Avaliação de *Runtimes* de Alto Desempenho para Aplicações de *Stream Processing*

Breno Spohr Bernardi[1], **Eduardo M. Martins**[1], **Dalvan Griebler**[1]

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

(breno.bernardi, e.martins01)@edu.pucrs.br, dalvan.griebler@pucrs.br

Resumo. *Este trabalho avalia o potencial de desempenho de abstrações em C++ para stream processing, por meio da execução de aplicações reais de streaming, comparando uma implementação nativa com OpenMPI à ResiPipe, uma versão desenvolvida com uma dessas abstrações. Os resultados indicam que a versão abstraída introduz uma redução significativa na complexidade de desenvolvimento, acompanhada de um impacto negativo médio de 9.33% na vazão das aplicações em relação à implementação com OpenMPI. Esses achados quantificam o trade-off entre desempenho e nível de abstração.*

1. Introdução

A computação de alto desempenho (HPC) é historicamente vinculada à execução de aplicações científicas distribuídas, caracterizadas por cálculos numéricos intensivos e por *workloads* em *batch*. Esse modelo tradicional foi fundamental para avanços em diversas áreas, como simulações físicas, modelagem climática e análise científica em larga escala [Bailey et al. 1995]. Ao longo dos últimos anos, o cenário computacional passou a enfrentar novos desafios impulsionados pelo crescimento expressivo do volume de dados gerados de forma contínua. Nesse contexto, a dificuldade não se limita apenas ao armazenamento, mas também ao processamento imediato à medida que são produzidos [Akidau et al. 2018], caracterizando o paradigma de *stream processing*. Aplicações modernas operam com um volume alto de dados e com cargas computacionais gradativamente mais complexas, tornando soluções de HPC atrativas.

Embora comumente associados a cenários distintos, o HPC clássico e o paradigma de *stream processing* compartilham requisitos de desempenho e escalabilidade. No contexto de aplicações distribuídas de alto desempenho, a especificação Message Passing Interface (MPI) consolidou-se como o principal padrão de programação paralela. Por outro lado, no contexto de *stream processing*, *frameworks* baseados na JVM—como o Apache Flink e o Apache Storm—tornaram-se amplamente adotados, principalmente devido à sua portabilidade e ao seu ferramental. Atualmente, um dos principais desafios para o desenvolvimento de aplicações modernas de *stream processing* em ambientes de HPC reside na ausência de abstrações de alto nível que reduzam a complexidade de programação e ofereçam suporte a conceitos modernos, como tolerância a falhas [Alf and Griebler 2025].

Embora o MPI ofereça desempenho e controle explícito da comunicação, o desenvolvimento direto sobre esse modelo é custoso para o programador. Pesquisas recentes têm explorado o desenvolvimento de abstrações de alto nível para o paradigma de *stream processing*, usando linguagens com compilação estática, como C++, e runtimes de HPC, como o MPI [Martins et al. 2025]. Além disso, outros estudos recentes têm se concentrado no desenvolvimento de *benchmarks* em C++, utilizando cargas de trabalho representativas baseadas em aplicações reais de *stream processing* [Bordin et al. 2020]. Embora

muitas das soluções de *streaming* em C++ já tenham sido avaliadas com *benchmarks* sintéticos, que não refletem cenários reais de produção, ainda não está claro na literatura como essas soluções se comportam em condições realistas, utilizando aplicações clássicas de *stream processing*.

Nesse contexto, este trabalho tem como objetivo avaliar o potencial de desempenho de abstrações em C++ na execução de cargas de trabalho de *stream processing*. Para isso, utilizou-se o DSPBenchCPP [Martins et al. 2026], um *benchmark* representativo composto por aplicações clássicas de *stream processing* implementadas em C++. O *benchmark* inclui uma versão paralela baseada em OpenMPI, que representa uma abordagem nativa fundamentada em um runtime tradicional de HPC. Paralelamente, foi implementada uma versão dessas aplicações utilizando a ResiPipe [Alf and Griebler 2025], uma biblioteca C++ baseada em MPI projetada para simplificar a construção de pipelines lineares de *streaming* por meio de uma interface de alto nível e suporte embutido à tolerância a falhas. Como referência externa, considera-se também uma implementação com o Apache Flink, um *framework* amplamente adotado. Especificamente, neste trabalho é analisado o *overhead* introduzido pela implementação com a ResiPipe em relação à implementação com OpenMPI, bem como os ganhos de programabilidade da ferramenta no desenvolvimento de aplicações reais de *stream processing*.

2. Metodologia

Esta seção descreve a metodologia de experimentação utilizada. Os experimentos foram executados em um *cluster* virtual com 4 máquinas. Cada uma é equipada com CPUs Intel Xeon X6550 operando a 2.0 GHz, totalizando 16 núcleos. A arquitetura de memória inclui caches L1 separados para dados e instruções, ambos com 128 KiB, além de 1 MiB de cache L2 e 72 MiB de cache L3. Cada nó dispõe de 6 GB de memória principal. As máquinas executam o sistema operacional Linux, com *kernel* na versão 6.8.0-86-generic, sobre a distribuição Ubuntu 24.04.2 LTS. Os *softwares* utilizados foram: gcc 13.3.0, OpenMPI 4.1.6, Apache Flink 1.19.2 e Apache Kafka 2.12-3.9.1.

Para a realização dos experimentos, foram testadas quatro aplicações do DSPBenchCPP: *Fraud Detection* (`FD`), voltada à detecção de fraudes em transações com cartão de crédito; *Sentiment Analysis* (`SA`), que classifica sentimentos em textos; *Spike Detection* (`SD`), fazendo o processamento de dados de sensores; e *Traffic Monitoring* (`TM`), voltada ao monitoramento de veículos.

Em todos os experimentos realizados, foi adotada uma configuração fixa para as fontes de dados (*sources*) e os destinos finais (*sinks*) das aplicações, uma vez que a ResiPipe suporta apenas replicação de operadores intermediários. Portanto, o paralelismo foi explorado exclusivamente por meio da replicação dos operadores intermediários do *pipeline*. Como o escalonador da ResiPipe opera com um modelo sob demanda, foi selecionada uma versão das aplicações em OpenMPI que também implementa esse padrão.

A métrica de desempenho utilizada foi a vazão das aplicações, medida em mensagens por segundo. A vazão foi calculada como a razão entre o número total de mensagens emitidas pelo *source* e o tempo total de execução da aplicação. O fluxo considerado é finito, correspondendo ao consumo completo de uma base de dados armazenada no Apache Kafka. Para cada configuração, a execução foi repetida dez vezes, obtendo-se a média aritmética das execuções e o respectivo desvio padrão. Além disso, também foram avaliados aspectos relacionados à produtividade e complexidade do código. Para isso, foram consideradas as métricas *Source Lines of Code* (SLOC) e *Halstead*.

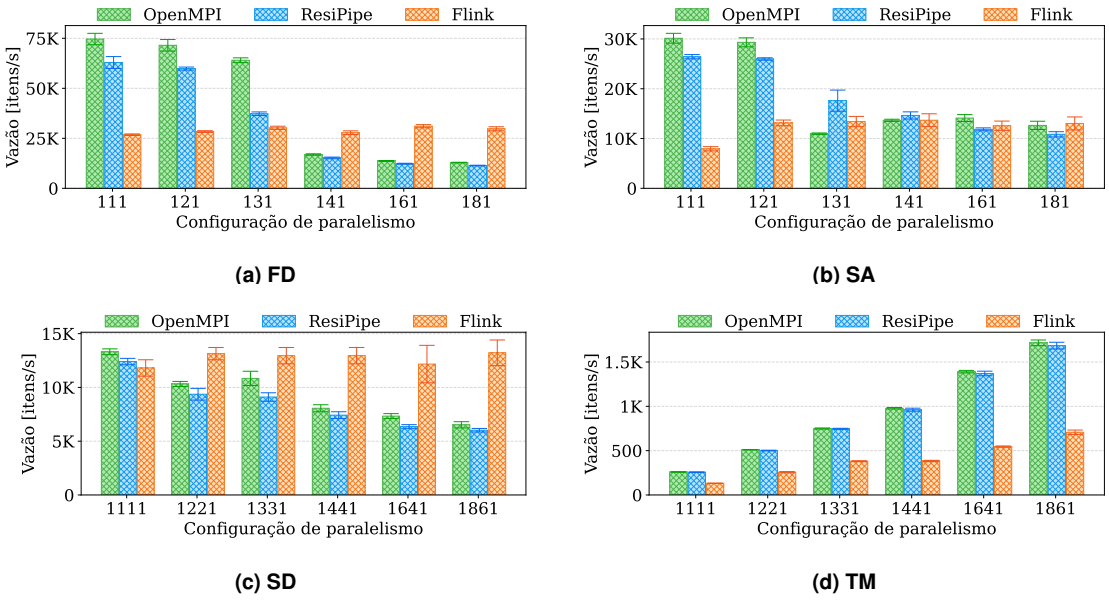

Figura 1. Resultados de performance.

Tabela 1. SLOC.

	SLOC			
Framework	**FD**	**SA**	**SD**	**TM**
OpenMPI	185	181	246	265
ResiPipe	111	105	145	164

Tabela 2. Halstead.

	Halstead			
Framework	**FD**	**SA**	**SD**	**TM**
OpenMPI	20.82	18.65	34.13	40.35
ResiPipe	7.51	5.77	10.61	15.76

3. Resultados

A Figura 1 apresenta a vazão das aplicações sob diferentes configurações de paralelismo, comparando OpenMPI, ResiPipe e Apache Flink. Cada configuração indica o número de réplicas no *pipeline* (ex.: `111` e `121`). Mais detalhes podem ser encontrados em [Bordin et al. 2020].

É importante mencionar que muitas aplicações reais de *stream processing* operam com alta taxa de eventos e baixo custo computacional por item. Portanto, o ganho computacional nem sempre compensa o custo de comunicação ao aumentar o paralelismo. Esse comportamento é observado nas aplicações `FD`, `SA` e `SD`, onde as implementações baseadas em OpenMPI obtiveram melhores resultados com menor grau de paralelismo, especialmente na configuração `111`. Em comparação com o OpenMPI, a ResiPipe apresenta uma degradação na vazão de 15.7%, 12.2% e 6.84% respectivamente em seus melhores casos. Por apresentarem baixo custo computacional por evento, essas aplicações são mais sensíveis a *overheads* adicionais introduzidos pela ResiPipe. Essa diferença de desempenho pode estar associada às camadas extras de abstração da biblioteca, como o uso de funções virtuais e um maior número de operações de serialização e desserialização. Ressalta-se, contudo, que essas explicações baseiam-se no comportamento observado e na estrutura da biblioteca. Por outro lado, esse comportamento não é observado nos resultados com o Apache Flink. Nas aplicações com baixo custo computacional por evento, o framework mantém a vazão mais estável ao variar o grau de paralelismo, possivelmente devido ao seu mecanismo de alocação de operadores, que busca otimizar a comunicação.

A aplicação `TM` destaca-se por ser a mais intensiva computacionalmente. Dessa forma, diferentemente das demais, a aplicação consegue se beneficiar do paralelismo. As três versões alcançaram a melhor vazão na configuração de maior paralelismo, com uma diferença pequena de 1.86% em favor do OpenMPI em comparação com a ResiPipe. Nesse caso, o peso da computação é suficiente para amortizar também o *overhead* da abstração, fazendo com que as duas versões apresentem desempenho praticamente equivalente. Por outro lado, o *framework* Apache Flink não acompanha os níveis de vazão das outras duas implementações, uma vez que o maior custo computacional dos eventos favorece abordagens baseadas em OpenMPI, que se beneficiam de cenários com maior intensidade computacional.

No quesito complexidade de código, a Tabela 1 apresenta o SLOC de cada aplicação. A ResiPipe requer menos linhas de código em todos os casos, com uma variação relativa entre 38.1% e 41.9%. A Tabela 2 mostra os resultados da métrica de Halstead, que estima a complexidade cognitiva da implementação considerando a quantidade de operandos, operadores e palavras-chave de paralelismo. As implementações com a ResiPipe apresentam reduções entre 60.9% e 69% em comparação ao OpenMPI.

4. Conclusão

Este trabalho comparou o desempenho de implementações de aplicações tradicionais de *stream processing* utilizando OpenMPI de forma nativa, a abstração de programação ResiPipe e o Apache Flink como referência externa. Em média, a ResiPipe apresentou uma degradação de 9.33% na vazão em relação ao OpenMPI. O *overhead* demonstrou-se maior em cenários de baixa carga computacional por operador, mas é significativamente reduzido em aplicações mais intensivas. Em contrapartida, os ganhos de produtividade foram relevantes, com redução média de 40.1% no SLOC e 65.2% no *Halstead*. Esses resultados evidenciam o *trade-off* entre desempenho e programabilidade das ferramentas, indicando que a ResiPipe é uma alternativa viável quando produtividade e resiliência são requisitos relevantes.

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. and Griebler, D. (2025). Fault tolerance for high-level parallel and distributed stream processing in C++. Master's thesis, PUCRS, Porto Alegre, Brazil.

Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., and Yarrow, M. (1995). The nas parallel benchmarks 2.0. Technical report, Citeseer.

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

Martins, E. M., Bernardi, B. S., Fim, G. R., Hoffmann, R. B., Mencagli, G., and Griebler, D. (2026). HPC Meets Streaming: Benchmarking OpenMPI, Apache Flink, and Apache Storm. In *34th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, PDP'26, Cluj-Napoca, Romania. IEEE.

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.