Benchmarking da Aplicação de Comparação de Similaridade entre Imagens com Flink, Storm e SPar

Leonardo Gibrowski Faé, Dalvan Griebler, Isabel H. Manssour

[1] Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brasil

leonardo.fae@edu.pucrs.br,{dalvan.griebler,isabel.manssour}@pucrs.br

Resumo. *Este trabalho apresenta comparações de desempenho entre as interfaces de programação SPar, Apache Flink e Apache Storm, no que diz respeito à execução de uma aplicação de comparação de imagens. Os resultados revelam que as versões da SPar apresentam um desempenho superior quando executadas com um grande número de threads, tanto em termos de latência quanto de throughput (a SPar tem um throughput cerca de 5 vezes maior com 40 workers).*

1. Introdução

Com a popularização de métodos de geração automática de imagens, torna-se cada vez mais relevante que possamos processá-las em paralelo. Assim, este trabalho propõe uma análise e comparação de desempenho obtidos com a utilização dos *frameworks* Apache Flink, Apache Storm e SPar. A última foi escolhida por ser a solução desenvolvida em [Griebler et al. 2017a], com a qual queremos comparar em relação às outras duas, que são opções mais populares no mercado.

Nosso objetivo é realizar uma avaliação das ferramentas Flink, Storm e SPar com um *pipeline* mais robusto (contendo mais de 1 estágio paralelo) de processamento de imagens, dando assim, continuidade aos estudos apresentados em [Faé et al. 2022]. A aplicação de comparação de similaridade entre imagens desenvolvida é baseada no OpenCV. O trabalho está organizado da seguinte maneira: inicialmente descrevemos a implementação da aplicação e como foi feita a paralelização com cada *framework* na Seção2. Depois apresentamos os resultados dos experimentos (Seção 3) e finalizamos com as nossas conclusões (Seção 4).

2. Implementação

Nesta seção, apresentamos a implementação da aplicação sequencial e com os diferentes *frameworks*.

2.1. *Imgcmp* - uma aplicação de comparação de imagens

Desenvolvemos o *imgcmp* com o propósito específico de gerar um volume de trabalho significativo para mensurar o desempenho dos *frameworks* SPar, Flink e Storm. Esta aplicação busca imitar o processo de comparação encontrado na aplicação *Ferret* [Lv et al. 2006], uma aplicação mais conhecida e usada para *benchmarks* similares [Griebler et al. 2018]. Criamos uma aplicação nova devido a dificuldades que se apresentaram ao tentar usar o Ferret pela *Java Native Interface* (JNI). A aplicação possui 5 estágios: (1) ler as imagens, (2) extrair os seus descritores; (3) compará-los com todas as imagens extraídas previamente; (4) contar o número de boas correspondências (*matches*) e colocá-las em ordem,

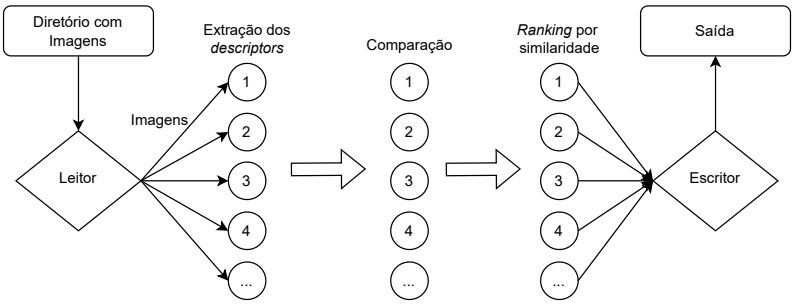

Figura 1. *Pipeline* do *imgcmp*

da mais similar para a menos similar; e (5) escrever a saída. A Figura 1 mostra a estratégia geral utilizada. O método de comparação em si, que envolve a extração dos descritores, utiliza o algoritmo descrito em [Bay et al. 2006], e o critério para selecionar uma "boa correspondência" está explicado em [Lowe 2004].

2.2. Flink e Storm

O Apache Flink e o Apache Storm são *frameworks* escritos em Java com o propósito de facilitar a utilização de *clusters* para processamento de *streams* de dados. Como o *imgcmp* foi originalmente escrito em C++ (pois utilizava a biblioteca OpenCV[1]), o código em Java tem que se comunicar com ele fazendo uso da JNI. A JNI introduz um custo computacional extra para efetuar a "tradução" das chamadas de função em C++ para Java. Utilizamos as *bindings* para Java geradas automaticamente pela própria OpenCV para isso.

2.3. SPar

A SPar[Griebler et al. 2017b] é um *framework* desenvolvido em C++ cujo objetivo é fornecer aos seus usuários a possibilidade de paralelizar códigos sequenciais com facilidade. Assim, para implementar o *imgcmp* na SPar, basta copiar e colar o código sequencial e colocar as anotações relevantes (no estilo C++11, com duplo colchetes - "`[[]]`") no código. Além disso, a SPar permite escolher diversas *runtimes* de paralelização diferentes. Neste trabalho, utilizamos o FastFlow[2] e o OpenMP[3].

3. Experimentos e Resultados

Para os testes, utilizamos uma máquina composta por dois processadores Intel(R) Xeon(R) Silver 4210 2.20GHz (total de 40 *threads*), com 64 GB de memória RAM, funcionando com o sistema operacional *Ubuntu Server 64-bits* e com versão de *kernel* 5.4.0-91-*generic*. Ainda, usamos as versões 11 do Java, 2.2.0 do Storm, 1.12.0 do Flink e 9.3.0 do compilador G++. O *standard* de C++ foi de 2011 (conforme exigido pela SPar) e todos os códigos de C++ foram compilados com *flag* de otimização "-O3". Todos os programas foram executados 10 vezes, com um número de *threads* variando de 1 a 40. As Figuras 3 e 2 mostram os resultados de latência e *throughput*, respectivamente, dos experimentos com 0 (serial) até número de trabalhadores paralelos por estágio configurados. *SPAR_FF*

[1]https://opencv.org/
[2]http://calvados.di.unipi.it/
[3]https://www.openmp.org/

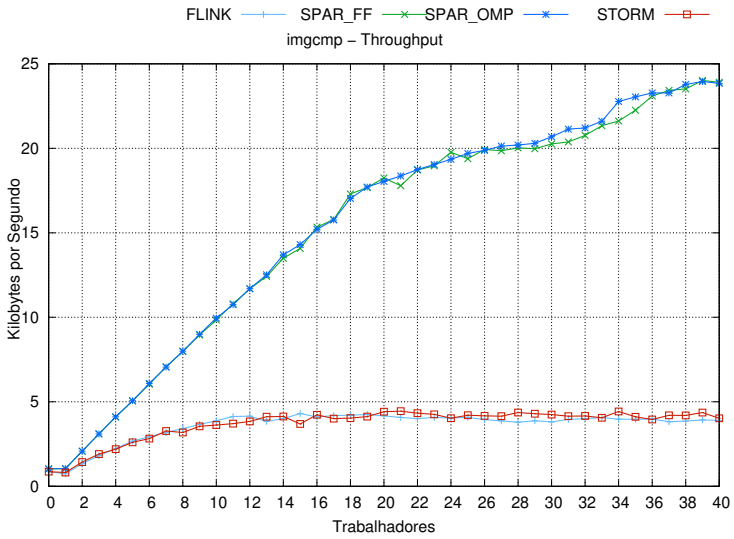

Figura 2. Imgcmp throughput

refere-se à SPar com a *runtime* do FastFlow, e *SPAR_OMP* à SPar com a *runtime* do OpenMP.

Os resultados de *throughput*(2) mostram que os *frameworks* escritos em Java não escalam na medida em que aumentamos o número de trabalhadores, sendo que o ganho de desempenho parece estagnar por volta de 14 *workers*. Já as versões da SPar apresentam maior desempenho de forma consistente. O ganho de desempenho diminui a partir de 20 trabalhadores porque começamos a usar *hyperthreading*, de modo que esse resultado era esperado.

Os resultados de *latência*(3) mostram o Flink como a aplicação com menor latência quando poucos trabalhadores são utilizados. Na medida em que aumentamos o número de trabalhadores, sua latência aumenta, até que eventualmente atinge a latência do Storm. O Storm, por sua vez, diminui muito pouco a sua latência a partir de aproximadamente 15 trabalhadores. Esses resultados de latência poderiam explicar os resultados de *throughput*, examinados anteriormente. Principalmente para o Flink, como a latência aumenta muito, os ganhos de velocidade com a paralelização aparentam estar sendo compensados pelas perdas com a latência e custos de comunicação.

Finalmente, o acréscimo final de latência da SPar com FastFlow pode ser explicado devido ao fato que o FastFlow reserva duas *threads* para fazerem exclusivamente o papel de entrada e saída. Assim, esperamos uma queda de desempenho quando tentamos usar todas as *threads* disponíveis no sistema. Neste caso, essa queda se manifestou no aumento de latência com 39 e 40 trabalhadores.

4. Conclusões

Em termos de *frameworks* para aplicações de *streaming*, a SPar aparenta fornecer uma solução com uma escalabilidade muito mais expressiva do que as alternativas examinadas. Mesmo para uma aplicação complexa como o *imgcmp*, ela consegue fazer uso dos trabalhadores disponíveis de modo a garantir um aumento consistente de *throughput*. Além disso, também apresenta a menor latência na medida em que aumentam os trabalhadores

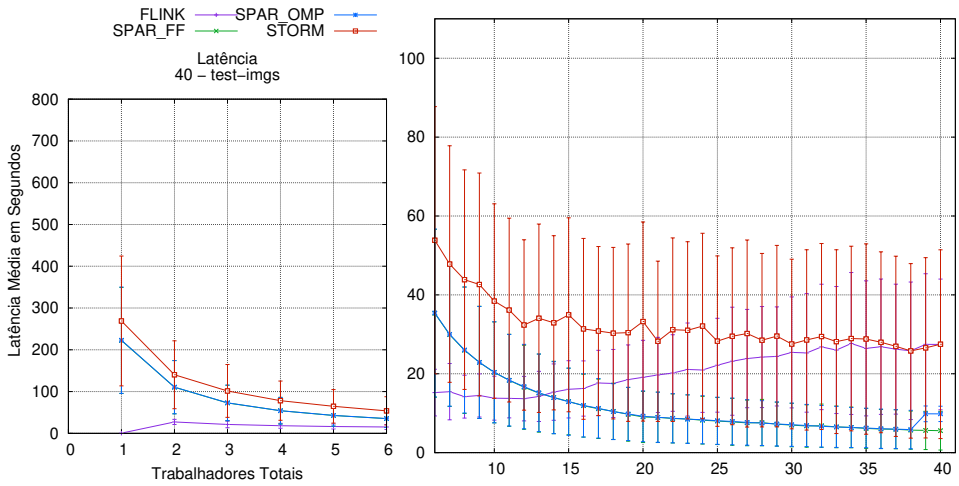

Figura 3. Latência do *imgcmp*

disponíveis.

Para trabalhos futuros, pode-se examinar com precisão por que a latência do Flink aumenta de forma tão expressiva, bem como por que a latência do Storm para de diminuir a partir de 15 trabalhadores. Poder-se-ia também examinar o desempenho das versões da aplicação em um ambiente distribuído, onde Storm e Flink talvez apresentassem um desempenho melhor, já que foram desenvolvidas com esses ambientes em mente.

Referências

Bay, H., Tuytelaars, T., and Van Gool, L. (2006). Surf: Speeded up robust features. In Leonardis, A., Bischof, H., and Pinz, A., editors, *Computer Vision – ECCV 2006*, pages 404–417, Berlin, Heidelberg. Springer Berlin Heidelberg.

Faé, L., Griebler, D., and Manssour, I. (2022). Aplicação de vídeo com flink, storm e spar em multicores. In *Anais da XXII Escola Regional de Alto Desempenho da Região Sul*, pages 13–16, Porto Alegre, RS, Brasil. SBC.

Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. (2017a). SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):1740005.

Griebler, D., Hoffmann, R. B., Danelutto, M., and Fernandes, L. G. (2017b). Higher-Level Parallelism Abstractions for Video Applications with SPar. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing*, ParCo'17, pages 698–707, Bologna, Italy. IOS Press.

Griebler, D., Hoffmann, R. B., Danelutto, M., and Fernandes, L. G. (2018). High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. *International Journal of Parallel Programming*, 47(1):253–271.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110.

Lv, Q., Josephson, W., Wang, Z., Charikar, M., and Li, K. (2006). Ferret: A toolkit for content-based similarity search of feature-rich data. *Operating Systems Review (ACM)*, 40(4):317–330.