Um estudo sobre uso do MPI para uma aplicação de detecção de picos em data streams

Caetano Müller, Dalvan Griebler

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

caemuller1@gmail.com,dalvan.griebler@pucrs.br

Abstract. *Aplicações de data stream podem ser implementadas com diferentes interfaces de programação paralela. Neste artigo, realizou-se um estudo e implementação da aplicação Spike Detection com MPI e a comparou-se com versões usando Flink, Storm e Windflow. Avaliou-se o trouhgput e conclui-se que a implementação com Windflow apresenta o melhor desempenho, enquanto as versões com MPI tiveram um throughput inferior as demais soluções.*

1. Introduction

A necessidade para paralelismo em aplicações de *stream* tem crescido constantemente nos últimos anos devido ao aumento de geração de dados em diversos setores. Este paradigma gera aplicações de *stream* de dados ou seja, um fluxo não necessariamente finito de dados sendo gerados para serem processados. O processamento desses dados pode ser constituído de qualquer função computável que, em alguns casos possuem alto custo de tempo e recursos computacionais. Uma das possíveis otimizações para programas com fluxo de dados contínuo é o uso de paralelismo, diversas formas de paralelismo de *stream* são estudados e desenvolvidos na literatura. O uso destas técnicas se justifica pela redução do tempo de execução da aplicação ou em casos de fim indeterminado, mais dados são processados pelo algoritmo em um determinado intervalo de tempo.

Diversas ferramentas foram desenvolvidas para diminuir a carga de trabalho no desenvolvimento de uma aplicação de *stream* paralela. Neste artigo focaremos em uma aplicação de *Dataflow* representada por um grafo de 4 nodos. Sua função é realizar a detecção de picos ao receber dados de sensores. Essa aplicação se chamada *Spike Detection* (SD). SD é melhor descrita em [Bordin et al. 2020]. Vale notar que SD representa uma aplicação sem condição de parada interna e pode possuir uma execução perpétua. Neste artigo usou-se a implementação de SD em 3 ferramentas, dentre elas duas desenvolvidas em Java (Flink e Storm) por [Bordin et al. 2020] e outra desenvolvida em C++ (WindFlow [Mencagli et al. 2021]). A implementação da aplicação SD neste artigo é realizada de 3 maneiras diferentes usando o MPI [Message Passing Interface Forum 2023] em C++. Em suma este trabalho busca realizar uma comparação do desempenho da SD implementada com Flink, Storm, Windflow e MPI.

A divisão deste artigo se da pelos trabalhos relacionados na segunda Seção 2. Informaçõe sobre a aplicação da SD em MPI são fornecidas na Seção 3. Uma descrição de como foram executados os experimentos, quais métricas foram extraídas e uma análise dos resultados são discutidos na Seção 4. Na Seção 5 são mostradas as conclusões finais obtidas com o trabalho e possíveis trabalhos futuros.

2. Trabalhos relacionados

A ideia de uso de *Benchmarks* para paralelismo de stream ja é trabalhado na literatura para a comparação do desempenho geral de diferentes bibliotecas. Um exemplo de um *Benchmark* deste tipo pode ser encontrado em [Bordin et al. 2020]. A implementação de técnicas específicas de programação para obter melhor desempenho de uma aplicação de *stream* de dados, também são encontradas na literatura como a ideia em [Garcia et al. 2022]. Vale notar que o [Garcia et al. 2022] também foca-se em um sistema multicore e não em um sistema distribuído. Este artigo também se restringe apenas a testes em apenas um sistema local.

3. Implementação

A implementação da aplicação *Spike Detection* segue uma topologia de fluxo de dados representado por um grafo de 4 vértices. A leitura do Dataset simulando envio de dados de sensores ocorre antes da execução da aplicação. Uma representação visual da aplicação em um grafo pode ser vista em 1. As mensagens são enviadas pelos nodos no sentido de *Source* ao *Sink*.

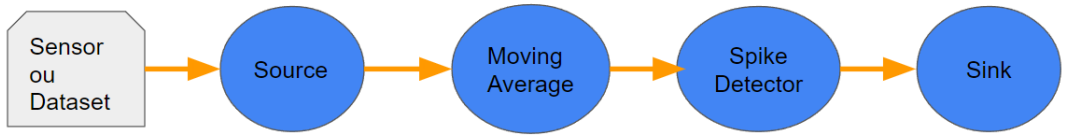

Figura 1. Representação de SD em um Grafo.

O Grafo apresentado usa os quatro vértices de maneira linear, porém, é possível gerar réplicas de cada um dos quatro estados do grafo para exploração de paralelismo. As mensagens são todas enviadas para as múltiplas réplicas de cada vértice criando assim a possibilidade de processar mais dados simultaneamente.

Para a implementação de SD com MPI com *spawn* de processos dinâmicamente foram usadas diferentes técnicas. A primeira testada foi um mecanismo de mensagens sob demanda chamada em 2 de *mpi ondemand*. Um dado é enviado para ser computado dos *Source* para o segundo vértice apenas se houver um pedido dele, ou seja, uma mensagem é enviada para o *source* demandando dados para serem processados. Vale notar que para o mecanismo sob demanda foi usado apenas comunicação não bloqueante na troca de mensagens. Estes referenciados em 2 respectivamente como *mpi non-blocking* e *blocking mpi*. Já para a implementação de *Spawn* estáticos de processos nos restringimos apenas à comunicação bloqueante.

A aplicação também foi implementada com o envio de mensagens pelo *Source* usando um mecanismo de *Round-Robin*, onde as mensagens são enviadas de maneira cíclica para os processos a sua frente na topologia. Usando o mecanismo de envio de mensagens *Round-Robin* foi implementada a aplicação de duas maneiras diferentes, uma usando a mesma comunicação não-bloqueante do mecanismo sob-demanda e outra usando uma estratégia de comunicação bloqueante no fluxo das mensagens.

O funcionamento da aplicação é uma leitura contínua de uma base de dados simulando a entrada de dados pelos sensores. para que seja possível obter as métricas

trabalhadas neste artigo, implementou-se um intervalo de tempo de 60 segundos, este intervalo representa o tempo total de execução da topologia, com o *Source* enviando dados *Downstream*. Após os 60 segundos de aplicação serem atingidos, é enviado à topologia uma mensagem finalizando os processos terminando no ultimo estágio *Sink* apresentado em 1. Para o caso de fim de execução é garantido que independente do grau de paralelismo de um estágio todas suas réplicas recebam a informação e repassem ela aos estados à sua frente na topologia, garantindo o fim da execução.

4. Experimentos

Os experimentos foram executados em uma máquina Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, contando com 20 núcleos e 40 *threads*. Cada *hyper-threaded* core tem 640KB privada L1, 20MB privada L2 e 27.5MB de L3 compartilhada. A máquina possui 141GB de *RAM*. O kernel era o Linux 5.4.0-135-generic e usava OS Ubuntu 20.04.5 LTS.

Nos experimentos houve foco no aumento de réplicas do segundo vértice do grafo. Todos experimentos foram executados 5 vezes para cada um dos diferentes números de replicas testados. Foi extraído o *throughput* das execuções ou seja, o número de dados de entrada processados por segundo. Todas as versões do SD foram testadas e estão representadas em 2. Vale notar que o *throughput* representado no gráfico está em escala de 10^6. Os valores na figura são as médias dos *throughputs* obtidos nas 5 execuções realizadas. Junto à cada valor está apresentado o desvio padrão das execuções.

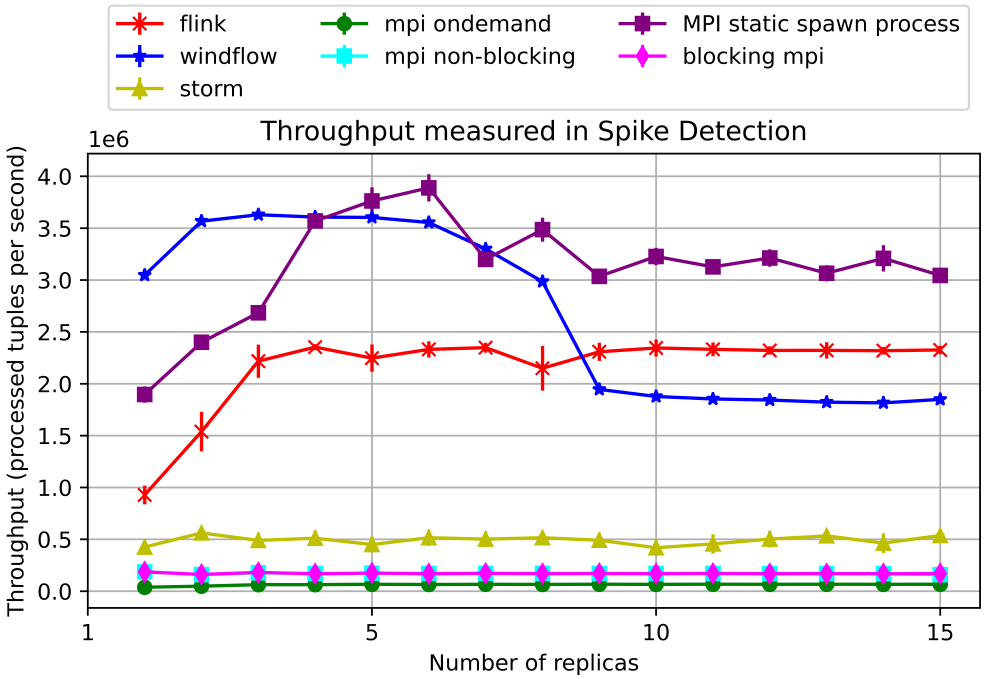

Figura 2. Throughput de SD em diferentes réplicas de Moving Average.

Implementações com inicialização de processos dinâmica no MPI obtiveram os piores resultados que comparados ao pico de performance do Windflow e MPI com *Spawn* estático foi em média 18 vezes inferior. Em caso de *Spawn* estático de processos o MPI

obteve em pico os melhores resultados dos experimentos realizados. Já o Flink em comparação ao Storm, ambos implementados em Java, obteve em seu melhor caso um *throughput* 4.6 vezes maior.

Entre os resultados das implementações em MPI podemos notar que ambas as versões implementadas com o mecanismo de envio de mensagens *Round-Robin* ficaram semelhantes, enquanto a versão sob-demanda de mensagens se encontra com um *throughtput* 44% inferior em média. E por todas as implementações com *Spawn* dinâmico não obtiveram resultados competitivos na decisão de melhore performance.

Os melhores resultados Em média foram obtidos pelo MPI e obteve o maior *throughput* do experimento em sua sexta réplica. Para maiores números de réplicas, o desempenho obteve um declínio e depois tende a estabilizar em $3.1M$ de dados processados por segundo já o Windflow regrediu em 25% até atingir essa estabilidade. Isso se dá pelo *Overhead* gerado, pois múltiplos processos estão ativos e seu estágio anterior não consegue gerar dados suficientes para todas suas réplicas criadas. Uma simples solução para resolver este problema é aumentar o grau de paralelismo do *Source*.

5. Conclusões

Esse trabalho trouxe uma implementação e análise de uma aplicação de paralelismo de *stream* com fluxo contínuo de dados. As implementações em MPI com *Spawn* dinâmico de processos tiveram um desempenho inferior até mesmo quando comparadas às versões em Java do código. Tais resultados nos motivam a continuar investigações do motivo das trocas de mensagem em MPI serem mais custosas em tempo quando seus processos são gerados dinamicamente. Análises de latência (tempo para uma dado de entrada ser computado por todos estágios da topologia) pode ser muito importante quando medindo o desempenho. A extração de mais métricas é com certeza necessária para uma investigação mais detalhada dos resultados obtidos em um trabalho futuro.

É recorrente também o teste de desempenho de MPI em outras aplicações para experimentar e descobrir se o padrão se repete. Ressaltamos que a aplicação SD não possui nenhum estágio de computação pesada, o que justifica os altos *throughputs* registrados nos experimentos. Um ponto também relevante é que o MPI pode ser usado assim como Flink e Storm em um ambiente Distribuído enquanto o Windflow não. Testar em um sistema distribuído também pode ser um trabalho futuro para esta e novas aplicações a serem implementadas com MPI.

Referências

Bordin, M. V., Griebler, D., Mencagli, G., Geyer, C. F. R., and Fernandes, L. G. (2020). DSPBench: a Suite of Benchmark Applications for Distributed Data Stream Processing Systems. *IEEE Access*, 8(na):222900–222917.

Garcia, A. M., Griebler, D., Schepke, C., and Fernandes, L. G. (2022). Evaluating Micro-batch and Data Frequency for Stream Processing Applications on Multi-cores. In *30th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, PDP'22, pages 10–17, Valladolid, Spain. IEEE.

Mencagli, G., Torquati, M., Cardaci, A., Fais, A., Rinaldi, L., and Danelutto, M. (2021). Windflow: High-speed continuous stream processing with parallel building blocks. *IEEE Transactions on Parallel and Distributed Systems*, 32(11):2748–2763.

Message Passing Interface Forum (2023). MPI: A message-passing interface standard version 4.0.