2.4.3 Approche combinant les méthodes exactes et les métaheuristiques 40
2.5 Exemple d'algorithmes d'ordonnancement dans les grilles de calcul 40
2.5.1 L'algorithme de partage équitable Max-Min 41
2.5.2 Les algorithmes de liste 42
2.6 Conclusion 43

3 État de l'art **45**
3.1 Introduction 45
3.2 Partage équitable de ressources 49
3.2.1 Algorithme «FS» (*FairScheduler*) 49
3.2.2 Algorithme «CS» (*Capacity Scheduler*) 51
3.2.3 Algorithme MARLA 52
3.2.4 Algorithme «DRF» (*Dominant Resource Fairness*) 53
3.2.5 Algorithme «NATA» (*Network-Aware Task Assignement*) 54
3.2.6 Autres algorithmes 54
3.3 Amélioration des performances de la grappe 55
3.3.1 Algorithme «DS» (*Delay Scheduling*) 55
3.3.2 Algorithme «LATE» (*Longest Approximate Time to End*) 56
3.3.3 Algorithme «BASE» (*BASE Scheduler*) 57
3.3.4 Algorithme «MAESTRO» (*Replica-Aware Map Scheduling for MapReduce*) 58
3.3.5 Autres algorithmes 58
3.4 Respect de la qualité de service 59
3.4.1 Algorithme «DCS» (*Deadline Constraint Scheduler*) 59
3.4.2 Algorithme «EDCS» (*Extended Deadline Constraint Scheduler*) 60
3.4.3 Algorithme «RATCTP» (*Resource-Aware Scheduling Through Coupling Task Progress*) 61
3.4.4 Algorithme «ARIA» (*ARIA-SLO-based scheduler*) 61
3.4.5 Algorithme «MRA++» (*Scheduling and data placement on MapReduce for heterogeneous environments*) 62
3.4.6 Algorithme «DPS» (*Dynamic Proportional Share*) 62
3.5 Traitement locale des données pour améliorer les performances 63
3.5.1 Algorithme «PURLIEUS» (*Locality-aware resource allocation*) 63
3.5.2 Algorithme «MS» (*Matchmaking Scheduling*) 64
3.5.3 Algorithme «CoGRS» (*Center-of-Gravity Reduce Task Scheduling*) 65
3.5.4 Algorithme Mantri 66
3.5.5 Algorithme Tarazu 66
3.5.6 Autres algorithmes 67
3.6 Problème d'optimisation de la consommation énergétique 68
3.6.1 Mécanismes de réduction de la consommation énergétique 68

3.6.2 Algorithme «HybridMR» (*Hierarchical Mapreduce Scheduler for Hybrid data centers*) . . . 70
3.7 Tableau de synthèse des différents travaux . . . 70
3.8 Conclusion . . . 71

4 Résolution optimale du problème hors-ligne 75
4.1 Introduction . . . 75
4.1.1 Formalisation du problème et hypothèses spécifiques . . . 76
4.1.2 Définition du problème d'ordonnancement . . . 76
4.2 Problème de la minimisation du C_{max} . . . 77
4.2.1 Variables du modèle . . . 77
4.2.2 Formalisation du modèle . . . 78
4.2.3 Évaluation du modèle . . . 80
4.2.4 Présentation et évaluation de l'heuristique de résolution . . . 84
4.3 Problème pour la minimisation de $\sum w_j C_j + C_{max}$ . . . 90
4.3.1 Paramètres et Variables . . . 90
4.3.2 Formalisation du modèle . . . 91
4.3.3 Critères de performance . . . 92
4.3.4 Évaluation du modèle . . . 92
4.3.5 Relaxation linéaire et simplification du modèle . . . 96
4.3.6 Evaluations expérimentales et résultats . . . 97
4.4 Conclusion . . . 99

5 Résolution du problème d'ordonnancement en ligne 101
5.1 Introduction . . . 101
5.2 Considérations générales concernant l'évaluation des heuristiques . . . 101
5.3 Minimisation du critère C_{max} . . . 103
5.3.1 Présentation de l'heuristique LocFirst (V2) . . . 104
5.3.2 Analyse de la complexité de LocFirst (V2) . . . 107
5.3.3 Évaluation de l'heuristique "LocFirst (V2)" . . . 109
5.4 Minimisation du critère $\sum w_j C_j + C_{max}$ . . . 116
5.4.1 Présentation de l'heuristique "Proximity-aware resource allocation for Map and Reduce Tasks" (ProxForMapReduce) . . . 116
5.4.2 Présentation de l'heuristique "$MRSchedulingBasedOnCost$" . . . 123
5.4.3 Analyse de complexité . . . 129
5.4.4 Évaluation expérimentale des heuristiques . . . 130
5.5 Analyse des résultats . . . 136
5.5.1 Conclusion . . . 137

Conclusion et perspectives 139

A Déploiement de Hadoop sur les nuages informatiques **143**
A.1 Introduction . . . 143
A.2 Introduction aux technologies de virtualisation . . . 147
A.2.1 La virtualisation lourde, isolation complète : . . . 147
A.2.2 La virtualisation légère, isolation partielle : . . . 148
A.2.3 Différence entre les deux types de virtualisation . . . 148
A.3 Méthodologie des expérimentations . . . 149
A.4 Impact de la virtualisation sur le temps d'exécution des travaux . . . 151
A.5 Comparaison des technologies de virtualisation . . . 152
A.5.1 Étude de l'influence sur l'utilisation des ressources de calcul CPU (grappe homogène) . . . 154
A.5.2 Étude de l'effet de l'accès concurrent aux supports de sauvegardes et aux ressources réseau (grappe homogène) . . . 155
A.5.3 Étude de la variation de performance de Hadoop sur les grappes hétérogènes . . . 157
A.6 Étude de l'influence des technologies de virtualisation sur la consommation énergétique . . . 159
A.7 Discussion . . . 160
A.8 Conclusion . . . 162

B Publications Personnelles **163**
B.1 Publications Internationales . . . 163
B.2 Publications Nationales . . . 163
B.3 Chapitre dans un livre . . . 164

Bibliographie **165**
Table des matières

Liste des tableaux

1.1 Ensembles d'indices et d'exposants utilisés dans les notations 24
1.2 Les notations et les données 25
1.2 Les notations et les données 26

3.1 Listes des politiques d'ordonnancement étudiées leur objectifs (1/2). 72
3.2 Listes des politiques d'ordonnancement étudiées leur objectifs (2/2). 73

4.1 Configuration des machines utilisées dans l'évaluation 81
4.2 Tailles des travaux utilisés lors de l'évaluation du modèle 81
4.3 Formules de génération des quantités de mémoire et de disque par tâche 81
4.4 Scénarios de génération des données d'entrée de la 1^{re} version du modèle 81
4.5 Résultats de l'évaluation de la 1^{re} version du modèle 83
4.6 Scénarios de génération des données d'entrée de la 2^{me} version du modèle 93
4.7 Résultats de l'évaluation de la 2^{me} version du modèle 93
4.8 Résultats obtenus suite aux différentes relaxations 98

5.1 Évaluation de "LocFirst (V2)" par rapport à la solution optimale 109
5.2 Scénarios utilisés dans l'évaluation en ligne des heuristiques 111
5.3 Tailles des travaux utilisés dans l'évaluation en ligne des heuristiques 111
5.4 Différents types de travaux utilisés lors de l'évaluation des heuristiques 111
5.5 Temps de calcul moyen (en sec) associés aux heuristiques pour chaque scénario . 113
5.6 Résultats fournis par "LocFirst (V2)" dans un contexte en ligne 114
5.7 Évaluation en ligne de "LocFirst (V2)" par rapport aux heuristiques de la littérature 114
5.8 Résultats de l'évaluation de "ProxForMapReduce" dans un contexte hors ligne . 132
5.9 Résultats de l'évaluation de "ProxForMapReduce" dans un contexte en ligne . . . 133
5.10 Évaluation en ligne de "*ProxForMapReduce*" par rapport aux heuristiques de la littérature 133
5.11 Résultats de l'évaluation de "*MRSchedulingBasedOnCost*" en mode hors ligne 134
5.12 Résultats fournis par "MRSchedulingBasedOnCost" dans un contexte en ligne . . 135
5.13 Résultats de l'évaluation de "MRSchedulingBasedOnCost" par rapport aux heuristiques de la littérature 135

A.1 Configuration des machines pour les expériences sur une grappe homogène 150

A.2 Configuration des machines pour les expériences sur une grappe hétérogène . . . 151

Table des figures

1.1 Exemple d'exécution d'un travail MapReduce 15
1.2 Architecture client-serveur d'une grappe Hadoop 19
1.3 Présentation des ressources disponibles sur une machine esclave Hadoop 22
1.4 Relation de précédences entre les tâches d'un travail MapReduce 23
1.5 Architecture réseau d'une grappe Hadoop 23

2.1 Classification de la combinaison des algrithmes exactes et des métaheuristiques . 40
2.2 Réseau utilisé pour l'illustration du partage équitable 41

3.1 Exécution des travaux avec l'ordonnanceur FIFO (*Fisrt in first out*) 45
3.2 Exécution des travaux avec l'ordonnanceur «FS». 50
3.3 Exécution des travaux avec l'ordonnanceur «CS» 51
3.4 Architecture utilisée dans l'ordonnanceur «ARIA» 62
3.5 Relations d'évolution entre certains algorithmes d'ordonnancement MapReduce . 74

4.1 Moyenne des consommation des ressources par rapport au temps. 83
4.2 Moyenne des consommation des ressources (en Mo) par rapport au temps. 95
4.3 Degrés de localité par scénario (10 instances par scénario) 95

5.1 Évaluation de "LocFirst (V2)" dans un contexte hors ligne. 110
5.2 Moyenne des valeurs du critère C_{max} 110
5.3 Moyennes des temps de fin d'exécution des travaux C_{max} de chacun des scénarios ordonnés selon le degré de parallélisme 112
5.4 Moyennes des temps de fin d'exécution des travaux C_{max} de chacun des scénarios ordonnés selon le nombre de tâches par scénarios 112
5.5 Moyenne des temps de calcul de chacun des scénarios 114
5.6 Résultats de l'évaluation de "$ProxForMapReduce$" dans un contexte hors ligne. 131
5.7 Moyennes des temps de calcul de chacune des heuristiques 133
5.8 Évaluation de "MRSchedulingBasedOnCost" dans un contexte hors ligne. 134
5.9 Évaluation de "MRSchedulingBasedOnCost" par rapport aux temps de calcul des solutions 135
5.10 Comparaison entre "$ProxForMapReduce$" et "$MRSchedulingBasedOnCost$" par rapport aux temps de calcul 137

5.11 Comparaison entre "ProxForMapReduce" et "MRSchedulingBasedOnCost" par rapport aux résultats 138

A.1 Comparaison entre l'architecture des outils Docker et VMware 147
A.2 Mesure du niveau de surcharge selon le nombres de machines esclaves et les types d'outils de virtualisation 152
A.3 Exemple de la consommation des ressources dues à l'exécution du travail TeraGen 153
A.4 Durées des exécutions des différents travaux 154
A.5 Taux de transfert réseau, des accès disque suite à l'exécution de TestDFSIO . . . 156
A.6 Durée d'exécution des travaux TeraGen et TeraSort sur des grappes hétérogènes . 158
A.7 Consommation énergétique de différentes tailles de grappes avec TeraGen et TeraSort 160

Introduction

La première utilisation scientifique du terme « gros volume de données ou Big Data » remonte à l'année 1997 par les chercheurs Michael Cox et David Ellsworth [40] de la NASA. Ils y expliquent en détails les enjeux technologiques et les défis à porter en manipulant de gros volumes de données. En 2001, Doug Laney [96] a présenté la fameuse définition de la gestion de gros volumes de données sous la forme des trois V (Volume, Vitesse et Variété). C'est cette définition qui est aujourd'hui présente dans la plupart des documents introduisant le contexte de gestion des gros volumes de données. Le rapport de Yigitbasi et al. [167] redéfinit en 2011 avec plus de précision le terme « BigData ». Ce rapport a présenté le début d'une grande campagne médiatique, cette dernière focalise sur la nouvelle génération de logiciels miracles qui bouleverseront les technologies existantes. De plus en plus d'entreprises ([48][109]) s'intéressent à ce phénomène. En conséquence, plusieurs questions se sont imposées : Est ce que le « BigData » est vraiment une révolution technologique, une nouvelle approche de traitement de données ou tout simplement une nouvelle mode créée par les éditeurs de logiciels pour promouvoir leurs produits ? Quelle est la différence entre les gros volumes de données (Bigdata) et les données traditionnelles utilisées dans nos bases de données relationnelles ?
Depuis le début du dix-huitième siècle, de grands volumes d'informations ont été collectés à grand échelle : le recensement des individus au dix-huitième siècle est une opération qui durait un an. La collecte des tickets de caisse dans les grandes chaines de distributions, la simulation des changements climatiques et la réservation aérienne des vols à travers le monde sont autant d'exemples plus récents de collecte de gros volumes de données. Ces exemples sont la preuve concrète que la gestion de gros volumes de données n'est pas un phénomène nouveau. Les entreprises manipulaient les grands volumes de données depuis longtemps. Le développement des outils de gestion des gros volumes de données a été rendu possible par l'augmentation des capacités de stockage et de la puissance de calcul. Les outils informatiques existant au début des années 2000 présentaient des limites techniques. Des entreprises telles que Google, Yahoo et Général Électrique cherchaient à passer à une échelle de parallélisme de l'ordre de milliers de serveurs. Les coûts engendrés par les techniques des années 2000 étaient très élevés. Les technologies disponibles de l'époque ne permettaient pas de paralléliser le stockage et les traitements de façon simple et avec de faibles coûts. Les outils de traitement Big Data regroupent des infrastructures, des technologies et des services permettant de transformer les données en informations et les informations en connaissances. Elle présente un champ d'application des algorithmes connus de datamining et d'apprentissage automatique. Elle leur fournit les quantités importantes de données et la puis-

sance de calcul nécessaire pour plus de potentiel et de pertinence dans leurs résultats.
Dès l'apparition des nouvelles technologies de gestion de gros volumes de données, plusieurs études ont été réalisées pour évaluer, comparer et décrire la relation entre les outils nommés BigData et les anciens outils de gestion d'informations ([5][34][83]). Hadoop est le logiciel le plus populaire parmi une diversité d'outils dont l'usage ne cesse d'augmenter. Il est intégré à plus de 80% des projets BigData. Depuis sa création (en 2007), Hadoop est considéré comme le logiciel de référence en matière de gestion (de sauvegarde et de traitement) de gros volumes de données. Il est basé sur l'exécution distribuée et parallèle des traitements et repose sur une architecture client-serveur. Les grappes Hadoop chez Yahoo occupent plus de 40000 serveurs pour plus de 100000 CPU [109]. Plusieurs études ([50][73][82][83]) cherchent à détecter les détails et les mécanismes à améliorer pour optimiser l'exécution des travaux sur Hadoop. Nous présentons dans le chapitre 1 le logiciel Hadoop et les notions de base afférentes.
La taille des grappes Hadoop encourage plusieurs entreprises à utiliser les technologies de virtualisation [108] pour mettre en place les infrastructures de gestion de gros volumes de données. L'utilisation de ces technologies a des avantages et des inconvénients. Parmi les avantages, elles permettent *(i)* de masquer la complexité des infrastructures, *(ii)* d'optimiser et de réduire les coûts de l'exploitation des serveurs, *(iii)* d'augmenter la productivité en offrant la possibilité de partager une machine (telle qu'un serveur) entre plusieurs machines virtuelles. Cependant, les outils de virtualisation sont des middlewares. De ce fait, leur principal inconvénient est l'augmentation du niveau de charge engendré suite à leur utilisation.

Contexte

La présente thèse fait partie d'un projet qui vise à promouvoir et améliorer les performances des grappes de gestion de gros volumes de données. Elle est réalisée sous la tutelle du Laboratoire d'Informatique de l'université de Tours et l'entreprise Group Cyres. L'entreprise Group Cyres est leader dans la Région Centre Val de Loire, dans l'hébergement et le traitement des données. Son pôle de recherche et de développement est de façon continue confronté aux problèmes rencontrés par ses clients. Elle coopère avec les laboratoires de la Région Centre pour améliorer la qualité de services fournie à ses clients. L'entreprise Group Cyres utilise des outils de virtualisation basés sur les deux technologies Microsoft Azure et VMware VSphère pour l'exploitation des ressources de son centre de données. Elle est composée de plusieurs entités spécialisées et complémentaires dont l'entité responsable de répondre aux besoins des clients pour des applications "Big Data", nommée Ingensi. Ingensi propose des solutions qui se basent sur des outils de l'écosystème Hadoop. Elles sont déployées principalement sur le cluster de l'entreprise.
Suite à l'évaluation et aux tests effectués sur des grappes Hadoop, par des déploiements sur plusieurs infrastructures, les ingénieurs chez Cyres ont remarqué des dégradations importantes de performance (entre 20% et 30%) du logiciel Hadoop et de sa plateforme. Afin de pallier à ce problème et d'améliorer la qualité de services aux clients, l'entité Ingensi, représentée par M. Christophe Cerqueira et Guillaume Polaert en tant que directeur et responsable du pôle de recherche, ont lancé une coopération avec le Laboratoire d'Informatique. Cette coopération focalise sur l'optimisation des performances du logiciel Hadoop et le présent sujet de thèse est l'un de

ses résultats. Cette thèse est financée par l'Association Nationale de Recherche Technologique (ANRT) et l'entreprise Group Cyres.

Contribution

Après une première étude expérimentale, nous nous sommes fixés deux objectifs. Le premier consiste à optimiser la politique d'ordonnancement des travaux sur une grappe Hadoop. Nous pensons que la politique existante ne considère pas tous les aléas dont cette opération doit dépendre. Le deuxième objectif considère l'environnement de déploiement utilisé. Nous travaillons pour minimiser la dégradation de performance due à l'utilisation des technologies de virtualisation. Nous pensons que la définition d'un protocole de déploiement d'une infrastructure Hadoop adaptée aux besoins de l'entreprise Group Cyres doit être, pour les administrateurs du système, une solution à court et à moyen terme. Elle nous permet d'identifier les éléments essentiels de l'infrastructure dont dépendent les performances d'Hadoop et d'évaluer les solutions existantes et de fournir des éventuelles alternatives.
Cette thèse présente notre apport dans le domaine de l'optimisation des performances logicielles à travers la définition et l'amélioration de la politique d'ordonnancement des tâches sur une grappe Hadoop. Notre contribution suit deux axes :

— Dans une première partie, nous identifions le problème et nous focalisons sur le déploiement d'Hadoop sur une plateforme virtuelle. Dans ce contexte, une étude de scénarios et de procédures est réalisée pour la définition d'un protocole de déploiement. Cette étude a permis l'évaluation des performances d'Hadoop en fonction des outils de virtualisation utilisés, de la taille de la grappe et des types de données.

— Dans la deuxième partie de ce travail, nous réalisons une étude de l'état de l'art sur l'ordonnancement des travaux du type MapReduce. Ensuite, nous considérons une modélisation simpliste du problème d'ordonnancement. Nous considérons la minimisation de la date de fin du dernier travail (C_{max}) comme critère à optimiser. Nous considérons des tâches de durée unitaire. Nous modélisons le problème hors ligne de l'ordonnancement des travaux MapReduce en utilisant la programmation linéaire indexée par rapport au temps. Nous résolvons le modèle à l'aide du solveur mathématique CPLEX et nous calculons une borne inférieure (associée au critère que nous cherchons à optimiser). Nous proposons une heuristique de résolution basée sur les algorithmes de listes, nous l'évaluons par rapport à la borne inférieure calculée sur des petites instances. Ensuite, nous considérons le problème en ligne et nous évaluons l'heuristique sur des instances de taille réelle. Cette évaluation est réalisée par comparaison à des algorithmes de la littérature.
  Cette première modélisation nous a permis d'étudier le problème de façon simpliste et d'avoir une première solution. Cependant, dans un cas réel, où plusieurs utilisateurs utilisent une grappe Hadoop, le critère d'optimalité C_{max} ne répond plus aux besoins puisqu'on peut favoriser la fin des travaux associés à un utilisateur par rapport aux autres. De ce fait, nous faisons évoluer notre modèle et nous considérons des tâches de durée quelconque. Nous considérons que notre objectif est de minimiser la somme des deux critères suivants :

la somme pondérée des dates de fin des travaux et la date de fin du dernier travail. Nous calculons une borne inférieure pour la nouvelle version du modèle mathématique. Nous proposons deux heuristiques de résolution, nous les évaluons, en mode hors ligne, par rapport à la borne inférieure sur des petites instances. Ensuite, nous considérons le problème en ligne et nous les évaluons sur des instances de taille réelle, toujours par comparaison avec des algorithmes de la littérature.

Afin de présenter les travaux réalisés dans les deux axes que nous avons défini. Le manuscrit associé à cette thèse est composé de cinq chapitres.

Le chapitre 1 focalise sur la présentation du logiciel Hadoop et des notions sur lesquelles il se base. À la fin de ce chapitre, nous introduisons le problème d'ordonnancement des travaux du type MapReduce.

Le chapitre 2 est une étude générale des problèmes d'ordonnancement sur les grappes informatiques. Nous les présentons comme un sous-ensemble des problèmes d'optimisation et nous présentons les méthodes les plus utilisées pour la résolution de problèmes similaires.

Le chapitre 3 présente l'étude de l'état de l'art associée à l'ordonnancement des travaux du type MapReduce. Nous énumérons les travaux de recherche les plus référencés et nous les regroupons en fonction de l'objectif qu'ils cherchent à optimiser. À la fin de ce chapitre, nous regroupons les heuristiques étudiées en différentes classes et nous les synthétisons dans un tableau.

Le chapitre 4 focalise sur la version hors ligne du problème d'ordonnancement que nous étudions. Nous modélisons le problème, nous commençons par une version simplifiée du modèle et nous calculons une borne inférieure en considérant la minimisation du C_{max} comme critère à optimiser. Ensuite, nous faisons évoluer notre modèle pour s'approcher d'avantage du problème réel et nous cherchons à optimiser la fonction $\sum w_j C_j + C_{max}$. La résolution de la nouvelle version nous permet de calculer une borne inférieure en considérant la nouvelle fonction objectif.

Le chapitre 5 synthétise notre proposition, nous présentons trois heuristiques : la première est nommé "$LocFirst$", et a pour objectif la minimisation du C_{max}. Elle peut être utilisée dans le cas où un seul utilisateur soumet des travaux sur la grappe Hadoop. Les deux autres heuristiques proposées sont "$ProxForMapReduce$" et "$MRSchedulingBasedOnCost$". Elles ont pour objectifs la minimisation de la somme pondérée des dates de fin des travaux et de la date de fin de tous les travaux à exécuter ($\sum w_j C_j + C_{max}$).

L'annexe 1 synthétise l'étude effectuée pour faciliter le déploiement et améliorer l'exploitation des grappes Hadoop dans un environnement basé sur les nuages informatiques. Nous réalisons une batterie d'expérimentations pour, d'une part, définir les facteurs importants dans le processus d'ordonnancement, et d'autre part, répondre aux besoins à court terme de l'entreprise d'accueil et établir un protocole définissant les procédures à suivre pour le déploiement d'infrastructures Hadoop. Cette étude est prise en compte au niveau de l'entreprise Group Cyrès, son résultat a accéléré l'adoption d'un nouvel outil de virtualisation (du monde du logiciel libre).

Chapitre 1

Présentation du problème

Notre travail de recherche est réalisé dans la cadre d'une thèse CIFRE, cette dernière fait partie d'un travail de collaboration entre l'entreprise Cyres-Group et le laboratoire informatique de l'université François-Rabelais de Tours. Son objectif s'intègre dans les travaux de l'entreprise pour optimiser les performances de leurs plateformes de gestion de gros volumes de données. Ces plateformes sont particulièrement basées sur le logiciel Hadoop et son écosystème. L'objectif de ce chapitre est d'introduire le modèle MapReduce, son implémentation Hadoop et le problème d'ordonnancement que nous étudions. Il est particulièrement essentiel :

1. de rappeler les principes du modèle MapReduce et les caractéristiques de son implémentation Hadoop. Cette partie nous permet de définir des notions utiles pour aborder le problème d'ordonnancement.
2. de présenter le problème d'ordonnancement et ses caractéristiques.

Lorsque nous présentons les différentes notions liées à la définition du modèle MapReduce et de ses caractéristiques, nous nous référons à son implémentation Hadoop pour apporter des exemples ou des explications.

1.1 Présentation générale du modèle MapReduce

Le modèle MapReduce est un modèle créé pour simplifier le développement des applications parallèles et distribuées. Les développeurs se concentrent, en conséquence, sur le développement de deux types de fonctions, la fonction Map et la fonction Reduce. Afin d'expliquer la différence entre les termes travail et tâche dans le modèle MapReduce, deux définitions s'imposent :
Une tâche : est une unité de traitement qui s'exécute sur la grappe. Elle peut exécuter un script associé à une fonction Map ou à une fonction Reduce [152].
Un travail : est un ensemble de tâches Map et Reduce qui respectent le modèle MapReduce [152]. Lorsque toutes les tâches Map et Reduce associées à un travail ont terminé leur exécution, le travail est considéré comme achevé.

1.1.1 Présentation et principes

Depuis son introduction par Google, le modèle MapReduce, qui présentait des idées révolutionnaires, a réussi à s'imposer dans la plupart des centres de données pour effectuer des calculs intensifs. Le mérite de ce modèle est confirmé par l'engagement de plusieurs entreprises dans son implémentation. Actuellement, plus de sept implémentations et distributions sont sur le marché (Hadoop, HDP, CDH, Dryad, MapR...). Les questions principales qui se posent dans ce contexte sont : pourquoi une telle réussite ? Quels sont les facteurs clef de cette réussite ?
Pour apporter des réponses, nous étudions les besoins des entreprises lors de l'apparition du modèle MapReduce. Au début des années 2000, l'arrivée d'Internet et la croissance exponentielle de la quantité des données métier des entreprises ont créé un besoin d'outils de calcul intensif. Des outils révolutionnaires qui sont capables de répondre aux trois critères suivants : *(i)* une puissance de calcul élevée, *(ii)* l'extensibilité attendue lors de la montée en charge et *(iii)* une capacité à assurer un certain niveau de sécurité. Les outils existants à l'époque ne peuvent pas garantir les performances attendues lorsqu'on parle de données de plusieurs téraoctets. Ces limites justifient les fondements du modèle MapReduce : Google le propose comme une solution pour associer la force de calcul à une capacité de sauvegarde élevée. En conséquence, ces fondements se résument en quatre points :

— Deux types de tâches sont utilisés : les tâches Map et les tâches Reduce. Les tâches Map doivent s'exécuter avant les tâches Reduce associées au même travail. Une tâche Map traite seulement une tranche de données. Elle écrit en sortie des données nommées intermédiaires sur le support de stockage (généralement en mémoire sinon sur le disque dur). Chaque tâche Reduce récupère les données intermédiaires pour effectuer des traitements depuis les machines contenant les tâches Map. Elle génère un résultat qui sera écrit sur le support de stockage.

— La notion de localité des exécutions des tâches : l'exécution des tâches doit se faire le plus proche possible des machines contenant les données qu'elles traitent afin d'éviter le transfert des données à travers le réseau.

— La capacité de l'environnement à détecter les anomalies et à fournir des solutions : L'environnement doit posséder la capacité de détecter les tâches qui posent problèmes, et les relancer sur d'autres machines.
Le modèle intègre une politique de tolérance aux pannes. Le modèle MapReduce est destiné à faire des traitements sur des milliers de machines (serveurs) informatiques. Face à une telle taille de grappe, il utilise la redondance des données et sauvegarde l'historique des opérations système. Si les machines exécutant des tâches ne sont plus disponibles, le modèle s'occupe de relancer ces tâches sur d'autres machines pour assurer la progression et la finalisation de l'ensemble des travaux restant.

— L'environnement d'exécution des travaux MapReduce s'occupe d'exécuter les tâches en parallèle et coordonne leurs données d'entrée et de sortie.

Comme tout autre programme informatique, une tâche Map ou Reduce exploite les ressources de la machine sur laquelle elle s'exécute. La section 1.2.2 détaille les ressources utilisées par les

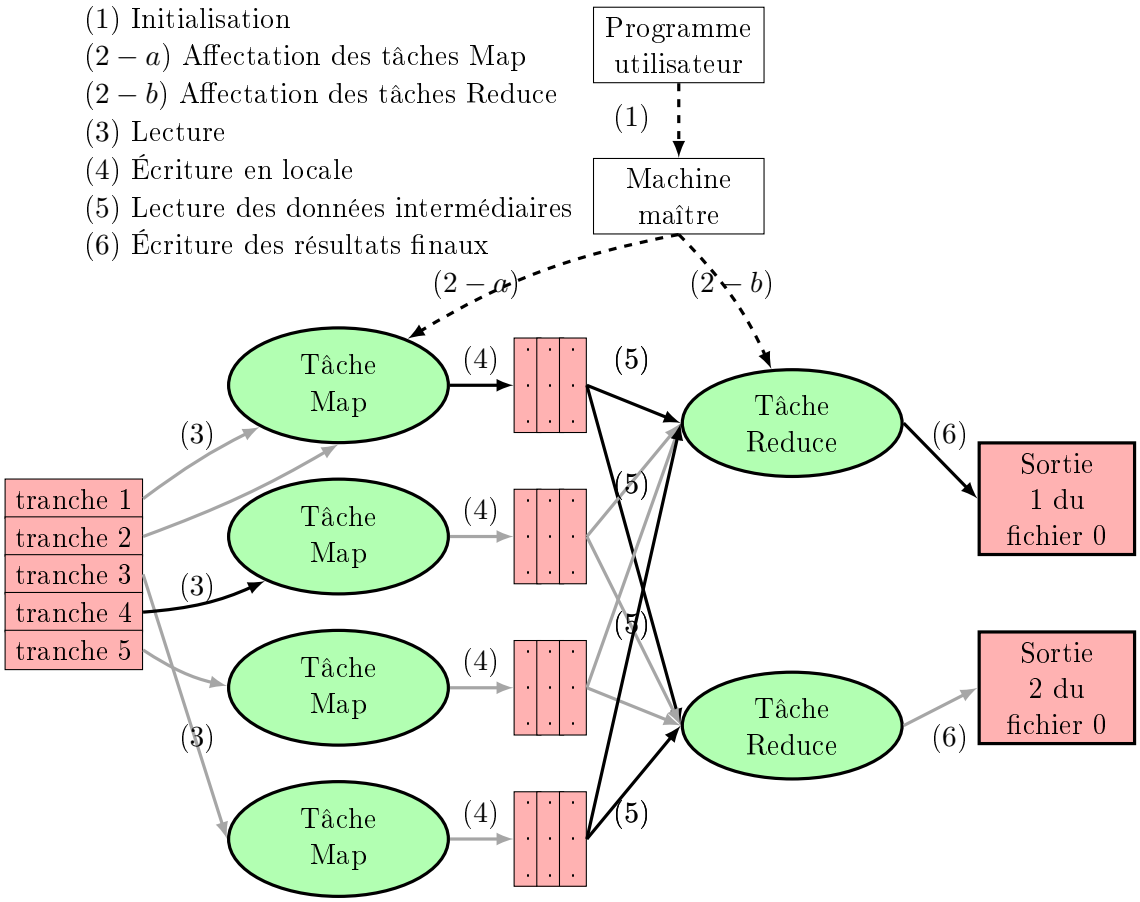


FIGURE 1.1 – Exemple d'exécution d'un travail MapReduce

tâches Map et Reduce dans le contexte de Hadoop.

L'environnement d'exécution des travaux MapReduce doit assurer les opérations suivantes (figure 1.1) :

1. Le fichier d'entrée est divisé en NB blocs de données de tailles similaires (de 16 à 128 Mb par portion en fonction de la configuration utilisateur). Elles sont distribuées sur les différentes machines de la grappe.
2. L'ordonnanceur s'occupe d'affecter les NB tâches Map (une tâche Map par bloc de données) et les R tâches Reduce aux unités de traitement.
3. L'unité de traitement exécutant une tâche Map formate le bloc de données qui lui est affectée. Elle fournit des paires de clé/valeur à la fonction Reduce. Les données de sortie de la fonction Map sont enregistrées en mémoire. Si la taille de ces données est supérieure à la taille de la mémoire-tampon, elle sera écrite sur le disque.
4. L'unité de traitement exécutant une tâche Map utilise une fonction de partition pour partager les paires de données intermédiaires en R ensembles. Ces derniers seront périodiquement enregistrés sur le disque dans R fichiers (un fichier pour chaque tâche Reduce). Quand la

tâche Map termine son exécution, l'unité de traitement associée envoie l'emplacement de ces données de sorties (les fichiers sur le support de stockage local) à l'ordonnanceur.

5. Chaque fois que l'ordonnanceur reçoit des informations après la fin d'une tâche Map, il informe les unités de calcul exécutant les tâches Reduce de la localisation des fichiers de sortie de la tâche Map. Après la lecture de toutes les données intermédiaires, elles les trient et les regroupent en fonction des clés intermédiaires. Cette étape est nommée « shuffle », c'est le système Hadoop qui est responsable d'assurer le transfert des données intermédiaires pendant cette étape.
6. Les unités de calcul des tâches Reduce formatent les données d'entrée en les fournissant en paramètre au programme Reduce (programme qui constitue le code utilisateur de la tâche Reduce). Les données de sortie du programme Reduce sont ajoutées à l'emplacement réservé à ce Reduce dans le système de fichiers distribué.
7. Après la fin de l'exécution de tous les programmes Map et Reduce, le processus appelant du travail récupère le contrôle et continue son exécution.

L'environnement MapReduce possède un rôle important, il prend en charge et garantit la bonne exécution des tâches. Les développeurs auront pour mission de se focaliser sur ce qu'ils savent faire, le développement des applications et l'amélioration de la qualité de leurs codes.

1.1.2 Caractéristiques des tâches de type Map et Reduce

Le modèle MapReduce est basé sur les deux types de tâches Map et Reduce. Ces tâches interagissent avec l'environnement pour récupérer les informations nécessaires à leurs exécutions telles que la présence des données d'entrée et leurs emplacements sur la machine. L'exécution de chaque tâche dépend des trois informations suivantes :

La génération des tâches : les tâches Map ou Reduce sont générées de façon statique ou dynamique. Par exemple, si le nombre de tâches Map et Reduce de chaque travail est défini par le développeur depuis l'écriture de son code, la génération des tâches est dite statique. Lorsque les programmeurs ne fournissent aucune information concernant le nombre des tâches, c'est le système Hadoop qui génère les tâches Map et Reduce associées à ce travail en utilisant des algorithmes dynamiques de génération. Dans ce cas, la génération des tâches est dite dynamique.

La taille de la tâche est le temps relatif nécessaire pour la traiter. La complexité de trouver une affectation (ou un ordonnancement) des tâches dépend de si oui ou non les tâches sont uniformes ; c'est-à-dire, si oui ou non elles exigent le même temps d'exécution. Si les tâches sont de durée non identique, alors, on dit qu'elles sont non uniformes.

Les tailles des données associées aux tâches : une autre caractéristique importante d'une tâche est la taille des données qui lui sont associées. La raison pour laquelle la taille des données est considérée comme importante pour l'ordonnancement, est que les données associées à une tâche doivent être disponibles lors de l'exécution de cette dernière. La taille

et l'emplacement de ces données peuvent déterminer l'unité de calcul qui peut exécuter la tâche sans encourir les pénalités dues aux mouvements excessifs des données.

Pour une tâche, il n'y pas de relation entre la taille des données d'entrée et la taille des données de sortie, cependant, la durée d'une tâche est proportionnelle à la taille des données d'entrée. Les données d'entrée peuvent avoir des petites tailles par rapport à la taille des données de sortie, ou vice-versa. Par exemple, dans le problème du calcul d'un minimum d'une séquence de nombre, la taille des données d'entrée est proportionnelle au nombre d'éléments de la séquence, cependant, la sortie de la tâche est un seul nombre. Dans la formulation parallèle de l'algorithme de tri rapide, les données d'entrée et de sortie possèdent la même taille et ont le même ordre que le temps nécessaire pour exécuter la tâche de façon séquentielle.

1.1.3 Contraintes de précédence entre les tâches Map et Reduce

L'un des fondements du modèle MapReduce est la relation de précédence entre les tâches Map et les tâches Reduce d'un même travail. Cette relation définit un certain ordre d'exécution. Par exemple la relation $i \rightarrow j$ désigne que la tâche j ne peut commencer avant la fin de la tâche i et se traduit par l'équation suivante :

$$S_i + p_i \leq S_j \tag{1.1}$$

avec S_i, la date du début de la tâche i et p_i, la durée de la tâche i.
Si on considère la situation suivante : on a n tâches qui ne sont pas interruptibles et on cherche à minimiser le $C_{max} = \max\{C_i\}$ avec $i \in [1 \dots n]$ et $C_i := S_i + p_i$. La relation 1.1 peut être généralisée à la relation :

$$S_i + d_{i,j} \leq S_j (d_{i,j} \in \mathbb{Z}) \tag{1.2}$$

Le nombre $d_{i,j} \in$ Z définit la position de la tâche i par rapport à la tâche j. Il permet à l'équation 1.2 de modéliser tous les types de relations entre les tâches. Par exemple, avec $d_{i,j} \geq 0$, j doit commencer $d_{i,j}$ unités de temps après la tâche i. Sinon, j commence $-d_{i,j}$ unités de temps avant la date de début de la tâche i. Pour $d_{i,j} = p_i$, on trouve la relation de précédence entre les tâches. On peut modéliser la date de début au plus tôt r_j et la date de fin au plus tard d_j en se basant sur l'équation 1.2. On suppose que $S_0 = 0$, r_j peut être modélisé en ajoutant la restriction $S_0 + r_j \leq S_j$ et d_j en ajoutant la restriction $S_i - (d_i - p_i) \leq S_0$. Si $r_j \leq d_j$, l'intervalle $[r_j, d_j]$ est appelé fenêtre de temps associée à la tâche j

Dans le cas d'un nombre infini de machines et lorsqu'on considère la minimisation du C_{max} comme critère à optimiser, le problème d'ordonnancement de n tâches possédant des contraintes de précédence entre elles est polynomial. Il suffit d'utiliser l'algorithme de Bellman et les deux techniques de gestion de projet CPM et PERT [21].

Dans le contexte de tâches MapReduce, cette relation fixe des règles reliées à leur ordre partiel d'exécution. La relation de précédence définit un ordre entre l'ensemble des travaux qui la constituent. Ces travaux sont reliés entre eux à travers un graphe acyclique direct [49] (un graphe de précédence avec une orientation dans un sens unique). En conséquence, la relation de précédence définit un ordre au niveau des tâches et au niveau des travaux. Les tâches Map et

Reduce associées à l'ensemble de ces travaux constituent à leur tour un graphe acyclique direct, ce graphe définit un ordre d'exécution global du flux de travail.

1.1.4 Différence entre ordonnancement hors ligne et ordonnancement en ligne

Il existe principalement deux types d'algorithmes de résolution des problèmes d'ordonnancement : les algorithmes hors ligne pour les problèmes hors ligne et algorithmes en ligne pour les problèmes en ligne.
Un algorithme est dit hors ligne [129] s'il a toutes les données pour calculer un ordonnancement avant son lancement.
Un algorithme est dit en ligne [129] s'il calcule une solution d'un ordonnancement par partie et seulement une partie des données est disponible à l'instant où l'algorithme commence son exécution. Il est en-ligne si des données sont connues pendant l'exécution de la séquence et permettent de re-calculer la séquence (ou la modifier).
Il arrive parfois que les données d'entrée d'un problème en ligne soient partiellement connues. L'algorithme en ligne est nommé dans ce cas algorithme semi-en ligne. Le manque d'information subi, dans le cas des problèmes d'ordonnancement en ligne, peut provenir de plusieurs faits. Par exemple :

- Les dates d'arrivées des tâches au cours du temps sont successives (une par une). La décision d'ordonnancement doit être prise sans connaissance des futures tâches à ordonnancer.
- Le temps de traitement d'une tâche n'est pas connu à l'avance ni durant son exécution. Il sera connu seulement à sa fin.
- L'intervalle d'indisponibilité d'une machine n'est pas connu.

Dans le cas de l'ordonnancement en ligne, savoir si une solution (séquence de tâches) est optimale avant de la lancer est impossible. Le problème d'ordonnancement des travaux MapReduce est connu pour son aspect en ligne. Nous envisageons d'optimiser l'exécution de travaux bien identifiés et dont les informations concernant la durée et les besoins en ressources des tâches Map et Reduce sont connus. Cependant, la date de soumission de chaque travail n'est pas connue à l'avance.
Après la présentation des fondements de base du modèle MapReduce, nous nous intéressons à son implémentation dans Hadoop.

1.2 Caractéristiques des grappes Hadoop

Afin de répondre au besoin de calcul et de traitement des données, Google introduit le modèle MapReduce [42], comme un modèle de développement des applications parallèles et distribuées, et GFS [62] comme système de fichiers distribués. GFS peut s'adapter aux besoins fonctionnels du modèle MapReduce. Depuis 2007, Yahoo a bénéficié des publications de Google pour le développement d'une version libre du modèle MapReduce et de son système de fichiers nommé Hadoop et HDFS. Nous focalisons sur le logiciel Hadoop et nous cherchons à améliorer sa politique

d'ordonnancement.
Hadoop est l'implémentation la plus utilisée du modèle MapReduce. Il est implémenté en Java, il est libre et soutenu par les plus grandes entreprises d'informatique [109]. Actuellement, il est dans sa deuxième version, cette version lui offre la possibilité d'aller plus loin que le modèle de développement MapReduce [152].

1.2.1 Architecture d'une grappe Hadoop

Hadoop est constitué de deux types d'éléments, les premiers sont responsables de l'exécution des tâches sur la grappe, les deuxièmes sont responsables de gérer le système de fichiers distribués (HDFS). Les deux types d'éléments possèdent une architecture client-serveur. La figure 1.2 présente l'architecture des entités de calcul Hadoop et de son système de fichiers HDFS. Les

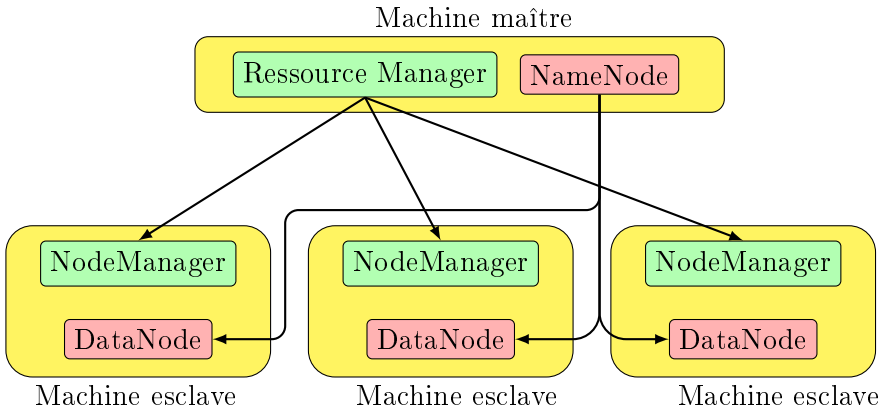


FIGURE 1.2 – Architecture client-serveur d'une grappe Hadoop

entités responsables de l'exécutions des tâches sont les suivantes :

— Un élément maître appelé « ResourceManager » : Il est responsable de la gestion des ressources de la grappe et de l'affectation des tâches sur ses ressources. Il implémente la politique d'ordonnancement des travaux associés aux applications utilisateurs sur les ressources.

— Un élément esclave appelé « NodeManager» : Il est associé à chaque machine esclave de la grappe. Il est responsable de l'exécution des tâches sur les ressources allouées par le « ResourceManager ». Il gère l'affectation et l'utilisation des ressources sur la machine dont il est responsable.

Les éléments responsables de la gestion du système de fichiers HDFS sont :

— Le « NameNode » est l'élément maître, il a la cartographie de tous les blocs de données et leur répartition sur la grappe. Il garantit le respect du nombre de répliquas de chaque bloc.

— Le « DataNode » est l'élément esclave, il assure la gestion des données sur chaque machine de la grappe Hadoop.

C'est l'ordonnanceur, au niveau « ResourceManager », qui gère l'affectation des ressources de la grappe entre les travaux. Il a toutes les informations concernant les ressources de la grappe, les travaux soumis et les besoins des tâches associées. Depuis la soumission d'un travail par l'utilisateur, les emplacements des blocs de données dont le travail a besoin sont récupérés du « NameNode » et envoyés à l'ordonnanceur. Ce dernier gère l'exécution des tâches associées aux travaux soumis.
Le paragraphe suivant explique les différentes notions liées à la gestion des ressources dans Hadoop.

1.2.2 Relation entre tâches, ressources et conteneurs dans Hadoop

Chaque machine de la grappe Hadoop possède un ensemble de caractéristiques physiques telles que : la quantité de la mémoire disponible, la capacité de calcul CPU, l'espace disque et la capacité de bande passante nécessaire pour l'échange de l'information avec les autres machines. Lors de l'exécution des travaux, Si la grappe fournit une puissance élevée de calcul, plusieurs travaux peuvent s'exécuter à la fois. Sinon, le système se trouve face à des problèmes de partage de ressources entre plusieurs travaux, utilisateurs ou organismes.

Cette section focalise sur la définition des ressources utilisées par les tâches Map et Reduce pour assurer leur exécution. Nous procédons de la façon suivante : nous définissons les différentes ressources utilisées par une tâche du modèle MapReduce, ensuite, nous présentons la notion de conteneur introduite dans l'implémentation Hadoop.

Notion de ressource

Les ressources sont les moyens utilisés par les tâches pour s'exécuter. Nous nous focalisons, particulièrement, sur les ressources matérielles.
Les versions actuelles des ordonnanceurs développés dans Hadoop gèrent le partage de la mémoire et la capacité de calcul CPU sur la grappe. Ils gèrent des mégaoctets MB de mémoire et des cœurs virtuels. En effet, la capacité de calcul sur la grappe est exprimée en nombre de cœurs virtuels, connus sous les noms de "Vcores" ou "slot" dans la communauté Hadoop.
Un "Vcore" ou "slot" [152] est une fraction fixe de la capacité de calcul d'une machine, elle est la réservation d'une puissance CPU pendant une durée indéterminée. Le nombre de cœurs virtuels par machine est fixé à travers les fichiers de configuration. Toutes les machines de la grappe, indépendamment de leurs configurations matérielles, possèdent le même nombre de cœurs virtuels.
En conséquence, la gestion de la ressource de calcul CPU sur une grappe présente un point de faiblesse puisque le critère d'hétérogénéité des machines pénalise les performances et dégrade la qualité des décisions prises par l'ordonnanceur.
La gestion de la mémoire dans Hadoop est plus optimisée que la gestion de la puissance de calcul, chaque tâche peut demander la quantité de mémoire nécessaire à son exécution. Des travaux futurs visent à ajouter la prise en compte de la gestion d'autres ressources telles que l'espace disque, la bande passante réseau et les GPU.

Afin de simplifier la gestion des ressources, la notion de conteneurs est introduite dans Hadoop depuis la deuxième version.

Notion de conteneur

Un conteneur est une collection de ressources physiques telles que la mémoire (en MB), les processeurs CPU (ou cœur virtuel), la quantité de disque et la bande passante réseaux. Un conteneur fournit des droits à un travail J pour utiliser des quantités spécifiques de ressources sur une machine particulière. Le conteneur est la réservation de quantités de ressources à une ou à l'ensemble des tâches associé à J.
Un conteneur Hadoop assure à une tâche des quantités de ressources (mémoire et Vcores) et l'environnement dont elle a besoin pour toute la durée de son exécution. Le nombre de conteneurs affecté à chaque travail et la durée de vie (ou période de réservation) de chaque conteneur sont gérés par l'ordonnanceur, ce dernier se trouve au niveau de l'élément "Ressource Manager". Les conteneurs permettent une meilleure gestion des ressources en améliorant l'isolation entre les différentes tâches qui s'exécutent sur la même machine.
L'utilisation des conteneurs dans Hadoop a commencé depuis la version 2 connue sous le nom de "Hadoop Yarn". Elle prend en considération deux types de ressources : la quantité de mémoire (RAM) exprimée en MB et la puissance de calcul (Vcores).
Sur le plan conceptuel, Hadoop suppose que la configuration physique de la grappe sur laquelle il est déployé est homogène. C'est-à-dire que, toutes les machines ont la même configuration matérielle. Les ordonnanceurs utilisés supposent que la grappe est homogène. Cette supposition permet aux applications utilisateur d'utiliser les ressources de la grappe de façon partagée et sécurisée. En conséquence, un ordonnanceur doit avoir une grande quantité d'informations concernant les différents travaux, leurs besoins en ressources et toute autre information concernant les données qu'elles traitent.
Suite à cette présentation de Hadoop, nous focalisons sur le problème d'ordonnancement.

1.3 Problème d'ordonnancement de travaux MapReduce

Dans le contexte du modèle MapReduce, un ordonnanceur fait face à trois problèmes principaux :

— Le choix de la tâche à exécuter,

— Le choix de la machine où chaque tâche s'exécutera,

— Le temps d'allocation des ressources à chaque tâche.

Dans cette section, des notations sont introduites et utilisées tout au long de ce manuscrit pour présenter le problème d'ordonnancement des travaux MapReduce. Les différentes contraintes modélisant le problème sont expliquées et les différents critères pris en compte durant l'optimisation du problème sont détaillés.

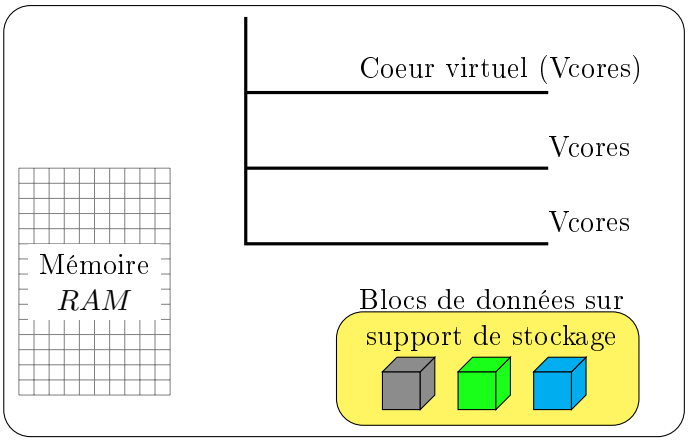


FIGURE 1.3 – Présentation des ressources disponibles sur une machine esclave Hadoop

1.3.1 Description du problème

Ce travail a pour objectif l'amélioration des performances de la plateforme Hadoop en améliorant sa politique d'ordonnancement. Le problème d'ordonnancement des tâches MapReduce dans le cas d'une grappe Hadoop dépend de plusieurs facteurs tels que le besoin en ressources des différentes tâches, la disponibilité des ressources sur les machines de la grappe et l'emplacement des données à traiter.

Description des différentes entités

Une grappe Hadoop est composée d'un ensemble de machines (serveurs) connectées. Chaque machine possède des capacités en ressources, par exemple (figure 1.3), une machine Hadoop est caractérisée par un nombre de cœurs virtuelles, noté "Vcores", un espace mémoire et un ensemble de blocs de données situés sur son disque dur.
Lorsqu'une tâche est affectée pour être exécutée sur une machine, elle effectue des traitements sur des blocs de données. Elle ne peut pas être interrompue, c'est-à-dire que l 'on ne considère ni la préemption ni les surcouts qu'elle engendre (la migration des traitements et des données). Dans la version actuelle du logiciel, une tâche Map dans Hadoop effectue des traitements sur un seul bloc de données [172]. Une tâche occupe un cœur virtuel et réserve un espace mémoire et un espace sur le support de stockage. De ce fait une machine Hadoop possédant suffisamment de ressources peut exécuter plusieurs tâches à la fois. Au maximum, une machine possédant trois cœurs virtuels peut exécuter trois tâches à la fois.

Relation entre les différents types de tâches

Les tâches Map s'exécutent avant les tâches Reduce. Les données des tâches Map sont transférées vers les tâches Reduce tel que représenté dans la figure 1.4, c'est-à-dire, les données, sortie d'une tâche Map, peuvent être transférées à travers le réseau vers une ou plusieurs tâches Reduce.

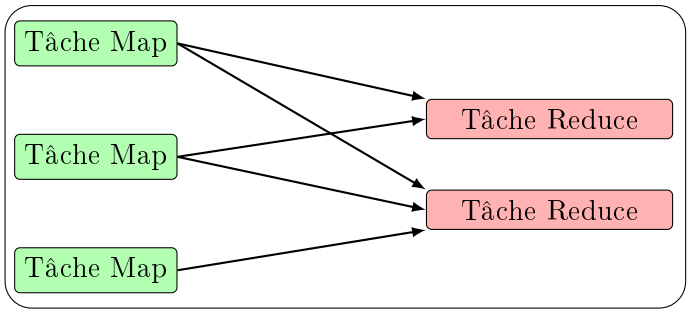


FIGURE 1.4 – Relation de précédences entre les tâches d'un travail MapReduce

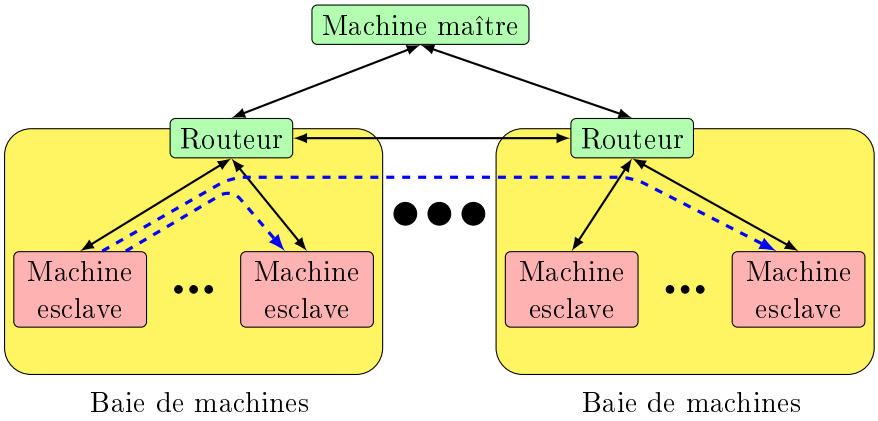


FIGURE 1.5 – Architecture réseau d'une grappe Hadoop

Description de l'architecture réseau

L'architecture d'une grappe Hadoop est synthétisée dans la figure 1.5. Par défaut, elle est constituée d'une arbre de profondeur trois. La racine est l'ensemble des machines contenant les éléments maîtres, les feuilles sont les machines esclaves. Les différentes machines appartenant à un même baie sont reliées à l'aide d'un routeur. Dans la figure 1.5, les flèches en pointillés désignent les chemins de transfert de l'information (données et communication) entre les machines de la grappe. Les flèches noires désignent les liaisons filaires.

1.3.2 Notations

Les notations associées à la définition des paramètres du problème sont accompagnées d'un ensemble des indices et d'un ensemble des exposants. Les indices sont utilisés pour parcourir les ensembles des entités et les exposants permettent d'indiquer à quelle ressource se rapporte ce paramètre. Contrairement aux indices, les exposants sont utilisés dans la définition du nom des paramètres (des labels sur les paramètres) et n'ont pas d'influence sur les résultats.
Nous commençons par présenter le tableau 1.1, la première colonne contient les notations les plus

utilisées pour les indices et les exposants. La deuxième contient le symbole associé à la famille d'indices. La troisième fournit la description des indices, exposants et ensembles définis.

TABLE 1.1 – Ensembles d'indices et d'exposants utilisés dans les notations

Indice	**Ensemble**	**Description**
w'	EW	indice sur l'ensemble de tous les travaux soumis (EW)
t	T	indice du temps pour un horizon de temps entre 1 et T
j, j'	M	indices sur les M machines de la grappe
s		indice des cœurs virtuelles (Vcores) des machines
	L^m	ensemble de tous les $N^m (= card(L^m))$ tâches "Map"
	L^r	ensemble de tous les $N^r (= card(L^r))$ tâches "Reduce"
i, l		indice sur les ensembles de tâches $L^m \cup L^r$
	N	nombre de tâches soumises ($N = N^m + N^r$)
	N^m	nombre de tâches "Map" soumises
	N^r	nombre de tâches "Reduce" soumises
b	A^b	indice sur l'ensemble A^b des blocs existants sur la grappe
u	E	indice sur l'ensemble des liaisons filaires entre les machines de la grappe
Exposant	**Description**	
ra	la notation qui utilise $'ra'$ en exposant aborde la manipulation de la ressource mémoire.	
h	la notation qui utilise $'h'$ aborde la manipulation de la ressource de sauvegarde.	
s	la notation qui utilise $'s'$ aborde la manipulation des cœurs virtuels.	
Sr	la notation qui utilise $'Sr'$ aborde la manipulation des Vcores qui exécutent des tâches "Reduce"	
Sm	la notation qui utilise $'Sm'$ aborde la manipulation des Vcores qui exécutent des tâches "Map"	
b	la notation qui utilise $'b'$ aborde la manipulation des blocs de données.	

Le problème que nous abordons consiste en une grappe constituée de M machines. Chaque machine M_j est caractérisée par *(1)* un ensemble de Vcores m_j^s dont m_j^{Sm} Vcores dédiés à l'exécution de tâches "Map" et m_j^{Sr} Vcores dédiés à l'exécution de tâches "Reduce", *(2)* une capacité m_j^{ra} de mémoire (RAM) et *(3)* une capacité m_j^h de disque dur. On considère un ensemble EW de W travaux de types Map / Reduce et un ensemble de blocs de données noté A^b. Chaque travail possède un poids $Weight_{w'}$ et est composé d'un ensemble $E_{w'}$ de tâches Map et de tâches Reduce.
Ces travaux sont décomposés en un ensemble de N tâches d'indice $i \in [1 \dots N]$. L'indice i est l'identifiant unique d'une tâche dans l'ensemble global des N tâches. Ce dernier est composé de deux sous-ensembles : L^m tâches "Map" et L^r tâches "Reduce", donc $N^m = card(L^m)$ tâches "Map" et $N^r = card(L^r)$ tâches "Reduce" avec $N = N^m + N^r$. Chaque tâche $T_i^{w'}$ est associée à un travail $w' \in EW$ dont la durée d'exécution est notée p_i. On note $b_{i,i'}$ la bande passante réservée pour assurer l'échange des données entre deux tâches $T_i^{w'}$ et $T_{i'}^{w'}$ qui s'exécutent sur deux

machines différentes. Pour chaque tâche $T_i^{w'}$, on considère E_i l'ensemble des n_i tâches qui doivent s'exécuter avant la tâche $T_i^{w'}$. Lorsque la tâche $T_i^{w'}$ s'exécute, elle demande une quantité n_i^{ra} de RAM, une quantité n_i^h de disque dur, un Vcore et traite un ensemble B_i de données divisées en $n_i^b \geqslant 1$ blocs lus sur le système de fichiers. Les notations utilisées pour la description des tâches sont d'ordre générique, nous les utilisons pour la description des tâches "Map" et "Reduce". Par exemple, dans un cas réel, chaque tâche "Map" traite un seul bloc ($n_i^b = 1$) et chaque tâche "Reduce" traite des blocs de données ($n_i^b \geqslant 1$) provenant de plusieurs tâches "Map" [152].
Tous les blocs de données ont la même taille S. On note r_b le nombre de réplications d'un bloc b. D_b est alors l'ensemble des machines qui contiennent le bloc b et bwd la bande passante allouée pour migrer un bloc d'une machine à une autre.
L'architecture de la grappe est modélisée par un graphe $G = (V, E)$ où l'ensemble des nœuds V présente les machines et l'ensemble des arcs E présente les liaisons filaires entre elles. On considère P l'ensemble des chemins entre les machines, P_u est l'ensemble des couples de machines qui utilisent l'arc e_u. Chaque chemin est composé de plusieurs arcs $e_u \in E$ dont la capacité maximale de bande passante b_{max} est identique pour tous les arcs. On note T l'horizon de temps nécessaire pour l'ordonnancement des tâches sur les machines.
Le tableau 1.2 présente un résumé de ces notations, ces dernières étant regroupées en familles en fonction des entités qu'elles décrient. La première colonne présente les notations des données, la deuxième présente leur description.

TABLE 1.2 – Les notations et les données

Notation	Description
Données associées aux machines	
m_j^s	nombre de cœurs virtuels sur la machine j
m_j^{Sm}	nombre de cœurs virtuels sur la machine j qui exécutent des tâches "Map"
m_j^{Sr}	nombre de cœurs virtuels sur la machine j qui exécutent des tâches "Reduce"
m_j^{ra}	capacité de mémoire (RAM) de la machine j
m_j^h	quantité de disque dur de la machine j
Données associées aux travaux	
EW	ensemble de tous les travaux soumis
W	nombre de travaux soumis
$E_{w'}$	ensemble des tâches appartenant au travail w'
$Weight_{w'}$	poids des tâches appartenant au travail w'
Données associées aux tâches	
n_i^{ra}	quantité de mémoire RAM nécessaire pour exécuter la tâche i
n_i^h	quantité de disque dur nécessaire pour exécuter la tâche i
n_i^b	nombre de blocs de données utilisés par la tâche i
B_i	ensemble des blocs de données que la tâche i traite
n_i	nombre de tâches qui doivent s'exécuter avant la tâche i
p_i	durée d'exécution de la tâche i
E_i	ensemble des n_i tâches qui doivent s'exécuter avant la tâche i

La deuxième partie est dans la page suivante

TABLE 1.2 – Les notations et les données

Notation	Description
$b_{i,i'}$	bande passante réservée pour assurer l'échange des données entre deux tâches i et i' si elles s'exécutent sur deux machines différentes.
Données associées aux blocs de données	
S	taille d'un bloc de données
r_b	nombre de réplications d'un bloc b
D_b	ensemble des machines qui contiennent le bloc b
bwd	bande passante allouée pour migrer un bloc d'une machine à une autre
Données associées à la description de l'architecture de la grappe	
$G = (V, E)$	le graphe présentant l'architecture de la grappe. L'ensemble des nœuds V présente les machines et l'ensemble des arcs E présente les liaisons filaires entre elles.
P	ensemble de chemins entre les machines
P_u	ensemble de couples de machines qui utilisent l'arc e_u
b_{max}	bande passante maximale des arcs de la grappe

Ces notations couvrent l'ensemble du problème et sont utilisées à travers tout le manuscrit.

1.3.3 Contraintes

Le modèle MapReduce et son implémentation Hadoop imposent certaines contraintes lors de l'exécution des tâches. Ces contraintes sont associées au problème d'ordonnancement que nous abordons et sont classées en quatre types :

Les contraintes de ressources sont reliées aux capacités des machines. Elles tiennent compte de la quantité des ressources consommées par les tâches à un instant t ; elle ne doit pas dépasser la capacité des ressources disponibles sur la machine où elles s'exécutent. Ces contraintes focalisent sur la gestion de la mémoire, des cœurs virtuels, de l'espace disque et la quantité des données transférées à travers le réseau à un instant donné. Les données transférées englobent la migration de bloc et l'échange des données entre les tâches.

Les contraintes reliées à l'exécution des travaux et des tâches focalisent sur :

- ▷ La date de disponibilité des travaux : l'ordonnancement des travaux est caractérisé par son aspect en ligne. La date de soumission des travaux n'est pas connue à l'avance donc les travaux ne sont pas disponibles à la même date. (idem pour les tâches)
- ▷ La nature des tâches : notons que les tâches Map et Reduce ne supportent pas la préemption. Cette contrainte classique dans l'ordonnancement parallèle et distribué suppose que si une tâche est affectée sur une machine, elle ne peut pas être interrompue. Si elle est interrompue, elle sera relancée comme si elle s'exécutait pour la première fois.

- ▷ La nature de la relation entre les tâches, les tâches sont reliées entre elles par des relations de précédences, ces relations organisent l'exécution des tâches, elles fournissent un ordre partiel par travail et un ordre global par flux de travaux.

Les contraintes reliées à la gestion des données : Elles spécifient l'emplacement des blocs et leurs répliquas sur les machines. Elles précisent la migration et le transfert des données d'une machine vers une autre et imposent qu'une tâche ne puisse s'exécuter que si le bloc associé est sur la machine où elle s'exécute.

1.3.4 Critères

Dans nos travaux, nous commençons par une modélisation simpliste et nous considérons un critère classique qui est la minimiser de la date de fin des travaux. Face aux cas d'utilisation rencontrés à l'entreprise Group-Cyres, lorsque plusieurs personnes utilisent la même grappe Hadoop, plusieurs travaux sont soumis. Ces derniers n'ont pas la même importance et n'appartiennent pas aux mêmes utilisateurs. Nous devons, en conséquence, privilégier les travaux les plus prioritaires sans pénaliser les travaux restants. L'idée est d'associer un poids à chaque utilisateur, ce poids est ensuite hérité par les travaux qu'il soumet.
De ce fait, l'étape suivante consiste à associer un poids à chaque travail et chercher à minimiser les deux critères d'importance égale suivants : *(i)* la somme pondérée des dates de fin des travaux et *(ii)* la date de fin du dernier travail. Nous cherchons à assurer que les travaux soumis soient terminés dans un temps raisonnable toute en privilégiant les travaux les plus prioritaires. C'est-à-dire, si on considère que $C_{w'}$ est la date de fin des tâches associées au travail w' et C_{max} la date de fin de tous les travaux, la fonction objectif est alors la minimisation de $\sum_{w'=1}^{W} Weight_{w'} C_{w'} + C_{max}$.

1.4 Conclusion

L'ordonnancement et la gestion des ressources sont des facteurs importants pour améliorer les performances des applications informatiques. Les grappes exécutent différents types d'applications et chacune a ses propres caractéristiques et besoins en ressources. L'ordonnancement des tâches dépend de plusieurs paramètres système tel que la vitesse CPU, la taille de la mémoire, la bande passante réseau, etc. Ces paramètres influencent le choix de la stratégie d'ordonnancement à utiliser. En fonction des caractéristiques du système [136] et la nature des travaux (préemptive ou non, basé sur des travaux de priorités égales ou différentes, etc.), plusieurs algorithmes statiques et dynamiques sont proposés.
Le chapitre 2 est une présentation des problèmes d'ordonnancement dans des plateformes informatiques, les spécificités de ce type de problèmes et les éventuelles méthodes de résolutions existantes.

Chapitre 2

Optimisation et ordonnancement

2.1 Introduction

Le présent chapitre est une étude dont le but est le recensement de notations, des méthodes et des algorithmes qui permettent de résoudre les problèmes d'optimisation et d'ordonnancement sur les grappes informatiques. Les lecteurs possédant ces notions de base peuvent entamer directement la lecture du chapitre suivant.
Ce chapitre est organisé en trois parties, nous commençons par une introduction aux problèmes d'optimisation et d'ordonnancement. Nous focalisons sur la modélisation des problèmes d'optimisation et d'ordonnancement, les critères d'optimalité et l'analyse de complexité des algorithmes. Ensuite, nous présentons des méthodes de résolution (exactes et heuristiques) telles que la programmation linéaire et les méthodes basées sur l'amélioration progressive (heuristiques). Enfin, nous focalisons sur la présentation des techniques utilisées pour réduire l'influence du transfert des données sur l'exécution des travaux en mode distribué.

2.2 Les problèmes d'optimisation

La résolution des problèmes d'optimisation commence par l'étude et l'identification du problème, ensuite, la construction et la résolution du modèle et enfin l'évaluation et l'implémentation de la solution [130].

Étude et identification du problème.

L'identification d'un problème consiste à décrire les décisions à prendre, les contraintes, les critères d'évaluation des différentes décisions prises et les objectifs finaux que la solution doit permettre d'atteindre. Ces objectifs sont, soit une solution optimale, soit, une solution approchée. Dans la plupart des cas, la solution approchée doit, soit, fournir de bons résultats face aux critères d'évaluation, soit, fournir un résultat meilleur que celui fourni par la solution existante. Il est important de fournir la définition d'un problème la plus large possible pour assurer l'obtention de résultats pertinents et il est, également, important de fournir la définition la plus

restreinte et simple afin de le résoudre. Le résultat de cette première étape est souvent une description simplifiée du problème.

La construction et la résolution du modèle.

Un modèle est une présentation, généralement simplifiée, du monde réel. Il consiste à représenter un problème ; il est une abstraction de ses propriétés importantes.
Les tests intermédiaires sont utiles pour confirmer la pertinence du modèle choisi et la qualité de ses données d'entrée. Le modèle défini est ensuite résolu par des algorithmes d'optimisation. La construction d'un modèle, par exemple, mathématique possédant un niveau d'abstraction acceptable est une tâche difficile, qui commence souvent avec une modélisation réelle mais complexe et insolvable par les solveurs mathématiques dans un temps acceptable. De façon itérative, on simplifie le modèle pour le rendre plus facile à résoudre avec les méthodes d'optimisation existantes. La simplification du modèle peut consister à négliger des contraintes du problème réel. À ce niveau, on identifie un compromis entre la capacité des méthodes d'optimisation à résoudre le modèle (la traçabilité du modèle) et la capacité du modèle à présenter le problème réel (la relative fidélité du problème).
Dans la plupart des cas, un modèle est choisi en fonction des méthodes d'optimisation qui peuvent le résoudre. Dans la pratique, les utilisateurs pensent que ces méthodes sont regroupées dans une sorte de boîte noire et utilisées pour tous les problèmes de façon similaire. Cette optique peut être appliquer dans le cas des méthodes classiques, qui nécessitent un nombre réduit d'interventions pour les adapter à un problème spécifique. Cependant, les méthodes heuristiques ne peuvent pas appartenir à cette boîte noire [6] car elles montrent généralement des limites en performances (temps nécessaire pour touver une solution) ou en qualité des solutions trouvées. Les méthodes heuristiques doivent être appliquer aux domaines auxquels elles sont créées. Seule, l'utilisation des heuristiques spécifiques au problème appartenant au bon domaine d'application fournit une solution de bonne qualité [46][127].

La validation et l'implémentation de la solution.

Après la résolution du modèle, toutes les solutions trouvées, optimales ou approchées, sont évaluées. Les analyses effectuées visent à étudier comment chaque solution obtenue dépend de la variation des paramètres et des données d'entrée du modèle. Plusieurs méthodes sont utilisées, nous présentons trois d'entre elles. La première consiste en la réalisation de tests prospectifs. Chaque test se base sur les données historiques pour évaluer le modèle et comparer les solutions obtenues par rapport aux solutions précédentes. Ce type de test est utilisé pour valider les solutions du modèle et mesurer le gain obtenu suite à l'utilisation des nouvelles solutions. De façon générale, l'utilisation des données historiques ne garantit pas une évaluation sûre des solutions, elle fournit une idée approximative sur leurs qualités.
La deuxième méthode consiste à utiliser un environnement maîtrisé pour faire l'évaluation. C'est-à-dire, utiliser des configurations et des instances de petite taille, comme des données d'entrée, pour l'évaluation du modèle et des solutions. Il est essentiel d'avoir, en avance, une idée sur les résultats à obtenir lors de l'utilisation de cet environnement.

La troisième méthode s'appelle l'analyse de compétitive [139]. On suppose que pour un problème d'ordonnancement particulier, on cherche à minimiser sa fonction objectif et on a deux algorithmes calculant une solution pour ce problème, un algorithme en ligne A qu'on veut comparer à un algorithme hors ligne OPT (qui donne toujours une solution optimale). On suppose que A (I) et OPT (I) présentent respectivement les fonctions objectifs (des solutions) générées par l'algorithme A et par l'algorithme OPT pour une instance donnée I du problème.
L'algorithme A est appelé « c-compétitive » [143] s'il existe deux constantes c et k qui vérifient $A(I) \leq c.OPT(I) + k$ pour toute instance I. La constante c ainsi définie est appelée le rapport de compétitivité de l'algorithme A [143].

2.3 Les problèmes d'ordonnancement

Les problèmes d'ordonnancement sont un sous ensemble des problèmes d'optimisation. Il existe dans la littérature plusieurs définitions du problème d'ordonnancement, par exemple, celle présentée dans [122] : un problème d'ordonnancement est l'allocation d'une quantité limitée de ressources à un ensemble de tâches à travers le temps. C'est un processus de prise de décision qui a comme but l'optimisation d'un ou plusieurs objectifs.
Les problèmes d'ordonnancement appartiennent à la classe des problèmes d'optimisation où on cherche à minimiser (ou maximise) une fonction objectif C. Pour chaque problème d'ordonnancement, on peut associer un problème de décision défini de la façon suivante : pour un seuil donné y, est-ce qu'il existe une solution faisable S telle que $C(S) \leq y$? Un ordonnancement faisable S (avec $C(S) \leq y$) fournit un certificat pour les instances "oui".
Parmi les différents domaines d'application cités de l'ordonnancement, nous nous intéressons aux problèmes d'ordonnancement dans les systèmes informatiques. Ces problèmes sont définis comme des problèmes d'ordonnancement avec des contraintes de ressources, leur objectif est de trouver une solution faisable qui satisfait un ensemble de contraintes.
Dans le cadre de l'ordonnancement à machines parallèles, chaque machine possède un ensemble de ressources qui doivent être utilisées pour exécuter les tâches. On peut traiter ces ressources selon la façon dont les tâches les exploitent et selon leurs disponibilités à travers le temps. On peut organiser les ressources par classes et par types. On considère deux classes de ressources :

1. les ressources disjonctives sont des ressources affectées à une seule tâche à la fois,
2. les ressources cumulatives sont les ressources qui peuvent être consommées par plusieurs tâche simultanément.

On considère deux types de ressources :

1. les ressources renouvelables sont des ressources $k = 1, \ldots, r$ de quantité R_k, disponibles tout le temps,
2. les ressources non-renouvelables sont disponibles avec des quantités limitées sur l'horizon de temps. Au fur et à mesure qu'une tâche J s'exécute et consomme une quantité r_{jk} de la ressource k, la quantité de la ressource R_k disponible décroît de r_{jk}.

Dans un processus d'ordonnancement, une tâche est une tâche qu'on doit ordonnancer pour être exécutée sur les machines de la grappe. Elle peut s'exécuter suivant deux modes différents :

1. le monomode : une tâche j doit être exécutée pendant p_j unité de temps. La durée p_j est constante, fixe et non variable durant le processus d'ordonnancement. On peut définir pour chaque tâche la date de début r_j, la date de fin d_j au plus tard et un poids w_j. La tâche j ne peut commencer qu'après sa date de début et doit finir son exécution avant la date de fin d_j. Si $r_j \leq d_j$, l'intervalle $[r_j, d_j]$ est appelé fenêtre de temps associée à j. On dit qu'on n'a pas de date de début si $r_j = 0$ et on n'a pas de date de fin au plus tard si $d_j = \infty$. Le poids w_j définit la priorité entre les tâches.
2. Le multimode : Une tâche qui s'exécute en multimode possède une durée d'exécution variable selon le mode utilisé, c'est-à-dire, qu'on associe un ensemble M_j de modes définis à chaque tâche j. Le temps de calcul nécessaire pour une tâche j dans le mode m est p_{jm} et la quantité de ressources renouvelables consommées par j dans le mode m est r_{jkm}. La solution du problème d'ordonnancement dans cette situation doit affecter un mode à chaque tâche et ordonnancer les tâches dans le mode choisi. On définit r_{jm}, d_{jm}, w_{jm} la date de début au plus tôt, la date de fin au plus tard et le poids de la tâche j dans le mode m.

Une tâche peut être interruptible ou non. Une tâche interruptible est celle qui peut être interrompue et reprise ultérieurement avec ou sans migration vers une autre machine.

2.3.1 Les types d'ordonnancement

Un problème d'ordonnancement S peut être présenté comme un quadruplet $S = (I, M, R, \varphi)$ où I est l'ensemble des tâches, M est l'ensemble des machines, R est l'ensemble des autres ressources nécessaires à l'exécution des tâches et φ est la fonction objectif à optimiser. On peut trouver différents ordonnancements pour le problème S. On définit $Z(S)$ l'ensemble des solutions trouvées pour le problème d'ordonnancement S. Cet ensemble de solutions peut contenir plusieurs types d'ordonnancements. On présente dans cette sous-section les différents types d'ordonnancements possibles :

— On définit un ordonnancement comme faisable si toutes les contraintes de ressources et de précédence sont respectées. L'ensemble des solutions faisables est noté $Z_{feas}(S)$.

— On définit un ordonnancement comme semi-actif, s'il est obtenu d'un ordonnancement faisable en déplaçant les tâches pour commencer le plus tôt possible sans violation des contraintes de précédence ni de la date de début r_j. L'ensemble des solutions semi-actives est noté $Z_{s-act}(S)$.

— On définit un ordonnancement actif, s'il est obtenu d'un ordonnancement semi-actif en déplaçant les tâches pour commencer le plus tôt possible sans violation des contraintes de précédence ni de la date de début r_j. L'ensemble des solutions actives est noté $Z_{act}(S)$.

— On définit un ordonnancement comme optimal si la valeur de la fonction définissant le critère d'optimalité est optimale. L'ensemble des solutions optimales est noté $Z_{opt}(S)$.

— Un ensemble d'ordonnancement est dit dominant s'il contient au moins un ordonnancement optimal.

Suite à ces définitions, on obtient la relation suivante entre les ensembles : $Z_{act}(S) \subseteq Z_{s-act}(S) \subseteq Z_{feas}(S) \subseteq Z(S)$.

2.3.2 Définition du critère d'optimalité

À chaque problème d'ordonnancement S, on associe une fonction qui permet la définition du critère d'optimalité des solutions à trouver. Cette fonction est définie pour l'ensemble des solutions faisables $Z_{feas}(S)$. Elle vise à définir le critère à optimiser et elle est choisie selon les objectifs du problème d'ordonnancement.
Soit $C_1, C_2, ..., C_n$ l'ensemble des dates de fin des tâches dans un ordonnancement. Une fonction de définition de critère d'optimalité $f(C_1, C_2, ..., C_n)$ peut être l'un des critères suivants ou la combinaison de plusieurs de ces critères :

- La date de fin d'ordonnancement (makespan) $C_{max} = \underset{1 \leq j \leq n}{max} \{C_j\}$.
- La date de fin moyenne $\sum C_j = \sum_{j=1}^{n} C_j$.
- La date de fin moyenne pondérée $\sum w_j C_j = \sum_{j=1}^{n} w_j C_j$.
- Le plus grand retard algébrique $L_{max} = \underset{1 \leq j \leq n}{max} \{L_j\}$ avec $L_j = C_j - d_j$.
- La sommes des retards $T_{max} = \underset{1 \leq j \leq n}{max} \{T_j\}$ avec $T_j = \underset{1 \leq j \leq n}{max} \{0, L_j\}$.
- Le coût maximal $f_{max} = \underset{1 \leq j \leq n}{max} \{f_j(C_j)\}$ avec $f_1, f_2, ..., f_n$ sont des fonctions qui expriment les coûts.
- Le coût total $\sum f = \sum_{j=1}^{n} f_j(C_j)$ avec $f_1, f_2, ..., f_n$ sont des fonctions qui expriment les coûts ;
- Le nombre des tâches en retard $\sum U_j = \sum_{j=1}^{n} U_j$ avec $U_j = 0$ si $L_j \leq 0$ et $U_j = 1$ si $L_j > 0$.

Toutes ces fonctions objectifs sont monotones, non décroissantes par rapport à C_j. On dit que f est une fonction régulière si $f(C_1, C_2, ..., C_n) \leq f(C'_1, C'_2, ..., C'_n)$ avec C et C' des vecteurs de dates de fin possédant les critères suivants $C_j \leq C'_j$ [103].
Tous les critères d'optimalité, présentés précédemment, peuvent être associés à des problèmes mono-objectifs si ces derniers cherchent à optimiser un seul critère. Combiner plusieurs critères d'optimalité est une solution possible, un problème qui dépend de plusieurs critères est nommé multi-objectives.

2.3.3 Analyse de la complexité des algorithmes

Il est fréquent d'évaluer les heuristiques, soit en utilisant le ratio d'approximation, soit en les comparant aux performances fournies par d'autres heuristiques dans le cas où la solution optimale ne peut pas être calculée. Le ratio de performance est une mesure pour évaluer la qualité des algorithmes d'approximations dans le pire cas. Il est le rapport entre la solution fournie par l'algorithme et la solution optimale. Il est connu sous le nom de ratio de performance au pire des cas et ratio d'approximation. Il est important que le ratio de performance d'un algorithme soit bon et constant. Pour un problème donné Π : s'il n'y a pas d'algorithme avec un ratio de performance borné (limité), on dit que Π est non approximable. Si un algorithme avec un ratio

constant existe, on dit que Π est approximable [12] [147].
La théorie de la complexité [59] [118] propose un ensemble d'outils qui fournissent une idée sur les méthodes de résolution qu'on peut utiliser pour trouver une solution à un problème particulier. Elle se base sur la transformation d'un problème d'optimisation en un problème de décision, ensuite, trouver la complexité de ce dernier. Chaque problème est associé à une classe de complexité, elle nous fournit des informations concernant la complexité du meilleur algorithme capable de le résoudre. La complexité d'un algorithme consiste à estimer son temps de calcul ou l'espace mémoire que l'algorithme utilise.
Malgré l'importance de la complexité spatiale pour certaines classes d'algorithmes, telles que les algorithmes de la programmation dynamique (section 2.4.1), elle est moins prise en compte par rapport à la complexité temporelle. Dans le reste de ce manuscrit, nous utilisons le terme "complexité d'un algorithme" pour désigner la complexité temporelle, c'est-à-dire, le nombre d'opérations effectuées par un algorithme durant son exécution.
Il existe deux types de complexités, la complexité théorique et la complexité pratique. La complexité théorique évalue l'algorithme indépendamment de la configuration de la machine qui l'exécute. La complexité pratique nous permet de calculer le coût de l'algorithme en fonction de la machine qui l'exécute. L'avantage de la complexité théorique est qu'elle peut être utilisée comme référence puisqu'elle est indépendante de la machine. Nous utilisons la complexité théorique pour évaluer les heuristiques que nous proposons.
Si le temps d'exécution d'un algorithme est borné par une fonction polynomiale en fonction de n (taille des données d'entrée) et de complexité de l'ordre de $\mathcal{O}(n^k), k \in \mathbb{N}$, l'algorithme est dit polynomial. Si le temps d'exécution de l'algorithme est borné par une fonction polynomiale en fonction de la taille, en unaire, des données d'entrée, l'algorithme est nommé pseudo-polynomial [19]. Autrement, on dit que l'algorithme est exponentiel. La taille en unaire d'une instance est le nombre de bits nécessaires pour écrire cette instance.
Le livre de T'Kindt et al. [143] aborde la complexité des problèmes et des algorithmes. Il synthétise et présente les différentes classes de complexité introduites dans la théorie de la complexité.

2.4 Les méthodes de résolutions

Nous présentons, dans cette section, les méthodes utilisées pour la résolution de problèmes d'optimisation combinatoire et particulièrement les problèmes d'ordonnancement. Nous organisons ces différentes méthodes en deux groupes :

1. Les méthodes exactes sont les méthodes qui visent la recherche de la solution optimale. On présente trois méthodes suivantes :
   - La programmation linéaire.
   - La méthode par séparation et évaluation.
   - La programmation dynamique.
2. Les méthodes basées sur l'amélioration progressive d'une solution courante. Vu que la plupart des méthodes exactes proposent des algorithmes exponentiels pour les problèmes $NP - difficiles$, le temps de calcul de la solution est donc exponentiel et la méthode

n'est généralement pas utilisable dans le cas de problèmes réels. À partir d'une solution initiale déterminée par une heuristique, ces méthodes régénératrices cherchent à améliorer, de façon itérative, la solution initiale en appliquant des transformations locales. Cette exécution itérative se poursuit jusqu'à la satisfaction d'un critère d'arrêt. Nous présentons les méthodes suivantes :

— Méthode de recuit simulé

— Méthode de Glouton

— Méthode de recherche tabou

Il existe plusieurs autres métaheuristiques non abordées dans cette section telles que l'algorithme des colonies de fourmis [35]. Les algorithmes de colonies des fourmis sont introduits par Colorni et al. [35] en 1992 et appliqués pour la première fois au problème du voyageur de commerce.

2.4.1 Méthodes exactes

Ces algorithmes fournissent des garanties pour trouver la solution optimale. Les algorithmes exacts ne sont pas utilisés dans le cas des applications qui demandent des temps de réponse réduits. La principale cause est que ce type de méthode explore la majorité de l'espace de recherche pour identifier la solution optimale. De ce fait, plus le nombre d'instances est grand, plus complexe est la résolution de ces problèmes. Parmi les différentes méthodes existantes, on présente dans cette partie les trois méthodes suivantes :

Programmation Linéaire

Cette méthode (PL) [156] est la programmation mathématique la plus utilisée. Elle fait partie des méthodes d'optimisation dans lesquelles la fonction objectif et les contraintes du problème sont linéaires en fonction des variables de décisions. Si on considère que la fonction objectif est une fonction de minimisation, la forme générale d'un programme linéaire est la suivante :

$$\begin{aligned} &\text{Minimiser } \bar{c}\bar{x} \\ &A\bar{x} \geq \bar{b} \\ &\bar{x} \geq 0 \end{aligned}$$

A est une matrice de dimension $n \times N$, où n est le nombre de tâches et N est le nombre de ressources [122]. b est un vecteur colonne de taille N connu sous le nom de vecteur de ressource. La composante $\bar{x}$ est l'ensemble des variables du problème d'optimisation dont la taille est N.
Lorsque les variables, d'un programme linéaire, sont forcées à être des entiers, on parle de programme linéaire en nombre entier ($PLNE$), la forme générale d'un ($PLNE$) [133] est la suivante :

$$\begin{aligned} &\text{Minimiser } \bar{c}\bar{x} \\ &A\bar{x} \geq \bar{b} \\ &\bar{x} \in \mathbb{N}^n \end{aligned}$$

De façon générale, la relaxation des contraintes de précédences donne des bornes inférieures basées sur la consommation des ressources et la relaxation des contraintes de ressources donne des bornes inférieures du chemin critique.
La relaxation de contraintes dans un modèle permet de le simplifier. Ainsi, la solution optimale du problème relâché est une approximation pour le problème d'origine. Plusieurs méthodes de relaxation sont connues dans la littérature, on peut citer la relaxation linéaire et la relaxation lagrangienne [61]. Les différentes relaxations sont utilisées pour calculer des bornes inférieures. La résolution du programme relâché, d'un $PLNE$ par exemple, aboutit au calcul d'une solution entière présentant une borne inférieure proche de la solution optimale du problème d'origine.
La programmation linéaire est largement utilisée dans la littérature pour la résolution des problèmes basés sur la gestion des ressources. La formulation utilisée lors de la modélisation d'un problème a un rôle important lors de la résolution du modèle, le nombres de variables et contraintes générés durant la résolution peuvent être polynomiaux [13], pseudo-polynomiaux [3] [14] [32] ou exponentiels [111] [115]. Deux classes de formulation de ces problèmes [94] sont les plus utilisées :

— La classe des modèles qui se basent sur la discrétisation du temps. Ce type de modèle est utilisé pour faciliter la modélisation des contraintes de ressources.

— La classe des modèles qui se basent sur la gestion des évènements : ce type de modèle considère que le début et la fin d'une tâche sont associés au déclenchement de deux évènements. Un évènement est capable, d'une part, de récupérer les ressources d'une tâche, d'autre part, de les affecter à la tâche suivante. Ce type de modélisation a la capacité de réduire le nombre de variables et de contraintes.

Depuis que les programmes linéaires peuvent être résolus de façon efficace et simple, certaines recherches se sont orientées à leur utilisation dans le développement des bornes, des approximations et des heuristiques pour les problèmes d'ordonnancement NP-difficile.

La méthode de séparation et évaluation (Branch and Bound)

Cette méthode repose sur une méthode de recherche arborescente. Elle est constituée de deux mécanismes : la séparation et l'évaluation. Elle est connue sous le nom de "Branch and Bound" ($B\&B$) [33] [98] [119].
La séparation est une méthode de partitionnement de l'ensemble des solutions possibles en c sous-ensembles (c est une constante) dont chacune est parcourue pour chercher une solution meilleure que la solution existante. Ensuite, par un traitement récursif, les sous-ensembles $S_i, i \in [1 \dots c]$ sont partitionnés, à leur tour, en sous-ensembles. Si pour une partition S_i, on est sûr qu'on ne peut pas améliorer la solution existante alors S_i est rejetée.
Généralement, l'étape de séparation est modélisée sous la forme d'un arbre de recherche dont la racine correspond à l'ensemble de l'espace des solutions d'un problème Π noté $Z_\Pi(I)$, I étant l'instance d'entrée du problème. Les descendants de la racine sont les sous-ensembles de $Z_\Pi(I)$. De manière analogue, si un nœud de l'arbre est un sous-ensemble $S_i \in Z_\Pi(I)$ alors ces nœuds descendants sont les sous-ensembles de S_i. La racine de l'arbre est la solution vide qui présente toutes les solutions qui peuvent être trouvées. Les nœuds intermédiaires de l'arbre peuvent contenir des solutions partielles et les feuilles contiennent des solutions complètes.

L'évaluation est une étape qui consiste à parcourir l'arbre pour, soit, chercher les solutions associées à chaque nœud de l'arbre, soit, pour prouver que le nœud ne contient pas des solutions qui peuvent améliorer la solution existante.
Au fur et à mesure que l'opération de recherche progresse en profondeur, l'ensemble des solutions est progressivement construit. Cependant et en fonction de la taille de l'arbre, le nombre de nœuds peut exploser de façon exponentielle. À ce niveau, on identifie le besoin à des méthodes d'évaluation permettant de supprimer des branches de l'arbre qui ne peuvent pas améliorer la solution en cours.
Pour plus de détails concernant les techniques de réduction de l'espace de recherche dans les arbres de recherche, vous pouvez consulter les références suivantes [33] [98] [119], une version parallèle du $B\&B$ est définie dans [41].

La méthode de programmation dynamique

La programmation dynamique se base sur les principes d'optimalité de Bellman [17][16] appliqués aux problèmes de décisions. Généralement, avec l'approche de programmation dynamique, on utilise une approche ascendante qui commence à partir des sous-problèmes pour résoudre le problème principal.
Les principes de Bellman peuvent être utilisés pour construire une trajectoire optimale d'un état initial A à un autre état B. Il est souvent possible de calculer les caractéristiques de l'état en cours à partir des états précédents dans le chemin optimal. On peut commencer par un état initial (A) où aucune décision n'est prise et on essaie de trouver le chemin optimal vers un état (C) final accessible à partir de l'état initial. On analyse implicitement tous les chemins possibles de l'état initial ou aucune décision n'est encore prise vers l'état final où la solution finale est construite. Cette analyse est la manière optimale pour progresser d'un état vers un autre. Dans le cas des problèmes combinatoires. Le nombre de chemins est exponentiel et les algorithmes basés sur la programmation dynamique pour ces problèmes sont des algorithmes exponentiels. Pour plus de détails concernant la programmation dynamique, vous pouvez consulter les travaux de Cormen et al. [38].

2.4.2 Les méthodes basées sur l'amélioration progressive de la solution

Il est en général difficile de trouver des solutions optimales pour les problèmes NP-difficiles. Les algorithmes de résolution existants sont exponentiels et ne peuvent résoudre que des petites instances. Les algorithmes d'approximation et les heuristiques sont utilisés pour apporter des solutions à des problèmes réels auxquels une bonne solution rapide n'est ni existante ni simple à obtenir. Ces méthodes fournissent des solutions approchées qui n'ont pas de garanties pour être optimales. Si la borne inférieure est disponible, pour les problèmes de minimisation (ou de maximisation), l'écart entre la valeur objectif et la borne inférieure (ou supérieure) peut être calculé et l'heuristique peut en conséquence être évaluée. Dans cette section, nous discutons de trois métaheuristiques générales pour résoudre les problèmes d'optimisation : le recuit simulé, la méthode de glouton et la recherche tabou.

Recuit simulé

Cette métaheuristique est une méthode probabiliste qui a pour objectif la recherche d'un minimum global de la fonction objectif tout en évitant les minimas locaux. Elle est proposée par Kirkpatrick et al. [91] et Cerný et al. [22] et développée par Metropolis et al. [110].
L'algorithme 1 présente le recuit simulé. Après la génération d'une solution initiale, à l'aide d'une heuristique appropriée ou de façon aléatoire, l'algorithme cherche à l'améliorer. $c(s)$ est la fonction objectif associée à s, la fonction *Rand* est responsable de la génération d'un nombre aléatoire entre 0 et 1. La recherche peut être arrêtée après un certain nombre d'itérations, après un certain nombre de solutions non-améliorantes, ou quand un délai donné est atteint.
Supposons que S est l'ensemble des solutions et $N(s), s \in S$ est le sous-ensemble de solutions qu'on peut atteindre à partir de s ($N(s)$ est un sous-ensemble des voisinages de s). Cette méthode génère aléatoirement une solution $s^{'} \in N(s)$, ensuite, à l'aide d'une formule probabiliste, elle décide de l'acceptation ou du rejet de l'élément généré. En outre, pour une i^{eme} itération par exemple, $s^{'}$ est accepté avec la probabilité $min\{1, \exp^{-\frac{c(s^{'})-c(s)}{t_i}}\}$, t_i est une séquence de valeur positive avec $\lim_{i \longrightarrow \infty} t_i = 0$, elle est définie par $t_{i+1} = \alpha t_i, 0 < \alpha < 1$.

Algorithme 1 : Algorithme de recuit simulé général

```
1  Générer la solution initial s ∈ S ;
2  best := c(s) ;
3  s* := s ;
4  i := 0;
5  répéter
6  |  Générer aléatoirement une solution s' ∈ N(s) ;
7  |  si Rand(0,1) < min{1, exp^{-(c(s')-c(s))/t_i}} alors
8  |  |  s := s' ;
9  |  |  si c(s') < best alors
10 |  |  |  s* := s' ;
11 |  |  |  best := c(s') ;
12 |  i := i + 1 ;
13 jusqu'à ce qu'une condition d'arrêt soit satisfaite
```

L'acceptation de la solution $s^{'}$ dépend de : *(i)* si $c(s^{'}) \leq c(s)$ alors $\exp^{-\frac{c(s^{'})-c(s)}{t_i}} = 1$ et $s^{'}$ est acceptée. Ou, *(ii)* si $c(s^{'}) > c(s)$ alors $s^{'}$ est acceptée avec une probabilité qui diminue avec l'augmentation de i. Cela veut dire qu'un minimum local peut être accepté, mais la probabilité d'acceptation diminue au cours des itérations.

Méthode glouton

Dechter et al [44] définissent la méthode gloutonne comme une stratégie de recherche contrôlée. Un algorithme glouton construit une solution pas à pas, en effectuant à chaque étape le choix

qui semble être le meilleur et sans la garantie de trouver un résultat optimum global. L'idée de cette méthode se base sur le fait que : " Le candidat avec le maximum de profit parmi plusieurs est sélectionné". Cette méthode fait toujours le choix qui semble être le meilleur à cet instant, c'est-à-dire, elle effectue un choix optimal localement dans l'espoir que ce choix conduira à une solution optimale globale.
Les algorithmes Gloutons ont des avantages et des inconvénients :

- Il est assez facile de trouver un algorithme glouton (ou même plusieurs algorithmes gloutons) pour un problème.
- Analyser le temps d'exécution des algorithmes glouton est généralement beaucoup plus facile par rapport à d'autres techniques, telles que la méthode de séparation et d'évaluation.
- Il est difficile de prouver l'efficacité d'un algorithme glouton.
- Dans un algorithme glouton, on commence par prendre le choix qui semble être le meilleur à l'heure actuelle, ensuite, on résout le sous-problème résultat de ce choix. De ce fait, la solution candidate trouvée est seulement évaluée localement. En conséquence, même si on sélectionne le meilleur choix à chaque étape, on risque de manquer la solution optimale.

Deux caractéristiques importantes qui rendent la méthode gloutonne si populaire sont la simplicité de sa mise en œuvre et son efficacité. Il peut produire, pour certains problèmes d'optimisation tels que le problème de l'arbre couvrant de poids minimal [134][153], une solution optimale. Au cours des dernières années, plusieurs types d'algorithmes gloutons améliorés sont proposés, Witold [47] présente un travail synthétique des problèmes où la résolution est basée sur des algorithmes glouton. Cette méthode est à l'origine de plusieurs métaheuristiques telles que le recuit simulé et les algorithmes génétiques.

Recherche tabou

La recherche tabou ressemble à la méthode de recuit simulé, la principale différence entre les deux méthodes réside dans le mécanisme utilisé pour approuver une solution d'ordonnancement intermédiaire. Contrairement à la méthode du recuit simulé, qui se base sur un mécanisme probabiliste, la méthode de recherche tabou se base sur une méthode déterministe. Cette méthode repose sur l'utilisation d'une liste nommée liste taboue (LT) et sur l'acceptation d'une solution voisine de la solution courante si elle ne fait pas partie des éléments de cette liste.
L'algorithme commence par une solution initiale générée soit aléatoirement soit par une heuristique. À chaque itération, la solution courante est remplacée par une solution de voisinage qui ne fait pas partie de la liste tabou, la nouvelle solution doit améliorer la solution existante. Face aux limites mémoire, il n'est pas possible d'enregistrer toutes les solutions visitées. En conséquence, la liste tabou contient les derniers éléments visités dans les $\mid LT \mid$ itérations et seuls les cycles d'une longueur supérieure à la longueur de la liste tabou peuvent se produire. Dans la liste des solutions de voisinage, un cycle tabou désigne que les solutions avoisinantes appartiennent à la liste tabou. La recherche tabou est une méthode de recherche locale combinée avec un ensemble de techniques permettant d'éviter les minimums locaux ou la répétition du cycle.
Pour plus d'information concernant cette méthode, vous pouvez consulter le travail de Fred et al. [66].

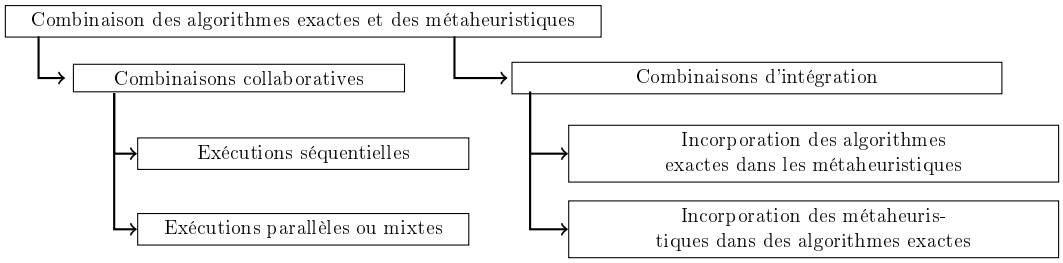


FIGURE 2.1 – Classification majeure de la combinaison des algrithmes exactes et des métaheuristiques

2.4.3 Approche combinant les méthodes exactes et les métaheuristiques

Plusieurs travaux de recherche combinent les deux méthodes exactes et métaheuristiques. Un recensement de ces travaux est synthétisé dans le travail de Puchinger et al. [126]. Ils présentent une classification des approches qui cherchent à combiner les métaheuristiques et les algorithmes exactes. La figure 2.1 présente la hiérarchie des différentes classes utilisées. Les deux classes du premier niveau de la hiérarchie sont :

— Les combinaisons collaboratives : la collaboration traduit le fait que les deux algorithmes sont capables d'échanger des informations et sont indépendants. Les algorithmes exactes et heuristiques peuvent être exécutés de façon séquentielle, entrecroiser ou en parallèle.

— Les combinaisons d'intégration : l'intégration traduit le fait qu'un algorithme est un composant intégré et subordonné d'un autre algorithme. Ainsi, il existe un algorithme maître, qui peut être soit un algorithme exacte soit un algorithme métaheuristique, et au moins un algorithme esclave.

Une bonne combinaison des algorithmes exactes et des métaheuristiques peut augmenter significativement les performances en matière de qualité de la solution et le temps de son obtention. Par exemple, Möhring et al. [112] utilisent un programme linéaire relâché avec relaxation lagrangienne pour préparer les données d'entrée de leur heuristique de liste qui cherche la solution finale. Certaines techniques de combinaison existantes sont matures, tandis que d'autres sont encore à leurs débuts et ont besoin d'être améliorées.

2.5 Exemple d'algorithmes d'ordonnancement dans les grilles de calcul

Cette section focalise sur les algorithmes existant pour l'ordonnancement des tâches dans les grilles de calcul et dont sont inspirés les algorithmes implémentés dans Hadoop. On commence par présenter les défis les plus rencontrés lors de l'ordonnancement des tâches sur des plateformes informatiques. On aborde, dans cette section, la gestion de charge, le partage de ressources et la gestion de la consommation énergétique.
L'enjeu de cette partie est de montrer l'importance des heuristiques dans l'ordonnancement des

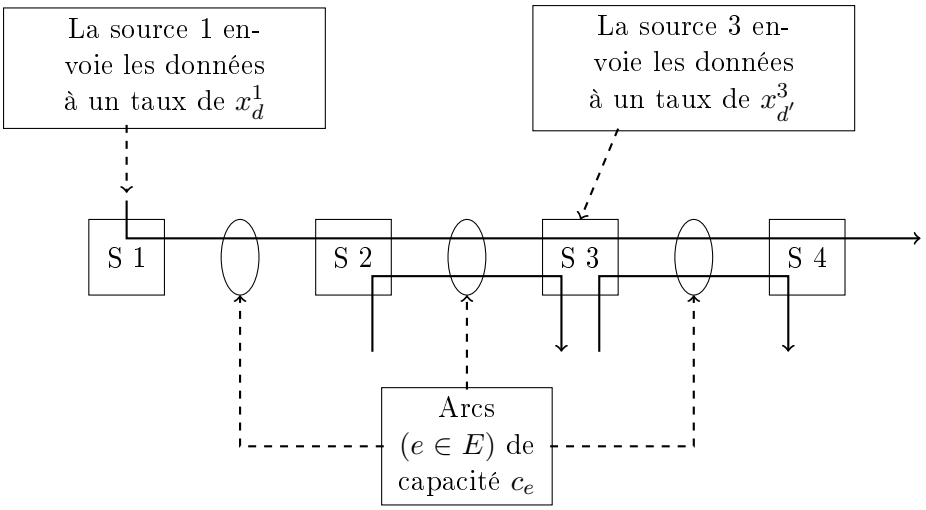


FIGURE 2.2 – Réseau utilisé pour l'illustration du partage équitable

applications qui nécessite des temps de réponse réduits.

2.5.1 L'algorithme de partage équitable Max-Min

L'algorithme "Max-Min Fairness" [18] est largement utilisé pour la résolution des problèmes d'allocation de ressources. Nous présentons dans cette partie les principes de cet algorithme à travers des cas d'utilisation.
On considère l'exemple suivant, soit un ensemble de nœuds sources $s = 1, \ldots, V$, E arcs de capacité $c_e, \quad e = 1, \ldots, E.$ et un ensemble de D demandes de connexions. Chaque connexion $d, (d = 1, 2, \ldots, D)$ est associée à un chemin prédéfini P_d entre les sources de ces deux extrémités. On définit le vecteur $\vec{x}$ où la i^{eme} coordonnée est l'allocation de l'utilisateur i et soit χ l'ensemble de toutes les solutions faisables. Soit $\delta_{e,d}$ un coefficient binaire tel que :

$$\delta_{e,d} = \begin{cases} 1 & \text{si } e \in P_d, \\ 0 & \text{sinon.} \end{cases}$$

Soit x_d la bande passante associée au chemin P_d et le vecteur d'allocation correspondant $\vec{x} = (x_1, x_2, \ldots, x_D) \in \Re^D$. On cherche à trouver un vecteur d'allocation $\vec{x} \in \chi$ équitable et faisable. Une allocation faisable des taux de $\vec{x}$ est dit "max-min fair" si et seulement si une augmentation de n'importe quel taux dans le domaine des allocations faisables doit forcément engendrer la diminution de certains parmi ceux qui possèdent les plus petits taux. Formellement, pour une autre allocation faisable $\vec{y}$, si $y_s > x_s$, il doit alors exister un certain $s^{'}$ tel que $x_{s'} \leq x_s$ et $y_{s'} \leq x_{s'}$.
Un arc $e \in P_d$ est saturé par rapport au chemin P_d si et seulement si l'arc l est saturé : $c_e = \sum_{d=1}^{D} \delta_{e,d} x_d^0$ tel que x_d^0 est au moins aussi large que la bande passante allouée à n'importe quelle connexion utilisant l'arc e.

Cette définition permet d'introduire le théorème suivant dont la preuve est dans [18] :
Une allocation faisable des taux de $\vec{x}$ est "max-min fair", si et seulement si, chaque source subit un goulot d'étranglement au moins à travers l'un des arcs qu'elle utilise.
On peut classer l'allocation de la bande passante dans l'exemple précédent comme une allocation "max-min fair". Le nom "max-min" vient de l'idée qu'il est interdit de diminuer la part des sources possédant de petites portions de la bande. Ainsi, nous accordons indirectement la priorité aux flux avec de petites portions de la bande.

L'algorithme nommé "Progressive Filling" [18] induit à l'obtention d'une allocation "max-min fair". Son principe consiste à commencer par des taux égaux à 0 et à les augmenter ensemble au même rythme jusqu'à ce que la capacité d'un ou plusieurs arcs soit atteinte. On n'augmente plus les taux associés aux sources qui utilisent ces arcs, cependant, on continue à augmenter les taux associés aux autres sources. En conséquence, toutes les sources arrêtées ont au moins un arc qui subit un goulot d'étranglement. L'algorithme continue à s'exécuter jusqu'à ce qu'on ne puisse plus augmenter les taux d'aucune source.
L'algorithme se termine lorsque, à un instant donné, toutes les sources ont arrêté l'envoie de données sur le réseau suite à un goulot d'étranglement sur au moins un des arcs qu'elles utilisent.
Une alternative de l'algorithme Max-Min est le "weighted max-min fairness" [9][107] : il associe aux candidats des ordres de priorités et partage les ressources en tenant compte de ces priorités.

L'algorithme de partage équitable Max-Min est appliqué à plusieurs domaines pour assurer le partage équitable des ressources sur plusieurs candidats, par exemple, dans la théorie des jeux, Schmeidler [92] est le premier à adapter l'algorithme Max-Min pour assurer le partage des gains du jeu. Récemment, des travaux [90][93] exploitent l'algorithme Max-Min pour atteindre les mêmes objectifs. Supposons qu'on a $N = 1, \ldots, n$ joueurs et une fonction objectif $v : 2^n \longrightarrow \mathbb{R}_+$ qui associe une valeur $v(S) \geq 0$ à chaque $S \subseteq N$. Le problème abordé par Schmeidler consiste à trouver un partage équitable du gain total $v(N)$ sur tous les utilisateurs ($i \in N$).
L'algorithme Min-Max fairness est étudié pour assurer l'amélioration des taux d'allocations et la gestion de congestion des réseaux TCP/IP, le routage et la gestion de charge réseau. Concernant les travaux traitant le contrôle de la congestion réseau, on peut citer [24], [31], [89], etc. Concernant la gestion des flux des applications, on peut citer [27][55][64] et [104].

2.5.2 Les algorithmes de liste

Les algorithmes de liste sont des l'algorithmes Glouton : l'idée est de ne pas laisser une ressource inoccupée tant qu'elle peut être utilisée pour exécuter une tâche disponible. En fonction des objectifs finaux de l'algorithme, la quantité des ressources disponibles et la priorité des tâches, l'algorithme cherche un ordonnancement partiel et tente de l'améliorer d'une itération à une autre jusqu'à satisfaire les conditions d'arrêt.
Après la définition des priorités des tâches, un algorithme de liste se base sur l'exécution itérative des étapes suivantes :

1. Placer les tâches dans les listes en fonction de leurs priorités. La définition de l'ordre et des priorités des tâches par liste doit satisfaire les contraintes du problème à résoudre.

2. Choisir une unité de calcul (machine) pour l'exécution de chaque tâche.
3. S'il existe des priorités entre les listes, affecter la tâche en tête de la liste la plus prioritaire à l'unité de calcul choisie à l'étape précédente (étape 2).

Bien que les algorithmes de liste soient des algorithmes d'approximation et fournissent des solutions approchées, ils sont caractérisés par la simplicité de leur mise en œuvre et le temps de calcul réduit pour trouver une solution.
L'architecture distribuée des grappes informatiques et le rôle important des données dans les systèmes du type "Big data" sont deux atouts qui influencent le temps d'exécution des travaux. Les tâches "Map" et "Reduce" doivent communiquer pour échanger des données à traiter. La section suivante présente des techniques utilisées dans la littérature pour limiter l'influence de l'échange des données sur le temps d'exécution des travaux.

2.6 Conclusion

À travers ce chapitre, nous avons présenté les problèmes d'optimisation et les étapes de leur résolution. Ensuite, nous avons abordé les problèmes d'ordonnancement, sur des plateformes informatiques, comme un sous-ensemble des problèmes d'optimisation. Nous avons introduit quelques méthodes de résolution.
Dans le chapitre suivant, nous présentons l'état de l'art sur l'ordonnancement des travaux du type MapReduce.

Chapitre 3

État de l'art sur l'ordonnancement des travaux de types MapReduce

3.1 Introduction

Dans le cadre du logiciel Hadoop, le premier algorithme d'ordonnancement implémenté est l'algorithme FIFO (*Fisrt in first out*), il est illustré dans la figure 3.1. Il est le seul ordonnanceur non adaptatif développé pour Hadoop, son comportement est indépendant des changements de l'environnement où il s'exécute. Les travaux sont exécutés selon leur ordre de soumission. Après la division d'un travail en un ensemble de tâches Map et Reduce, ces dernières sont chargées dans des listes et sont affectées aux ressources libres sur les machines de la grappe. Le chapitre 1 présente le modèle MapReduce, le besoin des tâches associées et les caractéristiques d'une grappe Hadoop.

En général, les algorithmes d'ordonnancement implémentés dans Hadoop sont adaptatifs (ou dynamique) : ils changent leurs politiques d'ordonnancement en fonction des valeurs des para-

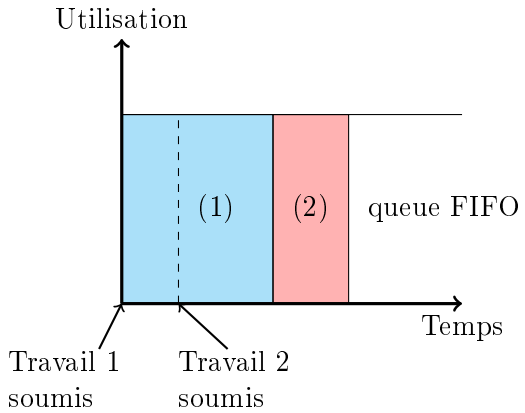


FIGURE 3.1 – Modélisation de l'exécution de deux tâches en utilisant la politique d'ordonnancement FIFO

mètres d'influences à travers le temps. Ces paramètres peuvent par exemple être le nombre de machines, le nombre des utilisateurs connectés à la grappe (etc.). La plupart des algorithmes d'ordonnancement présentés dans la littérature sont évalués de manière expérimentale, soit par rapport aux ordonnanceurs existants dans des implémentations de référence, soit en utilisant des simulateurs.
Dans la plupart des cas, les études théoriques du problème d'ordonnancement des travaux du type MapReduce dans Hadoop abordent le problème comme des variantes des problèmes d'ordonnancement classiques. Par exemple :

— Le modèle Flow Shop : Moseley et al. [113] généralisent une variante du modèle d'ordonnancement FlowShop nommée «FFS» (*Flexible Flow Shop*) à 2 étapes. Cette variante est NP difficile. Ils considèrent deux critères de façon indépendante : la minimisation du flux total "total flowtime" et la minimisation du C_{max}.
Lorsque Moseley et al. traitent de l'ordonnancement sur des machines identiques, ils considèrent que : *(i)* un travail MapReduce est composé de plusieurs tâches Map et plusieurs tâches Reduce. *(ii)* Chaque tâche associée à chaque type (Map ou Reduce) s'exécute sur une des machines dédiées. *(iii)* une tâche peut être interruptible mais non migratoire, c'est-à-dire, ils ne considèrent pas qu'il est intéressant de migrer une tâche avec des données partiellement traitées. *(iv)* Chaque travail est associé à une date de soumission a_J (*release date*).
Moseley et al. [113] proposent un algorithme pour le mode hors ligne et un autre pour le mode en ligne. L'algorithme hors ligne a un taux d'approximation de 12 et l'algorithme en ligne est de compétitivité $\Theta(\frac{1}{\varepsilon^2})$ avec $\varepsilon \in]0,1]$ et $(1+\varepsilon)$ la vitesse d'une machine.
Pour le mode hors ligne, l'algorithme qu'ils proposent commence par le calcul de deux ordonnancements séparés (σ_m et σ_r) qui sont utilisés pour définir les règles de priorité entre les tâches. Ensuite, il calcule un intervalle de disponibilité pour l'ordonnancement de chaque tâche. Enfin, il construit l'ordonnancement final (σ).
Les deux ordonnancements faisables sont : σ_m pour l'ordonnancement des tâches Map sur une seule machine qui possède une vitesse N_m et σ_r pour l'ordonnancement des tâches Reduce sur une seule machine qui possède une vitesse N_r. Ces deux ordonnancements sont calculés à l'aide de l'algorithme «SRPT» qui consiste à ordonnancer en premier les tâches possédant le temps restant le plus court. l'utilisation de «SRPT» respecte les 4 propriétés décrites précédemment dans ce paragraphes. Elle considère pour chaque tâche (et travail) une date de soumission et une date de fin limite (ou de disponibilité) qu'ils définissent. «SRPT» fournit un ordonnancement optimal lors de la minimisation du flux total "total workflow" sur une seule machine quand on considère : *(i)* une tâche par travail, *(ii)* que les tâches sont préemptibles, *(iii)* qu'il n'y a pas de relation de précédence entre les tâches et lorsque *(iv)* chaque tâche a une date de soumission.
Le calcul de l'intervalle de disponibilité commence par le calcul du poids de chaque travail w_j, ensuite, à considérer que la date de disponibilité des tâches Map associées à ce travail est égale à 0 et la date de disponibilité des tâches Reduce est égale à $2w_j$. La date limite pour l'ordonnancement d'un travail j doit être inférieure à $4*X$ avec X la valeur minimale

parmi *(i)* la date de fin de la dernière tâche Map du travail j et *(ii)* la date de fin de la dernière tâche Reduce du travail j et *(iii)* la durée la plus longue d'une tâche du travail j. Pendant l'étape suivante, les travaux sont triés en fonction de leur date de fin d'exécution. La date de fin d'exécution d'un travail est la date de fin associée à la dernière tâche exécutée du travail (dans σ_m et σ_r). Pour chaque travail de la liste triée, on commence par affecter chaque tâche Map à la machine Map la moins chargée. Ensuite, on affecte chaque tâche Reduce à la première machine Reduce disponible.
Pour le mode en ligne, où la date de soumission des travaux est inconnue à l'avance, Moseley et al. proposent un algorithme qui commence par une version évoluée de «SRPT» adaptée au contexte en ligne et l'utilise pour le calcul des deux ordonnancements en ligne σ_m et σ_r. Ces derniers sont mis-à-jour en fonction des travaux récemment soumis. Ensuite, il calcul le poids et la classe (notion inexistante dans la version hors ligne) de chaque travail. Une nouvelle formule est utilisée pour calculer le poids de chaque travail w_j. Les différents travaux sont groupés en fonction de leurs poids en classe, par exemple, un travail j appartient à la classe k si $w_j \in [2^k, 2^{k+1})$. L'algorithme sauvegarde la date de fin de la dernière tâche (Map m et Reduce r) associée à chaque classe sur chaque machine x ($U_{=k}^{m,x}$ et $U_{=k}^{r,x}$). Ensuite, à chaque instant t, l'algorithme ordonnance les tâches du travail j si et seulement si toutes les tâches de j ont terminé leurs exécutions dans σ_m et σ_r avant t. L'algorithme choisit la machine qui possède, pour la même classe et type de tâche, la date de fin la moins élevé (la valeur $U_{=k}^{t,x}$, avec $t \in \{m, r\}$).
Les critiques que nous pouvons associer à ce travail sont l'ignorance de plusieurs facteurs du système réel qui influencent considérablement la performance du système tel que *(i)* la non-considération du critère de localité, *(ii)* l'absence de la prise en compte du réseau, *(iii)* ne pas considérer l'étape qui consiste à transférer les données traitées par les tâches Map vers les tâches Reduce (nommée "shuffle") qui commence en parallèle avec les tâches Map,*(iv)* considérer que les tâches sont préemptibles.

— Problèmes d'ordonnancement à machines non reliées : Fotakis et al. [57] reprennent le modèle théorique de Moseley et al. Ils abordent le problème comme celui d'un ordonnancement à machines non reliées et généralisent leur modèle [113] pour supporter plusieurs tâches Map et Reduce par travail. Ils supposent que les tâches Map (respectivement les tâches Reduce) s'exécutent sur des machines réservées pour des tâches Map (respectivement les tâches Reduce). Ils considèrent les contraintes suivantes dans la modélisation du problème : *(1)* les machines sont hétérogènes, *(2)* tous les travaux sont disponibles à la date 0, *(3)* les tâches Reduce ne seront ordonnancées qu'après la fin des tâches Map et *(4)* toutes les tâches sont non interruptibles. Ils proposent un algorithme polynomial d'approximation $(32 + \varepsilon)$. Ce dernier se base sur l'algorithme proposé dans [99] qui possède un taux d'approximation égal à $(8 + \varepsilon)$.

Lors de son apparition, le logiciel Hadoop est basé sur les hypothèses [152] suivantes : *(i)* les machines sont équivalentes et possèdent la même puissance de calcul. En conséquence, *(ii)* les tâches progressent avec un ratio (quantité de données traitée par seconde) constant à travers le temps. *(iii)* les tâches ont tendance à s'exécuter en vagues, *(iv)* les tâches de la même catégorie

(Map ou Reduce) ont la même durée.
En fonction de l'environnement et du type de la grappe, la plupart de ces hypothèses ne sont pas valables. Le non-respect de ces hypothèses pénalise Hadoop et ses performances car les ordonnanceurs implémentés fournissent alors des mauvaises solutions. Plusieurs travaux récents ont pour objectifs de répondre aux besoins des utilisateurs et d'adapter les politiques d'ordonnancement aux plateformes hétérogènes. La plupart des travaux sur des grappes hétérogènes se basent sur la méthode nommée exécution spéculative.

Par définition, l'exécution spéculative [152] [171] consiste à détecter si une tâche est en retard et à relancer une copie de cette tâche sur une autre machine. Les résultats de l'exécution de la tâche la plus rapide sont récupérés. Google a montré que l'exécution spéculative améliore la date de fin des travaux de 44% [43]. L'exécution spéculative est adaptée au contexte des grappes à machines homogènes, elle se base sur les deux hypothèses suivantes : *(i)* la progression uniforme des tâches sur les machines, *(ii)* toutes les machines ont la même vitesse de traitement. Plusieurs travaux [171] [161] sont réalisés pour améliorer l'algorithme d'exécution spéculative sur des grappes hétérogènes.
Quand une machine contient des Vcores libres, Hadoop choisit une tâche appartenant à l'une des trois catégories suivantes :

1. La liste des tâches qui ont échoué (qui n'ont pas terminé leur exécution suite à une panne système). Hadoop affecte aux tâches qui ont échoué la priorité la plus élevée. Cette politique permet de détecter les tâches qui échouent de façon continue et de mieux gérer le travail associé.
2. La liste des tâches qui ne sont pas exécutées.
3. La liste des tâches qui doivent s'exécuter en mode spéculatif.

Hadoop détecte les tâches en retard et qui doivent s'exécuter en mode spéculatif en gérant un paramètre «SP» (*Score de progression*) par tâche, ($(SP) \in [0 \dots 1]$). Pour les tâches Map, «SP» dépend de la phase de lecture de données et de la phase de traitement. Pour les tâches Reduce, «SP» est divisé en trois phases de référence : *(i)* la phase de copie de données, *(ii)* la phase de trie et *(iii)* la phase de traitement. Chacune de ces phases présente le $\frac{1}{3}$ de «SP» . Par exemple le «SP» d'une tâche Reduce dans la moitié de sa phase de traitement est de $\frac{1}{3} + \frac{1}{3} + (\frac{1}{2} \times \frac{1}{3})$.
Hadoop calcul un «SES» (*Seuil moyen pour l'exécution spéculative*) par travail, un «SES» est la moyenne des «SP» associés aux tâches de chaque type de tâches (Map et Reduce). Quand les deux conditions suivantes sont vérifiées, une tâche est supposée être en retard et doit s'exécuter en mode spéculatif :

1. Le score de progression «SP» $>$ «SES» - 0.2.
2. La tâche s'exécute depuis au moins une minute

Par ailleurs, toutes les tâches qui possèdent un «SP» inférieure au «SES» sont considérées équivalentes et elles sont en retard. Donc, elles doivent être lancées en mode spéculatif. L'ordonnanceur dans Hadoop s'assure qu'une seule copie de chacune de ces tâches s'exécute en mode spéculatif. La méthode de définition du «SES» fournit de bons résultats sur les grappes homogènes puisque

les tâches ont tendance à s'exécuter en vagues et les tâches en spéculation commencent généralement avec la vague suivante. Ce qui assure que les résultats de ces tâches seront disponibles à la fin de la vague suivante
Dans ce chapitre, nous présentons des algorithmes existants dans la littérature en fonction des critères qu'ils cherchent à optimiser. Nous les classons en cinq groupes. La liste des algorithmes étudiés est synthétisée dans les tableaux 3.1 et 3.2 à la fin de ce chapitre.

3.2 Algorithmes de partage équitable de ressources

Avec l'évolution du logiciel Hadoop et la possibilité d'ajouter des priorités entre les travaux et entre les files d'attente (ou liste), plusieurs algorithmes d'ordonnancement sont présentés par la communauté. L'algorithme FIFO a évolué pour supporter plusieurs niveaux de priorités. Lorsqu'un Vcores est libre, l'ordonnanceur FIFO cherche parmi les travaux celui de plus haute priorité avec la date la plus ancienne de soumission. Ensuite, il ordonnance, pour ce travail, les tâches Map les plus proches des données. Cet ordonnanceur ne gère pas l'équilibre de partage des ressources ni entre travaux de petites et grandes tailles, ni entre plusieurs utilisateurs. Il existe plusieurs politiques pour partager équitablement les ressources. On peut rappeler par exemple :

— Partager équitablement des ressources sur un ensemble de n utilisateurs (ou travaux ou tâches). Chaque utilisateur aura $\frac{1}{n}$ de la ressource à partager.

— La politique Max-Min pour le partage d'une seule ressource : Elle consiste à affecter à chaque utilisateur la quantité minimale configurée et nécessaire à son exécution en fonction d'une ressource. Les ressources restantes sont partagées entre les utilisateurs avec le même ratio qu'ils ont obtenu de la ressource principale.

— La politique Max-Min basée sur un paramètre de poids : elle consiste à affecter à chaque utilisateur un poids selon son importance et à partager les ressources entre ces utilisateurs en tenant compte de leurs poids. Cette technique a été utilisée dans les deux ordonnanceurs implémentés actuellement dans Hadoop (FairScheduler et capacity Scheduler).

Nous présentons dans cette section des algorithmes utilisés dans l'implémentation Hadoop.

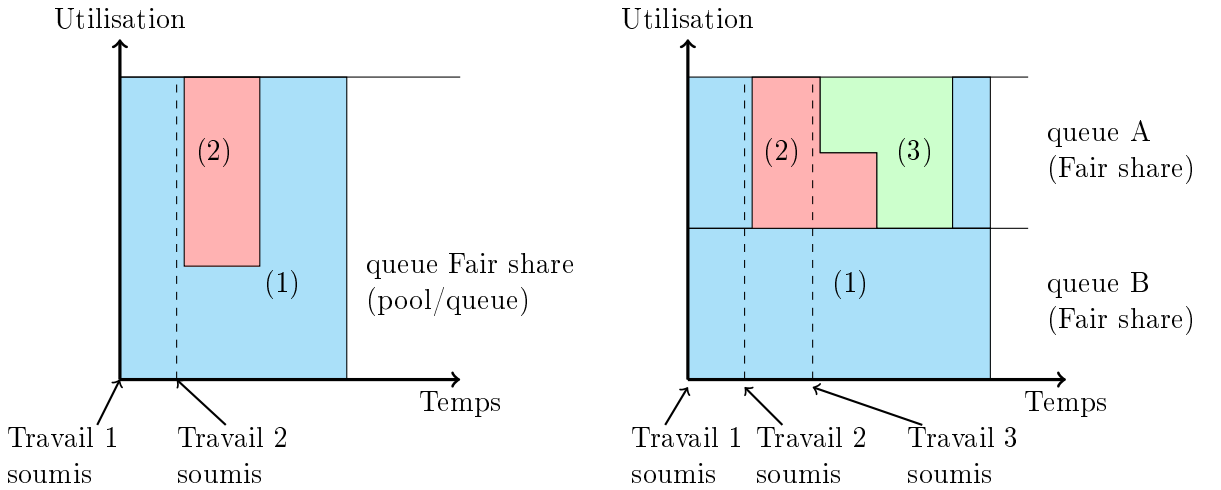
3.2.1 Algorithme «FS» (*FairScheduler*)

Cet ordonnanceur est proposé par FaceBook [169] pour fournir les services d'Hadoop à la demande. Il est une évolution de l'algorithme « max-min fairness ». Il a deux objectifs : il partage la grappe entre les différents utilisateurs (isolation entre utilisateurs) et il veille à ce que les ressources soient bien exploitées (redistribution des ressources non utilisées entre les utilisateurs). «FS» est basé sur la gestion hiérarchique des listes et sur la gestion de priorité entre elles. Il fournit à chaque utilisateur une liste et partage les ressources entre les différentes listes. Il contrôle le nombre de travaux affectés à chaque liste afin de limiter la concurrence sur les ressources. Au niveau des listes, du plus bas niveau de la hiérarchie, l'algorithme «FS» permet l'utilisation des politiques FIFO, «FS» ou «DRF» (section 3.2.4 [63]). Pour l'ordonnancement des tâches, «FS» assure une quantité minimale de ressource par liste. Si une liste n'a pas le taux minimal de

ressources, l'ordonnanceur tue les tâches les plus récentes et réaffecte les ressources récupérées à cette liste. Cette politique assure le non-blocage entre les tâches et le minimum de pertes lors de la récupération des ressources. Dans le cas où le «FS» est configuré pour prendre en considération les priorités entre les travaux, celles-ci seront utilisées pour associer un poids par travail et les ressources seront distribuées en tenant compte de ces poids. «FS» fournit aux administrateurs la possibilité d'affecter un travail à une liste et de fixer un taux minimal de ressources par liste. Seuls les Vcores sont les ressources partagées entre les listes [20]. Pour récupérer des ressources, «FS» combine deux techniques qui se résument soit à tuer la tâche, soit à attendre sa fin. Au cas où une liste de tâches s'exécute avec un nombre de Vcores inférieur au taux minimal nécessaire, l'ordonnanceur tue des tâches appartenant à des listes s'exécutant en surcapacité pour affecter ces ressources aux listes s'exécutant en sous capacité.
Lorsqu'une tâche est interrompue, elle est relancé comme si elle s'exécutait pour la première fois. Les administrateurs ont la possibilité d'influencer le comportement de l'ordonnanceur : ils peuvent augmenter la priorité des listes. En conséquence, les tâches seront ordonnancées d'une façon entrelacée sur la base de la priorité de la liste à laquelle ils appartiennent, la capacité de la grappe et l'utilisation des ressources associées à cette liste.

Exemple d'exécution des travaux en utilisant l'algorithme «FS».

En utilisant «FS», les ressources sont gérées de façon dynamique entre les applications et on n'a pas besoin de faire la réservation a priori. La figure 3.2a montre comment «FS» fonctionne avec des travaux appartenant à la même liste, suite au démarrage de l'exécution d'un premier et grand travail (numéro 1), il occupe toutes les ressources réservées de la grappe. Quand le deuxième travail, de petite taille, est soumis, il démarre et récupère la moitié des ressources (en utilisant la politique de partage équitable). Il y a un décalage entre la date de démarrage du travail (2) et la date où il reçoit sa part des ressources, il doit attendre la libération des ressources utilisées par



(a) Exécution de deux travaux de la même liste (b) Exécution de trois travaux appartenant à deux listes.

FIGURE 3.2 – Exécution des travaux avec l'ordonnanceur «FS».

le premier travail. Après la fin du travail (2), le travail (1) récupère ses ressources pour exécuter les tâches restantes.
La figure 3.2b montre comment «FS» fonctionne avec des travaux appartenant à des listes différentes, chacune de ces listes est associée à un utilisateur. On a deux utilisateurs A et B. A la soumission du travail (1) par l'utilisateur B, il alloue toutes les ressources de la grappe. Lorsque le travail (2) est soumis par l'utilisateur A, il récupère la moitié des ressources disponibles au fur et à mesure que le travail (1) les libère, la politique de partage est la même décrite précédemment. Lorsque l'utilisateur A soumet le travail (3), il partage les ressources réservées à l'utilisateur A avec le travail (2) (soumis par le même utilisateur). En conséquence chacun des travaux (2) et (3) récupère le $\frac{1}{4}$ des ressources de la grappe et le travail (1) continue à s'exécuter en utilisant la moitié des ressources disponibles. De ce fait, les ressources sont partagées équitablement entre les deux utilisateurs. Après la fin des travaux de l'utilisateur A, le travail (1) récupère les ressources libérées et termine son exécution.
«FS» ne donne pas d'importance à l'état de la grappe, il ne prend en compte ni l'équilibrage des charges entre les nœuds esclaves ni l'optimisation de la consommation énergétique. Son objectif est le partage équitable des ressources entre les utilisateurs, entre les travaux du même utilisateur et entre les tâches associées aux travaux.

3.2.2 Algorithme «CS» (*Capacity Scheduler*)

Le Capacity Scheduler est un ordonnanceur proposé par Yahoo [58]. Il permet une meilleure utilisation de la séparation logique des ressources en multiclient (multitenancy). Sa conception est très proche de «FS». Il est basé sur une hiérarchie entre les listes. Il permet la gestion d'une hiérarchie de liste plus profonde et il permet, en conséquence, une partition plus fine des ressources de la grappe. L'administrateur peut définir le taux maximal de ressources à affecter pour chaque liste de la hiérarchie. Si une liste n'a pas la quantité demandée de ressources, l'ordonnanceur récupère des ressources de façon analogue aux politiques utilisées dans «FS». «CS» réserve

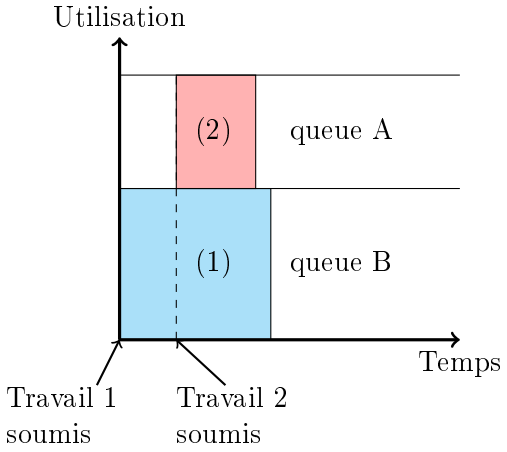


FIGURE 3.3 – Exécution des travaux avec l'ordonnanceur «CS»

à chaque liste sa quantité de ressources minimale demandée. Cette méthode a de l'influence sur le coût d'exploitation de la totalité de la grappe puisque la quantité de ressources associée à une liste est réservée seulement à l'ensemble de ses travaux. En conséquence, les travaux de grande taille nécessitent plus de temps pour finir qu'avec «FS» et FIFO. Comme exprimé à travers la figure 3.3, un travail ne peut pas utiliser plus que les ressources réservées à la liste à laquelle il est associé. Afin de mieux exploiter la grappe, «CS» permet de définir les quantités maximales de ressources dont une liste peut utiliser lorsqu'il y a des ressources libres sur la grappe. Cette règle permet à une liste de dépasser la quantité de ressources fixée a priori pour utiliser un supplément que l'administrateur doit définir. Cette règle améliore la rentabilité de la grappe puisqu'elle permet aux grands travaux d'exploiter les ressources libres (dans la limite du dépassement autorisé de ressources) de la grappe et réduire leurs temps d'exécution. Elle fournit un moyen pour éviter la famine des petits travaux en laissant aux listes vides (les listes qui ne contiennent pas des travaux en cours d'exécution) une quantité de ressources minimale leur permettant de lancer les travaux dès leurs soumissions.

3.2.3 Algorithme MARLA

Marla [52] [71] est une implémentation du modèle MapReduce différente d'Hadoop. Son idée principale est de permettre aux machines esclaves d'effectuer des demandes de tâches à leur rythme, c'est-à-dire, ne demander une tâche qu'après avoir terminé les anciennes avec succès. En conséquence, sur des grappes homogènes, la charge est divisée équitablement sur les machines. Alors que sur les grappes hétérogènes, les machines les plus performantes exécutent naturellement plus de charge.
Marla est constituée de trois éléments principaux :

— Le Splitter : au lieu de HDFS sur Hadoop, MARLA se base sur un système de fichier partagé tel que GPFS et NFS comme mécanisme de gestion des I/O. Il n'y a pas de redondance de données. Marla, différemment d'Hadoop, ne génère pas un nombre de portions presque équivalentes de tranches de données. Elle considère que les portions de données d'entrée sont des tâches et elles sont créées et distribuées en fonction des utilisateurs et de la configuration des machines. Elles sont divisées en portions très fines et les machines possédant la meilleure configuration exécuteront plus de tâches que les autres.
Il n'y a pas de relation entre les données d'entrée et la quantité de tâches à créer. On ajuste la taille des tranches et on crée plus de tâches de calcul (Map et Reduce). La machine la plus puissante exécute plus de tâches que les autres. Marla exploite la visibilité des données offertes par le système de fichier partagés pour ordonnancer les tâches selon la demande. De cette manière, la gestion des entrées et la distribution des portions de données ne sont pas gérées de l'extérieur du système de fichiers mais dépendent de son mécanisme par défaut.
La distribution des données est gérée par le système de fichiers. Quand l'utilisateur dépose des données, le système de fichiers partagé est optimisé pour mettre en cache et rendre les données visibles à toutes les machines.
En résumé MARLA profite des avantages apportés par les systèmes de fichiers partagés

(optimisations lors des transferts de données).

— Le TaskController : cet élément maître est responsable de la mise à disposition des tranches de données aux unités de calcul. Il profite du système de fichier partagé pour affecter les tâches aux machines et gérer la progression de leur exécution. L'affectation des tâches de calculs dépend de l'affectation des portions de données.

— Le FaultTracker : c'est un système de tolérance aux pannes qui consiste à relancer les tâches échouées. Contrairement à Hadoop qui utilise la redondance des portions de données pour la tolérance aux pannes. MARLA utilise une autre politique : si une tâche i échoue sur une machine j, elle est ajoutée à une liste spécifique et aucune nouvelle tâche n'est affectée à "j" jusqu'à ce que la tâche i soit affecté à une autre machine j'. Si la tache i réussit son exécution sur la nouvelle machine j', la machine j reçoit un avertissement d'erreur. Les machines qui reçoivent trois avertissements d'erreurs sont considérées erronées et restent au repos pendant une durée de pénalité.

Lorsque les machines dans Hadoop sont en panne, le système lance la réplication des données, la plateforme est en conséquence en surcharge. Cette opération, dans certains cas, engendre l'arrêt des applications puisque les données n'existent plus sur la grappe. Le principe de MARLA est que les applications sont servies tant qu'il y a des machines qui tournent sur la grappe. L'évaluation des idées implémentées dans MARLA a montré des résultats positifs et des performances meilleures que celles fournies par Hadoop avec «CS» et «FS».

3.2.4 Algorithme «DRF» (*Dominant Resource Fairness*)

L'ordonnanceur «DRF» [63] adresse le problème du partage équitable de ressources. Il est destiné à la gestion de plusieurs ressources. Il est une généralisation de l'algorithme de partage équitable max-min. Les ordonnanceurs basés sur l'algorithme max-min tel que «FS» ou «CS» gèrent et partagent une seule ressource à la fois. Dans le cas de plusieurs types de ressources, la procédure de partage devient plus complexe. Par exemple, si une application utilisateur A demande une grande quantité de ressources CPU et une petite quantité de mémoire. Si une autre application B demande une grande quantité de mémoire et une petite quantité de CPU. La question qui se pose est comment l'ordonnanceur gère l'affectation des ressources entre ces applications ?
«DRF» calcule pour chaque utilisateur sa part de chaque ressource et la ressource correspondante à la part la plus grande est nommée ressource dominante. À chaque étape «DRF» récupère l'utilisateur possédant la plus petite part dominante parmi celles liées à des tâches à exécuter pour cet utilisateur. Si la quantité de ressources pour une tâche, appartenant à cet utilisateur, est disponible (assez de ressources sur la grappe), la tâche peut être lancée sur ces ressources. Sinon on vérifie avec la tâche suivante. Si un utilisateur termine l'exécution de toutes ses tâches où s'il n'existe plus de ressource pour exécuter une de ses tâches, l'ordonnanceur passe à l'utilisateur suivant possédant la plus petite part dominante et exécute ses tâches. L'opération d'ordonnancement s'arrête dès qu'une ressource est saturée.
Algorithme de partage équitable de la ressource dominante pondérée. Dans la pratique, il existe plusieurs situations où le partage de ressources de façon égale entre les utilisateurs

ne répond pas aux besoins des administrateurs. Tel est le cas lorsqu'on veut augmenter la part des utilisateurs qui payent plus pour avoir plus de ressources.
En utilisant le «DRF» pondéré, on associe à chaque utilisateur i un vecteur de poids $W_i = \langle w_{i,1}, \cdot, w_{i,m} \rangle$ où $w_{i,j}$ présente le poids de l'utilisateur i pour la ressource j. La définition de la part dominante change pour devenir $s_i = max_j\{\frac{u_{i,j}}{w_{i,j}}\}$ où $u_{i,j}$ est la part de la ressource j associée à l'utilisateur i. Quand les utilisateurs i et j ont le même poids et le ratio, entre les parts dominantes, est égal à $\frac{w_i}{w_j}$, le " DRF " dominant est réduit à l'algorithme " DRF ".
Après l'affectation des ressources, par RDF, aux différents utilisateurs, les politiques FIFO ou «FS»sont utilisées aux niveaux inférieurs de la hiérarchie des listes (niveaux où on veut affecter les ressources aux travaux ou tâches).

3.2.5 Algorithme «NATA» (*Network-Aware Task Assignement*)

Cet ordonnanceur [159] prend en considération la bande passante réseau pour ordonnancer efficacement les tâches. Il cherche à minimiser le temps d'exécution des travaux en tenant compte de l'affectation des tâches Map et Reduce entre les racks. Le temps nécessaire pour le transfert de données est considéré comme un critère lors de la modélisation.
Le problème est modélisé sous la forme d'un problème d'optimisation min-max 0-1 et résolu à l'aide de la programmation dynamique. NATA utilise deux heuristiques gloutonnes pour minimiser le temps de complétion par phase Map puis Reduce. La solution proposée gère l'affectation et l'ordonnancement des tâches sur une grappe contenant des baies distants géographiquement. NATA est évalué dans un simulateur. Les résultats présentés sont encourageants face aux ordonnanceurs tels que les ordonnanceurs Hadoop originaux et l'ordonnanceur LARTS [69] qui ordonnance les tâches Map et Reduce en tenant compte de la localité des données selon une méthode gloutonne.

3.2.6 Autres algorithmes

Naduri et al. [114] cherchent à réduire le temps d'exécution des travaux en affectant aux machines esclaves des tâches de configuration les plus proches des tâches précédentes. Il tient compte de la consommation des ressources (cpu, mémoire, disque et consommation internet) lors de la comparaison entre les tâches. Il se base sur la détection des évènements de la JVM (Java virtual machine) pour le recueil des métriques des tâches en cours et sur un modèle de prédiction pour les tâches futures.
Polo at al. [124] proposent «RAS» (*Resource-aware Adaptive Scheduler*), dont les deux objectifs sont la maximisation de l'exploitation des ressources sur une grappe Hadoop et la minimisation du temps d'exécution des travaux. Par défaut, le nombre de Vcores par machine est constant et Polo et al. proposent le recueil dynamique des métriques et la construction des profils des travaux pour ajuster dynamiquement le nombre de ressources (Vcores) par machine. C'est-à-dire adapter la puissance associée aux Vcores, ce qui permet de gérer leur vitesse pour satisfaire le besoin des tâches. L'avantage de cet ordonnanceur est la prise en compte de la bonne affectation des ressources et la minimisation de la durée d'exécution dans un environnement multi-utilisateurs.

Yao et al. [166] proposent un ordonnanceur nommé LsPS pour optimiser les performances de la grappe. Le principe est de suivre dynamiquement les performances de la grappe pour optimiser le partage de ressources entre les utilisateurs selon les caractéristiques des flux de travaux soumis et identifiés.
Yong et al [168] cherchent dans leur proposition (YGM) à optimiser l'utilisation des ressources (équilibrage de charge sur la grappe). Ils proposent deux algorithmes qui visent à affecter les tâches aux machines les moins chargées tout en tenant compte de bien stabiliser la charge des machines à un seuil relativement élevé et mettre au repos les machines sous-exploitées.
Nathanael et al. [30] propose «HSRTF» (*Shortest Remaining Time First Policy in Shared Hadoop Clusters*) qui est une alternative de l'algorithme d'ordonnancement "SRTF" (commencer par les travaux possédant les dates de fin les plus proches [70]), pour l'amélioration des sommes des dates de fin des travaux et du taux de partage des ressources entre les utilisateurs.
D'autres ordonnanceurs existent comme, par exemple, l'ordonnanceur "Choosy" [65] qui est une extension de "Max-Min Fairness".

3.3 Algorithmes pour l'amélioration des performances de la grappe

La plupart des travaux de cette section cherche à améliorer les dates de fin des travaux. De ce fait, l'amélioration des performances est associée à la réduction de la durée d'exécution des travaux.

3.3.1 Algorithme «DS» (*Delay Scheduling*)

Lors des expérimentations sur Hadoop, Zaharia et al. [170] ont trouvé que lorsque la grappe est saturée avec des travaux en exécution, il n'est pas possible de respecter le critère de localité des données. Les travaux de petite taille se trouvent face au problème de famine. Les tâches, associées aux travaux de petite taille, sont affectées sans tenir compte du critère de localité et, en conséquence, le pourcentage des tâches qui s'exécutent en local devient négligeable. Zaharia et al. ont remarqué durant des observations expérimentales que le fait de retarder l'exécution d'une tâche de quelques secondes, peut augmenter la chance d'obtenir des ressources sur une machine demandée.
Avec l'utilisation de «DS», l'ordonnanceur n'affecte pas les tâches à la première machine avec des ressources disponibles qui se présente. Il attend la durée nécessaire pour augmenter la chance de répondre au critère de localité. A ce niveau, la question qui se pose est la suivante : comment calculer la durée maximale du retard que l'ordonnanceur peut prendre en compte s'il ne peut pas considérer la localité lors de l'ordonnancement d'une tâche ? Zaharia et al définissent cette durée de retard D en fonction *(i)* du taux de réplication des blocs de données (qu'on note R) et *(ii)* du niveau (taux) de localité λ qu'on veut respecter pour un travail j constitué de N tâches de durée T chacune. On suppose qu'on a une grappe de M machines, chacune possède L Vcores, on a :

$$D \geq -\frac{M}{R} \ln\left(\frac{(1-\lambda)N}{1+(1-\lambda)N}\right)$$

En moyenne, un Vcores se libère toutes les $\frac{T}{LM}$ secondes. Pour le travail j, l'ordonnanceur doit attendre $\frac{DT}{LM}$ secondes avant de se permettre d'ordonnancer les tâches de j sans le respect du critère de localité.
Par exemple, pour $\lambda = 0.95$, $N = 20$ et $R = 3$, on obtient $D \geq 0.23M$. C'est à dire, le temps maximal que l'ordonnanceur doit attendre avant d'affecter les tâches sans le respect du critère de localité est $\frac{D}{LM}T$. Pour $L = 8$ et $M = 1$, le temps d'attente ($= 0.23T$) est de 2.8% de la moyenne des durées des tâches. De façon générale, cette méthode réduit le transfert des données à travers le réseau et augmente l'efficacité de la grappe.
«DS»est actuellement intégré, comme option non activée par défaut dans Hadoop, aux deux ordonnanceurs «FS» et «CS». Cet algorithme est une solution qui relâche la contrainte du partage équitable des ressources à la faveur de l'amélioration de la localité des unités de traitement et des données. Lors de l'évaluation expérimentale de «DS» intégré à «FS» avec des travaux de petites tailles (entre 1 et 25 tâches Map) il apparaît que «DS» diminue le temps de réponse de l'ensemble des travaux d'un facteur de 5.

3.3.2 Algorithme «LATE» (*Longest Approximate Time to End*)

Zaharia et al. [171] proposent l'algorithme «LATE» qui est une version modifiée de l'algorithme de l'exécution spéculative par défaut de Hadoop. «LATE» est un algorithme glouton, qui ordonnance les tâches en mode spéculatif tout en évitant les machines chargées. Il peut détecter les tâches en retard et lancer une autre copie en parallèle. Il fournit de bons résultats si les cœurs de calcul associés aux machines ont une vitesse constante et s'il n'y a pas de coût lors du lancement d'une tâche en mode spéculatif. «LATE» borne la quantité maximale des tâches qui peuvent s'exécuter en spéculatif pour éviter de surcharger la grappe. Il prend en considération la nature de la grappe lors de l'affectation des tâches en spéculatif. Il utilise le paramètre «SP» qui est fournit par Hadoop pour détecter les tâches à lancer en mode spéculatif. Le paramètre «SP» et le mode d'exécution sont définis dans la section 3.1. «LATE» commence par le calcul du temps d'exécution restant, $left(i)$, d'une tâche. $left(i) = \frac{1 - \text{SP(i)}}{\text{Taux de progression}}$ et le taux de progression=$\frac{\text{SP(i)}}{T}$, avec T la durée d'exécution de la tâche i depuis son début jusqu'à la date courante.
«LATE» a montré de bonnes améliorations par rapport à l'implémentation par défaut de l'exécution spéculative. Cet ordonnanceur a été testé sur une grappe hétérogène sur le cloud EC2 [135] d'Amazon. Il permet de diminuer le temps de réponse de l'ensemble des travaux de 27% avec l'exécution spéculative activée et 31% sans son activation. Plusieurs améliorations ont été apportées à l'algorithme «LATE» telles que les algorithmes Mantri (section 3.5.4) et BASE (section 3.3.3).

3.3.3 Algorithme «BASE» (*BASE Scheduler*)

Proposé par Zhenhua et al. [67], «BASE» est une évolution de l'algorithme «LATE». Il suppose au départ que les ressources (Vcores) sont affectées aux travaux en utilisant les ordonnanceurs par défaut d'Hadoop («FS», «CS»ou FIFO). En fonction du taux de progression d'une tâche (définie dans la section 3.3.2), la décision de la lancer ou non en mode spéculatif est prise. «BASE» propose une méthode pour le calcul de la durée de chaque tâche (spéculative ou non) directement à partir des estimations de la valeur de son taux de progression. Il utilise la méthode de Zaharia et al [171] dans l'algorithme «LATE»pour détecter les tâches qui ralentissent.
«BASE» présente la notion de "Vol de ressource" : supposons qu'un ensemble de ressources (Vcores) sur une machine particulière est affecté à un travail j et que cet ensemble de ressources n'est utilisé que par les tâches associées à j. Les expérimentations effectuées par Zhenhua et al. montrent que la durée entre la fin d'une ancienne tâche et le début de la tâche suivante sur ces ressources est une durée non négligeable. Ils proposent de récupérer ces ressources lorsqu'elles sont en attente de la tâche suivante du travail j pour exécuter des tâches appartenant à d'autres travaux. Elles seront restituées lorsqu'elles seront sollicitées de nouveau par une tâche de j. Cette opération s'appelle le "vol de ressource". Elle s'adapte en temps réel au taux d'utilisation des ressources sur les machines esclaves. Elle est transparente pour les ordonnanceurs et peut être utilisée avec tous les ordonnanceurs existants.
Après l'affectation des tâches par l'ordonnanceur de Hadoop, elles sont exécutées sur les machines esclaves en respectant la politique FIFO. Zhenhua et al proposent d'améliorer l'ordre d'exécution des tâches sur les machines esclaves en utilisant d'autres politiques. Les politiques qu'ils proposent sont :

- «STMA» (*Shortest Time Left Most Algorithm*) : les tâches avec un temps d'exécution faible auront les ressources en premier.
- «LTLM» (*Longest Time Left Most*) : les ressources seront affectées, en premier, à la tâches avec le temps d'exécution le plus long.
- «STM» (*Speculative Task Most*) : les tâches qui s'exécutent en mode spéculatif sont privilégiées. S'il n'y a pas des tâches en exécution spéculatif, la politique FIFO par défaut est appliquée.
- «LTM» (*Laggard Task Most*) : les tâches en retard sont lancées. Cette politique ne fait pas de distinction entre les tâches en spéculation et les tâches traditionnelles. La liste des tâches est ordonnée en fonction d'un indice de faisabilité (qui dépend du temps d'exécution des tâches). Les tâches sont, en conséquence, ordonnancées selon l'ordre trouvé.
- «Even» : consiste à partager équitablement les ressources entre les tâches existantes.
- FIFO : premier venu, premier servi.

Durant les expérimentations, l'approche "vol de ressource" améliore le temps d'exécution des tâches Map d'un facteur entre 5% et 58% selon la politique d'ordonnancement utilisée sur les machines esclaves. De façon générale, l'algorithme «BASE» améliore les résultats de la gestion des tâches en mode spéculatif. Elles ne seront lancées que si leurs exécutions améliorent le temps d'exécution du travail. L'utilisation de «BASE» et l'approche "vol de ressource" avec «LTM»

fournissent des résultats meilleurs que les résultats donnés par les autres politiques ou par les ordonnanceurs par défaut dans Hadoop. Cependant son utilisation avec des tâches qui demandent un accès intensif aux données montre une dégradation des performances en temps de date de fin des travaux.

3.3.4 Algorithme «MAESTRO» (*Replica-Aware Map Scheduling for MapReduce*)

«MAESTRO» cherche à minimiser le temps d'exécution des travaux et à augmenter le taux de tâches Map qui s'exécutent en mode local. Suite à l'étude effectuée dans [76], 23% des tâches Map s'exécutent en mode non local, 55% de ces tâches engendrent le lancement d'une exécution spéculatif et seulement 50% de celles-ci aboutissent et se terminent. «MAESTRO» vise à augmenter la performance de la grappe Hadoop en augmentant le taux de tâches Map qui s'exécutent en mode local. L'idée principale est d'exécuter les tâches Map en deux vagues. La première servira à remplir les ressources (Vcores) vides des machines en tenant compte des réplications des données. Durant cette vague, la prise en compte de la localité n'est pas fortement considérée durant l'affectation. La deuxième vague profitera des résultats et des métriques récupérés des tâches terminées pour optimiser l'affectation des tâches restantes. Les heuristiques utilisées sont :

— On sélectionne la machine esclave qui exécute (probablement) un nombre minimal de tâches Map en locale et les données qu'ils traitent.

— On sélectionne la machine esclave qui a un impact minimal sur l'exécution locale des tâches Map. Cette machine possède le plus faible taux de partage avec les autres machines. Le taux de partage d'une machine j est égal à $max_{1 \leq i \leq N, i \neq j} Sc_i^j$ avec Sc_i^j le nombre de portion de données partagées entre j et i. N est le nombre de machines sur la grappe.

— On sélectionne les portions de données qui ont la probabilité la plus élevée d'être traitées en non locale.

En fonction de la puissance de calcul, «MAESTRO» autorise aux machines possédant une bonne puissance de calcul d'exécuter des tâches en mode non local. D'après les évaluations effectuées par Ibrahim et al., «MAESTRO» améliore le taux d'exécution locale des tâches Map de 16% par rapport à l'utilisation de Hadoop avec l'ordonnanceur «FS». L'amélioration de temps d'exécution des tâches dépend du nombre de tâches Map. Elle est de 24% pour 200 tâches Map et de 4% pour 1600 tâches Map pour une grappe hétérogène. Elle est de 34% pour 200 Map et de 6% pour 1600 Map pour une grappe homogène.

3.3.5 Autres algorithmes

Aysan et al. [116] proposent l'algorithme «COSHH» (*Classification and Optimization based Scheduler for Heterogeneous Hadoop*). Cet algorithme a pour objectifs la minimisation de la moyenne des dates de fin des travaux et le partage équitable de la grappe entre les différents utilisateurs. Il considère en plus la localité des données lors de l'ordonnancement des tâches. Il se base sur deux heuristiques pour la classification des travaux et la gestion des listes. MRperf est le simulateur utilisé pour évaluer l'ordonnanceur «COSHH». Feng Yan propose DyScale [162] [163]

pour minimiser la date de fin des travaux sur des plateformes hétérogènes (virtuelles). Ils divisent les cœurs en deux types : cœur rapide et cœur lent. Ils classent les travaux en fonction du nombre de tâches et du type des applications. Les cœurs rapides sont utilisés pour l'exécution des applications interactives, tandis que les cœurs lents sont utilisés pour l'exécution des tâches en lots (batch).
Quan chen [26] propose «HAT» (*History based Auto-Tuning mapreduce*). Il suit en temps réel la progression des tâches durant les différentes étapes pour la détection des tâches en retard. C'est une extension de l'algorithme «LATE». Il se base sur un modèle d'estimation des différents coûts et des différentes étapes. Sur cette base, ils classent les machines en faible ou puissance et il gère l'affectation des tâches en conséquence. L'algorithme de Aboulnaga et al. [4] a pour objectif de minimiser le temps d'exécution totale d'un ensemble de travaux MapReduce.
Le travail de wang et al [150] cherche à classer les travaux MapReduce en travaux consommateurs de puissance de calcul ou de bande passante. Cet ordonnanceur est une évolution de FIFO qui se base sur les historiques des exécutions pour adapter le comportement de l'ordonnanceur nommé «CICS» (*CPU and I/O Characteristic Estimation Strategy*) selon le type du travail.
Tayson et al. [36] proposent "Online MapReduce", qui est une nouvelle architecture de MapReduce et qui consiste à envoyer directement les données de sortie des tâches Map vers les tâches Reduce. Ces dernières commencent le traitement de ces données dès leur réception.
Minghong et al. [102] cherchent à minimiser la moyenne des dates de fin des travaux. Ils considèrent le problème d'ordonnancement en lot et le problème d'ordonnancement en ligne. Ils proposent deux algorithmes complémentaires MaxSRPT et SplitSRPT qui fournissent des résultats à ratio de compétitivité constant lors de leurs évaluations en mode hors ligne, et les meilleurs résultats lors de la comparaison à d'autres politiques d'ordonnancement de la littérature.
Chang et al. [23] cherchent à minimiser la somme des fin des travaux. l'algorithme proposé est de facteur d'approximation 3 par rapport à l'optimale.

3.4 Algorithmes pour le respect de la qualité de service

Dans certains cas, le système doit répondre aux contraintes clients, qui imposent des dates de fin que le système doit respecter. Par exemple, Wolf et al. [155] proposent «FLEX» (*Flexible Scheduling Allocation*) un algorithme multiobjectif qui fait intervenir l'administrateur pour choisir les objectifs de l'ordonnancement parmi les suivants : réduire la somme des dates de fin des travaux, réduire le nombre des travaux en retard, minimiser la somme des retards ou améliorer la qualité de service et minimiser les coûts. «FLEX» modélise le problème d'ordonnancement comme un problème d'ordonnancement de tâches malléables.
Les algorithmes que nous présentons dans les sous-sections suivantes sont des algorithmes dont l'optimisation de la qualité de service imposée par l'utilisateur fait partie des critères à optimiser.

3.4.1 Algorithme «DCS» (*Deadline Constraint Scheduler*)

«DCS» [88] est un algorithme d'ordonnancement qui optimise le taux d'exploitation de la grappe Hadoop et la qualité de service (le respect d'une date de fin fixée par l'utilisateur).

Afin d'assurer le respect de la date de fin des travaux, deux modules principaux sont mis en place : *(i)* un module pour l'estimation des coûts d'exécution des tâches en fonction des données et *(ii)* un module d'ordonnancement par contraintes. «DCS» teste les performances de la grappe pour déterminer si le travail peut être exécuté ou non.
Le module d'estimation détermine le nombre de tâches Map et Reduce nécessaires pour assurer le respect de la date de fin. Une fois que le travail est accepté, il sera ajouté à une file d'attente de l'ordonnanceur en fonction de sa priorité. L'ordonnancement des tâches est, en conclusion, basé sur :

— Le résultat des estimations du temps d'exécution des travaux.

— Le nombre minimal des tâches nécessaires pour respecter la date de fin imposée par l'utilisateur.

«DCS» relâche la contrainte de localité des données durant l'ordonnancement des tâches pour pouvoir respecter les contraintes imposées par l'utilisateur. Lors de son évaluation, «DCS» a pu respecter la date de fin imposée par l'utilisateur. «DCS» est évalué seulement avec le travail "*WordCount*". «DCS» n'est pas évalué dans le cas de plusieurs travaux qui s'exécutent en concurrence. De ce fait, il ne prend pas en considération le retard ou le temps d'attente subi de l'exécution des autres travaux.

3.4.2 Algorithme «EDCS» (*Extended Deadline Constraint Scheduler*)

«EDCS» [125] est une extension de l'algorithme «DCS» . Il apporte des solutions aux insuffisances rencontrées dans «DCS» . Il prend en considération l'amélioration de la qualité de service et l'exécution concurrente de plusieurs travaux. «EDCS» se base sur une phase d'apprentissage pour prédire le temps d'exécution des travaux futurs.
Polo et al. [125] ont étudié trois méthodes pour calculer les estimations du temps d'exécution : *(1)* La première se base sur le nombre de tâches soumises, en cours et restantes. *(2)* La deuxième se base sur la taille des données d'entrée, le temps écoulé et la taille des données de sortie des tâches Map terminées. *(3)* La troisième se base sur l'état de l'exécution d'un travail à un instant particulier. En plus, le modèle d'estimation permet la prédiction du temps d'exécution restant pour un travail de façon dynamique. Il aide à ajuster les ressources (Vcores) allouées, pour assurer le respect de la qualité de service.
La politique d'ordonnancement des tâches dépend de la classe à laquelle les travaux sont affectés et de leurs priorités. Trois classes principales sont définies :

— UNDEAD : le travail peut s'exécuter mais il dépassera sa date de fin prévue. Le travail ne peut pas estimer la durée du travail.

— NODATA : il n'y a pas assez de données pour estimer les besoins en ressources du travail.

— ADJUST : l'ordonnanceur est capable de décider si le travail a besoin de plus ou de moins de ressources.

En fonction de l'interaction avec les utilisateurs, les besoins des travaux dans les classes UNDEAD et NODATA sont mis à jour à des intervalles réguliers. Du moment qu'un utilisateur révise ces décisions (la définition de la date de fin attendue) ou des ressources se libèrent, l'acceptation des

travaux est recalculée. Les évaluations de cet algorithme ont montré un taux élevé (de l'ordre de 80%) des travaux qui se terminent avec le respect de la qualité de service.

3.4.3 Algorithme «RATCTP» (*Resource-Aware Scheduling Through Coupling Task Progress*)

Cet ordonnanceur [141] cherche à minimiser le temps total d'exécution d'un travail en assurant un partage équitable des ressources au niveau des tâches Map et Reduce.
Les ordonnanceurs implémentés dans Hadoop, tels que «FS», divisent l'ordonnancement des tâches Map et Reduce en deux étapes indépendantes. Ils partagent équitablement les ressources seulement au niveau des tâches Map et affectent un nombre fixe de ressources aux tâches Reduce. En général, ils leur affectent les mêmes ressources récupérées après la fin des tâches Map.
«RATCTP» rend l'ordonnancement des tâches Map et Reduce dépendant car il propose une méthode qui se base sur trois idées : *(1)* En fonction de la durée des tâches Map, générer de façon aléatoire la possibilité ou non de les ordonnancer sans tenir compte de la localité des données. *(2)* La deuxième idée consiste à exécuter les tâches Reduce en plusieurs vagues. *(3)* La troisième consiste à leur ajouter un facteur de retard si la machine la plus adaptée n'a pas de ressources disponibles.
Après leur démarrage, les tâches Reduce essayent de récupérer les données des tâches Map. Cette phase ralentit considérablement le système, vu l'accès concurrent aux données. L'idée dans «RATCTP» est d'organiser l'accès concurrent des tâches Reduce aux données des tâches Map. Cet accès est géré par l'ajout d'une durée de retard lors de l'exécution des tâches Reduce. Cette durée dépend de la charge au niveau de la grappe et de la priorité des tâches Reduce. La priorité des tâches Reduce est définie en fonction de la progression des tâches Map.
La politique d'exécution des tâches en plusieurs vagues est un inconvénient puisque le travail ne peut pas profiter de la totalité des ressources disponibles pour accélérer son exécution. Il doit attendre la deuxième vague pour exploiter toute la grappe. Dans certain cas, l'ordonnanceur ne considère pas la prise en compte de la localité des données pour pouvoir maximiser l'exploitation des ressources. Au lieu d'attendre une opportunité pour augmenter la localité des données, il préfère utiliser les ressources disponibles.
Les simulations de «RATCTP» montrent que dans le cas d'une grappe chargée (le nombre de tâches est largement supérieure au nombre de Vcores disponibles), «RATCTP» améliore les performances de Hadoop entre 5 et 20% par rapport à l'ordonnanceur «FS». Lorsque la grappe est non chargée, «RATCTP» dégrade les performance de Hadoop, ces dégradations dépendant du travail utilisé dans l'évaluation.

3.4.4 Algorithme «ARIA» (*ARIA-SLO-based scheduler*)

«ARIA» [148] est un système de prédiction et ordonnancement de travaux MapReduce qui a pour objectif d'améliorer la performance de leur exécution en respectant une date de fin donnée. Il est composé de trois éléments principaux (Figure 3.4) : le premier s'occupe de créer un profil pour le travail synthétisant sa consommation en ressources durant l'exécution des tâches Map et Reduce et la durée des différentes tâches. Le deuxième élément se base sur un modèle de

performance pour estimer la quantité de ressources nécessaires pour le respect de la date de fin imposée par l'utilisateur. Le troisième élément est l'ordonnanceur qui s'occupe de l'affectation des tâches et de la quantité de ressources à affecter à chaque tâche. «ARIA» est basée sur une politique d'ordonnancement connue, nommée «EDF» (*Earliest Deadline First*) et sur un modèle mathématique pour calculer le nombre de ressources (Vcores) Map et Reduce à utiliser pour assurer le respect de la date de fin (la SLA). L'évaluation de cet ordonnanceur a été effectuée en utilisant le simulateur fourni par IBM nommé MRperf [149] avant d'être implémentée et évaluée sur une grappe Hadoop de 66 machines. L'évaluation sur une grappe physique a montré que plus de 95% des travaux respectent la date de fin estimée par ARIA. Cependant, en moyenne, il y a un décalage de 5 à 15% entre la date de fin moyenne obtenu lors de l'évaluation physique et la date de fin moyenne estimée par simulation.

3.4.5 Algorithme «MRA++» (*Scheduling and data placement on MapReduce for heterogeneous environments*)

«MRA++» [11] est destiné à ordonnancer des tâches sur des plateformes hétérogènes. Depuis sa conception, Julio et al. focalisent sur la classification des machines en groupes, la distribution des données sur la grappe (en fonction de ces groupes) et enfin sur l'ordonnancement des tâches. Il propose trois politiques : *(1)* pour classer les machines en fonction de leurs puissances de calcul, *(2)* pour répartir les données en fonction de la puissance calculée des machines et *(3)* pour ordonnancer les tâches en fonction des données qu'ils traitent. Les simulations montrent que les résultats obtenus nous permettent de gagner 70% en performance par rapport à la version par défaut implémentée dans Hadoop. Le simulateur SimGrid est utilisé lors de l'évaluation.

3.4.6 Algorithme «DPS» (*Dynamic Proportional Share*)

Thomas et al. [132] ont proposé l'ordonnanceur «DPS» qui est une extension de FIFO. Les idées implémentées consistent en : *(1)* un module de validation qui décide si un travail peut s'exécuter sur la grappe ou non, *(2)* l'administrateur de la grappe affecte un budget à chaque

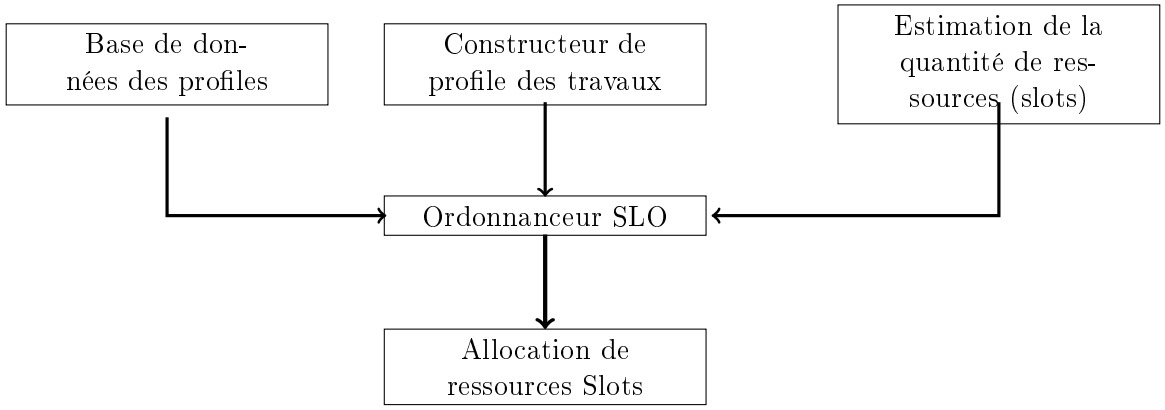


FIGURE 3.4 – Architecture utilisée dans l'ordonnanceur «ARIA»

utilisateur. Ce dernier décide de la façon dont il dépensera ce budget. De ce fait, l'utilisateur aide à définir les priorités entre les différents travaux.
«DPS» cherche à minimiser les deux critères suivants : la date de fin d'exécution des travaux et le budget par utilisateur. «DPS» permet aux utilisateurs de contrôler la quantité de ressources allouées en équilibrant leurs besoins à travers le temps. «DPS» permet en plus l'obtention des ressources (Vcores) sur la base du partage proportionnel. Ce partage dépend du prix que l'utilisateur est prêt à payer et des ressources utilisées par les utilisateurs. «DPS» utilise plusieurs techniques telles que le blocage des listes et la préemption des tâches pour éviter la famine.
Afin de répondre à la contrainte du respect de la date de fin d'exécution des tâches imposées par les utilisateurs, l'ordonnanceur doit décider de la puissance nécessaire à fournir pour respecter la qualité de service et exécuter les travaux en fonction du budget utilisateur.
«DPS» est inspiré des théories des jeux et des algorithmes d'ordonnancement des agents tels que dans [53]. Il peut être configuré pour simuler le comportement des autres ordonnanceurs. Par exemple, si on ne prend pas en compte la gestion des listes, DSP se comporte comme étant l'ordonnanceur FIFO. Si les listes partagent les capacités de la grappe et les intervalles d'allocations sont suffisamment grands, il simule le comportement de l'ordonnanceur Fair-Shair.
DSP permet la gestion parallèle des travaux appartenant à plus de 80 files d'attente (la seule limite est la mémoire de la machine sur laquelle «DPS» s'exécute). Cet avantage est expliqué par le fait que la priorité des travaux est définie par les utilisateurs de la grappe (lors de la gestion de leurs budgets) et non par des heuristiques. Ce qui allège considérablement l'utilisation des ressources sur la machine maître et fournit à l'ordonnanceur plus de puissance.

3.5 Algorithmes basés sur le traitement des données en locale pour améliorer les performances

La plupart des algorithmes présentés dans cette section sont multiobjectifs. Nous présentons dans cette section des algorithmes dont l'optimisation de la proportion des taches, qui s'exécutent sur les machines contenant les données qu'elles traitent, fait partie des critères à optimiser.

3.5.1 Algorithme «PURLIEUS» (*Locality-aware resource allocation*)

«PURLIEUS» [117] est un ordonnanceur développé par IBM pour l'allocation de ressources pour des travaux de type MapReduce sur le Cloud. Les machines virtuelles sont considérées comme des unités de calcul (vu qu'elles exécutent les tâches Map et Reduce). L'objectif de cet ordonnanceur est l'amélioration de la localité des données intermédiaires générées par les tâches Map. Les ordonnanceurs traditionnels dans Hadoop cherchent à rapprocher les tâches Map des données qu'ils traitent mais l'affectation des tâches Reduce est aléatoire. «PURLIEUS» s'intéresse au rapprochement au niveau des deux types de tâches Map et Rduce des données qu'ils traitent. Il classe les travaux selon la quantité de données auxquelles accèdent en *(1)* travaux $MapinputHheavy$ caractérisés par des données d'entrée volumineuses et des données intermédiaires négligeables,*(2)* travaux $Map-and-Reduce-inputheavy$ caractérisés par des données d'entrée et intermédiaires volumineuses,*(3)* travaux $Reduce-input-heavy$ caractérisés

par des données intermédiaires volumineuses. Le type de travail est défini par l'utilisateur dès la soumission. «PURLIEUS» propose une politique d'ordonnancement des tâches en réduisant leurs distances par rapport aux données qu'ils traitent en fonction du type de travail. Par exemple, pour un travail du type (1), il propose que les tâches Map s'exécutent les machines contenant leur données d'entrée et autorise les tâches Reduce à s'exécuter n'importe où sur la grappe vu que les données intermédiaires ne dégradent pas l'utilisation de la bande réseau. Pour les travaux de type (3), le comportement est inversé : «PURLIEUS» ordonnance en rapprochant les tâches Reduce des données intermédiaires. Ces critères supposent l'existence d'un système qui permet d'obtenir les métriques systèmes liées à l'exécution des travaux, telles que la charge engendrée et la quantité de données transférées par réseau. Ces connaissances permettent la construction d'un profil pour chaque travail. Ensuite, ces travaux sont ordonnancés en fonction de leur profils et leurs classes. «PURLIEUS» se base sur des fonctions de calcul de coûts. Le modèle de coût permet le calcul du coût total d'un travail A sur la donnée D ($Cost(A, D)$) qui est la somme des coûts des Map ($M_{cost}(A, D)$) et Reduce ($R_{cost}(A, D)$). Les coûts des tâches Map et des tâches Reduce sont définis en fonction des distances entre les machines et de la quantité des données échangées.
Le problème du placement des données et des unités de calcul est modélisé comme étant le problème de rangement « bin-backing » qui est un problème NP-Difficile. «PURLIEUS» est évalué par le simulateur PurSim, basé sur MRPerf [149]. En fonction des scénarios utilisés, «PURLIEUS» améliore la date de fin de tous les travaux de 50% et réduit le nombre de tâches qui s'exécutent en mode "$Rack - local$" de 70%.

3.5.2 Algorithme «MS» (*Matchmaking Scheduling*)

He et al. [72] proposent un algorithme proche de l'algorithme «DS». L'idée est de retarder l'affectation des tâches Map pour améliorer la prise en compte de la localité des données. La différence est que «DS»ne prend pas en compte la localité au niveau des tâches Reduce. L'algorithme proposé est simple. Il profite des politiques des autres ordonnanceurs tels que FIFO ou «FS» pour l'affectation des tâches en locale. S'il n'y a pas une tâche dans la liste qui peut s'exécuter en locale sur une machine (avec des Vcores libres), deux comportements sont définis : le premier est défini lorsque l'ordonnanceur reçoit une première demande de tâche, il consiste à ignorer la demande de tâche en espérant que la machine aura une tâche qui s'exécute en locale. Le deuxième consiste à affecter à cette machine une tâche en mode non local lors de sa deuxième demande de tâche. L'algorithme associe un indicateur de localité à chaque machine. Il compte le nombre de fois où la machine n'a pas reçu une tâche s'exécutant en mode local. Cet indicateur est réinitialisé à zéro à chaque fois que la machine reçoit une tâche qui s'exécute en mode local où à chaque fois qu'un nouveau travail est soumis. Lorsque l'ordonnanceur cherche à ordonnancer une tâche pour la première fois, il cherche à respecter le critère de localité sinon cette tâche ne sera pas affecté et l'indicateur de la machine est incrémenté de 1. Si la valeur de cet indicateur sur une machine est différente de zéro lors d'un deuxième souhait d'affectation, une tâche (appartenant au premier travail de la liste des travaux) est affectée à cette machine sans tenir compte de la localité des données.

Malgré la simplicité de cette méthode, les évaluations effectuées par He et al. [72] montrent que «MS» fournit des performances meilleures en temps d'exécution que «DS»en utilisant FIFO et «FS» .

3.5.3 Algorithme «CoGRS» (*Center-of-Gravity Reduce Task Scheduling*)

L'étude effectuée dans [68] a montré que l'étape de transfert des données vers des tâches Reduce (étape nommée « Shuffle ») et le temps de lancement de ces dernières influencent considérablement le trafic réseau.
Hammoud et al. propose l'ordonnanceur CoGRS [68]. Ses objectifs sont la minimisation du trafic réseau et la minimisation du temps global d'exécution des travaux. L'idée implémentée consiste à changer la politique d'ordonnancement aléatoire des tâches Reduce, existante dans «FS», «CS» «LATE» (etc.), par une méthode qui prend en compte la localité entre les données intermédiaires et les tâches Reduce. «CoGRS» calcule pour chaque tâche Reduce un centre de gravité. Ce dernier sera la machine sur laquelle la tâche s'exécutera. Les tâches Map sont ordonnancées en utilisant la politique par défaut dans Hadoop.
La topologie réseau dans hadoop est modélisée sous la forme d'un arbre. La bande passante entre deux machines est exprimée sous la forme d'une distance. Celle entre deux machines est la somme de la distance de chacune des deux à leur père commun. La distance entre une machine et son père est égale à 1. Pour chercher la machine centre de gravité d'une tâche Reduce R, Hammoud et al. cherchent la machine qui minimise la variable TND_R. Si plusieurs machines possèdent la même valeur de TND_R, ils cherchent la machine qui minimise $WTND$ parmi cette ensemble de machines. Ces deux variables sont définies comme suit :

- $TND_R = \sum_{i=0}^{n} ND_{iR}$, représente la distance réseau totale associée à la tâche Reduce R. Elle est la somme des distances entre la machine contenant R et toutes les machines contenant les tâches Map qui la précèdent. n est le nombre de portions de données à transmettre à R. ND_{iR} est la distance nécessaire pour transférer la partition de données intermédiaire i à la machine qui exécutera la tâche R.
- $WTND = \sum_{i=0}^{n} w_i \times ND_{iR}$, est le poids total de la distance réseau pour une tâche Reduce R. Elle prend en considération la localisation des machines exécutant des tâches Map et la taille des données générées par ces Maps pour désigner leurs poids. Le poids d'une partition i est $w_i = \frac{\text{taille de i}}{\text{Somme de toutes les tailles des partitions demandées par R}}$

Hadoop permet, durant l'étape « Shuffle », de commencer le transfert de données avant la fin des tâches Maps. Cette méthode a prouvé ses capacités à améliorer les performances d'exécution des tâches. CoGRS déclenche cette étape après qu'une partie des tâches Map aient terminé leur exécution. Les évaluations ont montré que CoGRS améliore l'exécution des travaux d'un taux variable entre 3.2% et 6.3% par rapport à la version standard d'Hadoop. Il améliore l'exécution locale des tâches Reduce de 1%, 32% et 57,9% dans Hadoop avec 8, 16 et 32 machines respectivement sur une grappe amazon EC2. "CoGRS" est une évolution de l'algorithme «LARTS» [69] (section 3.5.6) qui est à son tour une évolution de «LEEN»[75] et «LARS»[25] .

3.5.4 Algorithme Mantri

Cet ordonnanceur [10] est présenté par Microsoft. Contrairement à FaceBook, les cas d'utilisation de la grappe Hadoop chez Microsoft se basent sur l'exécution de tâches de longue durée. Mantri analyse en temps réel les exécutions des tâches : s'il détecte un problème, il en analyse la cause et lance une ou plusieurs méthodes pour le résoudre. Il est une évolution de l'algorithme d'exécution spéculatif «LATE». Mantri utilise plusieurs techniques telles que le redémarrage des tâches, la prise en compte de la localité des données et la protection des données de sortie des tâches importantes. L'importance de la tâche dépend du volume de ses données de sortie et du pourcentage de son avancement. Mantri possède des outils pour la génération en temps réel des métriques des exécutions des travaux et de l'utilisation du réseau. Il détecte les anomalies en utilisant les causes, les ressources et les estimations des coûts des actions à effectuer.
Mantri se base sur le principe d'analyse des causes pour lancer les traitements appropriés. Si la cause du retard d'une tâche est la saturation des ressources sur une machine, la tâche sera redémarrée ou dupliquée (en spéculatif) sur une autre machine qui permet d'accélérer son exécution (en tenant compte de la bande passante à utiliser). Mantri n'attend pas la fin de l'une des deux tâches (originale ou copie) pour arrêter l'autre. Lors de l'exécution spéculatif, Mantri profite des métriques systèmes et suit en temps réel l'avancement des tâches : si la tâche lancée en spéculation est en avance, la tâche d'origine sera arrêtée. Cet ordonnanceur vise à assurer l'exécution locale des tâches et la minimisation de la charge réseau. La dernière astuce utilisée consiste à identifier les tâches critiques et à dupliquer leurs données de sortie pour limiter la concurrence d'accès à ces données.

3.5.5 Algorithme Tarazu

Tarazu [8] est un mot Urdu qui a le sens du mot balance. Ce nom traduit en partie l'objectif de cet ordonnanceur. Son but est la minimisation du temps d'exécution des tâches Map et Reduce en assurant un équilibrage (charge et consommation réseau) lors des différentes exécutions. Tarazu [8] se base sur les caractéristiques du travail, de la grappe, de l'échantillonnage en temps réel pour assurer l'exécution des tâches.
L'échantillonnage de l'exécution de travaux en temps réel consiste à récupérer chaque laps de temps les informations concernant l'avancement de l'exécution des tâches, la quantité des données qu'elles traitent, l'état des machines de la grappe (etc.).
Tarazu [8] est constitué de trois éléments :

— CALB (Communication-Aware Load Balancing of Map computation) : cet élément gère l'équilibrage des charges de calcul lors de l'exécution des tâches Map. Il favorise l'exécution locale des tâches Map (dans le cas des données intermédiaires volumineuses) et tient compte de l'hétérogénéité des ressources. Il décide si l'exécution distante des tâches Map sera bénéfique ou non.

— CAS (communication-aware Scheduling of Map computation) : ce composant calcule par machine, combien de tâches Map seront exécutées en mode non local et quand les lancer. Tarazu lance des tâches dans les deux modes (locale ou non) pour s'exécuter en même temps

sur les machines les plus puissantes de la grappe. Cette politique fournit un équilibre au niveau des machines entre la puissance de calcul et la capacité de transfert du réseau. Elle permet dans le cas des exécutions en mode non local de compenser le temps perdu pour transférer les données vers ces machines par la puissance de calcul de ces dernières.

— PLB (Predictive Load Balancing of Reduce computation) : cet élément se base sur les résultats des exécutions des tâches Map pour prédire les affectations possibles des tâches Reduce. Dans la version par défaut d'Hadoop, le nombre de tranches des données intermédiaires des tâches Map est égal au nombre de tâche Reduce. L'idée de PLB est de créer plus de tranches que dans le comportement par défaut. Il crée quatre fois le nombre de tranches supposées être utilisées. Ce qui réduit la taille des données par tranche à transférer à travers le réseau.

L'affectation des tranches de données aux tâches Reduce se fait en tenant compte de la puissance des machines. Vu que la phase "shuffle" commence avant la fin de toutes les tâches Map, le calcul du nombre de tranches à utiliser n'est pas possible. La solution proposée est de commencer par un nombre prédéfini (ce nombre doit être un multiple du nombre défini par défaut des tranches) et mettre à jour cette valeur durant la phase Reduce.
La solution proposée est que CAS limite le nombre de tâches en exécution non locale qui peuvent effectuer un accès distant et concurrent à une tranche de données. Il interdit aux machines lentes l'exécution de tâches sur des données qui possèdent des répliquas sur des machines rapides. Il interdit aux machines exécutant un nombre de tâches Map en mode data locale inférieure à un seuil donné d'exécuter des tâches distantes (en mode non locale).

3.5.6 Autres algorithmes

L'algorithme «LARTS» (*Locality-Aware Reduce Task Scheduler*) [69] focalise sur la localité des tâches (Reduce et Map) lors de l'ordonnancement. Une idée utilisée par cet algorithme est de commencer le transfert des données intermédiaires avant la fin des tâches Map vers les tâches Reduce. Il est le premier ordonnanceur à utiliser cette idée dans le modèle MapReduce et à l'implémenter dans Hadoop. Il améliore les performances en réduisant la durée des travaux. Il augmente la localité entre les données et les traitements et réduit le trafic réseau.
Zhou et al. [142] s'intéressent à l'exécution d'un ensemble de travaux formant un graphe acyclique direct sur des grappes hétérogènes. L'algorithme proposé classifie les travaux en travaux intensif en I/O et intensif en calcul (qui demande plus de ressources de calcul). Il affecte, en conséquence, des priorités aux travaux en fonction de cette classification. La dernière phase consiste à ordonnancer les tâches en fonction de la taille des données et en respectant la localité des données.
Xiaoyn[140] propose «ESAMR» (*Enhanced Self-Adaptive MapReduce*) qui se base sur les techniques d'apprentissage pour classer les machines (algorithme K-means) en plusieurs clusters. Ses techniques calculent un poids pour chaque classe de machines et pour chaque tâche soumise. Ensuite, l'algorithme «ESAMR» affecte les tâches aux machines possédant le poids le plus proche de leurs poids.

3.6 Problème d'optimisation de la consommation énergétique

La réduction de la consommation énergétique est un critère d'optimisation identifié pour les centres de calcul. Les travaux de Chen et al. [28][29] et Leverich et al [100] étudient l'efficacité énergétique en variant le nombre d'éléments (démons) Hadoop par machines esclaves. Bampis et al. [105] cherche à minimiser la somme des poids des dates de fin en respectant un budget d'énergie. Ils proposent deux algorithmes de liste polynomiaux appelés EMRSA 1 et EMRSA 2 avec un facteur d'approximation constant. Les algorithmes sont évalués par simulation et permettent de réduire d'environ 40 % la consommation énergétique d'une grappe Hadoop utilisant le «FS».

3.6.1 Mécanismes de réduction de la consommation énergétique

Dans la littérature, la résolution se base principalement sur les deux approches suivantes.

Mécanismes de mise hors tension

On s'intéresse à l'analyse des systèmes avec deux états. Un état actif dans lequel le système consomme une unité d'énergie pendant une unité de temps et un état de sommeil où le système ne consomme aucune unité d'énergie pendant une unité de temps. Comme entrée de ce problème, on a une séquence alternée de périodes de temps actifs et sommeils. L'objectif est de trouver un ordonnancement pour spécifier la date de mise hors tension (si nécessaire) pour chaque intervalle de repos. On peut noter que ce problème de minimisation de la consommation énergétique est équivalent au problème de location des outils de ski [80].
Comme exemple, on aborde dans ce paragraphe l'étude du problème avec une période de repos I. On présente pour ce problème un algorithme hors ligne optimal et trivial : si on nomme I la durée de repos ($|I| > 0$), L'algorithme consiste à changer vers le mode sommeil immédiatement au début de I, sinon il consiste à rester dans le mode actif pendant la durée de I. On présente pour ce problème un algorithme en ligne nommé PDalg-2 [80]. Il suppose que pour chaque période de repos I, on a une consommation énergétique qui est au plus c (≥ 1) fois la consommation optimale sur l'instance I. PDalgo-2 ignore le temps de latence engendré par le temps de remise sous tension vers le mode actif.
PDalg-2 consiste à changer vers le mode sommeil après D unités de temps si la période de repos I n'est pas finit. PDalg-2 est de compétitivité 2.
En utilisant la randomisation, Karlin et al. [85] ont pu améliorer le ratio de compétitivité en proposant un algorithme qu'on nomme RPDalg-2. Ce dernier atteint un rapport de compétitivité de $\frac{e}{(e-1)} \approx 1.58$.
Dans les deux algorithmes présentés, on prend en considération seulement deux états (actif ou sommeil). Irani et al. ont généralisé les résultats présentés sur plusieurs états d'une machine. Ils ont proposé un premier algorithme dans [80] de compétitivité égale à 2. Cet article contient une étude expérimentale sur des machines IBM avec quatre états. Le deuxième algorithme, proposé dans [79], est de rapport de compétitivité qui atteint $3 + 2\sqrt{2} \approx 5.8$ pour 10 états.
Dans le cadre de Hadoop, GreenHDFS [86] divise la grappe en deux classes de machines : les machines de la la classe actif sont toujours actif et les machines de la classe sommeil peuvent

entrer en mode sommeil. Il place les données en fonction de ces deux classes de machines. La réduction de l'énergie consiste à changer l'état des serveurs entre les deux classes de machines.

Mécanismes de réajustement dynamique de la vitesse des processeurs (DVFS)

Les processeurs de la nouvelle génération, tels que les processeurs d'intel (Xscale et Speed Step) et AMD (Power now), peuvent exécuter des instructions avec une vitesse variable. Plus la vitesse d'un processeur est élevée plus sa consommation électrique est élevée. L'objectif est de trouver des algorithmes qui utilisent la vitesse des processeurs de façon optimale.
On se concentre donc dans cette partie sur la présentation des principes de base et des algorithmes les plus connus mettant en place cette méthode.
On considère le problème illustratif suivant : soit une séquence $I = J_1, J_2, ..., J_n$ de tâches, soit r_i la date d'arrivée de la tâche J_i et d_i la date de fin au plus tard autorisée. On suppose que $r_i \leq r_{i+1}, 1 \leq i < n$. La durée d'exécution p_i de J_i doit être calculée. Cette valeur peut être considérée comme le nombre de cycles CPU nécessaires pour terminer l'exécution de la tâche J_i. Le temps nécessaire pour l'exécution de J_i est égal à $\frac{p_i}{s}$ avec s la vitesse CPU, considérée comme constante durant l'exécution de la tâche. On considère dans ce contexte qu'il n'y a pas de relations de précèdence entre les tâches. L'objectif est de trouver un ordonnancement faisable de ces tâches en tenant compte de leur date d'arrivée et de leur date de fin au plus tard souhaitée.
Un premier travail a été présenté par Yao et al. [164] prenant en compte les contraintes suivantes :*(i)* les tâches sont interruptibles, *(ii)* il n'y a pas de borne supérieure pour la vitesse. Les deux scénarios en ligne et hors ligne sont traités. Dans la partie hors ligne, un algorithme nommé YDS a été proposé. Il se base sur la politique «EDF» (ordonnancer les tâches avec les dates de fin les plus proches).
Dans la partie en ligne de ce travail, deux algorithmes ont été proposés : Le premier est nommé « Average Rate ». Il se base sur le calcul de la plus petite vitesse moyenne nécessaire pour exécuter les tâches dans les temps. Le deuxième est nommé «Optimal Available». Il se base sur «EDF» (comme dans YDS) pour le calcul de la vitesse CPU nécessaire à l'exécution des tâches. Le ratio de compétitivité c de « Average Rate » est $\alpha^\alpha \leq c \leq 2^{\alpha-1}\alpha^\alpha$ et est égale à α^α dans « Optimal Available ». Plusieurs autres travaux dans d'autres contextes ont été présentés tels que FSA et « Round Robin » dans [15][81].
Cette méthode d'optimisation énergétique n'est pas abordée dans les travaux de recherche sur Hadoop car ce dernier ne fournit pas les outils techniques permettant la gestion de la fréquence des processeurs. Wirtz et al.[154] étudient l'influence de l'utilisation de la méthode DVFS [165] sur la consommation énergétique d'une grappe Hadoop. Ils comparent trois politiques DVFS et montrent que la politique CPUMiser [60] permet de gagner entre 0 et 4% de la consommation énergétique (par rapport à la consommation d'une grappe Hadoop basée sur «FS») en fonction des travaux utilisés dans l'évaluation sans perte en performances (sans retarder la date de fin des travaux). Lorsque la perte de 5% en performances est autorisée, CPUMiser réduit la consommation énergétique d'environ 23%.

3.6.2 Algorithme «HybridMR» (*Hierarchical Mapreduce Scheduler for Hybrid data centers*)

Sharma et al. [138] traitent du problème d'ordonnancement sur une grappe Hadoop hybride. Cette plateforme est constituée de machines virtuelles (VM) et de machines physiques. «HybridMR» [138] a pour objectif de minimiser le temps d'exécution des tâches et la consommation énergétique. L'idée principale est la séparation entre les flux de travaux qui doivent s'exécuter dans un environnement virtuel et les flux de travaux qui doivent s'exécuter dans un environnement physique. La solution proposée consiste à affecter en premier lieu les travaux des applications non interactives aux VMs et ensuite à les ordonnancer. Lorsqu'un travail MapReduce est soumis, il est lancé séparément sur une petite grappe d'apprentissage contenant les deux types d'environnement. Ensuite, après la comparaison des résultats des estimations des deux instances de chaque travail (une en native et l'autre en virtuel), une méthode statistique de profilage est utilisée pour : *(1)* effectuer l'estimation du temps d'exécution d'un travail et *(2)* calculer le niveau de surcharge des performances de la grappe virtuelle. Si le niveau de surcharge n'est pas important alors le travail est exécuté sur une grappe virtuelle, sinon il sera traité sur une grappe physique. Deux éléments assurent le bon fonctionnement de «HybridMR» :

— DRM (Dynamic Resources Manager) : il calcule la quantité de ressources et le temps d'exécution de chaque tâche. Ces informations sont utilisées pour créer un profile de chaque travail et optimiser les décisions de leur ordonnancement.

— IPS (Interface Prevention System) : c'est un outil de monitoring en temps réel. Il surveille les performances des applications interactives pour détecter si ces applications n'ont pas reçu assez de ressources, afin de satisfaire leur besoin en qualité de service, et réagir pour fournir une solution.

Les évaluations de cet ordonnanceur sont effectuées sur 24 serveurs physiques et 48 VMs. Elles montrent que «HybridMR» améliore le temps d'exécution des travaux MapReduce de prés de 40% dans un environnement virtuel par rapport à une grappe basée sur Hadoop avec l'ordonnanceur «FS». Il exécute des applications interactives en respectant les contraintes de temps qu'elles imposent. Il permet ainsi de réduire la consommation énergétique de 43% par rapport à une grappe basée sur Hadoop avec l'ordonnanceur «FS».

3.7 Tableau de synthèse des différents travaux

Les tableaux 3.1 et 3.2 synthétisent les travaux étudiés dans ce chapitre et comparent les objectifs qu'ils cherchent à optimiser. La première colonne présente les noms des algorithmes d'ordonnancement, la deuxième présente la liste des algorithmes où les auteurs ont proposé des implémentations dans Hadoop. La troisième colonne présente les algorithmes dont l'objectif ou l'un des objectifs est la réduction de la date de fin des travaux soumis. La quatrième colonne présente la liste des algorithmes qui cherchent à respecter une date de fin imposée par l'utilisateur. La cinquième colonne présente les algorithmes qui ont pour objectif le partage équitable des ressources de la grappe entre les utilisateurs, travaux et tâches. La sixième colonne présente

les algorithmes qui cherchent à améliorer la consommation énergétique. La septième colonne présente les algorithmes qui intègrent une alternative de la technique d'exécution spéculatif. La huitième colonne présente les algorithmes qui servent à ordonnancer les travaux MapReduce sur des grappes hétérogènes.

Durant l'étude de l'état de l'art associé à l'ordonnancement des tâches MapReduce, nous avons remarqué des pistes d'amélioration pour certains algorithmes.

Parmi les algorithmes étudiés, nous avons remarqué l'existence de liens entre plusieurs d'entre eux. Ces liens peuvent être décrit comme des relations d'évolution. Par exemple, si un algorithme "X" est une évolution d'un algorithme "Y", on constate que les auteurs de "Y" se sont basés sur l'algorithme "X" pour mettre en place l'algorithme "Y". Tel est le cas de l'algorithme «FS» qui se base sur l'algorithme "Max-min Fairness".

Nous regroupons dans la figure 3.5 certains algorithmes qui présentent des relations d'évolution existantes entre eux. Les algorithmes existant dans la figure ne synthétisent pas la totalité des travaux de recherche abordés dans ce chapitre. Cette figure focalise sur les algorithmes dont nous avons identifié l'existence d'une relation d'évolution entre eux. Nous identifions ces relations lorsque le auteurs les mentionnent ou lorsque nous remarquons qu'un algorithme développe les idées clés d'un autre algorithme appartenant à une publication plus ancienne. Par exemple, les algorithmes Mantri, «BASE» , «HAT»et «MRA++» sont des évolutions de l'algorithme «LATE»qui est, à son tour, une évolution de l'algorithme d'exécution spéculatif intégré par défaut de Hadoop.

3.8 Conclusion

Dans ce chapitre, nous focalisons sur le problème d'ordonnancement des travaux MapReduce. Nous énumérons les travaux de recherche les plus référencés et nous les regroupons en fonction de l'objectif qu'ils cherchent à optimiser.

À la fin de ce chapitre, nous regroupons les heuristiques étudiées dans un seul tableau. Nous les comparons en fonction des critères qu'ils cherchent à optimiser.

Le problème d'ordonnancement des travaux MapReduce est abordé de plusieurs manières. Il possède plusieurs contraintes et sa modélisation est complexe. Les différents travaux étudiés dans l'état de l'art ne prennent pas en compte toutes les contraintes du modèle et se contentent de certaines qu'ils jugent pertinentes. Dans les prochains chapitres, nous proposerons un modèle du problème que nous abordons et nous proposerons des heuristiques de résolution.

Politique d'ordonnancement	Implémenté dans Hadoop	Réduction de la date de fin des travaux	Respect de la date de fin imposée par l'utilisateur	Réduction du trafic réseau	Partage équitable des ressources	Amélioration de la consommation énergétique	Algorithmes d'exécution spéculative	Adapté à une grappe hétérogène
	Groupes et objectifs							
Fair Scheduler «FS» [169]	✓				✓		✓	
Capacity Scheduler «CS»[58]	✓				✓		✓	
Deley Scheduler «DS» [170]	✓	✓					✓	✓
Dominant resource fairness «DRF» [63]	✓	✓			✓			✓
Dynamic Proportional Share«DPS» [132]	✓		✓		✓			
Deadline Constraint Scheduler «DCS» [88]	✓		✓					
Extended Deadline Constraint Scheduler «EDCS» [125]			✓					
Locality-aware resource allocation «PURLIEUS» [117]		✓		✓				✓
NATA [159]	✓	✓			✓		✓	
Resource-Aware Scheduling Through Coupling Task Progress «RATCTP» [141]	✓		✓					
Matchmaking scheduling «MS» [72]	✓	✓		✓				
Resource-aware adaptive scheduler «RAS» [124]	✓	✓			✓			
ARIA-SLO-based scheduler «ARIA» [148]			✓					
Center-of-Gravity Reduce Task Scheduling «CoGRS» [68]	✓	✓		✓				✓
Mark Yong et al [168]			✓	✓			✓	
Naduri et al. [114]		✓			✓			
«FLEX» [155]			✓		✓		✓	
Replica-aware map scheduling for mapReduce«MAESTRO» [76]	✓	✓					✓	
Locality-aware reduce task scheduler«LARTS» [69]	✓	✓		✓				
Locality/Fairness-aware key partitioning for mapReduce «LEEN»[75]	✓	✓		✓				

TABLE 3.1 – Listes des politiques d'ordonnancement étudiées leur objectifs (1/2).

Groupes et objectifs

	Implémenté dans Hadoop	Réduction de la date de fin des travaux	Respect de la date de fin imposée par l'utilisateur	Réduction du trafic réseau	Partage équitable des ressources	Amélioration de la consommation énergétique	Algorithmes d'exécution spéculative	Adapté à une grappe hétérogène
Locality-aware reduce scheduling «LARTS» [25]	✓	✓		✓				
Longest approximate time to end«LATE» [171]	✓	✓					✓	✓
Mantri [10]		✓		✓			✓	✓
Base [67]	✓	✓					✓	✓
Tarazu [8]	✓	✓		✓				
MARLA [52]		✓			✓			✓
MapReduce online [36]	✓	✓					✓	
DyScale [162]		✓						✓
Hierarchical mapreduce scheduler for hybrid data centers «HybridMR» [138]	✓	✓				✓		
«MRA++» [11]			✓					✓
Shortest remaining time first policy in shared hadoop clusters«HSRTF» [30]	✓	✓		✓				
Zhou et al. [142]		✓		✓				✓
History based auto-tuning mapreduce«HAT» [26]	✓	✓					✓	✓
Enhanced self-adaptive mapreduce «ESAMR» [140]	✓	✓		✓		✓		✓
CPU and I/O Characteristic Estimation Strategy «CICS» [150]	✓	✓						✓
LsPS [166]	✓	✓			✓			✓
Classification and optimization based scheduler for heterogeneous hadoop«COSHH» [116]	✓	✓				✓		✓
Moseley et al. [113]	✓	✓			✓			
Aboulnaga et al. [4]	✓	✓						
GreenHDFS [150]		✓				✓		✓
EMRSA(1 et 2) [150]		✓				✓		
Bampis et al. [105]		✓				✓		
Chang et al. [23]		✓						✓
Choosy [65]					✓			

TABLE 3.2 – Listes des politiques d'ordonnancement étudiées leur objectifs (2/2).

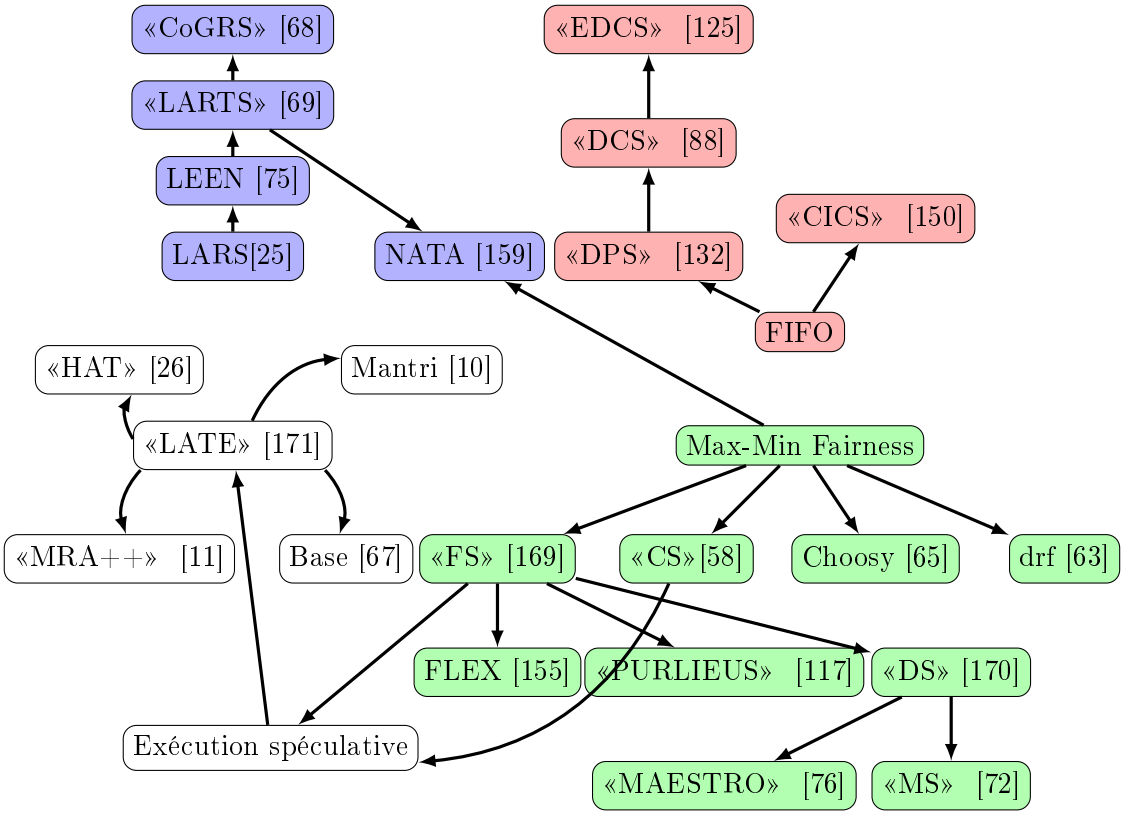


FIGURE 3.5 – Relations d'évolution entre certains algorithmes d'ordonnancement MapReduce

Chapitre 4

Résolution optimale du problème hors-ligne

4.1 Introduction

Dans le chapitre précédent, nous avons présenté des heuristiques, issues de l'état de l'art, utilisées pour optimiser l'exécution des travaux sur une grappe Hadoop. Ces heuristiques, qui dépendent des objectifs à optimiser, sont basées sur la méthode Glouton. La plupart des heuristiques présentées dans la littérature ont été évaluées à l'aide de la simulation.
Dans ce chapitre, nous abordons la version hors ligne. Nous modélisons le problème sous la forme d'un programme linéaire en nombres entiers. Nous évaluons le modèle et nous calculons la solution optimale du problème hors ligne. Cette dernière constitue une borne inférieure de la solution optimale du problème en ligne, ce qui nous permettra d'évaluer les algorithmes en ligne que nous proposons dans le chapitre suivant.
Dans une première étape, nous présentons une première version du modèle dans laquelle nous considérons des tâches unitaires ($p_i = 1$). Nous résolvons, jusqu'à l'optimalité, cette première version avec CPLEX et nous proposons une heuristique de résolution que nous évaluons.
Dans une deuxième étape, nous faisons évoluer la première version pour considérer, dans une deuxième version, des tâches non interruptibles et de durées quelconques. Nous considérons une modélisation plus réaliste de notre problème. Nous résolvons la nouvelle version du modèle jusqu'à l'optimalité. Sur la base de ce modèle, nous étudions la relaxation de contraintes, nous tentons de relâcher des contraintes pour trouver des solutions approchées dans un temps raisonnable et augmenter la taille des instances résolues.
Le présent chapitre est organisé comme suit. Dans la première section nous rappelons le problème d'ordonnancement et les différentes notations utilisées dans ce chapitre. Dans la deuxième, nous présentons la première version du modèle et nous considérons le problème d'ordonnancement dans Hadoop comme un problème de minimisation de la date de fin d'exécution du dernier travail (C_{max}). Nous calculons la solution optimale. Nous clôturons cette section avec la proposition d'une heuristique de résolution.
Dans la troisième section, nous faisons évoluer notre modèle et nous calculons des bornes d'éva-

luations pour celui-ci. Nous considérons des tâches de durées quelconques et nous cherchons à optimiser deux objectifs d'importance égales : *(i)* la minimisation de la somme pondérée des dates de fin d'exécution des travaux ($\sum w_j C_j$) et *(ii)* la minimisation de la date de fin d'exécution du dernier travail (C_{max}).

4.1.1 Formalisation du problème et hypothèses spécifiques

Vu le cadre de la thèse, nous ne pouvons pas dissocier les critères à optimiser des attentes des utilisateurs du système dont nous cherchons à améliorer les performances. Après discussion et suite aux études que nous avons menées avec les membres de l'équipe Ingensi de l'entreprise Cyres. Il apparait raisonnable de considérer les hypothèses suivantes :

— Il faut que les travaux soumis se terminent au plus tôt.

— Il faut trouver un équilibre, entre les différents flux de travail, pour que la durée d'exécution des travaux soit acceptée par tous les utilisateurs.

L'ordonnancement des tâches sur une grappe informatique consiste à affecter les tâches (ou les travaux) sur les machines de la grappe et décider de l'ordre de leurs exécutions. Dans notre grappe, les principales ressources d'exécution des tâches sont la mémoire, les cœurs virtuels, l'espace disque et la bande passante réseaux.
Dans notre modèle, nous considérons une phase préliminaire de préparation, elle consiste en la définition des données d'entrée du modèle. Ces entrées sont des informations concernant les tâches, les machines, le réseau et les blocs de données. L'horizon de temps nécessaire pour l'exécution des tâches n'est pas connu par avance. De ce fait, nous définissons l'horizon de temps en fonction de la durée des tâches et en fonction du degré de parallélisme. La section 4.3.4 présente les informations concernant cette phase et les règles de génération des données du modèle.

4.1.2 Définition du problème d'ordonnancement

On considère une grappe constituée de M machines. Chaque machine M_j est caractérisée par (1) un ensemble de Vcores m_j^s dont m_j^{Sm} Vcores dédiés aux tâches "Map" et m_j^{Sr} Vcores dédiés aux tâches "Reduce", (2) une capacité m_j^{ra} de mémoire (RAM) et une quantité m_j^h de disque dur. On considère un ensemble EW de W travaux de types Map / Reduce et un ensemble de blocs de données noté A^b. Chaque travail possède un poids $Weight_{w'}$ et est composé d'un ensemble $E_{w'}$ de tâches Map et de tâches Reduce.
Ces travaux sont décomposés en un ensemble de N tâches d'indice $i \in [1 \dots N]$. L'indice i est l'identifiant unique d'une tâche dans l'ensemble global des N tâches. Ce dernier est composé de deux sous-ensembles : L^m tâches "Map" et L^r tâches "Reduce", donc $N^m = card(L^m)$ tâches "Map" et $N^r = card(L^r)$ tâches "Reduce" avec $N = N^m + N^r$. Chaque tâche $T_i^{w'}$ est associée à un travail $w' \in EW$ dont la durée d'exécution est notée p_i. On note $b_{i,i'}$ la bande passante réservée pour assurer l'échange des données entre deux tâches $T_i^{w'}$ et $T_{i'}^{w'}$ qui s'exécutent sur deux machines différentes. Pour chaque tâche $T_i^{w'}$, on considère E_i l'ensemble des n_i tâches qui doivent s'exécuter avant la tâche $T_i^{w'}$. Lorsque la tâche $T_i^{w'}$ s'exécute, elle demande une quantité n_i^{ra} de RAM, une quantité n_i^h de disque dur, un Vcore et traite un ensemble B_i de données divisées en

$n_i^b \geqslant 1$ blocs lus sur le système de fichiers. Les notations utilisées pour la description des tâches sont d'ordre générique, nous les utilisons pour la description des tâches "Map" et "Reduce". Par exemple, dans un cas réel, chaque tâche "Map" traite un seul bloc ($n_i^b = 1$) et chaque tâche "Reduce" traite des blocs de données ($n_i^b \geqslant 1$) provenant de plusieurs tâches "Map" [152].
Tous les blocs de données ont la même taille S. On note r_b le nombre de réplications d'un bloc b. D_b est alors l'ensemble des machines qui contiennent le bloc b et bwd la bande passante allouée pour migrer un bloc d'une machine à une autre.
L'architecture de la grappe est modélisée par un graphe $G = (V, E)$ où l'ensemble des nœuds V présente les machines et l'ensemble des arcs E présente les liaisons filaires entre elles. On considère P l'ensemble des chemins entre les machines, P_u est l'ensemble des couples de machines qui utilisent l'arc e_u. Chaque chemin est composé de plusieurs arcs $e_u \in E$ dont la capacité maximale de bande passante b_{max} est identique pour tous les arcs. On note T l'horizon de temps nécessaire pour l'ordonnancement des tâches sur les machines.

4.2 Modélisation mathématique du problème de la minimisation du C_{max}

Dans cette section, nous présentons une première modélisation simplifiée du problème d'ordonnancement dans Hadoop en considérant des travaux de même poids et des tâches de durée unitaire ($p_i = 1$). Nous cherchons à minimiser la date de fin de toutes les tâches, notée C_{max}.

4.2.1 Variables du modèle

Nous utilisons les variables de décisions suivantes :

$$x_{i,s,t}^{j} = \begin{cases} 1 & \text{Si la tâche i est exécutée sur le Vcore s de la machine j sur l'intervalle (t, t+1].} \\ 0 & \text{Sinon} \end{cases} \quad (4.1)$$

$$y_{b,t}^{j,j'} = \begin{cases} 1 & \text{Si le bloc b est sur la machine j sur l'intervalle } (t, t+1] \text{ après une migration de la machine j'.} \\ 0 & \text{Sinon} \end{cases} \quad (4.2)$$

$$u_{b,t}^{j,j'} = \begin{cases} 1 & \text{Si le bloc b est en cours de migration de la machine j' vers la machine j sur l'intervalle (t, t+1].} \\ 0 & \text{Sinon} \end{cases} \quad (4.3)$$

$$z_{l,l',t}^{j,j'} = \begin{cases} 1 & \text{Si la tâche "Map" l' a terminé son exécution à l'instant t sur la machine j' et la tâche l est exécutée sur la machine j après l'instant t.} \\ 0 & \text{Sinon} \end{cases} \quad (4.4)$$

$$C_{max} \in \mathbb{N}^+ : \text{ Date de fin d'exécution de toutes les tâches.} \quad (4.5)$$

4.2.2 Formalisation du modèle

L'étape suivante est la présentation des contraintes du modèle. Nous les regroupons en trois ensembles :

- Les contraintes de ressources : cette classe définit les contraintes qui interdisent aux tâches de dépasser les quantités de ressources existantes sur les machines.
- Les contraintes liées à l'exécution des tâches : leur rôle est *(i)* de définir la date de fin de la dernière tâche de chaque travail, *(ii)* de définir la relation de précédence entre les tâches "Map" et les tâches "Reduce", *(iii)* de définir le nombre de Vcores qu'une tâche peut utiliser au maximum.
- Les contraintes liées à la migration et au transfert des données : cette classe de contraintes définit la relation entre les tâches et les données qu'elles traitent et la migration des données d'une machine vers une autre.

Rappelons que notre objectif dans cette section est la minimisation de la date de fin des travaux :

$$\text{Minimiser } C_{max}$$

La liste des contraintes du modèle est la suivante :

Les contraintes de ressources :

$$\sum_{s=1}^{m_j^s}\sum_{i=1}^{N} n_i^{ra} x_{i,s,t}^j \leq m_j^{ra}. \qquad \forall j = 1...M, \forall t = 1...T \tag{4.6}$$

$$\sum_{i \in L^r} x_{i,s,t}^j \leq 1. \qquad \forall j = 1...M, \forall t = 1...T, \forall s = 1...m_j^{Sr} \tag{4.7}$$

$$\sum_{i \in L^m} x_{i,m_j^{Sr}+s,t}^j \leq 1. \qquad \forall j = 1...M, \forall t = 1...T, \forall s = 1...m_j^{Sm} \tag{4.8}$$

$$\sum_{s=1}^{m_j^s}\sum_{i=1}^{N} n_i^h x_{i,s,t}^j + \sum_{j'=1, j' \neq j}^{M}\sum_{i=1}^{N}\sum_{b \in B_i} S(y_{b,t}^{j,j'} + u_{b,t}^{j,j'}) \leq m_j^h. \qquad \forall j = 1...M, \forall t = 1...T \tag{4.9}$$

$$\sum_{(j,j') \in P_e} \left[bwd \sum_{b \in A^b} u_{b,t}^{j,j'} + \sum_{i \in L^r}\sum_{l \in E_i}\sum_{s=1}^{m_j^s} z_{i,l,t}^{j,j'} * b_{i,l} \right] \leq b_{max}. \quad \forall e \in E, \forall t = 1...T \tag{4.10}$$

$$\sum_{s=1}^{m_j^{Sr}} x_{l,s,t"}^j + \sum_{s=1}^{m_{j'}^{Sm}} x_{l',m_{j'}^{Sr}+s,t'}^j - 1 \leq z_{l,l',t}^{j,j'} \tag{4.11}$$

$$\forall l \in L^r, \forall l' \in E_l, \forall t = 1...T-1, \forall t' = 1...t-1, \forall t" = t...T, \forall j, j' = 1...M, j \neq j'$$

Les contraintes (4.6) supposent qu'à chaque instant, la quantité de mémoire consommée par les tâches qui s'exécutent sur une machine j ne doit pas dépasser la quantité de mémoire existante sur la machine. Les contraintes (4.7) et (4.8) imposent qu'un cœur virtuel ne peut exécuter qu'une seule tâche à la fois. Les contraintes (4.9) contrôlent la quantité du disque dur utilisée par

machine. À chaque instant, la somme de la quantité du disque dur (HDD) utilisée par les tâches et la quantité de disque utilisée pour l'enregistrement et la migration des données ne doivent pas dépasser la quantité de disque disponible sur la machine. Les contraintes (4.10) imposent que les quantités de données transférées à travers le réseau, pour la migration de données et l'échange de données entre les tâches, ne dépassent pas la bande passante disponible sur les arcs. Les contraintes (4.11) permettent la réservation de la bande passante depuis la fin de la tâche "Map" jusqu'à la fin des tâches "Reduce" associées.

Les contraintes liées à l'exécution des tâches :

$$\sum_{t=1}^{T}\sum_{j=1}^{M}\sum_{s=1}^{m_j^s}(t+1)*x_{i,s,t}^{j} \leq C_{max}. \qquad \forall i \in L^r \cup L^m \tag{4.12}$$

$$n_k * x_{k,s,t}^{j} \leq \sum_{u=1}^{M}\sum_{s'=1}^{m_j^{Sm}}\sum_{t'=1}^{t}\sum_{l\in E_k} x_{l,m_u^{Sr}+s',t'}^{u} \tag{4.13}$$
$$\forall k \in L^r, \forall t = 1...T, \forall j = 1...M, \forall s = 1...m_j^{Sr}$$

$$\sum_{j=1}^{M}\sum_{s=1}^{m_j^{Sm}}\sum_{t=1}^{T} x_{i,m_j^{Sr}+s,t}^{j} = 1. \qquad \forall i \in L^m \tag{4.14}$$

$$\sum_{j=1}^{M}\sum_{s=1}^{m_j^{Sr}}\sum_{t=1}^{T} x_{i,s,t}^{j} = 1. \qquad \forall i \in L^r \tag{4.15}$$

$$\sum_{j=1}^{M}\sum_{s=1}^{m_j^{Sm}} x_{i,m_j^{Sr}+s,t}^{j} \leq 1. \qquad \forall i \in L^m, \forall t = 1...T \tag{4.16}$$

$$\sum_{j=1}^{M}\sum_{s=1}^{m_j^{Sr}} x_{i,s,t}^{j} \leq 1. \qquad \forall i \in L^r, \forall t = 1...T \tag{4.17}$$

Les contraintes (4.12) définissent la date de fin d'exécution de la dernière tâche. Les contraintes (4.13) définissent les relations de précédence entre les tâches "Map" et les tâches "Reduce". Les contraintes (4.14) et (4.15) définissent les durées des tâches "Map" et "Reduce". Les contraintes (4.16) et (4.17) imposent qu'à chaque instant, une tâche "Map" ou "Reduce" ne puisse s'exécuter que sur un seul cœur virtuel.

Les contraintes liées à la migration et au transfert des données :

$$y_{b,1}^{j,j'} = 0. \qquad \forall b \in A^b, \forall j = 1...M, \forall j' = 1...M, j' \neq j \tag{4.18}$$

$$y_{b,t}^{j,j} = \begin{cases} 1 & \forall j \in D_b \\ 0 & \forall j \notin D_b \end{cases} \qquad \forall t = 1...T, \forall b \in A^b \tag{4.19}$$

$$\sum_{s=1}^{m_j^{Sm}} x^j_{i,m_j^{Sr}+s,t} \leq \sum_{b\in B_i}\sum_{j'\in D_b} y^{j,j'}_{b,t}. \qquad \forall i \in L^m, \forall j = 1...M, \forall t = 1...T \tag{4.20}$$

$$x^j_{i,m_j^{Sr}+s,t} \leq \sum_{j'\in D_b} y^{j,j'}_{b,t-1} + \sum_{j'\in D_b} u^{j,j'}_{b,t-1} \tag{4.21}$$

$$\forall i \in L^m, \forall b \in B_i, \forall j = 1...M, \forall s = 1...m_j^{Sm}, \forall t = 2...T$$

$$\sum_{j'\in D_b, j'\neq j} y^{j,j'}_{b,t} \leq \sum_{t'=t+1}^{T}\sum_{s=1}^{m_j^{Sm}} x^j_{i,m_j^{Sr}+s,t'}. \qquad \forall i \in L^m, \forall b \in B_i, \forall j = 1...M, \forall t = 1...T \tag{4.22}$$

$$u^{j,j'}_{b,t} \leq 1 - y^{j,j'}_{b,t}. \qquad \forall j = 1...M, \forall j' = 1...M, \forall b \in A^b, \forall t = 1...T \tag{4.23}$$

Les contraintes (4.18) et (4.19) définissent si un bloc de données est enregistré sur une machine ou non. Les contraintes (4.20) et (4.21) imposent qu'une tâche "Map" ne peut s'exécuter sur une machine que si les données, qu'elle traite, y sont présentes. Remarquons que ces contraintes imposent qu'une unité de temps soit suffisante pour la migration d'un bloc de données vers la machine distante contenant la tâche "Map" qui le traitera. Les contraintes (4.22) et (4.23) forcent la suppression d'un bloc, déjà migré, lorsqu'il n'existe plus de tâche qui le traite sur la machine où il a été migré. Les contraintes (4.23) interdisent la migration d'un bloc s'il est déjà présent sur la machine de destination.

4.2.3 Évaluation du modèle

Dans cette section, nous commençons par présenter la méthode de génération des données d'entrée (instances d'entrée) du modèle ensuite nous présentons les résultats de l'évaluation. L'évaluation du modèle portent sur le nombre de variables, de contraintes et sur le temps nécessaire pour le résoudre. Sa résolution est utilisée pour le calcul de la borne inférieure et pour évaluer l'heuristique *LocFirst* que nous proposons dans la section 4.2.4. Rappelons que la solution optimale du problème hors ligne est une borne inférieure de la solution optimale du problème en ligne.

Génération des données d'entrée

Les données d'entrée de notre modèle sont les données liées aux machines, aux tâches, aux blocs de données et à la typologie du réseau.
La configuration des machines est récupérée depuis le site d'Amazon AWS. Le tableau (4.1) affiche les trois configurations de machines utilisées. Dans le tableau (4.1), les colonnes présentent par ordre : la description de la catégorie, le nombre de cœurs (physique) de calcul CPU, la quantité de mémoire RAM, l'espace d'enregistrement sur les disques et la fréquence de calcul de chaque cœur. Nous avons choisi de réserver un cœur (physique) CPU pour les opérations du système d'exploitation de chaque machine (hypothèse raisonnable en pratique). Les deux dernières colonnes du tableau (4.1) présentent la quantité de Vcores existante sur la machine pour exécuter les tâches "Map" et les tâches "Reduce". Nous générons aléatoirement la taille d'un travail parmi les trois catégories présentées dans le tableau (4.2).

TABLE 4.1 – Configuration des machines utilisées pour la génération de données du problème

Catégories	**Nbre de cœurs**	**RAM (Go)**	**SSD (Go)**	**freq de chaque cœur** (en GHZ)	**Bdw (Go)**	**Coeurs virtuels Map**	**Coeurs virtuels Reduce**
c3.2xlarge ($M1$)	8	15	160	2.8 Intel Xeon E5-2680v2	1	5	2
i2.2xlarge ($M2$)	8	61	1600	2.5 Intel Xeon E5-2670v2	1	4	3
r3.xlarge ($M3$)	4	30.5	1600	2.5 Intel Xeon E5-2670v2	2	2	1

TABLE 4.2 – Tailles des travaux utilisés lors de l'évaluation du modèle

Travail	**Nombre de tâches Reduce**	**Nombre de tâches Map**
$N1$	2	3
$N2$	2	6
$N3$	3	9

TABLE 4.3 – Formules utilisées pour la génération des quantités de mémoire et de disque pour chaque tâche

Type du travail	$n_i^r = n_i^b * S * XY$	$n_i^h = (n_i^b * S * WZ)/1024$
(1) CPU intensive	$XY \in [0.3, 0.6]$	$WZ \in [13, 26]$
(2) RAM intensive	$XY \in [0.4, 0.8]$	$WZ \in [30, 46]$
(3) I/O intensive	$XY \in [0.6, 1]$	$WZ \in [46, 76]$

TABLE 4.4 – Scénarios de génération des données d'entrée (nombre de tâches, machines et blocs de données)

	N1	N2	N3	M1	M2	M3	Nbre de Blocs	Nbre total de tâches	Nbre total de machines	Nbre de cœurs
Scénario 1	1	1	1	0	1	1	10	25	2	10
Scénario 2	1	1	1	1	1	1	10	25	3	17
Scénario 3	6	0	0	0	1	1	10	30	2	10
Scénario 4	6	0	0	1	1	1	10	30	3	17
Scénario 5	3	3	0	0	0	2	10	39	2	6
Scénario 6	3	3	0	0	1	2	10	39	3	13
Scénario 7	1	5	0	0	1	1	10	45	2	10
Scénario 8	1	5	0	1	1	1	10	45	3	17
Scénario 9	3	0	3	0	2	0	10	51	2	14
Scénario 10	3	0	3	0	2	1	10	51	3	17
Scénario 11	3	0	3	1	2	1	10	51	4	24

De la même manière, nous générons aléatoirement le type d'un travail parmi les trois catégories suivantes : (*i*) des travaux nommés "CPU-intensive" (N1), qui utilisent plus la ressource de calcul, (*ii*) des travaux nommés "RAM-intensive" (N2) qui ont une forte utilisation de la mémoire RAM, (*iii*) des travaux nommés "I/O-intensive" (N3) qui ont une forte utilisation du HDD.
Rappelons que nous générons des tâches de durée unitaire. Nous générons les quantités de mémoire, disque, bande passante de migration et de transfert de données de façon à différencier les quantités de ressources associées à chaque type de travaux. En fonction du type du travail auquel la tâche appartient, nous générons la quantité mémoire et disque qu'elle utilise. Nous utilisons les formules décrites dans le tableau (4.3) pour générer la quantité de mémoire RAM (n_i^{ra}) et la quantité d'espace disque (n_i^h) nécessaires pour chaque tâche. Ces deux valeurs dépendent de la taille d'un bloc de données, du nombre de blocs qu'une tâche traite et de deux nombres XY et WZ générés aléatoirement. XY et WZ garantissent que les tâches associées à un travail "I/O-intensive" récupèrent plus de mémoire et de disque que les autres travaux.
Les travaux du type "I/O-intensive" sont caractérisés par des tâches "Map" qui utilisent et génèrent des grandes quantités de données. De ce fait, le transfert de leurs données nécessite une bande passante plus élevé. Nous avons pris le choix que : *(i)* les quantités de données d'entrée et de sortie d'une tâche sont égales. *(ii)* La quantité de données d'entrée et de sortie d'une tâche "Reduce" est égale à la taille d'un bloc de données. *(iii)* La taille S des blocs est égale à 64 Mb. *(iv)* La bande passante nécessaire à la migration de chaque bloc (*bwd*) est égale à S pour les travaux de type "I/O *intensive*" et de $(S * 0.3)$ pour les travaux des autres types. *(v)* Le temps nécessaire à la migration d'un bloc de données est une unité de temps.
Si deux tâches "Map" et "Reduce", d'un même travail, sont affectées sur deux machines différentes, une quantité de bande passante est allouée pour assurer le transfert des données entre elles. Nous décidons de générer aléatoirement *(i)* la quantité de bande passante allouée pour assurer le transfert de données de sortie des tâches "Map" entre 30 et 100% de la taille de la donnée d'entrée de la tâche "Map". *(ii)* Le choix de l'emplacement des blocs de données (et leurs réplications) sur les machines et *(iii)* l'affectation des blocs aux tâches.
Nous modélisons le réseau sous la forme d'un arbre binaire dont les machines sont les feuilles de l'arbre et les nœuds intermédiaires sont des routeurs informatiques. Le niveau maximal de profondeur de l'arbre est 2, chaque routeur lie deux machines, nous générons les chemins entre les machines et les arcs qui les composent. Vu que l'architecture réseau forme un arbre binaire, il existe toujours un seul chemin entre deux machines du réseau. Les arcs présentent des connexions filaires et possèdent une capacité de transfert maximale de données définie par une constante. Nous générons pour chaque scénario 10 instances dont les différences entre elles sont les quantités de ressources générées pour les différentes entités de chaque instance.
L'horizon de temps T est calculé de la façon suivante :

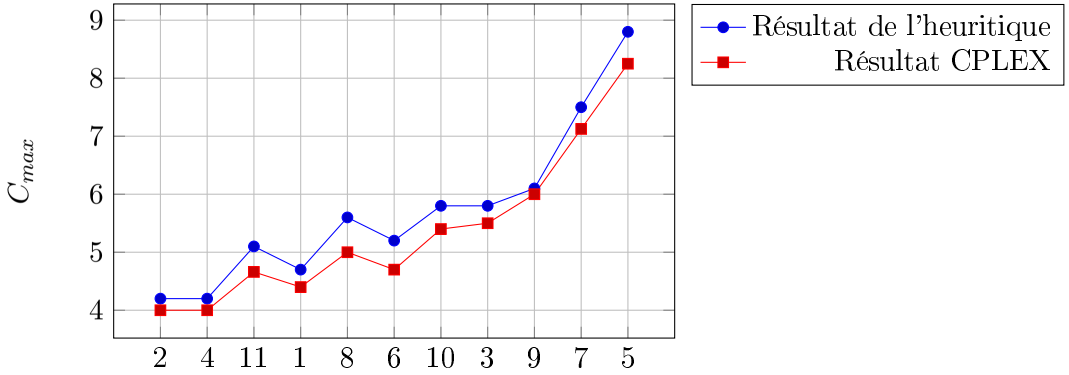
$$T = \left[\left(\frac{\sum_{i \in L^r} p_i}{\sum_{j=0}^{M} m_j^{Sr}} + \frac{\sum_{i \in L^m} p_i}{\sum_{j=0}^{M} m_j^{Sm}} \right) * 0.75 + \left(\frac{\sum_{i=1}^{N} n_i^{ra} * p_i}{\sum_{j=1}^{M} m_j^{ra}} \right) * 0.25 \right] + 2 * \underset{l \in L^r \cup L^m}{Max} (p_l) + 1 \quad (4.24)$$

La première partie à gauche de l'équation fournit le temps nécessaire pour exécuter les tâches comme si les Vcores sont les seules ressources à gérer.

TABLE 4.5 – Résultats de l'évaluation du modèle (10 exécutions par scénario)

	Nb inst. infaisables	Nb inst. à l'opt résolus	Lim Mémoire	Lim Temps	N_{min}	N_{moy}	N_{max}	T_{min}	T_{moy}	T_{max}	C_{max}^{moy}
Scénarios 1	0	10	0	0	160	120.7	259	0	0.7	1	4.4
Scénarios 2	0	10	0	0	107	711	1357	0	1.2	2	4
Scénarios 3	0	10	0	0	280	8257.2	34848	1	7.7	31	4.7
Scénarios 4	0	10	0	0	161	4667.4	22137	1	4.4	16	4
Scénarios 5	0	4	5	1	1481	31745.5	112635	4	83.75	303	8.25
Scénarios 6	0	10	0	0	17498	85037.5	311926	34	259	1401	5.4
Scénarios 7	0	8	2	0	6046	54182.3	174241	8	517.25	1697	7.12
Scénarios 8	0	3	5	2	17775	24641	34488	77	1251.66	1747	5
Scénarios 9	0	4	5	1	35399	49466.7	6135	782	1199	1739	6
Scénarios 10	0	6	3	1	16210	39698.8	74486	188	447.66	1807	5.5
Scénarios 11	0	6	4	0	2553	23839	52471	84	524	1124	4.66

FIGURE 4.1 – Moyenne des consommation des ressources par rapport au temps.



La deuxième partie à gauche calcule le temps nécessaire comme si la mémoire est la seule ressource à partager. Lorsque le nombre de tâches soumis est inférieur au nombre de Vcores, la somme des deux premières parties à gauche de l'équation sera inférieure à 1. Nous ajoutons une constante égale à $2 * \underset{l \in L^r \cup L^m}{Max}(p_l)$ pour garantir que face à cette situation, ces tâches auront le temps nécessaire pour s'exécuter. La constante 1 permet aux tâches "Map" de transférer les blocs de données qu'ils traitent.

Environnement de test

Les différentes simulations sont effectuées sur un PC contenant un processeur Intel(R) Core (TM) i5-3360M avec 4 cœurs, une puissance de 2.8 GHz par cœur et 4 Go de mémoire RAM. On utilise le solveur CPLEX version 12.4 pour évaluer le modèle présenté. CPLEX est configuré (*i*) pour utiliser 2 Go de mémoire RAM, (*ii*) une limite de temps de calcul égale à 1800 secondes et (*iii*) pour effectuer les traitements en mode parallèle (4 threads).

Expérimentations et résultats de l'évaluation du modèle

Les différents scénarios utilisés pour l'évaluation du modèle sont présentés dans le tableau (4.4). Les colonnes 2, 3 et 4 présentent le nombre de travaux de chaque taille, les colonnes 5, 6 et 7, le nombre de machines de chaque type et la 8^{eme} colonne désigne le nombre de blocs de données par scénario. Les trois dernières colonnes présentent le nombre total de tâches, de machines et de Vcores par scénario.
Pour chaque scénario, nous générons 10 instances, la différence entre elles est la quantité des ressources affectées aux tâches. Nous récupérons la moyenne des résultats optimaux obtenus suite aux 10 exécutions par scénario.
Les résultats d'évaluation du modèle sont affichés dans le tableau (4.5). La deuxième colonne désigne le nombre d'instances que le solveur n'a pas pu résoudre, la troisième présente le nombre d'instances résolues jusqu'à l'optimalité, la quatrième présente le nombre d'instances pour lesquelles CPLEX s'arrête parce qu'il a atteint la limite mémoire et la cinquième présente le nombre d'instances pour lesquelles CPLEX s'arrête parce qu'il a atteint la limite de temps fixé. Les colonnes 6, 7 et 8 présentent le nombre (minimal, moyen et maximal) de nœuds explorés par CPLEX durant la résolution du problème. Les colonnes 9, 10 et 11 présentent le temps d'utilisation du processeur pour trouver les différentes solutions. La douzième colonne présente la moyenne des dates de fin de l'ordonnancement de toutes les tâches par scénarios pour les instances possédant des solutions optimales.
Suite à cette évaluation du modèle, nous proposons une heuristique de résolution de la version hors ligne du problème. Nous utilisons les résultats obtenus lors de l'évaluation du modèle pour évaluer l'heuristique que nous proposons.

4.2.4 Présentation et évaluation de l'heuristique de résolution

Dans cette partie, nous présentons une heuristique de résolution pour la version hors ligne du problème. L'adaptation de l'heuristique à la version en ligne du problème est traité dans le

chapitre suivant.
L'heuristique de résolution est nommée "LocFirst" (algorithme 2), elle est basée sur la méthode Glouton. Elle prend en entrée : la liste des tâches, la liste des machines et la liste des blocs.

Algorithme 2 : Algorithme associé à l'heuristique LocFirst

Données :

1 $Lmachines \leftarrow \{m\}, \forall m = 1...M$ `/* Liste des machines */`
2 $LdataBloc \leftarrow \{b\}, \forall b \in A^b$ `/* Liste des blocs */`
3 $Ltasks \leftarrow \{i\}, \forall i = 1...N$ `/* Liste des tâches */`

Résultat : Pour chaque machine, l'algorithme fournit l'ensemble des tâches qu'elle exécute et l'ordre de leurs exécutions

4 `PriorityDefinition(`$Ltasks$`)` ; `/* Définition des priorités des tâches */`
5 $MoyenDateFin \leftarrow \emptyset$;
6 **pour chaque** *tâche* $i \in Ltasks$ **faire**
  `/* ListMachLoc : ensemble des machines sur lesquelles` i `peut s'exécuter en respectant la localité */`
7 $ListMachLoc \leftarrow$ `ReturnListMachineLoc(`$i, LdataBloc, Lmachines, TaskType$`)`
8 **pour** $m' \in ListMachLoc$ **faire**
9 **si** m' *contient les ressources nécessaires pour exécuter* i *avant* $MoyenDateFin$ **alors**
10 `ExecuteTask(`i,m', $MoyenDateFin$`)` ;
11 $Ltasks \leftarrow Ltasks \setminus \{i\}$;
12 $MoyenDateFin \leftarrow$ Calculer la moyenne des dates de fin d'exécution des dernières tâches "Map" sur les machines ;
13 **pour chaque** *tâche* $i \in Ltasks$ **faire**
  `/* ListMach : ensemble des machines sur lesquelles` i `peut s'exécuter */`
14 $ListMach \leftarrow$ `ReturnListMachineDis(`$Lmachines, i, MoyenDateFin$`)` ;
15 $m' \leftarrow$ la machine de ListMach qui contient de plus de ressource libre(Vcores, RAM) ;
16 `ExecuteTask(`$i, m', MoyenDateFin$`)`;
17 $Ltasks \leftarrow Ltasks \setminus \{i\}$;
18 $MoyenDateFin \leftarrow$ Calculer la moyenne des dates de fin d'exécution des dernières tâches "Reduce" sur les machines ;

Elle est constituée de deux niveaux : lorsqu'on a des ressources libres, on commence par ordonnancer les tâches sur les machines contenant les données qu'elles traitent. Ensuite, on ordonnance les tâches, qui ne peuvent s'exécuter en respectant le critère de localité, sur les ressources restantes sans le respect de ce critère. On commence par trier la liste des tâches (ligne 4) en fonction des ressources (selon l'ordre lexicographique entre RAM et HDD) qu'elles consomment.

Algorithme 3 : Algorithme associé à la fonction "ReturnListMachineLoc"

Données :

1 $Lmachines \leftarrow \{m\}, \forall m = 1...M$ `/* Liste des machines */`

2 $LdataBloc \leftarrow \{b\}, \forall b \in A^b$ `/* Liste des blocs et leurs propriétés */`

3 $TaskType \leftarrow \{t\}, t \in \{M, R\}$ `/* Type de la tâche à ordonnancer */`

4 $Task$ `/* Tâche à ordonnancer */`

Résultat : Retourner la liste des machines qui peuvent exécuter la tâche en respectant la localité

5 $Lmach \leftarrow \emptyset$

6 **si** *Task est une tâche "Map"* **alors**

7 | **pour** $c1 \in [1 \dots n^b_{Task}]$ **faire**

8 | | $Lmach \leftarrow Lmach \cup D_b(LdataBloc[c1])$;

9 **si** *Task est une tâche "Reduce"* **alors**

10 | **pour** $c2 \in E_i$ **faire**

11 | | $Lmach \leftarrow Lmach \cup$ la machine qui a exécuté la tâche Map $c2$;

12 `Return` $(Lmach)$

Algorithme 4 : Algorithme associé à la fonction "ReturnListMachineDis"

Données :

1 $Lmachines \leftarrow \{m\}, \forall m = 1...M$ `/* Liste des machines */`

2 $Task$ `/* Tâche à ordonnancer */`

3 $MoyenDateFin$ `/* Date limite pour la disponibilité des ressources */`

Résultat : Retourner une machine qui peut exécuter $Task$

4 $Lmach \leftarrow \emptyset$

5 **pour** $c1 \in Lmachines$ **faire**

6 | **si** *(c1 possède les ressources pour exécuter Task avant MoyenDateFin)*

7 | *et (|Lmach|<4)* **alors**

8 | | $Lmach \leftarrow Lmach \cup c1$;

9 $TempMach \leftarrow Lmach[0]$;

10 **pour** $Con \in Lmach$ **faire**

11 | **si** *(Con se libère avant TempMach)* **alors**

12 | | $TempMach \leftarrow Con$;

13 `Return` $(TempMach)$;

Si deux tâches consomment la même quantité de mémoire, on compare par rapport à l'espace disque qu'elles utilisent. Pour le premier niveau (ligne 6), pour chaque tâche i de la liste "Ltasks", on récupère la liste des machines contenant les données qu'elle traite et leurs réplications (ligne

7) et qui peuvent commencer l'exécution de i avant $MoyenDateFin$
Cette dernière est la moyenne des dates de fin de la dernière tâche sur les machines. La liste "$ListMachLoc$" contient les machines sur lesquelles la tâche i peut s'exécuter en respectant la localité. Pour les tâches "Map", "$ListMachLoc$" contiendra la liste des machines contenant les données et leurs réplications. Pour les tâches "Reduce", "$ListMachLoc$" contiendra la liste des machines qui ont exécuté les tâches "Map" qui précèdent la tâche "Reduce" courante. L'affectation des tâches aux machines s'effectue à l'aide des deux fonctions ReturnListMachineLoc() et ReturnListMachineDis(). L'exécution des tâches sur les machines est gérée par la fonction ExecuteTask().
L'algorithme 3 présente les instructions de la fonction ReturnListMachineLoc(). Si la tâche est une tâche "Map", la première condition est vérifiée et la fonction retourne les machines qui contiennent les données. Si la tâche est un "Reduce", elle retourne la liste des machines qui ont exécuté des tâches "Map" qui la précèdent.
L'algorithme 4 retourne la machine qui possède les ressources pour exécuter i. L'idée pour les tâches "Map" est qu'un bloc est répliqué seulement sur trois machines, donc, il nous suffit de récupérer parmi ces machines celle qui contient les ressources nécessaires pour exécuter i avant $MoyenDateFin$. S'il n'y a pas de machine, l'algorithme ne retourne rien.
L'algorithme 5 est associé à la fonction ExecuteTask(). Pour chaque tâche "Map" (resp. "Reduce"), nous cherchons le premier Vcore libre avant $MoyenDateFin$. Dès qu'on le trouve, la tâche $Task$ est ordonnancée. La variable $TempsDates$ calcule la date de début d'ordonnancement de $Task$.
Dans l'algorithme 5, les informations concernant les tâches et machines sont formulées dans des structures qui contiennent les informations du tableau 1.2 et les informations suivantes :

— Pour les tâches :

  * Le champs "$Task.machine$" désigne la machine sur laquelle la tâche $Task$ s'exécute.

  * Le champs "$Task.Vcores$" désigne le Vcore sur lequel $Task$ est exécuté.

  * Le champs "$Task.start$" désigne la date du début d'exécution de $Task$.

  * Le champs "$Task.End$" désigne la date de fin d'exécution de $Task$.

— Pour les machines : le champs "$Mach.LastDateofVcores[s]$" désigne la date de fin d'exécution de la dernière tâche sur le Vcore "s" de la machine "$Mach$".

La figure 4.1 est une comparaison des valeurs optimales, obtenues lors de l'évaluation du modèle, et des valeurs obtenues suite à l'évaluation de l'heuristique. Nous avons généré les données d'entrée selon la même méthode que celle décrite dans la section 4.3.4. Nous remarquons que l'heuristique LocFirst fournit des résultats qui ont des des gaps constants à l'optimalité ($\leq 7\%$). Ce gaps est calculé par rapport à la solution optimale obtenu lors de l'évaluation du modèle.

Algorithme 5 : Algorithme associé à la fonction "ExecuteTask"

Données :

1 $Mach, \quad \forall Mach \in 1...M$ `/* Machine sur laquelle la tâche doit être ordonnancée */`

2 $Task$ `/* Tâche à ordonnancer */`

3 $MoyenDateFin$ `/* Moyenne des dates de fin des tâches les machines */`

Résultat : La tâche $Task$ est ordonnancée sur un Vcore de la machine $Mach$

4 $TempVcores \leftarrow \varnothing$;
5 **si** *Task est une tâche Map* **alors**
6 **pour** *i de* $m^{Sr}_{Mach.num}$ *à* $m^{s}_{Mach.num}$ **faire**
7 **si** *le Vcore i est libre avant MoyenDateFin* **alors**
8 $TempVcores = i$;
9 $TempsDates =$ l'instant quant i possède les ressources pour exécuter $Task$;
10 **si** *Task est une tâche "Reduce"* **alors**
11 $FirstDate \leftarrow$ la date de fin d'exécution de la dernière tâche "Map" associée à $Task$;
12 **pour** *i de* 0 *à* $m^{Sr}_{Mach.num}$ **faire**
13 **si** *i est libre avant MoyenDateFin et après FirstDate* **alors**
14 $TempVcores = i$;
15 $TempsDates =$ l'instant quant i possède les ressources pour exécuter $Task$;
16 $Task.machine = Mach.num$;
17 $Task.Vcores = TempVcores$;
18 $Task.start = TempsDates$;
19 $Task.End = Task.start + p(Task.num)$;
20 $Mach.LastDateofVcores[TempVcores] = Task.End$;
21 `UpdateRessourceOnMachine ()` ;

Analyse de la complexité de l'heuritique

Théorème : L'algorithme "*LocFirst*" a une complexité temporelle de l'ordre de $\mathcal{O}((N^2 + NM) * \max_{\forall m=1...M} m^s_m)$.
N est le nombre total de tâches, M est le nombre total de machines et m^s_m est le nombre de Vcores de la machine m.

Calcul :
L'algorithme 3 est constitué de deux conditions $C1$ et $C2$ pour la vérification du type des tâches ("Map" ou "Reduce"). La vérification du type de la tâche est de complexité $\mathcal{O}(1)$. On suppose

que $f1$ et $f2$ sont les complexités associées à la vérification des conditions $C1$ et $C2$.

— si la tâche est une tâche "Map" ($C1$), on récupère pour chaque bloc, que la tâche $Task$ traite, l'emplacement des répliquas. La boucle "*pour*" exécutée dans la première condition est en $f_3 = \mathcal{O}(n^b_{Task} * r_b)$. Or, une tâche "Map" effectue des traitements sur un seul bloc de données. En plus, le nombre de répliquas d'un bloc est égal à la constante d'entrée r_b. De ce fait, la boucle "*pour*" est de complexité $f_3 = \mathcal{O}(n^b_{Task} * r_b) = \mathcal{O}(1 * r_b) = \mathcal{O}(r_b)$.

— si la tâche est une tâche "Reduce" ($C2$), la boucle "*pour*" consiste à chercher les machines sur lesquelles les tâches "Map", qui précèdent $Task$, sont exécutées. Elle est de complexité $f_4 = \mathcal{O}(N^m)$.

Rappelons que N^m est le nombre de tâches "Map" soumises.
On note la complexité totale de la première condition $f5$ ($= \mathcal{O}(max(f_1, f_3)) = \mathcal{O}(max(1, r_b)) = \mathcal{O}(r_b)$) et la complexité totale de la deuxième condition f_6 ($= \mathcal{O}(max(f_2, f_4)) = \mathcal{O}(max(1, N^m)) = \mathcal{O}(N^m)$). La fonction "*ReturnListMachineLoc*", associée à l'algorithme 3, est de complexité $F1 = \mathcal{O}(max(f_5, f_6)) = \mathcal{O}(max(r_b, N^m)) = \mathcal{O}(N^m) = \mathcal{O}(N)$.

La fonction "*ReturnListMachineDis*", associée à l'algorithme 4, est basée sur une boucles "*pour*". Dès qu'on trouve 3 machines libres qui peuvent exécuter $Task$, on retourne le résultat. La fonction "*ReturnListMachineDis*" est de complexité $F2 = \mathcal{O}(M)$.

La fonction "*ExecuteTask*" parcourt les Vcores pour chercher l'emplacement où $Task$ sera ordonnancée. "*ExecuteTask*" est de complexité $\max\limits_{\forall m=1...M} m^s_m$ avec m^s_m est le nombre de Vcores dans m.

Pour l'algorithme 2 :

1. la fonction de trie des tâches est de complexité $F3 = \mathcal{O}(N \log N)$.
2. La complexité associée à la première boucle "*pour*" est $\mathcal{O}(N^2 * \max\limits_{\forall m=1...M} m^s_m)$:
   (a) La complexité de la fonction "*ReturnListMachineLoc*" est $\mathcal{O}(N^m)$.
   (b) La boucle "*pour*" cherche : *(i)* si la tâche est une tâche "Map", elle effectue une recherche dans une liste de r_b machines. *(ii)* Si la tâche est une tâche "Reduce", elle effectue une recherche parmi n_i machine ($n_i < N^m$). Donc, la complexité de cette boucle est de l'ordre de $\mathcal{O}(N^m) = \mathcal{O}(max(N^m, r_b))$.
   (c) La fonction "*ExecuteTask*" est en $\max\limits_{\forall m=1...M} m^s_m$.
3. La complexité associée à la deuxième boucle "*pour*" est $\mathcal{O}(N * M * \max\limits_{\forall m=1...M} m^s_m)$:
   (a) La fonction "*ReturnListMachineDis*" est de complexité $F2 = \mathcal{O}(M)$.
   (b) La boucle "*pour*" imbriquée est de complexité $F5 = \mathcal{O}(3)$ puisque la longueur de la liste $ListMach$ est 3.
   (c) La fonction "*ExecuteTask*" est en $\max\limits_{\forall m=1...M} m^s_j(m)$.

En résumé, l'algorithme 2 est de complexité $\mathcal{O}(N \log N + N^2 * \max_{\forall m=1...M} m_m^s + N*M* \max_{\forall m=1...M} m_m^s) = \mathcal{O}((N^2 + NM)V)$; avec V le nombre maximal de Vcores sur une machine du réseau.

Lors de l'analyse des résultats fournis par CPLEX et par l'heuristique, nous remarquons que certains travaux commencent leur exécution très en retard par rapport à leur date de soumission. Cela est dû au fait que le critère de minimisation du C_{max} ne réduit pas le temps d'attente moyen des travaux. Nous décidons dans la section suivante de faire évoluer notre modèle en considérant des tâches non interruptibles et de durée quelconque. Nous changeons, en plus, le critère à optimiser.

4.3 Modélisation mathématique du problème pour la minimisation de $\sum w_j C_j + C_{max}$

Du point de vu de l'utilisateur, il est important que : *(i)* la durée moyenne d'attente des travaux qu'il soumet soit la plus petite que possible, *(ii)* ces travaux se terminent au plus tôt et *(iii)* que ces travaux soumis soient prioritaires. De ce fait, nous décidons d'associer à chaque utilisateur, identifié dans le système, un poids, que l'administrateur du système peut définir. L'utilisateur influence l'ordonnancement des travaux à travers le poids qu'on lui associe. Ce poids est hérité par les travaux qu'il soumet.
Dans cette partie, nous faisons évoluer la première version du modèle pour considérer des tâches non interruptibles et de durée quelconque. Notre objectif est de minimiser la somme des deux critères : *(i)* la date de fin du dernier travail et *(ii)* la somme pondérée des dates de fin des travaux. Cette fonction objectif nous permet de trouver un équilibre entre les deux critères qui la composent.
Dans la première partie de cette section, nous présentons une évolution de notre modèle mathématique, nous résolvons le modèle obtenu jusqu'à l'optimalité pour calculer une borne inférieure. La borne calculée est utilisée pour évaluer les heuristiques que nous proposons dans le chapitre 5. Dans une deuxième partie de cette section, nous testons trois relaxations. Nous pensons que ces relaxations peuvent être utilisées pour augmenter la taille des instances que CPLEX résoudra et pour réduire le temps de calcul de la solution. En utilisant ces relaxations, nous acceptons le calcul de solutions approchées pour résoudre des instances plus grandes en un temps limité.

4.3.1 Paramètres et Variables

Les variables du modèle sont composées de deux types, les variables de décision et les variables auxiliaires. Les variables de décisions sont utilisées pour exprimer les contraintes du modèle tandis que les variables auxiliaires sont utilisées pour suivre la consommation des ressources durant l'ordonnancement.
Nous considérons comme variables de décision les variables (4.1), (4.2), (4.3), (4.4), (4.5) de la section 4.2.1 et la variable (4.25) définie ci-dessous.

$$C_{w'} \in \mathbb{N}^+, \forall w' = 1...W : \text{ Date de fin d'exécution de la dernière tâche du travail w'.} \quad (4.25)$$

4.3.2 Formalisation du modèle

Nous présentons dans cette section une extension du modèle présenté dans la section 4.2.2. Les nouvelles contraintes que nous ajoutons gèrent les aspects suivants : *(i)* la durée des tâches est quelconque, *(ii)* les tâches ne sont pas interruptibles. Dans cette section, nous présentons seulement les contraintes qui sont mises à jour ou les nouvelles contraintes. Pour plus d'information concernant les contraintes non présentées dans cette section, vous pouvez vous référer à la section 4.2.2.
Dans cette section, nous choisissons de regrouper les contraintes en quatre familles :

— Les contraintes de ressources : ce sont les contraintes (4.6), (4.7), (4.8), (4.9), (4.10) et (4.11). Elles sont présentées dans la section 4.2.2.

— Les contraintes liées à l'exécution des tâches : ce sont les contraintes (4.12), (4.14), (4.15), (4.16), (4.17) présentées dans la section 4.2.2. Les contraintes (4.13) qui gèrent la gestion de la relation de précédence entre les tâches sont remplacées par les contraintes (4.29).
Les contraintes (4.26) servent à définir la date de fin de la dernière tâche de chaque travail.
Les contraintes (4.27) et (4.28) servent à définir la durée des tâches.
Les contraintes (4.30), (4.31), (4.32) et (4.33) gèrent l'aspect non intérruptible lors de l'exécution des tâches ; c'est-à-dire, elles interdisent l'interruption des tâches. Les contraintes (4.31) et (4.33) sont spécifiques aux tâches qui commencent au premier instant de l'horizon de temps. Les contraintes (4.30) et (4.32) assurent que les tâches qui commencent à partir du deuxième instant ne soient pas interrompues.

— Les contraintes liées à la migration et au transfert des données : ce sont les contraintes (4.18), (4.19), (4.20), (4.21), (4.22) et (4.23) présentées dans la section 4.2.2.

$$\sum_{j=1}^{M}\sum_{s=1}^{m_j^{sr}}(t+1)x_{i,s,t}^{j} \leq C_{w'}. \quad \forall w\prime = 1...W, \quad \forall t = 1...T, \quad \forall i \in L^r \cup E_{w'} \tag{4.26}$$

$$\sum_{j=1}^{M}\sum_{s=1}^{m_j^{Sm}}\sum_{t=1}^{T} x_{i,m_j^{Sr}+s,t}^{j} = p_i \qquad \forall i \in L^m \tag{4.27}$$

$$\sum_{j=1}^{M}\sum_{s=1}^{m_j^{Sr}}\sum_{t=1}^{T} x_{i,s,t}^{j} = p_i. \qquad \forall i \in L^r \tag{4.28}$$

$$\left(\sum_{l\in E_k} p_l\right) * x_{k,s,t}^{j} \leq \sum_{u=1}^{M}\sum_{s'=1}^{m_j^{Sm}}\sum_{t'=1}^{t}\sum_{l\in E_k} x_{l,m_u^{Sr}+s',t'}^{u} \tag{4.29}$$
$$\forall k \in L^r, \forall t = 1...T, \forall j = 1...M, \forall s = 1...m_j^{Sr}$$

$$p_i - (1 - x^j_{i,m_j^{S_r}+s,t} + x^j_{i,m_j^{S_r}+s,t-1}) * HV \leq \sum_{t'=t}^{t+p_i} x^j_{i,m_j^{S_r}+s,t'} \tag{4.30}$$

$$\forall i \in L^m, \forall t = 2...T - p_i, \forall j = 1...M, \forall s = 1...m_j^{Sm}, (HV : \text{high value})$$

$$p_i - (1 - x^j_{i,m_j^{S_r}+s,1}) * HV \leq \sum_{t'=t}^{t+p_i} x^j_{i,m_j^{S_r}+s,t'} \tag{4.31}$$

$$\forall i \in L^m, \forall t = 1...T - p_i, \forall j = 1...M, \forall s = 1...m_j^{Sm}, (HV : \text{high value})$$

$$p_i - (1 - x^j_{i,s,t} + x^j_{i,s,t-1}) * HV \leq \sum_{t'=t}^{t+p_i} x^j_{i,s,t'} \tag{4.32}$$

$$\forall i \in L^r, \forall t = 2...T - p_i, \forall j = 1...M, \forall s = 1...m_j^{Sm}, (HV : \text{high value})$$

$$p_i - (1 - x^j_{i,s,1}) * HV \leq \sum_{t'=t}^{t+p_i} x^j_{i,s,t'} \tag{4.33}$$

$$\forall i \in L^r, \forall t = 1...T - p_i, \forall j = 1...M, \forall s = 1...m_j^{Sm}, (HV : \text{high value})$$

4.3.3 Critères de performance

Afin de réduire le temps d'attente des travaux en fonction de leurs priorités, nous associons à chaque utilisateur un poids qui sera hérité par les travaux qu'il soumet. Les travaux les plus prioritaires doivent avoir plus de privilèges sans que les moins privilégiés ne soient arrêtés ou pénalisés. De ce fait, nous cherchons dans nos travaux à minimiser la somme des dates de fin pondérée des travaux et la date de fin de tous les travaux. Notre fonction objectif est :

$$\text{Minimiser} \sum_{w'=1}^{W} Weight_{w'} C_{w'} + C_{max} \tag{4.34}$$

Après la modélisation mathématique du problème, l'étape suivante focalise sur son évaluation.

4.3.4 Évaluation du modèle

L'évaluation du modèle se base sur la même méthode utisée dans la section 4.2.3. Elle se compose de deux étapes : *(i)* le pré-traitement est une étape qui consiste à générer les données d'entrées du modèle et *(ii)* la deuxième étape est la résolution du modèle.

Génération des données d'entrées

Les données d'entrée de notre modèle sont les données liées aux machines, aux travaux, aux tâches, aux blocs de données et à la typologie du réseau. La génération des données est décrite dans la section 4.3.4. Dans cette section, nous considérons des tâches de durée quelconque. Afin de réduire la quantité des variables générées lors de la résolution du modèle et pour limiter le nombre de fois où le solveur CPLEX n'arrive pas à résoudre des instances à cause de sa capacité réduite de mémoire, nous générons aléatoirement les durées d'exécution des tâches ($p_i \in [1, 2, 3]$).

TABLE 4.6 – Scénarios de la génération des données d'entrée lors de l'évaluation de la deuxième version du modèle mathématique. (nombre de tâches, machines et blocs de données)

	N1	N2	N3	M1	M2	M3	Nbre de Blocs	Nbre total des tâches	Nbre total de machines	Nbre de Vcores
Scénario 1	3	0	0	0	1	1	10	15	2	10
Scénario 2	3	1	0	1	1	1	10	23	3	17
Scénario 3	3	1	0	1	1	2	10	23	4	20
Scénario 4	0	3	0	0	0	2	5	24	2	6
Scénario 5	4	1	0	0	1	1	10	28	2	10
Scénario 6	1	3	0	1	1	1	10	29	3	17
Scénario 7	0	0	3	0	2	1	10	36	3	17
Scénario 8	0	1	3	0	2	0	10	44	2	14
Scénario 9	0	1	3	0	2	2	10	44	4	20
Scénario 10	0	1	3	0	2	1	10	44	3	17

TABLE 4.7 – Résultats des calculs obtenus suite à la résolution du modèle mathématique dans sa version 2 (10 instances par scénario)

	Nb inst. non résolus	Nb inst. résolus	Lim Mémoire	Lim Temps	N_{min}	N_{moy}	N_{max}	T_{min}	T_{moy}	T_{max}	Obj_{moy}	C_{max}^{moy}
Scénario 1	0	10	0	0	6	252.5	702	6	7	8	50.3	7.2
Scénario 2	0	10	0	0	688	3873.4	7015	6	51.5	161	76.1	7.0
Scénario 3	0	10	0	0	168	5050.9	17550	6	111.7	691	71.8	7.3
Scénario 4	0	10	0	0	6028	19157	35560	65	133.7	208	129.1	14.9
Scénario 5	0	8	1	1	4748	26843	44155	51	1553.8	3601	115.0	9.2
Scénario 6	0	9	1	0	4653	19648	50257	70	1098	3283	88.4	7.3

Nous utilisons l'équation (4.24) pour le calcul de l'horizon de temps T. L'environnement de test est présenté dans la sous section 4.2.3.

Expérimentations et résultats de l'évaluation du modèle

Une exigence imposée lors de l'évaluation du modèle est que toutes les informations concernant les tâches, machines et blocs de données soient fournies en entrée du modèle. Une hypothèse applicable seulement dans le contexte de l'ordonnancement hors ligne puisque dans un cas réel, on n'a pas d'information de la date de soumission des travaux ni les informations concernant les tâches associées.
Les différentes expériences sont répétées 10 fois. Les différents scénarios utilisés pour l'évaluation du modèle sont présentés dans le tableau 4.6, de la colonne 2 à la colonne 4, le nombre de travaux de chaque taille de travaux ; de la colonne 5 à la colonne 7, le nombre de machines de chaque type de machine ; la huitième colonne désigne le nombre de blocs de données par scénario. Les trois dernières colonnes présentent le nombre total de tâches, de machines et de Vcores par scénario.
Les résultats d'évaluation du modèle sont affichés dans le tableau 4.7. Nous affichons les scénarios pour lesquels le solveur CPLEX a réussi à résoudre plus de 80% des instances. La deuxième colonne désigne le nombre d'instances que le solveur n'a pas pu résoudre, la troisième présente le nombre d'instances résolues jusqu'à l'optimalité, la quatrième présente le nombre d'instances pour lesquelles CPLEX s'arrête parce qu'il a atteint la limite mémoire et la cinquième présente le nombre d'instances pour lesquelles CPLEX s'arrête parce qu'il a atteint la limite de temps fixé. Les colonnes 6, 7, 8 présentent le nombre (minimale, maximale et moyenne) de nœuds explorés par CPLEX durant la résolution du problème. Les colonnes 9, 10, 11 présentent le temps d'utilisation du processeur pour trouver les différentes solutions. La douzième colonne présente la moyenne des résultats de la fonction objectif obtenus pour les solutions optimales trouvées. La treizième colonne présente la moyenne des dates de fin de l'ordonnancement de toutes les tâches par scénarios pour les itérations possédant des solutions optimales.
D'autres mesures sont prises en compte pour suivre la manière dont les ressources sont exploitées à travers le temps. Ces valeurs permettent aux administrateurs d'étudier l'influence des poids affectés aux utilisateurs (et aux travaux en conséquence) sur le comportement de l'ordonnanceur et les performances de la grappe. Les figures (4.2a) et (4.2b) présentent les moyennes des consommations de la mémoire et HHD suite à l'exécution de 10 instances par scénarios. Dans la plupart des scénarios, le nombre de Vcores est plus faible que le nombre de tâches. Nous remarquons que :

— Les tâches qui occupent les Vcores commencent et terminent leurs exécutions presque en même temps. Le fait que les tâches occupent les Vcores et les libèrent en même temps créée des vagues d'exécution de ces tâches. L'exemple le plus clair est le scénario 4 où ce phénomène crée quatre vagues. Pour une vague particulière d'exécution de tâches (par exemple entre les instants 3 et 7), nous remarquons une augmentation des quantités de ressources consommées.

— Le fait que les tâches s'exécutent sous la forme de vagues engendre que les courbes de consommation de mémoires et de disques ont presque la même allure. Les quantités de

ressources consommées varient en fonction du nombre tâches affectées par vagues.

Rappelons que le degré de localité des tâches "Map" est le quotient du nombre de tâches "Map" qui s'exécutent sur les machines contenant les blocs de données qu'elles traitent sur le nombre total de tâches "Map". La figure (4.3) présente les dégrées de localité moyens dus à l'exécution des différents scénarios. Le niveau le plus élevé est noté lors de l'exécution du scénarios 4. Ce résultat est dû au fait que la grappe est constituée de 2 machines. Ces critères sont importants pour l'ordonnancement des travaux MapReduce mais ne font pas partie de l'objectif que le modèle cherche à optimiser.

Taille du modèle

Dans notre approche, nous cherchons à ordonnancer un nombre de tâches sur une grappe informatique. Vu la complexité du problème et le nombre de contraintes, la taille des instances que le solveur CPLEX peut résoudre jusqu'à l'optimalité est limitée à 29 tâches, 3 machines (17

FIGURE 4.2 – Moyenne des consommation des ressources (en Mo) par rapport au temps.

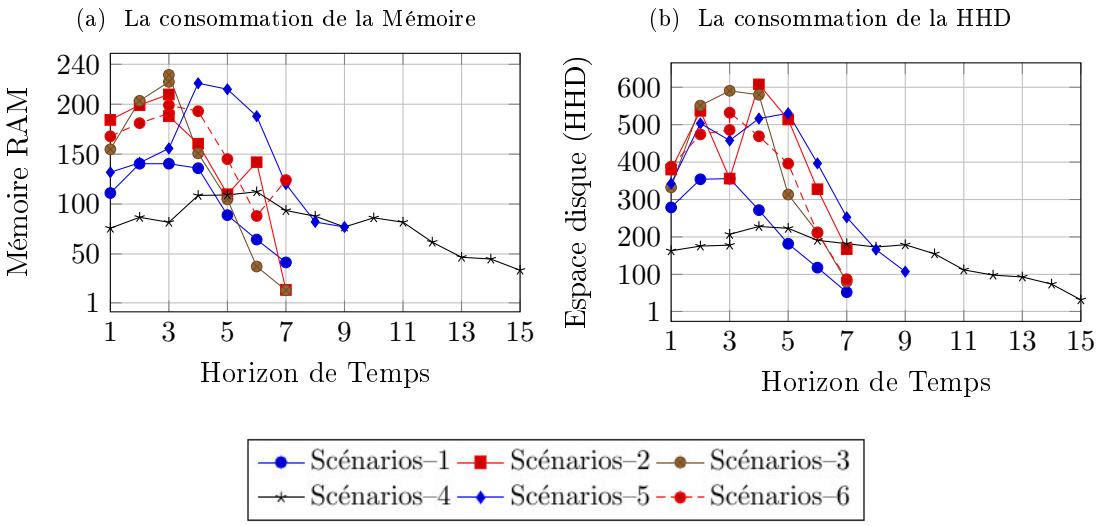
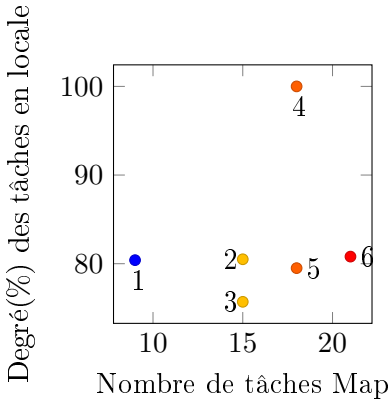


FIGURE 4.3 – Degrés de localité par scénario (10 instances par scénario)



coeurs virtuels) et 10 blocs. Pour le scénario 5, les résultats obtenus montrent que CPLEX parcourt en moyenne 26843 nœuds et au maximum 44155 nœuds de l'arbre de recherche. Il atteint ses limites mémoire et temps une fois (pour une instance). Malgré la petite taille des instances considérées, CPLEX ne parvient pas à résoudre jusqu'à l'optimalité toutes les instances.
Nous avons lancé certains scénarios pendant des durées plus grandes et sur des machines de configuration matérielle meilleure. Dans la grande majorité des cas, CPLEX ne parvient pas à résoudre jusqu'à l'optimalité les instances associées aux scénarios 7,8,9 et 10. Considérant l'instance suivante : *(i)* Une grappe de 3 machines de types respective M1, M2, M3 (11 Vcores "Map" et 6 slots "Reduce"). *(ii)* 3 travaux du type N2 (18 tâches "Map" et 6 tâches "Reduce"). Nous supposons que toutes les tâches ont une durée 2 et consomment 512 Mb de mémoire RAM. *(iii)* 10 blocs de données de taille 64 Mb *(iv)* 3 arcs (M1-M2, M1-M3, M2-M3). *(v)* Un horizon de temps égale à 13.
En tenant en compte cette instance, le modèle engendre environ 22620 variables et 74884 contraintes. Chaque unité de temps engendre une augmentation d'environ 1704 variables et 1010 contraintes en plus des contraintes générées par l'équation (4.11), le nombre de contraintes que cette dernière génère est égale à $\sum_{l=1}^{N^r} n_l^p * M(M-1) * \sum_{t=1}^{T-1}(t-1) * (T-t)$, soit 61776.
La quantité de contraintes générées dépend de la taille des instances et de l'horizon de temps ; elle influence la taille du modèle et le rend difficile à résoudre. Autrement dit, CPLEX ne parvient pas à trouver des solutions optimales, la plupart des solutions trouvées sont des solutions approchées. Nous nous trouvons confrontés aux problèmes d'insuffisance de la quantité mémoire réservée à CPLEX ou à l'insuffisance de la durée définie et imposée à CPLEX pour trouver la solution.

Afin d'augmenter la taille des instances que nous utilisons dans nos expérimentations, nous pensons à la relaxation des contraintes. Ce type de relaxation nous permettra de diminuer la taille du modèle, et d'augmenter la taille des instances.

4.3.5 Relaxation linéaire et simplification du modèle

Dans cette section, nous pensons utiliser la relaxation de contraintes pour calculer des bornes inférieures de bonne qualité (efficaces). En général, une borne de bonne qualité est une borne proche et possède une déviation acceptable par rapport à la solution optimale. Ensuite, nous pensons utiliser des algorithmes dédiés, plus rapide que la résolution du modèle mathématique, pour le calcul des solutions. Nous commençons par présenter la relaxation de la durée des tâches. Ensuite, nous présentons la relaxation des contraintes de précédence entre les tâches. La dernière relaxation que nous présentons est la relaxation des contraintes liées à la gestion des ressources de calcul. Enfin, nous évaluons les bornes obtenues suite à ces relaxations.

Relaxation de la durée des tâches

Notre modèle est indexé sur le temps, en conséquence, la taille de l'horizon de temps influence la quantité de contraintes générées. Nous décidons de relâcher la durée des tâches en supposant que la durée des tâches est 1 (c'est-à-dire $p_i = 1, \forall i \in L^r \cup L^m$). Puisque le calcul de l'horizon de temps dépend de la durée des tâches, la taille de l'horizon de temps sera considérablement

réduite.
Par rapport au modèle, les 4 contraintes qui gèrent la non interruption des tâches ne sont plus utiles puisque les tâches ne peuvent pas être interrompues car elles sont de durée unitaire. Pour le calcul de l'horizon de temps, nous utilisons la règle (4.24) : nous obtenons un horizon égal à 6. Le nombres de variables et de contraintes générées à partir du modèle sont de 10444 variables et de 38072 contraintes. Une réduction d'environ $\frac{2}{3}$ de la taille du modèle.

Cette relaxation permet de réduire l'horizon de temps nécessaire à l'ordonnancement des tâches donc elle réduit le nombre de contraintes générées et augmente la taille des instances que le solveur CPLEX peut résoudre.

Relaxation des contraintes de précédence

Les contraintes de précédence entre tâches "Map" et "Reduce" rendent le problème abordé difficile à résoudre. En conséquence, nous avons choisi dans cette partie de ne pas considérer ces relations entre tâches. Par rapport à notre modèle, le fait de relâcher la contrainte (4.29) réduit la complexité du problème. Nous utilisons l'équation (4.24) pour le calcul de l'horizon de temps.

Relaxation de contrainte de ressources

Dans le modèle présenté à la section 4.3.2, nous classons les cœurs virtuels en deux types, des cœurs virtuels responsables d'exécuter des tâches "Map" et des cœurs virtuels responsables d'exécuter des tâches "Reduce".
Dans cette partie, nous nous focalisons sur les ressources de calcul (Vcores) et nous supposons qu'un Vcore peut exécuter n'importe lequel des deux types de tâches "Map" ou "Reduce". Cette hypothèse fait évoluer la formule de calcul de l'horizon de temps (équation (4.24)) et nous permet d'obtenir une nouvelle équation (4.35) pour le calcul de l'horizon de temps.

$$T = \left[\frac{\sum_{i \in L^r \cup L^m} p_i}{\sum_{j=0}^{M} m_j^s} * 0.75 + \left(\frac{\sum_{i=1}^{N} n_i^{ra} * p_i}{\sum_{j=1}^{M} m_j^{ra}} \right) * 0.25 \right] + 2 * \underset{l \in L^r \cup L^m}{Max}(p_l) + 1. \quad \forall M \geq 2 \qquad (4.35)$$

4.3.6 Evaluations expérimentales et résultats

Après leurs présentations dans la section 4.3.5, nous commençons cette section par rappeler les formules de génération de données et les scénarios utilisés. Ensuite, nous discutons les résultats obtenues.
La méthode de génération des données d'entrée est similaire à la méthode décrite dans la section 4.3.4. Lorsque nous comparons les nouvelles bornes à la borne obtenu de l'évaluation sans relâcher aucune contrainte, nous utilisons les scénarios du tableau 4.6. La configuration des machines est dans le tableau 4.1, la configuration des travaux est dans le tableau 4.2 , les règles pour définir les quantités de ressources mémoire et espace de disque dur sont définies dans le tableau 4.3. La nature et le poids des travaux, la durée des tâches, l'emplacement et l'affectation des blocs de données aux tâches "Map" sont calculés de façon aléatoire. Les règles de génération de ces données sont définies dans 4.3.4.

TABLE 4.8 – Résultats des calculs : Nombre d'instances, par scénarios, résolus jusqu'à l'optimalité et la moyenne des fonctions objectifs (10 instances par scénario)

	Sans relaxation		Relaxation 1		Relaxation 2		Relaxation 3	
	Nb inst. résolus	Obj	Nb inst. résolus	Obj	Nb inst. résolus	Obj	Nb Inst. résolus	Obj
Sc 1	10	50.3	10	32.4	10	34.1	10	49.5
Sc 2	10	76.1	10	38.7	10	45.2	10	75.0
Sc 3	10	71.9	10	32.0	10	42.1	10	71.3
Sc 4	10	129.2	10	61.2	10	75.1	10	110.2
Sc 5	8	105.1	10	60.5	10	84.5	9	98.4
Sc 6	9	88.0	10	43.4	10	58.9	8	88.7
Sc 7	-	-	10	32.1	10	60.8	9	88.3
Sc 8	-	-	10	57.3	8	91.2	-	-
Sc 9	-	-	8	49.1	-	-	-	-
Sc 10	-	-	9	50.8	-	-	-	-

Résultats numériques et discussion

Lorsque nous évaluons la première relaxation (relaxation de la durée des tâches), nous générons des tâches de durée égale à 1. Nous utilisons les scénarios du tableau 4.6. Les critères que nous utilisons pour l'évaluation des relaxations sont la moyenne des solutions optimales obtenues, le nombre d'instances résolues jusqu'à l'optimalité par scénarios et le temps de calcul moyen des solutions. Dans le tableau 4.8, nous affichons les résultats, calculées à partir des différentes relaxations, donnés par CPLEX pour tous les scénarios. Nous présentons les résultats d'évaluation des scénarios que CPLEX a réussi à résoudre à l'optimum. Nous donnons dans le tableau 4.8, successivement pour chaque relaxation, le nombre d'instances et la moyenne des résultats de la fonction objectif obtenues pour les instances résolues. Pour chaque scénarios, nous générons 10 instances. Nous remarquons que la relaxation 1 permet de résoudre 44 tâches sur 4 machines (20 Vcores), suivie de la relaxation 2 ensuite la relaxation 3. Dans la majorité des cas, la relaxation 2 n'a pas pu résoudre les scénarios 9 et 10 et la relaxation 3 n'a pas pu résoudre les 8, 9 et 10. En étudiant la qualité des bornes, nous remarquons que la borne associée à la relaxation 1 a un écart moyen d'environ 44% par rapport aux moyennes des solutions obtenues du modèle sans les relaxations. Cet écart augmente avec l'augmentation du nombre de tâches et de leur durée. L'écart des solutions obtenues de la relaxation 2 est en moyenne de 30%. L'écart des solutions obtenues de la relaxation 3 est en moyenne moins de 10%. Cette dernière relaxation fournit la meilleure qualité de borne inférieure pour des tailles d'instances réduites. Notre analyse des résultats montre que pour les relaxations 2 et 3, CPLEX n'arrive pas à résoudre les instances liées aux scénarios 9 et 10 parce que CPLEX atteint les limites en terme de temps de calcul autorisé. Finalement, nous remarquons que les temps calcul des solutions pour les différents relaxations et scénarios ne sont pas négligeables. Toutes les relaxations ne permettent pas ni de résoudre des instances réelles ni de gagner en temps de calcul.
A la fin de cette section, nous synthétisons que parmi les relaxations considérées, ces relaxations

ne nous ont pas permis ni de réduire le temps de calcul ni d'augmenter la taille des instances résolues. De ce fait, nous utiliserons la borne optimale calculée à partir du modèle sans les relaxations pour évaluer les heuristiques que nous proposons dans le chapitre suivant.

4.4 Conclusion

Dans ce chapitre, nous modélisons le problème en utilisant la modélisation mathématique indexée par rapport au temps. Nous commençons par une version simplifiée de la modélisation du problème et nous calculons une première solution sur des petites instances, cette solution présente la borne inférieure que nous utilisons pour l'évaluation de l'heuristique que proposons. Ensuite nous faisons évoluer notre modèle pour s'approcher plus du problème réelle et nous cherchons à optimiser deux critères. Nous cherchons un équilibre entre les critères que nous prenons en compte, nous choisissons de minimiser la somme de ces deux critères. Les bornes que nous calculons dans ce chapitre nous permettront d'évaluer les heuristiques que nous proposons dans le chapitre suivant.

Dans le chapitre suivant, nous présentons des heuristiques de résolution du problème abordé. Nous les évaluons et nous étudions la qualité des solutions qu'elles proposent.

Chapitre 5

Résolution heuristique du problème d'ordonnancement en ligne

5.1 Introduction

Dans le chapitre 4, nous avons considéré le problème hors ligne, nous avons calculé une borne inférieure pour le problème de la minimisation du critère C_{max}. Pour résoudre rapidement le problème initial, nous avons proposé une première version d'une heuristique qu'on a nommé "LocFirst". Nous avons fait évoluer le modèle et nous avons calculé une borne inférieure pour le problème de la minimisation du critère $\sum w_j C_j + C_{max}$.
Dans ce chapitre, nous considérons le problème en ligne de l'ordonnancement des travaux Map-Reduce et nous présentons des heuristiques. Nous commençons par la minimisation du critère C_{max} et nous proposons une deuxième version de l'heuristique "LocFirst". Ensuite, nous abordons le problème en considérant la minimisation du critère $\sum w_j C_j + C_{max}$ et nous proposons deux heuristiques de résolution. Chaque fois que nous présentons une heuristique, nous analysons sa complexité et nous l'évaluons en terme de performances (par rapport aux objectifs fixés). Le présent chapitre est, en conséquence, organisé comme suit : la première section présente des considérations générales concernant l'évaluation des heuristiques. La deuxième section présente l'heuristique *LocFirst (V2)* pour la minimisation du C_{max}. La troisième section présente les heuristiques que nous proposons pour la minimisation du $\sum w_j C_j + C_{max}$. La dernière section est consacrée à la discussion et l'analyse des résultats obtenus.

5.2 Considérations générales concernant l'évaluation des heuristiques

Nous proposons dans ce chapitre trois heuristiques et nous les évaluons. Nous adoptons deux phases lors des évaluations : la première phase consiste à considérer que tous les travaux sont connus au début de l'ordonnancement et nous nous plaçons dans un contexte d'ordonnancement hors ligne. Nous évaluons les heuristiques sur de petites instances, par rapport aux bornes inférieures que nous avons obtenu dans le chapitre 4. Ensuite, la deuxième phase de l'évaluation

consiste à considérer le problème en ligne. Dans cette étape, nous utilisons des instances de taille moyenne et grande. Nous comparons les heuristiques que nous proposons à des heuristiques de la littérature. Les heuristiques choisies pour l'évaluation de nos propositions dépendent du critère utilisé.
Lorsque nous considérons la minimisation du critère C_{max}, nous évaluons nos heuristiques par rapport aux algorithmes suivants :

- «LTLM» (*Longest Time Left Most*) : les tâches de plus grande taille sont ordonnancées en premier,
- «STMA» (*Shortest Time Left Most Algorithm*) : les tâches de plus petite taille sont ordonnancées en premier,
- FIFO (*Fisrt in first out*) : le premier arrivé est le le premier servi,
- FIFO plus la prise en compte de la localité. Cette heuristique consiste à affecter les tâches sur les machines qui contiennent les blocs de données qu'elles traitent. S'il n'est pas possible de considérer ce critère, les tâches sont affectées aux machines qui peuvent les exécuter aux plus tôt.

Ces heuristiques concernent la définition des règles de séquencement des tâches. La règle d'affectation des tâches sur les machines consiste à affecter les tâches aux machines qui sont disponibles en premier (First available machine).
Lorsque nous considérons la minimisation du critère $\sum w_j C_j + C_{max}$, nous évaluons nos heuristiques par rapport aux algorithmes suivants :

- «FS» (*FairScheduler*)[169] est présenté dans la section 3.2.1 du chapitre 3.
- «LTLM» : les tâches de plus grande taille sont ordonnancées en premier,
- «STMA» : les tâches de plus petite taille sont ordonnancées en premier,
- FIFO : le premier arrivé est le le premier servi,
- FIFO plus la prise en compte de la localité.

Afin de simplifier la présentation algorithmique des heuristiques, nous introduisons les notations suivantes :

- $LJob$: la liste des travaux à ordonnancer. Chaque élément de la liste est caractérisé par les propriétés suivantes :
  - numéro (num),
  - date de soumission (SubmissionDate),
  - nombre de tâches Map (NbTaskMap),
  - nombre de tâches Reduce (NbTaskReduce),
  - date de fin de la dernière tâche Map (EndMapDate),
  - date de fin de la dernière tâche Reduce (EndReduceDate),
  - poids (Weight),
  - priorité (priority),

  - coût d'exploitation de la grappe (cost),
  - travail qui est ordonnancé (ou en cours) sur la grappe (scheduled).
- *Ltasks* : la liste des tâches à ordonnancer. Chaque tâche est caractérisée par les propriétés suivantes :
  - numéro (num),
  - type de tâche, Map ou Reduce (type),
  - durée (duration),
  - le travail auquel appartient la tâche (Job),
  - la machine sur laquelle la tâche est affectée (machine),
  - égale à 1 s'il y a une deuxième tentative d'affectation de la tâche (secondChance),
  - priorité (priority),
  - date de début d'exécution (start),
  - date de fin d'exécution (End),
  - le Vcore qui exécute la tâche (Vcore),
  - date à partir de laquelle la tâche doit s'exécuter (AllowStart),
  - date avant laquelle la tâche doit terminer son exécution (StrictEnd).
- *Lmachines* : la liste des machines. Chaque machine est caractérisée par les propriétés suivantes :
  - numéro (num),
  - la date de fin de la dernière tâche Map qui lui est affectée (LastDateMap),
  - la date de fin de la dernière tâche Reduce qui lui est affectée (LastDateReduce),
  - Liste des dates de fin d'exécution des dernières tâches sur les Vcores (LastDateofVcores),
  - Type des tâches que les Vcores exécutent (TypeVcores[s], $\forall s \in [1 \dots m_j^s]$),
  - Liste des tâches qui s'exécutent sur les Vcores de la machine (Vcore[s], $\forall s \in [1 \dots m_j^s]$),
  - Liste des dates de libération des Vcores (LastDateofVcore[s], $\forall s \in [1 \dots m_j^s]$).

5.3 Minimisation du critère C_{max}

Le critère C_{max} est le premier critère que nous avons considéré dans nos études de l'ordonnancement des travaux MapReduce. Dans le chapitre 4, une première version de l'heuristique "LocFirst" pour la version hors ligne du problème a été proposée. Dans cette section, nous adaptons cette heuristique à la version en ligne du problème pour obtenir une heuristique nommée "LocFirst (V2)".

5.3.1 Présentation de l'heuristique LocFirst (V2)

"LocFirst (V2)" est une heuristique Glouton. Elle est inspirée de l'algorithme FIFO. Elle commence par le calcul de la priorité de chaque tâche. Ensuite, le choix glouton consiste à chaque itération, à récupérer la tâche de priorité la plus élevée et à l'affecter à une machine. Elle commence par les machines qui contiennent les données que la tâche traite. La machine choisie doit respecter : *(i)* les contraintes sur les quantités de ressources et *(ii)* la contrainte sur la date de début d'exécution au plus tard *AverageEndDate* que nous définissons dans l'algorithme 6. L'heuristique "LocFirst (V2)" considère qu'un travail n'est pris en compte pour être ordonnancé qu'après sa soumission. Elle prend en entrée : la liste des tâches, la liste des machines et la liste des blocs. Elle considère qu'une ancienne affectation des tâches n'est jamais remise en cause, c'est-à-dire, on ne réaffecte pas une tâche qui est déjà affectée.

Algorithme 6 : Algorithme de l'heuristique LocFirst (V2)

```
   Données :
1      LJobs ← {m}, ∀m = 1...W                          /* Liste des travaux */
2      Ltasks ← {i}, ∀i = 1...N                         /* Liste des tâches */
3      Lmachines ← {m}, ∀m = 1...M                      /* Liste des machines */
4      LdataBloc ← {b}, ∀b ∈ A^b                        /* Liste des blocs */
   Résultat : Les tâches de Ltasks sont ordonnancées sur les machines de la liste Lmachines
5  PriorityDefinition(LJobs,Ltasks) ;
6  pour chaque tâche i ∈ Ltasks faire
7      si (Ltasks[i].machine n'est pas défini) alors
8          AverageEndDate = 0 ;
9          pour m' ∈ Lmachines faire
10             si Ltasks[i] est un Map alors
11                 AverageEndDate = AverageEndDate + Lmachines[m'].LastDateMap ;
12             si Ltasks[i] est un Reduce alors
13                 AverageEndDate = AverageEndDate + Lmachines[m'].LastDateReduce ;
14         MoyenDateFin = (1 + AverageEndDate) / M ;
           /* MachLoc et Mach: machines sur lesquelles i peut s'exécuter        */
15         MachLoc ←
            ReturnListMachineLoc(Ltasks[i], LdataBloc, Lmachines, AverageEndDate) ;
16         si MachLoc alors
17             ExecuteTask( Ltasks[i],MachLoc) ;
18         sinon
19             Mach ← ReturnListMachineDis(Lmachines, Ltasks[i], AverageEndDate) ;
20             ExecuteTask( Ltasks[i],Mach) ;
```

"LocFirst (V2)" (algorithme 6) commence par le calcul des priorités et le tri de la liste des tâches (la fonction *PriorityDefinition* à la ligne 5). Ensuite, l'heuristique calcule la moyenne des dates

de libération des ressources (AverageEndDate) sur la grappe (de la ligne 7 à la ligne 14). L'étape suivante considère les tâches associées aux travaux qui ne sont pas déjà ordonnancés avant de commencer le nouvel ordonnancement.
Pour chaque tâche "$Ltasks[i]$" non ordonnancée de la liste "Ltasks", la fonction RetounerListeMachineLocV2() (ligne 15) récupère dans "$MachLoc$" la machine contenant une des réplications des données qu'elle traite. "MachLoc" respecte une condition sur la date de fin autorisée pour $Ltasks[i]$. Si la tâche $Ltasks[i]$ n'est pas affectée en mode local, la fonction RetounerListeMachineDisV2() (ligne 19) récupère dans $Mach$ la machine qui respecte la condition sur la date de fin autorisée pour $Ltasks[i]$.
L'algorithme 7 présente la méthode de définition des priorités des tâches. L'idée est que les tâches associées au travail de durée la plus longue sont exécutées en premier. La durée d'un travail est la somme des durées de ses tâches.

Algorithme 7 : Algorithme de la fonction "PriorityDefinition"

```
   Données :
1      LJob ← {w'}, ∀w' = 1...W                          /* Liste des travaux */
2      Ltasks ← {i}, ∀i = 1...N                          /* Liste des tâches */
3      CurrentDate      /* Date de soumission du premier travail non ordonnancé */
   Résultat : Les tâches de Ltasks sont ordonnées selon leurs priorités
4  pour chaque w' ∈ LJob faire
5      pour chaque i ∈ E^w' faire
6          LJob[w'].priority ← LJob[w'].priority + p_i ;

7  HighestPriority ← −1 ;          /* HighestPriority : la priorité la plus élevé */
8  pour chaque i ∈ Ltasks faire
9      si Ltasks[i].submit > CurrentDate alors
10         si Ltasks[i].type == Map alors
11             Ltasks[i].priority ← LJob[w'].priority ;
12         si Ltasks[i].type == Reduce alors
13             Ltasks[i].priority ← LJob[w'].priority − 1 ;
14         si HighestPriority < Ltasks[i].priority alors
15             HighestPriority = Ltasks[i].priority ;

16 pour chaque i ∈ Ltasks faire
17     si Ltasks[i].submit < CurrentDate    /* Vérifier si la tâche est ordonnancée */
18      alors
19         Ltasks[i].priority = HighestPriority + 1 ;

20 Sort(Ltasks) ;                          /* trie en fonction des priorités des tâches */
```

La fonction "PriorityDefinition" commence par calculer les priorités des travaux (entre les lignes 4 et 6). La deuxième étape consiste à calculer les priorités des tâches et la valeur de la priorité la plus élevé existante sur la grappe (entre les lignes 8 et 15). La priorité de chaque tâche Reduce est calculée de façon à respecter la relation de précédence entre les tâches Map et Reduce.

Algorithme 8 : Algorithme de la fonction "ReturnListMachineLocV2"

Données :

1 $Task$ /* Tâche à ordonnancer */
2 $LdataBloc \leftarrow \{b\}, \forall b \in A^b$ /* Liste des blocs */
3 $Lmachines \leftarrow \{m\}, \forall m = 1...M$ /* Liste des machines */
4 $AverageEndDate$ /* Date limite au plus tard pour commencer $Task$ */

Résultat : Retourner la machine qui peut exécuter $Task$ en respectant le critère de localité

/* $Cond1$: $TempListMach[i]$ contient des ressources pour exécuter $Task$ */
/* $Cond2$: AverageEndDate $\geq TempListMach[i].LastDateMap$ */
/* $Cond3$: la machine $c2.machine$ a les ressources pour exécuter $Task$ */
/* $Cond4$: AverageEndDate $\geq Lmachines[c2.machine].LastDateReduce$ */
/* $Cond5$: $TempMach.LastDateMap < TempListMach[i].LastDateMap$ */
/* $Cond6$: $TempMach.LastDateReduce < TempListMach[c2.num].LastDateReduce$ */

5 $TempListMach \leftarrow \emptyset$
6 **si** *Task est une tâche Map* **alors**
7 $TempListMach \leftarrow D_{(LdataBloc[n^b_{Task.num}])}$; /* Ensemble de machines qui contiennent les données que Task traite */
8 $TempMach \leftarrow TempListMach[1]$;
9 **pour** *i de 2 à 3* /* un bloc de données est répliqué au maximum 3 fois */
10 **faire**
11 **si** $Cond1$ *et* $Cond2$ *et* $Cond5$ **alors**
12 $TempMach \leftarrow TempListMach[i]$;
13 **si** *Task est une tâche Reduce* **alors**
14 $TempListMach \leftarrow$ liste des machines sur lesquelles s'exécutent les tâches de E_{Task};
/* Ensemble de machines qui ont exécuté les tâches Map qui précèdent $Task$ */
15 $TempMach \leftarrow TempListMach[1]$;
16 **pour** $c2 \in E_{Task}]$ **faire**
17 **si** $Cond3$ *et* $Cond4$ *et* $Cond6$ **alors**
18 $TempMach \leftarrow TempListMach[c2]$;
19 Retourner ($TempMach$);

Enfin, les tâches sont triées de façon décroissante selon les priorités définies.
Les algorithmes 8 et 9 présentent les algorithmes de sélection de la machine sur laquelle une tâche est affectée. L'algorithme 8 récupère (si elle existe) la machine qui peut exécuter $Task$ en respectant le critère de localité. Pour les tâches Map, la localité est définie par les machines contenant les données de ces tâches. Pour les tâches Reduce, la localité est définie par les machines qui ont exécutées une des tâches Map qui les précède. L'algorithme doit trouver la machine qui peut commencer au plus tôt l'exécution de $Task$ en respectant la date limite au plus tard autorisée.
L'algorithme 9 présente la fonction "RetounerListeMachineDisV2" qui récupère la première machine qui se libère avant la date de fin moyenne autorisée ($MoyenDateFin$). Si la tâche est un "Map" (resp. pour les tâches "Reduce"), $MoyenDateFin$ est égale à la moyenne des dates de libération des Vcores qui exécutent des tâches "Map" (resp. pour les Vcores "Reduce").

Algorithme 9 : Algorithme de la fonction "RetounerListeMachineDisV2"

Données :

1 $Lmachines \leftarrow \{m\}, \forall m = 1...M$ /* Liste des machines */
2 $Task$ /* Tâche à ordonnancer */
3 $AverageEndDate$ /* Date limite au plus tard pour commencer $Task$ */

Résultat : Retourner la machine qui peut exécuter $Task$

/* $Cond1$: la machine $Lmachines[c1]$ a les ressources pour exécuter $Task$ */
/* $Cond2$: $AverageEndDate > Lmachines[c1].LastDateMap$ */
/* $Cond3$: $AverageEndDate > Lmachines[c1].LastDateReduce$ */

4 $Lmach \leftarrow \emptyset$
5 **pour** $c1$ *de 1 à* $M()$ **faire**
6 **si** $Cond1$ *et* $Cond2$ *et* $Task$ *est une tâche Map* **alors**
7 Retourner $(Lmachines[c1])$;
8 **si** $Cond1$ *et* $Cond3$ *et* $Task$ *est une tâche Reduce* **alors**
9 Retourner $(Lmachines[c1])$;

L'algorithme 10 est celui de la fonction "ExecuteTask" qui exécute les tâches sur les machines. Si la tâche est une tâche Map, elle est affectée au Vcore qui se libère le premier et qui peut commencer l'exécution de la tâche avant $MoyenDateFin$. Si la tâche est une tâche Reduce, elle est affectée au Vcore qui est libre après la fin de la dernière tâche Map, appartenant au même travail et avant $MoyenDateFin$.
"LocFirst (V2)" considère que si la décision d'affecter une tâche pour s'exécuter sur une machine est prise, elle n'est pas remise en cause. Cependant, elle met à jour la priorité de la tâche pour la maintenir en tête de la liste obtenue suite à une nouvelle soumission.

5.3.2 Analyse de la complexité de LocFirst (V2)

L'algorithme "*LocFirst*" a une complexité de l'ordre de $\mathcal{O}((N^2 + N * M) * m_j^s)$. Rappelons que N est le nombre de tâches, M est le nombre de machines et m_j^s est le nombre de Vcores dans la machine j.

Calcul :
L'algorithme 8 est constitué de 8 conditions :

- On nomme $C1$ et $C2$ les contraintes destinées à la vérification du type des tâches (Map ou Reduce). Elles sont de complexité f_1 et f_2. La vérification du type de la tâche est de complexité $f_1 = f_2 = \mathcal{O}(1)$.
- Les $Cond1$ et $Cond3$ sont de complexité noté f_3 et f_5. elles servent à la vérification si une machine contient les ressources pour exécuter une tâche. Elle est de complexité $f_3 = f_5 = \mathcal{O}(1)$.
- Les $Cond2$ et $Cond4$ sont de complexité noté $f4$ et $f6$. elles servent à la vérification de la

condition sur la durée autorisée. Elles sont de complexité $f4 = f6 = \mathcal{O}(1)$.

— Les $Cond5$ et $Cond6$ sont de complexité $f7 = f8 = \mathcal{O}(1)$.

Algorithme 10 : Algorithme de la fonction "ExecuteTask"

Données :
1 $Mach$, $\forall Mach \in 1...M$ `/* Machine sur laquelle la tâche doit être ordonnancée */`
2 $Task$ `/* Tâche à ordonnancer */`
3 $MoyenDateFin$ `/* Date limite au plus tard pour ordonnancer` $Task$ `*/`
Résultat : La tâche $Task$ est ordonnancée sur un Vcore de la machine $Mach$

4 $TempSlot \leftarrow \varnothing$;
5 **si** *Task est une tâche Map* **alors**
6 $FirstDate = HV$; `/* HV : grande valeur numérique */`
7 **pour** *i de* $m^{Sr}_{(Mach.num)}$ *à* $m^{s}_{(Mach.num)}$ **faire**
8 **si** *i est est libre avant* $MoyenDateFin$ *et avant* $FirstDate$ **alors**
9 $TempSlot = i$;
10 $FirstDate = Mach.LastDateofSlot[i]$;
11 **si** *Task est une tâche Reduce* **alors**
12 $FirstDate =$ date de fin de la dernière tâche Map associée à Task ;
13 **pour** *i de* 0 *à* $m^{Sr}_{(Mach.num)}$ **faire**
14 **si** *i est libre avant* $MoyenDateFin$ *et après* $FirstDate$ **alors**
15 $TempSlot = i$;
16 $FirstDate = Mach.LastDateofSlot[i]$;
17 $Task.machine = Mach.num$;
18 $Task.Vcore = TempSlot$;
19 $Task.start = FirstDate$;
20 $Task.End = Task.start + p(Task.num)$;
21 $Mach.Vcore[TempSlot] = Task.num$;
22 $Mach.LastDateofVcore[TempSlot] = Task.End$;
23 `UpdateResourceOnMachine ()` ;

L'algorithme 8 est constitué de deux boucles "pour" :

— si la tâche est une tâche Map (la condition $C1$ est vérifiée), on récupère pour chaque bloc, que la tâche $Task$ traite, l'emplacement des répliquas. La boucle "*pour*", qui s'exécute lorsque la première condition est vérifiée, est de complexité $f_9 = \mathcal{O}(n^b_{Task} * r_b * max(f3, f4, f7)) = \mathcal{O}(n^b_{Task} * r_b)$. Or, une tâche Map effectue des traitements sur un seul bloc de données. De plus, le taux de répliquas d'un bloc est la constante d'entrée r_b. De ce fait, la boucle "*pour*" est de complexité $f_9 = \mathcal{O}(n^b_{Task})$.

— si la tâche est une tâche Reduce (la condition $C2$ est vérifiée), la boucle "*pour*" consiste

TABLE 5.1 – Écart relatif moyen (en %) entre la solution fournie par "LocFirst (V2)" et la solution optimale (10 instances par scénarios)

Scénarios	**Solution optimale** (calculée par CPLEX)	**Solution calculée par "LocFirst (V2)"**	Écart relatif (%)
1	4.4	4.6	4.5
2	4	4.25	6
3	4.7	4.8	2.1
4	4	4.1	2.5
5	8.25	8.5	3
6	5.4	5.6	3
7	7.1	7.4	4.2
8	5	5.2	3
9	6	6.2	3.3
10	5.5	5.8	5.4
11	4.66	4.8	3
Moyenne			3.73

à chercher les machines sur lesquelles les tâches Map, qui précèdent $Task$, sont exécutées. Elle est de complexité $f_{10} = \mathcal{O}(N^m * max(f5, f6, f8)) = \mathcal{O}(N^m)$.

La fonction "*RetounerListeMachineLoc* (V2)" est de complexité $F1 = \mathcal{O}(max(max(f_1, f_9), max(f_2, f_{10}))) = \mathcal{O}(max(f_9, f_{10})) = \mathcal{O}(N^m)$.

L'algorithme 9, associé à la fonction "*RetounerListeMachineDis* (V2)", est basé sur trois conditions de complexité $\mathcal{O}(1)$ chacune et une boucle "pour" de complexité $\mathcal{O}(M)$. Dès qu'on trouve une machine libre qui peut exécuter $Task$, on retourne le résultat. La fonction "*Retouner ListeMachineDis*" est de complexité $F2 = \mathcal{O}(M)$.

L'algorithme 7 associé à la fonction "PriorityDefinition" est de complexité $F3 = \mathcal{O}(N + N + N \log N) = \mathcal{O}(N \log N)$. Le calcul de la priorité de chaque travail et le calcul de la priorité des tâches associées aux travaux est de complexité $\mathcal{O}(N)$. Le tri de la liste des tâches en fonction de leurs priorités est de complexité $\mathcal{O}(N \log N)$.

L'algorithme 10 associé à la fonction "*ExecuteTask*" est de complexité $F4 = \mathcal{O}(m_j^s), \quad \forall j \in [1..M]$.

En résumé :

1. la fonction de trie des tâches est de complexité $F3 = \mathcal{O}(N \log N)$.
2. la complexité de la fonction "*RetounerListeMachineLoc*" est $F1 = \mathcal{O}(N^m)$.
3. la fonction "*RetounerListeMachineDis*" est de complexité $F2 = \mathcal{O}(M)$.
4. la fonction "*ExecuteTask*" est de complexité $F4 = \mathcal{O}(m_j^s), \quad \forall j \in [1..M]$.

L'algorithme "*LocFirst (V2)*" est de complexité $\mathcal{O}(N \log N + N * m_j^s * (N^m + M)) = \mathcal{O}(N^2 + N * M) * m_j^s$.
Nous présentons dans la section suivante les résultats de l'évaluation expérimentale de cette heuristique.

5.3.3 Évaluation de l'heuristique "LocFirst (V2)"

L'évaluation de "LocFirst (V2)" est réalisée en deux étapes. La première consiste à considérer que les travaux sont soumis en même temps et nous disposons de toutes les informations qui les concernent avant le début de l'ordonnancement (contexte hors ligne). Nous considérons des

FIGURE 5.1 – Évaluation de "LocFirst (V2)" dans un contexte hors ligne.

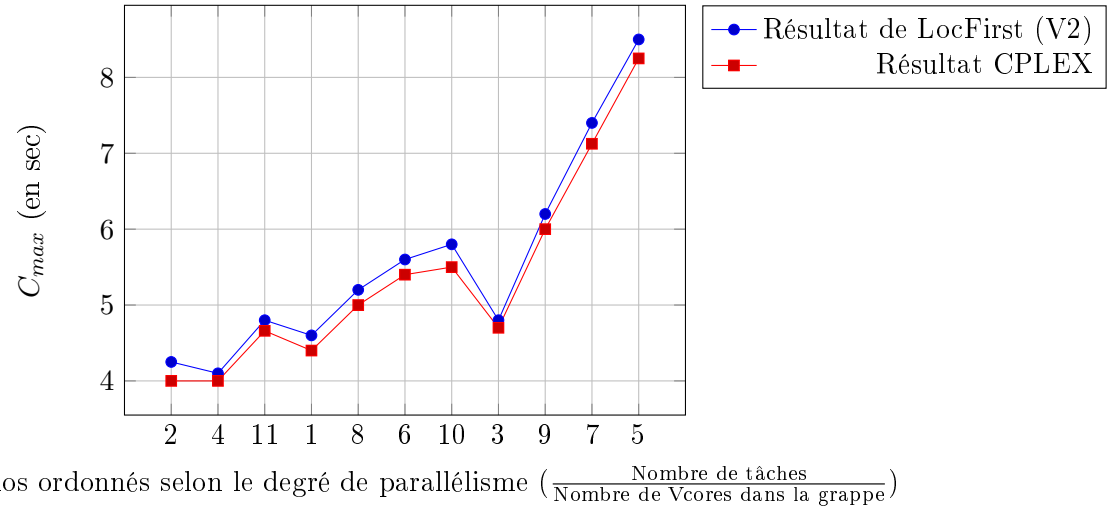
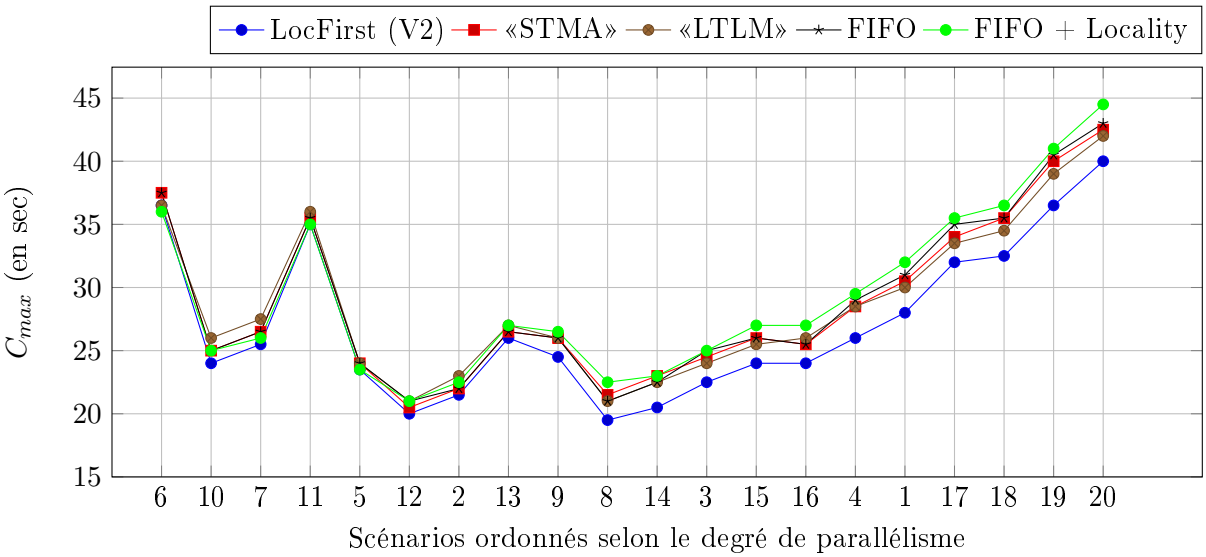


FIGURE 5.2 – Moyenne des valeurs du critère C_{max} (contexte hors ligne et tâches de durée unitaire)



tâches de durée égale à 1. Nous comparons, en conséquence, les résultats obtenus par l'heuristique "LocFirst (V2)" avec les résultats obtenus lors de l'évaluation du modèle dans la section 4.2 du chapitre 4. La génération des données est décrite dans la section 4.3.4 du chapitre 4. Nous utilisons les scénarios présentés dans le tableau 4.4. Pour chaque scénario, nous générons 10 instances. Les résultats de cette évaluation sont synthétisés dans la figure 5.1. Les valeurs affichées sont des valeurs moyennes du C_{max} des solutions calculées.

"LocFirst (V2)" fournit des résultats proches des résultats obtenus lors de l'évaluation de la

TABLE 5.2 – Scénarios utilisés dans l'évaluation en ligne des heuristiques (nombre de tâches, machines et blocs de données)

	N1	N2	N3	M1	M2	M3	Nbre de Blocs	Nbre des tâches	Nbre de machines	Nbre de Vcores	$\frac{\text{Nb tâches}}{\text{Nb Vcores}}$
Scéna. 1	2	2	2	2	1	1	30	370	4	24	15.41
Scéna. 2	2	2	2	2	2	2	30	370	6	34	10.88
Scéna. 3	4	2	2	2	2	2	40	430	6	34	12.64
Scéna. 4	4	4	2	2	2	2	80	500	6	34	14.7
Scéna. 5	4	4	2	2	4	2	80	500	8	48	10.41
Scéna. 6	7	4	3	4	5	4	100	690	13	75	9.2
Scéna. 7	7	4	4	4	6	4	100	790	14	83	9.5
Scéna. 8	7	7	5	4	6	6	100	1095	16	88	12.44
Scéna. 9	7	7	6	6	6	6	100	1230	18	102	12.05
Scéna. 10	7	7	10	10	10	10	200	1595	30	170	9.38
Scéna. 11	10	7	10	10	10	10	200	1685	30	170	9.91
Scéna. 12	10	10	10	10	10	10	250	1840	30	170	10.82
Scéna. 13	10	12	12	11	11	11	300	2220	33	187	11,87
Scéna. 14	10	12	15	11	12	13	300	2520	36	200	12.6
Scéna. 15	10	15	16	12	12	15	450	2800	39	217	12.9
Scéna. 16	15	15	18	12	12	15	450	3150	39	217	14.5
Scéna. 17	20	15	20	12	12	16	450	3500	40	220	15.9
Scéna. 18	20	20	24	14	14	16	500	4200	44	248	16.93
Scéna. 19	25	25	25	14	14	16	500	4750	44	248	19.15
Scéna. 20	27	27	27	14	14	16	500	5130	44	248	19.88

TABLE 5.3 – Différentes tailles des travaux utilisés lors de l'évaluation des heuristiques

Taille	**tâches Reduce**	**tâches Map**
$T1$	5	25
$T2$	10	50
$T3$	20	80

TABLE 5.4 – Différents types de travaux utilisés lors de l'évaluation des heuristiques

Type	**Description**
$N1$	CPU-intensive
$N2$	RAM-intensive
$N3$	I/O-intensive

première version. Le ratio par rapport à la solution obtenue lors de l'évaluation du modèle est d'environ 4% en moyenne sur les instances des scénarios considérés. Le tableau 5.1 présente pour chaque scénario l'écart relatif (en %) entre la solution fournie par l'heuristique "LocFirst (V2)" et la solution optimale calculée dans le chapitre 4. Étant donné que la taille des instances est petite, le temps d'exécution de l'heuristique "LocFirst (V2)" est de l'ordre de la microseconde. Le temps utilisé par "LocFirst (V2)" pour trouver les solutions est négligeable par rapport au

FIGURE 5.3 – Moyennes des temps de fin d'exécution des travaux C_{max} de chacun des scénarios ordonnés selon le degré de parallélisme

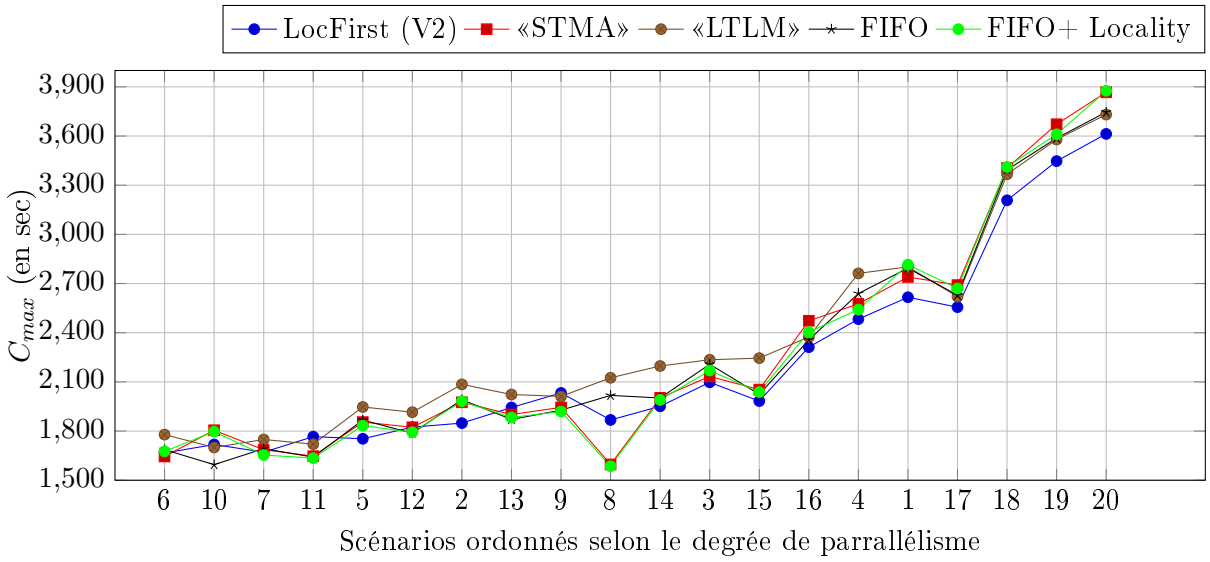
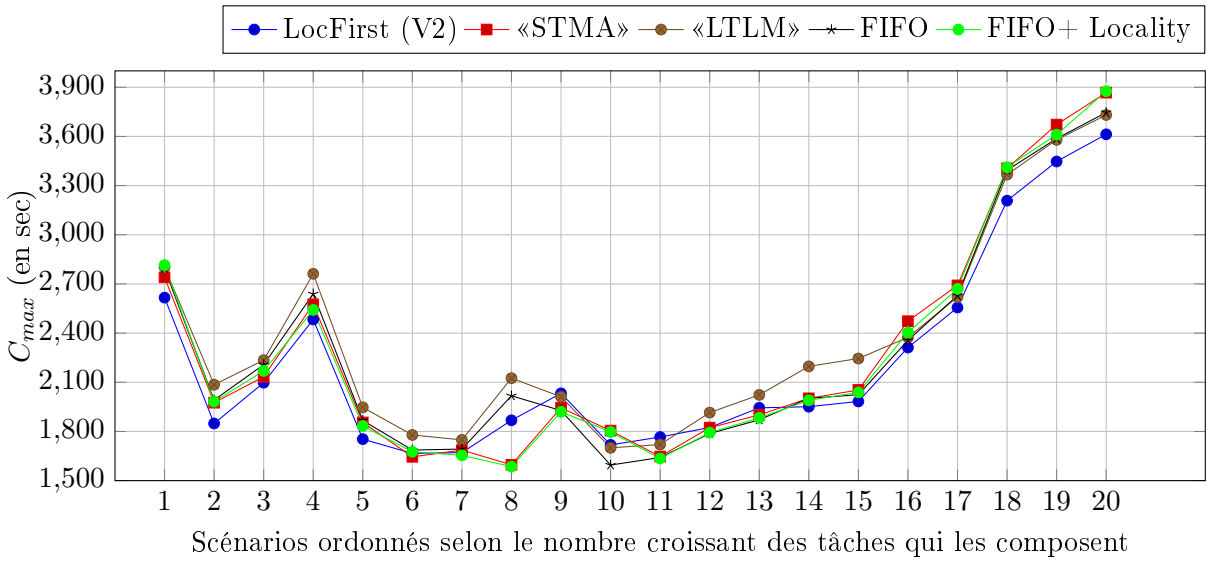


FIGURE 5.4 – Moyennes des temps de fin d'exécution des travaux C_{max} de chacun des scénarios ordonnés selon le nombre de tâches par scénarios



temps consommé par CPLEX pour calculer les solutions optimales de chaque scénario.

Après l'obtention de ces résultats sur des petites instances, nous évaluons "LocFirst (V2)" sur des moyennes et grandes instances. Nous la comparons aux heuristiques «LTLM», «STMA», FIFO et FIFO avec la localité. Les résultats de l'évaluation de "LocFirst (V2)" sont affichés dans la figure 5.2. Ces expérimentations sont effectuées dans les mêmes conditions que l'évaluation

TABLE 5.5 – Temps de calcul moyen (en sec) associés aux heuristiques pour chaque scénario

Scénarios	LocFirst	«STMA»	«LTLM»	FIFO	FIFO + Locality
1	1.3	3.9	6.2	3.1	4.9
2	0.6	5.2	6.1	2.8	7.1
3	1.5	5.6	6.4	4.1	7.7
4	2.2	5.2	7.5	4.8	8.2
5	1.7	7.1	8.4	3.3	9.1
6	4.5	11.2	12.1	5.9	11.1
7	4.9	13.5	14.8	7.1	12.6
8	8.2	15.8	18.1	8.9	15.1
9	12.9	24.2	25.1	14.0	20.8
10	29.1	50.2	48.9	28.1	45.0
11	32.2	62.9	55.1	39.8	49.3
12	37.7	70.3	72.1	64.8	57.2
13	51.4	79.2	81.2	72.7	73.1
14	65.4	97.7	93.8	80.2	91.9
15	91.1	125.4	127.2	111.3	140.1
16	97.3	145.1	148.7	120.3	154.2
17	120.6	160.2	171.7	140.2	179.1
18	174.3	235.1	223.2	169.8	235.9
19	191.2	255.1	264.7	183.7	278.3
20	206.3	279.7	290.1	209.8	317.1

sur des petites instances (c'est-à-dire le mode est hors ligne, des tâches de durée unitaire, la même méthode de génération des données d'entrée décrite dans la section 4.3.4 du chapitre 4, etc.). Les scénarios que nous utilisons pour évaluer "LocFirst (V2)" sur des instances de moyenne et grande taille sont affichés dans le tableau 5.2. "LocFirst (V2)" fournit de bonnes solutions, dans la plupart des cas, elle trouve des solutions meilleures que celles fournies par les autres algorithmes. L'écart entre "LocFirst (V2)" et les autres algorithmes est en moyenne entre 1% et 10 % en fonction du scénario.

Lors de l'évaluation de l'heuristique dans un contexte en ligne, nous supposons que les dates de soumission des travaux suivent la loi de poisson. Nous générons les données d'entrée de la façon suivante : nous utilisons les machines dont la configuration est affichée dans le tableau 4.1. Nous générons aléatoirement la taille et le type d'un travail. Les tailles des travaux générés dans cette évaluation sont affichées dans le tableau 5.3. Le type qui peut être associé à un travail est un des trois types décrit dans le tableau 5.4. En fonction du type choisi pour le travail, nous utilisons les formules du tableau 4.3 pour la génération des quantités de mémoire RAM et disque HHD utilisées par les tâches.

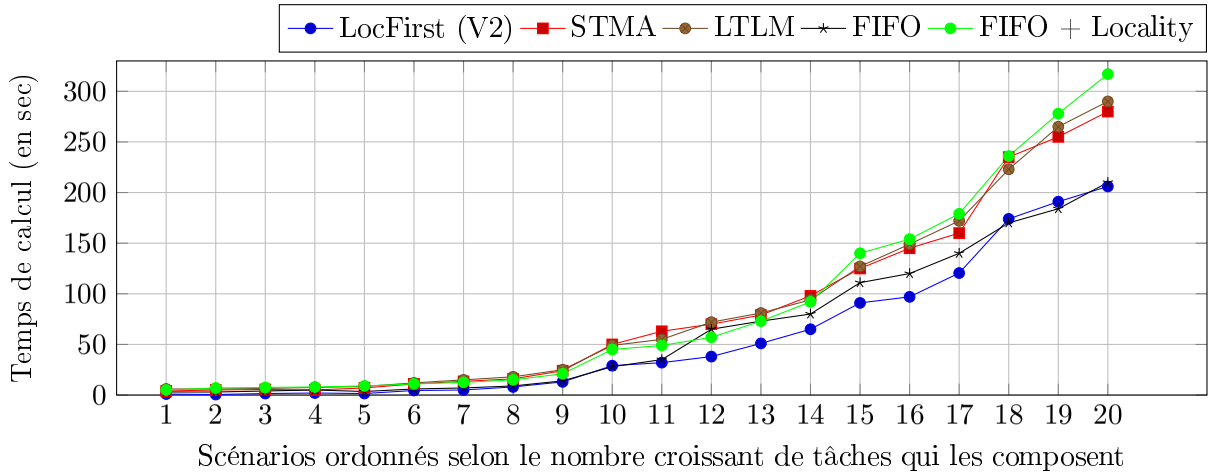
TABLE 5.6 – Moyenne des résultats des évaluations de LocFirst (V2)

Scénarios	C_{max}
1	2616.4
2	1848.1
3	2096.9
4	2483.3
5	1753.1
6	1665.7
7	1671.2
8	1868.4
9	2032.1
10	1717.8
11	1765.7
12	1824.3
13	1944.1
14	1950.8
15	1983.4
16	2312.5
17	2556.1
18	3207.9
19	3447.2
20	3613.5

TABLE 5.7 – Écart relatif moyen (en %) entre la valeur du C_{max} fourni par LocFirst (V2) et la valeur du C_{max} calculé par les heuristiques utilisées dans l'évaluation.

Scénarios	«STMA»	Ecart de LocFirst	«LTLM»	Ecart de LocFirst	FIFO	Ecart de LocFirst	FIFO+ Locality	Ecart de LocFirst
1	2740.3	4.7	2801.6	7.1	2794.4	6.8	2815.0	7.6
2	1975.1	6.8	2085.3	12.8	1991.2	7.7	1982.4	7.3
3	2132.7	1.7	2235.1	6.6	2207.5	5.3	2169.1	3.4
4	2576.4	3.7	2762.4	11.2	2638.6	6.3	2540.6	2.3
5	1855.2	5.8	1946.7	11	1866.9	6.5	1833.2	4.6
6	1645.2	-1.2	1603.1	-3.8	1686.3	1.2	1676.2	0.6
7	1686.0	0.9	1747.8	4.6	1692.1	1.3	1654.5	-1
8	1595.7	-14.6	2125.3	13.7	2017.1	8	1585.1	-15.2
9	1943.9	4.3	2011.1	-1.0	1927.2	-5.2	1918.7	-5.6
10	1805.3	5.1	1700.4	-1.0	1595.1	-7.1	1796.9	4.5
11	1646.6	-6.7	1720.1	-2.6	1640.6	-7.1	1635.2	-7.4
12	1823.1	-0.1	1914.9	5.0	1786.8	-2.1	1793.5	-1.7
13	1901.4	-2.2	2023.2	4.1	1871.3	-3.7	1882.6	-3.2
14	2002.3	2.6	2196.8	12.6	2002.1	2.6	1990.3	2
15	2053.0	3.5	2245.5	13.2	2023.7	2	2037.8	2.7
16	2472.8	6.9	2374.2	2.7	2356.4	1.9	2404.1	4
17	2691.2	5.3	2619.6	2.5	2628.9	2.8	2670.4	4.5
18	3404.9	6.1	3367.3	5	3394.5	5.8	3411.6	6.3
19	3672.1	6.5	3579.2	3.8	3587.1	4.1	3611.2	4.8
20	3867.5	7	3730.7	3.2	3746.5	3.7	3876.5	7.3

FIGURE 5.5 – Moyenne des temps de calcul de chacun des scénarios



De façon similaire à la génération des données hors ligne, nous générons les informations concernant :

— les quantités de données d'entrée et de sortie d'une tâche.

— la taille d'un bloc de donnée et son emplacement. Nous considérons que les blocs sont répliqués deux fois.

— l'architecture réseau et ses caractéristiques telles que la gestion de la bande passante de transfert de données et de migration de blocs.

Dans la littérature, il existe plusieurs formules pour estimer la durée des tâches. Cependant, d'après nos étude expérimentales, il n'y a pas de relation entre la taille des données et la durée d'exécution d'une tâche Map ou Reduce. La durée d'une tâche dépend de la nature du traitement qu'elle réalise. De ce fait, les formules existantes ne coïncident pas avec un comportement réel sur une grappe. Nous décidons dans ces travaux de générer aléatoirement la durée des tâches selon une loi normale entre 1 et 180 secondes. Les scénarios utilisés pour l'évaluation en ligne de "LocFirst (V2)" sont affichés dans le tableau 5.2.
Pour chaque scénario, nous calculons l'écart relatif (en %) entre la solution fournie par l'heuristique "LocFirst (V2)" et les solutions fournies par les heuristiques d'évaluation. Un écart positif de valeur "X" signifie que "LocFirst (V2)" fournit une solution inférieure (meilleure) d'un taux "$X\%$" à la solution fournie par l'heuristique d'évaluation. Un écart négatif signifie que "LocFirst (V2)" fournit une solution plus mauvaise. Ensuite, nous évaluons "LocFirst (V2)" par rapport aux temps nécessaires pour le calcul des solutions. Durant cette évaluation, nous comparons "LocFirst (V2)" aux algorithmes FIFO, STMA, LTLM et FIFO + Locality. Enfin, nous comparons le degré le localité fourni par les différents algorithmes.
Les résultats de ces évaluations sont affichés dans les figures 5.3 à 5.5 et les tableaux 5.5 à 5.7.
Les figures 5.3 et 5.4 présentent les valeurs moyennes de C_{max} associées aux différents algorithmes et scénarios. Les scénarios sont ordonnés selon l'ordre croissant du degré de parallélisme pour la première figure et selon l'ordre croissant du nombre de tâches pour la deuxième. Pour la plupart des scénarios, l'heuristique "LocFirst (V2)" fournit les meilleures solutions. "LocFirst (V2)" fournit les meilleurs temps de calcul. Par exemple, dans le cas du scénario 20 (5130 tâches, 248 Vcores et 500 blocs de données), "LocFirst" trouve la solution en moyenne en environ 3 minutes et 26 secondes. Suivent les algorithmes FIFO, STMA, LTLM et FIFO + Locality qui nécessitent en moyenne 210, 280, 290 et 317 secondes.
Le tableau 5.5 et la figure 5.5 présentent les valeurs numériques des temps moyens de calcul associés aux différents algorithmes. Pour des scénarios avec des instances de petite taille (inférieure à 1000 tâches, 14 machines, 83 Vcores et 100 blocs de données), "LocFirst (V2)" n'est pas dominante, elle fournit de bons résultats dans la plupart des scénarios. Cependant, elle demande plus de temps de calcul que les algorithmes «STMA», «LTLM» et FIFO (en fonction des scénarios). A partir du scénario 8 (1095 tâches, 16 machines, 88 Vcores et 100 blocs), "LocFirst (V2)" fournit le temps de calcul le plus faible et des solutions meilleures que les solutions fournies par les autres algorithmes.
De ce fait, malgré les résultats acceptables que "LocFirst" fournit, nous pouvons dans un cadre général utiliser les algorithmes par défaut dans Hadoop, tel que FIFO, pour les petites instances.

À la fin de cette section, nous concluons que "LocFirst (V2)" fournit de bons résultats. À partir de certaines tailles d'instance, elle arrive à trouver les meilleurs ordonnancements. Notons que "LocFirst (V2)" fournit de meilleurs temps de calcul que ceux des autres algorithmes évalués.
La minimisation du critère C_{max} ne permet pas de réduire le temps d'attente avant l'exécution des travaux et ne permet pas de prendre en compte le poids des travaux lors de leur exécution. "LocFirst(V2)" a pour objectif la minimisation C_{max}, et elle peut être appliquée dans le cas où les travaux sont soumis par un seul utilisateur. Dans la prochaine section, nous abordons le cas où les travaux sont soumis par plusieurs utilisateurs possédant chacun un poids. Nous faisons évoluer nos travaux et nous considérons deux critères d'importance égale. Nous proposons les deux heuristiques "$ProxForMapReduce$" et "$MRSchedulingBasedOnCost$" pour résoudre ce nouveau problème.

5.4 Minimisation du critère $\sum w_j C_j + C_{max}$

Nous considérons dans cette section la minimisation de la somme de la date de fin de tous les travaux avec la somme pondérée des dates de fin des travaux ($\sum w_j C_j + C_{max}$). Nous cherchons à minimiser les encours totaux (pondérés) et la date de fin de tous les travaux. C'est-à-dire, minimiser le temps d'attente moyen pondéré pour qu'un travail ne reste pas longtemps dans la plateforme Hadoop. Nous présentons deux heuristiques et nous les évaluons.
Les heuristiques que nous présentons travaillent en trois étapes : *(i)* la première consiste à définir l'ordre de prise en compte des tâches, *(ii)* la deuxième consiste à affecter les tâches sur les machines et *(iii)* la troisième consiste à définir leur ordre d'exécution sur ces machines.
Nous supposons que chaque utilisateur a un poids. L'ensemble des travaux que ce dernier soumet hérite ce poids. La première heuristique que nous proposons est caractérisée par la prise en compte des distances entre les machines lors de l'affectation des tâches. Alors que la deuxième heuristique dépend d'un coût que l'on définit pour chaque travail et de la capacité de la grappe à exécuter les différents travaux en fonction de leur coût. Les travaux qui sont ordonnancés sont les travaux dont le coût est inférieur à la capacité global disponible sur la grappe.

5.4.1 Présentation de l'heuristique "Proximity-aware resource allocation for Map and Reduce Tasks" (ProxForMapReduce)

"ProxForMapReduce" est une heuristique Glouton inspirée des algorithmes FIFOet «MS» [72]. Elle commence par le calcul de la priorité de chaque travail à travers la fonction $PriorityDefinition$. La priorité d'un travail dépend d'une part du poids et des durées des tâches qui le constituent et, d'autre part, de sa taille. La taille d'un travail est le nombre de tâches qui le composent.
Ensuite, le choix glouton consiste à choisir chaque itération, la tâche de priorité la plus élevée et l'affecte à une machine. Elle commence par les machines les plus proches des données que la tâche traite. La machine choisie doit respecter : *(1)* les contraintes sur les quantités de ressources et *(2)* la contrainte sur la date de début d'exécution au plus tard $AverageEndDate$ que nous définissons dans l'algorithme 11. Enfin, l'heuristique définit l'ordre d'exécution des tâches sur chaque machine, ces tâches sont affectées selon l'ordre décroissant de leurs priorités calculées à

travers l'algorithme 12. La recherche de la machine sur laquelle la tâche sera affectée n'impose pas le respect du critère de localité. Elle consiste à trouver, pour chaque tâche, les machines qui se trouvent à une certaine distance d'une machine contenant les données qu'elle traite.
La valeur de *AverageEndDate* définit la date d'ordonnancement au plus tard d'une tâche. Si la tâche est de type "Map", *AverageEndDate* est la moyenne des dates de disponibilité des Vcores de type "Map". Si la tâche est de type "Reduce", *AverageEndDate* est la moyenne des dates de disponibilité des Vcores de type "Reduce".
La fonction *PriorityDefinition* calcule la priorité de chaque travail et les priorités des tâches qui lui sont associées. L'ensemble des tâches est trié en fonction de ses priorités.

Algorithme 11 : Algorithme de l'heuristique "*ProxForMapReduce*"

Données :
1 $Lmachines \leftarrow \{m\}, \forall m = 1...M$ /* La liste des machines */
2 $DistanceMach[M][M]$ /* Matrice des distances entre les machines */
3 $LdataBloc \leftarrow \{b\}, \forall b \in A^b$ /* La liste des blocs */
4 $LJobs \leftarrow \{i\}, \forall i = 1...W$ /* La liste des travaux */
5 $Ltasks \leftarrow \{i\}, \forall i = 1...N$ /* La liste des tâches */
Résultat : Les tâches de Ltasks sont ordonnancées sur les machines de la liste Lmachines
/* *Cond*1 : la machine $Lmachines[m]$ a les ressources pour exécuter i */
6 $Ltasks \leftarrow$ PriorityDefinition(*LJobs,Ltasks*) ;
7 **pour chaque** *($i \in Ltasks$)* **faire**
8 **si** *($Ltasks[i]$ est un Map)* **alors**
9 $AverageEndDate \leftarrow \frac{\sum_{m=1}^{M} Lmachines[m].LastDateMap}{M}$;
10 $FirstDate = 0$;
11 **sinon**
12 $AverageEndDate \leftarrow \frac{\sum_{m=1}^{M} Lmachines[m].LastDateReduce}{M}$;
13 $FirstDate = w.EndMapDate$; /* tel que $Ltasks[i] \in E_w$ */
14 $ListMachLoc \leftarrow$ ReturnMachProximity(*Ltasks[i], LdataBloc, DistanceMach, AverageEndDate*) ;
15 $NbProxMachines \leftarrow |ListMachLoc|$; /* nombre de machines de $ListMachLoc$ */
16 **si** *($NbProxMachines \neq 0$ et ($Ltasks[i].machine$ n'est pas défini))* **alors**
/* Vérifier si la tâche n'est pas affectée à une machine */
17 *mach* ←FindMachine(*Lmachines, ListMachLoc, NbProxMachines, Ltasks[i], AverageEndDate,FirstDate*) ;
18 ExecuteTask(*mach, Ltasks[i], AverageEndDate, FirstDate*) ;
19 **sinon**
20 **si** *($NbProxMachines == 0$ et ($Ltasks[i].machine$ n'est pas défini)* **alors**
21 *mach* ← FindMachine(*Lmachines, Lmachines, NbProxMachines, Ltasks[i], AverageEndDate,FirstDate*) ;
22 ExecuteTask(*mach, Ltasks[i], AverageEndDate, FirstDate*) ;

Algorithme 12 : Algorithme de la fonction "PriorityDefinition"

Données :

$LJob \leftarrow \{w'\}, \forall w' = 1...W$ /* La liste des travaux */

$Ltasks \leftarrow \{i\}, \forall i = 1...N$ /* La liste des tâches */

Résultat : Les tâches de Ltasks sont ordonnées selon leur priorité

$HighestPriority = -1$; /* HighestPriority : la priorité la plus élevé */

pour chaque w' $\in LJob$ **faire**

  $SomTaskPriority \leftarrow \sum_{i \in E^{w'}} \frac{Weight_{w'}}{p_i}$;

  $LJob[w'].priority \leftarrow \frac{SomTaskPriority}{|E^{w'}|}$; /* $|E^{w'}|$: nombre de tâches de w' */

  si $HighestPriority < LJob[w'].priority$ **alors**

    $HighestPriority = LJob[w'].priority$;

pour chaque w' $\in LJob$ **faire**

  pour chaque $i \in E^{w'}$ **faire**

    si $Ltasks[i]$ *est en cours d'exécution* **alors**

      $Ltasks[i].priority = HighestPriority + 1$;

    sinon

      si $Ltasks[i]$ *est une tâche Map* **alors**

        $Ltasks[i].priority = LJob[w'].priority$;

      sinon

        $Ltasks[i].priority = LJob[w'].priority - 1$;

$Sort(Ltasks)$; /* Le tri est décroissant en fonction des priorités des tâches */

Algorithme 13 : Algorithme de la fonction *ReturnMachProximity*

Données :

$Task$ /* La tâche à ordonnancer */

$LdataBloc \leftarrow \{b\}, \forall b \in A^b$ /* La liste des blocs */

$DistanceMach[M][M]$ /* Les distances entre les machines */

$AverageEndDate$ /* La moyenne des dates de fin des tâches par machine */

Résultat : Les machines sur lesquelles la tâche $Task$ peut être exécutée

$ProxIndex \leftarrow 4$; /* La distance autorisée entre deux machines */

si *(Task est une tâche Map)* **alors**

  pour chaque *(* $ind \in [1 \dots r_{(B_{(Task)})}]$*)* **faire**

    $MachineProx \leftarrow$ DichotomicSearch(*DistanceMach[D(B(Task))[ind]][M],ProxIndex,AverageEndDate*) ;

    /* D et B sont définies dans le tableau 1.2 */

si *(Task est une tâche Reduce)* **alors**

  pour chaque *(* $i \in [1 \dots n_{Task}]$*)* **faire**

    $MachineProx \leftarrow$ DichotomicSearch(*DistanceMach[i.machine][M],ProxIndex,AverageEndDate*) ;

RetournerMachineProximit(*MachineProx*) ;

Algorithme 14 : Algorithme de la fonction *FindMachine*

Données :

1 $Lmach \leftarrow \{m\}, \forall m = 1...M$ /* La liste des machines */
2 $LmachProx \leftarrow \{m\}, \forall m = 1...M$ /* La liste des machines prioritaires */
3 $NbProxMachines$ /* Le nombre d'éléments de $LmachProx$ */
4 $Task$ /* La tâche à ordonnancer */
5 $AverageEndDate$ /* Date d'ordonnancement au plus tard de $Task$ */
6 $FirstDate$ /* Date à partir de laquelle $Task$ peut être ordonnancée */

Résultat : La machine sur laquelle la tâche $Task$ peut être exécutée

/* $Cond1$: la machine $TempMachList[m]$ a les ressources pour exécuter $Task$ */
/* $Cond2$: $TempMachList[m].LastDateMap < AverageEndDate$ */
/* $Cond3$: $TempMachList[m].LastDateReduce < AverageEndDate$ */
/* $Cond4$: $TempMachList[m].LastDateMap < TempMachList[m-1].LastDateMap$ */
/* $Cond5$: $TempMachList[m].LastDateReduce < TempMachList[m-1].LastDateReduce$ */
/* $Cond6$: $TempMachList[m]$ ne fait pas partie des machines de $LmachProx$ */
/* $Cond7$: $TempMachLis[m].LastDateReduce \geq FirstDate$ */

7 **si** $NbProxMachines > 0$ **alors**
8 $TempMachList \leftarrow LmachProx$;
9 **sinon**
10 $TempMachList \leftarrow Lmach$;
11 **si** $NbProxMachines > 0$ **alors**
12 $Solution \leftarrow TempMachList[1]$; /* Initialisation d'une variable temporaire */
13 **pour chaque** ($m \in [2 \dots NbProxMachines]$) **faire**
14 **si** ($Cond1$ et $Cond2$ et $Cond4$ et $Task$ est un Map) **alors**
15 $Solution \leftarrow LmachProx[m]$;
16 **si** ($Cond1$ et $Cond3$ et $Cond5$ et $Cond7$ et $Task$ est un $Reduce$) **alors**
17 $Solution \leftarrow LmachProx[m]$;
18 **si** ($Solution \neq \varnothing$) **alors**
19 RetournerMachine($Solution$) ;
20 $Solution \leftarrow \emptyset$; /* Initialisation d'une variable temporaire */
21 **pour chaque** ($ind \in [1 \dots M]$) **faire**
22 **si** ($Cond1$ et $Cond2$ et $Cond6$ et $Task$ est un Map) **alors**
23 $Solution \leftarrow Lmach[m]$;
24 **si** ($Cond1$ et $Cond3$ et $Cond6$ et $Cond8$ et $Task$ est un $Reduce$) **alors**
25 $Solution \leftarrow Lmach[m]$;
26 RetournerMachine($Solution$) ;

La fonction de définition de l'ordre de séquencement des tâches

L'algorithme 12 est associé à la fonction *PriorityDefinition*. Il commence par le calcul de la priorité de chaque travail (lignes 6). Cette dernière est égale à la moyenne de la division du

Algorithme 15 : Algorithme de la fonction "ExecuteTask"

Données :

1 $Mach$ /* La machine sur laquelle la tâche doit être ordonnancée */
2 $Task$ /* Tâche à ordonnancer */
3 $AverageEndDate$ /* Date de début d'exécution au plus tard de $Task$ */
4 $FirstDate$ /* Date de début d'exécution au plus tôt de $Task$ */

Résultat : La tâche $Task$ est ordonnancée sur un Vcore de la machine $Mach$

/* $Cond1$: le Vcore "s" est libre après $FirstDate$ et avant $AverageEndDate$ */
/* $Cond2$: $Mach$ contient les ressources mémoire, disque dur et réseau après $FirstDate$ et avant $AverageEndDate$ */
/* $Cond3$: $Mach.LastDateofVcores[s] < Mach.LastDateofVcores[s-1]$ */

5 **si** *$Task$ est une tâche Map* **alors**
6 $TempSlot \leftarrow m^{Sr}_{Mach.num} + 1$;
7 $TaskTempStart = HV$; /* HV : grande valeur numérique */
8 $AvarageVcoresDate = 0$;
9 **pour** *s de $m^{Sr}_{Mach.num} + 2$ à $m^{s}_{Mach.num}$* **faire**
10 **si** *$Cond1$ et $Cond2$ et $Cond3$* **alors**
11 $TempSlot = s$;
12 $TaskTempStart = Mach.LastDateofVcores[s]$;
13 $Mach.LastDateofVcores[s] += p_{Task.num}$;
14 $AvarageVcoresDate += Mach.LastDateofSlot[s]$;
15 **si** *$Task$ est une tâche Reduce* **alors**
16 $TempSlot \leftarrow 1$;
17 **pour** *s de 2 à $m^{Sr}_{Mach.num}$* **faire**
18 **si** *$Cond1$ et $Cond2$ et $Cond3$* **alors**
19 $TempSlot = s$;
20 $TaskTempStart = Mach.LastDateofSlot[s]$;
21 $Mach.LastDateofVcores[s] += p_{Task.num}$;
22 $AvarageVcoresDate += Mach.LastDateofVcores[s]$;
23 $Task.machine = Mach.num$;
24 $Task.Vcore = TempSlot$;
25 $Task.start = TaskTempStart$;
26 $Task.End = Task.start + p_{Task.num}$;
27 $Mach.Vcore[TempSlot] = Task.num$;
28 **si** *$Task$ est un Map* **alors**
29 $Mach.LastDateMap = \frac{AvarageVcoresDate}{m^{s}_{Mach.num} - m^{Sr}_{Mach.num}}$;
30 **sinon**
31 $Mach.LastDateReduce = \frac{AvarageVcoresDate}{m^{Sr}_{Mach.num}}$;
32 `UpdateResourceOnMachine` ();

poids de chaque tâche de ce travail par sa durée (lignes 5). Les priorités des tâches récemment soumises sont affectées de la façon suivante : la priorité d'un travail est affectée à toutes les tâches Map qui lui sont associées ; les tâches Reduce auront une priorité inférieure à la priorité du travail d'une valeur de 1. Enfin, l'algorithme se termine avec le trie décroissant de la liste des tâches en fonction des priorités calculées.

La fonction de recherche de machine parmi les machines du voisinage

L'algorithme 13 est celui de la fonction $RetournerMachProximity$. Pour chaque tâche Map, il cherche les 20 premières machines qui ont une distance inférieure à 4 d'une machine contenant une réplication du bloc de données. Si la tâche est de type "Reduce", l'algorithme cherche des machines proches (de distance égale à 4) aux machines qui ont exécuté les tâches "Map" qui la précèdent. L'idée est de chercher une machine parmi les machines de voisinage pour limiter l'utilisation de la bande passante dû à la migration des données. Si parmi les machines du voisinage, il n'y a pas de machine disponible pour exécuter la tâche avant $AverageEndDate$, la recherche d'une machine qui peut exécuter une tâche s'effectue à l'aide de la fonction "$FindMachine$". Cette dernière cherche une machine parmi toutes les machines de la grappe.

La fonction de recherche de machine parmi Toutes les machines de la grappe

L'algorithme 14 est associé à la fonction $FindMachine$. Cette dernière vérifie si parmi les machines de voisinage, il existe une machine qui peut exécuter la tâche. Sinon, Chaque tâche "Map" est affectée à la première machine de la grappe qui se libère avant $AverageEndDate$. Pour chaque tâche "Reduce", l'algorithme 14 lui affecte la première machine qui se libère après $FirstDate$ et avant $AverageEndDate$.

La fonction d'exécution des tâches par machine

L'algorithme 15 est celui de la fonction "$ExecuteTask$". À la fin de l'exécution cette fonction, la tâche $Task$ est affectée à un Vcore de la machine $Mach$ en respectant les contraintes temporelles et de ressources. Si la tâche est une tâche "Map", la fonction $ExecuteTask$ cherche entre les dates $FirstDate = 0$ et $AverageEndDate$ le Vcore ("Map") qui se libère au plus tôt et qui est capable d'exécuter $Task$ en respectant les contraintes de ressources. Si la tâche est une tâche "Reduce", la fonction $ExecuteTask$ cherche entre les dates $FirstDate$ et $AverageEndDate$ le Vcore ("Reduce") qui se libère au plus tôt et qui est capable d'exécuter $Task$ en respectant les contraintes de ressources. Lors de l'ordonnancement d'une tâche sur une machine, la fonction $ExecuteTask$ ne remet pas en cause les anciens ordonnancements sur cette machine. En fonction du type de la tâche : *(i)* la variable $TempSlot$ est utilisée pour enregistrer le Vcore capable d'exécuter la tâche $Task$ au plus tôt. *(ii)* La variable $TaskTempStart$ est utilisée pour calculer la date de début de $Task$. *(iii)* la variable $AvarageVcoresDate$ est utilisée pour mettre à jour la date de fin moyenne des dernières tâches "Map" (respectivement les tâche "Reduce") sur les Vcores "Map" (respectivement "Reduce") de la machine $Mach$.

Algorithme 16 : Algorithme de l'heuristique "MRSchedulingBasedOnCost"

Données :

1 $Lmachines \leftarrow \{m\}, \forall m = 1...M$ /* La liste des machines */
2 $LJobs \leftarrow \{i\}, \forall i = 1...W$ /* La liste des travaux */
3 $Ltasks \leftarrow \{i\}, \forall i = 1...N$ /* La liste des tâches */

Résultat : Les tâches de Ltasks sont ordonnancées sur les machines de la liste Lmachines

/* Cond1: $Ltasks[i]$ est une tâche Map */
/* Cond2: $Ltasks[i]$ est une tâche Reduce */
/* Cond3: $Ltasks[i].priority < TotalClusterCapacity * 0.1$ */
/* Cond4: $Lmachines[j].LastDateMap > AverageEndDateForMap$ */
/* Cond5: $Lmachines[j].LastDateReduce > AverageEndDateForReduce$ */
/* Cond6: $l.End > FirstDate$ */

4 $TotalClusterCapacity \leftarrow$ PriorityDefinition3($LJobs$,$Ltasks$,$Lmachines$) ;
/* $ListJobToSchedule$: ensemble des travaux qui peuvent accéder à la grappe en respectant le rapport entre les coûts des travaux et la capacité de la grappe */
5 **tant que** ExistJobToSchedule *(LJobs,TotalClusterCapacity, ListJobToSchedule)* **faire**
6 **pour chaque** $w' \in ListJobToSchedule$ **faire**
7 **pour chaque** $i \in E^{w'}$ **faire**
/* ComputeAverageDate: retourne la moyenne des dates de libération des Vcores "Map" et "Reduce" */
8 ComputeAverageDate($Lmachines, AverageEndDateForMap, AverageEndDateForReduce$) ;
9 $FirstDate = 0$;
10 **si** *Ltasks[i] est un Map* **alors**
11 AverageEndDate = AverageEndDateForMap ;
12 **sinon**
13 AverageEndDate = AverageEndDateForReduce ;
14 **pour chaque** $l \in E_{Ltasks[i].num}$ **faire**
15 **si** *Cond6* **alors**
16 $FirstDate = l.End$;
17 **si** *Cond1 et Cond3* **alors**
18 $MachLoc \leftarrow$ ReturnListMachineLocV2(*Ltasks[i], LdataBloc, Lmachines, AverageEndDate*) ;
19 **si** *Cond2 et Cond3* **alors**
/* Si le coût d'une tâche Reduce est faible, elle est affectée sur une machine qui a exécuté les tâches Map qui la précède */
20 $MachLoc \leftarrow$ ReturnListMachineLocV2(*Ltasks[i], -1 , Lmachines, AverageEndDate*) ;
21 **si** *MachLoc* **alors**
22 ExecuteTask(*MachLoc, Ltasks[i], AverageEndDate, FirstDate*) ;
23 **sinon**
24 $Mach \leftarrow$ RetounerListeMachineDisV2(*Lmachines, Ltasks[i], AverageEndDate*) ;
25 ExecuteTask(*Mach, Ltasks[i], AverageEndDate, FirstDate*) ;

```
27 ComputeAverageDate(Lmachines, AverageEndDateForMap, AverageEndDateForReduce);
28 ListStressedMach ← ∅;
29 pour j de 1 à M() faire
30     si Cond4 ou Cond5 alors
31         ListStressedMach ← Lmachines[j];
32 NbStressedMach = |ListStressedMach|;        /* Nombre d'éléments de la liste  */
33 si NbStressedMach < 0.25 * M() alors
34     pour j de 1 à NbStressedMach faire
35         LTaskOnMachine ← ListOfTasksOnMachine(Ltask, ListStressedMach[j]);
36         NbTaskOnMachine ← |LTaskOnMachine|;
37         si NbTaskOnMachine > 1 alors
38             SecondChanceToOptimize(ListStressedMach, LJobs, LTaskOnMachine,
               NbTaskOnMachine);
```

5.4.2 Présentation de l'heuristique "*MRSchedulingBasedOnCost*"

La deuxième heuristique est nommée "MapReduce Scheduling based on the management of the cost of jobs" (MRSchedulingBasedOnCost). Elle est inspirée des algorithmes «DRF» [63] et «FS»[169]. Ensuite, un travail n'est autorisé à être exécuté que lorsque son coût est inférieur à la capacité disponible sur la grappe.
À chaque fois qu'un travail "X" est autorisé à utiliser la grappe, le niveau de capacité totale disponible sur la grappe diminue d'une valeur égale au coût de "X". Quand les tâches associées à ce travail terminent leur exécution, la capacité de la grappe augmente d'une valeur égale au coût de "X". Les travaux qui engendrent un coût faible sont ordonnancés en tenant compte de la localité des données qu'ils traitent. Sinon, ils sont ordonnancés sans tenir compte de ce critère. Après la fin de l'ordonnancement, les dates de fin d'exécution des machines sont analysées. Dans certain cas, les tâches Reduce qui s'exécutent sur les machines les plus en retard sont réordonnancées. L'objectif est de réduire la date de fin des travaux.
L'algorithme 16 est celui de l'heuristique "MRSchedulingBasedOnCost". Il commence par le calcul des coûts des travaux et des priorités des tâches (ligne 4). Ensuite, il récupère (lignes 5) la liste des travaux qui peuvent utiliser la grappe en même temps. Si le coût d'un travail est faible, on l'affecte aux machines en tenant compte de la localité des données (ligne 18 et 20). Si le coût d'un travail est élevé, il est affecté sans tenir compte de la localité des données (ligne 24). La fonction "*ExecuteTask*" exécute la tâche sur la machine choisie. Après l'obtention des premiers résultats, l'étape suivante consiste à étudier le décalage entre les dates de libération des machines. Si ce décalage est élevé, l'ordonnancement de certaines tâches "Reduce" est mis en cause. Nous détectons que le décalage est élevé de la façon suivante : nous calculons la moyenne des dates de libération des machines "*AverageEndDate*". Lorsque le nombre de machines, dont la date de libération est supérieur à "*AverageEndDate*", est inférieur à 25 % de M, nous considérons que le décalage est élevé. Dans ce cas, l'heuristique récupère ses machines (de la ligne 28 à la ligne 31)et réordonnance les tâches "Reduce" de façon à privilégier la réduction des dates de fin des travaux.

Algorithme 17 : Algorithme de la fonction "PriorityDefinition3"

Données :

1 $LJobs \leftarrow \{i\}, \forall i = 1...W$ /* La liste des travaux */

2 $Ltasks \leftarrow \{i\}, \forall i = 1...N$ /* La liste des tâches */

3 $Lmachines \leftarrow \{m\}, \forall m = 1...M$ /* La liste des machines */

Résultat : La capacité de la grappe (TotalClusterCapacity) et les priorités des travaux soumis sont définies

4 $Cost_{Map} = 0,45 * \sum_j m_j^{Sm} + 0,35 * \sum_j m_j^{ra} + 0,2 * \sum_j m_j^h$;

5 $Cost_{Reduce} = 0,3 * \sum_j m_j^{Sr} + 0,6 * \sum_j m_j^{ra} + 0,1 * \sum_j m_j^h$;

6 $TotalClusterCapacity = Cost_{Map} + Cost_{Reduce}$;

7 **pour** *(w' de 1 à W)* **faire**

8 $LJobs[w'].cost = \frac{0,45 * LJobs[w'].NbTaskMap + 0,3 * LJobs[w'].NbTaskReduce}{2 * LJobs[w'].Weight} + \frac{\sum_j m_j^s}{LJobs[w'].NbTaskMap + LJobs[w'].NbTaskReduce}$;

9 $HighestPriority \leftarrow HV$;

10 **pour** *w' de 1 à W* **faire**

11 **pour chaque** $i \in E^{w'}$ **faire**

12 **si** *Ltasks[i] est une tâches Map* **alors**

13 $Ltasks[i].priority \leftarrow \frac{LJobs[w'].cost}{p_i}$;

14 **sinon**

15 $Ltasks[i].priority \leftarrow \frac{LJobs[w'].cost}{p_i} - 1$;

16 **si** $HighestPriority < Ltasks[i].priority$ **alors**

17 $HighestPriority = Ltasks[i].priority$;

18 **pour** *i de 1 à N* **faire**

19 **si** *Ltasks[i] est en cours d'exécution* **alors**

20 $Ltasks[i].priority = HighestPriority + 1$;

21 $Sort(Ljobs)$;

22 `Return` (TotalClusterCapacity) ;

La fonction de définition de l'ordre de séquencement des tâches

L'algorithme 17 est associé à la fonction "$PriorityDefinition3$". L'idée est que les travaux les moins couteux sont exécutés en premier et que les tâches les plus prioritaires de ces travaux sont exécutées en premier. Il commence par le calcul de la capacité de la grappe (de la ligne 4 à la ligne 6) qui est la somme des coûts associés aux ressources exploitées par les tâches. Le coût de chaque type ("Map" ou "Reduce") est égal à la somme pondérée des trois ressources (Vcores, mémoire, disque dur). Ensuite, on calcule les coûts des travaux. Pour cette heuristique, la priorité d'un travail représente son coût : elle dépend du nombre de tâches qui le composent et de son poids (entre les lignes 7 et 8). La troisième étape consiste à calculer les priorités des tâches

(entre les lignes 10 et 17). La quatrième étape consiste à affecter la priorité la plus élevé aux tâches associées aux travaux qui sont en cours d'exécution lorsque le calcul de l'ordonnancement a commencé (entre les lignes 18 et 20). Enfin, les tâches sont triées de façon décroissante selon les priorités définies et les travaux sont triés de façon croissante selon les coûts qu'ils engendrent (ligne 21).

Algorithme 18 : Algorithme de la fonction "ExistJobToSchedule"

Données :

1 $LJobs \leftarrow \{i\}, \forall i = 1...W$ /* La liste des travaux */

2 $TotalClusterCapacity$ /* Capacité globale de la grappe */

3 $ListJobToSchedule$ /* Ce paramètre sert à retourner la liste des travaux autorisés à utiliser la grappe */

Résultat : Retourne 1 s'il y a de nouveaux travaux qui peuvent utiliser la grappe et la liste de ces travaux dans $ListJobToSchedule$. 0, sinon.

/* Cond1: $LJobs[compteur].scheduled == 0$, le travail n'a pas utilisé la grappe */

/* Cond2: $LJobs[compteur].cost \leq TotalClusterCapacity$, le travail est autorisé à utiliser la grappe */

4 $SizeList = 0$;
5 $compteur \leftarrow W$;
6 **tant que** $compteur\ != 0$ **faire**
7 **si** *Cond1 et Cond2* **alors**
8 $ListJobToSchedule \leftarrow LJobs[compteur]$;
9 $TotalClusterCapacity = TotalClusterCapacity - LJobs[compteur].cost$;
10 $compteur\ --$;
11 **si** $SizeList == |ListJobToSchedule|$ **alors**
12 Return (0) ;
13 **sinon**
14 Return (1) ;

La fonction de recherche des travaux à ordonnancer

L'algorithme 18 vérifie s'il y a des travaux qui peuvent accéder à la grappe. Il retourne l'ensemble des travaux dont la somme des coûts qu'ils engendrent est inférieure à la capacité disponible sur la grappe. En plus, cet algorithme retourne une valeur binaire pour indiquer s'il y a ou non des travaux non encore ordonnancés.

La fonction de mise en cause des premiers résultats d'ordonnancement

Lorsque moins de $\frac{M}{4}$ machines sont libres après "*AverageEndDate*", l'heuristique "MRSchedulingBasedOnCost" ajuste l'ordonnancement de certains tâches "Reduce". Nous décrivons l'algorithme 19 associé à la fonction "*SecondChanceToOptimize*". Son Objectif est de fournir un ordonnancement meilleur de l'ensemble des tâches "Reduce" qui tournent sur cette machine.

Algorithme 19 : Algorithme de la fonction "SecondChanceToOptimize"

Données :

1 $StressedMach$ /* Une machine */
2 $LJobs \leftarrow \{i\}, \forall i = 1...W$ /* La liste des travaux */
3 $LTaskOnMachine \leftarrow \{i\}, \forall i = 1...N$ /* Liste des tâches sur $StressedMach$ */
4 $NbTaskOnMachine$ /* Nombre de tâches dans $LTaskOnMachine$ */

Résultat : La capacité de la grappe et les priorités des travaux soumis sont définies

/* Cond1: $StressedMach$ a les ressources pour exécuter $LtaskTemporary[i]$ à partir de t_{start} */
/* Cond2: $StressedMach$ a les ressources pour exécuter $LtaskTemporary[i]$ avant t_{limit} */
/* Cond3: $StressedMach.LastDateofVcores[s] < Temp_{V}arStart$ */
/* Cond4: $LTaskOnMachine[i].AllowStart[s] \leq StressedMach.LastDateofVcores[s]$ */

5 $t_{start} = HV$;
6 $LtaskIterate \leftarrow LTaskOnMachine$; /* Liste pour contenir les tâches non encore ordonnancées */
7 **pour** *i de 1 à NbTaskOnMachine* **faire**
8 $LTaskOnMachine[i].AllowStart = LJobs[LTaskOnMachine[i].Job].EndMapDate$;
9 $LTaskOnMachine[i].StrictEnd = LJobs[LTaskOnMachine[i].Job].EndReduceDate$;
10 **si** $LTaskOnMachine[i].AllowStart < t_{start}$ **alors**
11 $t_{start} = LTaskOnMachine[i].AllowStart$;

/* Trie croissant en fonction de la valeur de $LTaskOnMachine[i].StrictEnd - (LTaskOnMachine[i].AllowStart + p_{LTaskOnMachine[i]})$. "$i$" indice sur les tâches */
12 Sort($LtaskIterate$) ;
13 $t_{limit} \leftarrow t_{start} + 1.5 *$ LargerTaskDuration($LTaskOnMachine$) ;
14 $LtaskTemporary \leftarrow$ ListTaskTemporary($LTaskOnMachine, t_{limit}, NbLtaskTemp$) ;
15 **tant que** NotScheduled *(LtaskTemporary)* **faire**
16 $TempVcore1 = -1$; /* Variable qui contient le Vcore pour exécuter la tâche */
17 $Temp_{V}arStart = HV$; /* Variable qui contient la date de début d'ordonnancement de la tâche */
18 **pour** *i de 1 à NbLtaskTemp* **faire**
19 **pour** *s de 1 à* $m^{Sr}_{StressedMach.num}$ **faire**
20 **si** *Cond1* **alors**
21 **si** *Cond2* **alors**
22 $TempVcore1 = s$; /* Ordonnancer la tâche à partir de t_{start} et avant t_{limit} */
23 ScheduleTaskOn($LtaskIterate[i], TempVcore, t_{start}$) ;
24 **sinon**
/* Ordonnancer la tâche après t_{start} sans influencer négativement la fonction à optimiser */
25 **si** *Cond3 et Cond4 et* !$DelayOBJ(LtaskIterate[i], LJobs)$ **alors**
26 $TempVcore1 = s$;
27 $Temp_{V}arStart = StressedMach.LastDateofVcores[s]$;
28 ScheduleTaskOn($StressedMach, LtaskIterate[i], TempVcore1, Temp_{V}arStart$) ;
29 $t_{start} \leftarrow$ FirstDateTemporary($StressedMach, t_{start}$)
30 $t_{limit} \leftarrow t_{start} + 1.5 *$ LargerTaskDuration($LtaskIterate$) ;
31 $LtaskTemporary \leftarrow$ ListTaskTemporary($LtaskIterate, t_{limit}, NbLtaskTemp$) ;

Un ordonnancement est jugé meilleur s'il améliore la date de fin d'exécution d'au moins un travail sans pénaliser les autres ou s'il n'influence pas négativement les objectifs que nous optimisons.
La fonction "*SecondChanceToOptimize*" commence par initialiser une fenêtre de temps absolue ($[i.AllowStart, i.StrictEnd]$, $\forall i \in Ltasks$) pour l'ordonnancement de chaque tâche (lignes 8 et 9). Elle initialise une première valeur de t_{start}, qui est égale à la plus petite valeur de "$LTaskOnMachine[].AllowStart$" associée à une des tâches de "$LTaskOnMachine$".
Si nous notons "i" l'indice sur les tâches, la liste des tâches $LTaskOnMachine$ est ensuite triée de façon croissante en fonction de "$LTaskOnMachine[i].StrictEnd - (LTaskOnMachine[i].AllowStart + p_{LTaskOnMachine[i]})$". C'est-à-dire, en fonction
Ensuite, on initialise t_{limit} (ligne 13) tel que $[t_{start}, t_{limit}]$ soit une première fenêtre relative. L'étape suivante consiste à récupérer les tâches qui respectent les contraintes temporelles à l'aide de la fonction "*ListTaskTemporary*" (Par exemple, la liste des tâches qui doivent terminer leur exécution avant t_{limit}). S'il n'y a pas de tâche qui respecte cette condition, les $m^{Sr}_{StressedMach.num}$ tâches sont récupérées (dans $LtaskTemporary$) de la liste ordonnée $LTaskOnMachine$. Si possible, on affecte les tâches en essayant de ne pas retarder la fin des travaux (entre les lignes 20 et 23), sinon, on les affecte en essayant de minimiser les objectifs qu'on cherche à optimiser (entre les lignes 25 et 28).

Algorithme 20 : Algorithme de la fonction "ListOfTasksOnMachine"

Données :
1 $Ltasks \leftarrow \{i\}, \forall i = 1...N$ `/* La liste des tâches */`
2 $Machine$ `/* Une machine */`
Résultat : Retourner la liste des tâches qui s'exécutent sur $Machine$
3 $LResult \leftarrow \varnothing$;
4 **pour** i *de* 1 *à* $N()$ **faire**
5 **si** $Ltasks[i].machine == Machine.num$ **alors**
6 $LResult \leftarrow Ltasks[i]$;
7 `ReturnListOfTasks` (LResult) ;

L'algorithme 20 permet de récupérer la liste des tâches qui sont ordonnancées sur la machine passée dans le paramètre "*machine*".
L'algorithme 21 vérifie si parmi les tâches de la liste "$LTaskTemporary$" passée en paramètre, il existe une tâche non ordonnancée.
L'algorithme 22 récupère les tâches, non encore exécutées et qui doivent s'exécuter avant la date t_{limit}.

Les fonctions d'exécution des tâches par machine

L'algorithme 23 retourne la date à partir de laquelle une tâche peut être ordonnancée. Elle retourne la première date à partir de laquelle un Vcore est libre et peut exécuter une tâche.
La fonction "*DelayOBJ*" calcule la nouvelle valeur de la fonction objectif en supposant que la tâche "$LTaskTemporary[i]$" est exécutée sur la machine "s" à partir de la date "Stressed-

Mach.LastDateofVcores[s]". Si la valeur trouvée est inférieure à l'ancienne valeur, elle retourne 1 sinon elle retourne 0.
L'algorithme 24 est associé à la fonction "*ScheduleTaskOn*", elle ordonnance la tâche "*Task*" et met à jour la date de fin du travail associé.

Algorithme 21 : Algorithme de la fonction "NotScheduled"

Données :
1 $LTaskTemporary \leftarrow \{i\}, \forall i = 1...N$ /* La liste des tâches */
Résultat : Retourner 0 si toutes les tâches sont ordonnancées, 1 sinon

2 **pour** *i de* 1 *à* $N()$ **faire**
3 **si** $(LTaskTemporary[i].secondChance == -1)$ **alors**
4 Returner (1) ;
5 Returner (0) ;

Algorithme 22 : Algorithme de la fonction "ListTaskTemporary"

Données :
1 $LTaskTemporary \leftarrow \{i\}, \forall i = 1...N$ /* La liste des tâches */
2 t_{limit} /* Date limite de recherche */
3 $NbLtaskTemp$ /* Elle retourne le nombre d'éléments de "LTaskTemporary" */
Résultat : Retourner les tâches qui doivent se terminer avant t_{limit}

4 $LResult \leftarrow \varnothing$;
5 **pour** *i de* 1 *à* $N()$ **faire**
6 **si** $(LTaskOnMachine[i].StrictEnd < t_{limit}$ *et* $Ltasks[i].secondChance$ **alors**
7 $LResult \leftarrow LTaskOnMachine[i]$;
8 $NbLtaskTemp \leftarrow |LTaskTemporary|$ Returner (LResult) ;

Algorithme 23 : Algorithme de la fonction "FirstDateTemporary"

Données :
1 $StressedMach$ /* Une machine */
2 $VarTemp$ /* Date de début de recherche */
Résultat : Retourner une date à partir de laquelle on peut ordonnancer les tâches restantes

3 $Result \leftarrow HV$; /* HV: une grande valeur */
4 **pour** *s de* 1 *à* $m^{Sr}_{StressedMach.num}$ **faire**
5 **si** $StressedMach.LastDateofVcores[s] < Result$ *et* $VarTemp < Result$ **alors**
6 $Result = StressedMach.LastDateofVcores[s]$;
7 Returner (Result) ;

Algorithme 24 : Algorithme de la fonction "ScheduleTaskOn"

Données :

1 $Mach$ /* Une machine */
2 $Task$ /* La liste des tâches */
3 $Vcore$ /* Le Vcore où $Task$ doit s'exécuter */
4 $Date$ /* La date à partir de laquelle $Task$ doit s'exécuter */

Résultat : La tâche est ordonnancée sur "$Mach$"

5 $Task.secondChance = Mach.num$;
6 $Task.Vcore = Vcore$;
7 $Task.start = Date$;
8 $Task.End = Task.start + p_{(Task.num)}$;
9 $Mach.Vcores[Vcore] = Task.num$;
10 $Mach.LastDateofVcores[Vcore] = Task.End$;
11 $LTask \leftarrow LTask / \{Task\}$;
12 **si** $Task.End > LJobs[Task.Job].EndReduceDate$ **alors**
13 $\quad LJobs[Task.Job].EndReduceDate = Task.End$;

5.4.3 Analyse de complexité

Dans cette section, nous étudions les complexités des deux heuristiques "$ProxForMapReduce$" et "$MRSchedulingBasedOnCost$".

Analyse de complexité de "ProxForMapReduce"

L'algorithme "$ProxForMapReduce$" est de complexité de l'ordre de $\mathcal{O}(N(\log N + \log M + m_j^s))$.

Calcul :
L'heuristique "$ProxForMapReduce$" est composée de trois grandes parties :

- La fonction "$PriorityDefinition$" est utilisée pour la définition des priorités des travaux et tâches. Elle est de complexité $F1 = \mathcal{O}(max(N, N, N \log N)) = \mathcal{O}(N \log N)$.
- Le calcul de "$AverageEndDate$" est de complexité $F2 = \mathcal{O}(M)$.
- La fonction $RetournerMachProximity$ qui récupère pour chaque tâche Map ou Reduce les machines privilégiées pour l'exécuter. Elle est de complexité $F3 = \mathcal{O}(N^m)$.
- L'affectation des tâches aux machines est basée sur la fonction $FindMachines$, dont la complexité est égale à $F4 = \mathcal{O}(\log M)$.
- La fonction "$ExecuteTask$" est responsable de l'exécution de la tâche sur la machine. Elle est de complexité égale à $F5 = \mathcal{O}(max(m_j^{Sm}, m_j^{Sr}))$.

La complexité de l'heuristique $ProxForMapReduce$ est en conséquence $\mathcal{O}(N * (F1 + F2 + F3 + F4 + F5)) = \mathcal{O}(N * (\log N + M + N^m + \log M + max(m_j^{Sm}, m_j^{Sr}))) = \mathcal{O}(N * (\log N + \log M + max(m_j^{Sm}, m_j^{Sr})))$.

Analyse de complexité de "MRSchedulingBasedOnCost"

L'algorithme "$MRSchedulingBasedOnCost$" est de complexité $\mathcal{O}(MN(\log N + m_j^{Sr}))$

Calcul :
L'heuristique $MRSchedulingBasedOnCost$ est basée sur deux parties : la première calcule un ordonnancement des tâches sur la grappe. La deuxième met en cause certain décisions d'ordonnancement et cherche à améliorer le premier ordonnancement trouvé.
La première partie est de complexité $F20 = \mathcal{O}(N \log N + NMm_j^s)$. Les fonctions qui la composent ont les complexités suivantes :

- La fonction "$PriorityDefinition3$" définit les priorités des travaux et des tâches. Elle est de complexité $F6 = \mathcal{O}(W + 2N + N \log N)) = \mathcal{O}(N \log N)$. Car au minimum, $N == 2W$ (une tâche Map et une Reduce), Sinon, $N \gg W$ (N et W sont définies dans le tableau 1.1).
- la fonction "$ExistJobToSchedule$" est de complexité $F7 = \mathcal{O}(W)$.
- la fonction "$ComputeAverageDate$" est de complexité $F8 = \mathcal{O}(M)$.
- la fonction "$ReturnListMachineLocV2$" est de complexité $F9 = \mathcal{O}(N^m)$.
- la fonction "$ReturnListMachineDisV2$" est de complexité $F10 = \mathcal{O}(M)$.
- la fonction "$ExecuteTask$" est de complexité $F11 = \mathcal{O}(m_j^s)$.

La deuxième partie est de complexité $F21 = \mathcal{O}(2M + M(N + N \log N + Nm_j^{Sr})$. Les fonctions qui la composent ont les complexités suivantes :

- La vérification du nombre de machines est de complexité $F12 = \mathcal{O}(M)$.
- la fonction "$ListOfTasksOnMachine$" est de complexité $F13 = \mathcal{O}(N)$.
- la fonction "$SecondChanceToOptimize$" est de complexité majorée par $F14 = \mathcal{O}(N + N \log N + m_j^{Sr} + N + N * m_j^{Sr}) = \mathcal{O}(N \log N + N * m_j^{Sr})$:
  - la fonction "$NotScheduled$" est de complexité : $F15 = \mathcal{O}(N)$.
  - la fonction "$FistDateTemporary$" est de complexité : $F16 = \mathcal{O}(m_j^{Sr})$.
  - la fonction "$LargerTaskDuration$" est de complexité : $F17 = \mathcal{O}(m_j^{Sr})$.
  - la fonction "$ListTaskTemporary$" est de complexité : $F18 = \mathcal{O}(N)$.
  - la fonction "$ScheduleTaskOn$" est de complexité : $F19 = \mathcal{O}(1)$.

La complexité de l'heuristique $MRSchedulingBasedOnCost$ est en conséquence $\mathcal{O}(MN(\log N + m_j^{Sr}))$.

5.4.4 Évaluation expérimentale des heuristiques

Dans cette section, nous évaluons les deux heuristiques "ProxForMapReduce" et "MRSchedulingBasedOnCost". Rappelons que les critères utilisés dans l'évaluation sont la minimisation de la somme pondérée des dates de fin des travaux et de la date de fin de tous les travaux ($\sum w_j C_j + C_{max}$).

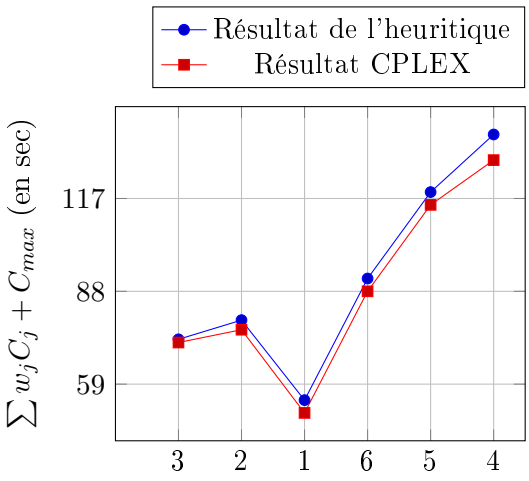
Ces deux heuristiques sont évaluées dans un contexte hors ligne sur de petites instances par comparaison aux résultats obtenus suite à l'évaluation du modèle présenté dans la section 4.3.2 du chapitre 4. Ensuite, elles sont évaluées sur des instances moyennes et grandes et dans un contexte en ligne. Nous les comparons aux heuristiques de la littérature qui sont LTLM, FS, STMA, FIFO, et FIFO avec prise en compte de la localité. Dans toutes les expériences, nous considérons 10 instances par scénario. Nous considérons le critère de la minimisation du $\sum w_j C_j + C_{max}$.
Vu le contexte d'utilisation des heuristiques, la temps de calcul (la durée nécessaire pour que l'heuristique trouve une solution) et la qualité des solutions obtenues fournissent une idée sur l'efficacité des heuristiques que nous présentons.
Mesurer la quantité des tâches qui s'exécutent en local revient à avoir une idée sur la capacité de l'heuristique à réduire l'utilisation de la bande réseau lors de la migration des blocs de données.
Lors de l'évaluation hors ligne, nous utilisons la même méthode que celle décrite dans la section 4.3.4 du chapitre 4. Nous utilisons les scénarios présentés dans le tableau 4.6. Lors de l'évaluation en ligne des heuristiques, nous utilisons la même méthode que celle décrite dans la section 5.3.3 pour la génération des données. Nous utilisons les scénarios affichés dans le tableau 5.2.
L'environnement de test est décrit dans la section 4.2.3 du chapitre 4.

Évaluation de "ProxForMapReduce"

Nous présentons dans cette section les résultats de l'évaluation de "*ProxForMapReduce*". Pour l'évaluation hors ligne de l'heuristique, nous synthétisons les résultats dans la figure 5.6, nous affichons les résultats détaillés dans le tableau 5.8. La première colonne présente les scénarios. Les deuxième et troisième colonnes présentent respectivement la moyenne des résultats

FIGURE 5.6 – Résultats de l'évaluation de "*ProxForMapReduce*" sur de petites instances dans un contexte hors ligne.

(a) Résultats de l'évaluation.



(b) Moyennes des degrés de localité par scénario

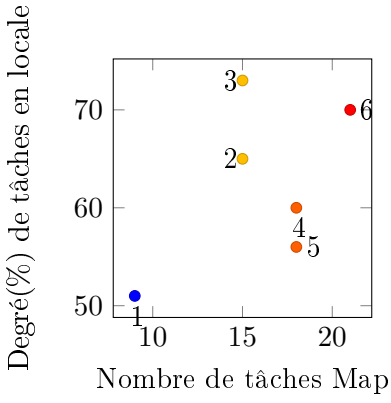


TABLE 5.8 – Écarts relatifs moyens (en %) entre la solution fournie par "ProxForMapReduce" et les solutions optimales calculées lors de la résolution du modèle mathématique

Scénarios	**Solution optimale** (calculée par CPLEX)	**Solution calculée par** "*ProxForMapReduce*"	Écart relatif (%)
1	50.3	54.2	7.5
2	76.1	78.4	3.0
3	71.8	72.9	1.5
4	129.1	137.1	6.2
5	115.0	119.5	3.9
6	88.4	92.1	4.2
Moyenne			4.4

(fonction objectif) calculés par CPLEX et la moyenne des résultats obtenus suite à l'exécution de "*ProxForMapReduce*". La figure 5.6a présente les résultats obtenus ordonnés selon le degré de parallélisme. La figure 5.6b présente la moyenne des degrés de localité selon les scénarios. Le degré de localité est le ratio entre le nombre de tâches Map qui s'exécutent sur les machines contenant les données qu'elles traitent et le nombre total de tâche Map. Remarquons que sur des petites instances, "*ProxForMapReduce*" présente un décalage entre 3% et 8% par rapport à la solution optimale trouvée lors de la résolution du modèle. Nous remarquons que les moyennes des degrés de localité pour les scénarios considérés sont supérieurs à 50%. Les temps moyen pour le calcul d'une solution par l'heuristique sont de l'ordre de microsecondes. Pour l'évaluation de l'heuristique en mode en ligne, nous présentons dans le tableau 5.9 la moyenne des résultats obtenus par scénarios et la moyenne des temps de calcul.
Le tableau 5.10 présente les écarts entre les résultats fournis par "*ProxForMapReduce*" et les autres heuristiques. L'heuristique que nous proposons résout les instances de 5000 tâches et 44 machines en environs 4 minutes. Ce qui représente environ 5% entre le temps de soumission du premier travail et la date de fin d'exécution du dernier travail. Le temps de calcul de la solution est un critère d'évaluation secondaire de nos heuristiques dans le contexte en ligne. Nous remarquons que pour des instances de petite et moyenne tailles, "*ProxForMapReduce*" ne fournit pas les meilleurs temps de calcul (depuis le scénario 6 jusqu'au scénario 15). Pour ces tailles d'instances, elle fournit des temps de calcul meilleurs que ceux fournis par «LTLM» et «STMA». Entre les scénarios 16 et 18, elle fournit des temps de calcul meilleurs que ceux fournis par «LTLM», «STMA» et «FS». À partir du scénario 19, "*ProxForMapReduce*" fournit les meilleurs temps de calcul. Vu l'importance du temps nécessaire pour le calcul de la solution, nous présentons dans la figure 5.7 la moyenne des temps de calcul associés aux différents algorithmes en fonction des scénarios. Ce qu'on doit retenir à ce niveau est que sur des instances de petite et moyenne tailles, on a un compromis entre la qualité de la solution et le temps de calcul. Cependant, pour des instances de grande taille, "*ProxForMapReduce*" fournit les meilleures solutions et les meilleurs temps de calcul.

TABLE 5.9 – Moyennes des résultats des évaluations de "*ProxForMapReduce*"

Sc.	**Temps de calcul**	$\sum w_j C_j + C_{max}$
1	9.4	22394.2
2	10.1	17371.3
3	20.8	33971.1
4	29.2	42449.4
5	43.9	37327.7
6	53.3	45939.1
7	61.1	47633.9
8	76.2	73283.3
9	98.5	79982.1
10	121.1	94839.1
11	131.9	100320.4
12	142.3	103967.8
13	158.1	112393.9
14	164.7	166030.1
15	181.8	178863.5
16	196.4	210854.3
17	201.3	295217.6
18	226.9	287436.2
19	241.3	439392.1
20	254.9	441345.7

TABLE 5.10 – Écarts relatifs moyens (en %) entre les solutions fournies par "*ProxForMapReduce*" (Heur1) et par les heuristiques de référence utilisées dans l'évaluation.

Sc.	«STMA»	Écart de "Heur1"	«LTLM»	Écart de "Heur1"	FIFO	Écart de "Heur1"	FIFO+ Locality	Écart de "Heur1"	«FS»	Écart de "Heur1"
1	25875.4	15.5	25356.3	13,2	24495.2	9.4	23945.2	6.9	23157.2	3.4
2	20397.2	17.4	19768.1	13.8	20190 ;3	16.2	19134.5	10.2	17934.4	3.2
3	36478.2	7.4	34933.8	2.8	34892.2	2.7	34716.3	2.2	33870.1	-0.3
4	38846.6	-8.5	41297.9	-2.7	36972.8	-12.9	37389.3	-11.9	39178.1	-7.7
5	40374.4	8.2	42045.2	12.6	40947.3	9.7	40047.1	7.3	39389.3	5.5
6	48928.4	6.5	48394.1	5.3	46488.9	1.2	47388.9	3.2	46256.4	0.7
7	49274.2	3.4	48792.1	2.4	48824.2	2.5	48393.7	1.6	48293.7	1.4
8	77283.3	5.5	75033.4	2.4	75824.4	3.5	76823.2	4.8	74194.2	1.2
9	84238.9	5.3	83581.9	4.5	83153.1	4	82073.3	2.6	81713.8	2.2
10	97384.1	2.7	98072.5	3.4	96835.8	2.1	96294.8	1.5	95193.1	0.4
11	103844.8	3.5	102852.3	2.5	102485.2	2.2	103284.2	3	102382.2	2.1
12	105936.9	1.9	105838.5	1.8	105382.1	1.4	104748.3	0.8	104836.9	0.8
13	127384.5	13,3	125837.6	12	113938.5	1,4	125384.1	11,6	123284.4	9,7
14	170380.5	2.6	157958.3	-4.9	168373.9	1.4	168834.4	1.7	168197.9	1.3
15	181293.3	1.4	180283.1	0.8	180936.8	1.2	179364.1	0.3	179173.1	0.2
16	216837.2	2.8	209837.8	-0.5	214384.3	1.7	214252.6	1.6	212373.8	0.7
17	299387.1	1.4	298934.3	1.3	300034.1	1.6	293314.3	-0.6	296902.4	0.6
18	304738.7	6,0	291838.5	1,5	300203.7	4,4	291339.1	1,4	319833.1	11,3
19	463912.6	5.6	471094.2	7.2	452049.5	2.9	511945.4	16.5	461930.3	5.1
20	450274.2	2.2	451894.3	2.4	455288.1	3.2	536924.8	21.7	453427.7	2.7

FIGURE 5.7 – Moyennes des temps de calcul de chacune des heuristiques

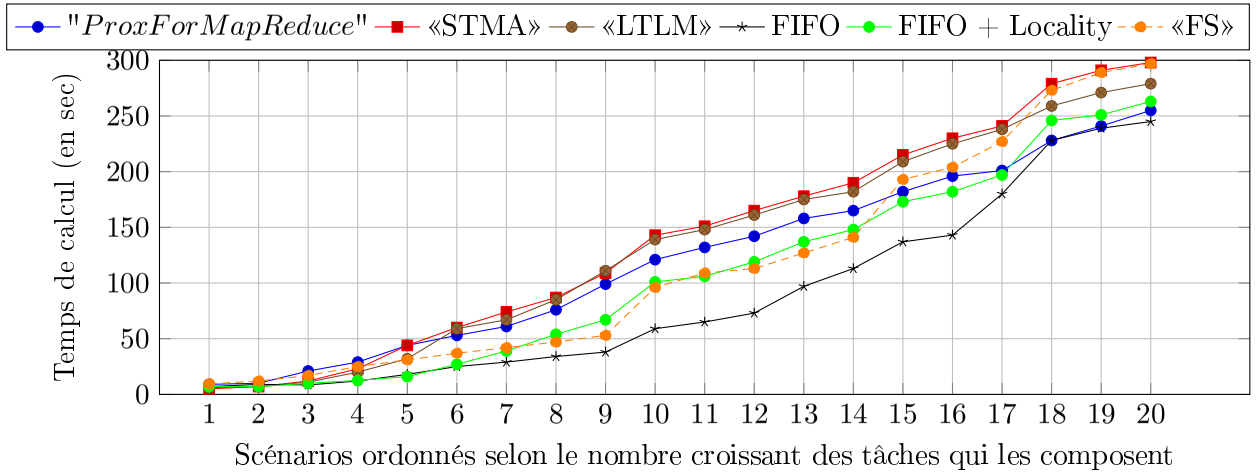


TABLE 5.11 – Écarts relatifs moyens (en %) entre les solutions fournies par "*MRSchedulingBasedOnCost*" et les solutions optimales

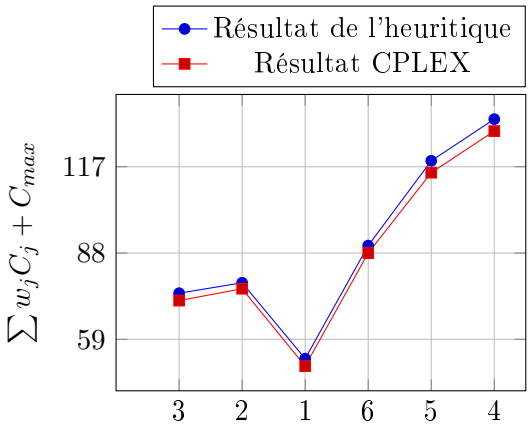
Scénarios	**Solution optimum** (calculée par CPLEX)	**Solution calculée par** "*MRSchedulingBasedOnCost*"	Écart relatif (%)
1	50.3	52.5	4.4
2	76.1	78.2	2.8
3	71.8	74.5	3.8
4	129.1	133.2	3.2
5	115.0	118.7	3.2
6	88.4	90.5	2.4
Moyenne			3.3

Évaluation de "MRSchedulingBasedOnCost"

Nous suivons la même méthode pour l'évaluation de "*MRSchedulingBasedOnCost*", en évaluant l'heuristique sur de petites instances et en mode hors ligne, puis en l'évaluant avec des instances moyennes et grandes en mode en ligne.
Les résultats de l'évaluation hors ligne sont synthétisés dans la figure 5.8 et dans le tableau 5.11. Ce dernier présente les écarts relatifs obtenus suite l'évaluation à "MRSchedulingBasedOnCost" par rapport à la solution calculée lors de la résolution du modèle mathématique. La moyenne des décalages entre les deux solutions est d'environ 3.5 %. La moyenne des taux associés aux tâches qui s'exécutent sur les machines en locale est égale à 70% pour les scénarios considérés.
Les résultats de l'évaluation en ligne de "*MRSchedulingBasedOnCost*" sont affichés dans les tableaux 5.12 et 5.13. Les temps de calcul sont affichés dans la figure 5.9. Pour des instances de petite et moyenne tailles, nous remarquons que "*MRSchedulingBasedOnCost*" fournit les meilleurs résultats dans la plupart des cas (du scénario 1 au scénario 16) avec des temps de calcul comparables à ceux fournis par les autres heuristiques.

FIGURE 5.8 – Évaluation de "MRSchedulingBasedOnCost" sur de petites instances dans un contexte hors ligne.

(a) Résultats de l'évaluation



(b) Moyennes des degrés de localité par scénario

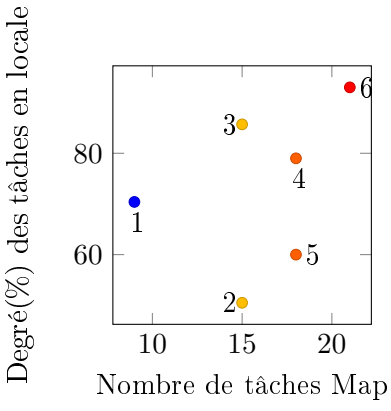


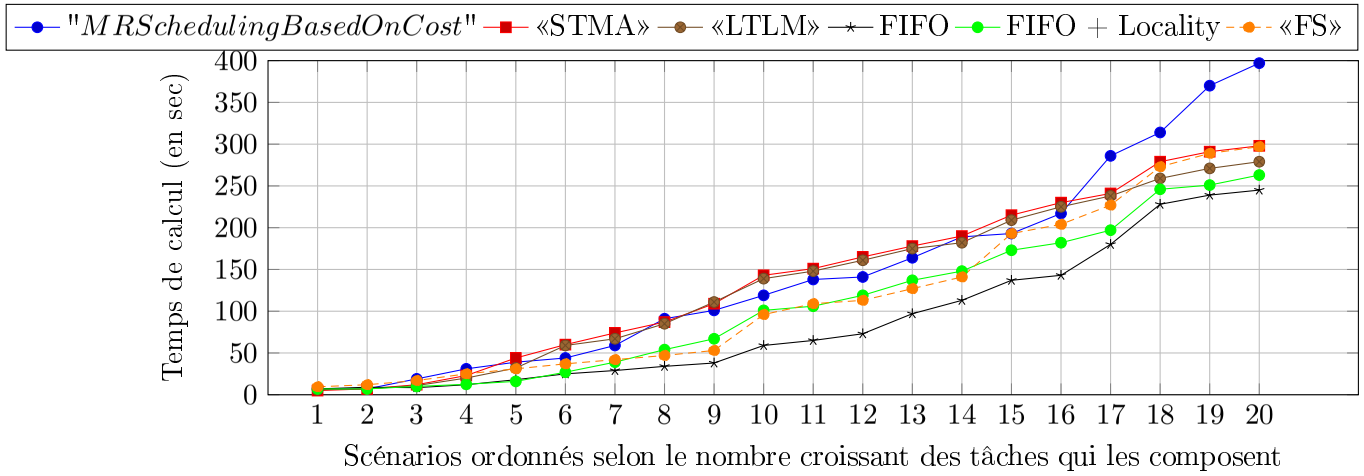
TABLE 5.12 – Moyenne des résultats des évaluations de "MRSchedulingBasedOnCost"

Scénarios	**Temps de calcul** (sec)	$\sum w_j C_j + C_{max}$
1	7.4	21542.3
2	7.2	17983.8
3	18.7	31391.8
4	31.2	35102.4
5	38.9	37019.1
6	44.4	43985.2
7	59.3	44957.5
8	91.1	69344.3
9	100.6	74551.1
10	119.1	90239.3
11	137.9	95942.9
12	141.3	100195.5
13	164.1	110739.2
14	189.4	157561.7
15	192.8	170723.9
16	216.9	187396.3
17	286.2	253185.7
18	314.4	303415.1
19	370.1	384583.6
20	397.2	403985.2

TABLE 5.13 – Écarts relatifs moyens (%) entre les solutions fournies par "MRSchedulingBasedOnCost" (Heur2) et par les heuristiques utilisées dans l'évaluation.

Sc.	«STMA»	Écart de "Heur2"	«LTLM»	Écart de "Heur2"	FIFO	Écart de "Heur2"	FIFO+ Locality	Écart de "Heur2"	«FS»	Écart de "Heur2"
1	25875	20.1	25356	17.7	24495	13.7	23945	11.2	23157	7.5
2	20397	13.4	19768	9.9	20190	12.3	19134	6.4	17934	-0.3
3	36478	16.2	34934	11.3	34892	11.2	34716	10.6	33870	7.9
4	38847	10,7	41298	17,6	36973	5,3	37389	6,5	39178	11,6
5	40374	9.1	42045	13.6	40947	10.6	40047	8.2	39389	6.4
6	48928	11.2	48394	10,0	46489	5.7	47389	7.7	46256	5.2
7	49274	9,6	48792	8,5	48824	8,6	48394	7,6	48294	7,4
8	77283	11,5	75033	8,2	75824	9,3	76823	10,8	74194	7.0
9	84239	13,0	83582	12,1	83153	11,5	82073	10,1	81714	9,6
10	97384	7,9	98072	8,7	96836	7,3	96295	6,7	95193	5,5
11	103845	8,2	102852	7,2	102485	6,8	103284	7,7	102382	6,7
12	105937	5,7	105838	5,6	105382	5,2	104748	4,5	104837	4,6
13	127384	15,0	125838	13,6	113938	2,9	125384	13,2	123284	11,3
14	170380	8,1	157958	0.3	168374	6,9	168834	7,2	168198	6,8
15	181293	6,2	180283	5,6	180937	6.0	179364	5,1	179173	4,6
16	216837	15,7	209837	12.0	214384	14,4	214253	14,3	212374	13,3
17	299387	18,2	298934	18,1	300034	18,5	293314	15,8	296902	17,3
18	304739	0,4	291838	-3,8	300203	-1,1	291339	-4.0	319833	5,4
19	463913	20,6	471094	22,5	452049	17,5	511945	33,1	461930	20,1
20	450274	11,5	451894	11,9	455288	12,7	536925	32,9	453428	12,2

FIGURE 5.9 – Moyennes des temps de calcul de chacune des heuristiques dans le cas de "MRSchedulingBasedOnCost"



5.5 Analyse des résultats

L'évaluation hors ligne du problème nous a permis d'illustrer le comportement des heuristiques. Les résultats obtenus suite à ces évaluations nous permettent de prouver les performances enregistrées sur de petites instances.

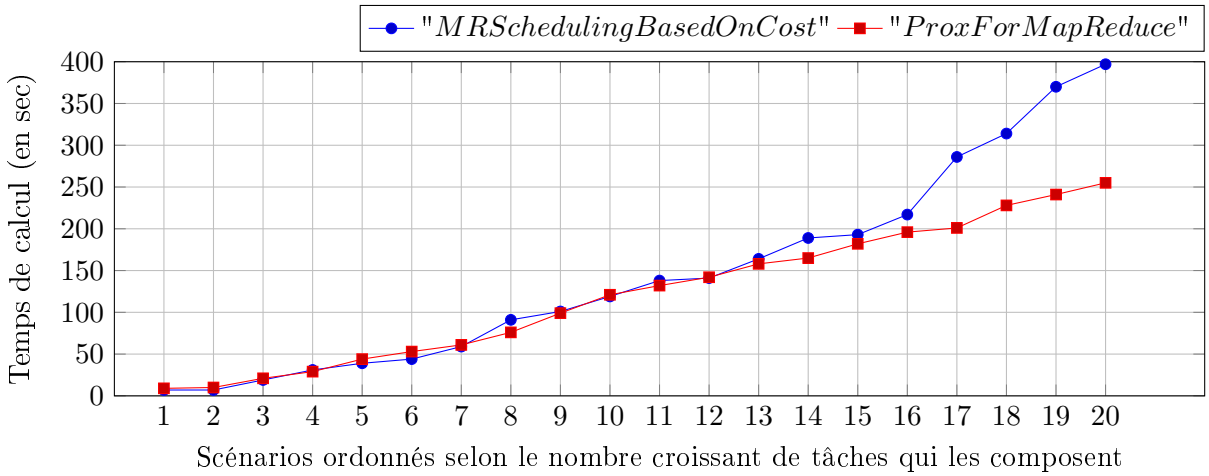
Les deux heuristiques que nous présentons dans la section 5.4 fournissent un écart moyen inférieur à 7% par rapport à la solution optimale calculée lors de résolution du modèle mathématique sur des petites instances. Nous remarquons qu'elles fournissent en moyenne des degrés de localité élevés (supérieur à 60%). En plus, le temps de calcul des solutions à l'aide des deux heuristiques est de l'ordre de la microseconde. Ces premiers résultats nous permettent de supposer que le comportement et les écarts calculés resteront en moyenne les mêmes pour des instances plus grandes. Une hypothèse que nous vérifions lors de l'évaluation des heuristiques en mode en ligne.

Les résultats de l'évaluation en ligne sont synthétisés dans les figures 5.10 et 5.11, nous remarquons que chaque heuristique possède ses propres avantages et inconvénients. Sur tous les scénarios à l'exception des scénarios 4 et 13 les résultats fournis par l'heuristique "MRSchedulingBasedOnCost" sont meilleurs que les résultats fournis par "ProxForMapReduce". Pour les instances de taille moyenne (les scénarios 1 à 15), les écarts entre les résultats fournis par les deux heuristiques sont en moyenne égales à 2%. En plus, les deux heuristiques consomment approximativement les mêmes temps de calcul. Les écarts moyens enregistrés entre les temps de calcul associés aux deux heuristiques sont inférieurs à 3%.

Pour des instances de taille plus grande (du scénarios 16 et plus), "MRSchedulingBasedOnCost" fournit les meilleures résultats. Elle atteint une amélioration d'environ 10% dans le cas du scénario 20. Cependant, l'heuristique "ProxForMapReduce" trouve des solutions moins bonne avec un gain entre 20 et 40% en temps de calcul. Pour les instances de taille moyenne, "MRSchedulingBasedOnCost" consomme plus de temps pour trouver les différentes solutions que l'heuristique "ProxForMapReduce". Par exemple, nous enregistrons un écart moyen de 25 % pour le scénario 17 et 30% pour le scénario 20 (figure 5.10). Malgré ces temps de calcul enregistrés, nous trouvons qu'elle présente entre 7 et 15% du temps entre la date de soumission du premier travail et la date de fin d'exécution du dernier. Ce qui offre aux résultats fournis par cette heuristique une efficacité respectable pour être utilisée en pratique. Dans certain cas, perdre 5% dans la qualité de la solution pour gagner 10% à 15% du temps de calcul est une bonne alternative. De ce fait, nous définissons une méthode qui permet d'identifier l'heuristique à utiliser en fonction de la taille des instances :

- Lorsqu'un seul utilisateur est connecté à la grappe, "LocFirst" s'adapte bien à ce contexte et son utilisation fournit de bonnes performances.
- Lorsque plusieurs utilisateurs sont connectés à la grappe, l'heuristique choisie doit dépendre de la taille des instances utilisées :
  - l'heuristique "*MRSchedulingBasedOnCost*" est utilisée pour les scénarios avec instances de taille petite et moyenne. Des instances qui ont une taille inférieure à 3150 tâches et 220 Vcores.

FIGURE 5.10 – Moyennes des temps de calcul des deux heuristiques "*ProxForMapReduce*" et "*MRSchedulingBasedOnCost*"

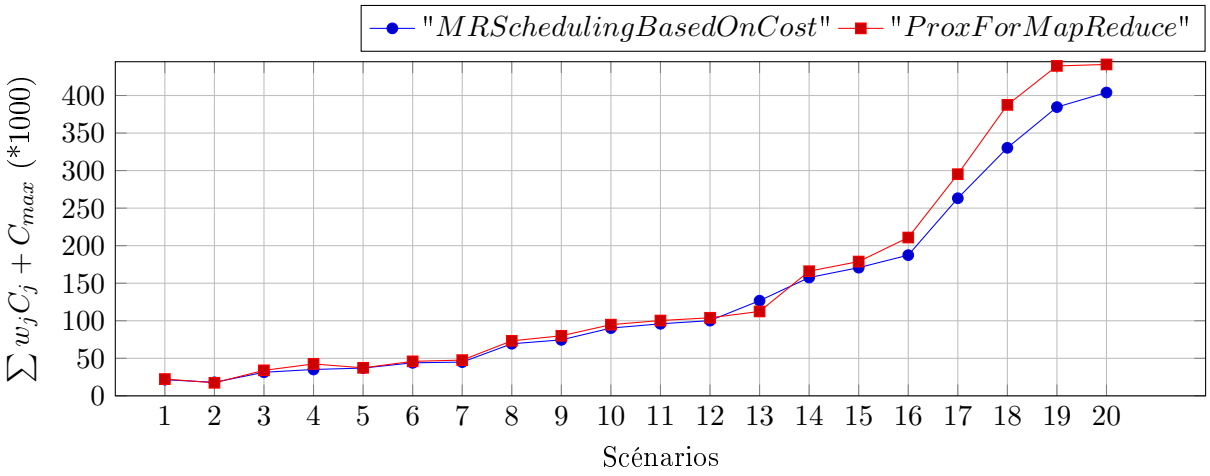


— l'heuristique "*ProxForMapReduce*" est utilisée pour les scénarios avec instances de grande taille.

5.5.1 Conclusion

Ce chapitre synthétise notre proposition pour résoudre le problème de l'optimisation des performances d'une plate forme informatique basée sur le logiciel Hadoop. Nous présentons trois heuristiques qui s'adaptent au problème identifié. La première est nommé "*LocFirst*", elle a pour objectif la minimisation du C_{max}. Elle peut être utilisée dans le cas ou un seul utilisateur soumet des travaux sur la grappe Hadoop. Les deux heuristiques restantes sont "*ProxForMapReduce*" et "*MRSchedulingBasedOnCost*". Elles ont pour objectifs la minimisation *(i)* de la somme pondérée des dates de fin des travaux et *(ii)* de la date de fin de tous les travaux à exécuter ($\sum w_j C_j + C_{max}$). Les résultats obtenus suite à l'évaluation hors ligne nous ont fourni une idée approximative sur les rendus de ces heuristiques. Chacune des trois heuristiques présentent respectivement un décalage constant par rapport à la solution optimale calculée lors de l'étude basée sur la modélisation mathématique du problème. Cependant, pour confirmer cette hypothèse, nous comparons nos solutions à des heuristiques de la littérature et sur des instances de tailles moyenne et grande. Nous utilisons la simulation lors l'évaluation des heuristiques. Nous trouvons que lorsque nous cherchons à minimiser le $Cmax$, l'heuristique "*LocFirst*" fournit les meilleures temps de calcul et à partir d'une certaine taille d'instances (1095 tâches, 88 Vcores), elle fournit les meilleures résultats. Chacune des heuristiques "*ProxForMapReduce*" et "*MRSchedulingBasedOnCost*" possède des caractéristiques spécifiques. Pour des petites et moyennes instances, la moyenne des écarts entre les résultats des deux heuristiques est négligeable (inférieur à 2%). L'heuristique "*MRSchedulingBasedOnCost*" fournit dans la majorité des cas les meilleures solutions, cependant, au détriment du temps de calcul. A partir d'une certaine taille

FIGURE 5.11 – Comparaison des moyennes des résultats des heuristiques "ProxForMapReduce" et "MRSchedulingBasedOnCost"



d'instances (2220 tâches, 187 Vcores), les écarts entre les temps de calcul des heuristiques s'agrandit, il atteint environ 30% avec des instances de grande taille (5130 tâches et 248 Vcores). Par rapport à la qualité des solutions obtenues, nous remarquons que "*MRSchedulingBasedOnCost*" permet d'améliorer la qualité des solutions d'en moyenne 4% par rapport aux solutions obtenues par "*ProxForMapReduce*". Le rapport entre la qualité d'une solution et le temps de calcul est un facteur important lors du choix de l'heuristique à utiliser. Suite à l'analyse des différents résultats, il nous est apparu nécessaire de définir à la fin de ce chapitre les circonstances de l'utilisation de chaque heuristique.

Conclusions et perspectives

Nous pensons, à travers cette thèses, apporter des solutions aux problèmes auxquels l'entreprise Group Cyrès est confrontée lors de l'exploitation de ses grappes "Big data".
Nous avons étudié le problème d'optimisation des performances d'une grappe basée sur le logiciel Hadoop. Nos travaux focalisent sur deux axes. Le premier est caractérisé par son aspect expérimental et il consiste à fournir une réponse à court et à moyen terme pour *(i)* améliorer la manière dont les ressources sont exploitées, *(ii)* réduire le coût d'exploitation des grappes et *(iii)* donner plus de flexibilité pour faciliter le déploiement des infrastructures Hadoop. Le deuxième axe focalise sur l'optimisation de l'ordonnancement des travaux MapReduce, ce problème de la gestion dynamique des travaux permet à des politiques d'ordonnancement de gérer à la volée la répartition des tâches dans la grappe en respectant certaines contraintes.
La première étude (axe) nous a permis d'identifier les points à améliorer et de définir les étapes que nous avons suivi pour réaliser les travaux liés au second axe. Dans les conditions réelles d'utilisation des grappes Hadoop, les utilisateurs veulent que leurs travaux se terminent dans les plus brefs délais. S'il existe plusieurs personnes qui utilisent la grappe en même temps, chacun d'entre eux souhaite que les travaux qu'il soumet se terminent les premiers. De ce fait, il faut que la politique d'ordonnancement utilisée fournisse une solution qui résout le compromis entre le souhait de chaque utilisateur et la réduction de la date de fin des travaux. Nous avons décidé, en conséquence, de minimiser le temps d'attente moyen pondéré pour qu'un travail ne reste pas trop longtemps dans la plateforme Hadoop tout en veillant à minimiser la date de fin du dernier travail. Nous avons décidé d'étudier, dans le deuxième axe, le problème de la gestion dynamique des travaux MapReduce. Cette approche dynamique permet à des politiques d'ordonnancement de gérer à la volée la répartition des tâches dans la grappe en respectant certaines contraintes.
Ce document présente mes activités de recherche pour l'optimisation de l'exécution des travaux MapReduce. Ces travaux engendrent, lorsqu'ils sont soumis, des tâches du type "Map" et "Reduce", non préemptibles, qui sont caractérisées par des relations de précédence et des durées d'exécutions. Elles consomment les ressources disponibles sur les machines. Elles traitent des blocs de données en entrée et fournissent des données en sortie. Les blocs de données d'entrée peuvent être migrés entre les machines de la grappe et les données de sortie doivent être transférées vers les tâches Reduce. La grappe est constituée de M machines identiques inter-connectées par des liens de communication. Chacune de ces machines possède un ensemble de ressources que les tâches exploitent. Cet ensemble est composé de la mémoire, disque dur, CPU et la bande passante du réseau.

L'ordonnancement des travaux sur la plateforme Hadoop est caractérisé par son aspect en ligne. De ce fait, la politique d'ordonnancement adoptée doit fournir une solution de bonne qualité (qui répond aux objectifs fixés) en un temps de calcul réduit. Nous avons choisi de proposer des heuristiques gloutonnes pour le calcul de solutions apportées.

Nous avons traité d'abord le problème hors-ligne par lequel nous avons utilisé une approche basée sur la programmation linéaire en nombre entier. Cette approche consiste à modéliser le problème avec la modélisation mathématique et à calculer une borne inférieure. Ensuite, nous focalisons sur la version en ligne du problème et nous proposons des heuristiques que nous évaluons.
Lorsque nous avons étudié le problème hors ligne, nous avons utilisé le solveur CPLEX pour le calcul de la borne inférieure et résoudre le modèle mathématique en essayant de respecter les différentes contraintes du problème. Nous avons commencé avec un premier modèle simpliste, nous avons considéré des tâches de durée unitaire et la minimisation du critère C_{max}. Nous l'avons résolu pour calculer une borne inférieure qui est utilisée pour l'évaluation de la première heuristique proposée, nommée "LocFirst". Ensuite, nous avons fait évoluer le modèle pour considérer un modèle plus réaliste avec des tâches de durée quelconque et avec la minimisation de $\sum w_j C_j + C_{max}$ comme fonction objectif. Nous avons calculé la borne inférieure pour ce modèle. Vue la complexité de ce dernier, CPLEX n'a pu le résoudre qu'avec des instances de petite taille. De ce fait, nous avons utilisé des algorithmes de la littérature pour évaluer les heuristiques que nous proposons avec des instances moyennes et grandes. Nous avons proposé deux heuristiques "$ProxForMapReduce$" et "$MRSchedulingBasedOnCost$". Nous les avons évaluées sur des instances allant de quelques tâches à environ 5000 tâches. Les simulations effectuées ont montré qu'avec la plupart des instances, les heuristiques fournissent des résultats soit à écart faible par rapport à la borne inférieure (avec les petites instances) soit meilleur que les solutions fournies par, les heuristiques de la littérature, utilisées dans l'évaluation. Pour la plupart des scénarios, l'heuristique "$MRSchedulingBasedOnCost$" fournit des résultats meilleurs que ceux fournis par "$ProxForMapReduce$". Cependant, lorsque les instances sont de grande taille, "$MRSchedulingBasedOnCost$" consomme plus de temps pour le calcul de la solution. Lorsque les instances sont de petite et moyenne tailles, les temps de calcul enregistré pour les deux heuristiques sont très proches. Nous en concluons que "$MRScheduling\text{-}BasedOnCost$" est adaptée pour les instances de petite et moyenne taille puisqu'elle fournit les meilleurs résultats dans un temps réduit.
Durant les différentes études effectuées tout au long de cette thèse, nous étions confronté à certaines limites qui permettent d'envisager des perspectives de recherche intéressante suite à cette thèse. L'influence de la monté des charges de travail et l'influence de l'emplacement des blocs de données sont des axes de travail que nous souhaitons étudier. Nous avons veillé, durant cette thèse, à respecter certaines contraintes du monde réel. Nous pensons que faire intervenir l'administrateur pour une meilleure prise de décision de l'ordonnancement nous permettra de mieux paramétrer de la grappe en amont et en conséquence améliorer les résultats obtenus.
Dans les sections suivantes, nous présentons les perspectives que nous jugeons les plus pertinentes en les organisant de manière plus globale du court au plus long terme.

Étude de l'élasticité d'une grappe Hadoop

Les nouvelles installations des grappes Hadoop se basent sur des plateformes virtuelles. Ce type de plateforme offre plus de flexibilité et d'adaptabilité des grappes aux besoins des entreprises. Cependant, dans certaines conditions, des dégradations importantes des performances d'une grappe peuvent être remarquées. Par exemple, on trouve cette dégradation lorsqu'un nombre important de grands travaux sont soumis sur la grappe ou lors des pertes de machines suite à des pannes. En général, la façon la plus simple et rapide pour remédier à ce problème est d'ajouter soit de la mémoire à chaque machine (extension verticale) soit des machines à grappe (extension horizontale).
Nous pensons que l'étape suivante consiste à étudier l'élasticité du système Hadoop et à définir la méthodologie nécessaire pour l'assurer. L'élasticité d'un système informatique est sa capacité à s'adapter aux changements des quantités des ressources (en approvisionnant et désapprovisionnant) de la plateforme informatique. Nous pensons que la capacité actuelle du système Hadoop à s'adapter aux demandes en approvisionnant et désapprovisionnant des ressources de manière automatique est très limitée. Plusieurs évolutions techniques integrées à Hadoop justifient de cette piste. Par exemple, nous pouvons citer *(i)* la séparation entre les entités responsables de la gestion du système de fichiers et les entités responsables de l'exécution des tâches et *(ii)* l'intégration de la virtualisation par conteneur (Docker) [145] au sein du système système Hadoop. La séparation entre les entités permet de faciliter la gestion et la migration des tâches et des données entre les machines. L'intégration de l'outil de virtualisation Docker réduit le temps de mise en place et d'extinction des nouvelles machines. Ce qui limite le temps de latence entre l'instant d'approvisionnement ou le désapprovisionnement du système.

Amélioration de la méthode de placement des blocs de données

Notre approche actuelle focalise sur la stratégie d'ordonnancement des tâches sur la grappe. La répartition des blocs de données sur la grappe influence clairement les performances d'Hadoop et a un impact sur l'ordonnancement. Elle réduit les échanges réseaux et peut être exploitée pour réduire la consommation énergétique. Nous pensons travailler sur le problème de la consommation énergétique des grappes Hadoop à travers l'optimisation des politiques d'emplacement des blocs de données. Nous pensons que le fait d'organiser les machines en classes en fonction de la répartition des blocs de données nous permettrait de réduire la consommation énergétique. Ensuite, nous éteignons ou activons l'ensemble des machines appartenant à une classe particulière en fonction du taux d'exploitation des ressources et des utilisateurs connectés au système. Nous pensons que cette piste est adaptée aux perspectives d'amélioration de l'élasticité dans Hadoop puisqu'elle profitera de la flexibilité offerte par la gestion élastique de la grappe.

Donner plus de valeur à l'administrateur pour l'amélioration des performances des heuristiques

Hadoop dépend d'un nombre important de paramètres que l'administrateur doit gérer pour assurer son bon fonctionnement. En conséquence, nous avons veillé depuis le début de cette thèse

à ce que nos heuristiques soient autonomes. Elles doivent fournir les meilleurs résultats sans faire intervenir l'administrateur du système. A ce niveau, nous pensons qu'une étude multi-critère plus pointue peut apporter de la valeur ajoutée. Nous pensons que faire intervenir l'administrateur dans la prise de décision de l'ordonnancement nous permettra d'améliorer les résultats obtenus. De ce fait, dans le long terme, aborder le problème d'ordonnancement comme un problème multi-critère. Nous permettra de mieux répondre aux exigences des utilisateurs et assurer le meilleur service. Cependant et afin de palier au problème de paramétrage de Hadoop, nous pensons automatiser la gestion des paramètres, c'est-à-dire, nous pensons mettre en place un module pour leur configuration. Ce dernier dépendra des métriques remontées par le système et d'un modèle de définition des valeurs des paramètres les plus influents. La définition d'un module de définition automatique des valeurs des paramètres est un besoin communautaire. Il n'existe pas d'outils libres permettant d'assurer le paramétrage de la grappe de façon automatique. Nous concluons que l'intégration de cet outil est un besoin que nous devons satisfaire pour faciliter l'utilisation de Hadoop.

Annexe A

Étude du déploiement de Hadoop dans les nuages informatiques

A.1 Introduction

Hadoop est une application de gestion de gros volumes de données, elle est connue pour son besoin en ressources informatiques et sa capacité à exécuter un grand nombre de tâches de manière distribuée. L'entreprise Cyrès a pour objectif de fournir des services basés sur Hadoop à ses clients à travers les nuages informatiques (Cloud).
Nous présentons dans ce chapitre une étude expérimentale réalisée pour résoudre et répondre à un besoin identifié chez le pôle Ingensi de l'entreprise Group Cyrès. L'objectif est d'identifier les outils de virtualisation qui permettent de faciliter le déploiement des grappes Hadoop sur le Cloud. Lorsque j'ai commencé cette étude, les membres d'Ingensi installaient directement Hadoop sur des plateformes physiques. En même temps, l'entreprise Group Cyrès est un client de l'entreprise VMware et possède déjà des plateformes VMware installées dans son centre de données. La question à laquelle nous devons apporter une réponse à travers cette étude est la suivante : est ce qu'il existe un outil, autre que VMware, moins cher qui fournit des performances meilleures tout en restant facile à utiliser ? La première intuition, que nous avons eue, est de chercher parmi les outils libre un outil qui répond aux besoins identifiés.
Les technologies de virtualisation sont les technologies les plus utilisées pour camoufler la complexité de la gestion des plateformes physiques, pour optimiser la gestion de leurs ressources et pour diminuer le coût énergétique des grappes informatiques. Après l'étude de l'état de l'art et les discussions en équipe. Nous décidons de mettre à l'épreuve la technologie basée sur les conteneurs et nous avons choisie d'évaluer l'outil Docker.
Dans notre étude, nous divisons les technologies de virtualisation en deux classes : la virtualisation lourde (ou complète) qui est basée sur la gestion de machines virtuelles (VM) et la virtualisation légère (ou partielle) qui est basée sur la gestion de conteneurs. Vous trouverez la définition de ces classes dans la section A.2.

Traditionnellement, les plateformes déployant des logiciels de gestion de gros volumes de

données utilisent la virtualisation lourde. Les raisons principales sont : la capacité de ces environnements à assurer la séparation complète entre les systèmes d'exploitation client et hôte, et la non-maturité des outils de virtualisation légère pour la production. Cependant, durant les dernières années, la monté des performances des outils tels que Docker a encouragé certaines entreprises à l'adopter pour le déploiement de leurs infrastructures. De ce fait, plusieurs travaux de recherche ont vu le jour. Ces travaux abordent les performances des grappes Hadoop dans le Cloud et l'amélioration de leur consommation énergétique. Les travaux de recherche, qui étudient les performances de Hadoop sur des plateformes basées sur la virtualisation légère, sont restreints aux travaux de Jiang et al. [84] et Verma et al. [146]. De ce fait, nous commençons par présenter les travaux réalisés pour évaluer Hadoop sur des plateformes virtuelles lourdes et légères. Ensuite, vu les travaux limités sur Docker, nous présentons les travaux réalisés pour évaluer Docker dans d'autres domaines :

Performance. Feller et al [54] évaluent les performances de Hadoop en le déployant sur *(i)* des plateformes basées sur la virtualisation lourde (traditionnelle) et *(ii)* en déploiement directement sur des plateformes physiques. Ils s'intéressent particulièrement aux deux modèles suivants, *(1)* le premier consiste à faire cohabiter les services de calcul et de sauvegarde de données sur les mêmes machines.*(2)* le deuxième consiste à les séparer sur des machines différentes. Ils focalisent sur l'évaluation de la vitesse d'accès aux disques. Ils concluent que la séparation des services de calcul et des données augmente la consommation énergétique et ne fournit pas des garanties de performances puisque les performances dépendent fortement de la quantité des données à traiter. Jiang et al. [84] évaluent Hadoop en utilisant plusieurs outils de virtualisation légère, autre que la technologie Docker, tels que VServer, OpenVZ et "Linux Containers". Ils focalisent principalement sur l'étude de la dégradation de performances lors de la séparation des services de calcul et sauvegarde de données. Ce chapitre peut être vu comme la continuité de ces deux travaux puisque, d'une part, il prend en considération la technologie Docker et la compare à un outil de virtualisation lourde (traditionnelle). Il étudie la consommation énergétique des grappes basées sur les deux technologies de virtualisation (lourde et légère). D'autre part, il fournit une étude détaillée de la consommation des ressources (CPU, mémoire, accès au support de sauvegarde et la consommation de la bande passante réseau) des grappes Hadoop basées sur les deux types de déploiement. En plus, les expériences effectuées sont présentées comme des perspectives définies par Jiang et al. [84] telle que l'évaluation du compromis entre la performance et la consommation énergétique d'une grappe Hadoop en utilisant les deux technologies de virtualisation (lourde et légère).

D'autres travaux tels que ceux de Shadi et al. [1], Xu et al. [160] et Kontagora et al. [95] étudient et évaluent la variation des performances de Hadoop déployé en utilisant des outils de virtualisation lourde tel que OpenStack. Shafer et al. [137] focalisent sur l'étude du système de fichiers HDFS de Hadoop, ils identifient des points à améliorer et définissent des bonnes pratiques à suivre pour remédier à certaines des faiblesses identifiées. Moise et al. [2] étudient la possibilité d'utiliser les nuages informatiques pour réduire le coût d'exploitation des grappes Hadoop, ils étudient le compromis entre le coût de location des ressources informatiques et les performances obtenues. Les expériences effectuées par Iordache et al. [78] étudient la possibilité d'exécuter les applications MapReduce sur des Clouds différents, ils étudient le compromis entre performances/

prix de location des ressources.
La consommation énergétique. Il existe, dans la littérature, deux techniques pour améliorer la consommation énergétique des infrastructures informatiques, la première consiste à régler dynamiquement la fréquence et la tension des cœurs CPU (Dynamic Voltage and Frequency Scaling, DVFS) [121], la deuxième consiste à mettre en veille les machines faiblement exploitées. Lang et al. [97] s'intéressent à la deuxième technique et étudient deux politiques nommées *(i)* ensembles couvrants (Covering Subset, CS) et *(ii)* "All-In Strategy" (AIS). L'objectif de la politique CS, pendant les périodes de faible exploitation, est de ne laisser activé qu'un sous-ensemble de machines. L'objectif d'AIS est d'utiliser toute la grappe Hadoop pour exécuter les différents travaux ensuite éteindre toutes les machines de la grappe. Lang et al. [97] montrent que l'utilisation de la politique AIS fournit des meilleures consommations énergétiques que la politique (CS). Kaushik et al. [87] fournissent une politique basée sur CS pour améliorer l'emplacement des blocs et proposent une politique d'ordonnancement pour l'optimisation de la consommation énergétique et des performances. Leverich et al. [101] améliorent la politique d'équilibrage de charge existante dans Hadoop. Cette amélioration consiste à assurer la meilleure répartition des blocs sur la grappe. Ils implémentent et évaluent une politique basée sur (CS) pour améliorer la consommation énergétique des grappes virtuelles Hadoop basées sur la virtualisation lourde. Wirtz et al. [154] définissent une politique d'allocation de ressources et d'ordonnancement basée sur la politique DVFS de réduction énergétique. Polato et al. [123] présente $HDFS_H$, un mécanisme hybride de stockage entre des disques de type HDD (Hard Disk Drive) et des disques de types SSD (Solid State Drive). $HDFS_H$ permet à Hadoop de profiter des avantages de l'utilisation des disques durs HDD (coût réduit par GB et une grande capacité de sauvegarde) et des avantages de l'utilisation des SSDs (vitesse rapide d'accès en lecture-écriture et faible consommation énergétique). Les expériences effectuées montrent une amélioration des performances, en durée d'exécution des travaux, et une réduction de la consommation énergétique globale de la grappe.
La technologie de virtualisation Docker [157] est réputée comme un standard pour la gestion des conteneurs Linux, sa première version est apparue en 2013. Verma et al. [146] se basent sur une architecture complexe pour déployer Hadoop dans des conteneurs qui s'exécutent sur des machines virtuelles. Verma et al. se trouvent face à des dégradations importantes de performances et des problèmes de partage des ressources (bande passante réseau et puissance CPU). Leur travail se base sur LXC pour la gestion des conteneurs.
Docker est évalué dans le contexte des outils de calcul à haute performance HPC. Xavier et al. [158] évaluent les performances de la virtualisation avec Docker avec des outils HPC, ils étudient le compromis entre les performances et la capacité de Docker à assurer la bonne isolation des ressources. Ils comparent les performances (en matière de durée des travaux) d'une plateforme basée sur Docker aux performances fournies par un déploiement sur une infrastructure physique. Xavier et al. confirment que la surcharge due à l'utilisation de Docker engendre une dégradation d'en moyenne 5% des performances. Reshetova et al. [128] étudient la sécurité offerte par les outils de virtualisation légère. Ils étudient plusieurs types d'isolation (isolation de processus, de système de fichiers, des réseaux, des ressources, des fichiers de configuration). Ces différentes formes d'isolation sont basées sur des bibliothèques Unix pour la gestion des noms de domaines (namespace) et les bibliothèques Cgroups et rlimits pour la gestion des droits d'accès.

Reshetova et al. confirment que les systèmes basés sur les outils de virtualisation légère sont fiables et sécurisés. Cependant, ils rappellent quelques bonnes pratiques et des consignes de sécurité. La consigne principale est qu'il faut bien vérifier les bonnes installations des bibliothèques citées. Pour l'instant, un énorme effort d'intégration est en cours pour faciliter l'installation et la gestion des différents éléments du système de sécurité dans Unix. Par exemple, l'intégration de la bibliothèque rlimits dans Cgroups est une tâche en cours de développement.
Felter et al. [54] montrent que le niveau de surcharge faible engendré suite à l'utilisation des outils de virtualisations légères améliore les performances des applications. Les résultats des expériences confirment que Docker fournit des performances meilleures (ou au pire des cas égales) que les performances fournies par KVM . Docker est un nouvel outil de virtualisation qui est adopté par plusieurs organismes, plusieurs projets sont créés pour l'adapter aux contextes de son utilisation, Peinl et al. [120] synthétisent les différents projets libres qui cherchent à développer et à adapter Docker au contexte du Cloud informatique.
Le domaine de la gestion des gros volumes de données et le domaine de calcul à hautes performances sont deux domaines divergents. Chacun possède ses propres besoins, caractéristiques et affinités. Ce travail focalise d'une part sur l'évaluation des performances d'une grappe Hadoop basée sur Docker, d'autre part, à comparer cette technologie émergente aux technologies lourdes de virtualisation.

Dans ce travail, l'outil de virtualisation que nous choisissons doit améliorer les critères suivants : *(1)* le niveau d'augmentation de productivité (faciliter et rapidité du déploiement d'une grappe Hadoop), *(2)* les performances obtenues après le déploiement et *(3)* le gain en coût d'exploitation des grappes. De ce fait,

— Nous étudions les hypothèses concernant la configuration des plateformes de gestion de gros volumes de données. Notre objectif est d'identifier des écarts de performances entre les outils de virtualisation que nous évaluons, de fournir la motivation pour des nouvelles recherches concernant le déploiement d'Hadoop sur le nuage. Nous voulons attirer l'attention sur le fait que la virtualization légère est maintenant prête pour le déploiement des infrastructures de gestion de gros volumes de données, elle génère un niveau de surcharge très bas.

— Nous nous restreignons à l'utilisation du logiciel Hadoop et nous abordons l'impact de l'utilisation de la virtualisation légère sur la performance et la consommation énergétique des grappes Hadoop.

— Nous évaluons Hadoop avec deux technologies de virtualisation, la virtualisation lourde et légère. Les outils utilisés pour effectuer la comparaison sont VMware vSphere et Docker. D'après plusieurs travaux de comparaison sur des outils HPC, Docker est l'outil le plus avancé par rapport aux outils de virtualisation légère existants.
L'évaluation d'Hadoop avec les deux outils nous permet d'une part de les comparer, d'autre part, d'analyser la capacité de chacun à répondre aux besoins du contexte. Durant les expérimentations, nous considérons les critères suivants : le temps pour terminer les travaux soumis, la quantité des ressources physiques utilisée et la quantité d'énergie consommée.

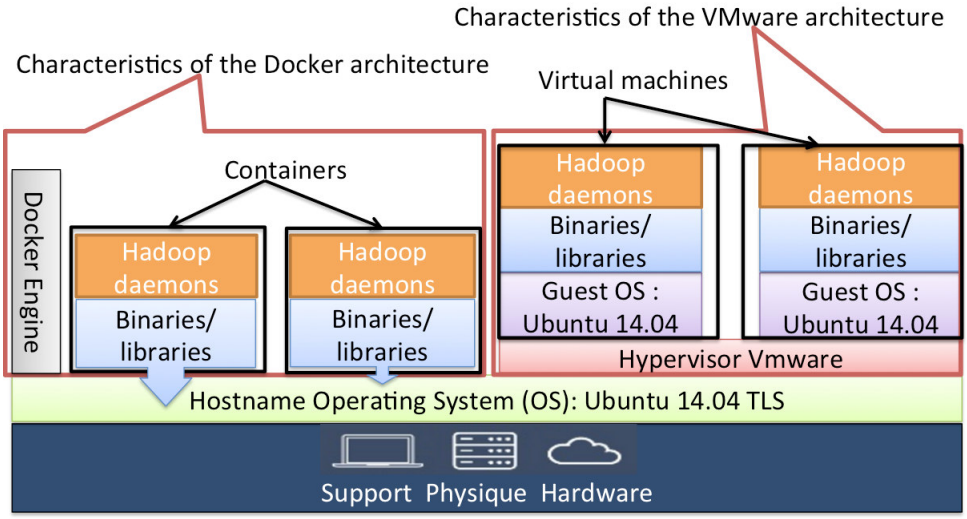


FIGURE A.1 – Comparaison entre l'architecture des outils Docker et VMware

— Nous pensons fournir à l'entreprise Group Cyrès une étude approfondie sur les avantages et inconvénients de l'utilisation de chaque technologie de virtualisation et de l'influence, de ces derniers, sur la performance et la consommation énergétique des grappes Hadoop.

Ce chapitre est organisé comme suit : nous commençons par présenter les deux types de virtualisation et les travaux effectués pour l'évaluation d'Hadoop et Docker. Ensuite, nous présentons la méthodologie que nous utilisons durant nos expérimentations. Enfin, nous présentons les résultats des expérimentations et nous les analysons.

A.2 Présentation des technologies de virtualisation

Les technologies de virtualisation sont présentées, pour la première fois, en 1960 par IBM [151]. De façon transparente, elles permettent le partage de temps et de ressources sur des serveurs informatiques. Depuis leur introduction, elles avaient comme objectif l'amélioration de la productivité en donnant la possibilité à plusieurs machines clientes de tourner sur le même serveur physique. Plusieurs types d'outils de virtualisation sont utilisés dans les centres de données [151]. Dans notre travail, nous les classons en deux types qui sont présentés dans les sous-sections suivantes.

A.2.1 La virtualisation lourde, isolation complète :

La virtualisation lourde consiste en un moniteur de machines virtuelles (hyperviseur) et un ensemble de machines virtuelles (VM). Chaque (VM) possède son propre système d'exploitation. Le concept de "machine virtuelle" est basé sur la réservation des ressources. La quantité de mémoire, la quantité de disque, le nombre de processeurs et la bande passante réseau sont réservés depuis le démarrage de la machine. Ces ressources sont réservées pour exécuter les instructions

de la machine cliente. Au cas où cette dernière est au repos, c'est-à-dire ne contient pas des instructions à exécuter, les ressources restent réserver et ne peuvent être réutilisées.
L'utilisation de ce type de virtualisation engendre la surcharge des machines physiques. Cette surcharge est due aux critères suivants :

— utilisation des pilotes de gestion des services de virtualisation.

— utilisation des codes intermédiaires responsables de l'interprétation des instructions du système client.

— aux instructions de gestion associées à l'hyperviseur.

En utilisant ce type de virtualisation, l'isolation est assurée au niveau logiciel et au niveau matériel [77]. Dans nos expérimentations, nous utilisons l'outil VMware vSphere, la figure A.1 présente les différentes couches qui composent une infrastructure virtuelle basée sur la virtualisation complète, particulièrement sur VMware.

A.2.2 La virtualisation légère, isolation partielle :

La virtualisation légère est basée sur les conteneurs, les instructions d'un conteneur s'exécutent comme une application ordinaire tournant sur le système d'exploitation de la machine hôte. Les conteneurs peuvent être installés sur des machines virtuelles, appartenant à la virtualisation lourde. Les politiques de gestion des ressources sont différentes des méthodes de gestion traditionnelles, elles consistent à borner la quantité maximale de chaque ressource qu'un conteneur peut utiliser. Un conteneur est l'équivalent d'une VM dans le contexte de la virtualisation lourde. Il possède d'autres politiques de gestion de ressources et possède un système d'exploitation dépendant du système installé sur la machine physique. Lorsqu'un conteneur exécute des instructions, il récupère la totalité des ressources qui lui sont attribuées, cependant, lorsqu'il n'y a pas d'instructions à exécuter (situation de repos), ses ressources sont libérées.
Dans nos expérimentations, nous utilisons les conteneurs Docker [40]. Ce dernier est présenté pour la première fois en 2013. Il se base sur l'outil LXC [131] et l'enrichit avec de nouvelles fonctionalités. Aujourd'hui, Docker utilise une nouvelle bibliothèque dédiée, connue sous le nom de "libcontainer" pour accéder au kernel du système hôte. Les conteneurs Docker utilisent les "kernel control groups" (Cgroups) , "systemd" [37] et le "kernel namespaces libraries" pour :

— limiter la consommation des ressources,

— consolider l'isolation des ressources,

— la gestion des droits et des processus.

L'idée principale est de créer des conteneurs bien isolés qui ne peuvent pas être visible de l'extérieur.

A.2.3 Différence entre les deux types de virtualisation

Lorsqu'ils s'exécutent, les conteneurs utilisent, dans une grande partie, des librairies du système hôte. De ce fait, la quantité des instructions système interprétées est réduite d'environ la moitié par rapport aux instructions système associées aux machines virtuelles.

Un autre point déterminant est le temps nécessaire pour la mise en place d'une plateforme virtuelle. l'initiation d'un conteneur est de l'ordre de la seconde, alors que démarrer une machine virtuelle est de l'ordre de la minute. Lorsque vous voulez réduire la taille de votre grappe Docker, il suffit de supprimer les conteneurs tandis que les machines virtuelles doivent être éteintes avant de s'en débarrasser. Avec les conteneurs, on réduit la taille d'une grappe en quelques secondes, cependant, avec une grappe de VMs, cette opération est de l'ordre de la minute. Ce paramètre traduit la capacité de chaque technologie à assurer l'adaptation instantanée aux besoins en puissance. La différence entre les deux technologies, à ce niveau, est leur capacité à remédier en un temps limité aux besoins en puissance à un instant donné. Par exemple, lors de la montée en charge d'une infrastructure et la dégradation de ces performances, la virtualisation légère est capable d'initier de nouveaux conteneurs en quelques secondes et de faire face à cette montée en charge pour conserver la qualité de service fournie aux clients. Les politiques de gestion des ressources utilisées par virtualisation légère sont plus flexibles que les politiques utilisées par les technologies de virtualisation lourde. Par exemple, pour la gestion des cœurs CPU, Docker permet de *(i)* fixer le nombre de cœur CPU par conteneur, ou, *(ii)* il permet d'effectuer un partage relatif de la puissance de calcul de la machine physique entre les conteneurs. Pour la gestion de la mémoire, les conteneurs utilisent la mémoire seulement quand ils en ont besoin et en respectant la limite maximale autorisée lors de leurs initialisations. Si un conteneur n'a pas besoin de la mémoire que l'administrateur du système lui a attribuée, il la libère. De cette façon, le système de l'hôte aura à sa disposition plus de ressources mémoire pour assurer la meilleure gestion de l'ensemble des conteneurs qu'il gère. Les outils de la virtualisation lourde gèrent autrement les ressources. Dès le démarrage d'une VM, toutes les ressources lui sont réservées et ne sont libérées que lorsque la VM est éteinte.

A.3 Méthodologie des expérimentations

Pendant les différents travaux, deux séries de tests sont effectuées. Dans la première, nous étudions les performances d'une grappe à machines homogènes. Dans la deuxième, nous étudions l'influence de l'utilisation des grappes à machines hétérogènes. Chaque expérience est répétée en utilisant les deux outils de virtualisation VMware VSphere et Docker. Chaque expérience est répétée cinq fois. Nous commençons dans la première série par la comparaison des deux outils. Nous nous intéressons à étudier la variation des performances (la date de fin des travaux). Nous focalisons sur l'utilisation des ressources et nous récupérons les métriques associées suite aux différentes exécutions. Ensuite, nous focalisons sur les grappes hétérogènes. Nous étudions les politiques de gestion des ressources dans Docker et nous les évaluons afin d'analyser leurs influences sur Hadoop. Dans la deuxième série d'expériences, nous étudions sur la consommation énergétique des grappes Hadoop.

Nous utilisons une machine physique de capacité 12 cœurs CPU, 32 GB de mémoire et 500 GB de disque dur. La configuration des machines virtuelles varie en fonction des expériences et du type de la grappe :

Grappe à machines homogènes : les machines clientes (machines virtuelles ou conteneurs)

TABLE A.1 – Configuration des machines (physiques ou virtuelles) utilisées dans les expériences sur une grappe homogène

	Machine hôte (physique)	Machine cliente (virtuelle)	Conf. des éléments Hadoop (Ressource Manager et Node manager)
Processeur	Intel ®Xeon(R) CPU E5-26200 @ 2.00GHz		
CPU cores	12	2 cœurs (4 threads)	1 cœurs (2 threads)
RAM (GB)	32	5	3
HDD (GB)	500	80	65
OS	Ubuntu 14.10		

possèdent la même configuration suivante, 2 cœurs CPU virtuelles, 5 GB de mémoire et 80 GB d'espace de disque dur. Le tableau A.1 synthétise les différentes configurations associées à cette catégorie de grappe.

Grappe à machines hétérogènes : les machines clientes possèdent la configuration affichée dans le tableau A.2.

Lorsque nous évaluons des grappes homogènes, tous les éléments de Hadoop ont la même configuration affichée dans le tableau A.1. Lorsque nous évaluons une grappe à machines hétérogènes, les éléments Hadoop ont la configuration affichée dans le tableau A.2. Le nombre de réplication des blocs de données dans Hadoop est 2. Sur chaque machine cliente, nous utilisons les logiciels Ganglia et hsflow [106] pour récupérer les métriques reliées aux :

— niveaux de la charge sur les machines : il représente le nombre moyen de processus dans les files d'attentes (en attente de ressources) sur la grappe (par rapport au nombre de CPU).

— taux de consommation des ressources CPU.

— taux de consommation des ressources mémoire.

— taux de consommation de la bande passante réseau.

— taux de consommation de la bande I/O des lectures disques.

Ces outils de mesures engendrent un surcoût d'en moyenne 2% sur la grappe [77]. Nous utilisons un outil de mesure de la consommation électrique de la grappe, il est monté entre la prise électrique et les machines de la grappe et enregistre la consommation toutes les deux secondes. Les résultats de la consommation sont enregistrés sur une carte mémoire que nous récupérons et analysons en fin d'expérience.

Durant les tests, nous utilisons les travaux "TeraGen", "TeraSort", "TestDFSIO-write" et "TestDFSIO-read". Ils sont référencées et utilisées dans plusieurs travaux d'évaluation de performance [51] tels les travaux des entreprises Intel [74] et AMD [45]. Les quatre travaux utilisés dans nos expérimentations sont du type MapReduce.

Les travaux "TeraGen" et "TestDFSIO" sont utilisés pour évaluer l'utilisation du disque dur et de la bande passante. Ils sont constitués d'un ensemble de tâches "Map" qui écrivent de façon

TABLE A.2 – Configuration des machines esclaves et de l'élément "NameNode" utilisé dans les expériences sur une grappe hétérogène

Ressources	Machine esclave 1	Machines slave 2 et 3	Conf. du "NameNode"	Conf. du "DataNode"
CPU (ou cœurs virtuelles)	4 cœurs (8 threads)	2 cœurs (4 threads)	6 cœurs virtuelles	1 cœurs (2 threads)
Mémoire (GB)	10	5	6	4
Espace HDD (GB)	80	80	-	65

séquentielle des données sur HDFS. Durant ces expériences, je génère trois tailles de données, 10 Go, 15 Go et 20 Go en utilisant 2 puis 4 machines esclaves. Le travail "TeraSort" est utilisé pour évaluer les ressources, de calcul (CPU), la mémoire et l'utilisation de la bande passante réseau.

A.4 Impact de la virtualisation sur le temps d'exécution des travaux

La surcharge d'une machine est la différence entre la charge des machines physiques sans aucune machine cliente et sa charge après le démarrage des machines clientes. La charge des machines est le nombre de processus en attente d'exécution par rapport au nombre de CPU.
La figure A.2 présente les taux de surcharge des machines hôtes suite au démarrage des différentes grappes Hadoop. Elle focalise sur l'utilisation des ressources CPU, RAM et le nombre total de processus en attente d'exécution sur les différentes grappes. Elle illustre les trois cas suivants : *(i)* la charge des machines physiques sans l'exécution d'aucune machine cliente, *(ii)* la charge des machines physiques avec des machines virtuelles au repos *(iii)* la charge des machines physiques avec des conteneurs au repos. Une machine au repos est une machine qui n'exécute aucun traitement utilisateur.
Nous remarquons que les machines virtuelles réservent la quantité mémoire nécessaire à leurs exécutions depuis leur mise en route. C'est-à-dire, pour une grappe de 3 machines virtuelles, 17 GB de mémoire RAM sont réservés et pour une grappe de cinq machines virtuelles, 28 GB de mémoire RAM sont réservés. Cependant, Docker utilise presque 5 GB de RAM lors du démarrage d'une grappe de 3 conteneurs et 10 GB pour une grappe de 5 conteneurs. La quantité mesurée avec les conteneurs est la quantité nécessaire pour faire fonctionner les systèmes d'exploitation de la machine hôte (physique), des conteneurs et les entités du système Hadoop. Les résultats de la figure A.2 montrent que la technologie Docker engendre une surcharge des machines physique entre 3% et 5% face à une valeur entre 10% et 25% pour VMware Vsphere.
Dans la grande majorité de nos expériences avec deux machines Hadoop esclaves (VM ou conteneur), l'ajout, à la volet, d'une nouvelles machines esclaves Hadoop améliore le temps d'exécution des travaux. Lorsque nous augmentons le nombres des machines clientes (VM ou conteneurs),

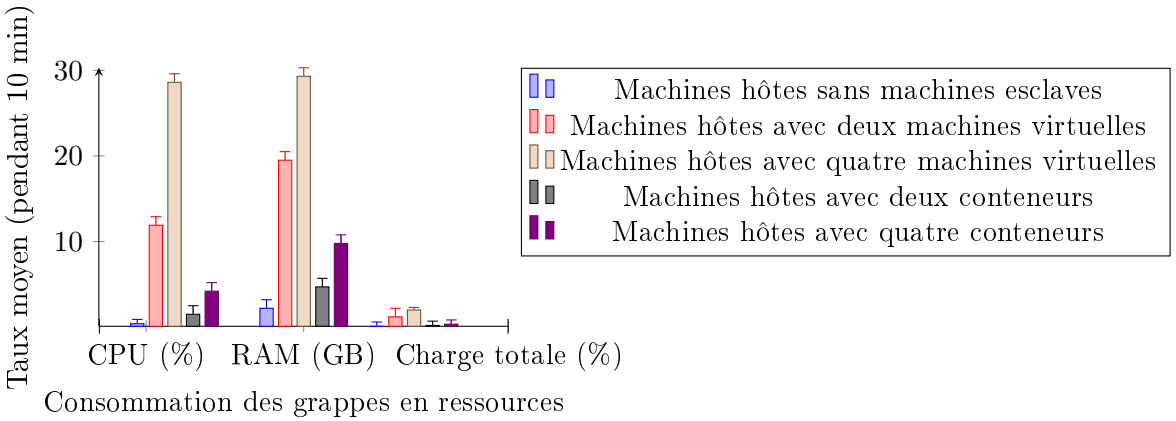
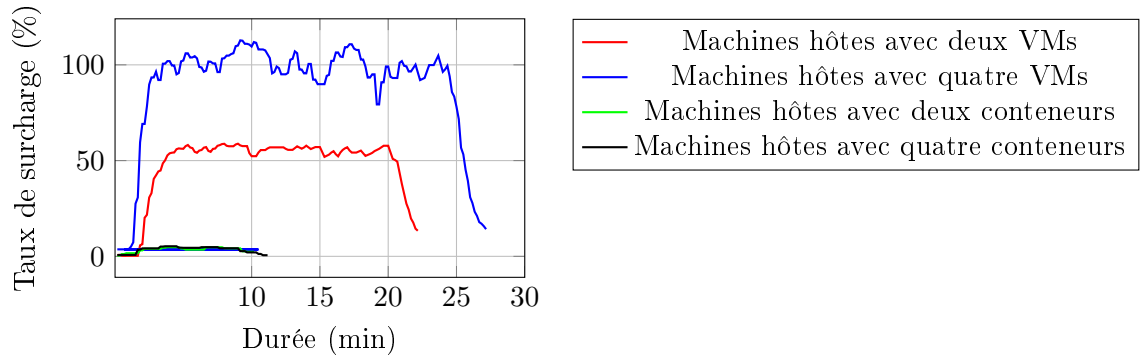


FIGURE A.2 – Mesure du niveau de surcharge avec différents nombres de machines esclaves et avec différents types d'outils de virtualisation

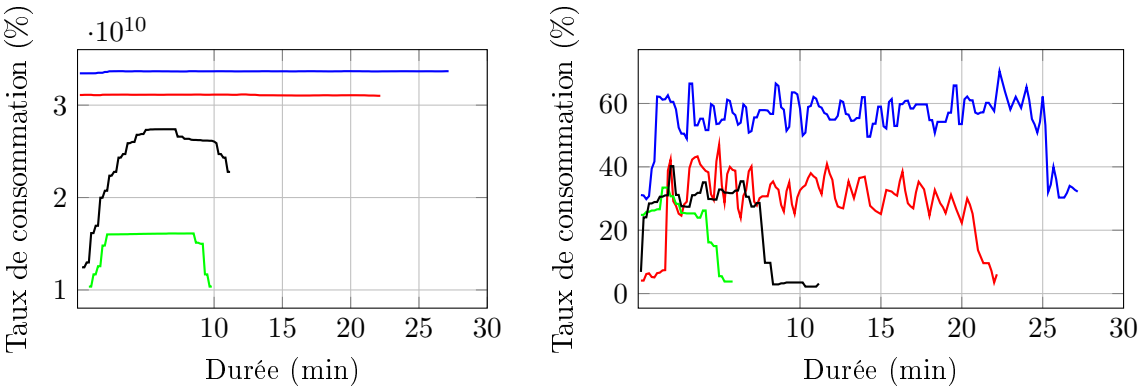
nous nous trouvons dans le cas de congestion de ressources. Par exemple, lors de l'exécution des travaux TestDFSIO, les bandes passantes réseaux et d'entrée/sortie sont les ressources en congestion (figure A.5). Dans le cas du travail TeraGen et avec 4 VM, la ressource en congestion est la mémoire (figure A.3).
En conséquence, on se trouve face au problème de consolidation de serveur qui est largement étudié dans la littérature [144]. Nous concluons à ce niveau que le choix du nombre de machines clientes sur une machine physique est déterminant pour éviter la dégradation des performances. De façon générale, nous remarquons que les deux outils de virtualisation nous permettent de gagner en flexibilité d'exploitation de ressources. Cette flexibilité offre plus de moyens pour faire face à la montée du taux de surcharge des différentes grappes testées. Par exemple, nos mesures montrent que le temps de démarrage d'un conteneur Hadoop configuré est d'en moyenne 20 secondes et que le temps de démarrage d'une VM est d'en moyenne de 1 minute et 30 secondes. Feller et al [54] évaluent les performances de Hadoop en le déployant sur *(i)* des plateformes basées sur la virtualisation lourde et *(ii)* en déploiement directement sur des plateformes physiques. Sur la base des résultats des expériences que nous avons mené et les résultats obtenus par Feller et al, nous concluons que la dégradation de performance est compensée par la flexibilité obtenue suite à l'utilisation de la virtualisation.

A.5 Comparaison des technologies de virtualisation

Pendant les expériences, nous enregistrons le niveau de charge des machines physiques. Ce dernier est un paramètre calculé par Ganglia comme le nombre total des processus en attente par rapport au nombre de CPU. Dans la figure A.3a, nous présentons les résultats d'évaluation des niveaux charges dus à l'exécution du travail TeraGen avec les deux outils de virtualisation. Nous considérons plusieurs tailles de grappes (2 et 4 machines esclaves Hadoop) et 20 GB de données générées. En tenant compte de ses conditions, nous confirmons le fait que les conte-



(a) Taux de surcharge dû à l'exécution de TeraGen



(b) Consommation mémoire due à l'exécution de TeraGen

(c) Consommation CPU due à l'exécution de TeraGen

FIGURE A.3 – Exemple de la consommation des ressources dues à l'exécution du travail TeraGen avec les différents déploiements des grappes Hadoop.

neurs sont moins lourds que les machines virtuelles, ils engendrent un faible niveau de charge (par rapport aux machines virtuelles). Pour deux grappes de même taille et de technologie de virtualisations différentes, la différence entre les niveaux de charge causés par chaque technologie est considérable. La figure A.3b illustre la quantité de mémoire utilisée par chaque grappe durant l'exécution du travail TeraGen. Cette figure illustre la politique de réservation de mémoire que les VMs utilisent. Depuis son démarrage, chaque machine virtuelle réserve la quantité de mémoire qui lui est associée. En plus, chaque machine réserve entre 100 et 200 MB afin que l'hyperviseur les utilise lors de l'interprétation des instructions qui lui sont associées.

Face à la réservation fixe de la mémoire imposée par la technologie de virtualisation lourde, les conteneurs Docker nous permettent de fixer la quantité maximale de mémoire à utiliser par conteneur. Cependant, lorsqu'une quantité de mémoire n'est pas exploitée par un conteneur, elle peut être utilisée par un autre. En cas de besoin, un conteneur, qui n'a pas atteint sa borne maximale autorisée, peut récupérer de la mémoire. Cette récupération demande à respecter un temps d'attente pour permettre à l'application utilisant actuellement cette quantité mémoire de la libérer.

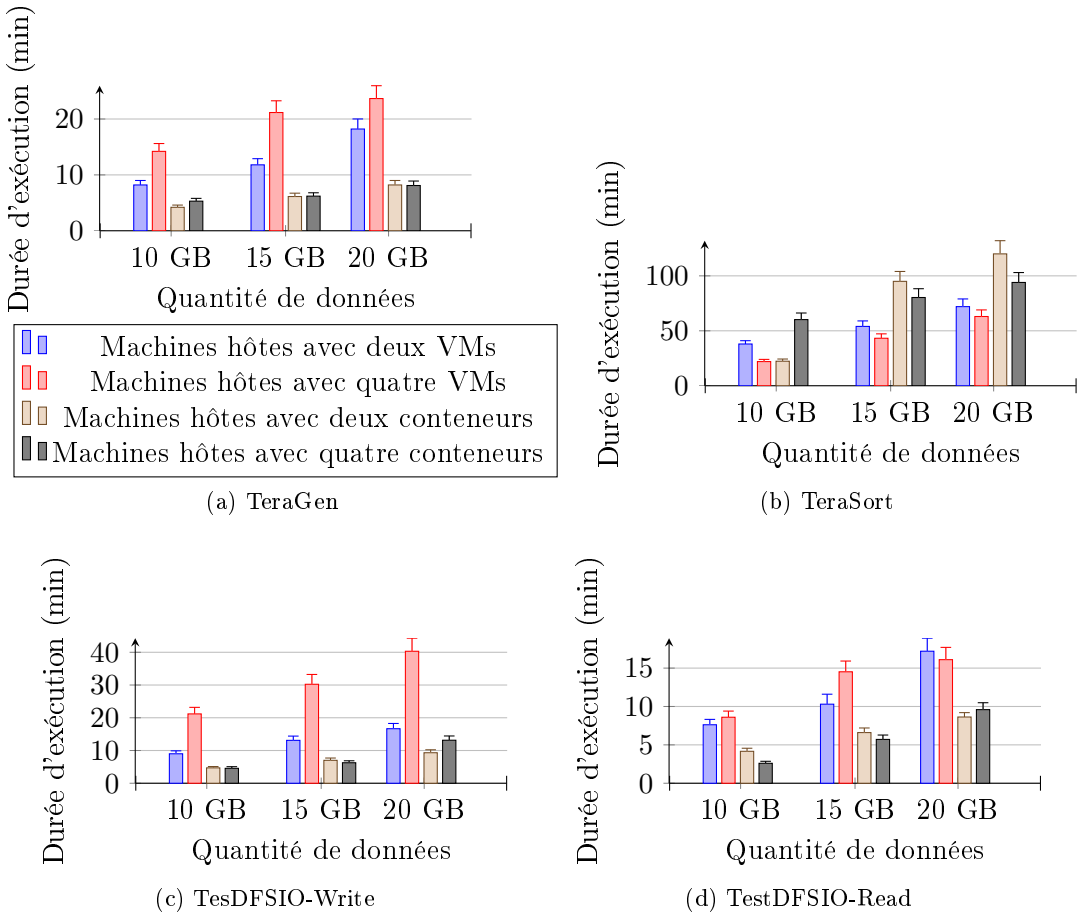


FIGURE A.4 – Durées des exécutions des travaux en fonction de la quantité des données, les tailles des grappes et des technologies de virtualisation utilisées

Nous remarquons que dans ces conditions, les grappes basées sur la technologie Docker sont plus performantes en terme de date de fin d'exécution des travaux par rapport aux grappes basées sur la virtualisation lourde, basée sur des machines virtuelles VMware (Figure A.4a et A.4b). Pour les deux technologies que nous évaluons avec le travail TeraGen, nous remarquons que le fait de faire cohabiter deux conteneurs (respectivement machines virtuelles) fournit des meilleures performances que de faire cohabiter quatre conteneurs (respectivement machines virtuelles) sur la même machine physique.

A.5.1 Étude de l'influence sur l'utilisation des ressources de calcul CPU (grappe homogène)

Afin d'évaluer l'influence de la variation des ressources de calcul CPU dans l'amélioration des performances de Hadoop, nous utilisons les travaux TeraGen et TeraSort. Pendant ces expériences, nous affectons deux cœurs virtuels CPU à chaque machine esclave Hadoop, nous exécutons les travaux cités sur des grappes contenant deux et quatre machines esclaves avec des

tailles de données différentes 10, 15 et 20 GB.
Les deux outils de virtualisation utilisés dans nos évaluations, se basent sur différentes politiques de gestion des ressources de calcul. La politique utilisée par la technologie VMware consiste à affecter un nombre de cœurs virtuels pour chaque machine virtuelle, l'hyperviseur se charge de distribuer la puissance de calcul de la machine physique entre les machines clientes en fonction de ses affectations. La technologie Docker utilise deux politiques pour la gestion des ressources de calcul CPU. La première consiste à associer un nombre de cœurs CPU à chaque conteneur. La deuxième politique consiste à définir un taux de partage relatif entre les conteneurs, c'est-à-dire, associer un poids à chaque conteneur et partager la puissance de calcul entre les conteneurs en fonction de ce poids. Dans toutes les expériences effectuées dans ce chapitre, nous utilisons l'affectation directe des cœurs (virtuel ou non) aux machines clientes. Nous évaluons la politique de partage relatif, implémentée dans Docker, dans la section A.5.3.
La figure A.3 présente la charge, la consommation mémoire et la consommation de la ressource de calcul CPU suite à l'exécution du travail TeraGen. Nous remarquons que l'utilisation des conteneurs Docker diminue de 50% le niveau de charge et le taux d'utilisation des ressources de calcul par rapport à l'utilisation des machines virtuelles VMware. Les figures A.4a et A.4b présentent la durée d'exécution (en moyenne) des travaux TeraGen et TeraSort sur 2 puis 4 machines esclaves Hadoop. Lors de l'exécution du travail TeraGen, nous remarquons que l'utilisation d'une grappe avec deux machines esclaves fournit des meilleurs résultats que l'utilisation d'une grappe avec quatre machines. Ce résultat s'explique par le fait que TeraGen écrit de façon séquentielle des blocs des données sur le support de sauvegarde et augmente la concurrence pour l'accès en écriture sur ce support. Il prolonge, en conséquence, la durée d'exécution du travail. Nous remarquons le même comportement pour le travail TestDFSIO-write (figure A.4c). Dans certains cas, VMware fournit des résultats meilleurs que Docker, tel est le cas lors de l'exécution du travail TeraSort avec 15 et 20 GB (figure A.4a). Pour Docker, les ressources sont utilisées lorsque les éléments Hadoop ont besoin d'elles. Tandis que l'outil VMware procède par réservation de ressources. D'où l'effet négative de la congestion des ressources et de la concurrence à les utiliser.

A.5.2 Étude de l'effet de l'accès concurrent aux supports de sauvegardes et aux ressources réseau (grappe homogène)

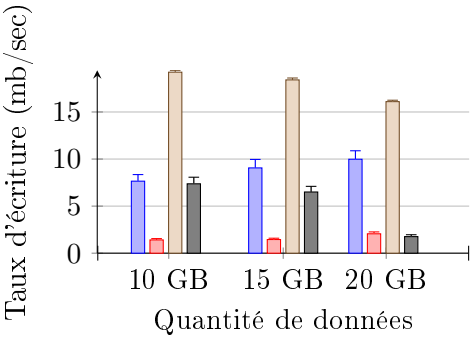
Les travaux TestDFSIO (write et read) sont utilisés pour évaluer l'état du système de fichiers HDFS. Ils focalisent plus, sur l'évaluation des ressources de sauvegarde et le transfert des données à travers le réseau. Nous exécutons les travaux TestDFSIO en respectant les critères suivants :

— lancer les exécutions sur des grappes contenant deux et quatre machines esclaves Hadoop,

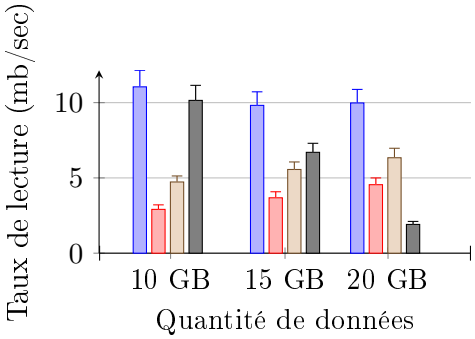
— lancer les exécutions avec les deux technologies de virtualisation,

— lancer les exécutions sur plusieurs tailles de données (10, 15 et 20 GB).

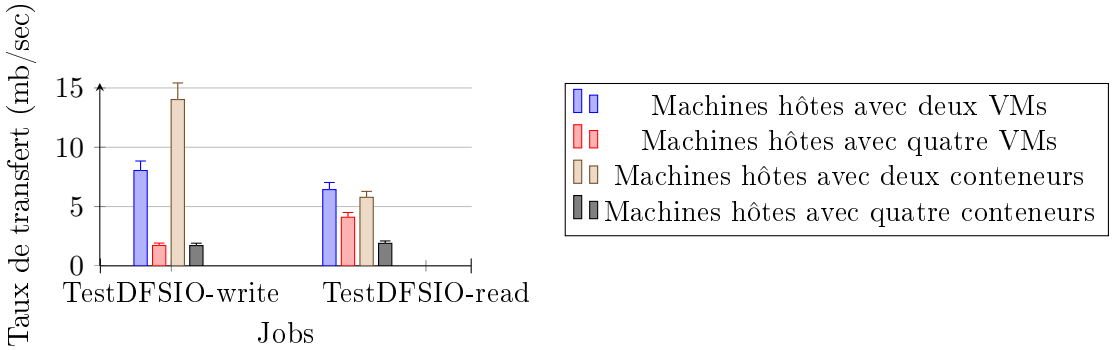
Nous présentons dans la figure A.5 la moyenne du taux de transfert des données envoyées à travers le réseau (figure A.5c) et la moyenne de la vitesse d'écriture/ lecture disque (figures A.5a et A.5b). Au fur et à mesure que le nombre de machines esclaves par machine physique augmente,



(a) Moyennes des taux d'écriture en exécutant le travail TestDFSIO-Write



(b) Moyennes des taux de lecture en exécutant le travail TestDFSIO-Read



(c) Moyennes des taux de transfert de données en exécutant le travail TestDFSIO-Write (20 GB)

FIGURE A.5 – Taux de transfert réseau, de lecture et d'écriture disque enregistrés suite à l'exécution des travaux TestDFSIO avec les deux outils de virtualisation

la charge relevée suite à l'exécution des travaux augmente et les mesures de la vitesse d'accès aux supports de sauvegarde et de transfert réseau, diminuent. Par exemple, lors de l'exécution des travaux TestDFIO sur une grappe de 4 machines esclaves Hadoop et sur 20 GB de données. Au fur et à mesure que la surcharge des machines augmente jusqu'à 85%, la vitesse d'accès aux supports de sauvegarde et la vitesse de transfert réseau décroissent pour atteindre leurs plus bas niveaux.
Nous remarquons lors de ces expériences les limites de la virtualisation par conteneur. Pour des grappes de même taille, les vitesses de lecture/ écriture disque et de transfert réseaux n'ont pas un comportement uniforme ni en fonction de la taille des grappes ni en fonction de la taille des données. Par exemple, dans les figures A.5a et A.5b, nous remarquons que :

— Plus nous augmentons le nombre de machines esclaves, plus faible est la moyenne des vitesses d'écriture et de lecture sur le support de stockage. En même temps et pour une grappe de même taille, plus nous augmentons la taille des données à traiter, plus faible est la moyenne des taux d'écriture sur le support de stockage.

— Pour la technologie VMware et en analysant le débit de transfert réseau, nous remarquons

que plus faible est le nombre de machines esclaves par machine physique plus fort est le débit de transfert des données à travers le réseau.

Les résultats obtenus de ces expériences sont :

— La moyenne des taux de transfert réseau dépendent du niveau de surcharge de la machine physique, ce qui confirme la nature critique de la gestion de la bande passante réseau dans le cas des applications de gestion de gros volumes de données.

— Les deux critères (accès aux supports de sauvegarde et aux ressources réseau) ne sont pas pertinents pour être utilisé pour comparer les deux outils de virtualisation.

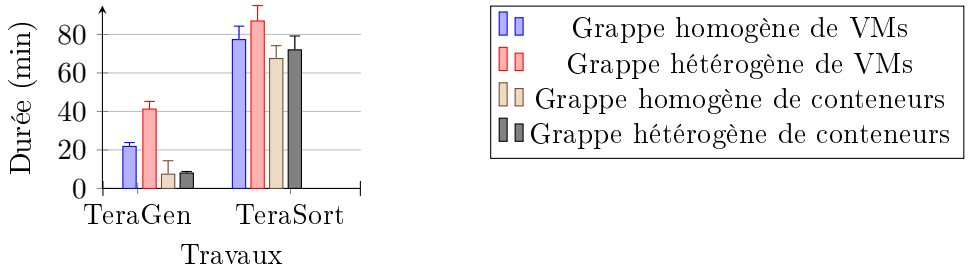
A.5.3 Étude de la variation de performance de Hadoop sur les grappes hétérogènes

Nous nous intéressons dans cette section à l'influence de l'utilisation d'une grappe hétérogène sur la performance du logiciel Hadoop, en terme de date de fin des travaux. J'ai organisé les expériences en trois étapes, la première consiste à utiliser deux configurations différentes des machines clientes qui constituent une grappe (tableau A.2). Les deux étapes restantes focalisent sur la technologie Docker. La deuxième étape consiste à étudier l'influence de la variation de la quantité de chaque ressource sur les performances de Hadoop. Par exemple, tous les conteneurs de la grappe ont les mêmes quantités de ressources sauf la ressource mémoire qui varie entre deux VM. Dans les deux étapes, j'utilise une grappe de quatre machines dont trois sont des machines esclaves Hadoop. Les résultats obtenus des grappes hétérogènes sont évalués par rapport aux résultats obtenus depuis des grappes homogènes avec le même nombre de machines. La dernière étape focalise sur les politiques de gestion des ressources de calcul dans Docker.

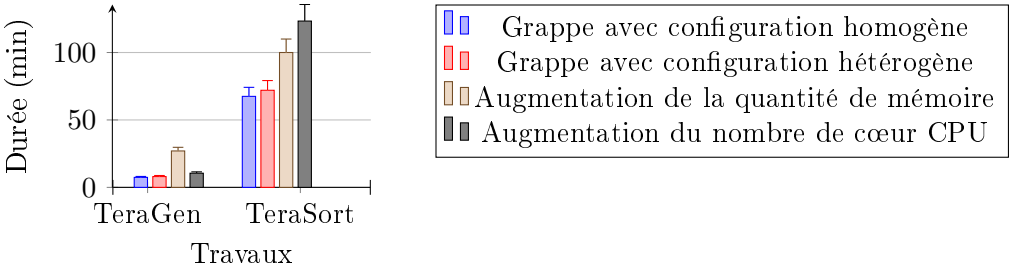
Dans la première étape, les configurations des différentes machines de la grappe sont affichées dans le tableau A.1 pour les grappes homogènes et le tableau A.2 pour les grappes hétérogènes. La figure A.6a permet de comparer les durées d'exécution des travaux TeraGen et TeraSort. Les résultats confirment que Hadoop fournit de meilleure performance quand il est déployé sur des grappes homogènes. La figure A.6b présente les résultats de l'exécution des travaux TeraGen et TeraSort dans les cas d'utilisation suivants :

1. Grappe de trois conteneurs homogènes.
2. Grappe dont la configuration est affichée dans le tableau A.2.
3. Grappe de trois machines dont toutes les quantités de ressources sont identiques à l'exception de la mémoire RAM. Nous augmentons la quantité mémoire des machines.
4. Grappe de trois machines dont toutes les quantités de ressources sont identiques à l'exception du nombre de cœurs CPU affectés par machines esclaves (conteneurs ou machines virtuelles)

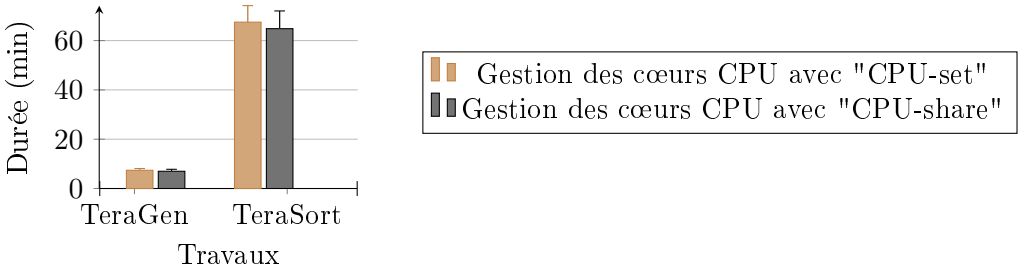
L'augmentation de la quantité mémoire par conteneurs diminue la performance de la grappe Hadoop. En analysant le comportement des machines physiques. Nous remarquons que l'augmentation de la capacité mémoire des conteneurs pénalise le système d'exploitation du système hôte et réduit la quantité mémoire qui lui est réservée. Le même raisonnement est remarqué lors



(a) Influence des grappes hétérogènes sur les performances de Hadoop



(b) Influence de la variation d'une ressource particulière sur les performances des grappes Hadoop déployées avec Docker



(c) Comparaison entre les deux politiques de gestion de la puissance de calcul offertes par la technologie Docker

FIGURE A.6 – Étude de la durée d'exécution des travaux TeraGen et TeraSort sur des grappes hétérogènes

de l'augmentation de la quantité des cœurs CPU affectée à chaque conteneur. Cette augmentation pénalise le système d'exploitation du système hôte et réduit le nombre de cœurs qui sont mis à sa disposition. De ce fait, cette expérience montre que :

— il est conseillé d'utiliser des machines clientes (VMs ou conteneurs) de même configuration.

— l'augmentation de certaines ressources (exemple mémoire) de quelques machines virtuelles d'une grappe Hadoop ne peut que réduire les performances de la grappe. Cette dégradation de performance est due à la croissance de la demande sur les ressources des machines physiques.

— Le bon choix du nombre de conteneurs (ou machines virtuelles) par machine physique est un élément déterminant pour garantir les meilleures performances d'une grappe Hadoop installée sur une plateforme virtuelle.

Dans la figure A.6c, nous focalisons sur la technologie Docker et ses politiques de gestion des ressources de calculs. Nous utilisons une grappe constituée de deux machines esclaves. En fonction des environnements utilisés, les résultats montrent que le gap entre les deux politiques est faible. La politique de partage relatif (CPU-share) fournit de meilleurs résultats que la politique basée sur l'affectation directe de cœurs (CPU-Set). La politique "CPU-share" offre la possibilité d'utiliser les ressources de calculs inutilisées, soit en les partageant entre les conteneurs soit en les affectant aux applications de la machine hôte. Dans le contexte Hadoop, la politique "CPU-share" augmente la capacité des cœurs virtuels, cette augmentation de puissance n'a pas d'effet négatif sur le système hôte puisqu'elle n'utilise que la puissance non encore exploitée. Lorsque le niveau de surcharge de la machine hôte est élevé (plus de 80%), les deux politiques fournissent les mêmes performances.

A.6 Étude de l'influence des technologies de virtualisation sur la consommation énergétique

Une grappe Hadoop chez Yahoo ! dépasse les 4000 serveurs, Facebook déploie Hadoop sur une grappe de 600 serveurs, General Electric déploie Hadoop sur une grappe de 1700 serveurs. Á cette échelle, la consommation énergétique est un facteur important puisqu'elle a une influence considérable sur le coût d'exploitation d'une grappe. Plusieurs travaux de recherche, tels que les travaux réalisés par l'U.S. Environmental Agency [7] et le Natural Resources Defense Council [39] présentés en 2014, montrent que le coût énergétique dû à l'exploitation des grappes informatiques est très élevé et ne cesse d'augmenter d'une année à une autre. En 2008, l'Union Européen a défini une feuille de route [56] à suivre pour assurer la meilleure efficacité des centres de données. À travers l'étude de la consommation énergétique, je voulais étudier l'influence de l'utilisation des deux outils de virtualisation sur la consommation énergétique d'une grappe Hadoop. Nous enregistrons la consommation des grappes pendant l'exécution des travaux TeraGen et TeraSort. À travers ces expériences, il est clair que la variation de niveau de surcharge de la grappe et la variation de la consommation énergétique sont proportionnelles. Lorsque nous comparons une grappe contenant quatre machines esclaves par machine physique qu'on nomme "grappe1", à une grappe contenant deux machines esclaves par machine physique,qu'on nomme "grappe2". Nous remarquons que la consommation énergétique de "grappe1" est plus élevée que la consommation de "grappe2". Lorsque le niveau de surcharge, d'une grappe, est à son maximum (plus de 80%), sa consommation énergétique est à son maximum.
La figure A.7 présente les résultats de l'exécution du travail TeraSort sur des grappes de quatre machines clientes (VMs ou conteneurs) et les résultats de l'exécution du travail TeraGen sur deux machines esclave. Ces résultats montrent que :

— les grappes basées sur la technologie VMware consomment plus que les grappes basées sur les grappes Docker. Ce résultat s'explique par le fait que les conteneurs Docker engendrent un niveau de surcharge inférieur à celui engendré par les VMs. Cette différence du niveau de surcharge entre les deux outils évalués explique la consommation électrique des deux grappes.

— le niveau de surcharge des machines physiques influence directement les performances des grappes ainsi que leur consommation énergétique.

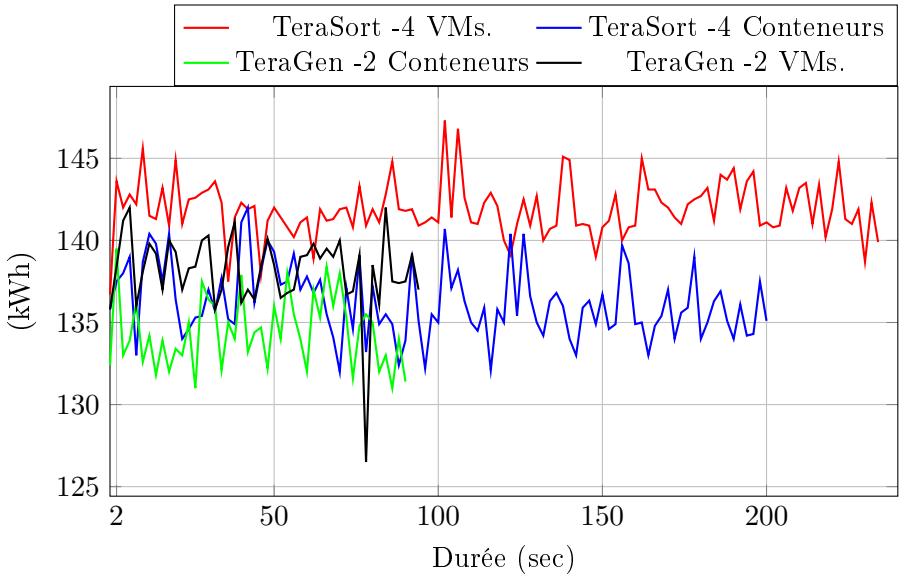


FIGURE A.7 – Consommation énergétique de différentes tailles de grappes suite à l'exécution des travaux TeraGen et TeraSort

Rappelons que le niveau de surcharge enregistré par Ganglia est la moyenne du nombre des processus en attente de la libération des ressources sur la grappe. C'est-à-dire, plus le niveau de surcharge est élevé, plus élevée sera l'utilisation des ressources. Cette utilisation de ressources influe directement sur la variation de la consommation énergétique.
De ce fait, la consommation énergétique d'une grappe hadoop dépend des travaux à lancer. La technologie de virtualisation Docker permet de la réduire en limitant le niveau de surcharge des machines physiques. Nous avons enregistré un gain relatif de consommation moyenne d'environ 5%, ce qui présente un gain important pour une grappe de cette taille.

A.7 Discussion

L'objectif de ce travail est d'étudier la capacité de la virtualisation légère à être utilisée dans le déploiement des applications de gestion de gros volumes de données. De ce fait, nous nous intéressons à *(i)* l'étude des performances des grappes virtuelles Hadoop, *(ii)* l'étude de la consommation énergétique des grappes Hadoop, *(iii)* comparer les technologies de virtualisation. Les principaux résultats de notre étude sont :

— les résultats des expériences que nous avons mené montrent qu'une grappe Hadoop est plus performante lorsqu'elle est déployée avec la technologie Docker qu'avec la technologie VMware. En moyenne, les conteneurs Docker améliorent les performances de Hadoop de 170% en fonction des travaux exécutés. Par exemple, *(i)* l'exécution du travail TeraGen

avec 20 GB de données est 2.4 fois plus rapide avec les conteneurs Docker qu'avec les machines virtuelles VMware,*(ii)* l'exécution du travail TeraSort est 150% plus rapide sur les conteneurs.

— la technologie Docker est prête pour être utilisée dans le déploiement des applications de gestion de gros volumes de données.

— la technologie de virtualisation légère, et particulièrement Docker, permet de réduire en moyenne de 5% (au maximum 8%) la consommation énergétique des grappes Hadoop comparé aux plateformes basées sur la virtualisation lourde.

— le choix du nombre de machines esclaves par machine physique est déterminant pour garantir les meilleures performances. Il doit faire l'étude d'expérimentations spécifiques en fonction des jobs à exécuter.

— Les politiques de gestion de ressources (mémoire, CPU) dans Docker sont matures pour être utilisées pour le déploiement des infrastructures Hadoop.

Les logiciels de gestion de gros volumes de données sont confrontés à la taille des données qu'ils traitent. De ce fait, les architectures matérielles des plateformes, sur lesquelles ces logiciels sont déployés, influencent directement leurs performances.
Nos expérimentations se basent sur le logiciel Hadoop, présenté dans le chapitre 1. L'ordonnanceur de Hadoop considère que la grappe utilisée est homogène lorsqu'il affecte les tâches aux machines qui vont assurer leurs traitements. Il ne considére que les ressources mémoire "RAM" et de calcul "CPU" lors de l'affectation des tâches. En utilisant l'outil Docker, les résultats de nos expériences prouvent l'amélioration des performances d'un facteur 1.7% en moyenne (par rapport à VMware). Les politiques de gestion des ressources RAM et CPU utilisées par Docker ont un rôle important pour l'obtention de ces résultats. Par exemple, si les éléments Hadoop n'utilisent pas la quantité de mémoire, qui lui sont réservés, les conteneurs Docker la libèrent. Elle peut, en conséquence, être utilisée soit par d'autres conteneurs soit par le système hôte pour faire face au monté de niveau de surcharge.
De la même manière, les politiques de gestion CPU utilisées par Docker améliorent la puissance du logiciel Hadoop. Du point de vue architecture, les éléments de calcul Hadoop manipulent des cœurs virtuels pour exécuter les tâches et chaque élément est déployé dans un conteneur. En conséquence, la capacité de calcul non utilisé peut être partagée par les conteneurs tournant sur la même machine physique et qui exécutent des traitements. Quand des tâches sont affectées à un conteneur au repos, ce dernier récupère la quantité de mémoire qu'il a libérée. La récupération des ressources est gérée de manière à limiter les dégâts, par exemple l'envoie d'un signal sur la machine locale, et marquer un temps d'attente pour que le conteneur utilisant des ressources en excès les libère.
Nos expériences prouvent le lien direct entre l'augmentation du niveau de surcharge de la machine physique et l'augmentation de la consommation énergétique. Elles montrent que les conteneurs engendrent, en moyenne, 40% de surcharge en moins que les machines virtuelles. En plus, Ils montrent que les conteneurs Docker engendrent une baisse de l'utilisation des ressources de calcul de 50% par rapport aux VMs. La diminution des niveaux de surcharge des ressources des machines physiques influence positivement la performance et la consommation énergétique (élec-

trique).
L'accès aux supports de sauvegarde est un facteur de faiblesse pour les deux technologies. Lors de l'exécution d'un travail plusieurs fois dans les mêmes conditions, nous remarquons que les variations de la bande passante enregistrées varie considérablement d'une exécution à une autre. En conséquence, ce problème reste ouvert. Une des solutions premières, pour réduire son influence, consiste à augmenter le nombre de disques durs par machine physique afin d'augmenter le degré de parallélisme et augmenter en conséquence la vitesse d'accès à ses supports.
Le déploiement d'une grappe Hadoop dans des conteneurs installés sur des machines virtuelles est un cas d'utilisation qui a été évalué dans les travaux de Verma et al. [146], les résultats confirment que l'utilisation jointe des deux technologies dans une architecture complexe engendre des dégradations importantes en performances et des difficultés de gestion de la grappe.
Avec Docker, le démarrage d'un conteneur configuré se mesure en seconde, à la fin de son utilisation, il suffit de le supprimer. Le temps nécessaire pour le déploiement d'une grappe Hadoop avec Docker se mesure en seconde, ce qui augmente les avantages de l'outil pour être utilisé avec Hadoop. Lorsque la dégradation des performances de la grappe Hadoop est remarquée, suite à la montée en charge, l'ajout de nouveaux conteneurs limitera cette dégradation en quelques secondes. Ce qui permettra à Docker de faciliter la mise en place d'un système Hadoop élastique et performant.

A.8 Conclusion

Ce chapitre est une étude exploratoire, son objectif est : *(i)* de comparer deux outils de virtualisaiton pour étudier la possibilité d'utiliser la technologie Docker lors du déploiement des grappes Hadoop. Ensuite, nous focalisons sur Docker pour étudier les politiques de gestion de ressources et d'ordonnancement des tâches qu'il utilise. *(ii)* d'étudier la consommation énergétique des grappes Hadoop basées sur ces outils.
Nos expérimentations font partie des premières études détaillées qui, d'une part, évaluent le déploiement de Hadoop avec Docker, d'autre part, comparent Docker à une autre technologie considérée mature pour être utilisée dans le processus de production des entreprises.
La plupart des expériences confirment la capacité de Docker à répondre aux besoins de l'entreprise Group Cyrès en terme d'amélioration de performance et de réduction de coût d'exploitation. Ces expérimentations nous ont permis d'orienter nos travaux de recherche et de proposer une solution d'ordonnancement pour l'amélioration des performances et de l'exploitation des ressources. Actuellement, l'entité Ingensi de l'entreprise Group Cyrès utilise Docker pour le déploiement des grappes Hadoop pour ses clients.

Annexe B

Publications Personnelles

B.1 Publications Internationales

Jlassi A. and Martineau P. (2016). Benchmarking Hadoop Performance in the Cloud - An in Depth Study of Resource Management and Energy Consumption. In Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 2 : CLOSER, Rome, Italy, ISBN 978-989-758-182-3, pages 192-201.

Jlassi A. and Martineau P. (2016). Virtualization technologies for the big data environment. In Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16). ACM, New York, NY, USA, pages 542-545.

Jlassi A., Martineau P. and Tkindt V. (2015). Offline Scheduling of Map and Reduce Tasks on Hadoop Systems. In Proceedings of the 5th International Conference on Cloud Computing and Services Science - Volume 1 : CLOSER, Lisbon, Portugal, ISBN 978-989-758-104-5, pages 178-185.

B.2 Publications Nationales

Jlassi A., Martineau P. (2016). Problème d'ordonnancement dans Hadoop, ordonnancement sur des machines parallèles équivalentes et distribuées. In : Conférence d'informatique en Parallélisme, Architecture et Système, Compas'2016, 5-8 Juillet 2016, Lorient, France.

Jlassi A., Martineau M., T'Kindt V. (2016). Optimisation du Problème d'ordonnancement à Machines Parallèles dans Hadoop. In : Le 17^{me} congrès annuel de la Société française de recherche opérationnelle et d'aide à la décision (ROADEF 2016), 10-12 Février 2016, Compiègne, France.

Jlassi A., Martineau P., T'Kindt V. (2014). Modélisation Mathématique du problème d'ordonnancement dans Hadoop. In : Le 15^{me} congrès annuel de la Société française de recherche opérationnelle et d'aide à la décision (ROADEF 2014), 26-28 Février 2014, Bordeaux, France.

B.3 Chapitre dans un livre

Jlassi A. and Martineau P. Experimental Study on Performance and Energy Consumption of Hadoop in Cloud Environments, In : Cloud Computing and Services Science. Springer International Publishing, 2017 , Cham, pages 255–272.

Bibliographie

[1] *Cloud Computing : First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings*, chapter Evaluating MapReduce on Virtual Machines : The Hadoop Case, pages 519–528. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[2] *Data Management in Cloud, Grid and P2P Systems : 5th International Conference, Globe 2012, Vienna, Austria, September 5-6, 2012. Proceedings*, chapter MapReduce Applications in the Cloud : A Cost Evaluation of Computation and Storage, pages 37–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[3] Alan B. Pritsker A., Lawrence J. Waiters, and Philip M. Wolfe. Multiproject scheduling with limited resources : A zero-one programming approach. *Management Science*, 16(1) :93–108, 1969.

[4] Ashraf Aboulnaga, Ziyu Wang, and Zi Ye Zhang. Packing the most onto your cloud. In *Proceedings of the First International Workshop on Cloud Data Management*, CloudDB '09, pages 25–28, New York, NY, USA, 2009. ACM.

[5] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb : An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1) :922–933, August 2009.

[6] Russell L. Ackoff. *Science in the Systems Age : Beyond IE, OR, and MS*, pages 325–335. Springer US, Boston, MA, 1991.

[7] U.S. Environmental Protection Agency. Report to congress on server and data center energy efficiency. 2007.

[8] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu : Optimizing mapreduce on heterogeneous clusters. *SIGARCH Comput. Archit. News*, 40(1) :61–74, March 2012.

[9] Miriam Allalouf and Yuval Shavitt. *Maximum Flow Routing with Weighted Max-Min Fairness*, pages 278–287. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[10] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 265–278, Berkeley, CA, USA, 2010. USENIX Association.

[11] Julio C.S. Anjos, Iván Carrera, Wagner Kolberg, Andre Luis Tibola, Luciana B. Arantes, and Claudio R. Geyer. Mra++ : Scheduling and data placement on mapreduce for heterogeneous environments. *Future Generation Computer Systems*, 42 :22 – 35, 2015.

[12] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *J. ACM*, 45(5) :753–782, September 1998.

[13] Christian Artigues, Peter Brucker, Sigrid Knust, Oumar Koné, Pierre Lopez, and Marcel Mongeau. A note on "event-based MILP models for resource-constrained project scheduling problems". *Computers & OR*, 40(4) :1060–1063, 2013.

[14] Christian Artigues, Philippe Michelon, and Stéphane Reusser. Insertion techniques for static and dynamic resource constrained project scheduling. *European Journal of Operational Research*, 149 :249–267, 2003.

[15] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Speed scaling to manage energy and temperature. *J. ACM*, 54(1) :3 :1–3 :39, March 2007.

[16] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[17] Richard Bellman. Letter to the editor—comment on dantzig's paper on discrete variable extremum problems. *Operations Research*, 5(5) :723–724, 1957.

[18] Dimitri Bertsekas and Robert Gallager. *Data Networks (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[19] P. Brucker and S. Knust. *Complex Scheduling*. GOR-Publications. Springer Berlin Heidelberg, 2013.

[20] Alain Busser and Florian Tobé. Problèmes de remplissage de seaux. http://irem.univ-reunion.fr/spip.php?article715, 2013. last accessed 16 January 2016.

[21] Henri Casanova, Arnaud Legrand, and Yves Robert. *Parallel Algorithms*. Chapman and Hall/CRC Press, 2008.

[22] V. Černý. Thermodynamical approach to the traveling salesman problem : An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1) :41–51, 1985.

[23] H. Chang, M. Kodialam, R. R. Kompella, T. V. Lakshman, M. Lee, and S. Mukherjee. Scheduling in mapreduce-like systems for fast completion time. In *2011 Proceedings IEEE INFOCOM*, pages 3074–3082, April 2011.

[24] Anna Charny and Raj Jain. Congestion control with explicit rate indication. In *PROC. ICC'95*, pages 1954–1963, 1995.

[25] Po-Cheng Chen, Yen-Liang Su, Jyh-Biau Chang, and Ce-Kuen Shieh. *Variable-Sized Map and Locality-Aware Reduce on Public-Resource Grids*, pages 234–243. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[26] Quan Chen, Minyi Guo, Qianni Deng, Long Zheng, Song Guo, and Yao Shen. Hat : history-based auto-tuning mapreduce in heterogeneous environments. *The Journal of Supercomputing*, 64(3) :1038–1054, 2013.

[27] Shigang Chen and Klara Nahrstedt. Maxmin fair routing in connection-oriented networks. In *in Proceedings of Euro-Parallel and Distributed Systems Conference (Euro-PDS '98*, pages 163–168, 1998.

[28] Yanpei Chen, Sara Alspaugh, Dhruba Borthakur, and Randy Katz. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 43–56, 2012.

[29] Yanpei Chen, Laura Keys, and Randy H. Katz. Towards energy efficient mapreduce. Technical report, EECS Department, University of California, Berkeley, Aug 2009.

[30] Nathanaël Cheriere, Pierre Donat-Bouillud, Shadi Ibrahim, and Matthieu Simonin. On the Usability of Shortest Remaining Time First Policy in Shared Hadoop Clusters. In *SAC 2016-The 31st ACM/SIGAPP Symposium on Applied Computing*, Pisa, Italy, April 2016.

[31] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1) :1–14, June 1989.

[32] Nicos Christofides, R. Alvarez-Valdes, and J. M. Tamarit. Project scheduling with resource constraints : A branch and bound approach. *European Journal of Operational Research*, 29(3) :262–273, 1987.

[33] Edward Grady Coffman. *Computer and job-shop scheduling theory.* Wiley, New York, 1976. A Wiley-Interscience publication.

[34] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. Mad skills : New analysis practices for big data. *Proc. VLDB Endow.*, 2(2) :1481–1492, August 2009.

[35] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. In Reinhard Männer and Bernard Manderick, editors, *PPSN*, pages 515–526. Elsevier, 1992.

[36] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 1–21. USENIX Association, 2010.

[37] cores. Getting started with systemd. https://coreos.com/docs/launching-containers/launching/getting-started-with-systemd/, 2015. Last accessed 16 March 2016.

[38] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[39] Natural Resources Defense Council. American data centers are wasting huge amounts of energy. www.nrdc.org/energy, 2014. last accessed 16 January 2016.

[40] Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of the 8th Conference on Visualization '97*, VIS '97, pages 235–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[41] Teodor Gabriel Crainic, Bertrand Le Cun, and Catherine Roucairol. *Parallel Branch-and-Bound Algorithms*, pages 1–28. John Wiley and Sons, Inc., 2006.

[42] Jeffrey Dean and Sanjay Ghemawat. Mapreduce : Simplified data processing on large clusters. *Commun. ACM*, 51(1) :107–113, January 2008.

[43] Jeffrey Dean and Sanjay Ghemawat. Mapreduce : Simplified data processing on large clusters. *Commun. ACM*, 51(1) :107–113, January 2008.

[44] Avi Dechter and Rina Dechter. On the greedy solution of ordering problems. *ORSA Journal on Computing*, 1(3) :181–189, 1989.

[45] Advanced Micro Devices. Hadoop performance tuning guide - amd, October 2012.

[46] Stefan Droste, Thomas Jansen, and Ingo Wegener. Natural computing optimization with randomized search heuristics—the (a)nfl theorem, realistic scenarios, and difficult functions. *Theoretical Computer Science*, 287(1) :131 – 144, 2002.

[47] Witold Bednorz (ed.). Advances in greedy algorithms, 2008.

[48] Corinne Erhel and Laure de la Raudière. Rapport d'information sur le développement de l'économie numérique française. Technical report, Commission des affaires économiques, 2014.

[49] Sink Eric. Directed acyclic graphs (dags), 2011.

[50] Z. Fadika, M. Govindaraju, R. Canon, and L. Ramakrishnan. Evaluating hadoop for data-intensive scientific operations. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 67–74, June 2012.

[51] Z. Fadika, M. Govindaraju, R. Canon, and L. Ramakrishnan. Evaluating hadoop for data-intensive scientific operations. *IEEE CLOUD 2012*, 2012.

[52] Zacharia Fadika, Elif Dede, Jessica Hartog, and Madhusudhan Govindaraju. Marla : Mapreduce for heterogeneous clusters. *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 0 :49–56, 2012.

[53] Michal Feldman, Kevin Lai, and Li Zhang. A price-anticipating resource allocation mechanism for distributed shared clusters. In *Proceedings of the 6th ACM Conference on Electronic Commerce*, EC '05, pages 127–136, New York, NY, USA, 2005. ACM.

[54] Eugen Feller, Lavanya Ramakrishnan, and Christine Morin. Performance and energy efficiency of big data applications in cloud environments : A hadoop case study. *Journal of Parallel and Distributed Computing*, 79 :80 – 89, 2015. Special Issue on Scalable Systems for Big Data Management and Analytics.

[55] Gábor Fodor, Gábor Malicskó, Michal Pióro, and Tomasz Szymanski. Path optimization for elastic traffic under fairness constraints. In Nelson L.S. da Fonseca Jorge Moreira de Souza and Edmundo A. de Souza e Silva, editors, *Teletraffic Engineering in the Internet EraProceedings of the International Teletraffic Congress - ITC-I7*, volume 4 of *Teletraffic Science and Engineering*, pages 667 – 680. Elsevier, 2001.

[56] Institute for Energy and Transport (IET). Data centres energy efficiency. http://iet.jrc.ec.europa.eu/energyefficiency/ict-codes-conduct/data-centres-energy-efficiency, 2015. Last accessed 16 March 2016.

[57] Dimitris Fotakis, Ioannis Milis, Orestis Papadigenopoulos, Emmanouil Zampetakis, and Georgios Zois. *Scheduling MapReduce Jobs and Data Shuffle on Unrelated Processors*, pages 137–150. Springer International Publishing, Cham, 2015.

[58] Apache Software Foundation. Hadoop : Capacity scheduler. https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html, 2016.

[59] Michael R. Garey and David S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[60] R. Ge, X. Feng, W. c. Feng, and K. W. Cameron. Cpu miser : A performance-directed, run-time system for power-aware clusters. In *2007 International Conference on Parallel Processing (ICPP 2007)*, pages 18–18, Sept 2007.

[61] Arthur M. Geoffrion. *Lagrangian Relaxation for Integer Programming*, pages 243–281. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[62] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5) :29–43, October 2003.

[63] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andrew Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness : Fair allocation of multiple resource types. Technical Report UCB/EECS-2011-18, EECS Department, University of California, Berkeley, Mar 2011.

[64] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy : Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 365–378, New York, NY, USA, 2013. ACM.

[65] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy : Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 365–378, New York, NY, USA, 2013. ACM.

[66] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[67] Z. Guo, G. Fox, M. Zhou, and Y. Ruan. Improving resource utilization in mapreduce. In *2012 IEEE International Conference on Cluster Computing*, pages 402–410, Sept 2012.

[68] Mohammad Hammoud, M. Suhail Rehman, and Majd F. Sakr. Center-of-gravity reduce task scheduling to lower mapreduce network traffic. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 49–58, Washington, DC, USA, 2012. IEEE Computer Society.

[69] Mohammad Hammoud and Majd F. Sakr. Locality-aware reduce task scheduling for mapreduce. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 570–576, Washington, DC, USA, 2011. IEEE Computer Society.

[70] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst.*, 21(2) :207–233, May 2003.

[71] J. Hartog, R. Delvalle, M. Govindaraju, and M. J. Lewis. Configuring a mapreduce framework for performance-heterogeneous clusters. In *2014 IEEE International Congress on Big Data*, pages 120–127, June 2014.

[72] Chen He, Ying Lu, and David Swanson. Matchmaking : A new mapreduce scheduling technique. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 40–47, Washington, DC, USA, 2011. IEEE Computer Society.

[73] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite : Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51, March 2010.

[74] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite : Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010*, pages 41–51, March 2010.

[75] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi. Leen : Locality/fairness-aware key partitioning for mapreduce in the cloud. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 17–24, Nov 2010.

[76] Shadi Ibrahim, Hai Jin, Lu Lu, Bingsheng He, Gabriel Antoniu, and Song Wu. Maestro : Replica-aware map scheduling for mapreduce. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 435–442, Washington, DC, USA, 2012. IEEE Computer Society.

[77] Intel. Intel virtualization technology (intel vt). http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html, 2015. Last accessed 5 March 2016.

[78] A. Iordache, C. Morin, N. Parlavantzas, E. Feller, and P. Riteau. Resilin : Elastic mapreduce over multiple clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 261–268, May 2013.

[79] S. Irani, S. Shukla, and R. Gupta. Competitive analysis of dynamic power management strategies for systems with multiple power saving states. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 117–123, 2002.

[80] Sandy Irani and Anna R. Karlin. Approximation algorithms for np-hard problems. chapter Online Computation, pages 521–564. PWS Publishing Co., Boston, MA, USA, 1997.

[81] Sandy Irani and Kirk R. Pruhs. Algorithmic problems in power management. *SIGACT News*, 36(2) :63–76, June 2005.

[82] Nusrat Sharmin Islam, Xiaoyi Lu, Md. Wasi-ur Rahman, Jithin Jose, and Dhabaleswar K. (DK) Panda. *Specifying Big Data Benchmarks : First Workshop, WBDB 2012, San Jose, CA, USA, May 8-9, 2012, and Second Workshop, WBDB 2012, Pune, India, December 17-18, 2012, Revised Selected Papers*, chapter A Micro-benchmark Suite for Evaluating HDFS Operations on Modern Clusters, pages 129–147. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[83] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce : An in-depth study. *Proc. VLDB Endow.*, 3(1-2) :472–483, September 2010.

[84] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce : An in-depth study. *Proc. VLDB Endow.*, 3(1-2) :472–483, September 2010.

[85] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki. Competitive randomized algorithms for nonuniform problems. *Algorithmica*, 11(6) :542–571, 1994.

[86] Rini T. Kaushik and Milind Bhandarkar. Greenhdfs : Towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, pages 1–9, Berkeley, CA, USA, 2010. USENIX Association.

[87] Rini T. Kaushik and Milind Bhandarkar. Greenhdfs : Towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, pages 1–9, Berkeley, CA, USA, 2010. USENIX Association.

[88] Kamal Kc and Kemafor Anyanwu. Scheduling hadoop jobs to meet deadlines. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 388–392, Washington, DC, USA, 2010. IEEE Computer Society.

[89] P. F. Kelly, K. A. Maulloo, and H. D. K. Tan. Rate control for communication networks : shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49(3) :237–252, 1998.

[90] Walter Kern and Daniël Paulusma. Matching games : The least core and the nucleolus. *Mathematics of Operations Research*, 28(2) :294–308, 2003.

[91] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598) :671–680, 1983.

[92] Elon Kohlberg. On the nucleolus of a characteristic function game. *SIAM Journal on Applied Mathematics*, 20(1) :62–66, 1971.

[93] Elon Kohlberg. On the nucleolus of a characteristic function game. *SIAM Journal on Applied Mathematics*, 20(1) :62–66, 1971.

[94] Oumar Koné. *Nouvelles approches pour la résolution du problème d'ordonnancement de projet à moyens limités*. Theses, Université Paul Sabatier - Toulouse III, December 2009.

[95] M. Kontagora and H. Gonzalez-Velez. Benchmarking a mapreduce environment on a full virtualisation platform. In *CISIS 2010*, pages 433–438, Feb 2010.

[96] Doug Laney. 3d data management : Controlling data volume, velocity, and variety., 2001.

[97] Willis Lang and Jignesh M. Patel. Energy management for mapreduce clusters. *Proc. VLDB Endow.*, 3(1-2) :129–139, September 2010.

[98] J. K. Lenstra. Sequencing by enumerative methods. *Mathematical Centre Tracts*, (69), 1977.

[99] J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Math. Program.*, 46(3) :259–271, February 1990.

[100] Jacob Leverich and Christos Kozyrakis. On the energy (in)efficiency of hadoop clusters. *SIGOPS Oper. Syst. Rev.*, 44(1) :61–65, March 2010.

[101] Jacob Leverich and Christos Kozyrakis. On the energy (in)efficiency of hadoop clusters. *SIGOPS Oper. Syst. Rev.*, 44(1) :61–65, March 2010.

[102] Minghong Lin, Li Zhang, Adam Wierman, and Jian Tan. Joint optimization of overlapping phases in mapreduce. *SIGMETRICS Perform. Eval. Rev.*, 41(3) :16–18, January 2014.

[103] W. Miller Louis, W. Conway Richard, and L. Maxwell William. *Theory of Scheduling*, chapter Measures for schedule evaluation, page 304. Courier Corporation, 2012, 1967.

[104] Qingming Ma. *Quality-of-Service Routing in Integrated Services Networks*. PhD thesis, School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213, 1998.

[105] L. Mashayekhy, M. M. Nejad, D. Grosu, D. Lu, and W. Shi. Energy-aware scheduling of mapreduce jobs for big data applications. In *2014 IEEE International Congress on Big Data*, pages 32–39, June 2014.

[106] Matt Massie, Bernard Li, Brad Nicholes, Vladimir Vuksan, Robert Alexander, Jeff Buchbinder, Frederiko Costa, Alex Dean, Dave Josephsen, Peter Phaal, and Daniel Pocock. *Monitoring with Ganglia*. O'Reilly Media, Inc., 1st edition, 2012.

[107] L. Massoulié and J.W. Roberts. Bandwidth sharing and admission control for elastic traffic. *Telecommunication Systems*, 15(1) :185–201, 2000.

[108] Peter Mell and Tim Grance. The nist definition of cloud computing, 2011.

[109] Peter Mell and Tim Grance. Powered by apache hadoop, 2016.

[110] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6) :1087–1092, 1953.

[111] Aristide Mingozzi, Vittorio Maniezzo, Salvatore Ricciardelli, and Lucio Bianco. An exact algorithm for the resource-constrained project scheduling problem based on a new mathematical formulation. *Manage. Sci.*, 44(5) :714–729, May 1998.

[112] Rolf H. Möhring, Andreas S. Schulz, Frederik Stork, and Marc Uetz. Solving project scheduling problems by minimum cut computations. *Manage. Sci.*, 49(3) :330–350, March 2003.

[113] Benjamin Moseley, Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. On scheduling in map-reduce and flow-shops. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 289–298, New York, NY, USA, 2011. ACM.

[114] R. Nanduri, N. Maheshwari, A. Reddyraja, and V. Varma. Job aware scheduling algorithm for mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 724–729, Nov 2011.

[115] Ramon Alvarez-Valdes Olaguibel and JoseManuel Tamarit Goerlich. The project scheduling polyhedron : Dimension, facets and lifting theorems. *European Journal of Operational Research*, 67(2) :204–220, 1993.

[116] Aysan Rasooli Oskooei and Douglas G. Down. COSHH : A classification and optimization based scheduler for heterogeneous hadoop systems. *Future Generation Comp. Syst.*, 36 :1–15, 2014.

[117] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan Jain. Purlieus : Locality-aware resource allocation for mapreduce in a cloud. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 58 :1–58 :11, New York, NY, USA, 2011. ACM.

[118] Christos H. Papadimitriou. Computational complexity. In *Encyclopedia of Computer Science*, pages 260–265. John Wiley and Sons Ltd., Chichester, UK.

[119] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization : Algorithms and Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[120] René Peinl and Florian Holzschuher. The docker ecosystem needs consolidation. In *CLOSER 2015*, pages 535–542, May 2015.

[121] Padmanabhan S. Pillai and Kang G. Shin. *Dynamic DVFS Scheduling*, pages 243–258. Springer Netherlands, Dordrecht, 2007.

[122] Michael L. Pinedo. *Scheduling : Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008.

[123] Ivanilton Polato, Denilson Barbosa, Abram Hindle, and Fabio Kon. Hybrid HDFS : decreasing energy consumption and speeding up hadoop using ssds. *PeerJ PrePrints*, 3, 2015.

[124] Jordà Polo, Claris Castillo, David Carrera, Yolanda Becerra, Ian Whalley, Malgorzata Steinder, Jordi Torres, and Eduard Ayguadé. *Resource-Aware Adaptive Scheduling for MapReduce Clusters*, pages 187–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[125] Jordà Polo, Nadal D, David Carrera, Yolanda Becerra, Beltran Y, Jordi Torres, and Eduard Ayguadé. Adaptive task scheduling for multi-job mapreduce environments. In *In Proceedings of Jornadas de Paralelismo Conference*, pages 96–101, 2009.

[126] Jakob Puchinger and Gunther R. Raidl. *Combining Metaheuristics and Exact Algorithms in Combinatorial Optimization : A Survey and Classification*, pages 41–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[127] Markus Puchta, Jens Gottlieb, Stefano Cagnoni, Jens Gottlieb, Emma Hart, Martin Middendorf, and Gunther R. Raidl. *Solving Car Sequencing Problems by Local Optimization*, pages 132–142. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[128] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan. Security of OS-level virtualization technologies : Technical report. *ArXiv e-prints*, July 2014.

[129] Yves Robert and Frederic Vivien. *Introduction to Scheduling*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[130] Franz Rothlauf. *Design of Modern Heuristics : Principles and Application*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[131] Creative Commons CC BY NC SA. Linuxcontainers.org infrastructure for container projects. http://linuxcontainers.org, 2016. Last accessed 16 March 2016.

[132] Thomas Sandholm and Kevin Lai. Dynamic proportional share scheduling in hadoop. In *Proceedings of the 15th International Conference on Job Scheduling Strategies for Parallel Processing*, JSSPP'10, pages 110–131, Berlin, Heidelberg, 2010. Springer-Verlag.

[133] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[134] Robert Sedgewick and Kevin Wayne. Minimum spanning trees, 2014. [Online ; accessed 26-August-2016].

[135] Amazon Web Services. Amazon ec2. https://aws.amazon.com/ec2/?nc1=h_ls, 2017. Last accessed 16 March 2017.

[136] K.C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19(2) :107 – 140, 1994.

[137] J. Shafer, S. Rixner, and A. L. Cox. The hadoop distributed filesystem : Balancing portability and performance. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 122–133, March 2010.

[138] Bikash Sharma, Timothy Wood, and Chita R. Das. Hybridmr : A hierarchical mapreduce scheduler for hybrid data centers. In *IEEE 33rd International Conference on Distributed Computing Systems, ICDCS 2013, 8-11 July, 2013, Philadelphia, Pennsylvania, USA*, pages 102–111, 2013.

[139] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2) :202–208, February 1985.

[140] X. Sun, C. He, and Y. Lu. Esamr : An enhanced self-adaptive mapreduce scheduling algorithm. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 148–155, Dec 2012.

[141] J. Tan, X. Meng, and L. Zhang. Coupling task progress for mapreduce resource-aware scheduling. In *INFOCOM, 2013 Proceedings IEEE*, pages 1618–1626, April 2013.

[142] Zhuo Tang, Min Liu, Almoalmi Ammar, Kenli Li, and Keqin Li. An optimized mapreduce workflow scheduling algorithm for heterogeneous computing. *The Journal of Supercomputing*, 72(6) :2059–2079, 2016.

[143] Vincent T'Kindt and Jean-Charles Billaut. *Multicriteria scheduling : theory, models and algorithms.* Springer, 2006.

[144] Linjiun Tsai and Wanjiun Liao. *Virtualized cloud data center networks issues in resource management.* Springerbriefs in electrical and computer engineering. Springer, Cham, 2016.

[145] P. China Venkanna Varma, Venkata Kalyan Chakravarthy K., V. Valli Kumari, and S. Viswanadha Raju. Analysis of a network io bottleneck in big data environments based on docker containers. *Big Data Research*, January 2016.

[146] P. China Venkanna Varma, Venkata Kalyan Chakravarthy K., V. Valli Kumari, and S. Viswanadha Raju. Analysis of a network io bottleneck in big data environments based on docker containers. *Big Data Research*, January 2016.

[147] Vijay V. Vazirani. *Approximation Algorithms.* Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[148] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria : Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 235–244, New York, NY, USA, 2011. ACM.

[149] G. Wang, A. R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *2009 IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, pages 1–11, Sept 2009.

[150] Zhe Wang, Zhengdong Zhu, Pengfei Zheng, Qiang Liu, and Xiaoshe Dong. A new schedule strategy for heterogenous workload-aware in hadoop. In *Proceedings of the 2013 8th ChinaGrid Annual Conference*, CHINAGRID '13, pages 80–85, Washington, DC, USA, 2013. IEEE Computer Society.

[151] Yan Wen, Jinjing Zhao, Gang Zhao, Hua Chen, and Dongxia Wang. A survey of virtualization technologies focusing on untrusted code execution. In *IMIS'12*, pages 378–383, July 2012.

[152] Tom White. *Hadoop : The Definitive Guide*. O'Reilly Media, Inc., 2012.

[153] Wikipedia. Activity selection problem — wikipedia, the free encyclopedia, 2016. [Online ; accessed 26-August-2016].

[154] T. Wirtz and R. Ge. Improving mapreduce energy efficiency for computation intensive workloads. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8, July 2011.

[155] Joel Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, and Andrey balmin. Flex : A slot allocation scheduling optimizer for mapreduce workloads. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, pages 1–20, Berlin, Heidelberg, 2010. Springer-Verlag.

[156] Laurence A. Wolsey. *Integer programming*. Wiley-Interscience series in discrete mathematics and optimization. J. Wiley & sons, New York (N.Y.), Chichester, Weinheim, 1998. A Wiley-Interscience publication.

[157] M.G. Xavier, M.V. Neves, F.D. Rossi, T.C. Ferreto, T. Lange, and C.A.F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. pages 233–240, Feb 2013.

[158] M.G. Xavier, M.V. Neves, F.D. Rossi, T.C. Ferreto, T. Lange, and C.A.F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. pages 233–240, Feb 2013.

[159] Fei Xu, Fangming Liu, Dekang Zhu, and Hai Jin. *Boosting MapReduce with Network-Aware Task Assignment*. Springer International Publishing, 2014.

[160] G. Xu, F. Xu, and H. Ma. Deploying and researching hadoop in virtual machines. In *Automation and Logistics (ICAL), 2012 IEEE International Conference on*, pages 395–399, Aug 2012.

[161] Huanle Xu and Wing Cheong Lau. Optimization for speculative execution of multiple jobs in a mapreduce-like cluster. *CoRR*, abs/1406.0609, 2014.

[162] F. Yan, L. Cherkasova, Z. Zhang, and E. Smirni. Dyscale : a mapreduce job scheduler for heterogeneous multicore processors. *IEEE Transactions on Cloud Computing*, PP(99) :1–1, 2015.

[163] Feng Yan, Ludmila Cherkasova, Zhuoyao Zhang, and Evgenia Smirni. Optimizing power and performance trade-offs of mapreduce job processing with heterogeneous multi-core processors. In *Proceedings of the 2014 IEEE International Conference on Cloud Computing*, CLOUD '14, pages 240–247, Washington, DC, USA, 2014. IEEE Computer Society.

[164] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 0 :374, 1995.

[165] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 374–382, Oct 1995.

[166] Y. Yao, J. Tai, B. Sheng, and N. Mi. Scheduling heterogeneous mapreduce jobs for efficiency improvement in enterprise clusters. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 872–875, May 2013.

[167] Nezih Yigitbasi, Kushal Datta, Nilesh Jain, and Theodore Willke. Energy efficient scheduling of mapreduce workloads on heterogeneous clusters. In *Green Computing Middleware on Proceedings of the 2Nd International Workshop*, GCM '11, pages 1 :1–1 :6, New York, NY, USA, 2011. ACM.

[168] Mark Yong, Nitin Garegrat, and Shiwali Mohan. Towards a resource aware scheduler in hadoop, 2009.

[169] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009.

[170] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling : A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[171] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[172] W. Zhou, J. Han, Z. Zhang, and J. Dai. Dynamic random access for hadoop distributed file system. pages 17–22, June 2012.

Aymen JLASSI

Optimisation de la gestion des ressources sur une plate-forme informatique du type "Big Data" basée sur le logiciel Hadoop

Résumé

L'entreprise "Cyres-group" cherche à améliorer le temps de réponse de ses grappes Hadoop et la manière dont les ressources sont exploitées dans son centre de données. Les idées sous-jacentes à la réduction du temps de réponse sont de faire en sorte que *(i)* les travaux soumis se terminent au plus tôt et que *(ii)* le temps d'attente de chaque utilisateur du système soit réduit. Nous identifions deux axes d'amélioration :

1. nous décidons d'intervenir pour optimiser l'ordonnancement des travaux sur une plateforme Hadoop. Nous considérons le problème d'ordonnancement d'un ensemble de travaux du type MapReduce sur une plate-forme homogène.
2. Nous décidons d'évaluer et proposer des outils capables *(i)* de fournir plus de flexibilité lors de la gestion des ressources dans le centre de données et *(ii)* d'assurer l'intégration d'Hadoop dans des infrastructures Cloud avec le minimum de perte de performance.

Dans une première étude, nous effectuons une revue de la littérature. À la fin de cette étape, nous remarquons que les modèles mathématiques proposés dans la littérature pour le problème d'ordonnancement ne modélisent pas toutes les caractéristiques d'une plateforme Hadoop. Nous proposons à ce niveau un modèle plus réaliste qui prend en compte les aspects les plus importants tels que la gestion des ressources, la précédence entre les travaux, la gestion du transfert des données et la gestion du réseau. Nous considérons une première modélisation simpliste et nous considérons la minimisation de la date de fin du dernier travail (C_{max}) comme critère à optimiser. Nous calculons une borne inférieure à l'aide de la résolution du modèle mathématique avec le solveur CPLEX. Nous proposons une heuristique (LocFirst) et nous l'évaluons. Ensuite, nous faisons évoluer notre modèle et nous considérons, comme fonction objective, la somme des deux critères identifiés depuis la première étape : la minimisation de la somme pondérée des dates de fin des travaux ($\sum w_j Cj$) et la minimisation du (C_{max}). Nous cherchons à minimiser la moyenne pondérée des deux critères, nous calculons une borne inférieure et nous proposons deux heuristiques de résolution.

Abstract

"Cyres-Group" is working to improve the response time of his clusters Hadoop and optimize how the resources are exploited in its data center. That is, the goals are to finish work as soon as possible and reduce the latency of each user of the system.
Firstly, we decide to work on the scheduling problem in the Hadoop system. We consider the problem as the problem of scheduling a set of jobs on a homogeneous platform. Secondly, we decide to propose tools, which are able to provide more flexibility during the resources management in the data center and ensure the integration of Hadoop in Cloud infrastructures without unacceptable loss of performance.
Next, the second level focuses on the review of literature. We conclude that, existing works use simple mathematical models that do not reflect the real problem. They ignore the main characteristics of Hadoop software. Hence, we propose a new model; we take into account the most important aspects like resources management and the relations of precedence among tasks and the data management and transfer.
Thus, we model the problem. We begin with a simplistic model and we consider the minimisation of the C_{max} as the objective function. We solve the model with mathematical solver CPLEX and we compute a lower bound. We propose the heuristic "LocFirst" that aims to minimize the C_{max}. In the third level, we consider a more realistic modelling of the scheduling problem. We aim to minimize the weighted sum of the following objectives : the weighted flow time ($\sum w_j Cj$) and the makespan (C_{max}). We compute a lower bound and we propose two heuristics to resolve the problem.