Interface para Programação de Pipelines Lineares Tolerantes a Falha para MPI Padrão C++

Eduardo M. Martins[1] **, Renato B. Hoffmann**[1] **, Lucas M. Alf**[1] **, Dalvan Griebler**[1]

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

{e.martins01, renato.hoffmann, lucas.alf}@edu.pucrs.br
dalvan.griebler@pucrs.br

Resumo. *Sistemas de processamento de stream são projetados para operar continuamente e devem ser capazes de se recuperar em caso de falhas. No entanto, programar aplicações de alto desempenho em ambientes distribuídos introduz uma alta complexidade de desenvolvimento. Este trabalho apresenta uma interface de programação que facilita a construção de pipelines lineares tolerantes a falhas para aplicações de processamento de stream em C++. A solução utiliza MPI (Message Passing Interface) para comunicação e o protocolo ABS (Asynchronous Barrier Snapshotting) juntamente com um agente monitor para a etapa de recuperação. Os resultados experimentais indicam uma redução significativa no tempo estimado de desenvolvimento para o programador, com impacto médio de -0.98% até 6.73% na vazão das aplicações. Além disso, o processo de recuperação mitiga o impacto das falhas na vazão do programa.*

1. Introdução

Os sistemas de processamento de *stream* necessitam de paralelismo para atender às demandas de baixa latência e alta vazão. Embora *multithread* possa ser uma alternativa, a escalabilidade é limitada pelos processadores de uma única máquina [Löff et al. 2022, Griebler 2016]. Devido ao aumento no volume de dados, esse paradigma de processamento evoluiu para sistemas distribuídos. Em ambientes de alto desempenho, a coordenação entre os diferentes nós computacionais é predominantemente feita usando alguma implementação do padrão MPI (*Message Passing Interface*) [Yin and Shi 2022].

Apesar de ser uma solução flexível e de alto desempenho, desenvolver aplicações com MPI impõe uma complexidade significativa, exigindo que programadores lidem diretamente com detalhes como serialização de dados, comunicação, sincronia entre processos e construção manual das estruturas de algoritmos paralelos [Nielsen 2016]. Para mitigar esse problema, uma alternativa comum na literatura é utilizar ou criar abstrações de alto nível que usam MPI, mas encapsulam os padrões paralelos em interfaces simples para o usuário final [Griebler 2016]. Assim, os desenvolvedores conseguem focar nos aspectos computacionais das aplicações em si, sem lidar com detalhes de baixo nível.

Além disso, as primeiras implementações do MPI—desenvolvidas na década de 1990—não priorizavam mecanismos de tolerância a falhas. Naquele contexto, os *clusters* possuíam poucos núcleos, e o tempo médio entre falhas costumava ser superior ao tempo de execução da maioria das aplicações. No entanto, esse cenário mudou significativamente: os *clusters* passaram a ter milhares de núcleos, e novos paradigmas de

processamento tornaram-se comuns—como o *streaming*, onde as aplicações operam de forma contínua e ininterrupta [Carbone et al. 2017].

Com o surgimento de aplicações que levam mais tempo para executar do que o tempo médio entre as falhas, a resiliência se tornou uma característica essencial em sistemas distribuídos. Os principais modelos de recuperação existentes são baseados em *logs* e *Checkpoint-Restart* (C/R) [Joshi and Vadhiyar 2025]. Enquanto *logs* registram eventos e alterações de estado para permitir a reprodução incremental da execução após uma falha, o modelo C/R realiza capturas periódicas do estado global da aplicação, possibilitando que a execução seja retomada a partir do último ponto consistente salvo [Egwutuoha et al. 2013]. As primeiras abordagens de tolerância a falhas no contexto do MPI baseavam-se em técnicas *ad hoc* de C/R, desenvolvidas de forma específica para cada aplicação. Revisões do padrão MPI adicionaram suporte mais estruturado à tolerância a falhas, entretanto, o problema é que muitas dessas implementações não estão amplamente disponíveis e portanto não são portáveis [Bland et al. 2013].

Entendendo essa realidade, este trabalho propõe uma interface de programação para facilitar o desenvolvimento de aplicações de processamento de *stream*, com suporte à tolerância a falhas. A solução consiste em uma biblioteca *header-only* para C++ baseada em MPI, que abstrai computações na forma de pipelines lineares. Para a recuperação de falhas, a biblioteca implementa o protocolo ABS (*Asynchronous Barrier Snapshotting*)—uma abordagem consolidada e utilizada por soluções do estado da arte, escritas em Java—e emprega de forma portável um agente monitor responsável pela coordenação e recuperação dos processos. As contribuições desta pesquisa consistem (1) uma interface de abstração focada em balancear programabilidade e desempenho, (2) uma metodologia para coordenar o processo de recuperação de falhas em ambientes MPI e (3) uma análise experimental da solução proposta.

Esse texto segue a seguinte estrutura: A Seção 2 contempla o referencial teórico e a Seção 3 aponta os trabalhos relacionados. A Seção 4 apresenta a interface proposta. A Seção 5 demonstra os experimentos realizados e discute os resultados. Por fim, na Seção 6 são expostas as conclusões obtidas e as direções futuras da pesquisa.

2. Referencial Teórico

Esta Seção tem dois objetivos: introduzir formalmente o conceito de pipelines lineares (2.1); e descrever o funcionamento do protocolo ABS, utilizado para *checkpointing* de estados globais consistentes em aplicações de processamento de *streaming* (2.2).

2.1. Aplicações de Streaming e Pipelines Lineares

Aplicações de *streaming* são comumente representadas por grafos direcionados [Carbone et al. 2017]. O fluxo de mensagens percorre o grafo desde um operador *source*, responsável por interagir com meios externos à aplicação para coletar os dados de entrada, até um operador *sink*, que geralmente envia as informações processadas para sistemas externos. Ao longo do percurso, os dados passam por uma sequência de operadores, onde cada um aplica uma etapa de processamento sobre os itens individualmente, como filtragem, transformação, agregação ou redução. As arestas direcionadas da topologia indicam os caminhos possíveis que o fluxo de dados pode seguir dentro do sistema de *streaming*.

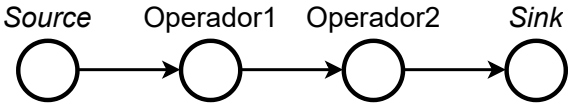


Figura 1. Aplicação de *streaming* abstraída como um pipeline linear.

A Figura 1 ilustra uma aplicação de *streaming* abstraída como um pipeline linear. Esse padrão é uma especialização derivada de um grafo direcionado que descreve uma aplicação de *streaming* através de uma cadeia sequencial de estágios de processamento. Formalmente, o pipeline linear é representado por um par $G = (V, E)$, em que V é um conjunto finito de vértices (operadores), e $E \subseteq V \times V$ é um conjunto de arestas direcionadas (dependências de dados). Cada aresta $(V_a, V_b) \in E$ representa uma conexão dirigida do vértice V_a (origem) para o vértice V_b (destino), sendo que $(V_a, V_b) \neq (V_b, V_a)$. Além disso, uma aresta é válida apenas se seguir o padrão (V_n, V_{n+1}). No contexto deste trabalho, somente operadores que não têm o papel de *source* ou *sink* podem ser replicados na topologia. No exemplo, os operadores 1 e 2 podem ser replicados para aumentar o paralelismo.

2.2. Protocolo ABS

O protocolo ABS consiste em um mecanismo de *snapshotting* projetado para sistemas distribuídos de processamento de *streaming*. Ele foi projetado com o objetivo de reduzir o impacto no desempenho e o custo de armazenamento. A ideia central do algoritmo é particionar o fluxo de dados em estágios consistentes por meio da injeção periódica de marcadores especiais de *snapshot* no fluxo, sem interromper a execução do sistema.

Durante a execução, um coordenador central injeta periodicamente marcadores de *snapshot* em todos os operadores *source*. Ao receber um marcador, o operador *source* realiza um *snapshot* do seu estado atual e propaga o marcador para todos os seus canais de saída. Quando um operador recebe um marcador em um de seus canais de entrada, ele bloqueia esse canal até receber o mesmo marcador em todos os canais de entrada restantes. Em seguida, o operador realiza o *snapshot* do seu estado atual, propaga o marcador para todos os seus canais de saída e então desbloqueia os canais de entrada. O *snapshot* global é considerado completo quando todos os operadores tiverem realizado um *snapshot* correspondente ao marcador injetado [Carbone et al. 2015].

O protocolo ABS garante a consistência dos estados capturados porque os marcadores de *snapshot* impõem um ponto de corte lógico no fluxo de dados, delimitando eventos que devem ser incluídos ou excluídos de cada *snapshot*. O algoritmo assume que os canais de rede são confiáveis, capazes de lidar com falhas de processo ou perda de mensagens, respeitam a ordem de entrega FIFO (*first-in*, *first-out*) e podem ser bloqueados e desbloqueados.

3. Trabalhos Relacionados

Diversos estudos já exploraram aspectos de tolerância a falhas no MPI e o processamento de *stream* em C++. Entretanto, essas continuam sendo áreas em desenvolvimento, com oportunidades para investigar questões relacionadas à programabilidade e ao desempenho. Esta Seção apresenta os trabalhos relacionados ao tema e discute como a solução proposta se diferencia das abordagens existentes.

Algumas das primeiras iniciativas que buscaram superar as limitações de resiliência do padrão MPI focaram em implementações com suporte a falhas nativo, porém elas tiveram pouca continuidade e são pouco utilizadas [Bouteiller et al. 2006]. Outras soluções foram desenvolvidas de maneira específica, limitando-se a uma única aplicação. Atualmente, grande parte dos estudos se concentra no *User-Level Failure Mitigation* (ULFM) [Bland et al. 2013], que permite a reconfiguração dinâmica de comunicadores após falhas. Esta extensão já está integrada em algumas implementações do MPI, como o OpenMPI a partir da versão 4.0.

Entre as estratégias consolidadas para o processo de recuperação, destaca-se o modelo de *Checkpoint-Restart* (C/R), amplamente utilizado em aplicações MPI [Egwutuoha et al. 2013]. Abordagens mais recentes combinam esse método com mecanismos como o ULFM. A biblioteca FTHP-MPI [Joshi and Vadhiyar 2025] integra C/R e replicação de processos com funcionalidades como o *communicator shrinking*, para remover processos falhos e permitir a continuidade da execução. Outras alternativas, como o CRAFT [Shahzad et al. 2018] seguem metodologias similares. Legio [Rocco et al. 2024] é uma proposta recente que atende a aplicações *embarrassingly parallel*, focando na degradação gradual de processos, desde que não sejam críticos. Apesar dos avanços, não há um padrão consolidado entre essas soluções, e muitas implementações ainda não estão amplamente disponíveis e, portanto, não são portáveis.

Em relação aos sistemas de processamento de *stream* distribuído, destacam-se as soluções da Apache. O Apache Flink [Carbone et al. 2017] é um projeto de código aberto que oferece suporte ao processamento contínuo de dados, com uma API declarativa e fluente para a construção de grafos acíclicos direcionados (DAGs) de transformações. O Flink implementa tolerância a falhas com o modelo C/R, por meio do protocolo ABS. Uma alternativa similar é o Apache Storm [Pathirana et al. 2015], que permite a definição de computações em tempo real na forma de DAGs compostos por `Spouts` (fontes de dados) e `Bolts` (unidades de processamento). A tolerância a falhas do Storm baseia-se em um *source* dedicado que emite mensagens de *checkpoint* em intervalos regulares. Outros sistemas populares incluem Apache Spark Streaming [Apache 2025], Google Dataflow [Akidau et al. 2015] e Hazelcast Jet [Gencer et al. 2021].

Grande parte das soluções para processamento de *stream* distribuído—incluindo Flink, Storm, Spark Streaming, Dataflow e Hazelcast Jet—são implementadas com linguagens de programação como Java, devido à sua portabilidade. Embora ambientes de execução como o baseado na Máquina Virtual Java (JVM) tenham sido amplamente estudados e otimizados [Manchana 2015], eles ainda apresentam desempenho inferior quando comparados a compiladores com otimização estática, especialmente em relação às linguagens consolidadas como C/C++ [Gherardi et al. 2012]. Por outro lado, desenvolver aplicações de *streaming* distribuídas nesses ambientes de baixo nível envolve uma complexidade maior. Existem soluções na literatura que propõem abstrações de alto nível para o processamento de *stream* em C/C++. No entanto, as abordagens geralmente se restringem a arquiteturas de memória compartilhada ou não oferecem tolerância a falhas.

A SPar [Griebler et al. 2017] é uma linguagem de domínio específico (DLS) que busca equilibrar programabilidade e desempenho. Ela introduz atributos do C++11 que permitem definir os estágios de um pipeline linear diretamente no código sequencial, com mínimas alterações. Inicialmente voltada para ambientes de memória compartilhada, a

SPar integrou a DSParLib [Löff et al. 2022] para gerar código de arquiteturas distribuídas—embora ainda sem suporte à tolerância a falhas. Spidle [Consel et al. 2003] é outra DSL voltada para ambientes de memória compartilhada em C. Ela permite a definição de uma rede de tarefas conectadas, na qual se especifica como os dados devem fluir entre os componentes, bem como os tipos de dados envolvidos. Soluções relacionadas também incluem o OmpSs-2 [Grant and Voorhies 2025]—para memória compartilhada—e o OpenStream [Pop and Cohen 2013]—para arquiteturas distribuídas, mas sem suporte à tolerância a falhas. Ambas se baseiam em anotações `pragma` da linguagem C, de forma similar ao OpenMP.

4. Abstração proposta

Esta Seção apresenta a proposta de abstração de pipelines lineares desenvolvida para MPI padrão C++, originalmente introduzida em uma dissertação de mestrado [Alf and Griebler 2025]. Primeiro são apresentados detalhes de design e implementação (4.1) e depois a interface de programação (4.2).

4.1. Design e Implementação

A solução proposta possui dois componentes: uma biblioteca *header-only* e um agente monitor portável. A biblioteca utiliza OpenMPI para a troca de mensagens, sincronização e manejamento dos processos, além de implementar *checkpointing* de acordo com o protocolo ABS. Além disso, ela fornece a API para o usuário final, descrita em mais detalhes na Seção 4.2. O outro componente é o `monitor`, que é responsável por iniciar os processos MPI, monitorar a disponibilidade das máquinas e redistribuir os processos em caso de falhas. Ele pode ser executado em uma máquina dedicada ou compartilhada com os processos da aplicação.

A Figura 2 ilustra a arquitetura do sistema de recuperação. Durante a execução, o monitor envia periodicamente (5 segundos) *heartbeats* para verificar a atividade das máquinas. Caso alguma deixe de responder, seus processos e respectivas cargas de trabalho são automaticamente redistribuídos entre os nós restantes. Se uma máquina inativa voltar a ficar disponível, o monitor é capaz de reintegrá-la ao sistema e redistribuir os processos conforme necessário. Por outro lado, falhas fatais de processos MPI são detectadas e tratadas imediatamente. Todas as ações de mudança da configuração do sistema reiniciam a partir do último *checkpoint* global válido, como determinado pelo protocolo ABS. Para garantir a persistência dos *snapshots*, é necessário um sistema de arquivos distribuído

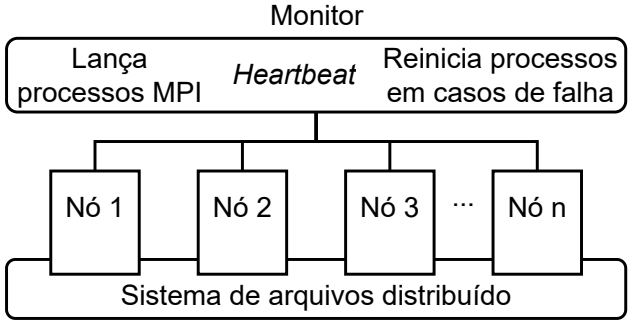


Figura 2. Arquitetura do agente monitor.

Tabela 1. Funções disponíveis ao usuário.

namespace	Função	Retorno	Descrição
	init(argc, argv)	void	Inicializa o ambiente.
	finalize()	void	Finaliza o ambiente.
serialization	*serialize(value)*	*std::string*	Serializa o um valor para binário.
serialization	*deserialize<type>(value)*	*template*	Desserializa uma *string* para um determinado tipo.
keyStore	*store(key, value)*	void	Armazena um estado a partir de uma chave.
keyStore	*retrieve<type>(key)*	*template*	Recupera um estado a partir de uma chave.
keyStore	*contains(key)*	bool	Verifica se uma chave está sendo usada.
pipeline	*run(...)*	void	Inicia a execução do pipeline.
utility	*getRank()*	int	Retorna o identificador do processo MPI.
utility	*getCommunicatorSize()*	int	Retorna o tamanho do comunicador MPI.
utility	*getProcessorName()*	*std::string*	Retorna o nome da máquina.
utility	*getExecutionPlan()*	struct	Retorna uma estrutura com informações do pipeline.
utility	*getNode()*	struct	Retorna uma estrutura com informações do nó.

persistente e tolerante a falhas. A implementação atual baseia-se na arquitetura HDFS (Hadoop Distributed File System).

A Tabela 1 apresenta as funções destinadas ao usuário final. A biblioteca expõe uma série de funções divididas em quatro *namespaces*: `pipeline`, `utility`, `keyStore` e `serialization`. O *namespace* `pipeline` inclui funções para a criação de aplicações em forma de pipelines lineares, como a definição do *source*, dos estágios intermediários e do *sink*. O `utility` reúne funções auxiliares, como para obter o nome da máquina, o identificador do processo MPI e o número de processos no pipeline. Já o *namespace* `serialization` oferece funções para serialização e desserialização de dados em formato binário. Internamente é utilizada a biblioteca cereal[1], que permite a serialização eficiente dos tipos definidos no padrão C++. Por fim, o `keyStore` é utilizado para gerenciamento de estados, através de funções de armazenamento e recuperação de dados em uma estrutura de chaves tolerante a falhas mantida em memória.

Em relação ao balanceamento de carga, a abstração adota um modelo de comunicação sob demanda, no qual os operadores enviam suas saídas para o primeiro canal que requisitar trabalho. Essa estratégia melhora a distribuição de carga entre os operadores, priorizando aqueles que estão desocupados, ainda que implique no envio adicional de mensagens de requisição. Além disso, cada estágio pode adotar diferentes modos de entrega, conforme a necessidade da aplicação. No modo *default*, a mensagem segue o modelo sob demanda. No modo *broadcast*, uma cópia da mensagem é enviada a todas as réplicas do operador seguinte. Já o modo *affinity* direciona mensagens com propriedades em comum—definidas pelo usuário—para o mesmo canal. A propriedade pode ser configurada passando um parâmetro opcional na chamada `Emit`.

4.2. Modelo de Programação

Um dos principais objetivos da interface é abstrair o desenvolvimento de aplicações em formato de pipeline, ou linha de produção. Para isso, as aplicações são estruturadas por meio da implementação de três interfaces de classe em C++: *ISourceOperator*, *IMiddleOperator* e *ISinkOperator*. Essas interfaces representam, respectivamente, o *source*, os operadores intermediários e *sink*.

[1]GitHub cereal: https://github.com/USCiLab/cereal

```
1  int main(int argc, char **argv)
2  {
3    init(argc, argv);
4    Source src;
5    Middle mdl;
6    Sink snk;
7    mdl.setReplicate(2);
8    snk.setOrdered(true);
9    pipeline::run(&src, &mdl, &snk);
10   finalize();
11 }
```

Código 1. *main*.

```
1  class Source : public ISourceOperator
2  {
3   public:
4    void onProduce() {
5      while(true) {
6        Event e = consumeEvent();
7        Emit(e.id,
8          serialization::serialize(e));
9      }
10   }
11 };
```

Código 2. *source*.

```
1  class Middle : public IMiddleOperator
2  {
3   public:
4    void onReceive(const Message &m) {
5      Event e = serialization::
6          deserialize<Event> (m.value);
7      int out = compute(e);
8      Emit(serialization::serialize(out));
9    }
10 };
```

Código 3. Operador intermediário.

```
1  class Sink : public ISinkOperator
2  {
3   public:
4    void onReceive(const Message &m) {
5      int val = serialization::
6                deserialize<int>
7                (m.value);
8      print_output(m.id, val);
9    }
10 };
```

Código 4. *sink*.

O Código 2 apresenta um exemplo de operador *source*, responsável por ingerir dados de sistemas externos. A abstração suporta implementações personalizadas ou qualquer sistema que disponibilize uma API em C++, como *librdkafka*[2] e *rabbitmq-c*[3]. A classe do *source* deve implementar a interface `ISourceOperator`, que exige a definição do método `onProduce`. Esse método especifica como os dados serão ingeridos, e dentro dele é possível fazer a chamada `Emit`, que envia os novos registros ao pipeline. Os operadores trocam mensagens por meio da estrutura `Message`, que encapsula metadados e o *payload* serializado. Para lidar com as mensagens recebidas, operadores intermediários e o *sink* devem definir o método `onReceive`, responsável por tratar a chegada de novos registros. Conforme o Código 3 e Código 4, a principal diferença entre eles é que a interface *ISinkOperator* não implementa a chamada `Emit`.

O Código 1 apresenta um exemplo de uso da API dentro da função *main*. No início da aplicação, é necessário chamar a função `init`, que inicia o ambiente MPI subjacente e os processos de comunicação e *snapshot*. O usuário então pode declarar os operadores do pipeline e definir configurações, como o número de réplicas de cada operador e se o operador de destino deve preservar a ordenação das mensagens ou não. A partir disso, a execução do pipeline é iniciada com a chamada à função `run` do *namespace* `pipeline`. Caso a aplicação exija múltiplos operadores intermediários, é possível fornecer um vetor de operadores à função. Ao final da execução, é necessário fazer a chamada `finalize` para encerrar corretamente os processos e finalizar o ambiente MPI.

Por fim, as principais **limitações** da implementação atual incluem: falta de suporte à pipelines não lineares, encontrados em estruturas complexas de *dataflow*; impossibilidade de replicação dos operadores *source* e *sink*; ausência de suporte à escalabilidade dinâmica dos operadores; ponto de falha único no agente monitor.

[2]GitHub Kafka: https://github.com/confluentinc/librdkafka
[3]GitHub RabbitMQ: https://github.com/alanxz/rabbitmq-c

5. Experimentos

Esta Seção apresenta os experimentos realizados para avaliar a abstração proposta. A avaliação foi estruturada em torno de três questões de pesquisa (Q) principais: (Q1, 5.1) Qual é o impacto da interface na programabilidade? (Q2, 5.2) Qual é o impacto da interface no desempenho? (Q3, 5.3) Qual é o desempenho do mecanismo de *snapshot* e recuperação? Por praticidade, nesta Seção o OpenMPI também é referido como `OMPI`. Em virtude da ausência da implementação dessas aplicações em Java e Apache Flink, não foi possível realizar esta análise comparativa. Dado o trabalho significativo de implementação, este será abordado em uma investigação futura.

Os experimentos para avaliar o desempenho foram realizados em um *cluster*[4] composto por 14 nós com sistema operacional Ubuntu $20.04.6$ LTS, kernel $5.4.0-156-generic$, com OpenMPI $1.10.7$ e GCC 9.4. Cada nó é equipado com dois processadores Intel Xeon E5-2620 com frequência de *clock* de 2 GHz, em um total de 12 núcleos e 24 *threads* por nó. Os nós possuem 24 GB de RAM, duas interfaces de rede Gigabit Ethernet e uma interface de rede Infiniband QDR 4x (32 Gbit/s).

Os testes apresentados na Seção 5.1 e na Seção 5.2 baseiam-se na implementação de quatro aplicações distintas de processamento de *stream*. Embora todas adotem topologias compostas por três estágios (*source*, trabalhador e *sink*), elas foram selecionadas por apresentarem diferentes características de carga de trabalho. As aplicações *bzip2 compression* e *bzip2 decompression* realizam, respectivamente, compressão e descompressão de arquivos em disco utilizando o algoritmo `BZip2`. A aplicação *prime numbers* é um exemplo sintético que calcula os números primos dentro de um intervalo fixo. Por fim, a aplicação *eye detector* realiza a detecção de rostos e olhos em vídeos, utilizando classificadores *Haar-Cascade* da biblioteca OpenCV[5].

5.1. Q1: Qual é o impacto da interface na programabilidade?

Para avaliar o impacto da abstração proposta na programabilidade, foram utilizadas duas métricas relevantes na análise da complexidade de software: o tempo estimado de desenvolvimento, baseado nas métricas de Halstead [Andrade et al. 2022], e o SLOC (*Source Lines of Code*). As métricas de Halstead levam em conta a quantidade e a variedade de operadores e operandos presentes no código, enquanto o SLOC corresponde à contagem de linhas de código-fonte, desconsiderando comentários e linhas em branco.

A Tabela 2 apresenta os resultados obtidos com a métrica SLOC para as quatro aplicações implementadas com a abstração proposta e com OpenMPI puro. Em termos absolutos, a interface desenvolvida reduziu o número de linhas de código necessárias em aproximadamente 37,5% em comparação ao uso direto de OpenMPI. Da mesma forma, a Tabela 3 exibe o tempo estimado de desenvolvimento de cada implementação, calculado com base nas métricas de Halstead. Nesse caso, a interface proposta proporcionou uma redução de aproximadamente 69,73% em termos absolutos.

Os resultados obtidos demonstram que a solução proposta contribui para a melhoria da programabilidade. Enquanto o MPI se restringe a abstrair a camada de comunicação, o modelo apresentado neste trabalho estende essa abstração ao englobar a estrutura

[4] *Cluster* Cerrado do labLAD da PUCRS: https://www.pucrs.br/ideia/laboratorios-ideia/lablad

[5] GitHub OpenCV: https://github.com/opencv/opencv

do padrão pipeline e a serialização dos dados, o que justifica os ganhos observados. Além disso, a diferença percentual mais expressiva no Halstead ocorre porque, além das chamadas diretas do OpenMPI serem mais complexas e verbosas, a maior quantidade de linhas de código torna o programa suscetível a conter mais operadores e operandos. Portanto, a interface proposta tem um impacto positivo na etapa de desenvolvimento das aplicações.

Tabela 2. SLOC.

Framework	SLOC			
	prime numb.	*eye detec.*	*bzip2 decomp.*	*bzip2 comp.*
Este trabalho	60	106	133	106
OMPI	119	154	204	171

Tabela 3. Halstead.

Framework	Halstead			
	prime numb.	*eye detec.*	*bzip2 decomp.*	*bzip2 comp.*
Este trabalho	1.53	3.46	6.29	3.41
OMPI	8.33	11.61	18.44	13.46

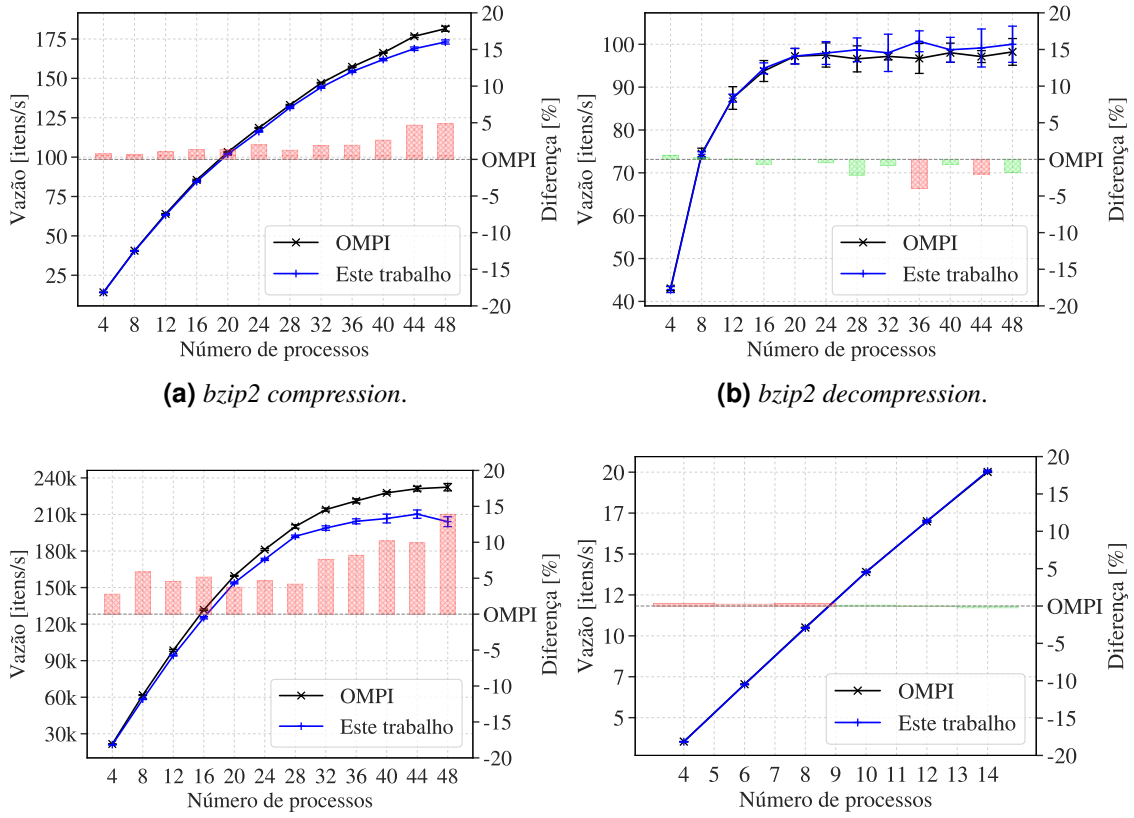
5.2. Q2: Qual é o impacto da interface no desempenho?

Para avaliar o impacto da abstração no desempenho, analisou-se a vazão das aplicações em mensagens por segundo.Os testes foram repetidos dez vezes, e os resultados apresentados correspondem à média aritmética e ao desvio padrão obtidos. Como o foco é isolar o impacto da abstração, o mecanismo de tolerância a falhas foi desabilitado durante esse experimento. Além disso, as implementações com OpenMPI seguiram a mesma abordagem de comunicação sob demanda adotada pela biblioteca desenvolvida.

Os gráficos da Figura 3 apresentam, para cada aplicação, a vazão no eixo $y1$ e o número de processos no eixo x (onde somente o operador intermediário é replicado). Com as barras, no eixo $y2$ é possível ver a diferença relativa entre este trabalho e a implementação com OpenMPI. Para avaliar se as diferenças observadas entre as amostras de cada implementação são estatisticamente significativas, foi aplicado o teste de hipótese de *Mann–Whitney*. A cor vermelha nas barras indica que as amostras apresentam diferenças estatisticamente significativas ($\texttt{p-value} < 0{,}05$), sugerindo que pertencem a populações distintas. Já a cor verde indica ausência de diferença significativa, o que é compatível com a hipótese de que as amostras provêm da mesma população.

A Figura 3a apresenta os resultados para a aplicação *bzip2 compression*. A solução proposta exibiu desempenho semelhante ao OpenMPI puro, alcançando uma vazão máxima de 173,09 itens por segundo, em comparação com 181,52 obtidos pelo OpenMPI. Embora a diferença relativa tenha sido constantemente baixa—em média 2,03%—ela mostrou uma tendência a aumentar à medida que a aplicação escala. Na aplicação *bzip2 decompression* (Figura 3b), ambas as versões exibem padrões de comportamento similares, com escalonamento que se estabiliza por volta de 20 a 24 processos. A interface proposta apresentou uma vazão levemente superior à do OpenMPI—em média -0,98%. Entretanto, os testes estatísticos indicam que não há evidências claras de diferença significativa entre as soluções.

No caso da aplicação *prime numbers* (Figura 3c), por se tratar de um modelo sintético com computação simples, o principal gargalo passa a ser o custo de comunicação. Nesse tipo de cenário, a abstração proposta tende a apresentar desempenho inferior ao OpenMPI puro. Essa aplicação exibiu a maior diferença relativa média observada nos testes, atingindo 6,73%. Em contra partida, a aplicação *eye detector* (Figura 3d) impõe maior carga à CPU, uma vez que a biblioteca OpenCV é utilizada para criar múltiplas



(a) *bzip2 compression.* **(b)** *bzip2 decompression.*

(c) *Prime numbers.* **(d)** *Eye detector.*

Figura 3. Comparação de desempenho entre aplicações desenvolvidas com a interface proposta neste trabalho e OpenMPI manualmente desenvolvido.

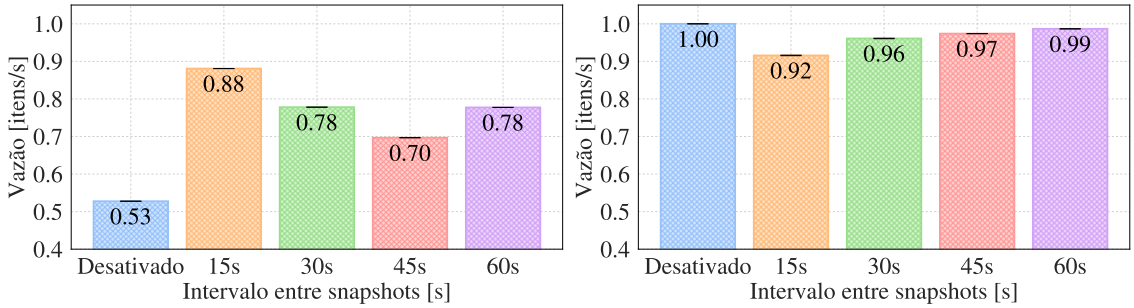
threads no processamento das imagens. Por esse motivo, o experimento foi configurado para distribuir apenas um processo por nó, que explora localmente o máximo de paralelismo disponibilizado pelas múltiplas CPUs. Neste cenário, os resultados apontam uma diferença negligenciável de 0,12% na vazão entre as diferentes implementações.

5.3. Q3: Qual é o desempenho do mecanismo de *snapshot* e recuperação?

Para mensurar o desempenho da recuperação de falhas e validar seu funcionamento por meio de uma prova de conceito, foram realizados dois experimentos. O primeiro considera um pipeline com quatro estágios e sem replicação (descrito na Figura 1), no qual uma falha é induzida no `Operador2` após 90 mensagens de um total de 100. O objetivo é avaliar a vazão da aplicação após a retomada, variando-se a frequência dos *snapshots*, em que o estado contém 8 bytes. O segundo experimento considera o mesmo pipeline, mas com um estado de 100MB e sem nenhuma falha. Nesse caso, busca-se analisar o impacto do protocolo de tolerância a falhas da interface durante a execução, na presença de um operador com estado não negligenciável.

Os gráficos da Figura 4 apresentam os resultados para os dois cenários descritos. Cada teste foi repetido dez vezes, e os resultados apresentados correspondem à média aritmética e ao desvio padrão, normalizados pela vazão da execução sem falhas e sem *snapshot*. Para o primeiro cenário (Figura 4a), independentemente da frequência dos

snapshots, a ativação do mecanismo de recuperação de falhas contribui para uma melhora da vazão, uma vez que a aplicação não precisa reprocessar os dados anteriores ao último ponto de salvamento válido. No caso do segundo cenário (Figura 4b), o gráfico indica que para um pipeline contendo um operador com 100MB de estado persistente, o mecanismo de tolerância a falhas introduz uma sobrecarga de aproximadamente 8,17% quando se utiliza um intervalo de *snapshots* de 15 segundos. Ao utilizar intervalos de 30, 45 e 60 segundos, a sobrecarga é reduzida para 4,26%, 6,25% e 7,22%, respectivamente.



(a) Impacto da recuperação na vazão considerando um cenário com falha.

(b) Impacto do *snapshot* na vazão considerando um operador de 100MB.

Figura 4. Análise do mecanismo de *snapshot* e recuperação.

6. Conclusão

Este trabalho apresentou uma nova interface para abstrair a programação de pipelines lineares tolerantes a falhas em sistemas distribuídos, com o objetivo de equilibrar desempenho e facilidade de desenvolvimento. Foi descrita uma metodologia portável para recuperação de falhas em ambientes MPI, baseada no protocolo ABS e em um agente monitor. A solução mostrou-se eficaz ao abstrair os detalhes de baixo nível típicos da construção de aplicações de *streaming* com MPI em C++. Em termos de desempenho, a abordagem demonstrou ser comparável às implementações com MPI puro, impondo um baixo *overhead* em troca de maior abstração. Os testes com falha induzida evidenciaram a capacidade da solução de recuperar-se e manter a estabilidade da execução do pipeline. Trabalhos futuros incluem comparar o desempenho com outras soluções do estado da arte, bem como a ampliação dos testes com falhas induzidas para contemplar outros cenários.

Agradecimentos

Este trabalho teve o suporte financeiro parcial da Empresa SAP, CAPES, CNPq (Nº 306511/2021-5) e FAPERGS 09/2023 PqG (Nº 24/2551-0001400-4).

Referências

Akidau, T. et al. (2015). The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803.

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

Andrade, G. et al. (2022). Analyzing Programming Effort Model Accuracy of High-Level Parallel Programs for Stream Processing. In *48th SEAA 2022*, pages 229–232, Gran Canaria, Spain. IEEE.

Apache (2025). Apache Spark Streaming.

Bland, W. et al. (2013). Post-failure recovery of mpi communication capability: Design and rationale. *IJHPCA*, 27(3):244–254.

Bouteiller, A. et al. (2006). Mpich-v project: A multiprotocol automatic fault-tolerant mpi. *IJHPCA*, 20(3):319–333.

Carbone, P. et al. (2015). Lightweight Asynchronous Snapshots for Distributed Dataflows.

Carbone, P. et al. (2017). State Management in Apache Flink: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.*, 10(12):1718–1729.

Consel, C. et al. (2003). Spidle: A dsl approach to specifying streaming applications. In Pfenning, F. and Smaragdakis, Y., editors, *GPCE*, pages 1–17. Springer.

Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. (2013). A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326.

Gencer, C. et al. (2021). Hazelcast jet: Low-latency stream processing at the 99.99 th percentile. *arXiv preprint arXiv:2103.10169*.

Gherardi, L., Brugali, D., and Comotti, D. (2012). A java vs. c++ performance evaluation: a 3d modeling benchmark. In *SIMPAR 2012*, pages 161–172. Springer.

Grant, S. and Voorhies, R. (2025). The OmpSs Programming Model.

Griebler, D. (2016). *Domain-Specific Language & Support Tool for High-Level Stream Parallelism*. PhD thesis, PPGCC - PUCRS, Porto Alegre, Brazil.

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

Joshi, S. and Vadhiyar, S. (2025). Fthp-mpi: Towards providing replication-based fault tolerance in a fault-intolerant native mpi library. *arXiv preprint arXiv:2504.09989*.

Löff, J., Hoffmann, R. B., Pieper, R., Griebler, D., and Fernandes, L. G. (2022). DSParLib: A C++ Template Library for Distributed Stream Parallelism. *International Journal of Parallel Programming*, 50(5):454–485.

Manchana, R. (2015). Java virtual machine (jvm): Architecture, goals, and tuning options. *International Journal of Scientific Research and Engineering Trends*, 1(3):42–52.

Nielsen, F. (2016). *Introduction to HPC with MPI for Data Science*. Springer.

Pathirana, P., Jankowski, M., and Allen, S. (2015). *Storm Applied: Strategies for real-time event processing*.

Pop, A. and Cohen, A. (2013). Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4).

Rocco, R. et al. (2024). Extending the legio resilience framework to handle critical process failures in mpi. In *2024 32nd PDP*, pages 44–51. IEEE.

Shahzad, F. et al. (2018). Craft: A library for easier application-level checkpoint/restart and automatic fault tolerance. *IEEE TPDS*, 30(3):501–514.

Yin, F. and Shi, F. (2022). A comparative survey of big data computing and hpc: From a parallel programming model to a cluster architecture. *International Journal of Parallel Programming*, 50(1):27–64.