Table des matières

Introduction **2**

1 Aperçu sur le calcul haute performance **3**

1.1 Calcul haute performance et parallélisme 3

1.2 Taxonomie des architectures parallèles et distribuées 4

1.2.1 Les multiprocesseurs 5

1.2.1.1 Les multiprocesseurs UMA 5

1.2.1.2 Les multiprocesseurs NUMA 6

1.2.2 Les multi-ordinateurs 6

1.2.2.1 Les multi-ordinateurs homogènes 7

1.2.2.2 Les multi-ordinateurs hétérogènes 7

1.3 Les modèles de programmation parallèle 7

1.3.1 Le modèle à échange de messages 8

1.3.2 Le modèle à mémoire partagée 9

1.3.3 Le modèle du parallélisme de données 10

1.4 Conclusion 10

2 Équilibrage de charge dans les systèmes parallèles et distribuées **12**

2.1 L'équilibrage de charge dans les systèmes distribués et parallèles : problématique 12

2.2 Classification des approches d'équilibrage de charge 13

2.2.1 Approche statique et approche dynamique 13

2.2.1.1 Approche statique 13

2.2.1.2 Approche dynamique 14

2.2.2 Approche centralisée et approche distribuée 14

2.2.2.1 Approche centralisée 14

2.2.2.2 Approche distribuée 14
2.2.3 Approche source-initiative, approche receveur-initiative et approche symétrique 15
2.2.3.1 Approche source-initiative 15
2.2.3.2 Approche receveur-initiative 15
2.2.3.3 Approche symétrique 15
2.2.4 Approche adaptative et approche non adaptative 16
2.2.5 Approche coopérative et approche non coopérative 16
2.3 Les politiques d'équilibrage de charge 16
2.3.1 La politique d'information 16
2.3.2 La politique de sélection 17
2.3.3 La politique de transfert 17
2.3.4 La politique de localisation 18
2.4 Algorithmes d'équilibrage de charge 18
2.4.1 L'algorithme Shortest Expected Delay (SED) 19
2.4.2 L'algorithme Never Queue (NQ) 19
2.4.3 L'algorithme Gradient Model (GM) 20
2.4.4 L'algorithme Average Neighbourhood (AN) 21
2.4.5 L'algorithme Asynchronous Round Robin (ARR) 22
2.4.6 La méthode d'équilibrage hiérarchique 23
2.4.7 Les méthodes génétiques d'équilibrage de charge 24
2.4.7.1 GCTA (Genetic Central Task Assigner) 25
2.4.7.2 CBLB (Classifier-Based Load Balancer) 25
2.4.8 Les modèles de comportement cognitifs 26
2.5 Conclusion 27

3 Équilibrage de charge dans les algorithmes d'extraction de règles associatives 29
3.1 Le datamining : motivations et techniques 29
3.1.1 L'extraction de règles associatives 30
3.1.2 La recherche de motifs séquentiels 31
3.1.3 La classification et la régression 31
3.1.4 Le groupement (clustering) 31
3.1.5 La recherche de similitudes 32

3.1.6 La détection d'écarts 32
3.2 Parallélisation de l'extraction de règles associatives 32
3.2.1 L'algorithme IDD (Intelligent Data Distribution) 34
3.2.2 L'algorithme HPA (Hash Partioned Apriori) 35
3.2.3 L'algorithme CCPD (Common Candidate Partitioned Database) . 36
3.2.4 L'algorithme pFP-Growth (parallel FP-Growth) 36
3.3 Les mécanismes d'équilibrage de charge dans les algorithmes HPA, IDD, CCPD et pFP-Growth 37
3.3.1 Équilibrage de charge statique 37
3.3.1.1 Le partitionnement dans CCPD 38
3.3.1.2 Le partitionnement dans IDD 38
3.3.2 Équilibrage de charge dynamique 39
3.3.2.1 L'équilibrage de charge dans HPA 39
3.3.2.2 L'équilibrage de charge dans pFP-Growth 40
3.4 Étude comparative 41
3.4.1 Analyse qualitative 41
3.4.2 Comparaison 43
3.5 Conclusion 48

4 Extraction des itemsets fermés fréquents sur un cluster de PCs 50
4.1 Extraction parallèle des itemsets fermés fréquents 50
4.1.1 Notions de base 50
4.1.2 L'algorithme EXPERT (EXtraction Parallèle d'itEmsets feRmés fréquenTs) 52
4.1.3 Stratégie d'équilibrage de charge 54
4.2 Evaluation expérimentale 54
4.2.1 Environnement expérimental 54
4.2.2 Expérimentations 55
4.2.2.1 Expérimentation 1 56
4.2.2.2 Expérimentation 2 57
4.2.2.3 Expérimentation 3 59
4.2.2.4 Expérimentation 4 59
4.3 Conclusion 61

Table des figures

1.1 Machine UMA à bus 6
1.2 Machine NUMA 6
1.3 Exemple d'architecture multi-ordinateurs 8

2.1 Modèle serveur multiple files d'attentes multiples [11]. 19
2.2 Exemple du modèle gradient avec deux nœuds surchargés et un sous-chargé [11]. 21
2.3 Domaine D de l algorithme AN dans une topologie mesh [13]. 22
2.4 Organisation hierarchique d un systeme de huit nœuds interconnectés par un reseau hypercube [11] 24

4.1 Exemple d'un contexte d'extraction avec l'itemset-Trie associé [39] . . . 51
4.2 Temps d'extraction des itemsets fermés fréquents pour la base T10I4D100k et la base T40I10D100k en fonction du support et du nombre de processeurs. 58
4.3 Temps d'extraction des itemsets fermés fréquents pour la base Mushroom et la base Chess en fonction du support et du nombre de processeurs. . . . 58

Liste des tableaux

2.1 Règles de comportement basées sur un schéma d'action 27
2.2 Règles de charge sur le comportement de l'utilisateur 28

3.1 Base de transactions 33
3.2 Tableau comparif des approches d'équilibrage de charge dans CCPD, HPA, pFP-Growth et IDD 48

4.1 Caracteristiques des bases de test 55
4.2 Temps d'extraction (sec) des itemsets fermés fréquents pour la base T10I4D100k en fonction du nombre de processeurs et de la valeur du support. 56
4.3 Temps d'extraction (sec) des itemsets fermés fréquents pour la base T40I10D100k en fonction du nombre de processeurs et de la valeur du support. 57
4.4 Temps d'extraction (sec) des itemsets fermés fréquents pour la base Chess en fonction du nombre de processeurs et de la valeur du support. 59
4.5 Temps d'extraction (sec) des itemsets fermés fréquents pour la base Mushroom en fonction du nombre de processeurs et de la valeur du support. . . 60

Liste des abréviations

ATM	Asynchronous Transfer Mode
CPU	Central Processing Unit
FDDI	Fiber Distributed Data Interface
Gflops	Giga (10^9) floating point operations per second
HPF	High Performance Fortran
MIMD	Multiple Instruction, Multiple Data
MPP	Massively Parallel Processors
MPI	Message Passing Interface
NUMA	Non Uniform Memory Access
PC	Personal Computer
Po	Pico(2^{50})octets
PVM	Parallel Virtual Machine
SIMD	Single Instruction, Multiple Data
SMP	Symmetrical Multiprocessor
Tflops	Tera (10^{12}) floating point operations per second
To	Tera (2^{40}) octets
UMA	Uniform Memory Access

Introduction

Avec les récents progrès technologiques réalisés dans la collecte et le stockage des données, il est désormais facile de produire en peu de temps une masse importante de données. Ces volumes de données de plus en plus distribués, disséminent des connaissances potentiellement utiles pour des décideurs, mais leur vitesse de production et leur taille dépassent largement notre capacité à les analyser. Il est donc nécessaire de développer de nouveaux outils permettant de réduire et d'analyser ces données. Exploiter de telles masses de données nécessite le recours à des systèmes informatiques, permettant d'extraire de manière plus ou moins automatique ces informations. Ces systèmes se basent sur un processus de datamining qui englobe plusieurs techniques dérivées de différents domaines, tels que l'apprentissage automatique, l'intelligence artificielle, les bases de données et l'analyse de données. Comme type de connaissances générées par un tel processus, le plus connu est celui des règles associatives, qui permettent d'identifier des relations entre données d'une base de transactions.

D'un point de vue algorithmique, il existe toute une variété d'algorithmes en majorité sequentiels permettant d'extraire ce type de connaissance. Malgré leur efficacité, ces algorithmes voient leurs performances se dégrader lorsque la taille ou la dimension des données augmente. Pour maintenir les performances de ces algorithmes, le développement et l'utilisation de méthodes et d'outils parallèles et distribués apparaît comme une solution naturelle pouvant aider à accélérer la vitesse de traitement. En effet, les environnements de traitement parallèles et distribués augmentent grandement notre capacité à traiter les grandes masses de données dans la mesure où ils permettent d'exploiter toutes les ressources disponibles (processeurs, mémoire, etc.).

La parallélisation d'algorithmes de génération de règles associatives pose un certain

nombre de problèmes, et en particulier celui de l'équilibrage de charge. En effet, lors de la mise en œuvre d'un algorithme parallèle de datamining, il arrive fréquemment que les ressources utilisées par cet algorithme (notamment les processeurs) soient mal exploitées. Ainsi, il est possible que certains processeurs soient surchargés alors que d'autres sont sous-chargés ou complètement inactifs. Pour éviter de telles situations, un certain nombre de techniques d'équilibrage de charge sont proposées pour permettre à un ensemble de processeurs d'avoir une charge de travail qui soit plus ou moins équitable pendant la phase d'exécution d'un algorithme de datamining.

C'est dans ce cadre que se situent nos travaux, qui ont pour objectif de proposer et de mettre en œuvre un algorithme parallèle pour la génération de règles associatives à partir d'itemsets fermés fréquents et une technique d'équilibrage de charge pour celui-ci en utilisant un réseau de PCs fonctionnant sous Linux.

Le reste de ce mémoire est organisé comme suit :
Dans le premier chapitre, nous donnons un aperçu sur le calcul de haute performance, en présentant une taxonomie des architectures parallèles et distribuées. Le deuxième chapitre passe en revue l'ensemble des techniques d'équilibrage de charge utilisées dans les systèmes parallèles et distribués. Dans le chapitre 3, nous présentons un état de l'art sur le problème de l'équilibrage de charge dans le contexte de l'extraction de règles associatives. Le chapitre 4, qui constitue notre contribution, est consacré à la présentation d'une approche d'équilibrage de charge pour la génération de règles associatives à partir des itemsets fermés fréquents. Nous terminons ce mémoire par une conclusion qui résume l'ensemble des résultats obtenus et par une proposition de quelques perspectives de recherche.

Chapitre 1

Aperçu sur le calcul haute performance

Dans ce chapitre subdivisé en trois parties, nous donnons un bref aperçu sur le calcul haute performance. Nous faisons apparaître dans la première partie les motivations pour le calcul haute performance et présentons dans la seconde partie une classification des architectures parallèles et distribuées. Dans la dernière partie, nous nous intéressons aux modèles de programmation parallèle.

1.1 Calcul haute performance et parallélisme

Le parallélisme est une stratégie pour accélérer les temps d'exécution de tâches de grande taille. Les tâches sont initialement décomposées en sous-tâches de taille plus petite et celles-çi sont ensuite assignées à plusieurs processeurs qui les exécutent simultanément. Traditionnellement, les programmes informatiques étaient écrits pour des machines monoprocesseurs (machines de von Neumann) qui n'exécutent qu'une seule instruction à la fois et ceci tous les 9ns pour les plus rapides d'entre elles [1]. Cette vitesse d'exécution peut paraître élevée mais pour les applications qui nécessitent beaucoup de puissance de calcul, tels que problèmes de simulation et de modélisation de phénomènes naturels complexes, les problèmes nécessitant de grandes capacités de calcul et la manipulation de grandes quantités de données comme le datamining, le traitement d'image, etc. les performances exigées sont nettement plus élevées. Devant les limites physiques de la taille de la mémoire et de la vitesse de calcul des systèmes monoprocesseurs, le parallélisme apparaît comme une solution compétitive (excellent rapport coût/performance) pouvant répondre aux importants besoins en puissance de calcul (de l'ordre du Gflops et même du Tflops) et

en espace mémoire (de l'ordre du To et même du Po) des classes de problèmes citées plus haut. Les machines parallèles permettent au programmeur de disposer d'un nombre important de processeurs. Ce nombre peut être modulable suivant la puissance de traitement nécessaire. D'autres avantages caractérisent ces systèmes tels que leur grande disponibilité, leur meilleure tolérance aux pannes, leur grande flexibilité et surtout, ils permettent de partager de nombreuses ressources (processeur, mémoire, disque etc.) à travers un réseau. L'utilisation des machines parallèles introduit cependant un certain nombre de problèmes comme celui de l'équilibrage de charge qui sera étudié dans le chapitre suivant, celui de la distribution des données, de la maîtrise du coût des communications entre processeurs, etc. Ces problèmes ont un grand impact sur les performances des systèmes parallèles qu'ils risquent de dégrader s'ils ne sont pas correctement pris en charge.

1.2 Taxonomie des architectures parallèles et distribuées

La multiplication des architectures comportant plusieurs processeurs entraîne une certaine confusion dans les terminologies utilisées dans la littérature particulièrement lorsqu'il s'agit de caractériser les systèmes distribués et parallèles. Assez souvent, la frontière entre un système distribué et système parallèle n'est pas très nette. Aussi nous donnons cette caractérisation :

- Un *système parallèle* est un système composé de plusieurs processeurs fortement couplés, localisés sur la même machine, connectés par des liens de communication point à point à base de switch (Hypercube, Crossbar, Omega, grid ...) et qui travaillent à la résolution d'une même problème. Les multiprocesseurs (UMA, NUMA) et les MPPs appartiennent à cette catégorie.
- Un *système distribué* est un système composé de plusieurs processeurs physiquement distribués et faiblement couplés, reliés entre eux par un réseau conventionnel de communication (ATM, FDDI, Token Ring, Ethernet etc.) et qui collaborent à la résolution d'un ou plusieurs problèmes. Internet, les intranets, les systémes embarqués, les systèmes mobiles et les systèmes de téléphonie appartiennent à cette catégorie.

Il existe de nombreuses méthodes de classification des architectures parallèles et distribuées dont celle de Flynn qui est sans doute la plus populaire d'entre elles. Dans cette classification [2] basée sur le flux d'instructions et le flux de données deux principales

classes sont établies. Il s'agit des machines SIMD dont les processeurs exécutent la même opération de façon synchrone sur des données différentes et des machines MIMD dont chaque processeur est autonome et exécute son propre programme sur ses propres données. Aujourd'hui les machines SIMD sont abandonnées au profit des machines MIMD qui offrent la souplesse nécessaire à la résolution de certains problèmes. Une classification plus fine des machines MIMD essentiellement basée sur l'organisation de la mémoire et les types d'interconnexion permet de distinguer deux grandes familles : les multiprocesseurs et les multi-ordinateurs [4].

1.2.1 Les multiprocesseurs

Un multiprocesseur est un système informatique dans lequel deux ou plusieurs processeurs partagent l'accès total à une mémoire commune [3]. Les processeurs, la mémoire et les périphériques d'entrée et sortie sont reliés par un réseau d'interconnexion pouvant prendre la forme d'un bus commun, d'un switch crossbar ou d'un réseau multi-étages. Il n'y a qu'une seule copie du système d'exploitation qui tourne sur ce type d'architecture. Les multiprocesseurs sont de deux types : Les machines UMA et les machines NUMA.

1.2.1.1 Les multiprocesseurs UMA

Dans cet archiecture, il existe une mémoire unique, partagée par tous les processeurs. La mémoire est équidistante de tous les processeurs qui ont ainsi le même temps d'accès sur une référence mémoire donnée. Tous les processeurs partagent les ressources globales disponibles (bus, mémoire, E/S) mais peuvent avoir des caches privées (voir Figure 1.1). Toute opération effectuée sur la mémoire est immédiatement visible par l'ensemble des autres processeurs.

Les machines UMA ont l'avantage d'être faciles à programmer. En effet elles supportent le modèle de programmation à mémoire partagée qui est proche du modèle séquentiel, ce qui simplifie le portage du code séquentiel vers le parallèle. De plus, il est plus facile de concevoir un système d'exploitation qui utilise les processeurs de manière symétrique. Cependant ces systèmes sont peu scalables, car au-delà de quelques dizaines de processeurs il devient très difficile de maintenir la cohérence des caches et un temps d'accès uniforme à la mémoire.

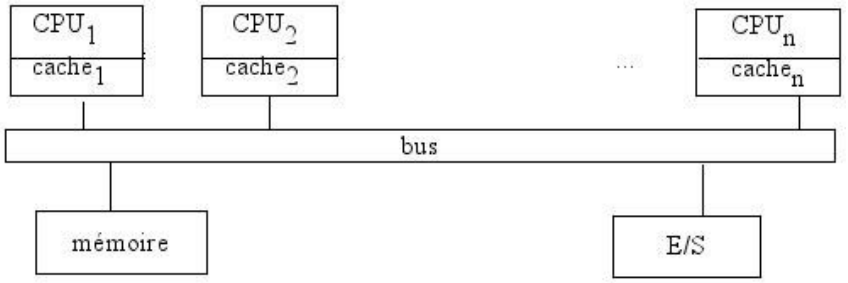


FIG. 1.1 – Machine UMA à bus

1.2.1.2 Les multiprocesseurs NUMA

Les machines NUMA sont des systèmes à mémoire partagée. Cependant ils ont la particularité d'avoir une mémoire physiquement distribuée, ce qui fait varier le temps d'accès en fonction de l'emplacement mémoire. Chaque processeur dispose d'une mémoire locale (voir Figure 1.2). L'ensemble de toutes ces mémoires forme un espace d'adressage global et unique, accessible à tous les processeurs. Il est plus rapide d'accéder à une mémoire locale qu'à celles distantes à cause du délai de latence associé au réseau d'interconnexion.

Les machines NUMA ont l'avantage de supporter efficacement le modèle de programmation à mémoire partagée et d'être facilement scalables. En effet, les performances restent bonnes même avec des centaines de processeurs. Un inconvénient de ces systèmes est la complexité rencontrée au niveau matériel ou logiciel pour garantir la cohérence des données et un espace d'adressage global.

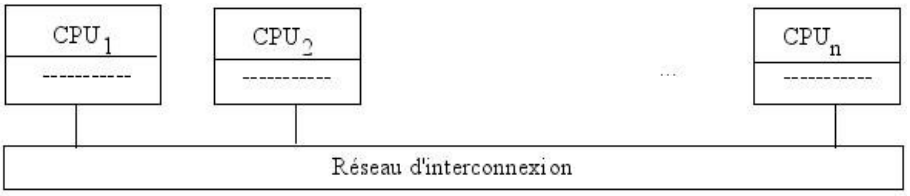


FIG. 1.2 – Machine NUMA

1.2.2 Les multi-ordinateurs

Un système multi-ordinateur est constitué par un ensemble de machines indépendantes reliées entre elles par un réseau d'interconnexion. Chaque machine possède une mémoire privée à laquelle elle accède directement (voir Figure 1.3). Les mémoire distantes ne sont pas directement adressables et les données sont partagées en envoyant explicitement ou en

recevant des messages. La nature du réseau d'interconnexion varie d'un multi-ordinateur à un autre. Certains utilisent un bus de type Ethernet, FDDI, ATM alors que d'autres utilisent un switch de type grid, cube ou hypercube. Il y a une copie séparée du système d'exploitation sur chaque nœud du système. Les multi-ordinateurs se repartissent en deux catégories : les multi-ordinateurs homogènes et les multi-ordinateurs hétérogènes.

1.2.2.1 Les multi-ordinateurs homogènes

Parfois appelés MPPs, ils sont constitués de processeurs identiques reliés par un réseau d'interconnexion unique qui utilise la même technologie partout. Ces systèmes sont particulièrement scalables puisqu'ils ne demandent pas un dispositif logiciel ou matériel pour maintenir la cohérence des données. Cependant ils sont relativement difficiles à programmer à cause de la charge supplémentaire dûe au placement et au partitionnement des données entre les différents processeurs, ainsi que les latences élevées dans certains types de réseaux.

1.2.2.2 Les multi-ordinateurs hétérogènes

C'est le cas le plus commun pour les systèmes distribués. Il s'agit de plusieurs ordinateurs aux caractéristiques matérielles et logicielles différentes, connectés à travers des réseaux utilisant des technologies différentes. Il n'y a pas une vision globale du système, les performances et les services offerts ne sont pas les mêmes partout pour les applications (à cause de l'hétérogénéité). Un tel système est relativement facile à construire et à étendre. De plus, la puissance sans cesse croissante des ordinateurs associée à leur faible coût et aux progrès technologiques réalisés dans les réseaux haut débit font que ce type de machine a un excellent rapport coût/performance même si le partage des ressources pose des problèmes de sécurité et nécessite un dispositif logiciel qui masque l'hétérogénéité de l'environnement

1.3 Les modèles de programmation parallèle

La programmation parallèle consiste à développer des programmes qui s'exécutent simultanément sur plusieurs processeurs à la fois. Un modèle de programmation parallèle est une abstraction du matériel que le programmeur utilise pour coder et exécuter des programmes parallèles. Il implique les langages, les systèmes de communication, les

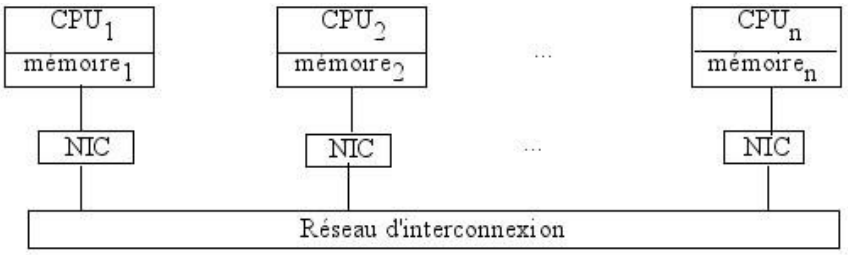


FIG. 1.3 – Exemple d'architecture multi-ordinateurs

bibliothèques et les compilateurs. Les modèles de programmation peuvent être implémentés de différentes manières : comme des librairies qu'on invoque à travers des langages de programmation séquentielle, comme des extensions de langages ou sous d'autres formes complètement nouvelles. Les modèles de programmation parallèle les plus populaires sont le modèle à échange de messages, le modèle à mémoire partagée et le modèle du parallélisme de données.

1.3.1 Le modèle à échange de messages

Dans ce modèle, un programme est constitué par un ensemble de processus opérant chacun dans un espace d'adressage privé. Les processus communiquent par échange explicite de messages. Un système d'échange de messages demande un schéma d'adressage qui repère les processus les uns par rapport aux autres et deux primitives de base (send/receive) permettant le transfert de données entre processus. Ce modèle donne un contrôle total de la mise en œuvre du parallélisme au programmeur qui doit cependant gérer explicitement la distribution des données, toutes les communications entre processus et les synchronisations. Le modèle à échange de messages est largement utilisé dans les machines à mémoire distribuée (multi-ordinateurs) mais il est aussi implémenté sur des machines à mémoire physiquement partagée (machines NUMA) avec de bonnes performances. L'ensemble des opérations de communication autorisées par une implémentation du modèle à échange de messages est disponible à travers des bibliothèques et des langages de programmation classiques.

MPI et PVM sont les bibliothèques les plus utilisées dans le modèle de programmation à échange de méssages. Elles supportent les langages C, C++ et Fortran. Elles permettent d'écrire de manière simple et efficace des programmes parallèles s'exécutant sur n'importe quelle plateforme où elles sont implémentées.

PVM donne l'image d'une machine parallèle virtuelle à une collection d'ordinateurs hétérogènes fonctionnant sous UNIX. La bibliothèque PVM est constituée d'un processus résidant dans tous les ordinateurs qui forment une machine virtuelle et d'une bibliothèque de fonctions pour l'échange de messages, la gestion des processus et la synchronisation des tâches [5].

En raison sa grande portabilité et des performances qu'elle offre, MPI est devenue la bibliothèque standard du modèle à échange de messages [45]. Elle fournit des routines pour la communication et la gestion des processus. Les processus utilisent une communication point à point pour l'échange de données entre deux processus. Un groupe de processus peut utiliser une communication collective pour effectuer des opérations de synchronisation globale, de diffusion globale ou sélective, de collecte ou de réduction globale de données [6].

1.3.2 Le modèle à mémoire partagée

Le modèle à mémoire partagée permet d'exprimer de manière simple et puissante le parallélisme dans une application. Dans ce modèle, plusieurs processus partagent un espace mémoire commun où ils peuvent lire et écrire des données de manière asynchrone. Pour éviter les conflits d'accès aux données partagées, des mécanismes de synchronisation sont nécessaires. Comme les données ne sont la "propriété" d'aucun processus, il n'y a pas de communication explicite des données selon le modèle producteur/consommateur, ce qui allége le travail du programmeur. Cependant il devient difficile au delà d'une centaine de processeurs de maintenir l'efficacité des applications surtout celles qui exigent un haut degré de parallélisme avec des milliers de processus. Le modèle à mémoire partagée est implémenté dans une large variété de multiprocesseurs (SMP et machines NUMA). Le regain d'intérêt pour les multiprocesseurs a poussé plusieurs constructeurs à définir un standard qui est OpenMP.

OpenMP est une interface de programmation d'applications parallèles sur une architecture SMP. Elle est multi-plateformes (UNIX et Windows NT) et supporte les langages C, C++ et Fortran. OpenMP est constituée de directives de compilation, de sous programmes et de variables d'environnement. Un programme OpenMP est une succession de régions parallèles et de régions séquentielles. Il est constitué par un processus unique (master thread) qui, à l'entrée d'une région parallèle, active des processus légers (thread). Chaque thread exécute une tâche et disparaît à la fin de la région parallèle pendant que

le master thread continue l'exécution du programme jusqu'à la prochaine région parallèle [7].

1.3.3 Le modèle du parallélisme de données

Cet autre modèle exploite le parallélisme dérivant de l'application de la même opération à tous les éléments d'un ensemble de données. Les processus exécutent le même code mais ils opèrent sur des données différentes. Le problème est d'abord décomposé en plusieurs tâches de petite taille. Chaque tâche est ensuite assignée à un processeur qui effectue sur les données locales les opérations devant déterminer leur nouvelle valeur. Conceptuellement ce modèle est plus simple que le modèle à échange de messages. Les détails de la distribution des données et de la communication entre les processeurs sont masqués par le compilateur. Le programmeur doit cependant fournir le maximum d'informations sur la distribution des données au compilateur afin de minimiser la communication entre les processus. Le parallélisme de données est utilisé aussi bien dans les systèmes à mémoire partagée que dans les systèmes à mémoire distribuée. Les implémentations de ce modèle sont souvent des extensions des langages de programmation séquentielle existants (Java, C++, Fortran etc.).

HPF est une extension du langage Fortran. Il est le langage le plus utilisé pour écrire des programmes en utilisant le parallélisme de données. Il a été conçu pour réaliser la portabilité des programmes à travers différentes architectures parallèles tout en maintenant élevées leurs performances. HPF comprend des directives qui aident à optimiser le code en indiquant au compilateur la manière de répartir les données entre les processeurs. Il dispose de fonctions qui sont des extensions directes du langage Fortran et qui peuvent affecter les résultats du calcul contrairement aux directives. HPF fournit aussi aux programmeurs une bibliothèque de fonctions particulièrement précieuses pou le calcul haute performance [8].

1.4 Conclusion

Dans ce chapitre premier, nous nous sommes intéressés à divers aspects du calcul haute performance. C'est ainsi qu'après avoir donné les motivations pour ce type de calcul, nous avons présenté les principaux modèles de programmation parallèle et une taxonomie des architectures parallèles et distribuées. Comme nous l'avons montré dans la

première partie de de chapitre, le parallélisme offre de multiples avantages (puissance de calcul, disponibilité, tolérance aux pannes, flexibilité etc.) mais il soulève aussi de nouveaux problèmes, qui s'ils ne sont pas correctement traités peuvent influencer négativement les performances obtenues. Dans le chapitre suivant, nous traiterons le problème de l'équilibrage de charge qui représente un aspect très important dans les systèmes parallèles et distribuées.

Chapitre 2

Équilibrage de charge dans les systèmes parallèles et distribuées

Dans ce chapitre, nous discutons de l'équilibrage de charge dans les systèmes distribués et parallèles en donnant d'abord une classification des différentes approches qui existent dans la littérature. Nous étudions ensuite les politiques utilisées dans un système d'équilibrage de charge et pour terminer nous présentons quelques algorithmes d'équilibrage de charge.

2.1 L'équilibrage de charge dans les systèmes distribués et parallèles : problématique

Les récentes évolutions technologiques en matière de puissance de calcul, de capacité de stockage et la disponibilité de réseaux à haut débit à moindre coût ont largement contribué au succès que connaissent aujourd'hui les systèmes distribués et parallèles. De tels systèmes permettent de disposer d'importantes ressources (CPUs, mémoires, disques) que doivent se partager les tâches qui leur soumises. Ce partage pose nécessairement le problème de l'utilisation efficiente de ces ressources. En effet, sans aucune politique de gestion de ces ressources, il peut arriver que sur certains nœuds du système, la totalité des ressources disponibles ne soit pas utilisée alors que sur d'autres nœuds du système, des tâches dont les exigences en ressources ne peuvent être satisfaites dans l'immédiat doivent attendre d'être satisfaites. Une telle situation allonge évidement les temps de traitement et influe négativement sur les performances du système. Dans les systèmes distribués et

parallèles, le coût des communications inter processus est aussi un facteur qu'il faut maîtriser. En effet, une répartition d'un ensemble de tâches entre les différents nœuds du système qui ne tiendrait pas compte des dépendances pouvant exister entre les tâches, augmenterait ce coût et allongerait par conséquent les temps de traitement. C'est pour éviter que de telles situations se produisent que différentes approches d'équilibrage de charge ont été proposées. Dans le contexte d'un traitement parallèle, les tâches sont initialement décomposées en tâches de taille plus petite et l'équilibrage de charge vise à assigner équitablement ces sous tâches aux différents processeurs du système, en tenant compte d'éventuelles dépendances entre les tâches de manière à réduire le coût des communications inter processeurs.

2.2 Classification des approches d'équilibrage de charge

Le problème de l'équilibrage de charge a été étudié depuis de nombreuses années dans les systèmes distribués et parallèles. Il existe une abondante littérature traitant de ce problème et dans laquelle on trouve une prolifération de terminologies incohérentes voir même contradictoires et des formulations légèrement divergentes du problème[9]. Une telle situation rend difficile la description et l'analyse qualitative des différentes approches proposées d'où la nécessité d'avoir une taxonomie qui permette d'uniformiser les différentes terminologies pour une meilleure description des approches et leur comparaison. Ainsi, plusieurs taxonomies ont été proposées dont celle de Casavant et Kuhl [9] qui est largement adoptée parce que très complète. Dans cette classification, on trouve un certain nombre d'approches d'équilibrage de charge que nous allons présenter dans les sections suivantes.

2.2.1 Approche statique et approche dynamique

Cette manière de classifier les approches d'équilibrage de charge a pour but de préciser à quel moment il faut assigner les tâches aux processeurs.

2.2.1.1 Approche statique

Les tâches sont assignées de manière déterministe ou aléatoire aux nœuds avant l'exécution du programme. Les informations concernant le temps d'exécution des tâches et les

ressources sont supposées connues au moment de la compilation. Une tâche est toujours exécutée sur le nœud qu'on lui a assigné. Cet approche est très efficace et très simple à mettre en œuvre lorsque la charge de travail est au préalable suffisamment bien caractérisée. Cependant il y a une dégradation des performances lorsque le système connaît des fluctuations.

2.2.1.2 Approche dynamique

Cet approche part de l'hypothèse qu'on a peu de connaissances sur les ressources nécessaires à une application. L'assignation des tâches aux processeurs se fait durant la phase d'exécution du programme, en fonction des informations collectées sur l'état du système, comme par exemple la terminaison ou l'arrivée d'une tâche. Il y a certes une amélioration des performances de l'application mais au prix d'une complexité dans la mise en œuvre de l'approche d'équilibrage de charge et d'un surcoût associé aux communications.

2.2.2 Approche centralisée et approche distribuée

Cette classification indique comment les informations sur l'état du système sont échangées et comment les tâches sont assignées aux processeurs.

2.2.2.1 Approche centralisée

Dans une politique centralisée, un nœud du système est choisi comme coordinateur. Il reçoit les informations de charge de tous les autres nœuds qu'il assemble pour obtenir l'état du système. Lorsque qu'il faut assigner une tâche à un processeur, le coordinateur choisit un nœud cible en se basant les informations qu'il a collectées. Cette centralisation réduit le coût des communications dans le système mais le coordinateur peut constituer un goulot d'étranglement. De plus une panne de ce dernier provoquerait l'effondrement du système.

2.2.2.2 Approche distribuée

Dans une approche distribuée, il n'existe pas de coordinateur central. Par contre, chaque nœud du système est responsable de collecter les informations de charge sur les autres nœuds et de les rassembler pour obtenir l'état du système. Les décisions de placement sont prises localement par chacun des nœuds. Cet approche offre une grande

résistance aux pannes mais elle peut engendrer un coût élevé des communications pour maintenir les informations sur l'état du système. C'est donc une solution difficilement pratiquable dans les systèmes distribués à large échelle.

2.2.3 Approche source-initiative, approche receveur-initiative et approche symétrique

Lorsque la décision de participer à l'équilibrage de charge est prise localement par les nœuds du système, l'initiative peut provenir exclusivement du nœud source ou du nœud receveur, mais elle peut aussi provenir des deux nœuds (source et receveur) de manière alternée.

2.2.3.1 Approche source-initiative

Dans cet approche, c'est le nœud-source ,c'est à dire le nœud surchargé, qui prend l'initiative de participer à l'équilibrage de charge. Il cherche alors à transférer son surplus de travail vers un nœud faiblement chargé. Dans une approche centralisée, le coordonnateur central choisit un nœud cible sur la demande du receveur tandis que dans une politique distribuée, la source envoie des requêtes à plusieurs nœuds pour trouver un receveur. Cet approche donne de meilleures performances dans les systèmes dont la charge est faible.

2.2.3.2 Approche receveur-initiative

Dans cet approche, c'est le nœud-receveur, c'est-à-dire le nœud légèrement chargé, qui prend l'initiative de participer à l'équilibrage de charge. Dans ce cas, le nœud-receveur demande à recevoir le surplus de travail des nœuds surchargés. Le choix de la cible se fait de la même manière que pour la stratégie source-initiative. Cet approche donne de meilleures performances dans les systèmes dont la charge est élevée.

2.2.3.3 Approche symétrique

C'est une combinaison des deux approches précédentes pour profiter des avantages de chacune d'elles. Lorsque la charge du système est forte, l'approche receveur-initiative est utilisée alors que dans le cas contraire on utilise l'approche source-initiative.

2.2.4 Approche adaptative et approche non adaptative

Dans une approche adaptative, les algorithmes et paramètres utilisés changent dynamiquement en fonction des performances actuelles et passées du système, en réponse aux décisions prises dans le passé par le système d'équilibrage de charge. Le système d'équilibrage de charge prévoit un ensemble de scénario qui sont choisis en réponse au comportement du système. Cet approche peut offrir de meilleures performances lorsque l'état du système change fréquemment. Par contre, une approche non adaptative exécute toujours le même algorithme avec les mêmes paramètres indépendamment du comportement du système.

2.2.5 Approche coopérative et approche non coopérative

Le critère de cette classification est le degré d'autonomie qu'a un processeur pour déterminer comment ses propres ressources doivent être utilisées. Dans l'approche non coopérative, les processeurs agissent comme des entités autonomes. Ils prennent des décisions sur l'utilisation de leurs ressources sans tenir compte des conséquences sur le reste du système. Dans l'approche coopérative, les processeurs prennent leurs décisions de manière concertée de sorte que les décisions prises se reflètent sur les performances globales du système.

2.3 Les politiques d'équilibrage de charge

Un système d'équilibrage de charge utilise plusieurs politiques dont les plus importants sont : la politique d'information, la politique de sélection, la politique de transfert et la politique de localisation. Ces politiques offrent à un algorithme d'équilibrage de charge des fonctionnalités lui permettant de mieux répartir la charge du système entre les nœuds d'un système et d'améliorer ses performances.

2.3.1 La politique d'information

La politique d'information est responsable de la collecte et de dissémination de l'information sur l'état du système. L'information de charge peut être obtenue par exploration (probing) d'un groupe de nœuds, par collecte périodique ou par diffusion. La méthode par exploration, pour un nœud désirant transférer une tâche, consiste à choisir un nœud parmi

un groupe et à vérifier si celui-ci peut participer à la redistribution de la charge de travail. La collecte périodique consiste à recueillir périodiquement l'information sur l'état du système. Il en résulte que l'image qu'un nœud peut avoir de l'état du système peut ne pas correspondre à l'état réel du système. La diffusion consiste pour chaque nœud à diffuser l'information sur son état, à chaque fois que celui-ci change sans qu'aucune requête explicite ne lui soit adressée par les autres nœuds. Toutes ces méthodes peuvent être centralisées ou distribuées.

2.3.2 La politique de sélection

Le rôle d'une politique de sélection, lorsqu'elle est déclenchée par la politique de transfert, est de choisir convenablement les tâches à transférer. Il existe plusieurs manières d'effectuer cette sélection. Une première méthode consiste à choisir les tâches de manière aléatoire. Une seconde méthode de sélection consiste à choisir la tâche ayant contribué à la surcharge du nœud, c'est-à-dire celle arrivée après que la longueur de la file d'attente du nœud ait dépassé un certain seuil. Il existe aussi une méthode de sélection basée sur le filtrage des tâches. Comme filtre, on peut prendre le temps moyen d'exécution qui permet de séparer les tâches dont le temps de traitement est court et celles dont le temps de traitement est long. La sélection se fait ensuite parmi les tâches dont le temps de traitement est long, celles dont le temps de traitement est court étant exécutées localement.

2.3.3 La politique de transfert

Le but de cette politique est de constater les cas de déséquilibre. Ce constat est souvent fait en se basant sur le nombre de tâches dans la file d'attente du processeur. Si ce nombre dépasse un seuil maximal, on considère que le nœud est surchargé et qu'il peut participer à un transfert de tâches comme source. Si ce nombre est inférieur à un seuil minimal, on considère alors que le nœud est sous chargé et qu'il peut participer à une operation d'équilibrage de charge comme receveur. Le choix des seuils (minimal et maximal) est fondamental pour les performances du système. Les meilleures performances ont été obtenues en adaptant ces seuils la charge du système. Ainsi, lorsque la charge du système est élevée on maintient ces seuils élevés pour éviter le transfert des tâches et lorsque la charge du système est basse on maintient les seuils bas pour favoriser le transfert.

2.3.4 La politique de localisation

La politique de localisation est responsable d'identifier un partenaire approprié (receveur ou source) pour un nœud une fois que la politique de transfert a décidé que le nœud est une source ou un receveur. Le choix du partenaire peut se faire de manière aléatoire ou en utilisant les informations de charge rassemblées par la politique d'information. L'interrogation est la méthode la plus utilisée dans une politique distribuée. Ici un nœud interroge un autre (ou plusieurs) pour déterminer s'il est un partenaire convenable pour la distribution de charge ou non. Par contre, dans une politique centralisée, le nœud qui cherche un partenaire pour la distribution de charge contacte le coordinateur qui collecte l'information sur le système. La politique de localisation utilise ensuite les informations collectées par le coordinateur pour sélectionner les nœuds sources ou receveurs.

2.4 Algorithmes d'équilibrage de charge

Dans les premiers travaux sur l'équilibrage de charge, la plupart des solutions proposées étaient statiques. Les algorithmes statiques nécessitent une connaissance à priori des tâches et du système sur lequel elles vont être exécutées. Par conséquent ces algorithmes ne conviennent pas pour les applications et les systèmes ayant un comportement imprévisible. Depuis plusieurs années maintenant, l'essentiel des travaux de recherche porte sur des algorithmes dynamiques qui se basent sur une connaissance de l'état actuel du système et sur les besoins dynamiques des applications pour prendre les décisions d'ordonnancement. L'intérêt qui leur est porté se justifie par la popularité des multi-ordinateurs caractérisés par une très grande hétérogénéité mais aussi par la complexité des tâches qui y sont souvent exécutées. Les algorithmes dynamiques peuvent être distribués ou centralisés, mais avec les systèmes massivement parallèles largement utilisés aujourd'hui pour le traitement parallèle, un équilibrage de charge distribué devient une exigence pour la scalabilté d'une part mais aussi pour éviter le goulot d'étranglement de l'approche centralisée d'autre part. Aussi, cette section sera consacrée à l'étude de quelques algorithmes d'équilibrage de charge dynamiques.

2.4.1 L'algorithme Shortest Expected Delay (SED)

L'algorithme SED [10] est basé sur un modèle mathématique avec N serveurs, chacun ayant une vitesse de service μ_k distribuée exponentiellement et variant d'un serveur à un autre. Chaque serveur a sa propre file d'attente locale avec une loi d'arrivée *Poisson* de paramètre λ. Le nombre de tâches dans file d'attente est noté x_k (voir Figure 2.1). L'algorithme essaie de minimiser le retard d'exécution prévu pour chaque tâche et il peut être centralisé ou distribué. Dans les deux cas (centralisé ou distribué), la tâche est envoyée vers le serveur ayant le plus petit S_k ou S_k représente une fonction de coût qui lorsque la tâche est exécutée par le k^{ieme} serveur. Cette fonction est définie comme suit :

$$S_k = \frac{1 + x_k}{\mu_k} \tag{2.1}$$

L'algorithme n'affecte pas les tâches en cours d'exécution mais uniquement celles qui sont actuellement dans la file d'attente du serveur et celles qui y seront ultérieurement. Cette dernière catégorie de tâches peut rester longtemps dans la file d'attente ou utiliser un serveur lent à cause de décisions d'ordonnancement prises antérieurement, ce qui dégrade les performances de l'algorithme. L'algorithme requiert que le nœud source obtienne l'information de charge sur les autres pour choisir un partenaire de transfert. L'information peut porter sur le système global ou sur un sous ensemble de nœuds (solution plus scalable). Les nœuds peuvent aussi choisir de diffuser leur information de charge périodiquement ou à chaque fois que leur état change.

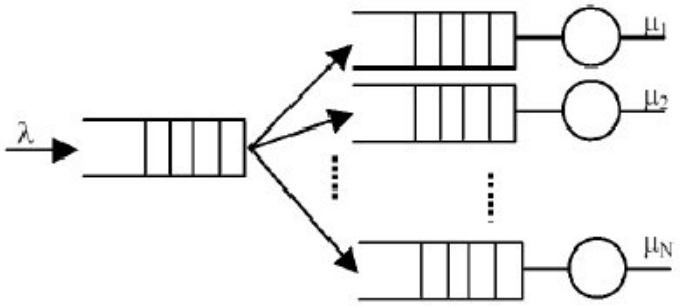


FIG. 2.1 – Modèle serveur multiple files d'attentes multiples [11].

2.4.2 L'algorithme Never Queue (NQ)

L'algorithme NQ [10] fonctionne selon le principe suivant : lorsqu'il faut placer une tâche sur un nœud, l'algorithme calcule le coût du transfert pour chacune des destinations

possibles et choisit le serveur disponible le plus rapide(ayant la plus grande vitesse d'exécution) pour lui envoyer la tâche. Si aucun serveur n'est disponible, l'algorithme SED est appliqué, c'est-à-dire qu'on choisit le serveur ayant le retard d'exécution prévu le plus petit.

L'algorithme planifie les tâches d'une manière qui, lorsqu'elles sont prises individuellement, augmente le temps d'exécution estimé. Cependant il minimise les retards supplémentaires causés par les tâches qui arrivent par la suite. La politique de transfert utilise un seuil nul par conséquent, les nouvelles tâches sont toujours envoyées vers un autre serveur à moins que la source ne soit le serveur disponible le plus rapide.

L'algorithme NQ tient compte de l'existence de serveurs rapides et de serveurs lents dans le système. Dans ce cas, lorsque la charge du système est élevée, et pour avoir assez de puissance de calcul, il vaut mieux affecter les nouvelles tâches aux serveurs lents que d'attendre un serveur rapide, aussi longtemps que les serveurs rapides sont occupés.

2.4.3 L'algorithme Gradient Model (GM)

Dans l'algorithme Gradient Model [12], les nœuds sous chargés informent les autres nœuds de leur état et ceux surchargés répondent en envoyant une partie de leur surplus de travail au nœud le plus proche ayant une faible charge. Ainsi les tâches migrent dans le système dans la direction des points sous chargés guidées par le gradient de proximité. L'algorithme utilise deux paramètres seuils *Lower-Water-Mark (LWM)* et *Hight-Water-Mark (HWM)* pour déterminer l'état des nœuds. Un nœud est sous chargé si sa charge est inférieure au seuil LWM, surchargé si sa charge est supérieure au seuil HWM et équilibré dans les autres cas. Pour chaque nœud, une proximité qui représente la distance la plus courte entre lui et le plus proche nœud sous chargé du système est définie. Tous les nœuds sont initialisés avec une proximité égale au diamètre du système. Lorsqu'un nœud passe à l'état sous chargé, sa proximité est mise à zéro. Les autres nœuds calculent leur proximité grâce à la formule 2.2 où p représente le nœud et n_i ses voisins les plus proches.

$$proximite(p) = min(proximite(n_i)) + 1 \tag{2.2}$$

Si la proximité d'un nœud change, il doit le notifier à ses voisins les plus proches. L'opportunité de l'équilibrage de charge est déterminée par les seuils (LWM) et (HWM). L'algorithme ne mesure pas le degré de déséquilibre mais il cherche uniquement s'il en existe un. Dans l'exemple de la Figure 2.2, il y'a deux nœuds surchargés et un sous chargé.

Les nœuds surchargés envoient tous les deux une charge δ dans la direction du nœud sous chargé. La valeur de δ peut être déterminée comme un pourcentage de la charge initiale ou comme un nombre fixe de tâches. Elle peut avoir un impact négatif sur les performances de l'algorithme si elle est choisie trop grande ou trop petite. La mise à jour des proximités se fait de manière itérative et génère par conséquent un nombre important de messages. La migration des tâches, des nœuds surchargés vers ceux sous chargés, entraîne un surcoût dû à la nature asynchrone de l'algorithme. En effet, puisque la destination finale n'est pas connue à l'avance, le nœud surchargé envoie une fraction de sa charge en direction du nœud sous chargé le plus proche et donc les nœuds intermédiaires sont obligés d'interrompre leur travail pour router les tâches qui transitent par eux.

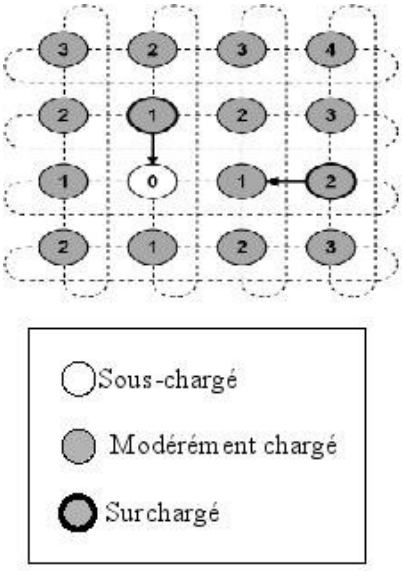


FIG. 2.2 – Exemple du modèle gradient avec deux nœuds surchargés et un sous-chargé [11].

2.4.4 L'algorithme Average Neighbourhood (AN)

Cet algorithme utilise le phénomène de diffusion qui permet d'équilibrer les distributions non homogènes uniquement sur la base des états locaux et en déplaçant localement les charges dans l'objectif de compenser les déséquilibres [13]. Pour cet algorithme il y a un élément appelé *Allocation Manager (AM)* qui est dupliqué sur chaque nœud du système et qui implémente localement la politique d'équilibrage ; il garantit aussi la cohérence et la sérialisation des actions d'équilibrage. L'algorithme AN s'applique à un domaine consti-

tué d'un nœud appelé *master* (centre du domaine) et de l'ensemble de ses voisins directs. Si K est le nombre de voisins directs d'un nœud donné n, n appartient au plus à $K+1$ domaines différents. Il est le centre dans un des domaines et un nœud périphérique dans les K autres domaines dont ses voisins sont le centre (voir Figure 2.3). L'Allocation Manager (AM) du nœud n prend les décisions relatives à l'équilibrage de charge dans le domaine dont il est le master. Il prend l'information de charge de chaque nœud du domaine et calcule dynamiquement la charge moyenne du domaine et deux seuils S_{source} et $S_{receveur}$, centrés autour de cette charge moyenne. Un nœud du domaine est source si sa charge excède le seuil S_{source}, il est receveur si sa charge est inférieure au seuil $S_{receveur}$ et il est neutre autrement. A chaque fois que le nœud master reçoit une mise à jour de l'information de charge, il calcule la nouvelle charge moyenne du domaine et met à jour les seuils S_{source} et $S_{receveur}$. L'AM du nœud master essaie ensuite de ramener la charge de chaque nœud du domaine aussi proche que possible de la charge moyenne du domaine.

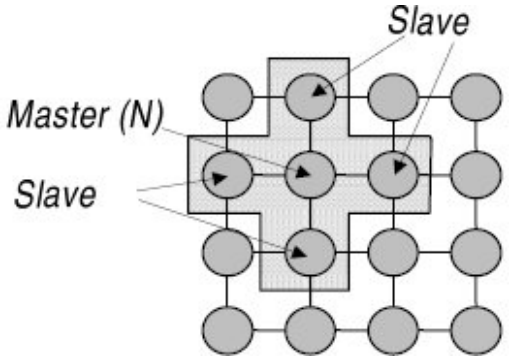


FIG. 2.3 – Domaine D de l algorithme AN dans une topologie mesh [13].

2.4.5 L'algorithme Asynchronous Round Robin (ARR)

Dans cet algorithme [14], basé sur l'approche receveur-initiative, chaque processeur maintient une variable indépendante appelée cible. A chaque fois qu'un processeur épuise sa charge de travail, il demande de recevoir une partie de la charge de travail du processeur dont le numéro d'identification correspond à la valeur contenue dans sa variable cible. Initialement, la variable cible de chaque processeur p (numéro d'identification) a pour valeur $((p+1) moduloP)$ et cette valeur est incrémentée $(moduloP)$ à chaque fois qu'elle est lue et qu'une demande de travail est émise. Cette stratégie est simple à mettre en place. Cependant il est possible que plusieurs processeurs adressent en même temps une

demande de travail au même processeur et cela n'est pas souhaitable puisque idéalement les demandes de travail doivent être propagées uniformément sur tous les processeurs.

2.4.6 La méthode d'équilibrage hiérarchique

La méthode d'équilibrage hiérarchique est une stratégie qui organise le système en domaines d'équilibrage hiérarchisés. Des nœuds spécifiques sont responsables de contrôler les opérations d'équilibrage (détection des déséquilibres, migration des tâches, etc.) à différents niveaux de la hiérarchie. Une représentation de cette organisation par un arbre binaire est donnée par la Figure 2.4. Les labels des nœuds intermédiaires à un niveau K représentent les numéros des prossesseurs responsables de contrôler les opérations d'équilibrage dans les domaines du niveau inférieur [12]. Dans cette représentation, le nombre de domaines d'équilibrage double lorsqu'on passe du niveau L_i au niveau L_{i-1}. Ainsi au plus haut niveau se trouve le nœud racine qui donne naissance à deux nœuds au niveau inférieur. Ces deux nœuds donneront chacun à leur tour naissance à deux autres nœuds et ainsi de suite jusqu'aux feuilles de l'arbre (niveau 0) constituées par tous les nœuds du système. Cet organisation en arbre binaire minimise le coût des communications et sa scalabilté lui permet de s'adapter facilement aux systèmes étendus.
Les nœuds responsables de contrôler les opérations d'équilibrage de charge au niveau L_i reçoivent l'information de charge des domaines situés au niveau L_{i-1}, calculent l'information de charge du sous arbre et la propagent vers la racine. Le mécanisme d'équilibrage hiérarchique fonctionne de manière asynchrone. Le processus d'équilibrage est déclenché à différents niveaux dans la hiérarchie par les nœuds intermédiaires dès qu'ils reçoivent des messages indiquant un déséquilibre entre les domaines du niveau inférieur. Si en valeur absolue la différence de charge entre deux domaines est supérieure à un certain seuil, alors un des domaines est considéré comme surchargé et l'autre sous chargé. Chaque nœud de la branche surchargée transfère alors une portion de sa charge vers le nœud identifié comme son correspondant situé dans le sous arbre sous chargé adjacent. Afin de faciliter le processus d'équilibrage, la portion de charge à transférer peut être identique pour tous les nœuds. Ainsi, pour un déséquilibre Δ_i au niveau i dans la hiérarchie, chaque nœud de la branche surchargée transfère une charge δ_i vers le nœud identifié comme son correspondant situé dans le sous arbre sous chargé adjacent. La charge δ_i est donnée par

la formule suivante :

$$\delta_i = \frac{\Delta_i}{2^i} \tag{2.3}$$

En théorie, l'équilibrage hiérarchique garantie qu'au niveau L_i de la hiérarchie, tous les nœuds du niveau L_{i-1} appartenant au sous arbre situé à gauche ont la même charge. De même que tous les nœuds du niveau L_{i-1}, appartenant au sous arbre situé à droite. Dans la pratique, cet hypothèse n'est vraie que si la charge ne connaît pas de fluctuations trop fréquentes.

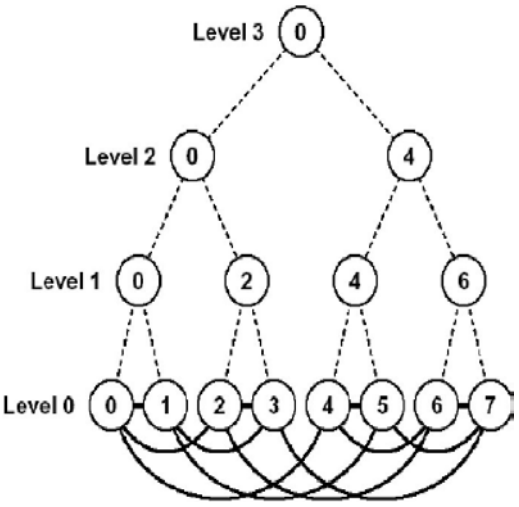


FIG. 2.4 – Organisation hierarchique d un systeme de huit nœuds interconnectés par un reseau hypercube [11]

2.4.7 Les méthodes génétiques d'équilibrage de charge

Les algorithmes génétiques constituent une nouvelle approche qui apporte une solution adaptative au problème de l'équilibrage de charge [15]. Un algorithme génétique est basé sur les principes de l'évolution et de la génétique naturelle. La solution d'un problème est codée sur une population initiale de chromosomes qui évolue par des itérations successives appelées *générations*. A chaque génération, les chromosomes les plus aptes à résoudre le problème posé sont sélectionnés pour créer la prochaine génération. La création de nouveaux chromosomes se fait par le *croisement*, la *mutation* ou la *copie* de chromosomes de la génération actuelle. Après plusieurs générations, l'algorithme converge vers une solution optimale ou sous optimale du problème. Dans [15], deux approches *GCTA (Genetic Central Task Assigner)* et *CBLB (Classifier-Based Load Balancer)*, basées sur les algorithmes génétiques, ont été proposées. La première approche utilise un

algorithme génétique pour effectuer entièrement l'équilibrage de charge tandis que la seconde améliore un algorithme d'équilibrage de charge déjà existant [16].

2.4.7.1 GCTA (Genetic Central Task Assigner)

GCTA est un algorithme centralisé où le coordinateur collecte périodiquement l'information sur l'état de charge des autres nœuds, détermine la meilleure distribution du travail dans le système et diffuse l'information à tous les processeurs qui transfèrent alors les tâches vers les nœuds spécifiés.

GCTA consiste en une population de chromosomes C qui représente toutes les décisions de transfert possibles, deux fonctions d'évaluation $E_s(C)$ et $E_c(C)$ et trois opérateurs : *croissement*, *reproduction* et *mutation*. Chaque position dans le chromosome correspond à une tâche en attente avec une valeur indiquant le nœud vers lequel elle doit être transférée. Si la tâche se trouve déjà dans le nœud indiqué alors le transfert n'est pas nécessaire.

Pour créer les premières générations de chromosomes, GCTA utilise la fonction simple $E_s(C)$ qui calcule le nombre total de transfert à effectuer et la quantité totale de travail à exécuter. Pour les dernières générations, il utilise la fonction complexe et très coûteuse $E_c(C)$ qui calcule le temps total d'exécution prévu pour un ensemble de tâches du système en tenant compte du coût des transferts.

Le croisement est opéré en divisant deux chromosomes parents et en les combinant tandis que la mutation s'opère avec une faible probabilité sur des positions particulières du chromosome. Ces deux opérations apportent la diversité nécessaire dans la population de chromosomes.

2.4.7.2 CBLB (Classifier-Based Load Balancer)

CBLB améliore l'algorithme *Centex* [16] et utilise un système classificateur situé au niveau de l'ordonnanceur central pour contrôler ses paramètres.

L'algorithme Centex utilise un nœud central pour prendre les décisions de placement. Chaque nœud envoie périodiquement l'information sur son état au nœud central.

Un système classificateur est un système d'apprentissage basé sur l'utilisation de règles *conditions/actions*. Il a plusieurs composants : les conditions d'entrée, les actions de sortie, une base de règles, un système d'exploration des règles, un algorithme de sélection et un système de rémunération.

Les conditions d'entrée pour chaque règle sont : le temps de réponse moyen, l'utilisation moyenne de chaque nœud et l'ampleur du déséquilibre de charge depuis la dernière mise à jour. Ces trois valeurs constituent la partie gauche des règles de la base de données. Les actions de sortie positionnent trois des paramètres du système : le seuil de la file d'attente des transferts, le seuil du CPU et la période des mises à jour. Ces trois actions constituent la partie droite des règles de la base de données.

La rémunération des règles est basée sur le temps de réponse moyen. Une règle est rémunérée uniquement si son franchissement fait décroître le temps de réponse moyen. Lorsque l'environnement fournit les conditions d'entrée, toutes les règles qui les vérifient, entrent en compétition pour pouvoir franchir leurs actions. Cette compétition se déroule selon un processus de sélection ou la probabilité de sélection d'une règle est proportionnelle à sa force (cumul de ses rémunérations). Les règles sélectionnées sont franchises et les changements appropriés effectués sur le système.

La création de règles a lieu périodiquement selon l'algorithme génétique jusqu'à la constitution d'un ensemble de règles performantes que le système stocke dans une base de données. CBLB permet ainsi d'optimiser le système d'équilibrage de charge.

2.4.8 Les modèles de comportement cognitifs

Les modèles de comportement cognitifs constituent une rupture avec les approches d'équilibrage de charge classiques qui sont basées sur l'analyse de l'état actuel du système et qui peuvent conduire à une hausse des communications dans le réseau. Dans cette approche on essaie d'estimer l'état futur du système en modélisant le comportement des utilisateurs qui est la véritable cause de la charge du système. Les modèles de comportement permettent d'estimer avec une grande précision la façon d'agir d'un individu sous certaines circonstances. Dans [17], le modèle de comportement *SMF (Schedule, Motivation, Fatigue)* proposé, utilise trois règles simples : le planning (journalier) du travail, la motivation et la fatigue de l'utilisateur. Le planning qui varie d'un utilisateur à un autre est défini à l'aide d'une fonction $S : R^3 \rightarrow \{0,1\}$ définie par la formule 2.4 où $t_{st} = 8 \pm 2$ indique quand débute le travail, $t_w = 8 \pm 1$ la durée de travail et t la situation à un moment quelconque.

$$S(t, t_{st}, t_w) = \begin{cases} 1 & \text{si } t \in [\mathrm{t}_{st} + 24.k, t_{st} + t_w + 24.k] \\ 0 & \text{sinon} \end{cases} \tag{2.4}$$

La motivation de l'utilisateur est modélisée à l'aide d'une fonction sinusoïdale avec une modulation de phase qui dépend de l'utilisateur. Cette fonction $M : R^2 \rightarrow [0,1]$ est définie par la formule 2.5 suivante :

$$M(t,\Delta\varphi) = \frac{1+\sin\left((t+\Delta\varphi).\frac{2\pi}{T_{MAX}}\right)}{2} \tag{2.5}$$

La fatigue accumulée par l'utilisateur est modélisée par la fonction logistique $F : R^2 \rightarrow [0,1]$ qui est définie par la formule 2.6 où t_{fcp} est le point de fatigue critique qu'on définit comme la moitié de la durée de travail t_w.

$$M(t,t_{fcp}) = \frac{1}{1+e^{-a(t-t_{fcp})}} \tag{2.6}$$

Sur la base de ces trois fonctions, on peut prédire quelques types de comportement de l'utilisateur (voir tableau 2.1).

Planning	**Motivation**	**Fatigue**	**Comportement**
pause	-	-	-
activité	élevée	basse	programmation
activité	élevée	élevée	documentation sur Internet
activité	basse	basse	jeu
activité	basse	élevée	surf sur Internet

TAB. 2.1 – Règles de comportement basées sur un schéma d'action

Avec le comportement de l'utilisateur, on peut approximer la charge de la station sur laquelle il travaille (voir tableau 2.2). Trois fonctions permettent de déterminer cette charge : *Alive* qui indique si oui ou non la station est en marche, *Processor* qui indique l'activité du processeur et *Memory* qui indique le taux d'utilisation de la mémoire. Une fois que la charge de chaque station de travail est connue, on peut calculer la charge globale du système. La dernière étape consiste à prédire l'évolution du système dans la journée en utilisant un système neuro-fuzzy. La simulation effectuée sur le modèle avec différents paramètres donne des résultats très satisfaisants.

2.5 Conclusion

Dans ce chapitre nous avons étudié le problème de l'équilibrage de charge dans les systèmes distribués et parallèles. Apres avoir posé la problématique de l'équilibrage de

Comportement	**Alive**	**Processor**	**Memory**
-	non	-	-
programmation	oui	moyenne	élevée
documentation sur Internet	oui	basse	basse
jeu	oui	élevée	moyenne
surf sur Internet	oui	basse	basse

TAB. 2.2 – Règles de charge sur le comportement de l'utilisateur

charge, nous avons présenté une classification des différentes approches proposées dans la littérature, puis nous avons analysé les principaux éléments d'un système d'équilibrage de charge. Pour terminer, nous avons étudié un certain nombre d'algorithmes dynamiques qui apportent une solution au problème de l'équilibrage de charge dans les systèmes distribués et parallèles. Dans le chapitre suivant, nous verrons comment le problème de l'équilibrage de charge est traité dans les algorithmes de datamining et en particulier dans l'extraction de règles associatives.

Chapitre 3

Équilibrage de charge dans les algorithmes d'extraction de règles associatives

Ce chapitre traite du problème de l'équilibrage de charge dans les algorithmes d'extraction de règles associatives. Après une brève présentation des motivations et techniques de datamining, nous présentons quelques algorithmes parallèles d'extraction de règles associatives. Nous étudions ensuite les approches d'équilibrage de charge utilisées dans ces algorithmes. Pour terminer, nous faisons une analyse qualitative et une comparaison des approches d'équilibrage de charge étudiées.

3.1 Le datamining : motivations et techniques

Le datamining ou fouille de données est une discipline à la frontière de plusieurs autres disciplines comme les statistiques, les bases de données, l'apprentissage, l'informatique distribué et parallèle, la visualisation des données ,etc [23]. Il est défini comme un processus itératif et interactif de découverte de modèles ou de schémas valides, compréhensibles, préalablement inconnus et potentiellement utiles à partir de grandes bases de données [18].

Aujourd'hui, avec les outils automatiques de collecte de données (code-barres, capteurs sensoriels, etc.) et les coûts de stockage et de traitement sensiblement réduits, nous assistons à une croissance phénoménale des données collectées et entreposées dans toutes

les sphères d'activité. Ces bases de données volumineuses (mesurées en Go et en To) peuvent dissimuler de précieuses informations utiles pour l'aide à la décision, la gestion des données, l'optimisation des requêtes, le contrôle de processus etc. Devant la nécessité économique et scientifique d'extraire l'information disséminée dans ces importantes masses de données et les fonctionnalités limitées offertes par les systèmes de gestion de bases de données actuelles, le datamining apparaît comme une méthode efficace d'extraction de connaissances utiles à partir de données préalablement selectionnées et préparées.

Le datamining s'applique à de nombreux domaines comme le marketing et la vente (étude du comportement des consommateurs, prédiction des ventes, espionnage industriel, etc.), la finance et les assurances (crédit, prédiction du marché, détection de fraudes, etc.) ou encore la médecine (analyse de séquence ADN, diagnostic, etc.). Il a deux objectifs :

- La *prédiction* qui consiste à construire un modèle capable de prédire les valeurs d'attributs qu'on juge intéressants à partir de valeurs connues d'autres attributs.
- La *description* qui consiste à trouver des motifs compréhensibles aux humains qui décrivent les données.

Les opérations de datamining peuvent être classées en deux catégories : les opérations mues par la *vérification* pour lesquelles l'utilisateur formule une hypothèse que le système essaie de valider et les opérations mues par la *découverte* qui permettent d'extraire automatiquement de nouvelles informations. On trouve parmi les opérations mues par la découverte l'extraction de règles associatives à laquelle nous nous intéressons dans ce mémoire, la recherche de motifs séquentiels, la classification, la régression, le groupement (clustering), la découverte de similitudes et la detection d'écarts.

3.1.1 L'extraction de règles associatives

Elle consiste à découvrir dans une base de données de transactions, les ensembles d'attributs apparaissant simultanément et les règles qui existent entre eux. Prenons l'exemple d'un supermarché ou les articles achetés par chaque consommateur sont enregistrés en même temps dans une base de données comme une transaction. A partir de cet exemple, on peut trouver une règle associative de la forme *"90% des utilisateurs qui achètent du thé et du sucre, achètent aussi de la menthe"*. *Thé* et *sucre* constituent l'antécédant de cette règle, *sucre* en est la conséquence et 90% la confiance. Les règles associatives ont été utilisées avec succès dans des domaines comme l'aide à la planification, l'aide au diagnostic en recherche médical, l'amélioration des processus de télécommunications, de l'organisa-

tion et de l'accès aux sites Internet, l'analyse d'images, de données spatiales, statistiques et géographiques, etc.

3.1.2 La recherche de motifs séquentiels

Dans de nombreuses bases de données, une étiquette de temps est associée à chaque donnée de la base. Avec des données de cette nature, rechercher des motifs séquentiels consiste à découvrir les séquences d'événements apparaissant fréquemment ensemble. Par exemple, en analysant les transactions effectuées dans un magasin de vente d'appareils electroménagers, nous pouvons obtenir une information du genre *"70% des consommateurs qui achètent un téléviseur, achètent un DVD le mois suivant"*. La recherche de motifs séquentiels trouve des applications dans le suivi des clients, le mailing, la recherche de séquences génétiques, l'aide au diagnostic, la catégorisation de documents, etc.

3.1.3 La classification et la régression

La classification consiste à assigner à de nouveaux objets une ou plusieurs classes prédéfinies en recherchant un ensemble de prédicats caractérisant une classe d'objets qui peut être appliqué à des objets inconnus pour prévoir leur classe d'appartenance. C'est une technique qui s'applique à divers domaines comme la finance, la detection de fraudes, le diagnostic médical, etc. Par exemple, une banque peut vouloir classer ses clients pour savoir si elle doit leur accorder un prêt ou non. Il existe plusieurs méthodes de classification comme les réseaux neuronaux, les algorithmes génétiques, ou encore les arbres de décisions qui sont les plus utilisés parce qu'étant très faciles à construire et à comprendre. La classification s'intéresse aux classes à valeurs discrètes contrairement à la régression qui s'intéresse elle aux classes à valeurs réelles.

3.1.4 Le groupement (clustering)

Le clustering consiste à partitionner une base de données en sous-ensembles ou groupes (clusters) tels que les éléments d'un groupe partagent un ensemble de propriétés communes qui les distinguent des éléments des autres groupes. Le clustering permet d'obtenir des groupes d'objets ayant un comportement similaire lorsque les classes ne sont pas prédéfinies. Il peut, dans la phase initiale de la classification, fournir une base utile dans la

définition des classes. Le clustering s'applique à beaucoup de domaines comme la démographie pour pouvoir par exemple identifier les traits communs d'un groupe de personnes.

3.1.5 La recherche de similitudes

La découverte de similitudes dans une base de données consiste à trouver les enregistrements les plus proches d'un enregistrement donné ou toutes les paires d'enregistrements proches d'une distance donnée. Ce type de découverte s'applique particulièrement aux bases de données spatiales et temporelles.

3.1.6 La détection d'écarts

La détection d'écarts consiste à trouver les enregistrements qui présentent des différences sensibles avec les autres enregistrements. cette détection peut être utilisée pour detecter des anomalies ou des bruits.

3.2 Parallélisation de l'extraction de règles associatives

L'extraction de règles associatives est un problème très important en datamining. Elle a pour origine l'analyse des transactions effectuées dans un supermarché. Ce problème a été largement étudié [21, 22, 24, 25, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38] et de nombreux algorithmes d'extraction de règles associatives ont été proposés. Dans cette section nous étudions quelques algorithmes parallèles d'extraction de règles associatives mais au préalable nous donnons une formulation de ce problème.

Soit $I = \{i_1, \ldots, i_m\}$ un ensemble de m attributs appelés *items*. Un *itemset* X est un sous ensemble d'items, i.e. $X \subseteq I$. Une *transaction* $t = (TID, X)$ est un 2-tuple où *TID* est un identifiant unique associé à t et X un itemset. Une transaction t contient un itemset Y si et seulement si $Y \subseteq X$. Une *base de données de transactions* $\mathcal{D}$ est un ensemble de transactions. Une règle associative est une implication de la forme $X \Longrightarrow Y$, où $X \subset I, Y \subset I$, et $X \cap Y = \emptyset$. Deux mesures sont definies pour une règle associative : le *support* et la *confiance*. La règle $X \Longrightarrow Y$ a pour confiance c dans $\mathcal{D}$ si $c\%$ des transactions dans $\mathcal{D}$ qui contiennent X contiennent aussi Y. La règle $X \Longrightarrow Y$ a pour support s dans $\mathcal{D}$ si $s\%$ des transactions dans $\mathcal{D}$ contiennent $X \cup Y$. Par exemple, dans la base de transactions de table 3.1, le support de la règle $\{crème\} \Longrightarrow \{pain\}$ est égal 2/5 = 40% et sa confiance est égale

2/4 = 50%.

Étant donné une base de transactions $\mathcal{D}$, la recherche de règles associatives consiste

TID	items
1	pain, crème, eau
2	crème
3	pain, crème, vin
4	eau
5	crème, eau

TAB. 3.1 – Base de transactions

à trouver toutes les règles ayant un support et une confiance supérieurs aux seuils (*minsup* et *minconf*) fournis par l'utilisateur. Ce problème peut être décomposé en deux sous problèmes :

1. Trouver tous les *itemsets fréquents* (itemsets qui ont un support supérieur à *minsup*).
2. Dériver à partir des itemsets fréquents toutes les règles qui ont une confiance supérieure ou égale à *minconf*.

Dans la plupart des algorithmes d'extraction de règles associatives, l'accent est mis sur la recherche des itemsets fréquents qui demande plus de calculs. Ces algorithmes appartiennent principalement à trois grandes familles [28].

– Les algorithmes qui énumèrent tous les itemsets fréquents. Ils ont l'inconvénient de produire des règles redondantes lors de la génération des règles associatives. C'est le cas des algorithmes *Apriori* [21] et *FP-Growth* [31].
– Les algorithmes qui génèrent seulement les itemsets fréquents maximaux. Ils réduisent la taille des résultats mais ne fournissent pas le support des sous ensembles, support qui est nécessaire pour la génération des règles associatives. C'est le cas des algorithmes *Maxminer* [36], *MAFIA* [38] et *GenMax* [26].
– Les algorithmes qui enumèrent les itemsets fermés frequents. Ils réduisent considérablement la taille des resultats tout en fournissant les informations nécessaires pour la génération des règles associatives. C'est le cas des algorithmes *Closet+* [32], *Close* [37], et de *CHARM* [27].

Nous limiterons notre étude à la premiére famille d'algorithmes car c'est pour cette famille qu'il exite des tentatives de parallélisation.

Les algorithmes de recherche de tous les itemsets fréquents peuvent être classés en deux catégories : Ceux comme *Apriori* et basés sur l'approche *Tester-et-Générer* et ceux comme *FP-Growth* basés sur l'approche *Diviser-pour-Régner*.

Dans l'approche Tester-et-Générer, à chaque itération $k \geq 1$, les $(k+1)$-itemsets candidats sont formés itérativement en joignant les k-itemsets fréquents. La base de données est ensuite parcourue pour le calcul du support des itemsets candidats. Cette méthode pose cependant quelques problémes comme le nombre important de candidats potentiels qui peuvent être générés et le nombre de parcours de la base de données.

Dans l'approche Diviser-pour-Régner, la base de données est projetée dans une structure de données compacte à partir de laquelle les itemsets fréquents sont extraits. Cette méthode a aussi ses inconveniants comme la taille importante de la structure de données pour certains types de données et les nombreux resultats intermédiaires qui sont produits, comme par exemple les bases de données conditionnelles dans le cas de l'algorithme FP-Growth.

Les premiers algorithmes de recherche d'itemsets fréquents sont pour l'essentiel des algorithmes séquentiels, qui malgré leur efficacité ne garantissent pas la scalabilité en terme de taille des données, de dimension (nombre d'items) et de temps d'exécution. Pour pallier à ces insuffisances, la recherche d'itemsets fréquents dans des environnements d'exécution parallèles et distribués apparaît comme une solution naturelle. Dans le reste de cette section nous présentons un échantillon non exhaustif mais assez représentatif des nombreux algorithmes parallèles de recherche d'itemsets fréquents proposés dans la littérature.

3.2.1 L'algorithme IDD (Intelligent Data Distribution)

IDD [29] est une amélioration de l'algorithme *Data Distribution (DD)* qui est une parallélisation de l'algorithme *Apriori*.

L'algorithme DD partitionne les transactions et les itemsets candidats entre les différents processeurs et laisse à chaque processeur la responsabilité de calculer le support de l'ensemble des itemsets candidats de sa partition pour toutes les transactions de la base de données. Pour ce faire, chaque processeur doit envoyer aux autres processeurs sa base de données locale.

L'algorithme IDD à été implémenté sur une architecture à mémoire distribuée et il remédie aux problèmes posés par le modèle de communication de l'algorithme DD (voir

[22] pour plus de détails) en utilisant une communication *all-to-all* basée sur une topologie d'anneau virtuel de processeurs. Chaque processeur dispose de deux tampons : *SBuf* pour l'envoi des données et *RBuf* pour la réception des données. Initialement *SBuf* est rempli avec un bloc des données locales. Ensuite chaque processeur initie une opération asynchrone d'envoi de données vers son voisin de droite avec *SBuf* et une opération asynchrone de réception de données de son voisin de gauche avec *RBuf*. Durant ces opérations, chaque processeur traite les transactions dans *SBuf* et calcule le support des candidats qui lui sont assignés. A la fin de cet opération, les rôles de *SBuf* et *RBuf* sont permutés. Ce processus est répété $(p-1)$ fois où p désigne le nombre de processeurs. Le mode de partitionnement des candidats de l'algorithme IDD est aussi différent de celui de DD. Dans IDD, les itemsets candidats sont partitionnés de manière intelligente sur la base de leur *item préfixe*. Ce partitionnement permet de savoir rapidement si une transaction contient un des itemsets candidats stockés sur un nœud donné.

3.2.2 L'algorithme HPA (Hash Partioned Apriori)

L'algorithme HPA [33] est conçu dans le même esprit que IDD. Il partitionne les itemsets candidats entre les différents processeurs à l'aide d'une fonction de hachage. Ce partitionnement permet résoudre les problèmes liés au débordement de la mémoire lorsque le nombre d'itemsets candidats est important. Il réduit aussi le nombre de communications puisque toutes les transactions de la base de données ne sont plus diffusées. HPA a été implémenté sur un cluster de PCs et chaque noeud procéde comme suit à l'itération k :

1. Génération des k-itemsets candidats : chaque processeur génère les k-itemsets candidats en utilisant les $(k-1)$-itemsets fréquents de l'étape précédente. La fonction de hachage est ensuite appliquée aux itemsets candidats pour déterminer l'ID (identifiant) de leur processeur hôte. Chaque itemset candidat est ensuite inséré dans la table de hachage de son processeur hôte.
2. Calcul du support : chaque processeur génère l'ensemble des k-itemsets pour les transactions stockées sur son disque local en éliminant ceux dont le support est inférieur à *minsup*. La fonction de hachage utilisée dans la phase 1 est alors appliquée à chaque k-itemset pour determiner l'ID de son processeur hôte. Chaque processeur est ensuite responsable d'incrémenter la valeur du support pour les itemsets générés localement et ceux envoyés par les autres processeurs.

3. Détermination des itemsets fréquents : une fois que toutes les transactions sont traitées, chaque processeur détermine localement les itemsets fréquents à partir de ses itemsets candidats. La totalité des itemsets fréquents est obtenue en sommant les itemsets fréquents provenant de tous les processeurs.

3.2.3 L'algorithme CCPD (Common Candidate Partitioned Database)

L'algorithme CCPD [24] est une implémentation de l'algorithme Apriori sur une architecture à mémoire partagée. Dans CCPD, la base de données est logiquement partitionnée entre les différents processeurs qui utilisent un arbre de hachage commun des candidats. La construction de l'arbre de hachage est parallélisée. Ainsi chaque processeur génère un sous-ensemble disjoint de candidats. Pour garantir la cohérence des données, un verrou est associé à chaque feuille de l'arbre. Lorsqu'un processeur veut insérer un candidat dans l'arbre, il commence à la racine et applique successivement le hachage sur les items constituant l'itemset candidat jusqu'à atteindre une feuille. Ensuite, le processeur acquiert le verrou associé à la feuille et insère le candidat. Pour le calcul des supports, chaque processeur compte la fréquence des itemsets de la base de données locale et le processeur maître selectionne ensuite les itemsets fréquents.

3.2.4 L'algorithme pFP-Growth (parallel FP-Growth)

L'algorithme pFP-Growth [35] est une parallélisation de l'algorithme FP-Growth [31]. Il a été implémenté sur un cluster de PCs. Dans FP-Growth, il y a deux opérations principales : la construction de la structure de données *FP-Tree (arbre de transactions)* et la recherche des itemsets fréquents dans le FP-Tree. Pour trouver les itemsets fréquents, FP-Growth traverse les noeuds du FP-tree en commençant par l'itemset le moins fréquent de la *Flist* (liste des 1-itemsets fréquents). En visitant chaque noeud, FP-Growth collecte les *chemins de préfixe* du noeud pour former sa *base de données conditionnelle* [31]. A partir de cette base de données conditionnelle, FP-Growth crée le *FP-tree conditionnel* de l'item contenu dans le noeud. Ce processus est répété jusqu'à ce que le FP-tree conditionnel résultant soit vide, ou qu'il contienne un chemin unique. Pour tous les items de la *Flist*, le même processus itératif est utilisé pour trouver tous les itemsets fréquents les contenant.

Puis que les bases de données conditionnelles peuvent être traitées de manière indépendante, elles peuvent être traitées parallèlement. Ainsi, chaque noeud reçoit et traite

intégralement une base de données conditionnelle avant d'en recevoir une autre. Pour cette parallélisation de FP-Growth, deux processus sont nécessaires sur chaque noeud : un processus SEND et un processus RECV. Après le premier parcours des bases de transaction locales, les processus SEND échangent les supports locaux des itemsets fréquents pour déterminer leur support global et contruire la *Flist*. Au second parcours des bases de données locales, les processus SEND construisent les FP-Tree locaux à partir des quels ils génèrent les bases de données conditionnelles qui sont ensuite routées vers leur destination en appliquant une fonction de hachage à l'itemset auquel elles sont associées. Le processus RECV des noeuds hôtes collectent toutes les bases de données conditionnelles envoyées par les processus SEND et leurs appliquent l'algorithmed FP-Growth.

3.3 Les mécanismes d'équilibrage de charge dans les algorithmes HPA, IDD, CCPD et pFP-Growth

La plupart des algorithmes parallèles d'extraction de règles associatives existants utilise un mécanisme d'équilibrage de charge statique inhérent au partitionnement initial des données et qui suppose un environnement d'exécution dédié et homogène. Cependant, la réalité des systèmes parallèles, qui sont de plus en plus hétérogènes, impose qu'on s'intérésse aux mécanismes d'équilibrage de charge dynamique. Dans cette section, nous traitons des mécanismes d'équilibrage de charge utilisés dans les algorithmes parallèles d'extraction de règles associatives etudiés dans la section précédente.

3.3.1 Équilibrage de charge statique

L'équilibrage de charge statique consiste à partitionner initialement la charge de travail entre les processeurs en utilisant une fonction heuristique de coût. Dans ce schéma, une fois que la répartition du travail est faite, il n'est pas possible de faire migrer des données ou des traitements d'un nœud vers un autre pour corriger les déséquilibres pouvant survenir par la suite. Un des critères d'un bon partitionnement est d'avoir une répartition équitable de la charge de travail entre tous les processeurs. Pour cette raison, le choix de l'algorithme de partitionnement est particulièrement crucial.

3.3.1.1 Le partitionnement dans CCPD

L'équilibrage de charge dans CCPD se fait lors de la génération des itemsets candidats et lors du partitionnement de la base de données. Dans la phase de génération des itemsets candidats C_k à l'étape k, les $(k-1)$-itemsets fréquents de l'étape précédente sont regroupés en classes d'équivalence sur la base de leur $(k-2)$-itemsets préfixes. A chaque itemset i de ces classes d'équivalence est associé un poids w_i représentant le nombre d'itemsets candidats qu'il génère. Le poids w_i est donné par $w_i = n-i-1$, pour $i = 0,\ldots,n-1$ ou n est le cardinal de la classe. Ces classes d'équivalence sont partitionnées en utilisant le partitionnement *bitonique* qui consiste à trier tous les w_i et à assigner les itemsets des classes d'équivalence aux processeurs de manière à ce que l'itemset avec le plus grand w_i soit toujours assigné au processeur le moins chargé. La répartition des transactions de la base de données se fait en utilisant le *partitionnement par bloc* qui permet d'avoir sensiblement le même nombre de transactions au niveau de chaque processeur. Malgré le partitionnement des classes d'équivalence, l'arbre de hachage résultant n'est pas équilibré. Le partitionnement bitonique peut être aussi utilisé pour équilibrer l'arbre de hachage en remplaçant tout simplement la taille de la table de hachage $\mathcal{F}$ par le nombre de processeurs P pour dériver les partitions $\mathcal{A}_0,\ldots,\mathcal{A}_{\mathcal{F}-1}$ de chaque processeur. Les $\mathcal{A}_i$ sont traitées comme une classe d'équivalence et sont implémentées par le biais d'un vecteur d'indirection.

3.3.1.2 Le partitionnement dans IDD

La stratégie utilisée pour équilibrer la charge des nœuds dans IDD consiste à répartir équitablement les itemsets candidats entre les différents processeurs. Ce partitionnement permet d'avoir des arbres de hachage de taille égale sur tous les nœuds. Pour ce faire, on calcule pour chaque item le nombre d'itemsets candidats dont il (l'item) est le préfixe. Ensuite, dans un système de p processeurs, l'algorithme partitionne en p parties les items préfixes de sorte que le nombre d'itemsets candidats, qu'ils préfixent, soit approximativement le même dans chacune des p parties. Une fois que l'emplacement de chaque itemset est determiné, chaque processeur régénère localement et stocke les itemsets qui lui sont assignés. Dans cet approche, à chaque fois que le nombre d'itemsets candidats commençant par un même item dépasse un certain seuil, il devient difficile d'avoir une répartition équitable des itemsets candidats. Pour résoudre ce problème, le second item des itemsets candidats est utilisé pour les partitionner.

3.3.2 Équilibrage de charge dynamique

Par nature, les algorithmes d'extraction de règles associatives sont dynamiques. Des déséquilibres peuvent survenir au niveau des nœuds aprés le partitionnement initial des données et le cas échéant, un équilibrage de charge dynamique est nécessaire pour les corriger. Ainsi, les algorithmes HPA et pFP-Growth utilisent un équilibrage de charge dynamique.

3.3.2.1 L'équilibrage de charge dans HPA

Dans l'approche d'équilibrage utilisée pour HPA [34], un nœud coordinateur collecte toutes les informations provenant des autres nœuds, calcule restT_i (temps de traitement restant pour le nœud i) et contrôle la distribution de la charge de travail. Comme l'objectif est que tous les processeurs terminent leur travail en même temps, l'algorithme contrôle dynamiquement la charge allouée à chaque processeur afin qu'ils aient tous le même restT_i. Pour chaque nœud i, on définit une fonction *skew* donnée par les expressions 3.1 et 3.2 mesurant le biaisement des données.

$$skew = \frac{max(restT_i) - min(restT_i)}{moy(restT_i)} \tag{3.1}$$

$$\begin{cases} skew \leq seuil & \text{pas de skew} \\ skew > seuil & \text{skew} \end{cases} \tag{3.2}$$

Le coordinateur peut ainsi juger que le contrôle de la charge est nécessaire pour un nœud i si la valeur de sa fonction *skew* dépasse un certain seuil. Dans ce cas, une première stratégie appelée migration de candidats est appliquée et si la valeur de la fonction *skew* continue d'être élevée, une deuxième stratégie appelée migration de transactions est appliquée.

La migration de candidats : La migration de candidats consiste à réallouer les itemsets candidats aux différents nœuds ; ainsi, tous les nœuds auront le même restT_i. Pour ce faire, le nœud coordinateur calcule le poids de chaque nœud en fonction du support des itemsets de leur table de hachage et en déduit un plan de migration de candidats qui sera envoyé à tous les nœuds pour le démarrage du processus de migration. L'implémentation de cette stratégie exige un nouveau mappage des tables de hachage. Au lieu d'un mappage itemset par itemset qui demande beaucoup d'espace, la migration se base sur les

intervalles d'appartenance des itemsets dans l'arbre de hachage. Lorsque tous les candidats d'un nœud ont migré et que la valeur de la fonction *skew* continue d'être élevée, il devient alors nécessaire d'effectuer une migration de transactions.

La migration de transactions : La migration de transactions consiste à envoyer à d'autres nœuds une partie des transactions qui étaient jusque là assignées à un nœud spécifique et à leur déléguer la génération des *k*-itemsets. Pour ce faire, chaque nœud maintient une liste, dynamiquement mise à jour, des nœuds destination et des tâches qui leur sont assignées. Lorsque la migration de candidats ne suffit pas pour réduire la valeur de la fonction *skew*, le nœud coordinateur établit un plan de migration de transactions qui sera envoyé à tous les nœuds pour le démarrage du processus de migration. Cette stratégie permet effectivement d'équilibrer la charge en transférant la charge de travail dûe au processus de génération des *k*-itemsets des nœuds chargés vers ceux qui ont une grande puissance de calcul.

3.3.2.2 L'équilibrage de charge dans pFP-Growth

La stratégie utilisée dans pFP-Growth [35] vise à répartir équitablement la charge de travail (recherche des itemsets fréquents) et la structure de données FP-Tree entre les différents nœuds du cluster. Pour cela deux stratégies sont employées : la migration des branches des FP-Tree dont la mémoire des nœuds hôtes est congestionnée et la migration des bases de données conditionnelles qui exigent de nombreux calculs lors de la recherche des itemsets fréquents.

La migration des bases de données conditionnelles : La migration des bases de données conditionnelles est la stratégie utilisée pour corriger les éventuels déséquilibres qui peuvent survenir après le partitionnement initial. Cette stratégie repose sur le concept de *profondeur de chemin*. Dans une base de données conditionnelle, la profondeur de chemin désigne la longueur du plus grand motif satisfaisant au support minimal. Lorsque le traitement complet d'une base de données conditionnelle est pris comme unité d'exécution, le temps d'exécution varie d'un nœud à un autre. Il faut donc trouver une granularité qui garantit un temps d'exécution uniforme pour tous les nœuds. Pour une répartition équitable de la charge de travail, la profondeur de chemin est prise comme mesure de la granularité d'exécution. Les bases de données conditionnelles avec une profondeur de chemin infé-

rieure à un seuil minimal sont traitées intégralement et immédiatement par le noeud hôte tandis que celles dont la profondeur de chemin est supérieure au seuil fixé ne seront traitées qu'à l'étape de génération des bases de données conditionnelles qu'elles engendrent. Ces bases de données conditionnelles sont ensuite stockées sur le noeud qui les a généré où elles peuvent être traitées ou bien être envoyées à d'autres nœuds.

La migration des branches de la structure de données FP-Tree : Lorsque la taille des données est importante, l'arbre (FP-Tree) peut ne pas tenir dans la mémoire d'un seul hôte. Ainsi, dans le cas d'un système à memoire distribuée, l'espace disponible sur les autres nœuds pourrait être utilisé. Lorsque la taille de la mémoire disponible sur un hôte est inférieure à un certain seuil fixé à l'avance, la plus grande branche du FP-Tree local va migrer vers une destination déterminée par une fonction de hachage [35]. Lorsqu'une branche quitte un nœud pour aller vers une autre destination, toutes les branches qui lui sont identiques et qui se trouvent sur d'autres nœuds doivent aussi migrer vers cette destination ou elles seront fusionnées. En lisant la base de transactions, les transactions ordonnées dont le préfixe est l'un des items racines des transactions ayant migrées sont envoyées vers le nœud approprié. A la fin si la taille des FP-Tree locaux n'est pas toujours équilibrée, certaines branches des FP-Tree surchargés sont réallouées à d'autres nœuds.

3.4 Étude comparative

Après avoir étudié les mécanismes d'équilibrage de charge mis en œuvre dans un certain nombre d'algorithmes d'extraction de règles associatives, nous faisons dans cette section une analyse qualitative et une étude comparative de ces mécanismes.

3.4.1 Analyse qualitative

L'analyse qualitative des différentes approches d'équilibrage de charge permet de dégager les défauts et les qualités de chacunes d'elles que nous allons passer en revue ci-dessus.

IDD (Intelligent Data Distribution) : IDD réduit linéairement le coût du parcours de l'arbre de hachage et celui de l'exploration de ses feuilles par un facteur de P (nombre de processeurs). Cependant, ces coûts dépendent non seulement de la taille de l'arbre de

hachage mais aussi des items contenus dans les transactions. L'algorithme de partitionnement ne peut donc pas garantir un bon équilibrage de charge en assignant le même nombre de candidats à tous les processeurs. En effet, les processeurs auxquels sont assignés des items préfixes ayant un support élevé risquent de recevoir plus de requêtes de calcul que les autres. Il se créent ainsi un déséquilibre qui s'accentue lorsque le nombre de processeurs est trop grand par rapport au nombre d'itemsets.

CCPD (Common Candidate Partitioned Database) : CCPD réduit linéairement le coût du parcours de l'arbre de hachage et celui de l'exploration de ses feuilles par un facteur de P (nombre de processeurs). Le partitionnement des classes d'équivalence combiné à l'équilibrage de l'arbre de hachage améliore nettement les performances de l'algorithme. Cependant, le mode de partitionnement (par bloc) de la base de données n'assure pas de manière rigoureuse l'équilibrage de charge puisqu'il ne vise qu'à assigner le même nombre de transactions à chaque processeur. Or la charge de travail d'un processeur dépend essentiellement du nombre d'items contenus dans les transactions et de leur support.

HPA (Hash Partioned Apriori) : Les résultats expérimentaux montrent que la migration de candidats et la migration de transactions utilisées dans HPA améliorent nettement le temps d'éxécution. Plus les skews sont élevés plus les performances de l'algorithme sont meilleures. Il faut cependant noter que la méthode d'estimation de la charge des nœuds basée sur le support des itemsets fréquents, n'est pas précise et les erreurs faites sur cet estimation s'accumulent graduellement. De plus, il est très difficile de determiner la valeur optimale du seuil de la fonction skew et on lui associe une valeur statique celle-ci peut ne pas convenir dans certaines situations. Il faut aussi remarquer que la synchronisation des nœuds dans la phase de migration des candidats entraine une surcharge de travail lorsque la taille du système est importante. Enfin, avec la centralisation du dispositif d'équilibrage de charge, les déséquilibres ne seront pas corrigés en cas de panne du coordinateur central.

pFP-Growth (parallel FP-Crowth) : Le concept de profondeur de chemin utilisé dans pFP-Growth permet effectivement d'équilibrer la charge de travail des nœuds d'un cluster puisqu'il permet aux nœuds surchargés de décomposer leur charge en tâches plus petites qu'ils enverront à d'autres nœuds. La suppression des branches redondantes des FP-Tree en les regroupant (par fusion) sur un seul nœud réduit aussi les pertes d'espace mémoire.

Cependant les résultats expérimentaux montrent que la valeur du seuil de la profondeur de chemin est un facteur determinant dans les performances de l'algorithme. En effet, lorsque la valeur du seuil de la profondeur de chemin est faible, il y a beaucoup de bases de données conditionnelles de petite taille qui doivent être stockées, ce qui augmente le surplus de travail et lorsque la valeur du seuil de la profondeur de chemin est élévée, la granularité d'exécution est très grande et la charge des nœuds n'est pas suffisament équilibrée. Ainsi, avec une valeur statique du seuil de la profondeur de chemin, le mécanisme d'équilibrage de charge pourrait ne pas avoir d'effets significatifs sur le système. Il faut aussi souligner que dans la migration des bases de données conditionnelles comme dans celle des branches des FP-Tree, les nœuds sources ignorent la charge des nœuds receveurs. Lorsque ces derniers sont déja surchargés, ils continuent de recevoir les surcharges des autres nœuds, ce qui fait qu'accentuer les déséquilibres. Enfin, l'échange de bases de données conditionnelles necessite le maintien d'un tampon de données au niveau de chaque nœud et lorsque le nombre de nœuds augmente, cet échange crée des encombrements dans le réseau.

3.4.2 Comparaison

L'objectif de cette comparaison est de faire sortir les aspects communs et les particularités des approches d'équilibrage de charge utilisées dans les algorithmes HPA, CCPD, IDD et pFP-Growth. Dans cette comparaison, nous nous intéressons à la manière dont les différentes politiques d'équilibrage de charge sont implémentées.

IDD (Intelligent Data Distribution)

Politique de transfert : sans rejet.

Tous les nœuds du système sont concernés par le placement des candidats et des transactions et par la migration des transactions.

Politique de localisation : source initiative.

Chaque nœud source envoie une partie de ses transactions à son voisin de gauche.

Politique de selection : sans filtrage.

Toutes les tâches sont éligibles pour la migration des transactions. Chaque processeur

envoie toutes ses transactions à tous les autres processeurs.

Politique d'information : aucune.
Il n'y a pas d'informations échangées entre les nœuds.

Politique de caractérisation de la charge : vecteur de charge dont les composantes sont le nombre de transactions N et le nombre de candidats M.
Le nombre de transactions initialement allouées à un nœud est déterminé avant l'exécution de l'algorithme en divisant le nombre total de transactions par le nombre de nœuds. Durant la phase d'exécution, chaque nœud reçoit la totalité des transactions. Le nombre de candidats alloués à un nœud est déterminé à chaque itération en divisant le nombre total d'items préfixes par le nombre de nœuds de sorte que le nombre de candidats générés par chaque groupe d'items préfixes soit le même.

Politique d'assignation des tâches : placement des candidats et des transactions, migration de transactions.
Les transactions sont placées avant l'exécution de l'algorithme mais lors de l'exécution, les processeurs s'échangent les transactions qui leurs étaient allouées initialement. A chaque nouvelle étape de la détermination des itemsets fréquents, les items préfixes des candidats sont placés sur les processeurs qui génèrent ensuite les candidats et calculent leur support.

CCPD (Common Candidate Partitioned Database)

Politique de transfert : sans rejet.
Tous les nœuds du système sont concernés par le placement des candidats et des transactions.

Politique de localisation : aucune.
Il n'y a pas de politique de localisation car il n'y a pas d'échange de candidats ou de transactions.

Politique de selection : aucune.
Il n'y a pas de politique de selection car il n'y a pas d'échange de candidats ou de transactions.

Politique d'information : aucune.
Il n'y a pas d'informations échangées entre les nœuds.

Politique de caractérisation de la charge : vecteur de charge dont les composantes sont le nombre de transactions N et le nombre de candidats M.
Le nombre de transactions allouées à un nœud est déterminé avant l'exécution de l'algorithme en divisant le nombre total de transactions par le nombre de nœuds. Le nombre de candidats qu'un nœud doit générer et insérer dans l'arbre de hachage est déterminé à chaque itération en divisant le nombre total d'itemsets fréquents par le nombre de nœuds, de sorte que le nombre de candidats générés par chaque groupe d'itemsets fréquents soit le même.

Politique d'assignation des tâches : placement des candidats et des transactions.
Les transactions sont assignées aux processeurs avant l'exécution de l'algorithme tandis que les candidats, qui doivent être générés à chaque étape sont assignés, à la formation des classes d'équivalence.

HPA (Hash Partioned Apriori)

Politique de transfert : seuil.
Si la valeur de la fonction skew d'un nœud est supérieure au seuil fixé, ce dernier est considéré comme source et dans le cas contraire le nœud sera considéré comme receveur.

Politique de localisation : centralisée et source initiative pour la migration de candidats, centralisée pour la migration de transactions.
Lors du calcul des supports, la destination des candidats, qui doivent être traités sur d'autres nœuds, est determinée par le nœud source à l'aide d'une fonction de hachage.Cependant pour corriger les déséquilibres, les nœuds receveurs, aussi bien pour les transactions que pour les candidats, sont déterminés par le coordinateur central.

Politique de selection : filtrage pour les candidats et sans filtrage pour les transactions.
Une fonction de hachage est appliquée aux itemsets candidats, générés à partir des transactions locales, et seuls ceux dont la valeur retournée par la fonction de hachage est différente de l'ID de l'hôte sont envoyés vers d'autres nœuds. Pour la migration de candidats, ce sont les candidats au support élevé qui sont éligibles. Par contre pour la migration de transactions, toutes les transactions sont éligibles.

Politique d'information : centralisée.
Le coordinateur central collecte périodiquement l'information de charge des noeuds en calculant la valeur de leur fonction skew.

Politique de caractérisation de la charge : vecteur de charge dont les composantes sont le nombre de transactions N et le nombre de candidats M.
Le nombre de transactions et de candidats alloués initialement à un nœud est déterminé avant l'exécution de l'algorithme en divisant le nombre total de transactions et de candidats par le nombre de nœuds. Durant la phase d'exécution, à chaque itération, la deuxième composante du vecteur de charge M d'un nœud est déterminée sur l'aide d'un facteur de poids (basé sur le support) sur chaque itemset fréquent de l'itération précédente.

Politique d'assignation des tâches : placement des candidats et des transactions, migration de transactions et de candidats.
Les transactions sont placées sur les nœuds avant l'exécution de l'algorithme et les candidats à la formation de chaque nouvelle génération. Lors du calcul des supports, les candidats, contenus dans des transactions initialement allouées à des processeurs différents de leur hôte, sont envoyés vers les destinations appropriées. Il y a aussi une migration de candidats qui peut être suivie d'une migration de transactions lorsque des déséquilibres sont constatés.

pFP-Growth (parallel FP-Growth)

Politique de transfert : seuil.
Dans un nœud, si la valeur de la profondeur de chemin d'une base de données conditionnelle est supérieure au seuil fixé, le nœud sera considéré comme source (migration

de candidats). De même lorsque l'espace mémoire restant dans un nœud est inférieur au seuil fixé, le nœud sera aussi considéré comme source (migration de transactions). Pour la migration des transactions et des candidats tous les nœuds peuvent être receveurs.

Politique de localisation : source initiative.
La destination des bases de données conditionnelles et des branches des FP-Tree est déterminée de manière aléatoire par le nœud source.

Politique de selection : filtrage.
Les bases de donneés conditionnelles dont la taille est supérieure à la granularité d'exécution sont réduites et ce sont ces bases réduites qui sont envoyées aux autres nœuds. Les branches dont le support de la racine est élevé sont celles qui migreront lorsque les FP-Tree ne peuvent pas tenir en mémoire et avec elles les transactions dont le préfixe est la racine de l'une de ces branches.

Politique d'information : aucune.
Il n'y a pas d'informations échangées entre les nœuds.

Politique de caractérisation de la charge : vecteur de charge dont les composantes sont le nombre de transactions N et le nombre de bases de donneés conditionnelles D.
Le nombre de transactions, dont est fonction la taille des FP-Tree locaux et le nombre de bases de donneés conditionnelles alloués initialement à un nœud, est déterminé avant l'exécution de l'algorithme en divisant le nombre total de transactions et d'items fréquents par le nombre de nœuds. Durant la phase d'exécution, une fois que les FP-Tree locaux sont équilibrés, la taille de la base de donneé conditionnelle couramment traitée représente la charge du nœud. Cette taille est exprimée en fonction de la profondeur de chemin.

Politique d'assignation des tâches : placement des transactions et des candidats, migration des transactions et des candidats.
Les transactions et les candidats sont placés sur les nœuds avant l'exécution de l'algorithme. Lors de la formation des FP-Tree locaux, des transactions représentées sous la forme de branches des FP-Tree peuvent migrer d'un processeur à un autre si elles ne peuvent pas tenir en mémoire. Les candidats sous la forme de bases de données condi-

Algorithmes	Politiques					
	Transfert	Localisation	Selection	Information	Determination de la charge	Assignation des tâches
IDD	sans rejet	source initiative[1]	sans filtrage	aucune	vecteur L(M,N)	migration[2] placement[3]
CCPD	sans rejet	aucune	aucune	aucune	vecteur L(M,N)	placement[3]
HPA	seuil	source initiative[2] centralisée[3]	sans filtrage[2] filtrage [1]	centralisée	vecteur L(M,N)	placement[3] migration[3]
pFP-Growth	seuil	source initiative[3]	filtrage[3]	aucune	vecteur L(N,D)	placement[3] migration[3]

TAB. 3.2 – Tableau comparif des approches d'équilibrage de charge dans CCPD, HPA, pFP-Growth et IDD

tionnelles migrent aussi d'un processeur à un autre si leur taille est importante.

Il apparaît, dans le tableau 3.2 résumant l'étude comparative faite dans cette section, que le dispositif d'équilibrage de charge de l'algorithme CCPD est plus facile à implémenter. Il a un nombre de composants réduit et le coût de communication est très faible car les données sont exécutées intégralement sur les nœuds ou elles sont initialement placées qui disposent de toutes les informations nécessaires. Dans HPA et pFP-Growth, l'équilibrage de charge est dynamique. Un tel dispositif est plus complexe à réaliser avec un coût élevé de communication entre les nœuds pour échanger des données ou des informations sur leur état. Le degré de complexité est plus élevé dans HPA que dans pFP-Growth du fait de l'existence dans le premier d'un coordinateur central pour la collecte de l'information et la localisation des nœuds disponibles.

3.5 Conclusion

Après une brève présentation des motivations du processus de datamining et de ses principales applications, nous avons, dans ce chapitre étudié une classe importante de problèmes de datamining à savoir l'extraction de règles associatives. Dans cet étude, nous avons mis l'accent sur l'impléméntation, dans des environnements de traitement distri-

[1]pour les transactions

[2]pour les candidats

[3]pour les transactions et les candidats

bués et paralleles, des concepts developpés autour de ce problème. Ce qui nous a aménés à discuter de plusieurs algorithmes parallèles conçus pour accélérer la recherche des itemsets fréquents. Nous avons aussi discuté des mécanismes d'équilibrage de charge mis en œuvre dans ces algorithmes pour une meilleure répartition de la charge de travail entre les ressources disponibles. Enfin, nous avons fait une analyse qualitative et une étude comparative de ces mécanismes d'équilibrage de charge pour en dégager les forces et les faiblesses. Dans le prochain chapitre, nous proposerons un algorithme parallèle d'extraction des itemsets fermés fréquents sur un cluster de PCs anisi qu'un modèle d'équilibrage de charge associé.

Chapitre 4

Extraction des itemsets fermés fréquents sur un cluster de PCs

Dans ce chapitre, nous proposons un algorithme parallèle d'extraction des itemsets fermés fréquents sur un cluster de PCs fonctionnant sous ParalleKnoppix appelé *EXPERT* ainsi qu'une stratégie d'équilibrage de charge pour celui-ci. Une évaluation des performances de l'algorithme proposé sera ensuite discutés.

4.1 Extraction parallèle des itemsets fermés fréquents

L'algorithme *EXPERT* utilise une structure de données appelée *Itemset-Trie* proposée par Slimani et al [39]. Le *Itemset-Trie* est particulièrement adapté au contexte de datamining interactif dans la mesure sa construction est indépendante du support. De plus, il a l'avantage, par rapport à *FP-Tree* d'offrir un meilleur taux de compression pour les données denses. L'extraction des itemsets fermés fréquents se fait selon l'approche *Diviser-pour-Régner* en utilisant les optimisations apportées à l'algorithme *Closet* [32].

4.1.1 Notions de base

Dans ce qui suit, nous donnons quelques définitions et des résultats provenant de la théorie des concepts formels qui ont été utilisés dans notre algorithme [32, 39].

Definition 1 (Itemset fermés fréquents) : Un itemset X est dit fermé si et seulement si il n'existe aucun itemset Y tel que *support(X)=support(Y)* et $Y \subset X$. Autrement dit un

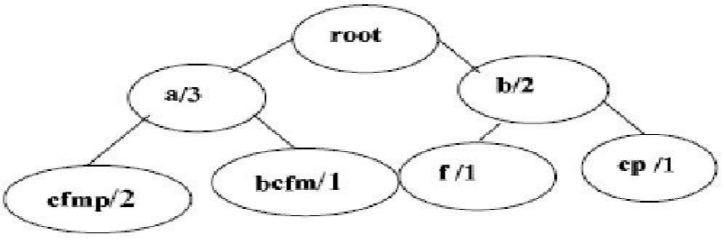
itemset est fermé si tous ses sur-ensembles ont un support inférieur au sien. Les itemets fermés dont le support est supérieur au support minimal sont dits fermés fréquents.

Definition 2 (Trie conditionnel) : Le Trie conditionnel d'un k-itemset Y est l'ensemble des transactions de la base de données contenant Y. De manière formelle, une transaction $t = (TID, X)$ appartient a un Trie conditionnel de Y si et seulement $Y \subseteq X$.

Definition 3 (liste globale et liste conditionnelle) : La liste globale est constituée de l'ensemble des items ayant un support supérieur ou égal au support minimal. La liste conditionnelle d'un k-itemset est constituée des items de son Trie conditionnel ayant un support supérieur ou égal au support minimal. Dans ces listes, les items sont classés par ordre lexicographique et accompagnés chacun de son support.

Propriétés 1 (Fusion d'items [32]) : Soit X un itemset fréquent. Si chaque transaction contenant X contient aussi Y mais pas un sur-ensemble de Y, alors $X \cup Y$ forme un ensemble fermé fréquent et il n'est pas nécessaire de chercher un itemset contenant X et non Y.

Propriétés 2 (Elagage de sous-ensembles d'itemsets [32]) : Soit X un itemset fréquent. Si X est un sous-ensemble d'un itemset fermé fréquent Y déja trouvé et *support(X)=support(Y)*, alors X et tous ses descendants (itemset dont X est le préfixe) ne peuvent pas être des itemsets fermés fréquents et doivent être omis de l'arbre d'énumération.



TID	items
1	acfmp
2	abcfm
3	bf
4	bcp
5	acfmp

FIG. 4.1 – Exemple d'un contexte d'extraction avec l'itemset-Trie associé [39]

4.1.2 L'algorithme EXPERT (EXtraction Parallèle d'itEmsets feRmés fréquenTs)

Dans l'algorithme *EXPERT* les processeurs utilisent un itemset-Trie commun qui contient toutes les transactions de la base de données. Ainsi, il n'est pas nécessaire que les processeurs communiquent pour échanger des informations concernant un item de la base. Puisque les listes peuvent être traitées de manière indépendante, une fois que le Trie est construit, la liste globale sera partitionnée et chaque processeur traitera la liste qui lui est affectée. Considérons le contexte d'extraction a droite de la figure 4.1. Chaque processeur construit à partir de ce contexte un itemset-Trie représenté à gauche de la figure 4.1. Si nous disposons de deux nœuds, l'extraction des itemsets fermés fréquents avec *minsup=1* se fait de la manière suivante sur le nœud 1 : Après la construction du Trie, la liste des 1-itemsets obtenue est $L = \langle a/3; b/3; c/4; f/4; m/3; p/3\rangle$. Le partitionnement de cette liste selon l'algorithme Round Robin donne une nouvelle liste $L_1 = \langle a/3; c/4; m/3\rangle$ qui sera traitée au niveau du nœud 1. En appliquant le paradigme Diviser-pour-Régner, nous cherchons d'abord tous les itemsets fermés fréquents contenant a puis ceux contenant c mais ne contenant ni a ni b et enfin ceux contenant m mais ne contenant ni a, ni b, ni f.

1. Recherche des itemsets fermés fréquents contenant a :
   Elle commence par la construction du Trie conditionnel de a et de sa liste conditionnelle $L_a = \langle b/1; c/3; f/3; m/3; p/2\rangle$. nous constatons que c, f et m ont le même support que a. D'aprés le principe de la fusion d'items $acfm/3$ constitue un itemset fermé fréquent et les autres itemsets fermés fréquents de L_a seront des surensembles de celui-ci. Les items c, f et m sont ensuite supprimés de L_a et puisque celle-ci n'est pas vide on continue en construisant respectivement le Trie conditionnel de l'itemset $acfmp$ et celui de $acfmb$ ainsi que les listes conditionnelles associées. Finalement nous obtenons $L_{acfmp} = \langle\emptyset\rangle$, $L_{acfmb} = \langle\emptyset\rangle$ d'ou l'on extrait les itemsets fermés fréquents $acfmp/2$ et $acfmb/1$.

2. Recherche des itemsets fermés fréquents contenant c mais ne contenant ni a, ni b :
   On constate que dans la liste conditionnelle de $L_c = \langle a/3; b/2; f/3; m/3; p/3\rangle$ aucun item[4] n'a un support égal à celui de c d'ou $c/3$ est un itemset fermé fréquent. Puisque L_c n'est pas vide, nous continuons avec les listes L_{cp}, L_{cm} et L_{cf}. Dans

[4]Il s'agit des items qui viennent après c dans l'ordre lexicographique. Les items qui viennent avant, bien que présents dans la liste, ne servent que dans l'élagage de sous ensembles d'itemsets.

$L_{cf} = \langle a/3; b/1; m/3; p/2 \rangle$, cf et m ont le même support donc $cfm/3$ est un itemset fermé fréquent potentiel, mais puisque dans L_{cf} il y a aussi a qui a le même support que cfm, ce dernier sera omis en vertu du principe de l'élagage[5] de sous-ensembles d'itemsets.L'item m est supprimée de L_{cf} et nous continuons avec $L_{cfp} = \langle a/2 \rangle$; là aussi cfp et a ont le même support et donc cfp sera omis. Le traitement de la liste L_{cm} ne donnera aucun itemset fermé fréquent tandis que celui de la liste L_{cp} donnera $cp/3$.

3. Recherche des itemsets fermés fréquents contenant m mais ne contenant ni a, ni b, ni f :
   Dans $L_m = \langle a/3; b/1; c/3; f/3; p/2 \rangle$ aucun des items qui vient après m, dans l'ordre lexicographique, n'a le même support ce qui en fait un itemset fermé fréquent potentiel, mais comme les items a, b, c et f ont le même support que m, il sera élagué. Nous continuons avec la liste $L_{mp} = \langle a/2; c/2; f/2 \rangle$ qui ne donnera aucun itemset fermé fréquent.

En résumé, chaque processeur effectuera les tâches suivantes :

1. Construction du Itemset-Trie :
   Chaque processeur scanne la base de données et insère les transactions dans son Trie qui est commun à tous les processeurs.
2. Construction de la liste globale :
   Chaque processeur parcourt l' itemset-Trie qu'il a construit et insére tous les items dans une liste et calcule leur support. Les items de cette liste qui ne vérifient pas le support minimal, seront supprimés pour obtenir la liste globale.
3. Determination de la charge initiale de travail :
   Chaque processeur supprime de la liste globale les items qu'il n'aura pas à traier. Ces items sont connus en utilisant l'algorithme *Round Robin*.
4. Calcul des itemsets fermés fréquents :
   Chaque processeur fait pour chaque item de sa liste le travail suivant :
   (a) Construction du itemset-Trie conditionnel de l'item.
   (b) Construction de la liste conditionnel de l'item.

[5] comme a et cfm ont le même support, cfm est contenu dans $acfm$ et puisque que nous utilisons l'approche Diviser-pour-Régner, tous les itemsets fermés fréquents contenant a sont déja trouvés ; donc cfm n'est pas un itemset fermé fréquent

(c) Calcul des itemsets fermés fréquents de la liste conditionnelle de l'item :
Ce calcul se fait selon le paradigme *Diviser-pour-Régner* en utilisant le principe de la *fusion des items* et *l'élagage de sous-ensembles d'itemsets*.

(d) Envoi des resultats au processeur maitre chargé de les collecter.

4.1.3 Stratégie d'équilibrage de charge

La première stratégie d'équilibrage de charge consiste à allouer les items de la liste globale aux nœuds du cluster selon l'algorithme Round Robin. De cette manière, les nœuds auront initialement à traiter le même nombre d'items. Cependant avec cette stratégie, nous constatons un déséquilibre du fait que les items de la liste globale n'ont pas le même coût de traitement[6]. Il est donc nécessaire de réajuster la charge des nœuds durant la phase d'exécution en fonction des informations collectées sur l'état du système. Pour ce faire, nous proposons un algorithme d'équilibrage de charge dynamique et centralisé basé sur l'approche receveur-initiative. Dans cet algorithme, lorsqu'un nœud a fini de traiter la liste qui lui était initialement allouée, il s'adresse au coordinateur central qui envoie des requêtes de transfert de charge aux nœuds du cluster qui n'ont pas encore épuisé leur charge initiale de travail. Les nœuds contactés par le coordinateur central envoient alors une partie du travail qui leur reste à effectuer au receveur. Il faut souligner que pour éviter un coût élevé des communications, une fois qu'un item a migré vers une destination, il n'est plus transférable.

4.2 Evaluation expérimentale

Dans cette section, nous analysons les performances de l'algorithme EXPERT. Comme critères de performance, nous avons retenu la scalabilté et l'efficacié de l'équilibrage de charge.

4.2.1 Environnement expérimental

Configuration matérielle : Pour les besoins de notre expérimentation, nous avons construit un Cluster Linux de PC's fonctionnant sous *ParallelKnoppix* [40, 41] dont le nombre de

[6]Les items dont les listes conditionnelles possèdent plus d'items vérifiant le support minimal ont un coût de traitement plus élevé.

nœuds varie de 1 à 4. Les nœuds sont interconnectés par des cartes réseaux Ethernet 100MBits et ils sont chacun équipé d'un processeur Pentium IV ayant une fréquence d'horloge de 2500 MHZ avec 512 Mo de mémoire RAM. Pour la parallélisation des processus, nous avons utilisé l'implémentation LAM [42, 43] de la bibliothèque MPI [44, 45].

Bases de données : Nous avons testé notre algorithme sur deux types de bases de données à savoir des bases denses (*Mushroom* et *Chess*) et des bases éparses (*T10I4D100k* et *T40I10D100k*). Ces bases de données sont disponibles sur Internet [46] et leurs caracteristiques sont données dans le tableau 4.1. La base *Mushroom* contient les caracteristiques de diverses espèces de champignons. La base *Chess* est dérivée des étapes du jeu d'echec. *T10I4D100k* et *T40I10D100k* sont des bases de données synthétiques qui immitent les transactions dans un environnement de vente au detail.

Base	Type	# de transactions	# d'items	Taille moyenne des transactions
mushroom	Dense	8124	120	23
Chess	Dense	3196	76	37
T10I4D100k	Eparse	100000	1000	10
T40I10D100k	Eparse	100000	1000	40

TAB. 4.1 – Caracteristiques des bases de test

4.2.2 Expérimentations

Nous avons effectué un certain nombre d'expérimentations en tenant compte des paramètres suivants :

1. Le type et la taille de la base de transactions : nous avons considéré des bases denses et des bases éparses.
2. La valeur du support minimal : selon le type de base nous avons défini 3 valeurs de support minimal.
3. Le nombre de processeurs : nous avons fait varier de 1 à 4 le nombre de processeurs de notre plateforme de test.

Les tableaux et les figures suivants résument l'ensemble de nos résultats.

4.2.2.1 Expérimentation 1

Base : T10I4D100k

Type : éparse

Supports : 0.03, 0.05, 1

Nombre de processeurs : 1...4

Dans cette expérimentation, nous examinons le temps d'extraction des itemsets fermés fréquents sur la base T10I4D100k en augmentant le nombre de processeurs et la valeur du support. Les résultats obtenus sont consignés dans le tableau 4.2.

Interprétation : A la lecture du tableau 4.2 et de la Figure 4.2(a), nous pouvons noter les remarques suivantes :

- Plus le support augmente plus le temps d'extraction diminue. En particulier, quand le support a été multiplié par 2 (passage de 0.05 à 1) nous avons relevé que le temps d'extraction a été pratiquement divisé par 5 et ce quelque soit le nombre de processeurs. Ceci s'explique par le fait qu'il y a moins d'itemsets fermés fréquents à générer (pour une valeur de minsup égale à 1 nous avons 385 itemsets fermés fréquents alors que pour une valeur de minsup égale à 0.05, nous en avons 46993).
- L'augmentation du nombre de processeurs a pour effet de réduire le temps d'extraction pour toutes les valeurs de support. Le passage de 1 à 3 processeurs a notamment réduit le temps séquentiel de moitié.

# de processeurs	Support (%)		
	0.03	0.05	1.00
1	763.77	595.11	123.65
2	546.88	426.33	97.84
3	374.19	281.18	65.48
4	292.34	243.50	49.02

TAB. 4.2 – Temps d'extraction (sec) des itemsets fermés fréquents pour la base T10I4D100k en fonction du nombre de processeurs et de la valeur du support.

4.2.2.2 Expérimentation 2

Base : T40I10D100k

Type : éparse

Supports : 5, 10, 15

Nombre de processeurs : 1...4

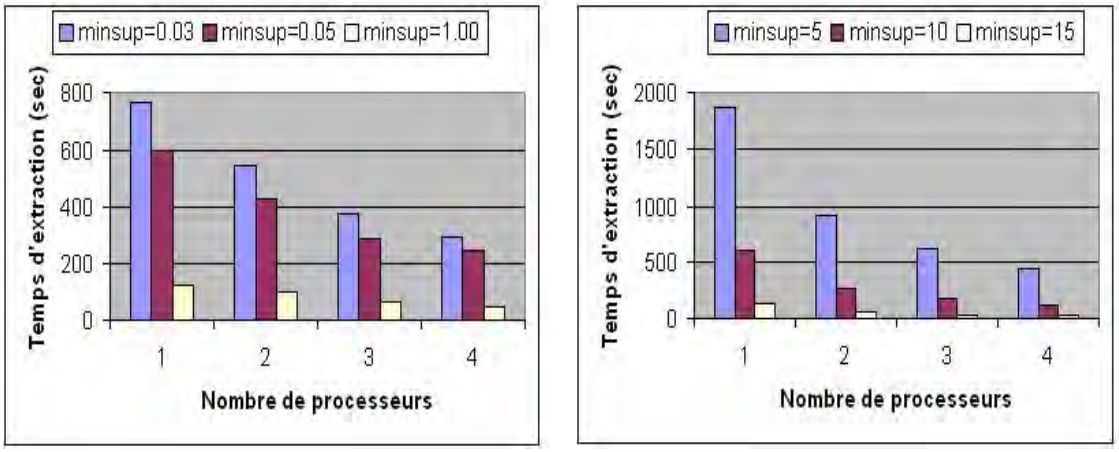
Dans cette expérimentation, nous examinons le temps d'extraction des itemsets fermés fréquents sur la base T40I10D100k en augmentant le nombre de processeurs et la valeur du support. Les résultats obtenus sont consignés dans le tableau 4.3.

Interprétation : A la lecture du tableau 4.3 et de la Figure 4.2(b), nous pouvons noter les remarques suivantes :

- Plus le support augmente plus le temps d'extraction diminue. En particulier, quand le support a été multiplié par 3 (passage de 5 à 15) nous avons constaté que le temps d'extraction est au moins divisé par 14 et ce quelque soit le nombre de processeurs du fait qu'il y a moins d'itemsets fermés fréquents à générer (19 pour une valeur de minsup égale à 15 contre 316 pour une valeur de minsup égale à 5).
- L'augmentation du nombre de processeurs permet de réduire le temps d'extraction pour toutes les valeurs du support. Le passage de 1 à 4 processeurs divise pratiquement par 4 le temps séquentiel.

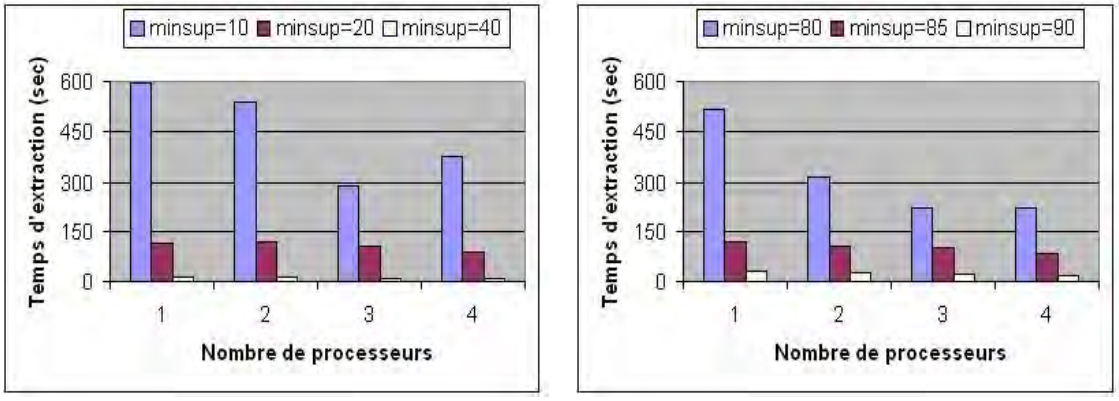
# de processeurs	Support (%)		
	5	10	15
1	1861.46	598.68	130.40
2	923.56	269.41	52.87
3	617.20	168.46	31.62
4	439.29	109.91	22.51

TAB. 4.3 – Temps d'extraction (sec) des itemsets fermés fréquents pour la base T40I10D100k en fonction du nombre de processeurs et de la valeur du support.



(a) T10I4D100k (b) T40I10D100k

FIG. 4.2 – Temps d'extraction des itemsets fermés fréquents pour la base T10I4D100k et la base T40I10D100k en fonction du support et du nombre de processeurs.



(a) Mushroom (b) Chess

FIG. 4.3 – Temps d'extraction des itemsets fermés fréquents pour la base Mushroom et la base Chess en fonction du support et du nombre de processeurs.

4.2.2.3 Expérimentation 3

Base : Chess

Type : dense

Supports : 80, 85, 90

Nombre de processeurs : 1...4

Dans cette expérimentation, nous examinons le temps d'extraction des itemsets fermés fréquents sur la base Chess en augmentant le nombre de processeurs et la valeur du support. Les résultats obtenus sont consignés dans le tableau 4.4.

Interprétation : A la lecture du tableau 4.4 et de la Figure 4.3(b), nous pouvons noter les remarques suivantes :

- Plus le support augmente plus le temps d'extraction diminue. En particulier, lorsque le support passe de 80 à 85 le temps d'extraction est divisé par 4 et ce quelque soit le nombre de processeurs.
- L'augmentation du nombre de processeurs entraîne une baisse du temps d'extraction pour toutes les valeurs du support. Nous constatons ainsi que lorsque le nombre d'itemsets fermés fréquents générés est important (5083 pour une valeur de minsup égale à 80), le passage de 1 à 3 processeurs réduit le temps d'extraction sequentiel de moitié.

# de processeurs	Support (%)		
	80	85	90
1	518.93	116.73	29.41
2	317.37	103.58	25.84
3	222.96	102.16	22.27
4	216.94	81.73	16.64

TAB. 4.4 – Temps d'extraction (sec) des itemsets fermés fréquents pour la base Chess en fonction du nombre de processeurs et de la valeur du support.

4.2.2.4 Expérimentation 4

Base : Mushroom

Type : dense

Supports : 10, 20, 40

Nombre de processeurs : 1...4

Dans cette expérimentation, nous examinons le temps d'extraction des itemsets fermés fréquents sur la base Mushroom en augmentant le nombre de processeurs et la valeur du support. Les résultats obtenus sont consignés dans le tableau 4.5.

Interprétation : A la lecture du tableau 4.5 et de la Figure 4.3(a), nous pouvons noter les remarques suivantes :

- Plus le support augmente plus le temps d'extraction diminue et ce quelque soit le nombre de processeurs. Lorsque la valeur du support passe de 10 à 20, le temps d'extraction est divisé par 9 sur 1 et 2 processeurs et il est divisé par 12 sur 3 et 4 processeurs.
- L'augmentation du nombre de processeurs entraîne une baisse du temps d'extraction pour une valeur minsup égale à 20 et ce quelque soit le nombre de processeurs. Pour une valeur minsup égale à 10, nous constatons que le passage de 1 à 3 processeurs fait baisser le temps d'extraction de moitié, mais nous relevons une hausse de 24% du temps d'extraction lors du passage de 3 à 4 processeurs. ceci s'explique par le fait qu'avec 4 processeurs il y a un important déséquilibre de charge. Pour une valeur minsup égale à 40, nous constatons que le passage de 1 à 2 processeurs entraine une hausse de 7% du temps d'extraction séquentiel à cause du déséquilibre de charge et de l'excès[7] de travail (overhead) du programme parallèle.

# de processeurs	Support (%)		
	10	20	40
1	595.68	112.35	11.85
2	539.35	116.95	12.75
3	289.53	106.08	8.80
4	378.72	89.52	6.64

TAB. 4.5 – Temps d'extraction (sec) des itemsets fermés fréquents pour la base Mushroom en fonction du nombre de processeurs et de la valeur du support.

[7] cet excès est essentiellemnt dû au temps de communication inter-processus et au temps de synchronisation

4.3 Conclusion

Dans ce chapitre, nous avons proposé un algorithme parallèle d'extraction d'itemsets fermés fréquents appelé EXPERT et une technique d'équilibrage de charge pour celui-ci. Une évaluation des performances de cet algorithme a été ensuite faite sur un cluster de PCs fonctionnant sous Linux Parallelknoppix. Les resultats des expérimentations montrent que :

1. Plus la valeur du support augmente plus le temps d'extraction diminue et ce quelque soient le type de la base de transactions et sa taille ou le nombre de processeurs utilisés.
2. L'augmentation du nombre de processeurs entraîne une baisse du temps d'extraction pour toutes les valeurs du support et ce quelque soient le type et la taille de la base de transactions. En particulier, pour les bases éparses où le temps d'extraction est divisé par 2 pour la base T10I4D100k et pratiquement par 5 pour la base T40I10D100k lorsque le nombre de processeurs passe de 1 à 4.

Du point de vue équilibrage de charge, nous constatons un déséquilibre entre processeurs pour l'extraction des itemsets fermés fréquents. Ce déséquilibre est plus important chez les bases de transactions denses.

Conclusion

Dans ce mémoire subdivisé en quatre chapitres, nous avons donné dans le premier chapitre un aperçu sur le calcul haute performance en présentant une taxonomie des architectures parallèles et distribués ainsi que les modèles de programmation parallèle les plus connus.

Dans le deuxième chapitre, nous avons étudié le problème de l'équilibrage de charge dans les systèmes parallèles et distribués en donnant une classification des approches d'équilibrage de charge. Nous avons aussi présenté les différentes politiques d'un système d'équilibrage de charge et passé en revue un certain nombre d'algorithmes d'équilibrage de charge existant dans la littérature.

Dans le troisième chapitre, nous avons etudié l'équilibrage de charge dans les algorithmes d'extraction de règles associatives. Ce qui nous amené à discuter de plusieurs algorithmes de recherche d'itemsets fréquents et des mécanismes d'équilibrage de charge qu'ils implémentent. Pour terminer, nous avons fait une analyse qualitative et une comparaison des différentes approches d'équilibrage de charge dans les algorithmes étudiés.

Dans le dernier chapitre, nous avons proposé un algorithme parallèle d'extraction d'itemsets fermés fréquents et une technique d'équilibrage de charge pour ce dernier. Nous avons procédé à une évaluation des performances de l'algorithme sur un cluster de PCs fonctionnant sous Parallelknoppix avec des bases de données denses et éparses en faisant varier le nombre de processeurs et la valeur du support. Les résultats obtenus montrent que l'augmentation du nombre de processeurs fait baisser le temps d'extraction dans toutes les bases de données et ce quelque soit la valeur du support. Cependant, nous avons observé un déséquilibre de charge entre processeurs surtout lorsque la base de transactions est de type dense. La technique d'équilibrage de charge utilisée bien que réduisant le déséquilibre engendre un important coût de communication qui s'explique par la durée parfois assez longue qui sépare la requête et la reception de la charge de travail

durant la phase d'équilibrage.

Dans nos futurs travaux, nous essayerons de réduire le coût de communication de l'algorithme d'équilibrage de charge en redéfinissant une granularité d'exécution plus fine que l'item. En effet, nous avons constaté que parfois le déséquilibre etait uniquement dû à un seul item qui, du fait de la nature récursive de l'algorithme d'extraction des itemsets fermés fréquents et des propriétés des items contenus sa liste conditionnelle générait un nombre important de resultats. Nous pensons que si le traitement des items necessitant beaucoup de ressources de calcul se faisait en parallèle sur plusieurs processeurs, le temps d'extraction en serait réduit. Comme autre perspective, nous envisageons aussi d'expérimenter notre algorithme sur d'autres types de plateformes, notamment des multiprocesseurs et des machines massivement parallèles.

Bibliographie

[1] Site MHPCC, *www.mhpcc.edu/training/workshop/parallel_intro/MAIN.html*, October 2005.

[2] M.J. Flynn, "Some Computer organizations and their effectiveness", *IEEE Transactions on Computers*, 21 :948-960, 1972.

[3] A. Tanenbaum, "Modern Operating Systems", *2nd Edition Prentice Hall*, 2001.

[4] A. Tanenbaum and van Steen : "Distributed Systems : Principles and Paradigms", *2nd Prentice-Hall, Inc*, 2002.

[5] A. Geist, A. Beguelin, J. Dongara, W. Jiang, B. Manchek, and V. Sunderam, "PVM : Parallel Virtual Machine A User's Guide and Tutorial for Network Parallel Computing", *MIT Press*, Cambridge, Mass, 1994.

[6] J. Chergui, I. Dupays, D. Girou, N. Grima, and S. Requena, "Message Passing Interface (MPI-1)", *http ://webserv2.idris.fr/data/cours/parallel/mpi*, October 2005.

[7] J. Chergui, and P.F. Lavallee, "OpenMP : Parallélisation Multitâches pour Machines à Mémoire Partagée", *http ://webserv2.idris.fr/data/cours/parallel/openmp*, October 2005

[8] High Performance Fortran Forum, "High Performance Fortran Language Specification Version 2.0", January 1997.

[9] T.L. Casavant, and J.G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems", *IEEE Transactions on Software Engineering*,14(2) :141–154, February 1988.

[10] A. Weinrib, and S. Shenker, "Greed is not enough : Adaptive load sharing in large heterogeneous systems", *Proceedings of the IEEE Infocom*, pages 986-994, March 1988.

[11] J. Ghanem, "Implementation of Load Balancing Policies in Distributed Systems" *Master's thesis*, Department of Electrical and Computer Engineering, The University of New Mexico, June 2004.

[12] M. H. Willebeek-LeMair, and A. P. Reeves, "Strategies for Dynamic Load Balancing on Highly Parallel Computers", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, NO. 9, pages 979-993, September 1993.

[13] A. Corradi, L. Leonardi, and F. Zambonelli, "On the Effectiveness of Different Diffusive Load Balancing Policies for Dynamic Applications", *HPCN Europe*, pages 274-283, Amsterdam, 1998.

[14] V. Kumar, A.Y. Grama, and V.N. Rao, "Scalable Load Balancing Techniques for Parallel Computers", *Technical Report 91-55*, Computer Science Department, University of Minnesota, 1991

[15] J. Baumgartner, D.J. Cook, and B. Shirazi, "Genetic Solutions to the Load Balancing Problem", *Proceedings of the International Conference on Parallel Processing*, Raleigh, North Carolina, 1995.

[16] S. Zhou, "A trace-driven simulation study of dynamic load balancing", *Technical report UCB/CSD 87/305*, University of California, Berkeley, 1986.

[17] M. H. Zaharia, F. Lorin, and D. Galea, "Load Balancing in Distributed Systems using Cognitive Behavioral Models", *Bulletin of Technical University of Iasi*, Romania, Tome XLVIII (LII), fasc. 1-4, 2002.

[18] G Piatetsky-shapiro, and W.J. Frawley, "Knowledge Discovery in Databases", *AAAI Press*, pages 229-248, 1991.

[19] R. Agrawal, T. Imielinski, and A. Swami, "Database Mining : A performance perspective", *IEEE Transactions on Knowledge and Data Engineering*, 5(6) :914-925, December 1993.

[20] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases", *Proceedings of 1993 ACM-SIGMOD Conference*, Washington, D.C, May 1993.

[21] R. Agrawal, and R. Srikant, "Fast Discovery of Association Rules", *Proceedings of 19th VLDB Conference*, Santiago, Chili, 1994

[22] R. Agrawal and J. Shafer, "Parallel Mining of Association Rules", *IEEE Trans. Knowledge and Data Eng*, Vol. 8, No. 6, Dec.1996, pp. 962-969.

[23] M.J. Zaki, "Scalable data mining for rules", *PhD Thesis*, University of Rochester, New York, 1998.

[24] M.J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li, "Parallel Data Mining for Association Rules on Shared-Memory Multi- Processors", *in Knowledge and Information Systems*, Volume 3, Number 1, pp 1-29, February 2001.

[25] M.J. Zaki, "Parallel and Distributed Association Mining : A survey", *IEEE Concurrency*, special issue on Parallel Mechanisms for Data Mining, Vol. 7, No. 4, pp14-25, December 1999.

[26] M.J. Zaki and K. Gouda, "Efficiently mining maximal frequent itemsets", *First IEEE International Conference on Data Mining* , San Jose, November 2001.

[27] M.J. Zaki and C.J. Hsiao, "CHARM : An efficient algorithm for closed itemset mining", *2nd SIAM International Conference on Data Mining*, April 2002.

[28] O. R. Zaïane, and Mohammad El-Hajj ,"Tutorial : Advances and Issues in Frequent Pattern Mining", *PAKDD*, Sydney 2004.

[29] V. Kumar, G. Karypis, and E.H. Han, "Scalable Parallel Data Mining for Association Rules", *IEEE Transactions On Knowledge and Data Engineering*, Vol. 12, No. 3, May/June 2000.

[30] V. Kumar, M. Joshi, and E.H. Han, "Efficient parallel algorithms for mining associations" *Parallel and Distributed Systems*, 1759 :418-429, 2000.

[31] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation", *Proc. 2000 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'00)*, Dallas, Texas, May 2000.

[32] J. Han, J. Wang, and J. Pei, "Closet+ : Searching for the best strategies for mining frequent closed itemsets", *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03)*, Washington, DC, USA, 2003.

[33] T. Shintani, and M. Kitsuregawa, "Hash Based Parallel Algorithms for Mining Association Rules", *Proceedings of the Conference on Parallel and Distributed Information Systems, IEEE Computer Soc. Press*, Los Alamitos, California, 1996.

[34] T. Shintani, and M. Kitsuregawa, "Dynamic Load Balancing for Parallel Association Rule mining on Heteregeneous PC Cluster System", *Proceedings of 25th VLDB Conference*, Edingburg, Scotland, 1999.

[35] P. Iko, M. Kitsuregawa, "Parallel FP-Growth on PC cluster", *Proceedings of Seventh Pacific-Asia Conference of Knowledge Discovery and Data Mining (PAKDD03)*, Seoul, Korea, 2003.

[36] R. J. Bayardo "Efficiently mining long patterns from databases", *Proceedings of the fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Diego, California, June 1998.

[37] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules", *International Conference on Database Theory (ICDT)*, pages pp 398-416, Jerusalem, Israel, January 1999.

[38] D. Burdick, M. Calimlim, and J. Gehrke, "Mafia : A maximal frequent itemset algorithm for transactional databases", *In Proceedings of the 17th International Conference on Data Engineering*, Heidelberg, Germany, April 2001.

[39] Y. Slimani, S. Ben Yahya, and J. Rezgui, "A Divide and Conquer Approach for deriving Partially Ordered Sub-structures", *PAKDD'2005*, Hanoi, Vietnam, The Ninth Pacific-Asia, 18-20 May 2005.2005.

[40] ParallelKnoppix website, *http ://pareto.uab.es/mcreel/ParallelKnoppix*, June 2006.

[41] M. Creel, "ParallelKnoppix Tutorial", *http ://pareto.uab.es/wp/2004/62604*, October 2005.

[42] The LAM/MPI Team website, *www.lam-mpi.org*, June 2006.

[43] The LAM/MPI Team, "LAM/MPI User's Guide", *Open Systems Laborator, Pervasive Technology Labs*, Indiana University, Bloomington, 7.1.1 edition, September 2004.

[44] Message Passing Interface Forum (1997), "MPI-2 : Extensions to the Message-Passing Interface", University of Tennessee, Knoxville, Tennessee.

[45] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, "MPI : The Complete Reference", *MIT Press*, Cambridge, MA, 1996.

[46] *http ://fimi.cs.helsinki.fi/data*, June 2006.