Algoritmos Paralelos e Distribuídos para Mineração de Regras de Associação

Bruno Estolano Grossi*
bgrossi@dcc.ufmg.br

22 de junho de 2004

Resumo

Mineração de Regras de Associação vem se tornando cada vez mais importante em diversas áreas. Grandes bases de dados são disponibilizados pelos sistemas e recuperar informações úteis não é uma tarefa fácil. Farei aqui uma revisão sobre o problema e mostrarei alguns dos algoritmos existentes para extrair regras de associação significativas a partir de um conjunto de transações.

1 Introdução

O uso de sistemas informatizados gera uma quantidade cada vez maior de dados. Registros de compras de supermercados e de utilização de planos de saúde, informações sobre conseções de crédito, e diversos outros dados que, sem nenhum tratamento especial, não trazem conhecimento relevante. Tratar corretamente esses dados pode trazer informações aplicáveis a várias áreas, como a tomada de decisões nas empresas. Esse é o objetivo da Mineração de Dados.

Mineração de Dados é um processo que usa uma variedade de ferramentas para análise de dados para descobrir padrões e relacionamentos no dados que podem ser utilizados para fazer prediçoes válidas [1]. Segundo Agrawal et. al.[2] referenciado em [3], Mineração de Regras de Associação (ARM, sigla em inglês) vem se tornando uma das principais tarefas de mineração de dados. O uso de ARM permite extrair regras que relacionam itens que freqüentemente ocorrem juntos em uma transação. Ou seja, definir como e quanto a presença de um subconjunto de itens influencia na presença de outro subconjunto. O uso de ARM tem muito valor para o marketing e disponibilização de produtos nas lojas, e pode ser usado em várias áreas, como pesquisas médicas. Com o uso de ARM podemos identificar regras como:

- 98% dos clientes que compram pneus e acessórios automotivos compram também o serviço pronto [4];
- a maior parte dos pacientes que têm os genes X e Y possuem a doença Z;
- quem compra os produtos A e B, na maioria das vezes, compra também C;
- quase todas as mulheres que possuem carros esportes vermelhos e cães pequenos usam uma determinada tintura para cabelo;

Analizar todas essas informações tem um custo computacional muito alto, já que envolve enormes quantidades de dados, e a maioria dos mais eficientes algoritmos sequenciais podem se tornar ineficientes [5]. Assim, a implementação de algoritmos de mineração de dados em ambientes paralelos de alta performance é crucial para melhorar os tempos de resposta.

Nesse *paper* pretendo apresentar algumas técnicas e algoritmos conhecidos para fazer Mineração de Regras de Associação. No restante dessa seção definirei o problema formalmente e mostrarei um exemplo. Na seção 2 mostrarei alguns fundamentos para a resolução do problema e algumas das possíveis formas de organizar os dados. Na seção 3 discutirei os desafios para a criação de algoritmos paralelos eficientes. Na seção 4 darei uma visão geral de alguns algoritmos sequenciais conhecidos, com enfoque para o Apriori e na secao 5 veremos os variantes paralelos.

1.1 Definição formal do Problema

Seja um conjunto de m ítens $\mathcal{I} = \{x_1, x_2, ..., x_m\}$ e $\mathcal{D}$ um conjunto de transações T, onde cada transação é um conjunto de ítens tal que $T \subseteq \mathcal{I}$ e possui um identificador único (*tid*). Dizemos que uma transação T contém X, um conjunto de ítens (*itemset*) em $\mathcal{I}$, se $X \subseteq T$. Um conjunto de ítens de tamanho k é chamado *k-itemset*. Um *itemset* possui *suporte* s em $\mathcal{D}$ se ele está presente em s% das transações de $\mathcal{D}$. Um *itemset* é dito freqüente (também chamado *large*) se possui suporte maior ou igual a um mínimo estipulado (min_sup). Chamamos de F_k o conjunto de todos os *k-itemsets* freqüentes.

*Aluno do curso de Projeto e Análise de Algoritmos, Mestrado em Ciência da Computação do DCC/UFMG.

Uma *regra de associação* é uma implicação da forma $X \Rightarrow Y$, onde $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$ (X e Y são *itemsets*) e $X \cap Y = \emptyset$. A regra $X \Rightarrow Y$ tem *confiança* c para o conjunto de transações $\mathcal{D}$ se c% das transações em $\mathcal{D}$ que contém X também contém Y. A regra $X \Rightarrow Y$ tem *suporte* s em $\mathcal{D}$ se s% das transações em $\mathcal{D}$ contém $X \cup Y$. Uma regra é dita freqüente se possui suporte s maior ou igual a um mínimo estipulado (min_sup). Uma regra é dita forte se possui confiança maior ou igual a uma confiança mínima (min_conf).

Dado um conjunto de transações $\mathcal{D}$, o problema de minerar regras de associação é gerar todas as associações que têm suporte e confiança maiores que um mínimo (*min_ sup* e *min_ conf*) especificado pelo usuário.

1.2 Exemplo

Seja um conjunto de ítens $\mathcal{I} = \{A, B, C, D, E, F\}$ e as seguintes transações em $\mathcal{D}$:

$$
\begin{array}{ll}
tid_0 = \{A, B, C\} & tid_5 = \{A, C, D, F\} \\
tid_1 = \{A, C, D, E, F\} & tid_6 = \{A, C, E\} \\
tid_2 = \{B, C, D\} & tid_7 = \{C, D, E\} \\
tid_3 = \{B, C, E, F\} & tid_8 = \{A, C, D, E\} \\
tid_4 = \{A, B, C, D, E\} & tid_9 = \{A, C, D\}
\end{array}
$$

Se considerarmos um suporte mínimo (min_sup) de 50% e quisermos encontrar todos os conjuntos de ítens que são freqüentes, ou seja, aparecem em, no mínimo 50% das transações, temos os seguintes conjuntos de *k-itemsets* freqüentes:

$$
\begin{array}{l}
F_1 = \{\{A\}, \{C\}, \{D\}, \{E\}\} \\
F_2 = \{\{A, C\}, \{A, D\}, \{C, D\}, \{C, E\}\} \\
F_3 = \{\{A, C, D\}\}
\end{array}
$$

2 Mineração de Regras de Associação

2.1 Busca por Conjuntos Freqüentes

É fácil notar que um *itemset* X de tamanho k é freqüente se, e somente se, todo subconjunto Y de X, de tamanho k-1, também é freqüente. Ou seja, $X_k \in F_k \Leftrightarrow \forall Y_{k-1} \subset X_k, Y_{k-1} \in F_{k-1}$. Em outras palavras, se Y não é freqüente, e X contém Y, não precisamos contabilizar X para saber que esse conjunto não é freqüente e pode ser descartado; por outro lado, se X é freqüente, Y também é freqüente.

Para um dado conjunto de transações, temos as seguintes opções de pesquisa por conjuntos freqüentes:

1. Calcular a freqüencia de cada conjunto de tamanho 1 (*1-itemset*), eliminando os que estão abaixo do mínimo. A partir do conjunto de *1-itemsets* freqüentes (F_1), geramos conjuntos de tamanho 2 (*2-itemset*) e selecionamos os conjuntos que atendem ao mínimo. O ciclo se repete para formarmos F_x de F_{x-1} até que não seja mais possível gerar novos conjuntos. Esse método é conhecido como bottom-up. (Figura 1)

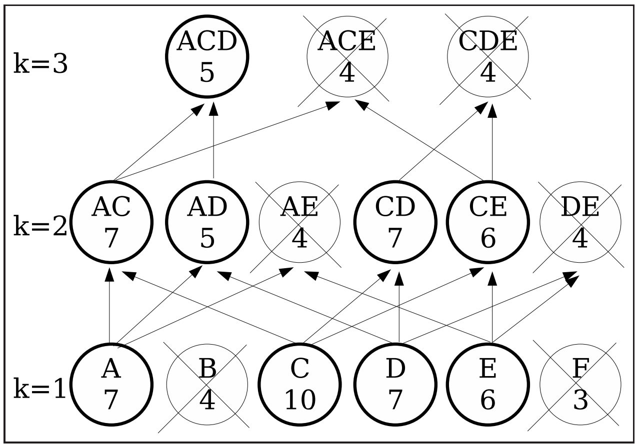

Figura 1: Método *bottom-up* para encontrar os *itemsets* freqüentes. No exemplo da seção 1.2, somente 15 subconjuntos de $\mathcal{I}$ precisam ser testados, e não $2^m = 32$ subconjuntos.

2. Calcular a freqüencia do conjunto de todos os ítens (*m-itemset* $= \mathcal{I}$); se este for freqüente, todos os seus subconjuntos são freqüentes. Caso não seja, geramos todos os subconjuntos de tamanho $m - 1$ e contabilizamos a quantidade de aparições; os que forem freqüentes indicam que seus subconjuntos são freqüentes. O ciclo se repete até que não haja mais conjuntos para serem testados. Esse método é conhecido como top-down, e pode se tornar inviável para um conjunto de ítens $\mathcal{I}$ muito grande, onde normalmente uma transação terá tamanho muito menor que a quantidade de ítens. Por exemplo, num supermercado dificilmente uma compra possui metade dos produtos disponíveis. Esse formato é mais utilizado quando se quer apenas os maiores conjuntos freqüentes, e não todos os conjuntos.

3. Alguns algoritmos usam uma pesquisa híbrida, combinando top-down e bottom-up.

2.2 Organização dos Dados

A maioria dos algoritmos tradicionais organizam os dados de forma Horizontal, ou seja, um conjunto de transações, com cada *tid* relacionado a uma lista de ítens da transação (como no exemplo da seção

1.2). Esse formato impõe uma maior complexidade durante a fase de contagem do suporte. Para cada transação de tamanho l, durante a interação k, é necessário gerar e testar se $\binom{l}{k}$ *k-subsets* da transação está contido em C_k. Para fazer a checagem dos subconjuntos de forma rápida, os candidatos são armazenados em uma estrutura complexa como árvore de *hash*[6].

No formato Vertical (ou lista invertida), os dados são disponibilizados como uma lista de ítens, associando a cada ítem a lista (*tidlist*) com todos os identificadores das transações que o contém. O formato Vertical não tem toda a complexidade do Horizontal. Se as *tidlists* estiverem ordenadas, o suporte de um candidato de tamanho k pode ser contabilizado simplesmente usando interseção de duas listas de *(k-1)-itemsets*. Não são necessárias complicadas estruturas de dados e não é necessário gerar todos os subconjuntos de tamanho k da transação. Por exemplo, as ocorrências do *2-itemset* $\{A, D\} = A \cap D = \{1, 4, 5, 8, 9\}$. Veja mais detalhes em [6].

A organização vertical para o exemplo anterior é: $tidlist_A = \{0, 1, 4, 5, 6, 8, 9\}$, $tidlist_B = \{0, 2, 3, 4\}$, $tidlist_C = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $tidlist_D = \{1, 2, 4, 5, 7, 8, 9\}$, $tidlist_E = \{1, 3, 4, 6, 7, 8\}$ e $tidlist_F = \{1, 3, 5\}$.

3 ARM Paralelo e Distribuído

O grande custo computacional necessário para o processamento de grandes bases de dados torna atraente o uso de paralelismo para resolver problemas dos algoritmos sequenciais, permitindo mais escalabilidade e melhorando o tempo de resposta. Porém, obter bom desempenho nos atuais sistemas multiprocessados não é trivial[3].

É preciso reduzir ao máximo a comunicação e sincronização entre os processadores, além da quantidade de acesso ao disco. Encontrar uma boa organização e decomposição dos dados e balancear a carga também são grandes desafios.

A principal diferença entre ARM paralelo e distribuído é a latência e largura de banda da rede de interconexão. A comunicação entre os processadores em ambientes distribuídos (uma rede local, por exemplo) é considerada mais lenta, mas vem se tornando mais comum devido uma melhor relação custo-desempenho[7].

3.1 Memória Compartilhada ou Distribuída

Algoritmos paralelos podem utilizar basicamente dois esquemas de memória: Memória Distribuída, onde cada processador tem sua própria memória local, ou Memória Compartilhada, onde todos os processadores acessam e utilizam uma mesma memória do sistema.

Uma arquitetura de memória compartilhada (SMP, sigla em inglês) é, a princípio, mais fácil de ser programado, pois não é necessário o particionamento e a distribuição dos dados. Porém, a largura de banda finita do barramento pode limitar a escalabilidade do sistema, já que todos os processadores terão de acessar a todo instante a memória. Na arquitetura com memória distribuída (DMM, em inglês), cada processador tem sua própria memória, que nenhum outro pode acessar diretamente. Isso gera a necessidade de mecanismo de troca mensagens entre os processadores e particionamento do problema, tornando a programação mais complicada. Porém, é mais escalável. Um exemplo de DMM muito utilizado atualmente é a rede de estações de trabalho[7].

Um outro paradigma combina memória distribuída e compartilhada, distribuindo a memória física entre os nodos, mas provendo um endereçamento global da memória em cada um dos processadores. Para isso é feito, por hardware ou software, cache consistente dos dados, que sempre reflete a última modificação daquele espaço. Outra forma de misturar o melhor do SMP e DMM é com *clusters* de máquinas SMP, o que é chamado de arquitetura paralela hierárquica, que usa primitivas de SMP em um nodo e troca de mensagens entre os nodos SMP. Isso tem se tornado comum com o barateamento de máquinas pessoais multiprocessadas.

3.2 Paralelismo de Dados e Paralelismo de Controle

Para ARM, paralelismo de dados corresponde ao caso onde a base de dados é particionada entre P processadores (fisicamente no DMM e logicamente no SMP). Cada processados trabalha em sua partição local mas executa a mesma tarefa de contabilizar o suporte para os *itemsets* candidatos globais.

Paralelismo de controle (ou tarefa) é o caso em que cada processador executa uma tarefa diferente e independente, como contar conjuntos disjuntos de candidados, mas precisam acessar a base inteira. Para DMM, acessar toda a base implica em controle de replicação dos dados ou comunicação explícita entre os processadores.

Há tambem paralelismo híbrido, que combina tanto paralelismo de dados quanto de controle.

3.3 Sincronização de Dados e Otimização

Para utilizar DMM, sincronização de dados envolve muita troca de mensagens entre os processadores.

Por tanto, para obter um bom desempenho é importante utilizar algoritmos que focam em reduzir a comunicação, por meio de uma boa decomposição dos dados.

No SMP, o sincronismo de dados é feito através de *locks*, onde um processo bloqueia a posição de memória que está utilizando. Por tanto, algoritmos para essa arquitetura devem posicionar bem os dados, para evitar que dois ou mais processadores fiquem acessando uma mesma posição todo o tempo.

Na arquitetura hierárquica, a otimização deve ser nos dois pontos: comunicação e decomposição dos dados.

4 Algoritmos Sequenciais

4.1 *Apriori*

O Apriori foi proposto em [4] e é considerado um dos melhores algoritmos de ARM, sendo a base para a maioria dos algoritmos utilizados. Usa pesquisa *bottom-up* (veja 2.2), com organização horizontal dos dados e enumera todos os conjuntos freqëentes.

O primeiro passo do algoritmo é contar as ocorrências dos ítens para determinar os *1-itemset*s freqüentes. Para cada passo subseqüente, chamado passo k, consiste de duas fases:

1. Os *itemset*s freqüentes encontrados no passo anterior (F_{k-1}) são usados para gerar o conjunto de candidatos C_k. No nosso exemplo, de $F_2 = \{\{A,C\},\{A,D\},\{C,D\},\{C,E\}\}$ geramos $C_3 = \{\{A,C,D\}\}$. A geração do candidato C_k é feita em duas etapas:

   (a) Na etapa de *join*, é gerado C_k da união entre os elementos de F_{k-1}. Teremos $C_3 = \{\{A,C,D\}, \{A,C,E\}, \{A,D,E\}, \{C,D,E\}\}$.

   (b) Na etapa de *prune*, todos os *itemset*s $c \in C_k$ tal que algum subconjunto de tamanho $k-1$ de c não está em F_{k-1} são removidas de C_k. Nesse caso, $\{A,C,E\}$, $\{A,D,E\}$ e $\{C,D,E\}$ serão removidos, pois $\{A,E\}$ e $\{D,E\}$ não estão em F_2.

2. A base de dados é pesquisada e o suporte de cada candidato em C_k é contabilizado, gerando o conjunto F_k.

Observe que a etapa de *prune* representa um ganho considerável, gerando e contabilizando apenas um *itemset*, $\{A,C,D\}$, pois conclui, *a priori*, que as outras combinações não têm o suporte mínimo. Porém essa etapa exige que sejam testados todos os subconjuntos de tamanho $k-1$ de C_k para verificar que estão em F_{k-1}. Para tornar essa verificação rápida, os *itemset*s freqüentes são guardados em uma tabela de *hash*.

Os *itemset*s candidatos em C_k são armazenados em uma *hash-tree*. Um nó raiz dessa árvore contém uma lista de *itemset*s e os nós internos uma tabela de *hash*. Com isso, podemos verificar se um *itemset* está contido em uma transação em $O(k)$, com o auxilio de um mapa de bits. Esse mapa de bits, onde cada ítem corresponde a um bit, é criado uma única vez para a estrutura de dados, com custo $O(l)$ para cada transação (sendo l o tamanho da transação). Veja mais detalhes em [4].

4.2 Partition

O algoritmo Partition, proposto em [8], foi motivado pela tentativa de reduzir acesso à base no Apriori. Para isso, organiza a base de dados na horizontal, e a divide em partições não sobrepostas. Para cada partição, gera uma lista invertida (na vertical) para cada ítem, gerando todos os *itemset*s freqüentes locais usando interseção. Os conjuntos freqüentes locais são fundidos para formar um conjunto candidato global. É feita, então, uma nova consulta às partições para contabilizar os candidatos globais usando a interseção entre as listas de transações.

Uma partição, com seus *itemset*s e *tidlist*s tem que caber na memória principal, melhorando o desempenho ao evitar muito acesso a disco. A escolha do tamanho da partição é feita por heurística, considerando a memória principal disponível e tamanho médio das transações[8]. Para bases pequenas, uma única partição pode ser usada, contendo todas as transações, não trazendo grande vantagem.

4.3 Dynamic Hashing and Pruning (DHP)

O DHP, proposto em [9], extende o Apriori usando uma tabela de *hash* para pré-computar o suporte aproximado dos *2-itemset*s durante a primeira interação. Essa técnica permite reduzir de forma significativa o número de candidatos em C_2 (e nos passos seguintes), melhorando o tempo de execução do processo inteiro.

Além disso, o DHP reduz o tamanho da base de dados $\mathcal{D}$ progressivamente, reduzindo o tamanho de cada transação e também eliminando transações na base, o que melhora o tempo necessário para as contagens dos candidatos nos passos seguintes. Isso é feito eliminado nas transações em $\mathcal{D}$ os subconjuntos que não gerarão *itemset*s freqüentes. No nosso exemplo, poderíamos, durante a contagem dos candidatos em C_2, eliminar o subconjunto $\{B\}$ das transações pois ele não pertence a F_1 e não será mais útil.

4.4 SEAR e Spear

O algoritmo Sequential Efficient Association Rules (SEAR), proposto em [10] é idêntico ao APriori, porém utiliza uma árvore de prefixos para armazenar os candidatos, ao invés da árvore de *hash*. Ele também usa uma otimização onde gera candidatos para vários passos, desde que caibam na memória.

O Spear é similar ao SEAR, porém usa a mesma técnica de particionamento do Partition, mas sem lista invertida. O Spear usa o formato horizontal e faz duas varreduras: primeiro recolhe os candidatos a conjuntos freqüentes e depois obtém o suporte global destes.

O objetivo dos autores com esses algoritmos era avaliar os benefícios do particionamento independente do formato dos dados usado. Eles concluíram que o particionamento não ajuda, pois aumenta a sobrecarga para processar múltiplas partições e gera muitos conjuntos freqüentes localmente, mas infreqüentes globalmente[3].

4.5 Dynamic Itemset Counting (DIC)

O DIC, proposto em [11], é uma generalização do Apriori, com uma tentativa de reduzir o número de leituras sobre a base. A base é dividida em p partições de mesmo tamanho M de tal forma que cada partição caiba na memória. Numa partição $k+1$, ele contabiliza os *k-itemsets* freqüentes da partição anterior. Na partição 1, coleta o suporte dos *1-itemsets*, e, com os freqüentes locais, gera candidatos de tamanho 2. Lê, então, a partição 2 e obtém o suporte para todos os candidatos atuais: os *1-itemsets* iniciais e *2-itemsets* candidatos (gerados na partição anterior). Esse processo se repete para as próximas partições. Após a última partição, o processamento volta para a partição 1 novamente. O suporte global de um candidato é conhecido quando se volta à partição onde foi inicialmente gerado. Ele usa hash tree como o Apriori, mas com algumas informações extras armazenadas em cada nodo, marcando cada conjunto como Suspeito de ser Freqüente, Suspeito de ser Infreqüente, Freqüente Confirmado, Infreqüente Confirmado.

Por exemplo, se estivermos minerando 40.000 transações e M=10.000, vamos contar todos os *1-itemsets* na primeira vez que lemos as 40.000 transações. Entretanto, começaremos a contar os *2-itemsets* logo após as primeiras 10.000 primeiras transações e os *3-itemsets* após 20.000 transações. Vamos considerar que não há conjuntos de tamanho 4. Ao alcançar o fim do arquivo, paremos de contar os *1-itemsets* (já será conhecido o suporte global), e voltaremos ao início para continuar a contar os *2-itemsets* e *3-itemsets*. Após os primeiros 10.000 registros, pararemos de contar os conjuntos de tamanho 2 e após os 20.000, pararemos de contar os de tamanho 3. No total, será feita a leitura das transações 1,5 vezes, ao invés de 3 como seria no Apriori.[11]

DIC é eficiente em reduzir o número de varreduras na base se a maioria das partições for homogênia (têm distribuição de conjuntos freqüentes similar). Se os dados não são homogênios, pode gerar muitos conjuntos freqüentes locais mas infreqüentes globais e varrer a base mais que o Apriori[3].

5 Algoritmos Paralelos

5.1 Baseados no Apriori

5.1.1 Count, Data, Candidate Distribution

Propostos em [12] pelos desenvolvedores do Apriori, consideram arquitetura de memória distribuída (DMM), onde cada processador tem memória e disco próprio e cada disco possui a mesma quantidade de transações.

O algoritmo Count Distribution é uma paralelização simples do Apriori. Todos os processadores geram a árvore de *hash* de candidatos inteira a partir de F_k. Cada processador extrai o suporte local para os candidatos a partir de sua partição da base. Uma sincronização é feita então para gerar a contagem global a partir dos locais. Apenas a contagem é enviada na comunicação, já que todos possuem a árvore inteira. Com o novo F_k gerado, cada processador gera novamente o conjunto inteiro de candidatos C_{k+1} em paralelo e repete o procedimento. A comunicação é minimizada, pois a maior parte do processamento é assíncrono, porém, como a árvore é inteiramente replicada, a memória total não é usada de maneira eficiente. O número de candidatos contabilizados por leitura da base não é maior que do algoritmo sequencial.

O Data Distribution usa a memória total do sistema gerando apenas conjuntos disjuntos de candidatos. Dessa forma, ao aumentar o número de processadores, um maior número de candidatos podem ser contabilizados em uma passagem pela base, ou seja, com N processadores, esse algoritmo pode avaliar em uma leitura o mesmo que o Count gastaria N leituras. Porém, para calcular o suporte global, cada processador tem que consultar a base inteira, em todas a interações. Para consultar a parte remota (disponível em outro processador), é usada muita comunicação, perdendo muito desempenho.

O Candidate Distribution particiona os dados e o conjunto de candidatos de tal forma que cada processador possa executar independente. Numa interação l, determinada heuristicamente, a lista de conjuntos freqüentes F_{l-1} é dividida entre os processadores de forma que cada processador possa gera um conjunto de candidatos independente (disjunto dos demais). Ao mesmo tempo, os dados são reparticionados para que cada processador possa contabilizar, unicamente com seus dados locais, o suporte global para o seu conjunto de candidatos, permitindo aí replicação de partes da base. Dessa forma, a única dependência dos outros processadores é a poda do conjunto local de candidatos durante o passo de *prune* da geração de candidatos. Entretanto, essa informação é enviada de forma assíncrona. A informação de poda que chega atrazada poderá ser usada mais à frente, em uma outra interação. Esse algoritmo se mostrou pior que o Count, por perder muito na redistribuição da base de dados.

5.1.2 Common Candidate Partitioned Database (CCPD) e Parttitioned Candidate Common Database (PCCD)

O CCPD e PCCD[13] são voltados para a arquitetura de memória compartilhada (SMP).

O CCPD usa uma árvore de *hash* de candidatos comum a todos os processadores, enquanto a base é dividida entre eles. A árvore é feita em paralelo, utilizando *lock* nos nós folha, para sincronização. Para a contagem do suporte dos *itemsets*, cada processador utiliza sua partição local. Finalmente, o processo mestre seleciona os conjuntos freqüentes.

O PCCD, ao contrário, particiona a árvore de candidatos, mas compartilha a base. Uma árvore de candidatos local é gerada por processador, que varre a base inteira para contabilizar o suporte dos conjuntos em sua árvore local. O processo mestre, então, executa uma redução e seleciona os conjuntos freqüentes para a próxima interação. Os resultados encontrados para esse algoritmo foram inaceitáveis, causando sobrecarga de mais de um processador.

5.2 Baseado no DHP

5.2.1 PDM

O PDM é uma generalização do DHP[14]. Foi desenvolvido para que o conjunto global de *itemsets* freqüentes possa ser paralelizado eficientemente e a quantidade de troca de dados entre processadores seja minimizada. A base de transações é particionada entre os processadores e cada um gera hashing para identificar os candidatos de tamanho k da sua base local. Um *broadcast* de todas as contagens locais gera uma contagem global. Para evitar comunicação em excesso, que pode se tornar muito custoso, uma otimização é feita para trocar apenas conjuntos que garantidamente são freqüentes.

Apenas a segunda interação utiliza tabelas de *hash*; as seguintes geram os candidatos diretamente de F_{k-1} (como no Apriori)[3]. Os candidatos são gerados em paralelo, com cada processador gerando seu próprio conjunto, que é trocado por *broadcast* para construir o conjunto de candidatos global. A seguir, o PDM obtem a contagem local para todos os candidatos e troca a informação entre os processadores para encontrar os conjuntos freqüentes globais.

O custo de comunicação desse algoritmo, com vários *broadcast*s de todos para todos, torna essa paralelização ineficiente.

5.3 Baseado o SEAR e Spear

5.3.1 PEAR

Sendo um dos primeiros algoritmos paralelos, o PEAR[10] é a versão paralela do SEAR. A cada interação, todos os processadores geram uma árvore de prefixos de candidados a partir dos conjuntos freqüentes globais. Cada processador possui a cópia inteira do mesmo conjunto de candidatos. Cada um recolhe os suportes locais, seguido de uma redução para obter o suporte global em cada processador.

5.3.2 Partitioned Parallel Association Rules (PPAR)

O PPAR é baseado no Spear, sugerido pelo autor do Partition[8], porém usando formato horizontal dos dados. Cada processador coleta localmente os conjutos freqüentes de todos os tamanhos em uma varredura da sua base local (que pode também estar particionado internamente). Em seguida, todos os potenciais conjuntos freqüentes são enviados por *broadcast* para os outros processadores e os candidatos globais são contabilizados em um segundo momento. Finalmente um *broadcast* obtem os conjuntos freqüentes globais.

Experimentos mostraram que o PEAR tem desempenho melhor que o PPAR, pois este último pode gerar muitos candidatos que terminam infreqüentes.

5.4 Baseado no DIC

5.4.1 Asynchronous Parallel Mining (APM)

O APM[15] é baseado no DIC, e voltado para arquitetura de memória compartilhada (SMP). A idéia é dividir a base de dados em p partições homogênias. Cada processador aplica independentemente o DIC à sua partição local. Entretanto, há uma árvore de prefixos compartilhada entre todos os processadores que é construída assincronamente. APM pára quando todos os processadores contabilizarem todos os candidatos, gerados por eles mesmos e pelos outros, e nenhum candidato novo for gerado. Para aplicar o DIC à sua partição local, cada processador a divide em r subpartições. DIC requer que tanto as p partições quanto as r subpartições de cada partição sejam o mais homogênio possível.

Os méritos do algoritmo são, segundo seus autores[15]: (1) conjunto de candidatos muito pequeno, (2) requer menos I/O que os algoritmos tradicionais baseados em nível, (3) é assincrono, o que o torna uma boa escolha para máquinas SMP. Experimentos indicaram que o APM tem melhor desempenho que o Count Distribution e o CCPD.

6 Conclusão

Os algoritmos para Mineração de Regras de Associação vêm evoluindo bastante. O tamanho cada vez maior das bases de transações que precisam ser analisadas inviabiliza o processamento seqüencial, e torna importante o desenvolvimento de algoritmos paralelos e distribuídos eficientes. Porém essa também não é uma tarefa fácil. É importante balancear a quantidade de comunicação entre processadores com a geração de informações importantes para atigir o objetivo, minizando o acesso à base (ou ao disco) e melhorando a distribuição de dados e tarefas.

Existem muitos outros tipos de algoritmos não abordados aqui, como, por exemplo, os baseados em classes (Eclat[6] e derivados) e grafos (Clique[6] e derivados) e os que usam modelo de programação baseado em *filter-streams*[5], onde o processo de mineração é representado como um fluxo de dados. Esses algoritmos devem ser estudados com mais cuidado, pois utilizam paradigmas totalmente diferentes dos algoritmos tradicionais.

Referências

[1] Two Crows Corporation. *Introduction to Data Mining and Knowledge Discovery.* Third edition, 1999.

[2] R. Agrawal et al. Fast discovery of association rules. *Advances in Knowledge Discovery and Data Mining*, 1995.

[3] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, 1999.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. 20th Intl. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[5] A. Veloso et al. Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining.

[6] Mohammed J. Zaki et al. New algorithms for fast discovery of association rules. Technical Report TR651, 1997.

[7] Anderson T. E. and orthers. A case for NOW (Networks of Workstations). *Micro, IEEE*, 15:54–64, 1995.

[8] Ashoka Savasere et al. An efficient algorithm for mining association rules in large databases. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proc. of 21th Intl. Conf. on Very Large Data Bases, Sep. 11-15, 1995, Zurich, Switzerland*, pages 432–444. Morgan Kaufmann, 1995.

[9] Jong Soo Park et al. An effective hash based algorithm for mining association rules. In Michael J. Carey and Donovan A. Schneider, editors, *Proc. of the 1995 ACM SIGMOD Intl. Conf. on Management of Data*, pages 175–186, San Jose, California, 22–25 1995.

[10] Andreas Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, Departure of Computer Science, University of Maryland, College Park, MD, 1995.

[11] Sergey Brin et al. Dynamic itemset counting and implication rules for market basket data. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD Intl. Conf. on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 255–264. ACM Press, 05 1997.

[12] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *Ieee Trans. On Knowledge And Data Engineering*, 8:962–969, 1996.

[13] Mohammed J. Zaki et al. Parallel data mining for association rules on shared-memory multiprocessors. Technical Report TR618, 1996.

[14] Jong Soo Park et al. Efficient parallel data mining for association rules. In *Proc. of the fourth intl. conf. on Information and knowledge management*, pages 31–36. ACM Press, 1995.

[15] David Wai-Lok Cheung et al. Asynchronous parallel algorithm for mining association rules on a shared-memory multipprocessors. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 279–288, 1998.