Algoritmo de Regras de Associação Paralelo para Arquiteturas *Multicore* e *Manycore*

João Saffran[1]*, Rodrigo C. O. Rocha[2], Luís Fabrício W. Góes[1]

[1]Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica de Minas Gerais
Minas Gerais, Brasil – 30535-901

[2]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
Minas Gerais, Brasil – 31270-901

joao.saffran@sga.pucminas.br, rcor@dcc.ufmg.br, lfwgoes@pucminas.br

Abstract. *Association rules are specific techniques to find simultaneous occurrence of frequent items in a non-numeric database. Among the most widespread techniques,* `Apriori` *algorithm stands out for being used for market basket analysis as well as the prediction of scheduling policies on parallel architectures. However,* `Apriori` *demands high computational cost. In this paper we developed and evaluated a parallel version of the* `Apriori` *algorithm in OpenMP exploring the computational power of multicore and manycore architectures. The experimental results show that the parallel* `Apriori` *achieves a performance gain of up to 10x on a multicore also on a manycore.*

Resumo. *Regras de associação são técnicas específicas para encontrar ocorrências simultâneas de itens frequentes em uma base de dados não-numéricos. Dentre as técnicas mais difundidas, o algoritmo* `Apriori` *se destaca por ser utilizado tanto para a análise de cesta de compras como também para a predição de políticas de escalonamento em arquiteturas paralelas. Contudo, o* `Apriori` *demanda alto custo computacional. Neste artigo é desenvolvida e avaliada uma versão paralela do algoritmo* `Apriori` *em OpenMP que explora o poder computacional das arquiteturas multicore e manycore. Os resultados experimentais mostram que o* `Apriori` *paralelo alcança um ganho de desempenho de até 10x em uma arquitetura multicore e também em um manycore.*

1. Introdução

O `Apriori` [Rao and Gupta 2012] é um algoritmo de regras de associação amplamente utilizado para encontrar padrões frequentes em grandes bases de dados não numéricas. Ele vem sendo utilizado em diversas áreas tais como bioinformática, diagnóstico médico, análise de dados científicos e até em escalonamento de processos em máquinas paralelas [Castro et al. 2014]. Esse algoritmo demanda alto desempenho computacional, devido ao seu grande espaço de busca envolvendo a geração dos candidatos de conjuntos de itens frequentes.

Com o advento de arquiteturas *multicore* e *manycore* [Borkar 2007, Hill and Marty 2008], aplicações que executam sobre grandes quantidades de dados, como o `Apriori`, devem ser implementadas por meio de ferramentas de

*Aluno de iniciação científica.

programação paralela para alcançarem alto desempenho computacional. O OpenMP é uma destas ferramentas que provê diretivas de compilação que auxiliam o compilador na paralelização de um código sequencial [Klemm and Terboven 2013]. Além de simplificar a programação paralela, o código OpenMP pode ser executado tanto em arquiteturas *multicore* quanto em arquiteturas *manycore* como o Intel Xeon Phi.

O objetivo deste trabalho é a implementação paralela e avaliação de desempenho do algoritmo `Apriori` para arquiteturas *multicore* e *manycore*. Os resultados experimentais mostram que o `Apriori` paralelo alcança um ganho de desempenho de até 10x em uma arquitetura *multicore* e uma *manycore*.

O restante deste trabalho está dividido em 5 seções, incluindo esta introdução. A seção 2 apresenta o algoritmo `Apriori` Paralelo. A seção 3 apresenta a configuração dos experimentos e os resultados experimentais. A seção 4 apresenta os trabalhos relacionados e a seção 5 apresenta as conclusões e trabalhos futuros.

2. Apriori Paralelo

Dada uma lista de conjunto de itens, o algoritmo `Apriori` identifica as regras de associação entre estes itens baseado em suas frequências. Essas regras revelam os subgrupos de itens que ocorrem juntos em um mesmo conjunto de itens frequentes [Rao and Gupta 2012]. O algoritmo segue a regra: *para todo conjunto de itens não vazio seus subconjuntos de itens devem ser frequentes também*. Essa regra permite que o algoritmo elimine todo conjunto de itens que não é composto por um subconjunto de itens frequentes, reduzindo significativamente o espaço de busca.

Para cada regra de associação $A \rightarrow B$, onde A e B são subconjuntos dos itens de um conjunto de itens frequentes, o algoritmo `Apriori` calcula a *confiança* (`Conf`), apresentada na equação 1, sendo o suporte, o número mínimo de ocorrências de um subconjunto de itens para que ele seja considerado frequente. Quanto maior o nível de confiança, significa que: com a presença de um subconjunto de itens A em um conjunto de itens frequentes, maior é a probabilidade do subconjunto de itens B também estar presente.

$$\texttt{Conf}(A, B) = \frac{suporte(A \wedge B)}{suporte(A)} \tag{1}$$

Para a implementação paralela deste algoritmo, foi utilizado o padrão MapReduce, que se comporta igualmente idependente da arquitetura utilizada. Desta forma o algoritmo é dividido em 2 etapas, a etapa de mapeamento, que consiste basicamente em dividir o trabalho entre um conjunto de tarefas independentes, e a etapa de redução, que é a junção dos resultados das tarefas independentes.

O algoritmo 1 apresenta a versão paralela do algoritmo `Apriori` implementado. Primeiramente, deve-se obter todos os subconjuntos de tamanho igual a 1 de todos os elementos da sua base de dados (S). Então executa-se pela primeira vez a etapa de mapeamento do padrão, onde os elementos de S são divididos entre as *threads* utilizadas, representado pelo comando *parallel foreach*, e cada uma contará a ocorrência destes elementos na base de dados. Com as frequências já descobertas, reagrupa-se os subgrupos de S, originalmente divididos entre as *threads*, fazendo-se assim a etapa de redução. Dos elementos contidos em S remove-se aqueles com suporte inferior ao determinado e calcula-se a confiança. Com os elementos de tamanho K já definidos, deve-se inserir

$A \rightarrow B$ em R com $\texttt{Conf}(A, B)$, formando assim as regras de associação que serão dadas na saída do algoritmo. Por fim realiza-se mais uma fase de mapeamento, dividindo-se novamente os elementos de S entre as *threads*, mas desta vez para descobrir os elementos de tamanho $k + 1$, realiza-se então uma redução para reagrupar estes elementos em S.

Algorithm 1: `Apriori Paralelo`(I, M)

input : Dados de entrada I, suporte mínimo M
output: regras de associação R

begin
  $S \leftarrow$ *Conjunto de itens de tamanho* $K = 1$ *itens de* I*;*
  while S *não é vazio* **do**
    parallel foreach $e_i \in S$
      Contar a ocorrência de cada conjunto de itens $e_i \in S$*;*
      Reagrupar os elementos de cada e_i *em* S*;*
    end
    Remover o conjunto de itens S_i *com suporte(*S_i*)* $< M$*;*
    foreach *Conjunto de itens* $S_i \in S$ **do**
      Inserir $A \rightarrow B$ *em* R *com* $\texttt{Conf}(A, B)$*;*
    end
    parallel foreach $e_i \in S$
      Inserir as permutações de tamanho $K + 1$ *itens nos conjuntos de itens restantes* $e_i \in S$*;*
    end
  end
end

Este processo é realizado até que não hajam mais itens de tamanho $K + 1$ na base de dados. Neste ponto são obtidos todos os elementos com frequência maiores que o suporte determinado e suas associações. O `Apriori` foi implementado em C++, para paralelizar foi utilizada a ferramenta OpenMP. Para coverter o algoritmo 1 para a implementação em C++ foi necessário substituir as estruturas de repetição *parallel foreach* pelas respectivas diretivas utilizadas no OpenMP.

3. Resultados Experimentais

Na implementação paralela foram realizados testes de escalabilidade, utilizando-se como métricas o *speedup* da aplicação, eficiência, tempo de execução e a taxa de falta de cache nas mémorias cache L2 e L3 da arquitetura *multicore*. Os testes foram realizados em uma arquitetura composta de dois processadores *multicore* Intel Xeon E5-2620 2.1GHz, com 6 núcleos cada um, cache L3 com 15 MB e memória principal de 64GB, e uma arquitetura Intel Xeon Phi Coprocessor 7120A 1.238GHz, com 61 núcleos, cache L3 com 30.5 MB e memória principal de 16GB. As métricas de falta de cache foram obtidas através da biblioteca PAPI [McCraw et al. 2014]. O particionamento do algoritmo foi feito automaticamente pelo OpenMP.

Apesar da existência de repositórios com bases de dados para testar algoritmos de Aprendizado de Máquina, como o UCI [Lichman 2013], a base de dados para teste de desempenho do `Apriori` foi criada para a previsão de combinações de cartas de um jogo digital de cartas chamado *Hearthstone*. A base é composta por 5 mil combinações de

cartas, extraídas de *logs* do jogo. O algoritmo `Apriori` foi executado 5 vezes para cada variação da quantidade de núcleos (1 até 12) na arquitetura *multicore* e 5 vezes, variando a quantidade de núcleos (10 até 50), na arquitetura *manycore*. A média geométrica e o desvio padrão foram calculados, garantindo-se um grau de confiança de 95%. A corretude da implementação paralela foi verificada comparando-se a sua saída com a do software WEKA [Hall et al. 2009]. O baseline utilizado para o cálculo do *speedup* nas duas arquiteturas foi a versão sequencial do `Apriori` executada na arquitetura *multicore*.

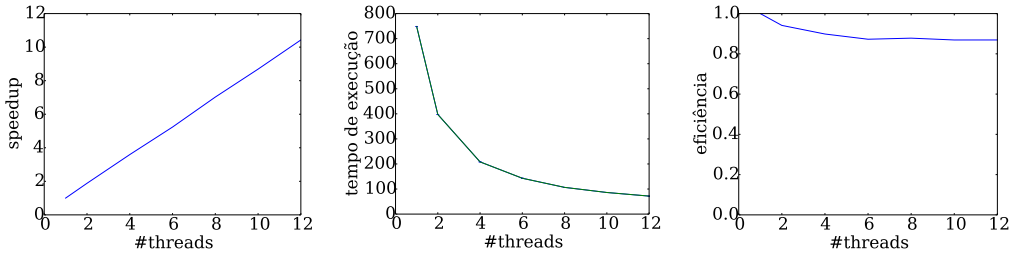

Figura 1. *Speedup*, tempo de execução e eficiência em *multicore*.

De acordo com a Figura 1, o resultado se mostrou linear em uma arquitetura *manycore*, conforme vai se aumentando o número de núcleos de processamento, o *speedup* também aumenta. É importante notar que o *speedup* máximo obtido foi de 10, usando 12 núcleos, se comparado com a implementação sequencial. O tempo de execução da aplicação, conforme se aumenta o número de núcleos de processamento, como esperado, vai decaindo de forma logarítmica decrescente. Também percebe-se que a aplicação atingiu 85% de eficiência utilizando-se os 12 núcleos (a eficiência é calculada, dividindo-se o *speedup* pelo número de núcleos utilizados para se obter aquele *speedup*).

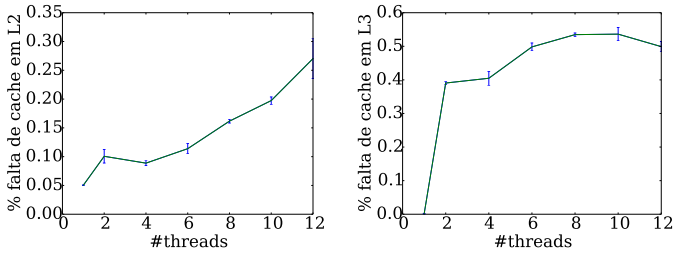

Figura 2. Taxa de falta de cache L2 e L3 variando-se a quantidade de threads.

A Figura 2 mostra as taxas de falta de cache em L2 e L3. Ressalta-se que a taxa de falta de cache em L2 e L3 foi crescendo conforme aumenta-se o número de núcleos, isso se deve por causa do aumento do número de *threads*, que causa um maior uso da cache de cada processador, o que aumenta a probabilidade de ocorrer uma falta de cache.

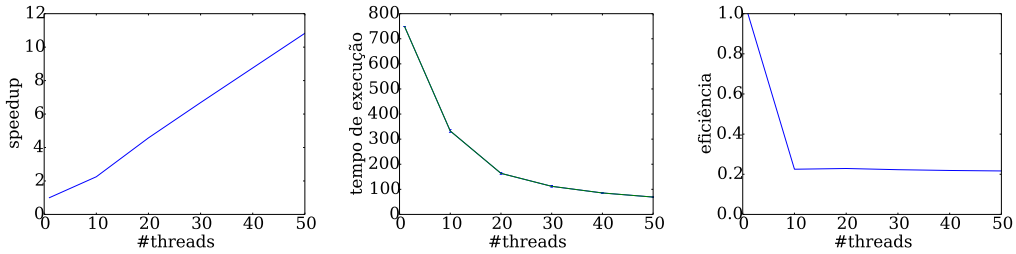

Figura 3. *Speedup*, tempo de execução e eficiência por número de threads em *manycore*.

A Figura 3 mostra os resultados na arquitetura *manycore*. A aplicação, neste ambiente, também mostrou-se linear e escalável, porém menos eficiente, devido as carac-

terísticas do *manycore*, que apresenta maior número de núcleos, mas com desempenho inferior se comparados aos núcleos do *multicore*, levando a uma baixa eficiência.

Além disso, a capacidade de vetorização do *manycore* não foi totalmente explorada, pois a implementação paralela do `Apriori` faz uso de estruturas de dados dinâmicas padrão do C++ que não necessariamente alocam os dados alinhados com a memória cache, impedindo a vetorização automática de partes específicas do código. Apesar disso, a execução do `Apriori` paralelo na arquitetura *manycore* obteve speedup de até 10 tal como a arquitetura *multicore*.

4. Trabalhos Relacionados

O `Apriori` é comumente implementado de forma paralela usando o padrão MapReduce [Yang et al. 2010, Li et al. 2012, Singh et al. 2014, Dasgupta 2015], principalmente para a plataforma Hadoop. Essas abordagens consistem basicamente de múltiplas iterações MapReduce, onde na fase de mapeamento realiza-se a contagem da frequência dos itens no conjunto de candidatos e na fase de redução realiza-se a agregação global das frequências, mantendo-se apenas o conjunto dos itens considerados frequentes.

As principais abordagens utilizadas para paralelizar o algoritmo `Apriori` em arquiteturas de memória compartilhada envolvem particionar a coleção dos candidatos gerados entre os processadores, de maneira que a contagem da frequência dos candidatos seja realizada em paralelo [Jin et al. 2005, Parthasarathy et al. 2001]. Nestes dois casos o `Apriori` é implementado em clusters SMP.

Alguns trabalhos propõem uma implementação paralela para arquiteturas de memória compartilhada de variações do algoritmo FP-Growth para mineração de padrões frequentes [Zaiane et al. 2001, Chen et al. 2006]. A paralelização do FP-Growth possui uma complexidade em gerenciar o acesso compartilhado à FP-Tree, uma solução geralmente adotada envolve gerar partições da FP-Tree que serão processadas em paralelo.

O Algoritmo DMTA (*Distributed Miltithread `Apriori`*), proposto por [Bolina and Pereira 2013], é uma proposta de um novo método de balanceamento de carga do `Apriori`, baseado no DPA (*Distributed Parallel Apriori*). O DMTA é implementado neste trabalho usando um sistema híbrido com OpenMP e MPI.

Este trabalho se difere de todos os outros citados anteriormente, por apresentar uma implementação do algoritmo `Apriori` no padrão MapReduce utilizando OpenMP avaliada em arquiteturas *multicore* e *manycore*.

5. Conclusões

Este trabalho apresentou uma implementação paralela do algoritmo de regras de associação `Apriori` em OpenMP. Os resultados mostraram que a aplicação é escalável tanto em uma arquitetura *manycore* quanto em *multicore*.

Como trabalho futuro, podemos otimizar o `Apriori` Paralelo utilizando-se da técnica de cache blocking para melhor exploração dos mecanismos automáticos de vetorização tanto da arquitetura *multicore* quanto do *manycore*. Pode-se testar o `Apriori` em bases de dados maiores e implementar outros algoritmos de regras de associação de forma paralela. Adaptar a aplicação para GPU e para ambientes distribuidos.

Referências

Bolina, A. C., P. D. A. E. A. A. A. and Pereira, M. R. (2013). Uma nova proposta de paralelismo e balanceamento de carga para o algoritmo apriori. *Revista de Sistemas de Informação da FSMA*, (11):33–41.

Borkar, S. (2007). Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749.

Castro, M., Góes, L. F. W., and Méhaut, J.-F. (2014). Adaptive thread mapping strategies for transactional memory applications. *Journal of Parallel and Distributed Computing*, 74(9):2845–2859.

Chen, D., Lai, C., Hu, W., Chen, W., Zhang, Y., and Zheng, W. (2006). Tree partition based parallel frequent pattern mining on shared memory systems. In *Parallel and Distributed Processing Symposium, 2006*, pages 8–pp.

Dasgupta, S. (2015). Study of various parallel implementations of association rule mining algorithm. *American Journal Of Advanced Computing*, 2(1):12—16.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18.

Hill, M. D. and Marty, M. R. (2008). Amdahl's law in the multicore era. *Computer*, (7):33–38.

Jin, R., Yang, G., and Agrawal, G. (2005). Shared memory parallelization of data mining algorithms: techniques, programming interface, and performance. *Knowledge and Data Engineering, IEEE Transactions on*, 17(1):71–89.

Klemm, M. and Terboven, C. (2013). Full throttle: Openmp 4.0. *The Parallel Universe*, 16:6–16.

Li, N., Zeng, L., He, Q., and Shi, Z. (2012). Parallel implementation of apriori algorithm based on mapreduce. In *Software Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing (SNPD), 13th ACIS International Conference on*, pages 236–241.

Lichman, M. (2013). UCI machine learning repository.

McCraw, H., Ralph, J., Danalis, A., and Dongarra, J. (2014). Power monitoring with papi for extreme scale architectures and dataflow-based programming models. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 385–391. IEEE.

Parthasarathy, S., Zaki, M. J., Ogihara, M., and Li, W. (2001). Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems*, 3(1):1–29.

Rao, S. and Gupta, P. (2012). Implementing improved algorithm over apriori data mining association rule algorithm. *proceeding of IJCST, ISSN*, page 489–493.

Singh, S., Garg, R., and Mishra, P. K. (2014). Review of apriori based algorithms on mapreduce framework. In *International Conference on Communication and Computing (ICC)*, pages 593–604.

Yang, X. Y., Liu, Z., and Fu, Y. (2010). Mapreduce as a programming model for association rules algorithm on hadoop. In *Information Sciences and Interaction Sciences (ICIS), 3rd International Conference on*, pages 99–102.

Zaiane, O., El-Hajj, M., and Lu, P. (2001). Fast parallel association rule mining without candidacy generation. In *Data Mining, Proceedings IEEE International Conference on*, pages 665–668.