Proposta de Modelo de Programação de *Dataflow* para Aplicações de *Stream Processing* em C++

Eduardo M. Martins[1], **Dalvan Griebler**[1]

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

e.martins01@edu.pucrs.br, dalvan.griebler@pucrs.br

Resumo. *Este trabalho propõe uma abstração de programação em C++ para execução distribuída de aplicações de stream processing modeladas como grafos acíclicos (dataflow). É proposta uma API capaz de expressar topologias complexas, preservando a eficiência de runtimes baseadas em MPI. Os resultados mostram um ganho de produtividade de 27.2% e um overhead de 1.93% comparado com OpenMPI nativo. Além disso, demonstramos um resultado superior a frameworks do estado da arte baseados em Java como Apache Flink.*

1. Introdução

Nas últimas décadas, o volume global de dados cresceu de forma exponencial. As *Big Tech companies* processam bilhões de eventos diariamente, enquanto na indústria sensores e dispositivos conectados ampliam ainda mais a geração de dados. O paradigma de processamento voltado para a coleta, filtragem e análise em tempo real desses fluxos contínuos é denominado *stream processing* [Akidau et al. 2018]. Grande parte das ferramentas amplamente utilizadas nesse domínio, como Apache Flink e Apache Storm, foi desenvolvida para a *Java Virtual Machine* (JVM), priorizando portabilidade. No entanto, com o aumento do volume de dados e as cargas computacionais cada vez mais complexas, a busca por soluções de alto desempenho tornou-se atrativa.

Linguagens com compilação estática, como C++, tendem a oferecer maior desempenho, especialmente quando comparadas a linguagens que utilizam código intermediário e execução em máquinas virtuais [Gherardi et al. 2012]. Na computação de alto desempenho (HPC), a comunicação e a sincronização entre máquinas costumam ser conduzidas por meio de implementações do padrão MPI (*Message Passing Interface*). O MPI destaca-se pela performance e pelo controle sobre comunicação e sincronização, tornando-se uma alternativa atrativa para aplicações de *streming* que exigem baixa latência e alta vazão. Entretanto, o potencial de *runtimes* orientados a HPC para cargas de trabalho contínuas ainda é pouco explorado na literatura, em grande parte devido à complexidade de programação e à ausência de abstrações com ferramental comparável aos *frameworks* de *Big Data*.

Embora pesquisas anteriores já tenham investigado o processamento distribuído de *streams* em C++, esse ainda é um campo ativo de pesquisa, com diversas oportunidades para aprimorar aspectos como expressividade, desempenho e programabilidade. Em particular, trabalhos anteriores concentraram-se principalmente na abstração e implementação de aplicações limitadas a pipelines lineares [Martins et al. 2025]. Em um nível mais alto de abstração, a topologia de uma aplicação de *streaming* pode ser representada como um grafo que descreve suas dependências de dados. Nesse contexto, o termo *dataflow* refere-se a aplicações que podem ser expressas como grafos acíclicos, um modelo suficientemente expressivo para representar uma ampla variedade de aplicações. Essa representação permite diferentes formas de composição, como pipelines, divisões e fusões de fluxos de dados [Bordin et al. 2020]. Diversas aplicações paralelas operam por meio de topologias complexas, tornando essencial a existência de APIs capazes de expressar tais aplicações.

Neste trabalho, propomos uma nova abstração de programação em C++ para execução paralela e distribuída de aplicações em forma de *dataflow*. O modelo oferece uma API de alto nível capaz de expressar topologias de *stream processing* complexas, preservando ao mesmo tempo a eficiência característica de *runtimes* de HPC. Além disso, a proposta foi desenvolvida para suportar diversas características não funcionais presentes em sistemas do estado da arte, como tolerância a falhas e estratégias de controle de fluxo.

2. Modelo de Programação

O modelo desenvolvido leva em consideração abordagens da literatura. Em ambientes de memória compartilhada, muitas soluções priorizam interfaces de alto nível através de anotações para o compilador. Já alguns *frameworks* voltados a *clusters* disponibilizam APIs fluentes que permitem compor *dataflows* declarativamente por meio de operadores encadeados e construções funcionais. Em outras ferramentas, a aplicação é construída pela criação de operadores e de conexões separadamente, oferecendo maior controle sobre a topologia ao custo de maior complexidade de implementação.

No contexto de C++, a maioria das soluções existentes utiliza arquiteturas baseadas em interfaces abstratas, nas quais cada classe representa um operador. Entretanto, as soluções existentes de *streaming* em C++ se limitam a pipelines lineares, sem lidar explicitamente com as conexões entre operadores. Neste trabalho, nos introduzimos o conceito de canais nesse modelo de programação, para permitir que topologias complexas de *dataflow* possam ser expressas em uma interface de alto nível em C++.

O Código 1 apresenta um exemplo de como um *dataflow* é construído na função principal utilizando a abstração proposta. Os canais são definidos como inteiros (linhas 2 e 3), e os operadores são implementados por meio de classes que herdam de classes abstratas (como mostrado no Código 2). Após instanciar o *dataflow*, é possível adicionar *sources* (linha 12), operadores intermediários (linha 14) e *sinks* (linha 17). A definição dos canais de saída para *sources* e intermediários é obrigatória, enquanto configurações como o grau de paralelismo são opcionais. Na linha 18, o *dataflow* é executado utilizando uma estratégia de balanceamento de carga, que pode ser sob demanda, *round-robin* ou baseada em créditos. A *runtime* usa OpenMPI para comunicação e o protocolo ABS [Carbone et al. 2015] mais um agente monitor [Alf et al. 2025] para resiliência.

O Código 2 exemplifica a criação de um operador intermediário que recebe dados de dois *sources* diferentes. Com a função mostrada na linha 3, o usuário pode definir

```
1  #include "resiflow.hpp"
2  #define CANAL_1 1
3  #define CANAL_2 2
4
5  class Src :public IBaseSource {/*...*/};
6  class Op :public IBaseOperator {/*...*/};
7  class Snk :public IBaseSink {/*...*/};
8
9  int main(int argc, char **argv) {
10   Init(argc, argv);
11   Dataflow dataflow;
12   dataflow.AddSource<Src>()
13           .SetOutputs(CANAL_1);
14   dataflow.AddOperator<Op>()
15           .SetOutputs(CANAL_2)
16           .SetParallelism(4);
17   dataflow.AddSink<Snk>();
18   dataflow.Run<Demand>();
19   Finalize();
20 }
```

Código 1. *main*.

```
1  class Operator : public IBaseOperator {
2   public:
3    void setInputs() override {
4
5      onReceiveFrom(CANAL_1, [this](int x) {
6        if (x < filtro) {
7          double y = compute(x);
8          Emit(CANAL_3, y);
9        }
10     });
11
12     onReceiveFrom(CANAL_2, [this](int z) {
13       filtro = z;
14     });
15   }
16
17   private:
18    int filtro = 50;
19 };
```

Código 2. Operador intermediário.

como vai lidar com dados recebidos nos canais que deseja escutar (linhas 5 e 12). A emissão de dados em canais específicos é feita pela chamada exibida na linha 8. Essa interface desenvolvida é expressiva o suficiente para representar aplicações em forma de grafos acíclicos, estendendo as soluções C++ existentes limitadas a pipelines lineares.

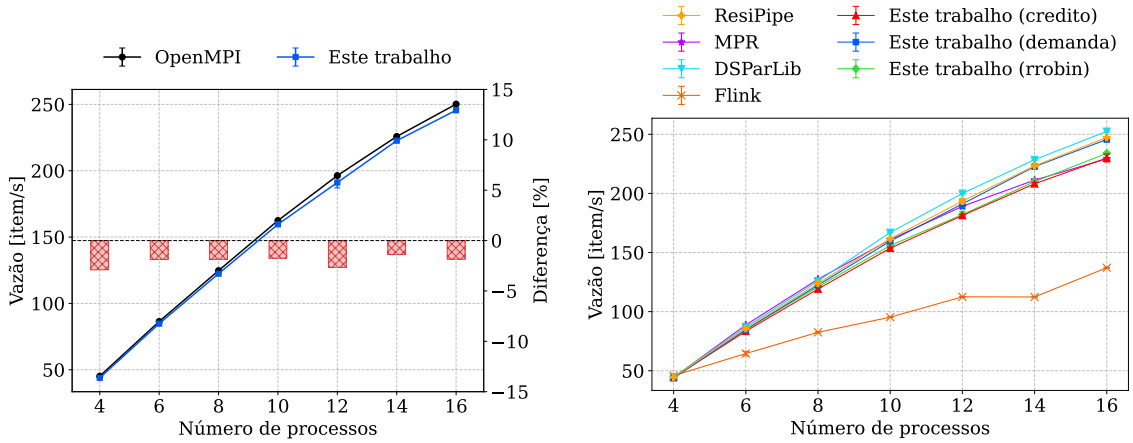
3. Experimentos

Para avaliar a *runtime* e o modelo de programação desenvolvidos, foram realizados testes de programabilidade e de desempenho. Para ambos, foi utilizada a aplicação *bzip2 compression*, pois ela está implementada e disponível em vários *frameworks* para C++ e Java. Ela consiste em um pipeline de três estágios e realiza compressão de arquivos. A performance foi medida com base na vazão em itens por segundo. O paralelismo foi explorado somente no operador intermediário, visto que as soluções baseadas em pipelines lineares usadas para comparação não suportam paralelismo no *source* e *sink*. Para a programabilidade, foram consideradas as métricas *Source Lines of Code* (SLOC) e *Halstead*.

O ambiente de testes é um *cluster* virtual com 4 máquinas, cada uma com 4 CPUs AMD Ryzen 9 9950X com frequência máxima de 5.7 GHz. Cada nó tem 8 GB de RAM. O sistema roda Linux, *kernel* 6.8.0-90-generic (Ubuntu Server 24.04.3 LTS). Os *softwares* disponíveis são: javac 11.0.29, gcc 13.3.0, OpenMPI 4.1.6 e Flink 1.19.2. Os testes foram repetidos 10 vezes. Os mecanismos de tolerância a falhas foram desligados.

A Figura 1a apresenta a comparação entre este trabalho e uma implementação manual com OpenMPI. Foi utilizado o balanceamento sob demanda da abstração, pois ela se aproxima da estratégia utilizada no OpenMPI. O eixo *x* indica o grau de paralelismo, o eixo *y1* (linhas) a vazão da aplicação e o eixo *y2* (barras) a diferença relativa entre as implementações. Em termos de escalabilidade, a abstração proposta foi capaz de acompanhar os ganhos do OpenMPI de forma proporcional ao aumento dos recursos. O *overhead* médio observado foi uma degradação de 1.93% na vazão da aplicação. Considerando apenas o grau máximo de paralelismo utilizado, a diferença é de 1.83%.

A Figura 1b apresenta a comparação entre este trabalho (considerando todas as estratégias de balanceamento) e diferentes abstrações de pipelines lineares em C++. Também foi incluído o Flink nos testes. A DSParLib apresentou o melhor desempenho,



(a) Comparação com OpenMPI (*overhead* introduzido).

(b) Comparação com outras abstrações da literatura.

Figura 1. Análises de performance usando a aplicação *bzip2 compression*.

Tabela 1. Análise de programabilidade com a aplicação *bzip2 compression*.

Framework	SLOC	*Halstead*
OpenMPI	150	12.72
Este trabalho	105	9.26

com uma vazão de 252.48 itens por segundo. ResiPipe e a abstração proposta neste trabalho, no modo sob demanda, apresentaram resultados próximos, com vazões de 247.31 e 245.6 itens por segundo, respectivamente. A MPR apresentou desempenho inferior, atingindo 229.26 itens por segundo. Por fim, o Flink não conseguiu escalar no mesmo nível das soluções baseadas em C++, obtendo uma vazão máxima de 137.13 itens por segundo. Nesse cenário, a solução proposta apresentou uma diferença de 2.7% em relação ao melhor desempenho. Vale destacar que abstrações baseadas em pipelines lineares permitem otimizações de comunicação que não são viáveis em modelos mais expressivos. Uma das principais diferenças é que, nesses pipelines, os itens são processados imediatamente ao serem recebidos, enquanto na abstração de *dataflow* é necessário primeiro identificar o canal da mensagem para determinar qual função de processamento deve ser aplicada.

O SLOC e o *Halstead* são apresentados na Tabela 1. A abstração proposta apresenta uma redução de 30% no número de linhas em comparação com o OpenMPI. O *Halstead* considera a quantidade de operandos, operadores e palavras-chave de paralelismo para mensurar a complexidade cognitiva da implementação. O resultado final é medido em horas de desenvolvimento. A implementação abstraída mostra-se mais simples de implementar, com uma redução de 27.2% no tempo de desenvolvimento.

4. Conclusão

Este trabalho apresentou a proposta de um modelo de programação em C++ para aplicações de *stream processing* representadas como grafos acíclicos. A solução integra uma *runtime* de HPC baseada em OpenMPI, suporte à tolerância a falhas e funcionalidades adicionais, como diferentes estratégias de balanceamento de carga. A abstração demonstrou um bom *trade-off* entre programabilidade e *overhead*. Além disso, apresentou desempenho competitivo em relação a outras soluções em C++ limitadas a pipelines lineares e desempenho superior ao Flink. Assim, o trabalho contribui para o avanço da pesquisa em processamento em tempo real com foco em alta performance.

Referências

Akidau, T., Chernyak, S., and Lax, R. (2018). *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing*. O'Reilly Media.

Alf, L. M. et al. (2025). Fault tolerance for high-level parallel and distributed stream processing in C++. Master's thesis, PPGCC - PUCRS, Porto Alegre, Brazil.

Bordin, M. V. et al. (2020). DSPBench: a Suite of Benchmark Applications for Distributed Data Stream Processing Systems. *IEEE Access*, 8(na):222900–222917.

Carbone, P., Fóra, G., Ewen, S., Haridi, S., and Tzoumas, K. (2015). Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*.

Gherardi, L. et al. (2012). A java vs. c++ performance evaluation: a 3d modeling benchmark. In *SIMPAR 2012, Japan, 2012. Proceedings 3*, pages 161–172. Springer.

Martins, E. M., Hoffmann, R. B., Alf, L. M., and Griebler, D. (2025). Interface para programação de pipelines lineares tolerantes a falha para mpi padrão c++. In *Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*, pages 133–144. SBC.