Exploração do Paralelismo em Algoritmos de Mineração de Dados com Pthreads, OpenMP, FastFlow, TBB e Phoenix++

Gabriell Araujo, Cleverson Ledur, Dalvan Griebler, Luiz G. Fernandes

[1]Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Grupo de Modelagem de Aplicações Paralelas (GMAP), Porto Alegre – RS – Brasil

{gabriell.araujo,cleverson.ledur,dalvan.griebler}@acad.pucrs.br
luiz.fernandes@pucrs.br

Resumo. *Com o objetivo de introduzir algoritmos de mineração de dados paralelos na DSL GMaVis, foram paralelizadas quatro aplicações com cinco interfaces de programação paralela. Este trabalho apresenta a comparação destas interfaces, a fim de avaliar qual oferece maior desempenho e produtividade de código. Os resultados demonstram que é possível atingir menor número de linhas de código e bom desempenho com OpenMP e FastFlow.*

1. Introdução

No contexto de processamento de grandes quantidades de dados, a DSL GMaVis [Ledur et al. 2015, Ledur 2016] foi proposta com o objetivo de facilitar a criação de visualizações geo-espacias, permitindo que usuários com pouco conhecimento em programação paralela possam abstrair o processamento de dados em paralelo na geração de visualizações. No entanto, a GMaVis ainda necessita de algoritmos que forneçam um pré-processamento inteligente dos dados. Logo, foi realizado um levantamento de algoritmos que poderiam ser embarcados no pré-processador da GMaVis, resultando no K-Means, Regressão Linear, PCA e Histograma[1].

A paralelização destes algoritmos é vital para a GMaVis, mas para isso é necessário considerar dois aspectos: (1) a paralelização deve utilizar uma interface/biblioteca que exija poucas linhas de código e forneça maior produtividade para facilitar a implementação no compilador; (2) a aplicação paralela deve apresentar ganho em desempenho. Portanto, foram utilizadas as seguintes interfaces: Pthreads, OpenMP, FastFlow [Aldinucci et al. 2012], TBB (Threading Building Blocks) e Phoenix++ [Talbot et al. 2011]. Este estudo visou analisar as implementações que apresentassem melhor desempenho e maior produtividade de código, para que elas sejam implementadas futuramente, de maneira nativa na GMaVis. Este artigo está organizado da seguinte forma. As Seções 2, 3, 4 e 5 descrevem as aplicações. A Seção 6 apresenta os resultados obtidos. Por fim, a Seção 7 conclui este artigo e apresenta trabalhos futuros.

2. K-Means

K-Means é um algoritmo não-supervisionado da área de aprendizado de máquina, que consegue particionar um volume de dados em grupos, classificando-os de acordo com os parâmetros estabelecidos. O código 1 apresenta o algoritmo de forma abstraída.

```
1 while (modificado){
2   modificado = false;
3   for (int i = 0; i < num_pontos; i++){//escalonamento estatico das threads
4     //encontra o centroide mais proximo do ponto i
5   }//sincronizacao das threads
6   for (int i = 0; i < num_centroides; i++){//escalonamento estatico das threads
```

[1]Os algoritmos sequenciais foram obtidos em https://github.com/kozyraki/phoenix.

```
7       //calcula a media do centroide i
8     }//sincronizacao das threads
9   }
```

Código 1. Trecho de código K-Means com maior custo computacional.

O algoritmo consiste em um laço `while` que possui dois blocos de código. No primeiro bloco temos um laço `for` paralelizável onde, considerando um conjunto de pontos $x = (x_0, ...x_n)$ e k centróides $m = (m_0, ...m_k)$, o algoritmo identifica qual é o centróide m_j mais próximo de cada ponto x_i. Para começar a execução do segundo bloco do algoritmo, primeiramente as *threads* são sincronizadas, pois é necessário ter os resultados de m adequados para cada ponto, para que então se possa calcular a média de x de cada centróide, e então, este laço também é paralelizado. Após isso, as threads são sincronizadas e o laço é executado novamente caso algum x_i tenha trocado de grupo.

3. Regressão Linear

O algoritmo de Regressão Linear após processar n valores $(x_i, y_i), i = 1, ..., n$, é capaz de estimar o comportamento esperado de um y para o qual é dado apenas o valor de x, através da fórmula $y = \alpha + x\beta$. O código 2 apresenta o trecho paralelizável do algoritmo de forma abstraída.

```
1 for(int i = 0; i < num_dados; i++){//escalonamento estatico das threads
2   //computa dados da posicao i e armazena resultados da thread
3 }//sincronizacao e combinacao dos resultados das threads
```

Código 2. Trecho de código Regressão Linear com maior custo computacional.

A região paralelizável do algoritmo consiste em um laço `for` que computa o somatório das variáveis x e y, acumulando estas em variáveis globais. Neste caso, existem dependências nas iterações do laço, ou seja, como os valores são acumulados nas variáveis de escopo global, o resultado de uma iteração depende da iteração anterior. Desta forma, foi criada uma estrutura de dados para *threads*, onde em vez de ler e escrever dados nas variáveis globais, cada *thread* computa um subconjunto de x e y e armazena os valores na sua estrutura, sendo assim, as iterações do laço `for` podem ser executadas em paralelo, eliminando as dependências do laço. No final, as *threads* são sincronizadas e os valores computados pelas *threads* são combinados, atualizando as variáveis globais.

4. PCA

O algoritmo *Principal Component Analysis* (PCA) é uma técnica que identifica relações entre as características dos dados, analisando-os em busca de sua redução, eliminando sobreposições, bem como escolhendo formas mais representativas destes. Tendo os dados organizados em uma matriz, o algoritmo calcula a média de cada linha e após gera uma matriz de covariância. O código 3 apresenta o algoritmo de forma abstraída.

```
1 for (int i = 0; i < num_linhas; i++){//escalonamento estatico das threads
2   //calcula a media da linha i
3 }//sincronizacao das threads
4 for (int i = 0; i < num_linhas; i++){//escalonamento dinamico das threads
5   //calcula covariancia
6 }//sincronizacao das threads
```

Código 3. Trecho de código PCA com maior custo computacional.

No cálculo das médias das linhas, o laço `for` é paralelizado usando escalonamento estático. Já no cálculo da matriz de covariância, o laço `for` é a região paralelizável da função, mas as iterações não foram divididas uniformemente para as *threads*. Neste trecho em específico, a quantidade de operações computacionais são desbalanceadas. Diante disso, optamos pelas *threads* calcularem as iterações de maneira dinâmica.

5. Histograma

Histograma é um algoritmo que analisa uma imagem e computa a sua frequência *RGB*. Paralelizamos este algoritmo com a mesma estratégia adotada no algoritmo de Regressão Linear. As *threads* varrem regiões da imagem em paralelo, computando a frequência *RGB* dos pixeis e armazenando nas suas estruturas. No final, os valores encontrados pelas *threads* são combinados, obtendo-se então o histograma da imagem. O código 4 apresenta o algoritmo de forma abstraída.

```
1
2 for (int i = 0; i < tamanho_imagem; i+=3){//escalonamento estatico das threads
3   //computa frequencia RGB do pixel e armazena resultado da thread
4 }//sincronizacao e combinacao dos resultados das threads
```

Código 4. Trecho de código do Histograma com maior custo computacional.

6. Resultados

Antes de adentrarmos nos resultados, salientamos que a biblioteca TBB não foi utilizada nos algoritmos de Regressão Linear e Histograma, pois esta não fornece recursos necessários para tratar as zonas críticas destes algoritmos (leituras e escritas simultâneas em uma mesma variável). Os resultados da avaliação de desempenho são apresentados na Figura 1, enquanto os resultados da avaliação de produtividade são apresentados na Figura 2 e na Tabela 1. Para avaliar o desempenho foram executadas três diferentes cargas de dados para cada algoritmo. Para cada carga de dados, o algoritmo foi executado usando entre um e doze núcleos de processamento. Utilizamos a média aritmética de dez execuções para cada combinação de threads e cargas. Os testes foram realizados em um computador Blade Dell PowerEdge com dois processadores Intel Xeon E5645 e 24 Gbytes de memória. Para analisar a produtividade de código, foi utilizada a quantidade de linhas de código de cada algoritmo. Logo, foi possível calcular a taxa de aumento de linhas de código necessária para a paralelização com cada interface.

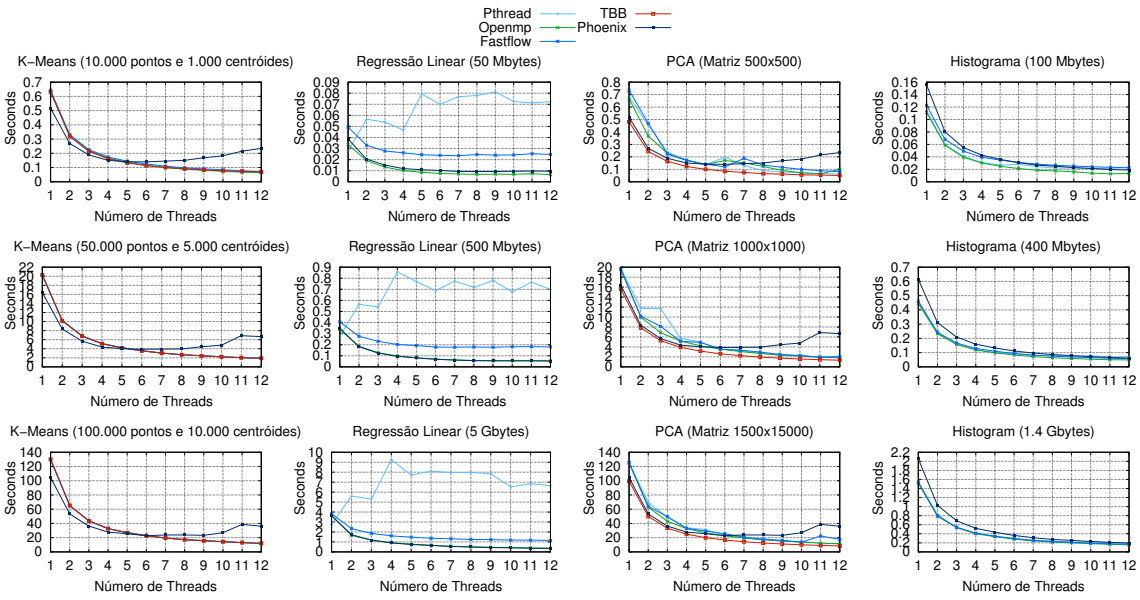

Figura 1. Resultados de desempenho.

Pthreads apresentou um melhor desempenho (exceto no Regressão Linear), porém necessita de uma refatoração significativa do código em baixo nível. Já os resultados de

OpenMP, TBB e FastFlow foram similares e levemente inferiores ao da Pthreads, mas são as bibliotecas que mais facilitam a paralelização de algoritmos, pois consistem em abstrações de alto nível. É necessário notar que embora reduzam o esforço, elas diminuem a flexibilidade de código por não fornecerem recursos de baixo nível das threads. Logo, não é possível paralelizar certos algoritmos ou adotar estratégias otimizadas.

Phoenix++ exige refatoração de código maior que as demais bibliotecas, e por implementar o padrão MapReduce, nem todos algoritmos se adequam ao uso desta. Destacamos também que a mesma implementa diversos recursos de alto nível, os quais possuem um elevado custo computacional, ocasionando em perda de desempenho em certos tipos de aplicações. Dentre todas as bibliotecas, foi a que apresentou menor desempenho.

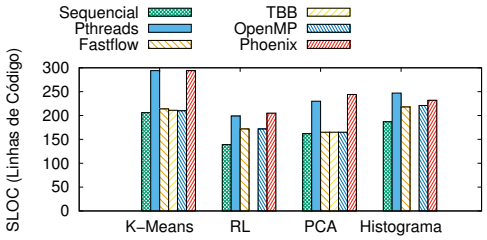

Figura 2. Total em linhas de código.

Tabela 1. Aumento de linhas de código para aplicar paralelismo.

Interface	K-Means	RL	PCA	Hist
Pthreads	42,72%	43,17%	41,98%	32,09%
FastFlow	3,88%	23,74%	1,85%	16,58%
TBB	2,43%	–	1,85%	–
OpenMP	1,94%	23,74%	1,85%	18,18%
Phoenix	42,72%	47,48%	50,62%	24,06%

Conforme a análise efetuada, podemos concluir que as bibliotecas OpenMP e FastFlow foram as que obtiveram melhores resultados considerando produtividade de código, desempenho e possibilidade de implementação em todas as aplicações.

7. Conclusões

O artigo apresentou os resultados obtidos da comparação de cinco interfaces de programação paralela na exploração de paralelismo de aplicações de mineração de dados. Foi observado que diferentes abordagens permitem maior ganho de desempenho, mas reduzem a produtividade de código. Como trabalhos futuros iremos implementar os algoritmos paralelizados que obtiveram melhor ganho de desempenho e produtividade de código dentro do compilador da GMaVis, de forma que sejam gerados automaticamente pelo compilador já aplicando paralelismo para arquiteturas multi-core.

Referências

[Aldinucci et al. 2012] Aldinucci, M., Danelutto, M., Kilpatrick, P., and Torquati, M. (2012). Fastflow: High-Level and Efficient Streaming on Multi-Core. In Pllana, S. and Xhafa, F., editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, pages 261–281. Wiley, Spain.

[Ledur 2016] Ledur, C. (2016). GMaVis: A Domain-Specific Language for Large-Scale Geospatial Data Visualization Supporting Multi-core Parallelism. Master's thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil.

[Ledur et al. 2015] Ledur, C., Griebler, D., Manssuor, I., and Fernandes, L. G. (2015). Towards a Domain-Specific Language for Geospatial Data Visualization Maps with Big Data Sets. In *ACS/IEEE International Conference on Computer Systems and Applications*, AICCSA'15, page 8, Marrakech, Marrocos. IEEE.

[Talbot et al. 2011] Talbot, J., Yoo, R. M., and Kozyrakis, C. (2011). Phoenix++: Modular MapReduce for Shared-memory Systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 9–16, New York, NY, USA. ACM.