Aplicação de Vídeo com Flink, Storm e SPar em Multicores

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 processamento de vídeo. Os resultados revelam que as versões da SPar apresentam um desempenho superior, enquanto o Apache Storm apresentou o pior desempenho.*

1. Introdução

Apesar do interesse na área de processamento paralelo de vídeos, não encontramos muitos trabalhos que tratam dessa questão com os *frameworks* do Apache Flink [Foundation 2021a] e Apache Storm [Foundation 2021b] em sistemas *multicore*. Assim, o presente trabalho propõe uma análise e comparação de desempenho obtidos com a utilização desses *frameworks*.

O objetivo é comparar com a SPar, que é uma *Domain Specific Language* (DSL) que permite expressar paralelismo em alto nível, através de anotações em código de C++ para sistemas *multicore* [Griebler et al. 2017a]. Além disso, ela permite integrações com diversas *runtimes*, sendo que neste trabalhado foram usadas *Threading Building Blocks* (TBB) [Hoffmann et al. 2020] e *OpenMP* (OMP) [Hoffmann et al. 2022]. O Apache Flink e o Apache Storm são *frameworks* desenvolvidos em Java, especificamente para lidar com *streams* de grandes volumes de dados fazendo uso de paralelismo. A hipótese é que a SPar apresente maior desempenho, sobretudo porque não incorre no *overhead* da *runtime* do Java. Este trabalho dá continuidade ao trabalho de [Mello et al. 2021], no qual foi feita uma comparação do desempenho entre Flink e SPar para o algoritmo de compressão Bzip2.

Este artigo foi organizado da seguinte forma. Inicia-se explicando a aplicação de vídeo, passando por uma breve apresentação da implementação da sua versão paralela com todos os *frameworks* utilizados (Seção 2). Após, analisamos os resultados obtidos (Seção 3) e finalizamos com nossas conclusões (Seção 4).

2. Implementação

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. Assim, ambos contém uma *runtime* robusta, com vários níveis de redirecionamentos. Processar dados nesses *frameworks* envolve inicializar um nodo mestre (*Flink Master* para o Flink e *Nimbus* para o Storm), que será responsável por distribuir tarefas a nodos trabalhadores. Essas tarefas consistem em programas descrevendo o processamento desejado. Os nodos mestres então iniciam-no conforme a configuração fornecida pelo usuário (um processamento desse tipo se chama de *Job* no Flink e *Topology* no Storm). Internamente,

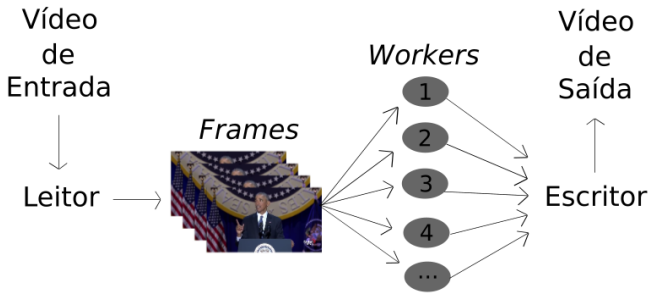


Figura 1. Estratégia de paralelismo usada.

o processamento em si possui mais subdivisões, conforme o número de tarefas a serem executadas. Essas tarefas podem ser divididas ainda mais, se for possível executar várias delas ao mesmo tempo no nodo. O Storm também exige que seja executado com o Apache Zookeeper [Foundation 2021c], um serviço que fornece coordenação para sistemas distribuídos. O Flink usa outros mecanismos para lidar com esse problema, evitando o *overhead* no qual o Storm incorre.

O programa utilizado para os testes foi o *Person-Recognizer*[1], que realiza reconhecimento facial. Os algoritmos específicos que utiliza para tanto fogem do escopo deste trabalho; nosso interesse está apenas nas etapas de processamento realizadas pelo programa. Dessas, há o treinamento do reconhecedor, que não é paralelizável, e o reconhecimento propriamente dito, que pode ser paralelizado, uma vez que a análise de um *frame* do vídeo pode ocorrer de forma totalmente independente da análise de outro.

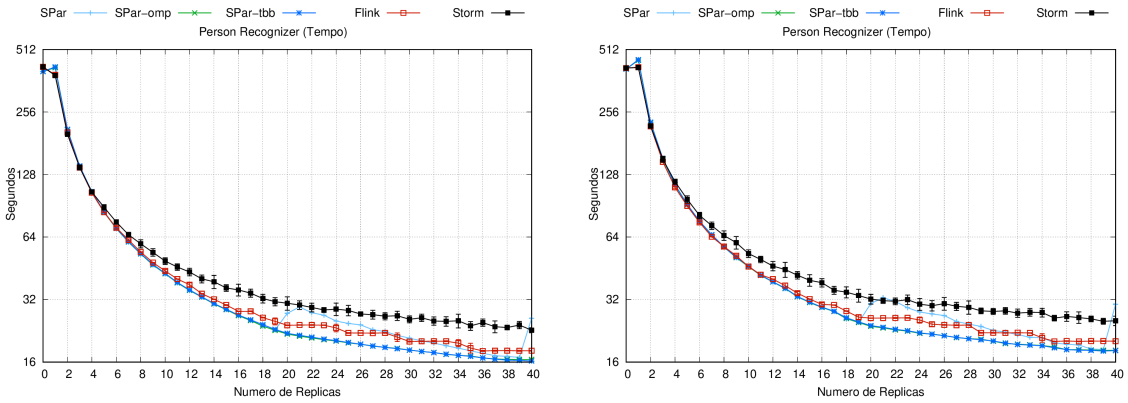
Assim, para cada *framework* analisado, foi criada uma versão do programa *Person-Recognizer* adaptada de [Griebler et al. 2017b], com o mínimo de modificações necessárias, para que executasse explorando o potencial paralelismo verificado. A estratégia utilizada foi um *farm*, no qual um leitor separa os *frames* e os envia a múltiplos *workers*, que executam o algoritmo de reconhecimento facial, como ilustra a Figura 1. Os *workers* enviam seus resultados para um escritor, que os ordenam utilizando a estratégia descrita em [Griebler et al. 2018]. As três Etapas formam um *pipeline* entre si, com um core dedicado à leitura e outro à escrita. Além de realizar a leitura, a primeira Etapa também realiza o treinamento do reconhecedor facial com base em imagens estáticas.

O processor de paralelização da SPar foi descrito em [Griebler et al. 2017b]. Para o Apache Flink e Apache Storm, as funções relevantes foram importadas para Java usando a *Java Native Interface*. O próprio projeto do *OpenCV* já gera um arquivo *.java*, facilitando a integração. O código em si foi escrito de modo a replicar o código original em C++ o mais próximo possível. Para ambos os *frameworks*, foi necessário dividir o processamento em três Etapas, correspondendo à leitura dos *frames*, processamento desses, e escrita ao final. Cada uma dessas Etapas corresponde a uma classe no código. As classes estabelecem uma *pipeline* entre si, de modo que a Etapa 2 começa logo após o primeiro *frame* ser processado pela Etapa 1. Da mesma forma para as Etapas 2 e 3.

3. Resultados

Para os testes, foram utilizadas a versão 11 do Java, a versão 2.2.0 do Storm, a versão 1.12.0 do Flink e a versão 9.3.0 do compilador G++. Ainda, o *standard* de C++ foi de

[1] `https://github.com/EyalAr/Person-Recognizer`



(a) Vídeo do Obama com plateia

(b) Vídeo apenas do Obama

Figura 2. Média do tempo de execução, em segundos.

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, para realizar a análise de dois vídeos de 15 segundos (450 frames), extraído de discursos do ex-presidente norte-americano Barack Obama. O primeiro mostra a plateia durante aproximadamente metade da sua duração, enquanto o segundo mostra apenas o rosto do ex-presidente. Os vídeos foram escolhidos devido à diferença de número de pessoas entre eles, o que daria mais ou menos oportunidades ao programa detectar rostos. Além disso, como o ex-presidente é uma influente pessoa pública, encontrar imagens de seu rosto para utilizar na Etapa de treinamento, antes de iniciar a leitura, é mais fácil e mais representativo de como o programa seria usado em uma situação real de produção (isto é, com imagens retiradas da *internet*).

A máquina usada é 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*. Para efeitos de comparação, duas versões sequenciais do programa, um Java e outra em C++, foram executadas antes de realizar os experimentos. Essas levaram em média em torno de 410 segundos, com a versão em Java levando cerca de 10 segundos a mais. A Figura 2 mostra a média aritmética e o desvio padrão do tempo de execução das versões paralelas dos programas. O eixo x representa o grau de paralelismo de 1 a 40, correspondendo ao número de *workers* da Figura 1.

Para verificar a implementação, comparamos os *hashes MD5* dos arquivos de saída. As versões sequenciais em Java e C++ resultam em *hashes* diferentes entre si, mas visualmente os vídeos de saída aparentam idênticos. Quanto às versões paralelizadas, essas apresentam *hashes* iguais aos seus equivalentes sequenciais.

De início, percebe-se que as duas versões mais eficientes são a SPar com os *runtimes* especializados (OpenMP e TBB), sendo que o gráfico não revela diferenças significativas entre uma e outra, estando suas linhas sobrepostas. A partir de grau de paralelismo 5, o tempo de execução do Storm passou a ser maior que todos os outros, e assim se manteve, com a diferença de eficiência aumentando, até grau 40, onde a SPar o superou. Já o Flink conseguiu manter uma eficiência mais próxima das versões mais eficientes do

programa, chegando a ser até mais eficiente a que SPar sem *runtime* especializada entre graus de paralelismo 20 e 30.

Conclui-se pelos resultados que a sobrecarga gerada pela *runtime* tanto do Java quanto do Flink e do Storm impactaram o desempenho dos programas. A ineficiência do Storm em comparação ao Flink pode ser explicada pelo fato que o Flink oferece um serviço de supervisão embutido dentro dele mesmo (exercido pelo que a documentação do Flink chama de *Job Manager*), enquanto o Storm é executado com o Apache Zookeeper, um outro *framework* totalmente independente do Storm, para fornecer a funcionalidade de supervisão. Finalmente, a SPar ser o *framework* mais rápido se deve também ao fato que sua execução não depende de toda uma preparação de ambiente de *runtime* específica. Não há necessidade de se inicializar o Apache Zookeeper ou um *Job Manager*; o processamento inicia quase que imediatamente quando da execução do programa.

4. Conclusões

O trabalho confirmou a hipótese inicial de que as versões paralelas do programa em C++ com SPar seriam mais eficientes que os seus equivalentes em Java. Apesar disso, verificou-se que em alguns casos o Apache Flink superou o desempenho da SPar sem *runtime* especializada (FastFlow) em tempo de execução. Como trabalhos futuros, pretende-se continuar a comparação de desempenho desses *frameworks* com outras aplicações de *stream*.

Referências

Foundation, A. S. (2021a). Apache flink® — stateful computations over data streams.

Foundation, A. S. (2021b). Apache storm™.

Foundation, A. S. (2021c). Apache zookeeper™.

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). Stream Parallelism with Ordered Data Constraints on Multi-Core Systems. *Journal of Supercomputing*, 75(8):4042–4061.

Hoffmann, R. B., Griebler, D., Danelutto, M., and Fernandes, L. G. (2020). Stream Parallelism Annotations for Multi-Core Frameworks. In *XXIV Brazilian Symposium on Programming Languages (SBLP)*, SBLP'20, pages 48–55, Natal, Brazil. ACM.

Hoffmann, R. B., Löff, J., Griebler, D., and Fernandes, L. G. (2022). OpenMP as runtime for providing high-level stream parallelism on multi-cores. *The Journal of Supercomputing*.

Mello, F., Griebler, D., Manssour, I., and Fernandes, L. G. (2021). Compressão de dados em multicores com flink ou spar? In *Anais da XXI Escola Regional de Alto Desempenho da Região Sul*, pages 77–80, Porto Alegre, RS, Brasil. SBC.