PGP-mc : extraction parallèle efficace de motifs graduels

Anne Laurent, Benjamin Negrevergne, Nicolas Sicard, Alexandre Termier

PGP-mc : extraction parallèle efficace de motifs graduels

Anne Laurent*, Benjamin Negrevergne***, Nicolas Sicard**, Alexandre Termier***

*LIRMM - UM2- CNRS UMR 5506 - 161 rue Ada - 34392 Montpellier Cedex 5
laurent@lirmm.fr
http://www.lirmm.fr
**LRIE - EFREI - 30-32 av. de la République, 94 800 Villejuif
nicolas.sicard@efrei.fr
http://www.efrei.fr
***LIG - UJF-CNRS UMR 5217 - 681 rue de la Passerelle, B.P. 72, 38402 Saint Martin d'Hères
{benjamin.Negrevergne@imag.fr,Alexandre.Termier@imag.fr}
http://www.liglab.fr

Résumé. Initialement utilisés pour les systèmes de commande, les règles et motifs graduels (de la forme "plus une personne est âgée, plus son salaire est élevé") trouvent de très nombreuses applications, par exemple dans les domaines de la biologie, des données en flots (e.g. issues de réseaux de capteurs), *etc.* Très récemment, des algorithmes ont été proposés pour extraire automatiquement de tels motifs. Cependant, même si certains d'entre eux ont permis des gains de performance importants, les algorithmes restent coûteux et ne permettent pas de traiter efficacement les bases de données réelles souvent très volumineuses (en nombre de lignes et/ou nombre d'attributs). Nous proposons donc dans cet article une méthode originale de recherche de ces motifs utilisant le *multi-threading* pour exploiter au mieux les multiples coeurs présents dans la plupart des ordinateurs et serveurs actuels. L'efficacité de cette approche est validée par une étude expérimentale.

1 Introduction

En fouille de données, la recherche de motifs fréquents est un sujet de recherche très actif. Initialement étudiés dans le cas de données transactionnelles, les algorithmes ont vite été étendus aux structures de données complexes (séquences, arbres, graphes etc.) Han et Kamber (2006). De manière générale, tous les algorithmes doivent explorer un espace de recherche très important. Ils sont donc très coûteux en temps de calcul, la complexité des calculs augmentant avec celle des structures de données à traiter. Une grande partie des travaux a été menée pour mettre au point des algorithmes de plus en plus efficaces permettant de fouiller de très grandes bases de données aux structures potentiellement complexes.

Récemment, un nouveau type de règles et motifs a été introduit : les *motifs graduels* (ou *itemsets graduels*). Ce problème vise à découvrir, à partir de bases de données numériques, des motifs du type "plus un individu est âgé, plus son salaire est élevé" et trouve de très nombreuses applications pour les bases de données numériques comme par exemple les données

biologiques et médicales. Dans ce contexte, l'algorithme GRITE proposé dans Di Jorio et al. (2009) est à notre connaissance le plus efficace en termes de temps de calcul et occupation mémoire. Développé à partir d'une approche par niveaux Agrawal et Srikant (1994), il permet de fouiller des bases de données comprenant jusqu'à plusieurs centaines d'attributs ou plusieurs milliers de lignes, quand les approches précédentes étaient limitées à six attributs Berzal et al. (2007). Cependant, ces performances restent faibles au regard des bases de données réelles (comprenant des milliers d'attributs et/ou des millions de lignes), sur lesquelles l'algorithme GRITE peut s'avérer très long, voire impossible à exécuter. Face à ce défi, deux solutions (non exclusives) sont envisageables à l'échelle de GRITE afin de l'appliquer aux données réelles : (i) des améliorations algorithmiques, par exemple en utilisant les techniques *pattern growth* Han et al. (2000) et en définissant la notion de *fermeture* sur les motifs graduels Pasquier et al. (1999); Uno (2005), ou (ii) en explorant les possibilités de parallélisation.

Dans cet article, nous explorons la seconde solution en exploitant le parallélisme sur les processeurs multi-cœurs. Au cours des 20 dernières années l'augmentation de la performance des processeurs passait en effet essentiellement par une augmentation de la fréquence d'horloge. Or, depuis 2005, les limites physiques empêchent d'améliorer cette fréquence. Cependant, il est possible d'intégrer de plus en plus de transistors, qui permettent de multiplier les cœurs de calcul dans un seul processeur. L'exploitation optimale de ces multiples cœurs requiert l'écriture de programmes parallèles, ce type d'architecture ayant des propriétés propres, différentes par exemple des clusters de machines. Les processeurs multi-cœurs ont souvent une architecture dite UMA (Uniform Memory Access) : la machine est dotée d'un bloc mémoire et tous les cœurs ont un temps d'accès équivalent à cette mémoire, à travers un bus unique, qui fait de l'usage de la mémoire le goulot d'étranglement classique des applications parallèles. Ces denières années, des recherches ont été menées en fouille de données parallèle pour exploiter ces architectures multi-cœurs Buehrer et al. (2006); Lucchese et al. (2007); Liu et al. (2007); Tatikonda et Parthasarathy (2009), montrant que plus le problème de fouille était complexe (e.g. arbres, graphes), plus la parallélisation améliorait les performances. La principale raison est que les accès mémoires sont relativement faibles par rapport aux calculs coûteux à effectuer sur les données chargées dans les caches. L'application de la parallélisation à la recherche de motifs graduels, qui reste une technique coûteuse, est donc prometteuse. Nous l'étudions ci-après et nous le validons au travers des expérimentations menées.

La suite de cet article est organisé de la manière suivante : la section 2, introduit la notion de motif graduel, tandis que la section 3 rappelle les principales définitions de ces motifs présents dans la littérature ainsi que les principales approches de la fouille de données parallèle. La section 4 présente notre algorithme parallèle d'extraction de motifs graduels, dont les résultats expérimentaux sont présentés en section 5. Enfin, la section 6 conclut et présente les principales perspectives associées à ce travail.

2 Motifs graduels

Les motifs graduels sont de la forme *"plus/moins* X_1*, ..., plus/moins* X_n*"*. On considère ici une base de données DB consistant en une relation définie sur l'ensemble d'attributs $\mathcal{I}$. Dans ce contexte, les motifs graduels sont définis sur un sous-ensemble de $\mathcal{I}$ dont les éléments sont associés à un ordre croissant ou décroissant. On note $t[I]$ la valeur de t sur l'attribut I.

Id	Size (S)	Weight (W)	Sugar Rate (SR)
t_1	6	6	5.3
t_2	10	12	5.1
t_3	14	4	4.9
t_4	23	10	4.9
t_5	6	8	5.0
t_6	14	9	4.9
t_7	18	9	5.2
t_8	23	10	5.3
t_9	28	13	5.5

FIG. 1 – *Fruits et leurs caractéristiques*

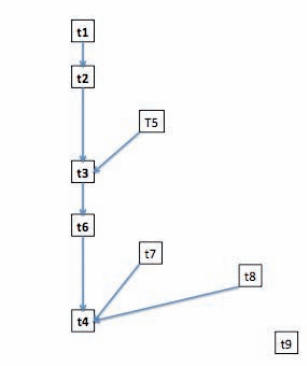


FIG. 2 – $g = \{(S, \geq), (SR, \leq)\}$

Par exemple, on considère la base de données de la figure 1 décrivant des fruits et leurs caractéristiques.

Définition 1 *(Item graduel) Soit DB une base de données définie sur les attributs $\mathcal{I}$, un motif graduel est un couple (I, θ) où I est un attribut de $\mathcal{I}$ et θ est un opérateur de comparison operator de $\{\geq, \leq\}$.*

Un itemset graduel ou motif graduel $g = \{(I_1, \theta_1), ..., (I_k, \theta_k)\}$ est un ensemble d'items graduels.

Par exemple, $(Size, \geq)$ est un item graduel, tandis que $\{(Size, \geq), (Weight, \leq)\}$ est un itemset graduel. Le support d'un itemset graduel dans une base de données DB revient à calculer à quel point le motif est présent dans DB. Plusieurs définitions formelles de cette notion de support ont été proposées (voir la section 3 ci-dessous). Dans cet article, on considère le nombre de n-uplets qu'il est possible d'ordonner pour respecter les opérateurs de comparaison sur chacun des attributs présents dans le motif considéré.

Définition 2 *(Support d'un itemset graduel) Soit DB une base de données et $g = \{(I_1, \theta_1), ..., (I_k, \theta_k)\}$ un itemset graduel. La cardinalité de g dans DB, notée $\lambda(g, DB)$, correspond à la longueur de la plus longue liste $l = \langle t_1, \ldots, t_n \rangle$ de n-uplets de DB tels que, pour tout $p = 1, \ldots, n-1$ et pour tout $j = 1, \ldots, k$, la comparaison $t_p[I_j]\, \theta_j\, t_{p+1}[I_j]$ est valide.*

Le support de g dans DB, noté $supp(g, DB)$, correspond au ratio de $\lambda(g, DB)$ par rapport à la cardinalité de DB, que nous notons $|DB|$. On a donc : $supp(q, DB) = \frac{\lambda(g, DB)}{|DB|}$.

Afin de calculer $\lambda(g, DB)$, Di Jorio et al. (2009) propose de considérer le graphe dans lequel les nœuds correspondent aux n-uplets de DB et dans lequel il existe un arc entre deux nœuds si les n-uplets associés sont ordonnés par rapport à g. Par exemple, la figure 2 montre l'ordonnancement des n-uplets de la base de l'exemple précédent (Fig. 1) pour l'itemset graduel $g = \{(S, \geq), (SR, \leq)\}$ (*plus la taille est grande, moins le taux de sucre est important*).

La longueur du plus long chemin de ce graphe est 5. Comme DB contient 9 n-uplets, on a donc $supp(g, DB) = \frac{5}{9}$.

L'algorithme est basé sur le principe APriori, en notant que les items ne sont pas explorés dans la mesure où toute base de données peut être ordonnée pour trier sur un unique attribut. L'algorithme débute donc en étudiant le support de toutes les paires d'items graduels et en passant ensuite d'un niveau k au suivant en constituant des candidats de niveau k en fonction des fréquents de taille $k-1$ puis en testant leur support. Le stockage de tous les ordonnancements à un niveau k, même sous format binaire, peut être très coûteux. Cependant, le principal goulot d'étranglement est dû au fait que (i) les n-uplets doivent être ordonnés en fonction de l'itemset graduel considéré et (ii) le calcul de la longueur du plus long chemin prend du temps.

3 Travaux connexes

Nous rappelons dans cette section les principales approches de la littérature pour la découverte de motifs graduels et l'extraction parallèle de motifs fréquents.

3.1 Découverte de motifs graduels

Les motifs graduels sont étudiés depuis de nombreuses années pour le contrôle, la commande (floue) et les systèmes de recommandation. Plus récemment, des algorithmes de fouille de données ont été étudiés pour extraire automatiquement de tels motifs Di Jorio et al. (2009); Berzal et al. (2007); Di Jorio et al. (2008); Fiot et al. (2008); Hüllermeier (2002); Laurent et al. (2009). Ainsi, Hüllermeier (2002) utilise la régression linéaire, tandis que Berzal et al. (2007) définit quatre types de règles graduelles de la forme *plus/moins X est A, alors plus/moins Y est B*, et propose un algorithme par niveaux pour extraire de telles règles. Cependant, le support est calculé en considérant les couples de n-uplets, ce qui rend l'approche complexe. Dans Fiot et al. (2008), les auteurs introduisent les motifs séquentiels graduels pour rendre compte de la force de modification (accélération). Dans Di Jorio et al. (2009) et Di Jorio et al. (2008), deux méthodes sont proposées, la différence étant liée au mode de calcul du support : tandis qu'une heuristique est utilisée dans Di Jorio et al. (2008), le support exact est calculé dans Di Jorio et al. (2009) grâce à la méthode vue précédemment. Dans Laurent et al. (2009), les auteurs proposent une autre manière de calculer le support, en considérant le *Kendall tau ranking correlation coefficient* qui calcule non pas la longeur du plus long chemin, mais le nombre de paires de n-uplets ordonnables dans la base de données pour être en accord avec le motif graduel considéré (on parle alors de paires concordantes et discordantes).

Dans cet article, nous considérons l'approche présentée dans Di Jorio et al. (2009) car elle est à notre connaissance la plus performante.

3.2 Extraction parallèle de motifs fréquents

Depuis 1996, les spécialistes en extraction de motifs fréquents ont travaillé sur des algorithmes parallèles. De nombreux travaux ont été réalisés pour extraire des motifs fréquents sur des clusters (Agrawal et Shafer (1996); Zaki et al. (1997)) ou des SMPs (Zaki (1999)). A cette époque, la mémoire vive des ordinateurs personnels était beaucoup plus petite que la taille de la plupart des bases de données (quelques centaines de MégaOctets / quelques GigaOctets),

donc l'intérêt principal du calcul parallèle était de permettre de traiter efficacement des bases de données de taille importante en les distribuant sur plusieurs machines. Avec l'augmentation de la capacité des mémoires vives et la découverte de manières plus efficaces d'explorer l'espace de recherche (motifs fréquents fermés par exemple), les publications sur l'extraction parallèle de motifs fréquents se sont raréfiées jusqu'en 2005. Depuis l'apparition des processeurs multicœurs (appelés en anglais *Chip MultiProcessors* ou CMP), une nouvelle thématique de recherche est apparue pour la définition d'algorithmes performants utilisant ces processeurs multicœurs. Cette thématique a été initiée par Buehrer et al. (2006), avec un algorithme d'extraction de graphes fréquents présentant d'excellentes techniques de passage à l'échelle. Cet algorithme est basé sur gSpan (Yan et Han (2002)), et leur contribution consiste d'une part en une manière efficace de partager le travail entre les cœurs, et d'autre part en une technique pour exploiter la localité temporelle du cache lors de décisions locales pour savoir s'il faut immédiatement traiter un appel récursif ou s'il faut le mettre en file d'attente. Lucchese et al. (2007) ont ensuite présenté le premier algorithme pour extraire des itemsets fréquents fermés sur CMP. Leur contribution se concentre sur la meilleure manière de répartir le travail, et ils montrent l'intérêt d'utiliser les instructions SIMD pour améliorer encore les performances. La même année, Liu et al. (2007) ont présenté une parallélisation du célèbre algorithme FP-growth (Han et al. (2000)). Plus récemment, Tatikonda et Parthasarathy (2009) ont présenté un algorithme pour extraire des arbres fréquents avec un speed-up quasi-linéaire. Ils montrent que le principal facteur limitant les performances lors de l'extraction de motifs fréquents sur un processeur multicœurs est que la mémoire est partagée entre tous les cœurs. Donc si tous les cœurs accèdent simultanément à une quantité importante de données, le bus entre la mémoire et le processeur va être saturé et les performances vont chuter : il y a trop de contention pour la bande passante Cette observation est opposée à tout ce qui donnait de bons résultats dans le cas séquentiel, où pour éviter des calculs redondants une grande quantité de résultats intermédiaires étaient stockés en mémoire. Ici Tatikonda et al. montrent que la taille de l'ensemble de travail doit être réduite autant que possible, même si cela doit conduire à refaire certains calculs plusieurs fois. Ils montrent également que les traditionnelles structures de données à base de pointeurs sont mal adaptées pour l'extraction parallèle de motifs fréquents sur CMP, à cause de leur mauvaise localité dans le cache, qui là encore amène à avoir trop de contention pour la bande passante.

Dans cet article, nous nous intéressons au problème complexe de l'extraction de motifs graduels. Nous sommes dans un cas favorable où il y a beaucoup de calculs à faire pour chaque bloc de données transféré depuis la mémoire. Il ne devrait donc pas y avoir de contention importante pour la bande passante, à condition de ne pas utiliser des structures de données inutilement volumineuses.

4 PGP-mc : Recherche parallèle de motifs graduels

4.1 Caractéristiques des motifs graduels

Le problème de l'extraction des itemsets graduels diffère des cas classiques liés aux itemsets simples. Dans ce dernier cas, pour chaque ligne de la base de données, il est possible de dire si elle supporte ou non l'itemset. Dans le cas graduel, toute la base de données est nécessaire à chaque comptage. Il n'est donc pas raisonnable d'envisager une distribution de données

par blocs de lignes. Notons que même l'extraction des motifs séquentiels est un problème intermédiaire car le comptage se fait par bloc de lignes, chaque bloc correspondant à toutes les lignes associées au même client.

Le calcul du support d'un itemset graduel est une opération assez complexe et nécessite deux tâches très coûteuses que nous englobons dans une procédure appelée $Join()$. Cette procédure, décrite en détail dans Di Jorio et al. (2009), assure (i) l'ordonnancement des lignes de la base de données et la construction de la matrice binaire associée et (ii) le calcul du plus long chemin.

Notre proposition repose sur le grand nombre de répétitions de ces opérations lors de l'exploration d'une base de données. Il s'agit d'un problème irrégulier dans le sens où il est difficile de prévoir à l'avance le nombre de motifs retenus et leur distribution dans l'espace des candidats construit dynamiquement niveau par niveau. En revanche, nous notons que le calcul du support d'un itemset ne dépend pas du calcul du support des autres itemsets d'un même niveau, ce qui nous permet d'envisager la construction et le test de plusieurs candidats en parallèle. Nous rappelons que nous adoptons une approche *multi-thread* qui sépare et exécute le travail sur différentes unités de calcul (processeurs ou coeurs) procédant de manière concurrente.

4.2 Approche proposée : GRITE-MT

L'algorithme GRITE repose sur une exploration du treillis niveau par niveau. Le premier niveau est initialisé à partir des colonnes de la base. Ensuite les candidats k du niveau $N+1$ sont construits à partir de chaque itemset i du niveau N combiné successivement à l'ensemble de ses itemsets frères j de rang supérieur (noté $Siblings_{j>i}(i)$) grâce à la procédure $Join(i, j)$. Lorsqu'un candidat dépasse un certain seuil de support, il est considéré comme fréquent et conservé. Le processus s'arrête quand le dernier niveau construit ne contient aucun fréquent. L'algorithme 1 montre une version simplifiée de la construction d'un niveau $N + 1$ à partir d'un niveau N.

Algorithme 1 GRITE : construction du niveau $N + 1$ à partir du niveau N.

```
1 PourChaque itemset i De niveau N Faire
2     PourChaque itemset j De Siblings_{j>i}(i) Faire
3         itemset k ← Join(i,j)
4         Si k est fréquent Alors
5             k devient nœud fils de i (et prend l'index j)
6             {k est conservé au niveau N+1}
7         FinSi
8     FinPourChaque
9 FinPourChaque
```

Ici, chaque niveau doit être construit avant de commencer le traitement du niveau suivant. Pour cette raison, nous nous sommes concentrés sur la parallélisation de chaque niveau pris individuellement, où la construction d'un candidat (via la procédure $Join()$) est une opération

essentiellement indépendante des autres. Le problème principal concerne alors l'équilibre de la distribution des tâches sur les unités de traitement disponibles. En effet, le nombre d'opérations au sein de la boucle interne de l'algorithme 1 est difficile à prévoir au delà du niveau 2 et le nombre d'itemsets "frères" pour chaque itemset i peut varier de façon sensible. Une parallélisation automatique de ces boucles risque de conduire à un déséquilibre de la charge.

Notre méthode vise à compenser cette irrégularité en affectant dynamiquement les constructions de candidats d'un même niveau à un *pool* de threads selon la règle "premier arrivé, premier servi". Au début, tous les itemsets fréquents d'un niveau N sont marqués non traités et stockés dans une file d'attente F_i. Un nouvel itemset i est défilé et tous ses nœuds frères de plus haut rang sont ajoutés dans une seconde file F_{si}. Chaque thread disponible extrait alors un itemset j de F_{si} et construit un nouveau candidat k à partir de i et de j. S'il est "fréquent", ce candidat est conservé au niveau $N+1$ comme nœud fils de i. Lorsque la file F_{si} est vide, un nouvel itemset i est extrait de F_i et le processus reprend au début. On s'arrête quand tous les itemsets i ont été traités (*i.e* les deux files sont vides). L'algorithme 2 montre une version simplifiée de cette approche.

Algorithme 2 GRITE-MT (*multithread*) : construction du niveau $N+1$ à partir du niveau N.

```
1  i, j : itemsets
2  P_t : pool de threads
3  F_i : file ← itemsets du niveau N
4  F_si : file ← ∅ {nœuds frères non traités}
5  PourChaque thread De P_t (en parallèle) Faire
6      TantQue F_i ≠ ∅ OR F_si ≠ ∅ Faire
7          Si F_si = ∅ Alors
8              i ← defiler(F_i)
9              F_si ← Siblings_{j>i}(i)
10         FinSi
11         j ← defiler(F_si)
12         itemset k ← Join(i, j)
13         Si k est frequent Alors
14             k devient nœud fils de i (et prend l'index j)
15               {k est conservé au niveau N+1}
16         FinSi
17     FinTantQue
18 FinPourChaque
```

4.3 Mise en œuvre et optimisations préliminaires

Outre les contraintes de bande passante de la mémoire, un autre problème des calculs massivement multithreadés peut apparaître lorsqu'un trop grand nombre d'allocations mémoires

dynamiques ont lieu simultanément. Ces tâches d'allocations sont par défaut dévolues au noyau du système d'exploitation et sont la plupart du temps sérialisées, pouvant introduire des attentes inutiles lors du calcul. Afin de simplifier l'utilisation de la mémoire, aussi bien en terme d'occupation que de nombre de transactions, nous avons profilé et optimisé la version initiale du programme C++ de Di Jorio et al. (2009) sans en modifier l'algorithme. Le programme résultant ne consomme en moyenne que la moitié de la mémoire et presque un tiers seulement du temps d'exécution. Nos travaux de parallélisation et nos expérimentations sont basés sur cette nouvelle version. Notons que les threads ont été implantés via la bibliothèque de threads standard POSIX.

5 Résultats expérimentaux et discussion

Dans cette section, nous présentons les résultats obtenus par l'exécution de notre programme sur deux architectures multi-processeurs comportant jusqu'à 32 cœurs :

- COYOTE dispose de 8 processeurs quadri-cœurs AMD Opteron 852, 64Go de mémoire, Linux Centos 5.1 et g++ 4.1.2
- IDKONN dispose de 4 processeurs Intel Xeon 7460 de 6 cœurs chacun, 64GB of RAM, Linux Debian 5.0.2 et g++ 4.3.2.

Les expériences ont été menées sur des bases artificielles créées automatiquement par un outil basé sur une version adaptée du *Synthetic Data Generation Code for Associations and Sequential Patterns* [1] d'IBM. Cet outil produit des bases numériques en fonction du nombre de lignes, du nombre d'attributs et du nombre moyen de valeurs distinctes par attribut.

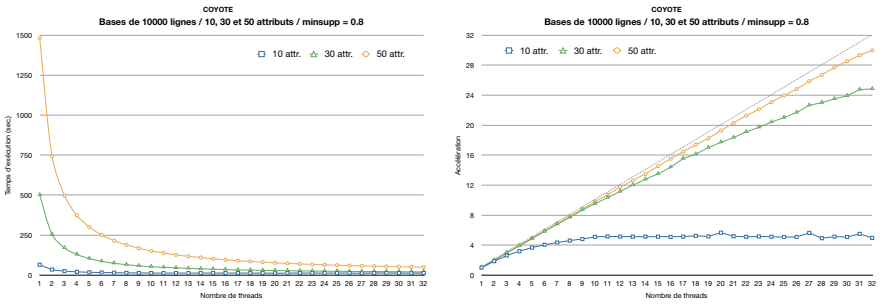
5.1 Accélérations

Les résultats suivants illustrent l'évolution des temps de calcul et des accélérations en fonction de la complexité du problème. Cette complexité peut avoir deux origines : le nombre d'attributs dont dépend le nombre de tâches (*i.e.* le nombre de candidats à synthétiser et à tester) et le nombre de lignes qui influe sur le temps de traitement moyen de chaque tâche (*i.e.* le temps de traitement de la procédure $Join()$). Nous présentons ici deux jeux de test permettant d'observer le comportement de notre solution suivant ces deux aspects.

Le premier jeu de tests concerne des bases de données qui comportent un grand nombre de lignes mais relativement peu d'attributs. Ce genre de bases produit en général assez peu d'itemsets fréquents pour un seuil de support suffisamment élevé. Dès lors le temps global d'exécution du programme est principalement dévolu à la construction des deux premiers niveaux de candidats. La figure 3a montre la diminution des temps d'exécution pour le traitement de bases de données de 10000 lignes pour 10 à 50 attributs sur la machine COYOTE. La figure 3b montre les accélérations correspondantes.

Les accélérations atteignent ici un niveau très satisfaisant dans les cas les plus complexes. Par exemple, l'accélération atteint environ 30 pour 50 attributs avec 32 threads. La limite atteinte pour 10 et 20 attributs s'explique essentiellement par le fait que la construction du premier niveau occupe une grande part du temps d'exécution global du programme. Or, à ce

1. www.almaden.ibm.com/software/projects/hdb/resources.shtml

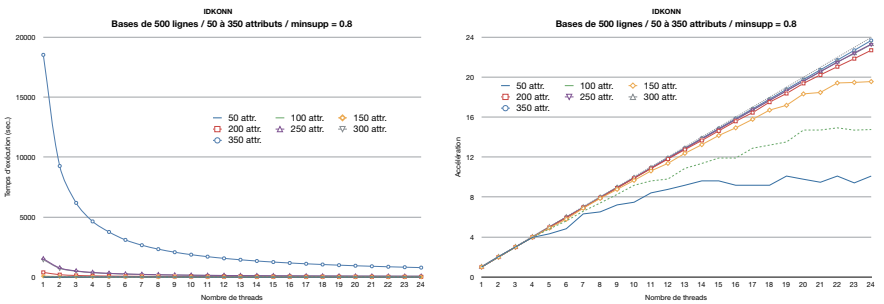


(a) Tps d'exécution (b) Accélération

FIG. 3 – *Tps d'exécution et accélération / nbre de threads (10000 lignes, sur COYOTE).*

niveau, le nombre de tâches - strictement égal au nombre d'attributs - est trop faible pour exploiter plus que quelques unités de traitement et surtout pour assurer un bon équilibre de la charge. Ainsi, le traitement séquentiel de la base de 10 attributs prend environ 64 secondes parmi lesquelles 9 secondes (soit 14%) sont nécessaires pour charger la base et construire le premier niveau. Avec 16 threads ou plus, le temps global d'exécution n'est plus que de 13 secondes mais 5,5 secondes (soit 42%) sont encore nécessaires pour le premier niveau. Les expériences pratiquées sur la machine IDKONN avec les mêmes bases montrent des résultats similaires. Tous les résultats détaillés (tables et courbes) sont consultables en ligne à l'adresse `http://www.lirmm.fr/~laurent/PGP/pgp-mc.EGC10.pdf`.

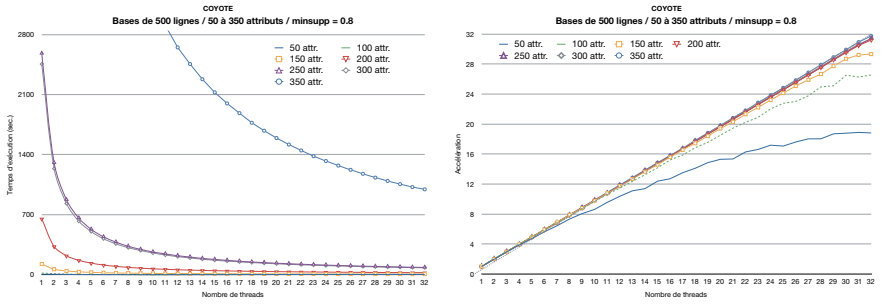
Le deuxième banc de tests concerne les bases de données de complexité croissante en fonction du nombre d'attributs. Les figures 4a et 4b montrent respectivement la diminution des temps de traitement et les accelérations obtenues pour des bases de 500 lignes contenant 50 à 350 attributs, sur la machine IDKONN.



(a) Tps d'exécution (b) Accélération

FIG. 4 – *Tps d'exécution et accélération / nbre de threads (500 lignes, sur IDKONN).*

Là encore, notre solution est très efficace pour les problèmes suffisamment complexes puisque les accélérations sont peu éloignées de la progression maximale théorique à partir de 150 attributs sur IDKONN. Les figures 5a et 5b montrent respectivement les temps d'exécution et les accélérations obtenues pour les mêmes bases entre 50 et 300 attributs sur la machine COYOTE. Le traitement de la base de 350 attributs passe d'une durée de plus de 8 heures et 46 minutes en séquentiel à environ 16 minutes et 33 secondes avec 32 threads sur cette machine.



(a) Tps d'exécution (b) Accélération

FIG. 5 – *Tps d'exécution et accélérations / nbre de threads (500 lignes, sur COYOTE).*

A vu de la similarité des résultats sur deux machines différentes (architectures des processeurs et de la hiérarchie mémoire différentes), on peut noter que les performances en terme d'accélération sont peu sensibles aux spécificités architecturales de la plateforme (mémoire, caches...). Précisons enfin que les résultats du traitement de la base de 500 lignes et 50 attributs ne sont pas nécessairement pertinents dans le cadre de calculs massivement parallèles puisque le temps d'exécution séquentiel est d'environ 3,3 secondes sur COYOTE. Nous les avons toutefois reportés afin de montrer que notre approche permet aussi dans ces situations d'obtenir des accélérations sensibles. Celles-ci peuvent s'avérer déterminantes dans le traitement de problèmes de fouille de données en temps réel ou quasi-temps réel, par exemple pour la détection d'intrusion, ou de manière plus générale la fouille de données arrivant en flots (*stream mining*) puisqu'il faut alors que le traitement soit plus rapide que le taux d'arrivée des données.

5.2 Limitations dues à l'occupation mémoire

La principale difficulté de ce genre de problème reste la maîtrise de la consommation mémoire, mise à mal par le très grand nombre de candidats traités (et éventuellement retenus) à chaque niveau de l'exploration du treillis. À titre d'illustration, nous avons appliqué notre programme sur une base dense de 30 lignes et 1500 colonnes pour un seuil de support élevé (0.9). Nous avons trouvé environ 1,6 millions d'itemsets fréquents au seul niveau 2. Le traitement des 10000 premiers a conduit à la création de 5 millions de nouveaux fréquents au niveau 3. Selon ce rythme de la consommation mémoire, près de 150Go de mémoire auraient été nécessaires pour stocker tous les itemsets fréquents du niveau 3. Ces limitations pour les bases de données très denses nous conduisent à explorer d'autres architectures (clusters notamment).

6 Conclusion et Perspectives

Dans cet article, nous proposons une nouvelle approche pour l'extraction de motifs et règles graduels de la forme *plus une personne est âgée, plus son salaire est élevé*. Notre approche est basée sur l'utilisation des multiples cœurs présents maintenant sur la plupart de nos ordinateurs et sur les serveurs maintenant accessibles à tous. Anciennement réservée à des domaines particuliers, l'utilisation de telles machines est désormais courante et il est important que les algorithmes de fouille de données puissent profiter au mieux de ces architectures. Les expérimentations menées montrent le grand intérêt et l'efficacité de notre approche, atteignant des *speed-ups* quasi linéaires sur les problèmes difficiles et réduisant considérablement les temps de calcul (*e.g.* 16 min. 33 sec. avec 32 threads contre 8h46 en séquentiel).

Les perspectives associées à ce travail sont nombreuses. Au-delà des optimisations très techniques (optimisations ad-hoc selon les différentes architectures), nous allons en particulier nous intéresser à trois pistes principales : l'utilisation des motifs graduels clos pour réduire les temps de calcul, des algorithmes en profondeur (patterns growth) et l'utilisation d'un autre mode de parallélisation, les clusters (y compris des clusters de machines mutliprocesseurs/multi-cores).

Remerciements. Les auteurs remercient Lisa Di Jorio pour leur avoir fourni les sources de son implémentation de recherche des motifs et règles graduels (Di Jorio et al. (2009)).

Références

Agrawal, R. et J. C. Shafer (1996). Parallel mining of association rules. *IEEE Trans. Knowl. Data Eng. 8*(6), 962–969.

Agrawal, R. et R. Srikant (1994). Fast algorithms for mining association rules. In *Proceedings of the 20th VLDB Conference*, pp. 487–499.

Berzal, F., J.-C. Cubero, D. Sanchez, M.-A. Vila, et J. M. Serrano (2007). An alternative approach to discover gradual dependencies. *Int. Journal of Uncertainty, Fuzziness and Knowledge-Based Systems (IJUFKS) 15*(5), 559–570.

Buehrer, G., S. Parthasarathy, et Y.-K. Chen (2006). Adaptive parallel graph mining for cmp architectures. In *ICDM*, pp. 97–106.

Di Jorio, L., A. Laurent, et M. Teisseire (2008). Fast extraction of gradual association rules : A heuristic based method. In *IEEE/ACM Int. Conf. on Soft computing as Transdisciplinary Science and Technology, CSTST'08*.

Di Jorio, L., A. Laurent, et M. Teisseire (2009). Mining frequent gradual itemsets from large databases. In *Int. Conf. on Intelligent Data Analysis, IDA'09*.

Fiot, C., F. Masseglia, A. Laurent, et M. Teisseire (2008). Gradual trends in fuzzy sequential patterns. In *Proc. of the Int. Conf. on Information Processing and Management of Uncertainty in Knowledge-based Systems (IPMU)*.

Han, J. et M. Kamber (2006). *Data Mining : Concepts and Techniques* (2nd ed.). The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers.

Han, J., J. Pei, et Y. Yin (2000). Mining frequent patterns without candidate generation. In *Proc. of the International Conference on Management of Data*, pp. 1–12.

Hüllermeier, E. (2002). Association rules for expressing gradual dependencies. In *Proc. of the 6th European Conf. on Principles of Data Mining and Knowledge Discovery, PKDD'02*, pp. 200–211. Springer-Verlag.

Laurent, A., M.-J. Lesot, et M. Rifqi (2009). Graank : Exploiting rank correlations for extracting gradual dependencies. In *Proc. of FQAS'09*.

Liu, L., E. Li, Y. Zhang, et Z. Tang (2007). Optimization of frequent itemset mining on multiple-core processor. In *VLDB '07 : Proceedings of the 33rd international conference on Very large data bases*, pp. 1275–1285. VLDB Endowment.

Lucchese, C., S. Orlando, et R. Perego (2007). Parallel mining of frequent closed patterns : Harnessing modern computer architectures. In *ICDM*, pp. 242–251.

Pasquier, N., Yves, Y. Bastide, R. Taouil, et L. Lakhal (1999). Efficient mining of association rules using closed itemset lattices. *Information Systems 24*, 25–46.

Tatikonda, S. et S. Parthasarathy (2009). Mining tree-structured data on multicore systems. In *VLDB '09 : Proceedings of the 35th international conference on Very large data bases*.

Uno, T. (2005). Lcm ver. 3 : Collaboration of array, bitmap and prefix tree for frequent itemset mining. In *In Proc. of the ACM SIGKDD Open Source Data Mining Workshop on Frequent Pattern Mining Implementations*, pp. 77–86.

Yan, X. et J. Han (2002). gspan : Graph-based substructure pattern mining. In *Proc. of the IEEE International Conference on Data Mining*, pp. 721. IEEE Computer Society.

Zaki, M. J. (1999). Parallel sequence mining on shared-memory machines. In *Large-Scale Parallel Data Mining*, pp. 161–189.

Zaki, M. J., S. Parthasarathy, M. Ogihara, et W. Li (1997). Parallel algorithms for discovery of association rules. *Data Min. Knowl. Discov. 1*(4), 343–373.

Summary

Gradual patterns of the form "the older, the higher the salary" have been extensively used in (fuzzy) command systems. They currently play a crucial role in many real world applications where huge volumes of complex numerical data must be handled, e.g., biological databases, survey databases, data streams or sensor readings. Only recently algorithms have appeared to mine efficiently such gradual patterns. However, due to the complexity of mining gradual rules, these algorithms cannot yet scale on huge real world datasets. In this paper, we thus propose to exploit parallelism in order to enhance the performances of the fastest existing one (GRITE). Through a detailed experimental study, we show that our parallel algorithm scales very well with the number of cores available.