Uma Estratégia Eficiente para Balanceamento de Carga em Algoritmos de Mineração de Conjuntos Freqüentes

Fernando Mourão Luciano Lanna Adriano Veloso
Wagner Meira Jr. Renato Ferreira Dorgival Guedes
Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade Federal de Minas Gerais

Resumo

Mineração de dados compreende um conjunto de técnicas para extração automática de informações a partir de grandes bases de dados, atividade cada vez mais fundamental tendo em visto o acúmulo contínuo e crescente de dados e uma carência crescente de extrair informações úteis a partir desses dados. Uma estratégia comum para atender essa demanda é a paralelização dos algoritmos, os quais, tendo em vista a natureza irregular de muitas das técnicas de mineração de dados, tendem a apresentar significativo desbalanceamento de carga. Neste artigo avaliamos algumas estratégias de balanceamento de carga para algoritmos paralelos de mineração de dados e discutimos os compromissos entre balanceamento estático e dinâmico. Também propomos e avaliamos uma nova técnica de balanceamento de carga que leva em consideração as peculiaridades da classe de algoritmos e que se mostrou mais eficiente que as outras técnicas discutidas, a despeito das características das bases de dados sendo mineradas.

1 Introdução

A busca por informações implícitas em grandes massas de dados tem se tornado cada vez mais importante. Áreas como a Bioinformática, Economia, Sociologia, Astrofísica, dentre outras, encontraram na aplicação desse tipo de busca uma importante fonte de informações algumas vezes preponderantes. Essa busca por informações em grandes massas de dados é conhecida como Mineração de Dados.

A aplicação de técnicas de mineração de dados a bases de dados reais, como as descritas acima, é geralmente um processo extremamente custoso do ponto de vista computacional, uma vez que a quantidade de dados envolvida na mineração em geral é muito grande, tende a crescer, e os algoritmos freqüentemente tem complexidade exponencial. A utilização de soluções paralelas e distribuídas consolida-se como crucial para a viabilidade da execução desses algoritmos. Mais ainda, esses algoritmos não somente demandam muitos recursos computacionais, mas também são um desafio para a extração de paralelismo, pois são geralmente irregulares e intensivos em termos de E/S. Um algoritmo ser irregular significa que o seu comportamento, e portanto custo, depende da natureza da entrada. O fato de que os programas são intensivos em termos de E/S, faz com que o desempenho seja bastante afetado pelos componentes do sistema e pela quantidade de sobreposição entre o processamento e a comunicação atingida durante a execução do programa.

Em termos de paralelismo, a irregularidade em geral resulta em desbalanceamento de carga, ou seja, alguns processadores são mais demandados que outros, reduzindo a eficiência da paralelização e aumentando o tempo de execução da aplicação como um todo. Desbalanceamento de carga ocorre quando os dados não estão distribuídos uniformemente entre os nós, ou quando os dados relevantes a uma aplicação estão localizados em apenas um subconjunto dos processadores. Estratégias de balanceamento de carga melhoram o desempenho de aplicações paralelas no sentido de equalizar a carga que cada processador recebe. Existem, basicamente, dois tipos de estratégias de balanceamento de carga: estática e a dinâmica. Estratégias estáticas distribuem as tarefas anteriormente à execução, enquanto estratégias dinâmicas buscam o balanceamento em tempo de execução.

Neste artigo, apresentamos um estudo mais detalhado sobre a questão que envolve o balanceamento de carga em algoritmos de mineração de dados implementados em sistemas distribuídos. Analisaremos, conseqüentemente, o impacto de se preterir um balanceamento estático ao dinâmico, bem como descrever as situações onde aquele consegue satisfatoriamente se aproximar dos resultados obtidos por este. Sabe-se que na grande maioria das situações em am-

bientes distribuídos, o balanceamento dinâmico proporciona uma divisão de trabalho mais homogênea entre os processadores. No entanto, para a implementação desse tipo de balanceamento é necessária uma intensa comunicação entre os processadores. Em alguns casos, esse excesso de comunicação pode comprometer o ganho obtido por uma divisão homogênea de trabalho. Estudaremos também várias formas de se implementar esse tipo de balanceamento, bem como a utilização de heurísticas capazes de proporcionar uma melhor divisão de tarefas entre os processadores. Além disso, apresentaremos uma nova técnica de balanceamento estático baseada num estudo probabilístico das bases de dados.

2 Mineração de Dados

Um importante problema em Mineração de Dados é encontrar padrões presentes em grandes bases de dados. Tais padrões podem ser utilizados de diversas formas, mas eles vêm sendo cada vez mais utilizados para a geração de regras de associação, onde, dado um conjunto de itens (ou atributos), deve-se prever a ocorrência de outro conjunto de itens com um certo grau de confiança. Devido a esta crescente importância das regras de associação, e à necessidade de reduzir a vasta gama de possibilidades de estudo, optamos como caso de estudo para este artigo o balanceamento de carga neste subgrupo de algoritmos, em particular, técnicas e algoritmos para encontrar padrões ou *itemsets* freqüentes [1]. A seguir apresentamos uma formalização deste problema.

Seja $I = \{I_1, I_2, ..., I_m\}$ o conjunto de m itens. Seja T o conjunto de transações onde cada uma delas é um subconjunto de I e unicamente identificada por um *tid*. A base de dados é constituída pelo conjunto T, e dizemos que T é denso caso o tamanho de cada transação e os itens que a compõem são semelhantes. Seja C, um subconjunto de I, chamado *itemset*. Se C possuir k itens ele é chamado de k-itemset. Define-se o suporte de um *itemset* C como o número (ou a fração) de transações em que C ocorre no conjunto de transações T. Para reduzir o espaço de busca, define-se que os *itemsets* interessantes são aqueles que ocorrem um número mínimo de vezes em T. Assim, os algoritmos descartam qualquer *itemset* que tenha suporte menor que um suporte mínimo especificado pelo usuário.

A seguir apresentamos o algoritmo Eclat [9], que é o algoritmo seqüencial que serve como ponto de partida do nosso trabalho. Eclat (*Equivalence Class Transformation*) combina a busca em profundidade com intersecções otimizadas entre conjuntos, chamadas de "intersecções rápidas". Para isso, é utilizada um representação da base de dados, projeção vertical, na qual cada itemset possui uma *tidlist* - lista dos identificadores de todas as transações que contém o item. Além disso, é realizada também um agrupamento dos itemsets nas chamadas classes de equivalências afim de obter um particionamento da base de dados que seja compatível com o algoritmo e maximize a sua localidade de referência. Dessa forma, os *itemsets* freqüentes são gerados concatenando-se todos os itemsets de uma classe de equivalência através das intersecções das *tidlists*. Detalharemos essas etapas nas próximas seções.

Projeção Vertical da Base de Dados

A base de dados de exemplo na Tabela 1 está no formato horizontal, com cada tid seguido pelos itens que formam a transação. Esse formato de base de dados impõe algumas restrições computacionais durante a fase de contagem do suporte dos *itemsets*, uma vez que ele nos força a acessar a base de dados pelo menos uma vez a cada iteração.

Eclat usa uma projeção vertical, que consiste de uma lista de *itemsets*, e cada *itemset* possui sua *tidlist* - uma lista de todos os identificadores das transações onde esse *itemset* ocorreu na base de dados. O formato vertical da base de dados possui três principais vantagens comparado ao formato horizontal. Primeiro, um k-itemset pode ser encontrado simplesmente unindo os itens e realizando a intersecção das *tidlists* de quaisquer dois de seus $(k-1)$-subconjuntos. Por exemplo, se unirmos os itens e interseccionarmos as *tidlists* de $\{A, B\}$ e $\{A, E\}$, nós obtemos o *itemset* $\{A, B, E\}$, o qual aparece somente na quarta transação da base de dados da Tabela 1. Segundo, as *tidlists* contêm toda a informação relevante sobre um *itemset*, e nos possibilita evitar o acesso a toda a base de dados para computar o suporte de um *itemset*. Terceiro, quanto maior for o *itemset*, menor será sua *tidlist*, fato que acontece praticamente sempre, e que resulta em intersecções cada vez mais rápidas.

Classes de Equivalência

Um problema que afeta algoritmos para determinação de conjuntos freqüentes é a quantidade de memória que eles utilizam, que é, em geral, limitada, impedindo que todos os itemsets sejam armazenados simultaneamente. Essa questão revela a necessidade de decompor o espaço de busca original em partições menores de tal forma que cada partição possa ser processada independentemente.

TID	Itens
1	B, C, E
2	A, B
3	A, C, D, E
4	A, B, D, E
5	C, D, E

Tabela 1. Base de Dados com as Transações

A seguir descrevemos como podemos realizar essa

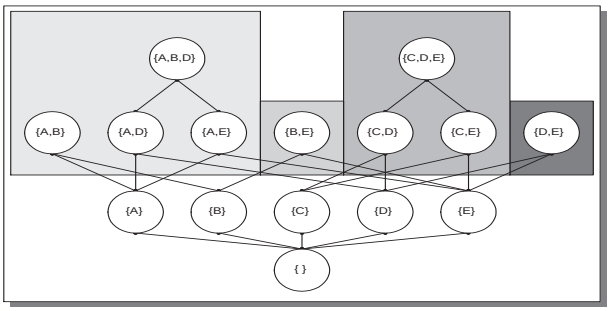

Figura 1. Decomposição do espaço de busca usando classes de equivalência com prefixo de tamanho 1

decomposição utilizando a base de exemplo apresentada na Tabela 1. Suponha que os itens em um *itemset* estejam lexicograficamente ordenados. Dizemos que dois *itemsets* pertencem a uma mesma classe de equivalência, F_k, se eles compartilham um prefixo de tamanho k, onde o tamanho é o número de itens que compõem o prefixo. A Figura 1 mostra quatro classes de equivalência com prefixo de tamanho 1: $\{A\}$, $\{B\}$, $\{C\}$ e $\{D\}$. Por exemplo, os *itemsets* $\{A, B\}$, $\{A, D\}$ e $\{A, E\}$ pertencem a uma mesma classe de equivalência, já que compartilham um prefixo de tamanho 1 (i.e., A). Ao final de cada iteração, uma classe de equivalência pode ser dividida em sub-classes de tamanho menor (i.e., no final da iteração k, um classe de equivalência pode ser quebrada em sub-classes com prefixos de tamanho $k-1$). A motivação para essa definição é que cada classe de equivalência pode ser processada independentemente, de tal forma que os *itemsets* produzidos por uma classe de equivalência são independentes dos produzidos por outra classe qualquer.

Busca por *Itemsets* Freqüentes

Eclat usa um esquema de busca *bottom-up* para enumerar todos os *itemsets* freqüentes. O ponto de partida são os itens freqüentes, os quais são determinados com apenas um acesso à base de dados. Para cada item encontrado na transação, atualiza-se sua *tidlist* e incrementa-se seu suporte. Os itens são usados para gerar os candidatos de tamanho 2. Os candidatos de tamanho 2 que forem infreqüentes são descartados e todos os $2-itemsets$ freqüentes são usados para criar as classes de equivalência com prefixo de tamanho 1. Partindo de uma classe de equivalência, somos capazes de enumerar todos os *itemsets* freqüentes com o prefixo que forma a classe de equivalência. Então, para cada classe de equivalência criada, Eclat gera os candidatos de tamanho 3. Na próxima iteração, os candidatos classificados como freqüentes são usados para determinar o próximo nível de *itemsets* freqüentes (i.e., os $(k-1)$-itemsets freqüentes são usados para determinar os k-itemsets freqüentes). O processo termina quando não existem mais candidatos, e todos os *itemsets* freqüentes foram encontrados.

3 Algoritmo Paralelo

Encontrar e contar o suporte dos *itemsets* freqüentes é uma tarefa intensiva do ponto de vista computacional, e, a partir de um certo tamanho da base de dados, torna-se crucial empregar o poder computacional combinado de vários processadores para uma rápida resposta e escalabilidade. Nessa seção apresentamos um algoritmo paralelo para a descoberta dos *itemsets* freqüentes. O algoritmo paralelo é baseado nos conceitos apresentados na seção anterior. Ele usa a decomposição do espaço de busca e a projeção vertical da base de dados. A carga de trabalho é distribuída entre os processadores de forma que cada processador possa determinar os *itemsets* freqüentes independentemente, utilizando simples intersecções de *tidlist*. Nosso algoritmo acessa a base de dados apenas uma vez, diminuindo drasticamente os custos impostos pela contenção na entrada e saída. Após uma fase de inicialização para a divisão estática de carga, ele não necessita de nenhuma sincronização. A seguir descrevemos cada etapa do algoritmo mais detalhadamente.

A fase de inicialização consiste de três etapas. Primeiro, a base de dados é acessada, todos os itens freqüentes são encontrados e suas $tidlists$ são criadas. Segundo, os 2-$itemsets$ são computados realizando intersecções entre as $tidlists$ dos itens freqüentes. Terceiro, os 2-$itemsets$ são agrupados em classes distintas aplicando-se a decomposição por classes de equivalência, e o conjunto de classes independentes é criado. Neste momento é que as estratégias de balanceamento de carga definem quais processadores vão processar quais classes, ou seja, cada estratégia gera um particionamento das classes de forma a assinalar um sub-conjunto a cada processador. Em termos de implementação, todos os processadores realizam a fase de inicialização, apenas para evitar que as *tidlists* tenham que ser transferidas entre eles.

Ao final da fase de inicialização, as classes de equivalência relevantes estão disponíveis localmente em cada processador. Agora, cada um deles pode gerar independentemente todos os $itemsets$ freqüentes provenientes das classes que foram atribuídas a ele. Cada classe é totalmente processada antes de passar para a próxima classe. Em um esquema estático de balanceamento de carga, nenhuma migração de carga para corrigir o desbalanceamento é permitido. O balanceamento de carga dinâmico procura resolver esse problema distribuindo a carga pertencente a processadores duramente carregados para os processadores menos carregados no momento. No entanto, obser-

vamos que a movimentação de computação implica em movimentação de dados, uma vez que o processador responsável por uma determinada tarefa necessita dos dados associados a ela. Por isso, essa forma de balanceamento dinâmico de carga implica em custos adicionais de trabalho e movimentação de dados, que num sistema de memória distribuída pode causar grande degradação devido ao custo de comunicação. Assim, o balanceamento de carga é um fator crucial para o desempenho em tarefas com grande desbalanceamento. Portanto, o principal desafio que procuramos resolver é a investigação de uma forma de balanceamento estático de carga que não necessite de movimentação de dados e comunicação entre os processadores. Várias técnicas são discutidas na próxima seção.

4 Estratégias de Balanceamento de Carga

Nesta seção apresentamos algumas estratégias de balanceamento de carga estático. Essas estratégias fazem uso de heurísticas com o objetivo de aumentar a qualidade de balanceamento.

Divisão Modularizada

Uma das formas mais simples de realizar o balanceamento estático é através da divisão modularizada, ou seja, agrupamos os itens da base de dados em classes de equivalências correspondentes às *n-1* classes geradas por um modularização com valor de módulo *n* . O valor do módulo a ser considerado é igual ao número de processadores existentes durante a execução do algoritmo. Dessa forma, cada processador analisa todos os itens de somente uma única classe de equivalência dessa modularização. Além da facilidade de implementação deste método, ele possui a vantagem de não adicionar nenhum trabalho extra, visto que a única operação a ser feita para obter o próximo item de uma mesma classe seria somar ao item atual o valor do módulo. Embora este método não consiga distribuir o trabalho de forma muito homogênea, ele seria particularmente adequado para certos tipos de bases densas, ou seja, com muitos itemsets freqüentes e com itemsets não muito longos, pois estas são geralmente mais homogêneas e assim não teria como haver um grande desbalanceamento.

Divisão Bitônica

Podemos empregar um processo mais complexo que envolve uma atribuição de pesos às classes. Este peso está associado com o número de *2-itemsets* de cada item. Ou seja, primeiro calculamos todos os *2-itemsets* freqüentes. A partir desses dados, podemos ordenar os *1-itemsets* por ordem decrescente de peso. Com os itens ordenados aplicamos uma divisão bitônica [5] dos itens, ou seja, atribuímos o primeiro e o último termos livres do vetor ordenado a um processador qualquer. Este processo se repete até que todos os itens sejam distribuídos entre os processadores. Um obstáculo para a implementação dessa técnica é a utilização de memória. A utilização dessa técnica para bases de dados com muitos itens distintos torna-se impraticável pois, seria necessário criar todos os itens de tamanho 2 a partir dos de tamanho 1 para gerar os pesos. Para um número grande de itens freqüentes de tamanho 1 teríamos um número equivalente à combinação de todos os itens tomados 2 a 2. Mesmo alocando espaço para apenas uma classe de item por vez, alocar memória necessária para armazenar tal quantidade de informação seria impraticável.

Divisão Gulosa

Uma heurística que podemos utilizar para aumentar o balanceamento é a utilização de uma estratégia gulosa sobre os itens ordenados por peso a fim de se aproximar da solução ótima. Como o balanceamento perfeito dos itens entre os processadores é um problema combinatório, tratando-se portanto de um problema exponencial, a utilização de uma estratégia gulosa é particularmente interessante pois se aproxima da solução ótima em um tempo polinomial.

A estratégia gulosa utilizada funciona da seguinte forma, primeiramente calculamos o número de *2-itemsets* que cada *1-itemset* produz. Paralelamente, contabilizamos o número de *2-itemsets* globalmente gerados. Em seguida ordena-se cada *1-itemset* decrescentemente pelo número de *2-itemsets* encontrados e calculamos o número médio de *2-itemsets* que cada processador deveria processar. Posteriormente, atribuímos a cada processador a classe correspondente o *1-itemset* com maior número de *2-itemsets* que ainda não foi atribuído a nenhum processador. Este valor é computado com vistas ao número de *2-itemsets* que cada processador recebe seja o mais próximo possível da média calculada.

Dessa forma, podemos satisfatoriamente balancear a carga de trabalho. Uma desvantagem desse método é a quantidade de trabalho necessário para se aproximar da solução ótima, muito superior aos dos outros métodos mencionados. É importante lembrar que essa técnica é apenas uma heurística de otimização. Mesmo com a distribuição perfeita dos itens entre os processadores não temos garantia de um bom balanceamento pois, a quantidade de *2-itemsets* freqüentes que cada item pode gerar pode variar muito entre os itens, e a determinação dessa diferença só é possível após o processo de mineração em si. Essa técnica possui o mesmo problema de alocação de memória descrito para a técnica de divisão bitônica.

Divisão por Classificação Estimada

Considerando apenas a freqüência com que cada item ocorre e o tamanho médio das transações em que estes apa-

recem, podemos obter importantes informações. Algumas dessas informações não são utilizadas por nenhuma técnica de balanceamento proposta na literatura. Dessa forma, propomos neste artigo uma nova técnica que explora as vantagens oferecidas por essas características na tentativa de proporcionar um balanceamento mais homogêneo. Este balanceamento classifica os itens de acordo com a quantidade de trabalho estimada que cada item possivelmente irá gerar. Como mencionado este método se baseia na análise de duas características observadas em bases de dados: a freqüência com que cada item aparece na base de dados (i.e., seu suporte) e o tamanho médio das transações. Como estamos tratando de uma medida baseada em um média comportamental, devemos ter, para fins de completude, uma terceira métrica estatística: o desvio padrão. O desvio padrão nos informa qual a variação, em módulo, da média encontrada. Utilizando essas três métricas, propomos a classificação dos itens em oito classes descritas a seguir:

Classe 1: Desvio padrão pequeno, alta freqüência e tamanho médio das transações grande. Há uma maior probabilidade de gerar mais trabalho, pois possivelmente gera um número grande de *2-itemsets* e um número maior de *itemsets* mais longos. Essa análise se verifica mais forte pela presença de um desvio padrão pequeno.
Classe 2: Desvio padrão pequeno, alta freqüência, e tamanho médio das transações pequeno. Possivelmente possui uma grande quantidade de *2-itemsets*, no entanto poucos *itemsets* de maior ordem.
Classe 3: Desvio padrão pequeno, baixa freqüência e tamanho médio das transações é grande. Possui poucos *2-itemsets* freqüentes, mas, possivelmente, alguns *itemsets* de maior ordem freqüentes.
Classe 4: Desvio padrão pequeno, baixa freqüência e tamanho médio das transações pequeno. Possui poucos *itemsets* freqüentes. Provavelmente é a classe que gera menos trabalho. Essa análise se verifica mais forte pela presença de um desvio pequeno.
Classe 5: Desvio padrão grande, alta freqüência e tamanho médio das transações grande. Provavelmente este grupo se aproxima das características presentes na classe 1, no entanto a medida do desvio padrão do tamanho médio sugere um número menor de *itemsets* freqüentes de maior ordem.
Classe 6: Desvio padrão grande, alta freqüência e tamanho médio das transações pequeno. Esta classe possui características análogas as apresentadas pela classe 2, no entanto a medida do desvio padrão do tamanho médio sugere um número menor de *itemsets* freqüentes de maior ordem.
Classe 7: Desvio padrão grande, baixa freqüência e tamanho médio das transações grande. Esta classe possui características análogas as apresentadas pela classe 3, no entanto a medida do desvio padrão do tamanho médio sugere um número menor de *itemsets* freqüentes de maior ordem.
Classe 8: Desvio padrão grande, baixa freqüência e tamanho médio das transações pequeno. Além de possuir um dos menores números de *itemsets* freqüentes, possuem também *itemsets* menores.

Sup.	#Procs	Gul.	Mod.	Bit.	Class.	Din.
0,015%	2	6314	3148	3136	2796	2799
	4	6147	1574	1571	1405	1550
	8	6134	792	792	707	1004
	16	6104	400	402	357	400
0,020%	2	1548	1151	777	709	789
	4	1537	395	392	360	398
	8	1537	201	206	183	267
	16	1538	104	104	95	118
0,025%	2	372	189	186	174	185
	4	372	98	96	91	130
	8	371	52	52	49	74
	16	371	29	29	28	37

Tabela 2. Tempos de execução para a Base 1

Sup.	#Procs	Gul.	Mod.	Bit.	Class.	Din.
7,5%	2	350.28	174,36	217.5	172.32	181.90
	4	268.04	102.12	112.4	100.8	111.90
	8	198.33	51.11	59.7	52.8	61.45
	16	163.43	28.92	42.4	29.55	42.57
10%	2	261.18	105.76	117.2	98.53	109.22
	4	179.76	52.57	70.3	52.72	62.19
	8	149.47	29.59	40.8	29.52	40.22
	16	129.80	19.24	31.5	18.31	27.60
15%	2	149.48	49.46	-	49.6	54.53
	4	103.28	28.42	-	28.39	39.59
	8	83.80	17.73	-	17.68	27.38
	16	80.32	13.19	-	12.38	19.80

Tabela 3. Tempos de execução para a Base 2

Para obter um bom balanceamento de tarefas é necessário dividirmos igualmente cada classe entre os processadores. Assim, estamos dividindo as cargas de trabalho estimadas para cada classe igualmente entre os processadores e conseqüentemente a carga de trabalho global estimada será também distribuída de forma mais homogênea. É importante salientar que a técnica descrita acima é apenas uma heurística baseada em conceitos probabilísticos, não sendo possível garantir bons resultados de balanceamento visto que o número de itens pode não estar de acordo com os padrões estimados, *itemsets* cuja classificação não condiz com a quantidade de trabalho gerado, se torna grande, degenerando portanto a divisão efetuada por tal método.

5 Avaliação Experimental

Nesta seção nós apresentamos resultados experimentais dos algoritmos de balanceamento descritos anteriormente. Analisamos nossos algoritmos em diferentes tamanhos de bases, porcentagem de itens freqüentes por base e quantidade de itens distintos. As bases utilizadas são sintéticas,

contém 1.600.000 transações e os seus tamanhos variam de 80MB a 120MB. As bases, que denominamos 1, 2 e 3, se diferenciam em termos de itens distintos (127, 159902 e 1743312) e tamanho médio das transações (20, 10 e 74, respectivamente). Cabe salientar que embora as bases usadas como testes caibam na memória principal, a grande dificuldade para estes algoritmos é a quantidade de dados gerada por eles durante suas execuções. Para bases de tamanho moderado, como as apresentadas, a quantidade de dados gerada é muito maior que a memória principal das máquinas de teste. Tais testes foram realizados em máquinas de 1 GB de memória RAM, 1 GB de swap, velocidade de processamento de 2 GHz e 80 GB de memória secundária.

5.1 O ambiente de programação Anthill

Nesta seção descrevemos o ambiente de programação Anthill, utilizado para a execução da avaliação experimental das estratégias de balanceamento de carga. O Anthill (Formigueiro) é o nosso ambiente para desenvolvimento e execução de aplicações distribuídas escaláveis. O ambiente foi desenvolvido tendo em mente aplicações paralelas não regulares, intensivas em processamento e em entrada-e-saída de dados (E/S). Nessas aplicações, que manipulam grandes volumes de dados, estes se encontram distribuídos em várias máquinas do sistema. Mover os dados para outros nós para então serem processados é freqüentemente uma operação ineficiente. Isso é verdade especialmente porque à medida que o processamento avança, os dados resultantes tendem a ser muitas vezes menores que os dados de entrada. Dessa forma, uma alternativa interessante é levar a computação aonde o dado está, reduzindo a comunicação através da rede. O sucesso desse enfoque depende da facilidade com que a aplicação possa ser dividida em etapas que sejam passíveis de execução em nós diferentes do sistema. Cada etapa dessa forma executará parte das transformações sobre os dados, iniciando com o conjunto de dados de entrada, até que se atinja o conjunto de dados de saída.

Essa discussão indica que uma boa paralelização de qualquer aplicação nesse contexto deve considerar simultaneamente tanto paralelismo de dados quanto de tarefas. A estratégia do Anthill aplica os dois enfoques, agregando uma terceira dimensão que nos permite explorar o grau de assincronia existente entre diferentes tarefas independentes no sistema ao longo do tempo. Os benefícios dessas três dimensões combinadas nos permitem atingir *speedups* elevados experimentalmente [6].

O modelo de programação do Anthill é denominado *filter stream*. Nesse modelo, o processamento é dividido em tarefas que operam sobre os dados que fluem pelo sistema. Cada filtro implementa uma tarefa que transforma os dados segundo a necessidade da aplicação e se comunicam com outros filtros pelos canais de comunicação responsáveis pela transmissão contínua de dados (*streams* ou fluxos). Essas duas abstrações podem ser combinadas formando grafos arbitrários que representem o processamento da aplicação.

Usando esse modelo, criar uma aplicação no *Anthill* é um processo de decomposição do processamento em filtros. Nesse processo, a aplicação é modelada como uma computação no modelo *dataflow* dividida em uma rede de filtros que transformam os dados. Durante a execução, o processo definido para cada filtro é instanciado em diferentes nós do ambiente distribuído. A esses processos dá-se o nome de cópias transparentes ou instâncias de um filtro. Dessa forma, cada estágio do processamento pode ser distribuído por muitos nós de uma máquina paralela e os dados que devem fluir por aquele filtro podem ser particionados pelas cópias transparentes, produzindo o paralelismo de dados desejado.

Nosso modelo de programação, dessa forma, nos permite extrair o máximo de paralelismo das aplicações através das três possibilidades discutidas anteriormente: paralelismo de dados, de tarefas e assincronia. Já que as unidades de processamento são na verdade cópias de estágios de um *pipeline*, pode-se ter um paralelismo de grão fino. Como todo esse processamento pode ocorrer assincronamente, a execução não terá qualquer tipo de retenção (*bottleneck*) devida ao sistema. Para garantir a redução da latência durante o processamento, a granulosidade das tarefas deve ser definida pelo projetista da aplicação.

5.2 Resultados

Os resultados da utilização das várias estratégias de balanceamento de carga são apresentados nas Tabelas 2 e 3. Essas tabelas apresentam os tempos de execução em segundos para vários números de processadores utilizados e valores do parâmetro de execução suporte.

Os resultados obtidos confirmam que apesar de uma estratégia gulosa balancear de forma bastante homogênea, sua utilização é muito custosa. Assim, a utilização deste método se torna adequada para distribuir tarefas cujo processamento é muito custoso, pois dessa forma o custo de sua implementação seria amortizado por uma distribuição mais homogênea. As tarefas são mais custosas, principalmente, em dois tipos de bases em particular: bases grandes com muitos itens freqüentes e com uma freqüência alta, e bases pequenas ou médias mas densas com poucos itens freqüentes e com uma freqüência alta, devido ao tamanho médio das transações e do número de itens freqüentes nessas bases. A estratégia gulosa é interessante principalmente para este segundo tipo de bases pois possuem poucos itens freqüentes. A restrição de se ter poucos itemsets freqüentes é explicada pelo fato desse algoritmo, inicialmente, calcular todas as combinações de todos os itens freqüentes tomados

dois a dois. Logo, este método é indicado para bases reais com poucos itens freqüentes mas, com alta freqüência pois, em geral estas bases possuem um alto desbalanceamento. Uma característica deste método é que sua aplicação normalmente iguala o tempo de execução. O problema é que este valor se mostra, geralmente alto, pois ele é a soma do tempo necessário para gerar todas os *itemsets* de tamanho dois e o trabalho que cada filtro recebe após a distribuição de cargas. Outro fator é que independente do número de máquinas, em bases que possuem poucos itemsets freqüentes como a base 2, esse tempo se mantém quase o mesmo.

Para bases com uma grande quantidade de itens freqüentes, o número de $itemsets$ de tamanho 2 gerados ocupa em poucos instantes toda a memória disponível nas máquinas utilizadas. Essa característica é independente da quantidade de máquinas utilizadas pois todo o trabalho de inicialização é executado em todas as máquinas. Bases que possuem uma baixa distorção, ou seja, que possuem muitos itens freqüentes concentrados a partir de uma pequena variação de suporte são difíceis de serem analisadas por este método. Para bases sintéticas a estratégia gulosa nunca supera o desempenho obtido pela divisão bitônica e principalmente o desempenho obtido pela modularização e pelo balanceamento por classificação estimada, como discutiremos a seguir.

Os resultados também mostram que, embora a divisão modularizada leve quase sempre ao pior balanceamento, esta estratégia teve bons resultados para a maioria das configurações, visto que não há custos adicionais para a sua implementação. Assim para bases de dados mais homogêneas, com um desbalanceamento mais suave, e para bases mais esparsas com muitos itens com baixa freqüência, o balanceamento estático implementado dessa forma se mostra muito bom. Como podemos perceber a partir dos resultados, este algoritmo possui uma boa escalabilidade.

Analisando os resultados obtidos para o balanceamento por classificação estimada, vemos que independente do número de processadores utilizado, suporte considerado ou base analisada este método conseguiu obter um desempenho superior aos demais. A explicação é que este método realiza, na maior parte dos casos, um bom balanceamento aliado a um custo razoavelmente baixo. O algoritmo percorre a base de dados classificando seus elementos de tamanho 1 freqüentes e posteriormente distribui os elementos de cada classe entre os processadores. O custo dessa análise é fortemente amortizado por uma boa distribuição de cargas. Dessa forma percebemos que este método é o ideal a ser aplicado para a maioria dos casos, principalmente em bases esparsas, ou seja, com poucos itens freqüentes, e bases com poucos elementos distintos mas com uma freqüência alta, pois assim teríamos poucos elementos para serem classificados mas que possivelmente geram um grande desbalanceamento. Entretanto, caso a base de dados analisada possua muitos elementos e a quantidade de trabalho gerada por todos os elementos seja muito próxima, este algoritmo não será o mais indicado.

Objetivando um estudo mais minucioso, comparamos também todas as técnicas de balanceamento estático com uma técnica simples de balanceamento dinâmico baseada no padrão de algoritmo mestre/escravos. Ou seja, criam-se dois tipos de filtros de processamento, um primeiro, chamado de mestre, responsável apenas pela distribuição de itemsets de tamanho 1 entre os outros nodos e, os outros, escravos, responsáveis pela geração dos itemsets restantes. Dessa forma, um escravo solicita o próximo itemset que ele deve processar ao mestre que, por sua vez, envia o próximo itemset a ser processado. Quando não houver mais itemsets a serem processados, o mestre envia uma mensagem a todos os escravos informando fim de processamento.

Analisando o comportamento desta implementação e comparando-a às demais técnicas estáticas discutidas, podemos concluir que este balanceamento é o mais homogêneo de todos, conforme mostram os gráficos da Figura 2. No entanto, este ganho é deteriorado devido ao overhead de comunicação necessário entre os nodos. Comparando-o com o desempenho das técnicas estáticas, percebemos que o tempo de execução da divisão por classificação estimada, mais um vez, se mostra melhor. Embora, a divisão de trabalho do balanceamento dinâmico seja melhor que a apresentada pela técnica de classificação estimada, esta melhora, em grande parte das vezes, não é suficiente para superar os custos de comunicação.

É importante salientar que tais análises foram realizadas considerando um ambiente dedicado, tal como um cluster. Evidentemente que em ambientes heterogêneos, onde não se pode prever o nível de utilização das máquinas para tarefas concorrentes, o balanceamento dinâmico tende a ser a técnica ideal. O balanceamento estático parte do princípio que todas as máquinas possuem a mesma capacidade de processamento sempre. Isto certamente não ocorre em ambientes heterogêneos.

6 Trabalhos Relacionados

Nesta seção provemos uma visão geral dos algoritmos paralelos e distribuídos já desenvolvidos para minerar *itemsets* freqüentes. Começamos pelo mais conhecido algoritmo, chamado Apriori [3]. Em [2] foi proposta a paralelização desse algoritmo, onde os autores propuseram uma abordagem paralela com balanceamento dinâmico, mas a alta taxa de comunicação realizada não permitiu a obtenção de resultados satisfatórios. Em [7] foi proposta a utilização de agregação de memória, aumentando um pouco a eficiência e diminuindo a taxa de comunicação. Em [4] os autores apresentaram um trabalho interessante, onde propuseram técnicas de balanceamento de carga baseadas no particionamento da base de dados. A forma como os dados

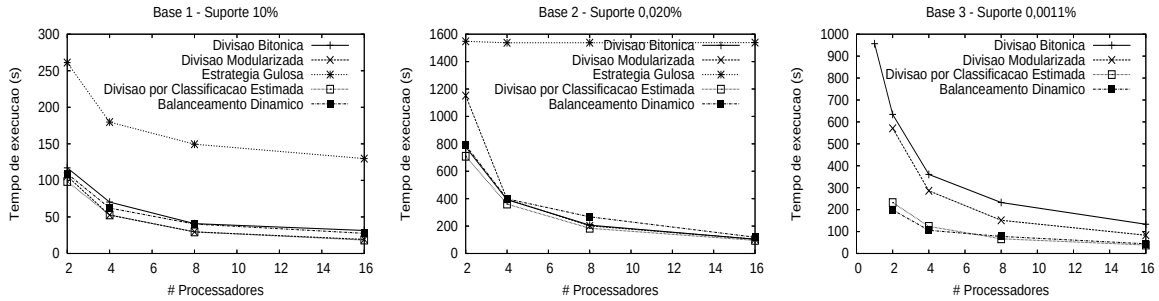

Figura 2. Comparação entre as técnicas

devem ser particionados é função da entropia observada em cada partição. Os resultados da paralelização do algoritmo Apriori apresentados nesse trabalho foram muito bons. Infelizmente não podemos compará-los com os nossos, uma vez que os algoritmos seqüenciais empregados por cada um, são diferentes. Tanto em [10] quanto em [8] os autores propuseram paralelizações do algoritmo Eclat. O algoritmo paralelo apresentou bons resultados, mas com as técnicas de balanceamento propostas neste artigo conseguimos resultados melhores.

7 Conclusões e Trabalhos Futuros

Neste artigo apresentamos um estudo minucioso dos efeitos do balanceamento de carga no desempenho de algoritmos para mineração de dados. Analisamos empiricamente que o balanceamento de carga é completamente dependente das características de cada tipo de base de dados. A partir desse estudo, propomos uma nova estratégia de balanceamento de carga, que é mais atraente seja pela melhor utilização de memória ou desempenho, de acordo com as características inerentes aos dados.

Dessa forma, concluímos que, não diferentemente de diversas aplicações computacionais, para se ter um bom desempenho em algoritmos de mineração de dados, tão importante quanto implementar um algoritmo eficiente, é conhecer as características dos dados a serem mineradas. Concluímos que na maior parte dos casos o balanceamento estático por classificação estimada se mostra superior a formas de balanceamento estático tais como por divisão modularizada, a formas mais complexas tais como aplicação de uma heurística gulosa ou divisão bitônica. E em alguns casos, podemos obter um desempenho melhor que o encontrado em uma implementação simples e eficiente do balanceamento dinâmico. Mostramos que em ambientes dedicados, a técnica de balanceamento estático proposta é mais eficiente que todas as outras estudadas.

Em termos de trabalhos futuros, temos a aplicação dos conceitos propostos a outros algoritmos de mineração de dados e a comparação utilizando tanto bases de dados reais quanto outras estratégias de balanceamento de carga dinâmico.

Referências

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of the Int. Conf. on Management of Data, SIGMOD*, pages 207–216, Washington, USA, May 1993. ACM.

[2] R. Agrawal and J. Shafer. Parallel mining of association rules. *Transactions on Knowledge and Data Engineering*, 8(6):962–969, 1996.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the Int. Conf. on Very Large Databases, VLDB*, pages 487–499, SanTiago, Chile, June 1994. VLDB.

[4] D. Cheung and Y. Xiao. Effect of data distribution in parallel mining of associations. *Data Mining and Knowledge Discovery*, 3(3):291–314, 1999.

[5] M. Cierniak, M. Zaki, and W. Li. Compile-time scheduling algorithms for a heterogeneous network of workstations. *The Computer Journal*, 40(6):356–372, December 1997.

[6] R. Ferreira, W. M. Jr., D. Guedes, L. Drummond, B. Coutinho, G. Teodoro, T. T. andR. Araújo, and G. Ferreira. Anthill: A scalable run-time environment for data mining applications. In *Proc. SBAC-PAD*, pages 159–167, Rio de Janeiro, Brazil, October 2005. IEEE.

[7] E. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. *Transactions on Knowledge and Data Engineering*, 12(3):728–737, 2000.

[8] M. Zaki and S. Parthasarathy. A localized algorithm for parallel association mining. In *Proc. Symp. on Parallel Algorithms and Applications, SPAA*, pages 120–128. ACM, Aug 1997.

[9] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proc. of the Int. Conf. on Knowledge Discovery and Data Mining, SIGKDD*, pages 283–290. ACM, August 1997.

[10] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery*, 4(1):343–373, December 1997.