Revista de Sistemas de Informação da FSMA
n. 11 (2013) pp. 33-41

SISTEMAS DE INFORMAÇÃO
http://www.fsma.edu.br/si/sistemas.html

TRILHA PRINCIPAL

Uma Nova Proposta de Paralelismo e Balanceamento de Carga Para o Algoritmo Apriori

André Camilo Bolina, *Graduando em Ciência da Computação, UFLA*,
Denilson Alves Pereira, *Professor Doutor em Ciência da Computação, UFLA*,
Ahmed Ali Abdalla Esmin, *Professor Doutor em Engenharia Elétrica, UFLA* e
Marluce Rodrigues Pereira, *Professora Doutora em Engenharia de Sistemas de Computação, UFLA*

***Resumo*—O principal objetivo da mineração de dados é descobrir informações relevantes em conteúdos digitais. O algoritmo Apriori é amplamente utilizado para este objetivo, mas sua versão sequencial tem baixo desempenho quando executado para grandes volumes de dados. Entre as soluções para este problema encontra-se a implementação paralela do algoritmo, e entre as implementações paralelas apresentadas na literatura com base no Apriori, destaca-se o DPA (*Distributed Parallel Apriori*) [10]. Este trabalho apresenta o algoritmo DMTA (*Distributed Multithread Apriori*), que se baseia no DPA, mas explora também o paralelismo em nível de *threads*, a fim de aumentar o desempenho. Além disso, o DMTA pode ser executado em plataformas de hardware heterogêneo, com diferentes números de núcleos de processamento. Os resultados mostraram que o DMTA supera o DPA, apresenta o equilíbrio de carga entre processos e *threads*, e é eficaz nas atuais arquiteturas multicores.**

***Palavras-chave*—Apriori, Paralelização de Algoritmo, MPI, OpenMP, DPA.**

A New Approach of Parallelism and Load Balance for the Apriori Algorithm

***Abstract*—The main goal of data mining is to discover relevant information on digital content. The Apriori algorithm is widely used to this objective, but its sequential version has a low performance when executed over large volumes of data. Among the solutions for this problem is the parallel implementation of the algorithm, and among the parallel implementations presented in the literature that based on Apriori, it highlights the DPA (Distributed Parallel Apriori) [10]. This paper presents the DMTA (Distributed Multithread Apriori) algorithm, which is based on DPA and exploits the parallelism level of threads in order to increase the performance. Besides, DMTA can be executed over heterogeneous hardware platform, using different number of cores. The results showed that DMTA outperforms DPA, presents load balance among processes and threads, and it is effective in current multicore architectures.**

***Index Terms*—Apriori, Parallelization of Algorithm, MPI, OpenMP.**

Autor correspondente: Marluce Rodrigues Pereira, *marluce@dcc.ufla.br*

I. Introdução

EXTRAIR informações de dados é um desafio, principalmente nos dias de hoje, onde o volume destes é imenso. Este desafio inclui também descobrir padrões e informações significantes em dados dispersos, e para isso, as mais diferentes técnicas vêm sendo desenvolvidas e aperfeiçoadas na área de mineração de dados.

O algoritmo Apriori [1] é um dos algoritmos mais representativos na mineração de padrões frequentes (que apresentam suporte maior que um suporte mínimo estabelecido). A Figura 1 apresenta um fluxograma de sua execução. Sua idéia principal é baseada na observação de que se um conjunto de itens (*itemset*) é frequente seus subconjuntos também são frequentes. Em cada passo de execução são gerados os conjuntos de itens frequentes que posteriormente são utilizados para gerar as regras de associação da base de dados. Mesmo que o algoritmo Apriori possa eficientemente encontrar padrões frequentes, o seu tempo de execução aumenta com o aumento do número de transações na base de dados, isso devido a cada *itemset* candidato ser testado com toda a base de dados.

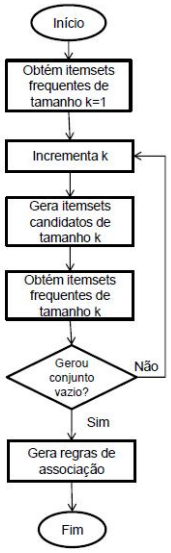

Fig. 1. Fluxograma de execução do Apriori Sequencial.

Existem também outros algoritmos representativos na mineração de padrões frequentes e cujas formas de trabalho diferem do Apriori, como o Eclat [18] e o FP-growth [8]. Estes algoritmos, além de suas implementações iniciais, possuem uma série de otimizações, como a implementação paralela proposta por [11], que utilizou a estrutura de dados FP-tree, mas encontrou problemas no balanceamento de carga. Além disto, outras pesquisas na área de mineração de padrões frequentes aplicaram técnicas de computação paralela e distribuída para efetivamente acelerarem o processo de mineração ([2], [5], [6], [17]).

Em ambientes distribuídos porém, a irregularidade e desequilíbrio nas cargas de computação dos processadores podem fazer com que o desempenho global seja fortemente degradado. Logo, o equilíbrio de carga entre os processadores no processo de mineração é muito importante para a mineração paralela e distribuída. No Apriori cada *itemset* candidato com o mesmo tamanho pode ser testado de forma independente contra a base de dados, permitindo que o algoritmo possa ser facilmente paralelizado de forma balanceada. Este fato faz com que sejam apresentadas diversas propostas de implementação paralela e distribuída para o Apriori ([7], [10], [12], [14], [15]). Desta forma, a escolha do algoritmo Apriori para este trabalho foi devido à sua ampla utilização na comunidade de mineração de dados e à sua facilidade de paralelização de forma balanceada.

Uma das técnicas para acelerar o processo de mineração está em projetar uma estrutura de dados mais adequada para armazenar os dados que o algoritmo vai gerando a cada *itemset* encontrado. Por exemplo, [17] propuseram um algoritmo paralelo distribuído baseado na árvore Trie [4]. Seu algoritmo distribui as cargas de trabalho de acordo com a árvore Trie para equilibrar e acelerar o processo de computação. Entretanto, os itens que são distribuídos para os nós baseiam-se apenas no primeiro nível da árvore Trie. Isso pode causar tamanhos significativamente variados de *itemsets* candidatos (cargas) entre os processadores. Além disso, esse método também requer que uma base de dados seja percorrida muitas vezes.

Foi proposta também uma forma eficiente de encontrar padrões frequentes por algoritmo de mineração baseado no algoritmo Apriori, chamado EDMA [16]. O algoritmo EDMA usa a estrutura de dados CMatrix para armazenar as operações de mineração. A estrutura de dados CMatrix representa a base de dados de transações como uma matriz binária onde as linhas representam as transações e as colunas representam os itens. Cada elemento a_{ij} da matriz é armazenado em 1 bit, que é setado para 1, se o item j ocorre na transação I, e setado para 0 caso contrário. O número de 1s por coluna corresponde ao valor do suporte por item. Essa forma de armazenar os dados dispensa o algoritmo de percorrer várias vezes a base de dados e minimizar o número de conjuntos candidatos, além de reduzir a troca de mensagens pela poda local e global. Uma vez que pode diminuir o tamanho médio das transações e bases de dados, o tempo de execução também pode ser reduzido. O tempo de execução, no entanto, fica maior quando o tamanho da base de dados é muito grande, uma vez que o EDMA acessará a estrutura CMatrix várias vezes para o cálculo dos *itemsets* frequentes.

O algoritmo *Distributed Parallel Apriori* (DPA) [10] armazena os dados em uma lista invertida, utilizando uma estrutura de tabela $Hash$, isto permitiu uma melhor paralelização e balanceamento de carga, além de alcançar resultados melhores do que os alcançados pelos demais algoritmos estudados, como [16] e [17]. Porém, existem oportunidades de otimizações de paralelismo e balanceamento de carga que são apresentadas neste trabalho.

Denominado *Distributed Multithread Apriori* (DMTA), o algoritmo proposto neste trabalho explora o paralelismo em nível de processos, como nos demais algoritmos, mas apresenta também um segundo nível de paralelismo, através da criação de *threads*. Isso permite uma melhor utilização das arquiteturas multicores atuais, pois utiliza a memória compartilhada entre os núcleos de processamento para compartilhar dados, evitando a troca explícita de mensagens. A memória compartilhada é gerenciada pelo sistema operacional e pela biblioteca de paralelização escolhida e, assim como na memória local, cria limites ao processo de mineração. Por isto são importantes para o algoritmo um maior nível de distribuição dos dados e uma boa estrutura para armazenamento de dados. No DMTA a estrutura de armazenamento de dados utilizada foi de lista invertida, assim como o DPA, porém no DMTA ela é formada pelo encadeamento de vetores da classe "vector", pré-definida na linguagem de programação C++, resultando em uma matriz tridimensional.

Quanto ao balanceamento, o DMTA realiza uma nova forma de divisão de carga entre os processos que gerou um melhor desempenho e equilíbrio entre os processadores. A criação de processos em processadores (núcleos de processamento) remotos (diferentes) é realizada utilizando-se a biblioteca MPI [13] e a criação de *threads* é feita utilizando-se a biblioteca OpenMP [3]. O número de *threads* a serem criadas em cada processo é definido em tempo de execução, de acordo com o número de núcleos de processamento da máquina onde o processo foi disparado.

Desta forma, as principais contribuições deste trabalho são:

1) explorar o paralelismo da aplicação para execução em ambientes multicores, realizando também, além da criação de processos, a criação de *threads*.
2) utilizar uma forma paralela de escalonamento de processos, visando diminuição no *overhead* e um bom balanceamento de carga entre os núcleos de processamento.
3) alterar a estrutura de armazenamento de dados para uma matriz tridimensional, formada pelo encadeamento de vetores de classe pré-definida pela linguagem utilizada e
4) reduzir o tempo de execução, permitindo suporte a bases de dados maiores.

O restante deste artigo está organizado da seguinte forma. Na Seção II são apresentadas uma contextualização e a descrição do problema de mineração de *itemsets*

frequentes ou padrões frequentes. Na Seção III é abordada a metodologia utilizada neste trabalho. A Seção IV descreve as características do algoritmo DPA e do algoritmo proposto DMTA. A Seção V mostra os resultados obtidos e a Seção VI apresenta as conclusões e trabalhos futuros.

II. Mineração de padrões frequentes

MINERAÇÃO de dados é um processo de descoberta de conhecimento em grande quantidade de dados e uma de suas principais tarefas é a descoberta de padrões frequentes. Esta tarefa consiste em encontrar subconjuntos frequentes em uma base de dados contendo conjuntos de itens.

A definição formal do problema de mineração de padrões frequentes, segundo [1], é como segue. Seja $I = \{i_1, i_2, ..., i_m\}$ um conjunto de itens. Seja D um conjunto de transações, onde cada transação T é um conjunto de itens tal que $T \subseteq I$. Associado a cada transação existe um único identificador, chamado TID. Dizemos que uma transação T contém X, um conjunto de alguns itens contidos em I, se $X \subseteq T$. Uma regra de associação é uma implicação da forma $X \Longrightarrow Y$, onde $X \subset I$, $Y \subset I$, e $X \cap Y = \emptyset$. A regra $X \Longrightarrow Y$ possui no conjunto de transações D a confiança c, de forma que c seja a porcentagem das transações em D que contêm X e também contém Y. A regra $X \Longrightarrow Y$ tem suporte s no conjunto de transações D, de forma que s seja a porcentagem das transações em D que contém $X \cup Y$. O problema de mineração de padrões frequentes é encontrar todos os conjuntos de itens X com suporte maior que s, para um limite $|D| >= s >= 1$.

Como exemplo, suponha que um programa de obtenção de conhecimento, depois de examinar milhares de candidatos ao vestibular de uma universidade particular, forneceu a seguinte regra: se o candidato é do sexo feminino, trabalha e teve aprovação com boas notas no vestibular, então não efetiva a matrícula. Uma reflexão é necessária para analisar a regra obtida: de acordo com os costumes, uma mulher em idade de vestibular, se trabalha é porque precisa, e neste caso deve ter feito inscrição também em universidade pública gratuita. Se a candidata teve boas notas, provavelmente foi aprovada na universidade pública onde efetivará matrícula. Claro que há exceções, mas a grande maioria poderia obedecer à regra encontrada.

A maioria dos métodos desenvolvidos para a mineração de padrões frequentes visa evitar a geração de todas as combinações possíveis de subconjuntos, o que possui um alto custo computacional. Muitos estudos contribuíram para a mineração eficiente de padrões frequentes e levaram ao desenvolvimento de diversos algoritmos, mas sendo a grande maioria baseada no Apriori [1], como o DPA [10], o qual é o foco deste trabalho.

III. DPA

O *Distributed Parallel Apriori* (DPA) [10] difere da versão inicialmente proposta para o Apriori [1] e de demais versões paralelas em dois aspectos: paralelização e distribuição de tarefas e divisão balanceada de carga entre os processos.

Os passos do algoritmo DPA são descritos a seguir.

Entrada: Um banco de dados $DB = (T_0, T_1, ..., T_{n-1})$, onde $T_i = (i_0, i_1, ..., i_{m-1})$; um conjunto de processos P, em que P_0 é o processo mestre (MP) e os processos escravos (SP) vão de P_1 até P_p; um suporte mínimo S.

Saída: Todos os padrões frequentes presentes no DB.

Etapa 1: Cada processo lê o DB.

Etapa 2: Cada processo define as TIDs das transações e inverte o sentido inicial do DB, de transações com seus itens para itens e suas TIDs, armazenando o novo formato em uma estrutura de dados.

Etapa 3: Cada processo calcula o peso dos 1-*itemsets* candidatos e, se o valor é maior que o suporte mínimo, define como frequente, descartando os demais.

Etapa 4: Definir k=1, onde k é o número de itens associados por *itemsets*.

Etapa 5: O processo mestre divide os k-*itemsets* frequentes em p subconjuntos disjuntos e atribui os subconjuntos aos processos escravos correspondentes.

Etapa 6: Cada processo escravo recebe seu próprio subconjunto de k-*itemsets* e gera seus (k+1)-*itemsets* candidatos .

Etapa 7: Cada processo calcula o peso dos seus candidatos, define quais são frequentes, descartando os demais.

Etapa 8: Cada processo escravo envia o seu conjunto de (k+1)-*itemsets* frequentes ao processo mestre.

Etapa 9: Se não existem, ou existir apenas um (k+1)-*itemsets* frequentes, encerra-se o algoritmo; senão soma-se 1 a k, o processo mestre envia o conjunto dos *itemsets* frequentes aos demais processos e repete-se os passos 5-9.

Na estrutura de armazenamento, os dados originalmente apresentados na base de dados (cada entrada representa uma transação e os itens presentes nela) são armazenados na forma de lista invertida. Cada entrada da base de dados tem uma identificação única de transação, chamada TID. As TIDs são armazenadas em uma estrutura de dados, onde cada entrada representa um item e o valor armazenado para a entrada é uma lista das transações onde o item ocorreu. Com isso, o número de *itemsets* pode ser rapidamente calculado sem a necessidade de outro escaneamento da base de dados.

Suponha uma base de dados de um supermercado onde cada linha representa uma compra e cada valor desta linha um item presente na compra. No momento de leitura dessa base de dados cada compra recebe um TID e os dados são armazenados em uma estrutura de forma que cada entrada represente um item e cada valor desta entrada uma lista das TIDs em que o item está presente. Essa estrutura é armazenada em memória principal e usada posteriormente para verificação dos *itemsets* frequentes.

Para conseguir uma boa distribuição de carga, o algoritmo DPA adota uma heurística baseada no número total de comparações entre os *itemsets* frequentes. Somando-se a cada *itemset* da estrutura de dados o número de *itemsets* abaixo dele, ao fim se tem o número total de comparações (interseções) que o algoritmo irá realizar. Assumindo que existam "p" processadores, então o subconjunto de *itemsets* é dividido em "*número total de comparações/p*".

Assim, um subconjunto dos *itemsets* frequentes é colocado no "*processador 1*" até que o número de comparações local naquele processador atinja "*número total de comparações/p*", em seguida coloca-se no "*processador 2*" e assim por diante.

IV. Algoritmo proposto: Distributed Multi-Thread Apriori (DMTA)

ATUALMENTE, as arquiteturas multicores tornaram-se financeiramente acessíveis e estão sendo bastante utilizadas. Para explorar melhor a capacidade destas arquiteturas é necessário que os algoritmos sejam adaptados, de forma que todos os núcleos executem de forma paralela. A criação de *threads* dentro de um processo é uma técnica que pode ser utilizada com esse objetivo, pois o sistema operacional poderá gerenciá-las de forma que cada uma execute eficientemente em um núcleo de processamento diferente. Além disso, *threads* de um mesmo processo podem compartilhar memória, reduzindo o tempo gasto em possíveis trocas de contexto e mensagens durante a execução.

O algoritmo proposto neste trabalho, denominado DMTA (*Distributed Multithread Apriori*), implementa a geração de conjuntos frequentes do algoritmo Apriori explorando a utilização de *threads*. Ele está descrito na subseção IV-A, representado na forma de uma entrada, uma saída e uma sequência de instruções divididas em 9 etapas. As principais diferenças entre os algoritmos DPA e DMTA ocorrem nas Etapas 5 e 6. Na Etapa 5, o balanceamento de carga entre os processos é realizado pelo processo mestre no DPA, enquanto que no DMTA cada processo é responsável por identificar quais conjuntos estão sob sua responsabilidade. Na Etapa 6, há criação de *threads* em cada processo no DMTA, enquanto que no DPA isso não ocorre.

A. O algoritmo DMTA

Entrada: Um banco de dados $DB = (T_0, T_1, ..., T_{n-1})$, onde $T_i = (i_0, i_1, ..., i_{m-1})$; um conjunto de processos P, em que P_0 é o processo mestre (MP) e os processos escravos (SP) vão de P_1 até P_p; um suporte mínimo S.

Saída: Todos os padrões frequentes presentes no DB.

Etapa 1: Cada processo lê o DB.

Etapa 2: Cada processo define as TIDs das transações e inverte o sentido inicial do DB, de transações com seus itens para itens e suas TIDs, armazenando o novo formato em uma estrutura de dados.

Etapa 3: Cada processo calcula o peso dos 1-*itemsets* candidatos e, se o valor é maior que o suporte mínimo, define como frequente, descartando os demais.

Etapa 4: Definir k=1, onde k é o número de itens associados por *itemsets*.

Etapa 5: Cada processo faz o cálculo das linhas (conjuntos) sob sua responsabilidade.

Etapa 6: Cada processo verifica os conjuntos atribuídos a ele, gera suas *threads*, divide os conjuntos entre elas, gerando os seus (k+1)-*itemsets* candidatos.

Etapa 7: Cada processo calcula o peso dos seus candidatos, define quais são frequentes, descartando os demais.

Etapa 8: Cada processo escravo envia o seu conjunto de (k+1)-*itemsets* frequentes ao processo mestre.

Etapa 9: Se não existem, ou existir apenas um (k+1)-*itemsets* frequentes, encerra-se o algoritmo; senão soma-se 1 a k, o processo mestre envia o conjunto dos *itemsets* frequentes aos demais processos e repete-se os passos 5-9.

O escalonamento das *threads* para os núcleos de processamento fica a cargo do sistema operacional.

A subseção IV-B apresenta um exemplo de execução para ilustrar melhor o funcionamento do DMTA e suas diferenças para o algoritmo DPA.

B. Exemplo

Suponha um ambiente com 2 máquinas (cada uma com 4 núcleos) e que sejam criados dois processos (cada um em uma máquina diferente). Considere também que há uma base de dados composta por 5 transações e 13 itens e um suporte de 0,15, ou 15%.

Nas Etapas 1 e 2 do algoritmo DMTA, cada processo lê a base de dados no formato horizontal (TID x Itens) da Figura 2(a), e o converte para o formato vertical (*itemsets* x TIDs) da Figura 2(b).

TID	Itens
1	A C D G P T
2	A F L M O
3	B F C M
4	B C K S L T
5	A C O P B

(a)

→

Itemsets	TIDS
A	1 2 5
B	3 4 5
C	1 3 4 5
D	1
F	2 3
G	1
K	4
L	2 4
M	2 3
O	2 5
P	1 5
S	4
T	1 4

(b)

Fig. 2. Base de dados e leitura inicial.

Na Etapa 3, cada processo calcula o suporte de cada *itemset* (número de transações em que o *itemset* ocorre), ordena-os decrescentemente e descarta aqueles abaixo do suporte mínimo (15% para o exemplo de 5, o que faz com que os *itemsets* com peso 1 sejam descartados), como apresentado na Figura 3.

A divisão de carga e tarefas proposta pelo DPA assume que os processos realizam a interseção entre conjuntos aproximadamente o mesmo número de vezes, para encontrar os *itemsets* frequentes. Porém a estrutura de dados que armazena os *itemsets* e as transações em que estão presentes, usada tanto para o DPA quanto para o DMTA, é ordenada decrescentemente para fins de divisão de carga, de forma que as primeiras linhas contenham os itens mais frequentes. Assim, o custo para realizar a interseção entre as primeiras linhas, atribuídas aos primeiros processos pelo DPA, é maior do que entre as últimas.

O algoritmo DMTA apresenta uma nova proposta de balanceamento, onde os dados são divididos de forma

Itemset	TID	Peso
C	1345	4
A	125	3
B	345	3
F	23	2
L	24	2
M	23	2
O	25	2
P	15	2
T	14	2
D	1	1
G	1	1
K	4	1
S	4	1

→

Itemset	TID	Peso
C	1345	4
A	125	3
B	345	3
F	23	2
L	24	2
M	23	2
O	25	2
P	15	2
T	14	2

Fig. 3. Definição dos *itemsets* frequentes.

circular entre os processos, como pode ser visualizado na Figura 4 (Etapa 5).

Itemsets	TID
C	1345
A	125
B	345
F	23
L	24
M	23
O	25
P	15
T	14

Processo 0 Processo 1

Fig. 4. Divisão dos *itemsets* frequentes entre processos.

O DMTA utiliza a biblioteca OpenMP para criar as *threads* em cada processo, dividindo ainda mais as cargas de trabalho e as comparações realizadas pelo algoritmo, reduzindo-se assim o tempo ocioso dos núcleos de processamento e acelerando a execução. Na Figura 5 é apresentado o escalonamento das *threads* para o exemplo (Etapa 6).

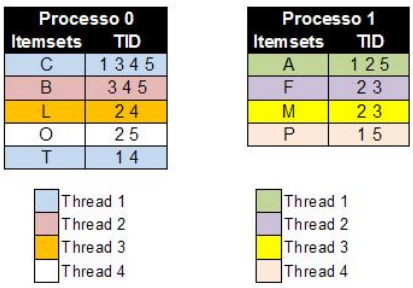

Fig. 5. Escalonamento das *threads*.

No escalonamento das *threads* foi utilizado o método "static", implementado pelo OpenMP e que define a divisão de linhas como fixa em um valor de linhas igual ao valor da variável "chunk", definida com valor 1 no DMTA. Ou seja, o escalonamento das *threads* foi circular, igual ao escalonamento realizado pelos processos. Foram testados outros tipos de escalonamento do OpenMP, como o "dynamic", em que as *threads* verificam qual a próxima linha a ser atribuida em tempo de execução, e o "guided", em que o valor de "chunk" é alterado durante a divisão das linhas. Porém o escalonamento "static"foi o que apresentou melhor balanceamento de carga.

A definição do número de *threads* que serão criadas dentro de um processo é realizada em tempo de execução, utilizando o número de núcleos de processamento da máquina obtido pela função *omp_get_num_procs*. Desta forma, é possível executar o algoritmo tanto em ambientes homogêneos (todas as máquinas com mesmo número de núcleos de processamento) quanto em ambientes heterogêneos (máquinas com diferentes números de núcleos de processamento).

O trecho que conta com a criação de *threads* é a parte de maior computação do algoritmo, no qual são realizadas as interseções das listas de TIDs dos *itemsets* buscando as TIDs comuns a dois conjuntos. Na Figura 6 é apresentada a interseção entre os conjuntos de 1-*itemsets* frequentes, C e A, para o exemplo.

Itemsets	TID	Peso
C	1345	4
A	125	3

→

Itemsets	TID	Peso
CA	15	2

Fig. 6. Interseção dos *itemsets*.

Esta interseção é de responsabilidade da *thread* 1 no Processo 0, conforme Figuras 4 e 5. A interseção também deve ocorrer entre o item C com todos os que estão abaixo dele na Figura 4, ou seja, CA, CB, CF, CL, CM, CO, CP, CT. É verificado o número de TIDs comuns entre eles e se este número atende ao suporte estabelecido. O mesmo processo deverá ser repetido para os demais *itemsets*. Na Figura 7 (a) é apresentado o resultado do exemplo proposto para os conjuntos de 2-*itemsets* e na Figura 7 (b) o resultado para os conjuntos de 3-*itemsets*.

Itemset	TID	Peso
CB	345	3
CA	15	2
CP	15	2
CT	14	2
AO	25	2
AP	15	2
FM	23	2

(a)

Itemset	TID	Peso
CAP	15	2

(b)

Fig. 7. Conjuntos de *itemsets* frequentes.

A estrutura de dados que armazena o resultado dos k-*itemsets* frequentes é compartilhada entre as *threads* do processo e o acesso a essa estrutura é realizado de forma atômica pelas *threads*.

V. Experimentos e Resultados

O algoritmo foi implementado utilizando a linguagem C++, por ser uma linguagem eficiente para implementação de software básico. A linguagem também possui a classe pré-definida "vector"que foi utilizada, de forma encadeada, como a estrutura para armazenamento

de dados do algoritmo. Essa estrutura é mais eficiente do que a tabela *hash*, pois as operações de acesso às transações são feitas de forma sequencial no algoritmo DMTA, onde os *itemsets* são armazenados de forma ordenada pelo suporte. As bibliotecas OpenMP e MPI apresentam suporte à linguagem escolhida e, além disso, a biblioteca MPI também foi utilizada na implementação do algoritmo base DPA e a biblioteca OpenMP apresenta fácil possibilidade de configuração e programação. O principal ambiente experimental utilizado para avaliar o algoritmo proposto consiste em um *cluster* composto por 8 máquinas com a mesma configuração disposta na Tabela I.

TABELA I
Especificações do ambiente

Ambiente de Hardware	
CPU	AMD Phenom (TM)II B95 X4 (4 Cores) 3.000 MHz 512Kb Cache
Memória	5GB DDR RAM
Disco Rígido	30GB
Rede	10/100 Mbps Fast Ethernet
Ambiente de Software	
Sistema Operacional	Ubuntu 10.04 LTS
Bibliotecas	Open-MPI 1.4.1 OpenMP 4.4.3

Foi utilizado um ambiente de hardware homogêneo, porém os experimentos também poderiam ser realizados em ambientes de hardware heterogêneo, pois o algoritmo, através de uma função da biblioteca OpenMP, é capaz de reconhecer o número de núcleos de processamento de cada máquina em que os processos foram iniciados e dividir a carga de trabalho entre todos, utilizando melhor os recursos computacionais dos processadores multicores.

Nos experimentos, cada máquina do *cluster* recebeu uma cópia local das 3 bases sintéticas, geradas pelo *Quest Synthetic Data Generator* da IBM [9], também utilizado nos experimentos do DPA. As bases que são a entrada do algoritmo, utilizaram a nomenclatura padrão do gerador e a Tabela II apresenta os detalhes de cada uma, onde T é a média de itens por transação, D é o número de transações, N é o número de itens e K é a representação de mil.

Variando o suporte, a carga de trabalho é fortemente impactada, já que com maiores suportes são encontrados menos *itemsets* frequentes, reduzindo assim a computação e o tempo de execução. Na Figura 8 é apresentada a visualização dos tempos de execução do DMTA com 4 suportes diferentes. Foi utilizado o suporte 0,0015 por apresentar o maior tempo de execução e permitir verificar maiores diferenças entre os algoritmos. Além disso, o nível de computação alcançado com este valor se assemelha ao que pode ser encontrado em execuções com bases de dados reais.

As Figuras 9 e 10 apresentam os tempos de execução do DPA, implementado conforme [10], e do DMTA com a criação de 4 *threads*, respectivamente. Os experimentos foram realizados com as 3 bases de dados e com a variação do número de processos até 8, sendo criado um processo por máquina. Pode-se observar que para ambos os algoritmos o tempo de execução reduz com o aumento do número de processos, principalmente para a base maior. Porém, o DMTA cria um total de 32 *threads* com 8 processos, explorando melhor a arquitetura multicore e gerando um tempo de execução menor do que o DPA.

TABELA II
Bases da dados

Nomeclatura	*Especificações*
T10I4D800KN100K	800.000 transações, com média de 10 itens por transação e 100.000 itens possíveis.
T10I4D1600KN100K	1.600.000 transações, com média de 10 itens por transação e 100.000 itens possíveis.
T10I4D3200KN100K	3.200.000 transações, com média de 10 itens por transação e 100.000 itens possíveis.

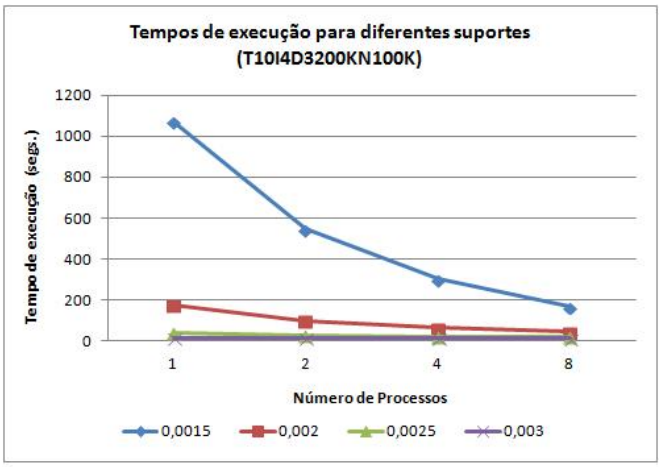

Fig. 8. Tempos de execução do DMTA para diferentes suportes.

Foi realizado um experimento em uma única máquina com 4 núcleos de processamento visando comparar o DPA com 4 processos e o DMTA com 4 *threads* no mesmo processo. Foi utilizada a base com 800.000 transações (T10I4D800KN100K). O intervalo de confiança de 95% do tempo de execução obtido pelo DPA foi entre 126,26 segundos e 128,51 segundos. Pelo DMTA foi entre 92,79 segundos e 93,48 segundos. Pode-se afirmar que o DMTA gastou entre 26,51% e 27,25% menos tempo de execução que o DPA. Isso ocorre devido ao escalonamento de *threads* utilizado pelo DMTA e devido ao fato que a criação e troca de contexto de processos, realizada pelo sistema operacional, é mais lenta do que com *threads*.

Foram realizadas também execuções do DMTA sem *threads* e com 4 *threads* para todas as bases de dados e quantidades de processos, como apresentado na Figura 11.

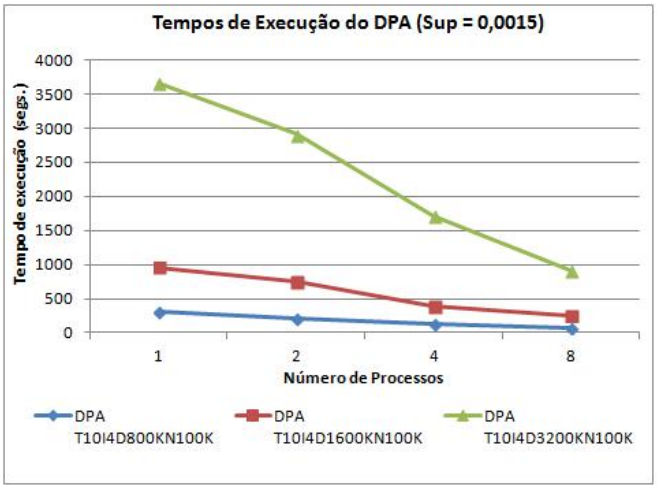

Fig. 9. Tempos de execução do algoritmo DPA.

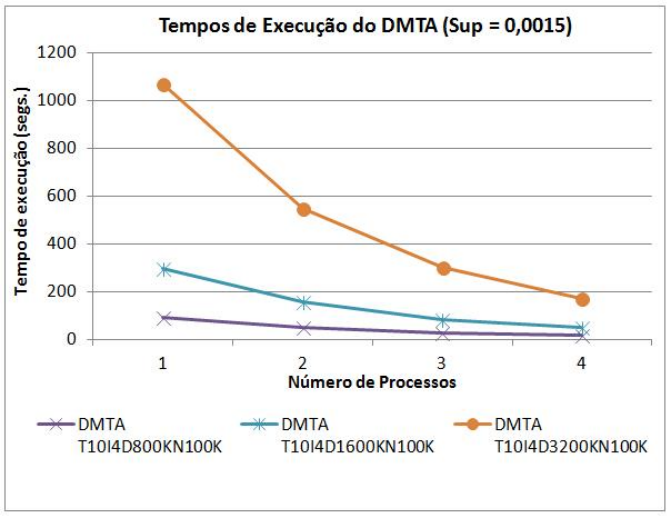

Fig. 10. Tempos de execução do algoritmo DMTA.

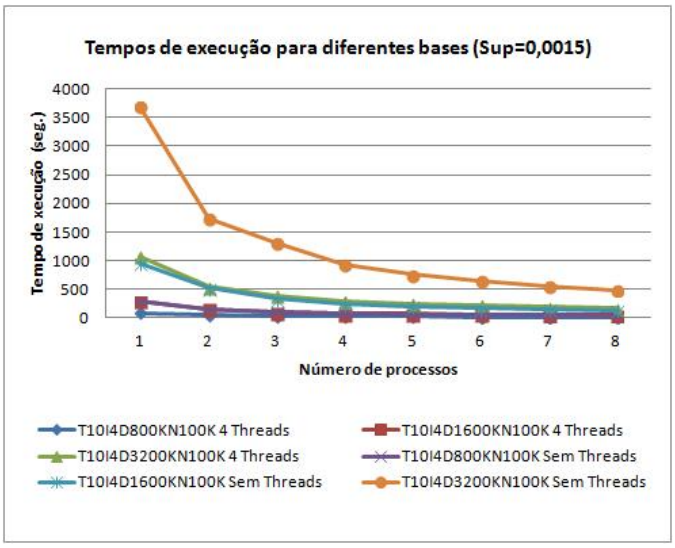

Fig. 11. Tempos de execução do DMTA variando bases de dados e processos.

Pode-se observar que o tempo de execução não reduz linearmente ao aumentar o número de transações. Este comportamento ocorre porque há trechos do algoritmo que devem ser executados sequencialmente. Além disso, ao duplicar o número de transações os dados da base inicial não são simplesmente dobrados. Com o aumento do número de transações, itens que não ocorriam em nenhuma transação de uma base menor, podem passar a ocorrer, e itens que estavam abaixo do suporte mínimo podem se tornar frequentes, ambas as situações causam novos cálculos, impactando na computação. De uma forma geral, a duplicação de transações não causa apenas a duplicação da carga, mas um aumento exponencial da mesma.

Na Figura 12 é apresentado o *speedup*, em relação à execução sequencial, com 1 processo e sem *threads*, para as 3 diferentes bases de dados. O *speedup*, assim como o tempo de execução, não se altera linearmente devido a *overheads* paralelos que acontecem ao aumentar o número de processos. Entretanto os resultados mostram que o tempo de computação reduzido pelo uso das *threads* e processos supera o tempo acrescido por *overheads*, tornando vantajosa a criação dos mesmos. A maior base de transações, com 3,2 milhões de transações, apresentou os maiores valores de *speedup*, comprovando que o DMTA é mais vantajoso ao aumentar o volume de dados.

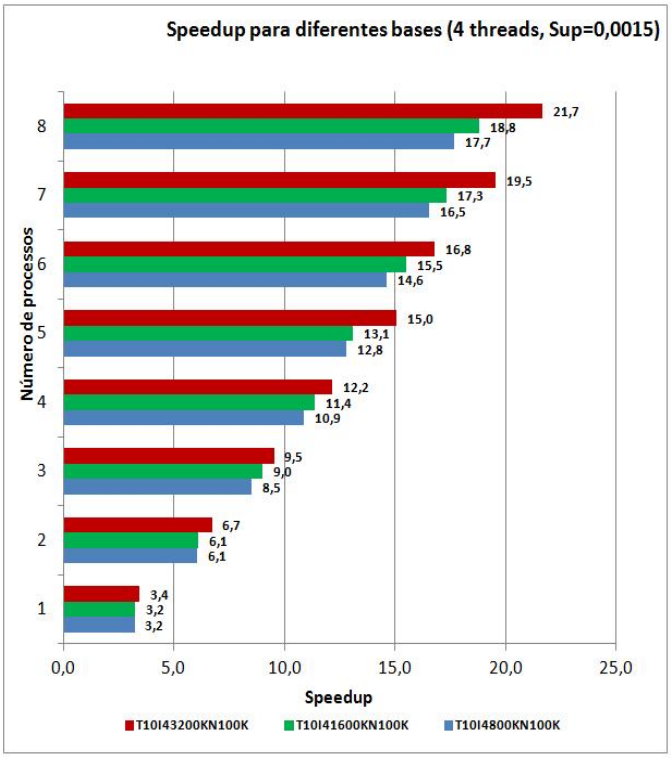

Fig. 12. *Speedup* em relação a execução sequencial do DMTA.

Na Figura 13 é apresentado o balanceamento de carga entre os processos, baseado no tempo de execução de cada processo, pois o processo com maior carga de trabalho gastará maior tempo de execução. Foi realizada uma execução do DMTA sem *threads* e uma execução com 4 *threads*, ambas com 8 processos. A base de dados utilizada

neste experimento foi a de 800.000 transações, com valor de suporte igual a 0,0015. O DMTA mostra um bom balanceamento de carga entre os processos, aperfeiçoado ainda mais pela segunda divisão de cargas entre os núcleos, acelerando assim o tempo total de execução.

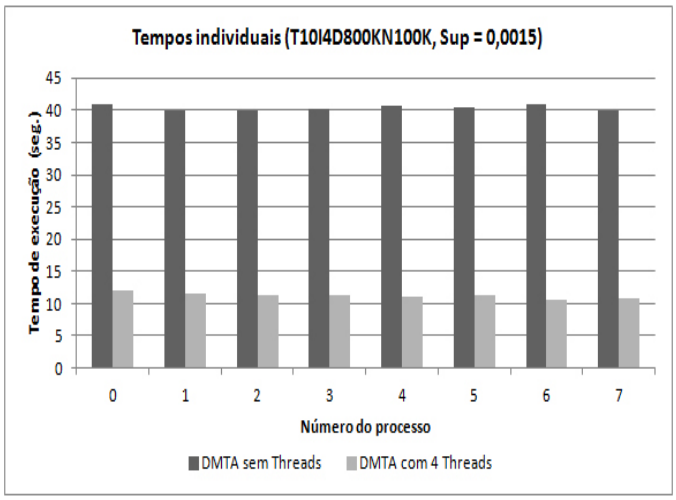

Fig. 13. Balanceamento de carga entre processos.

Na Figura 14 é apresentado o percentual do tempo total de execução que é gasto com comunicação por cada processo para uma execução do DMTA com 8 processos na maior base de transações (3,2 milhões), com valor de suporte igual a 0,0015. O percentual apresentado no gráfico, para cada processo, refere-se à soma dos tempos de envio e recebimento de dados realizados pelas chamadas do MPI (MPI_Send, MPI_Recv e MPI_Bcast) com o tempo de espera do processo para realizar a comunicação. Por exemplo, o processo 1 gastou 15% do tempo total de execução do programa realizando comunicação. O processo 0 apresenta um baixo percentual pois este é o processo mestre, responsável por receber e enviar dados sem concorrência com outros processos, enquanto os demais realizam a comunicação simultaneamente com o processo mestre, ocasionando espera até que este esteja livre para conseguir realizar a comunicação.

VI. Conclusão

DESCOBRIR padrões frequentes em grandes bases de dados é uma tarefa importante. No entanto, o processo de geração de *itemsets* e a busca de seus padrões consomem muito tempo de processamento. As estratégias de computação paralela e distribuída ofereceram soluções viáveis para este problema.

No algoritmo proposto, DMTA, a base de dados é armazenada na estrutura de dados de forma inversa à tradicionamente utilizada, permitindo efetivamente reduzir as iterações necessárias para percorrer uma base de dados e acelerar o cálculo de conjuntos de *itemsets*. Ele também adota uma heurística útil para a divisão dos conjuntos de itens entre os processadores ao realizar uma divisão de forma circular, efetivando assim o equilíbrio de carga entre os processadores e reduzindo o tempo ocioso dos

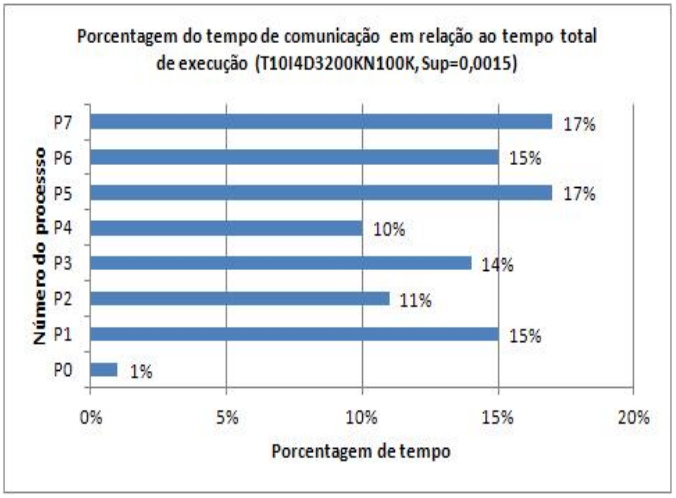

Fig. 14. Tempo de comunicação nos processos (P0, P1, ..., P7) no DMTA.

mesmos. Além disso, são criadas *threads* em cada processo, explorando ao máximo as arquiteturas multicores, popularizadas hoje em dia, evitando o tempo ocioso dos núcleos de processamento e a grande espera para comunicação entre os processos.

Os resultados experimentais obtidos, quando comparados com outro algoritmo semelhante mostram que o DMTA consegue resultados melhores do que o DPA, especialmente para bases de dados maiores e baixo suporte mínimo. Os resultados também mostram que o DMTA tem um bom equilíbrio de carga entre processos. O algoritmo apresentado pode, assim, proporcionar uma estratégia útil e distribuída para problemas de mineração de dados frequentes.

Como trabalhos futuros, pretende-se avaliar como alcançar ainda mais desempenho pelo aumento do número de *threads* e processos, além de estudar novas formas de balanceamento de carga e escalonamento de *threads*. Para isso, será analisado o comportamento do algoritmo variando esses parâmetros. O algoritmo será avaliado também para bases de dados reais, visando entender seu comportamento mediante bases com comportamento irregular. Outra possibilidade é a realização de otimizações nas estruturas de dados utilizadas e na forma de acesso aos dados. Por fim, novas execuções em diferentes sistemas operacionais, como o Microsoft Windows, devem ser realizadas para que se observe o comportamento das *threads* e processos gerenciados por este sistema.

Agradecimentos

Os autores desse artigo agradecem o apoio da FAPEMIG e do CNPq.

Referências

[1] AGRAWAL, R. and SRIKANT, R. “Fast algorithms for mining association rules”. Proceedings of the 20th international conference on very large databases. Pages: 487-499. 1994.

[2] AGRAWAL, R. and SHAFER, J. C. “Parallel mining of association rules”. IEEE Transactions on Knowledge and Data Engineering. 8(6). Pages: 962-969. 1996.

[3] BARNEY, B. "OpenMP". Disponível em <https://computing.llnl.gov/tutorials/openMP/>. Acesso em:30/08/2012.
[4] BODON, F. "A fast apriori implementation". Proceedings of the IEEE ICDM workshop on frequent itemset mining implementations. 2003.
[5] CHEUNG, D. W., LEE, S. D. and XIAO, Y. "Effect of data skewness and workload balance in parallel data mining". IEEE Transactions on Knowledge and Data Engineering. Volume 14. Issue 3. Pages: 498-514. 2002.
[6] CHEUNG, D. W., NG, V. T. and FU, A. W. "Efficient mining of association rules in distributed databases". IEEE Transactions on Knowledge and Data Engineering. Volume 8. Issue 6. Pages: 911-922. 1996.
[7] EINAKIAN, S. and GHANBARI, M. "Parallel implementation of association rules in data mining". Proceedings of the 38th southeastern symposium on system theory. Pages: 21-26. 2006.
[8] HAN, J., PEI, J. and YIN, Y. "Mining Frequent Patterns without Candidate Generation". Proceedings of Management of Data. Volume 29. Issue 2. Pages: 1-12. 2000.
[9] IBM Almaden. "Quest synthetic data generation code". 1994. Disponível em: <http://almaden.ibm.com/cs/quest/syndata.html>. Acesso em: 20/03/2012.
[10] KUN-MING, Y., JIAYI, Z., TZUNG-PEI, H. and JIA-LING, Z. "A load-balanced distributed parallel mining algorithm". Expert Systems with Applications. Volume 37. Pages: 2459-2464. 2010.
[11] LIU, L., LI, E., ZHANG, Y., and TANG, Z. "Optimization of Frequent Itemset Mining on Multiple-core Processor". Proceedings of the 33rd International Conference on Very Large Databases. Pages 1275-1285. 2007.
[12] MING-YEN, L., PEI-YU, L. and SUE-CHEN, H. "Apriori-based frequent itemset mining algorithms on MapReduce". Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication (ICUIMC '12). 2012.
[13] MPI. "The Message Passing Interface (MPI) standard". Disponível em <http://www.mcs.anl.gov/research/projects/mpi/>. Acesso em: 30/08/2012.
[14] PARTHASARATHY, S., ZAKI, M. J., OGIHARAAND, M. and LI, W. "Parallel data mining for association rules on shared-memory systems". Knowledge and Information Systems. Volume 3. Issue 1. Pages: 1-29. 2001.
[15] WU, C. C., LAI, L. F., HUANG, L. T., JHANAND, S. S. and LU, C. "A fine-grained scheduling strategy for improving the performance of parallel frequent itemsets mining". International Journal of Computational Science and Engineering. Volume 6. Issue 4. Pages: 264-274. 2011.
[16] WU, J. and LI, X. M. "An efficient association rule mining algorithm in distributed database". International Workshop on Knowledge Discovery and Data mining (WKDD). Pages: 108-113. 2008.
[17] YE, Y. and CHIANG, C. C. "A parallel apriori algorithm for frequent itemsets mining". Proceedings of the fourth international conference on software engineering research, management and applications. Pages: 87-94. 2006.
[18] ZAKI, M., PARTHASARATHY, S., OGIHARA, M. and LI, W. "New algorithms for fast discovery of association rules". Proceedings of Knowledge Discovery and Data Mining. Pages: 283-286. 1997.

André Camilo Bolina graduou-se no segundo semestre de 2012 em Ciência da Computação pela Universidade Federal de Lavras (UFLA), Lavras, Minas Gerais, Brasil. É colaborador do Laboratório de Inteligência Computacional e Sistemas Avançados (LICESA). Sua pesquisa é focada em processamento paralelo e distribuído e sistemas inteligentes.

Denilson Alves Pereira é professor do Departamento de Ciência da Computação da Universidade Federal de Lavras (UFLA), Minas Gerais, Brasil, onde atua na graduação e no mestrado. Seus interesses em pesquisa incluem recuperação de informação, mineração de dados e banco de dados. Obteve o título de doutor pelo Departamento de Ciência da Computação da Universidade Federal de Minas Gerais, em 2009.

Ahmed A. A. Esmin é professor do Departamento de Ciência da Computação da Universidade Federal de Lavras (UFLA), Minas Gerais, Brasil, onde coordena o Laboratório de Inteligência Computacional e Sistemas Avançados (LICESA). Sua pesquisa é em inteligência computacional, inteligência de swarm, machine learning

Marluce Rodrigues Pereira é doutora em Engenharia de Sistemas e Computação pela Universidade Federal do Rio de Janeiro (UFRJ), Brasil. É professora do Departamento de Ciência da Computação da Universidade Federal de Lavras (UFLA), Lavras, Minas Gerais, Brasil, desde 2006, onde também trabalha com pesquisa no Laboratório de Inteligência Computacional e Sistemas Avançados (LICESA). Sua pesquisa é focada em processamento paralelo e distribuído e sistemas inteligentes.