Implementação de Tolerância a Falhas no Método Lattice Boltzmann para Execução Resiliente em Instâncias Efêmeras da AWS

Rafael Luis Sol Veit Vargas[1], **Vanderlei Munhoz**[1,2], **Márcio Castro**[1]

[1]Universidade Federal de Santa Catarina (UFSC)
LaPeSD – Florianópolis – Brasil

[2]University of Bordeaux (UB), Inria
LaBRI – Talence – França

rafael.s.vargas18@gmail.com, vanderlei.munhoz-pereira-filho@inria.fr
marcio.castro@ufsc.br

Resumo. *Este artigo investiga o desempenho e o custo financeiro do uso de mecanismos de tolerância a falhas no método Lattice Boltzmann (LBM) executado em instâncias efêmeras (spot) da Amazon Web Services (AWS). Duas estratégias de recuperação são implementadas com a extensão ULFM da biblioteca MPI: (i) preemptiva, que suspende a aplicação até a alocação de uma nova instância usando persistência em disco; e (ii) não preemptiva, que permite a continuidade da execução com um número reduzido de instâncias usando persistência em memória. Os resultados indicam que a abordagem não preemptiva proporciona recuperação quase imediata, com um impacto no desempenho pós-falha. Já a abordagem preemptiva evita essa degradação, mas apresenta maior tempo de recuperação. Conclui-se que a estratégia não preemptiva com persistência em memória pode reduzir os custos financeiros em até 32%, mesmo com a ocorrência de falhas.*

1. Introdução

O avanço da Computação em Nuvem consolidou-a como uma área de intensa pesquisa, destacando-se pela capacidade de atender, com eficiência e menor custo, às demandas voláteis da sociedade. Tecnologias de virtualização e containerização simplificaram o acesso aos serviços, enquanto o modelo de pagamento sob demanda e as economias de escala tornaram a nuvem uma alternativa possível para execução de aplicações de *High Performance Computing* (HPC) [Netto et al. 2017]. Todavia, a migração de aplicações HPC para ambientes de nuvem apresenta desafios significativos que exigem considerável esforço de adaptação.

Um dos principais obstáculos é o complexo e demorado processo de migração de aplicações HPC legadas, pois estas não foram originalmente projetadas para um ambiente fundamentalmente distinto [Munhoz et al. 2022, Munhoz et al. 2023]. Adicionalmente, a abstração de *hardware*, uma característica inerente e vantajosa da nuvem, torna-se uma desvantagem para aplicações HPC, que frequentemente empregam otimizações de baixo nível diretamente no código-fonte para maximizar o desempenho [Netto et al. 2017].

A introdução de instâncias efêmeras, comumente conhecidas como *spot instances*, ampliou ainda mais o potencial de economia no aluguel de infraestrutura em nuvens

públicas, oferecendo reduções de custo de até 90% [Munhoz and Castro 2023]. Em contrapartida, essa modalidade concede ao provedor o direito de revogar a instância a qualquer momento para atender a picos de demanda. Contudo, estudos demonstram que a implementação de mecanismos de tolerância a falhas pode tornar o uso de instâncias *spot* economicamente viável e confiável, mesmo para aplicações com requisitos críticos de disponibilidade, como as aplicações *web* [Qu et al. 2016].

Nesse sentido, considerando a introdução da extensão *User-Level Failure Mitigation* (ULFM) à biblioteca *Message Passing Interface* (MPI) e as capacidades de personalização dos processos de *checkpointing* e recuperação que ela proporciona, este artigo analisa a viabilidade técnica e financeira da implementação de tolerância a falhas no *Lattice Boltzmann Method* (LBM), um método euleriano para simulação de dinâmica de fluidos, para viabilizar a sua execução em instâncias *spot* fornecidas pela *Amazon Web Services* (AWS). As principais contribuições deste trabalho residem na implementação e análise comparativa de duas estratégias de tolerância a falhas: uma preemptiva, com *checkpoints* em disco, e uma não preemptiva, com *checkpoints* em memória e *oversubscribe*. Os resultados quantificam o *trade-off* entre custo e desempenho, demonstrando que a abordagem não preemptiva, embora imponha um sobrecusto que duplica o tempo por iteração após a falha de um *rank* (processo MPI), oferece uma recuperação mais rápida, tornando-se a mais rápida no geral quando as falhas ocorrem tardiamente. Em contrapartida, a estratégia preemptiva apresenta um alto custo de recuperação, mas retorna o desempenho da aplicação ao estado original. Na versão não preemptiva, o uso de instâncias *spot* gerou uma economia financeira de 32% em detrimento a um aumento de 42% no tempo total de execução, mesmo com a ocorrência da falha de um *rank*.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta os fundamentos necessários para contextualizar os avanços na Computação em Nuvem e como esses avanços impulsionaram os estudos sobre a migração de aplicações HPC, além de abordar conceitualmente o método LBM. A Seção 3 discute os trabalhos relacionados e como eles convergem para o estudo proposto neste artigo. A Seção 4 desenvolve, de forma técnica, a proposta de implementação de tolerância a falhas no método LBM. Por fim, os resultados obtidos são apresentados na Seção 5 e discutidos e consolidados na Seção 6.

2. Fundamentação Teórica

Esta seção apresenta a base teórica desta pesquisa: primeiro, a Computação em Nuvem como paradigma consolidado de infraestrutura elástica; depois, a evolução dos provedores de *Infrastructure as a Service* (IaaS), que com máquinas virtuais otimizadas, GPUs e redes de alta velocidade, tornam a nuvem atrativa para cargas de trabalho de HPC. Por fim, descreve-se o LBM e sua implementação no *benchmark* da suíte SPEChpc® 2021, objeto de estudo deste trabalho.

2.1. Computação de Alto Desempenho em Nuvens Públicas

A Computação em Nuvem (*Cloud Computing*) transformou a forma de consumir e gerenciar recursos computacionais. Ela viabiliza o acesso, via internet, a um conjunto compartilhado de recursos configuráveis tais como servidores, armazenamento, redes e aplicações de forma elástica e escalável, permitindo que as organizações ajustem dinamicamente a

capacidade computacional conforme suas necessidades. Isso contrasta diretamente com o modelo tradicional *on-premise*, caracterizado por altos investimentos iniciais em *hardware* e pela complexidade na manutenção e atualização da infraestrutura física local.

O avanço das instâncias aceleradas com GPUs e redes de alta velocidade tornou a nuvem atrativa para aplicações intensivas típicas de HPC. Contudo, instâncias com desempenho comparável a *clusters* dedicados têm custo elevado, podendo superar o de aquisição e manutenção de um *cluster* próprio [Munhoz et al. 2022]. Para reduzir custos, provedores de IaaS oferecem instâncias efêmeras, até 90% mais baratas que as sob demanda. Todavia, por poderem ser encerradas pelo provedor a qualquer momento, a execução de aplicações nessas instâncias necessita de tolerância a falhas. Entretanto, por padrão, o MPI não possui suporte nativo a falhas; na ocorrência de falhas, toda a aplicação é encerrada.

Os mecanismos mais comuns de tolerância a falhas em aplicações HPC seguem a abordagem *Checkpoint/Restart*, como nas ferramentas *Berkeley Lab Checkpoint/Restart* (BLCR) e *Distributed MultiThreaded CheckPointing* (DMTCP). Por operarem em nível de Sistema Operacional (SO), dispensam alterações no código, mas, por não conhecerem as etapas internas da aplicação, apresentam desempenho inferior e escalabilidade limitada à medida que aumentam as falhas [Munhoz et al. 2022]. Para superar essa limitação, a extensão ULFM adiciona resiliência ao MPI, permitindo implementar *checkpoints* específicos no código. Assim, a aplicação pode se recuperar de falhas e prosseguir sem reiniciar do zero.

2.2. O Método Lattice Boltzmann (LBM)

O LBM é uma técnica cada vez mais usada para simular fluidos [Chen and Doolen 1998]. Em vez de resolver as equações clássicas da fluidodinâmica, que descrevem o fluido como um material contínuo, ele trata o fluido como uma coleção de "pacotes" de partículas fictícias que se movem e colidem em uma grade. Sua principal vantagem é lidar com cenários complexos, como fluxo sanguíneo em artérias, petróleo em rochas porosas ou escoamento de ar em geometrias intrincadas, como a de um avião.

O LBM surgiu como alternativa eficiente para simular fenômenos desafiadores aos métodos convencionais, como turbulência, mistura de fluidos e partículas suspensas. O método opera em nível mesoscópico, entre moléculas e fluxo visível. Em vez de rastrear cada partícula, o que seria computacionalmente custoso, o LBM simplifica o problema, rastreando em cada ponto da grade a probabilidade de pacotes de partículas se moverem em direções predefinidas. Assim, captura a física essencial de forma estatisticamente precisa e computacionalmente simples.

O algoritmo do LBM funciona em um ciclo contínuo de dois passos simples que se repetem em toda a grade [Calore et al. 2017]:

1. **Propagação ou *Streaming*:** Todos os pacotes de partículas na grade se movem simultaneamente para o nó vizinho na direção em que estavam viajando. É um passo de movimento sincronizado;
2. **Colisão:** Em cada nó da grade, os pacotes de partículas que acabaram de chegar interagem entre si. Essa operação não é uma simulação física literal, mas um cálculo matemático que redistribui as probabilidades associadas a cada partícula,

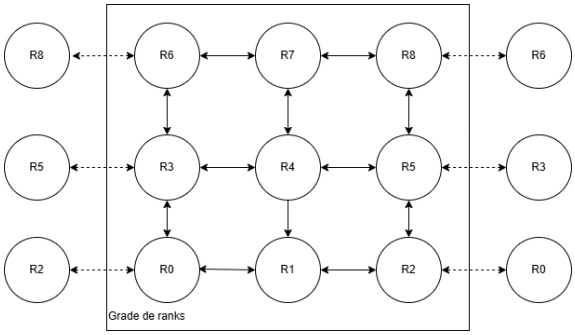

Figura 1. Troca de massa entre os *ranks* em uma grade 3x3 na aplicação LBM do SPEChpc.

alterando a direção dos pacotes de acordo com regras que conservam massa e momento, imitando, portanto, o que aconteceria em um fluido real.

A *Standard Performance Evaluation Corporation* (SPEC) disponibiliza diversos *benchmarks* para a comunidade científica. Dentre eles, destaca-se o SPEChpc, que inclui uma aplicação baseada na técnica de LBM com MPI para simular a troca de massa entre nós de uma grade. Esta aplicação enfatiza a paralelização e a otimização de desempenho, utilizando bibliotecas como OpenMP e OpenACC.

Inicialmente, a aplicação recebe `GX` e `GY` como dimensões da matriz global e gera os dados iniciais a partir de uma semente (`SEED`). Após a inicialização, inicia-se um *loop* com de iterações em que todos os ranks se comunicam entre si, enviando os dados das bordas para os vizinhos e recebendo os dados de volta, promovendo um intercâmbio contínuo de informações.

A Figura 1 ilustra a metodologia de troca de informações entre os *ranks*. Observa-se que há comunicação entre os *ranks* das bordas no eixo `X`. Por exemplo, o R_2 considera o R_0 seu vizinho à direita, enquanto o R_0 considera o R_2 seu vizinho à esquerda. No entanto, por definição da SPEC, essa reconexão não ocorre no eixo `Y`: o R_0, por exemplo, não se comunica diretamente com o R_6.

3. Trabalhos Relacionados

A interseção entre HPC e Computação em Nuvem representa um campo de pesquisa crucial e em rápida expansão. Essa união é impulsionada pela demanda por escalabilidade, e pela significativa redução de custos proporcionada pelas instâncias *spot*. Contudo, a volatilidade dessas instâncias impõe o desafio de garantir a continuidade da execução após revogações pelo provedor de IaaS. Para contornar esse problema, a principal linha de pesquisa foca em estratégias de tolerância a falhas, com destaque para a técnica de *Checkpoint/Restart* (CR).

3.1. Técnicas de Tolerância a Falhas

As abordagens tradicionais, como o BLCR [Hargrove and Duell 2006], que opera no nível de SO, possuem a grande vantagem de serem transparentes para a aplicação, não necessitando de alterações no código-fonte. Entretanto, essa abordagem pode se tornar ineficiente e onerosa [Munhoz et al. 2022]. Em cenários com muitas falhas, o custo de criar

checkpoints pode anular os benefícios das instâncias *spot*, reduzindo o desempenho e aumentando o custo total de Máquinas Virtuais (VMs).

Para superar essas limitações, surgem soluções mais granulares. No nível do mecanismo de recuperação, um trabalho recente demonstra que a estratégia ULFM, integrada à aplicação, pode gerar economias substanciais ao ser combinada com instâncias *spot*, superando o desempenho do BLCR [Munhoz et al. 2022]. No entanto, os autores notam que essa vantagem depende de uma carga de trabalho suficientemente grande para justificar os custos de comunicação.

Além do mecanismo, a estratégia de temporização dos *checkpoints* é crucial. Em [Amoon et al. 2019] a ineficiência do uso de intervalos fixos foi discutida e uma abordagem reativa denominada *Dynamic Change Checkpoint (DCC)* foi proposta, onde o intervalo entre *checkpoints* é ajustado dinamicamente com base na taxa de falhas observadas no ambiente. Os resultados experimentais confirmaram que essa estratégia adaptativa melhora significativamente o tempo de resposta e o custo monetário da aplicação.

Nesse mesmo sentido, trabalhos recentes utilizaram a biblioteca *Scalable Checkpoint/Restart* (SCR) para realização de *checkpointing* multinível [Moody et al. 2010] com o intuito de reduzir o gargalo de Entrada/Saída (E/S). Este tipo de abordagem prioriza o salvamento de *checkpoints* em armazenamentos locais e rápidos (RAM ou discos SSD), que são ordens de magnitude mais velozes que o sistema de arquivos paralelo. Embora exija modificações no código da aplicação via *Application Programming Interface* (API), o princípio do SCR de otimizar a hierarquia de armazenamento é diretamente aplicável à nuvem, onde a gestão de E/S e o uso de armazenamento efêmero são cruciais para a tolerância a falhas.

3.2. Armazenamento de *Checkpoint*

Em [Brum et al. 2023], os autores analisam como a escolha do armazenamento impacta o processo de *checkpointing*, demonstrando um balanço crítico entre a velocidade de escrita e o paralelismo na recuperação. Os autores apontam que, enquanto armazenamentos em bloco como o Amazon *Elastic Block Store* (EBS) são mais eficientes para salvar o estado e minimizar o sobrecusto da aplicação, serviços de objeto como o *Amazon S3* são frequentemente preferidos. Essa preferência se deve à capacidade do S3 de permitir acesso concorrente por múltiplos nós, uma característica crucial para a recuperação paralela de um *cluster* que o EBS não oferece.

Este trabalho se distingue do estado da arte atual ao analisar o comportamento do custo de desenvolvimento e execução no contexto específico de aplicações de resolução de dinâmica de fluidos, como o método LBM. A pesquisa desenvolve as versões preemptivas e não preemptivas da aplicação durante a recuperação de falhas, desenvolvendo duas abordagens de persistência de dados, uma em disco e outra em memória, sendo que esta permitirá a modificação da quantidade de falhas simultâneas toleráveis e recuperáveis. O objetivo é, com isso, otimizar e potencialmente diminuir o custo de criação de *checkpoints*, oferecendo uma análise mais aprofundada da viabilidade e eficiência do LBM em ambientes de nuvem voláteis.

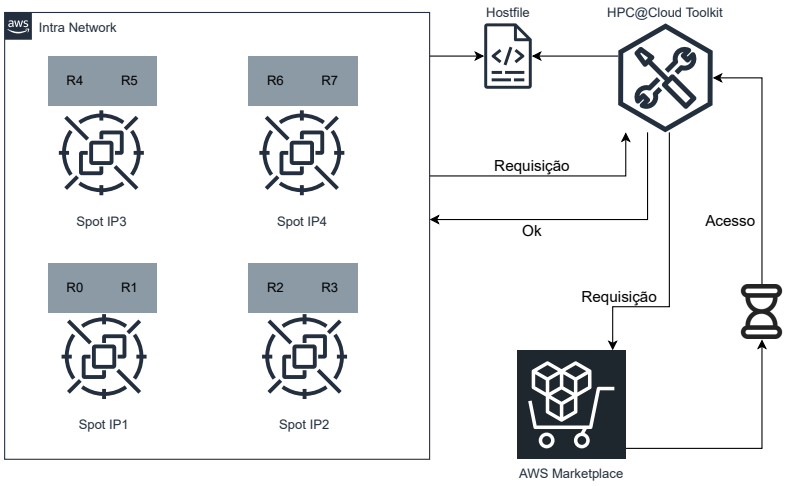

Figura 2. Estrutura de componentes da aplicação e fluxo de comunicação para requisição de nova instância.

4. Tolerância a Falhas na Aplicação LBM

Este artigo propõe a arquitetura ilustrada na Figura 2 para a requisição de novas instâncias *spot* no caso de falhas. A figura apresenta uma possível alocação de processos (ou *ranks* MPI) em um *cluster* de instâncias *spot* disponíveis no *Marketplace* da AWS e provisionadas com auxílio da ferramenta HPC@Cloud [Munhoz and Castro 2023]. A biblioteca MPI permite que uma mesma instância hospede múltiplos *ranks*; entretanto, isso exige atenção à quantidade de núcleos e à memória disponíveis na instância, para evitar sobrecarga. Os detalhes sobre os principais componentes são discutidos a seguir.

4.1. Checkpointing

Conforme definido pela extensão ULFM, a responsabilidade pela implementação da estratégia de tolerância a falhas recai sobre o desenvolvedor. Este artigo desenvolve duas abordagens diferentes de *checkpoint*:

Em disco O *checkpoint* permite que a execução seja recuperada mesmo quando há falha de todos os *ranks*, pois o armazenamento do estado da aplicação é feito em memória não volátil. A Figura 3 ilustra como este mecanismo foi implementado neste artigo. Nesta abordagem, todos os *ranks* salvam seus dados em um EBS compartilhado entre todas as instâncias.

Em memória Usando uma abordagem *Ring-Based*, em uma execução com n nós, cada nó R_i armazena seus próprios dados locais e os dados dos x *ranks* anteriores, isto é, $\{R_{(i-1) \bmod n}, \ldots, R_{(i-x) \bmod n}\}$, onde x é um valor positivo definido pelo usuário que indica o número de *ranks* anteriores cujos dados são replicados (ou seja, x representa a quantidade de falhas simultâneas que a aplicação é capaz de suportar). A Figura 4 ilustra o fluxo de dados para $x = 2$ e $n = 8$.

Ambas as abordagens utilizam mecanismos de sincronização nativos do MPI, como o $MPI\_Barrier$, que garantem que o processamento seja completado com sucesso. Dessa forma, caso ocorram falhas durante o processo de *checkpoint*, todos os

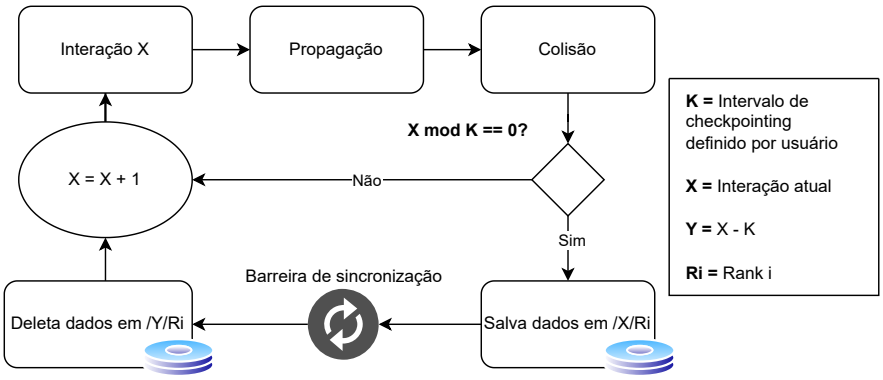

Figura 3. Algoritmo de *checkpoint* em disco (Amazon EBS) executando em *rank* Ri.

ranks reagem e iniciam o procedimento de recuperação de falhas. Além disso, como ilustrado na Figura 3, os algoritmos cuidam da liberação da memória associada a *checkpoints* antigos somente após a garantia de que o novo processo de *checkpoint* foi finalizado com sucesso em todos os *ranks*.

4.2. Identificação de Falhas

A identificação de falhas é um recurso nativo do ULFM, que possibilita detectar automaticamente falhas de *ranks* durante a execução paralela. No contexto do MPI, os *ranks* se comunicam por meio de comunicadores, que são estruturas lógicas que agrupam os *ranks* participantes. Quando um *rank* falha, os *ranks* diretamente conectados a ele são imediatamente notificados e seu fluxo de execução é direcionado para uma função definida pelo usuário.

Neste artigo, essa função é a $handle\_exception$, que executa operações coletivas específicas da ULFM, como $MPIX\_Comm\_revoke$ e $MPIX\_Comm\_shrink$. Essas operações invalidam o comunicador atual e geram um novo, contendo apenas os *ranks* sobreviventes, que então coordenam a recuperação e retomam a simulação. A função $handle\_exception$ é reentrante, permitindo que falhas ocorridas durante o próprio *rank* de recuperação também sejam tratadas de forma segura.

4.3. Recuperação de Falhas

Após a detecção da falha, a aplicação cria novos *ranks* para substituir aqueles que executavam no nó que falhou. As duas abordagens propostas estão descritas a seguir.

Abordagem preemptiva Após reconfigurar o comunicador com os *ranks* sobreviventes, a aplicação solicita ao HPC@Cloud o provisionamento de uma nova instância pela AWS para hospedar os *ranks* substitutos. A execução permanece bloqueada até a instância estar disponível. Em seguida, o *hostfile* é atualizado com o novo endereço IP, e os *ranks* são relançados via MPI_Comm_spawn. Essa abordagem tem como vantagem a clareza do fluxo de recuperação, mas sua limitação está na imprevisibilidade do tempo de alocação, que depende da disponibilidade do provedor.

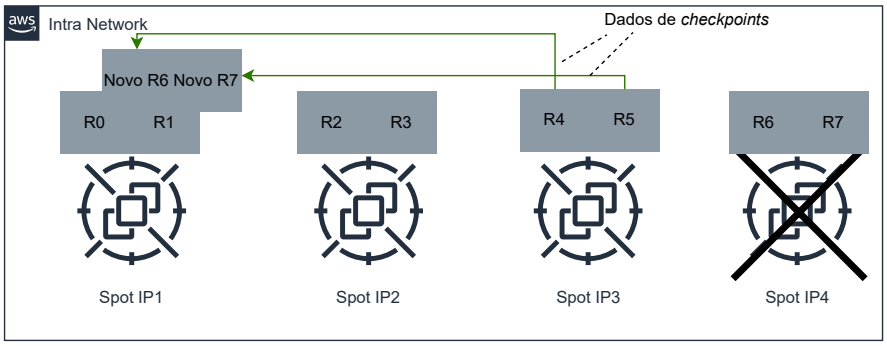

Figura 4. Realocação de *ranks* em nós existentes por meio de *oversubscribing*.

Abordagem não preemptiva Aplica-se o conceito de *oversubscribing*, executando mais *ranks* do que núcleos disponíveis em um nó. Como ilustrado na Figura 4, após a falha do nó $IP4$, os *ranks* R_6 e R_7 são reinicializados em outro nó ativo, o $IP1$. Por padrão, o MPI define a distribuição dos novos *ranks* nos nós existentes. A principal vantagem está na agilidade da etapa de recuperação: não é necessário aguardar o provisionamento de uma nova instância nem atualizar o *hostfile* para retomar a execução. Entretanto, essa abordagem compromete o desempenho da aplicação, uma vez que a execução prossegue com um número reduzido de nós, aumentando a disputa pelos recursos computacionais do nó que recebe os *ranks* sobressalentes.

4.3.1. Recuperação de Dados

Após a criação dos novos *ranks*, eles devem acessar os dados do último *checkpoint* salvo. Na abordagem com *checkpoint* em memória, que tolera até x falhas simultâneas, podem existir até x instâncias que possuam as informações de que o novo *rank* p necessita.

O algoritmo estabelece que o R_i de menor *rank* será responsável por fornecer os dados ao *rank* p. Assim, na situação ilustrada na Figura 4, com $x = 2$, tanto R_4 quanto R_5 armazenam os dados do *rank* falho R_6, mas cabe a R_4 realizar a entrega. Já no caso do *rank* falho R_7, apenas o *rank* R_5 possui os dados, assumindo, assim, a responsabilidade de fornecê-los. Já na abordagem em disco, essa comunicação se torna desnecessária, pois o *rank* p tem acesso direto aos *checkpoints*.

5. Resultados

Esta seção apresenta os resultados dos experimentos que avaliam duas abordagens de recuperação de falhas: (i) preemptiva com *checkpoint* em disco; e (ii) não preemptiva com *checkpoint* em memória.

5.1. Ambiente Experimental

Os testes foram executados em um *cluster* com quatro instâncias `C5.2xlarge` da AWS, provisionadas no modelo *on-demand* para garantir estabilidade. Cada instância possui 8 vCPUs (4 núcleos físicos), 16 GiB de memória e acesso a um armazenamento compartilhado via EBS. A aplicação de teste foi uma simulação do método LBM com 100

iterações sobre um domínio bidimensional de 2048x2048. Todos os resultados apresentados correspondem à média de 10 execuções.

Para a análise de custos, foram utilizados os preços de instâncias Linux na AWS, vigentes em 29 de julho de 2025: $0,1326/hora para instâncias *spot* e $0,34/hora para instâncias *on-demand*, representando um potencial de economia de 61%. O custo de instâncias *on-demand* foi adotado para o cenário de linha de base, enquanto para os demais cenários considerou-se o custo de instâncias *spot*.

5.2. Cenários de teste

Os cenários de teste foram montados para facilitar a comparação tanto entre as estratégias de *checkpointing* quanto entre as de recuperação de falhas. As falhas foram deliberadamente induzidas pelo término forçado de uma das instâncias ativas durante a execução.

Para evitar que o alto custo do *checkpoint* em disco prejudicasse a comparação dos resultados, foram definidos os seguintes intervalos para *checkpoint*: 20 iterações para a abordagem em disco e 10 para a em memória. Esses valores evitam que a discrepância de desempenho entre as estratégias de *checkpointing* distorça a análise das estratégias de recuperação de falhas.

Na abordagem preemptiva, uma falha é forçada após 1 minuto do início da execução. O momento da falha não interfere no tempo total de execução, já que esta é uma característica inerente desta abordagem. Entretanto, para a abordagem não preemptiva esse fator é crucial. Portanto, foram criados três cenários com falhas controladas com base no percentual de conclusão da aplicação, a fim de analisar o efeito do custo adicional por iteração após a recuperação.

As analises foram feitas sob os seguintes cenários de execução:

Linha de base (sem tolerância a falhas) Execução da simulação até a sua conclusão sem a indução de falhas, estabelecendo uma métrica de referência para comparações de tempo e custo;

Abordagem preemptiva Após o término anormal de uma instância, a aplicação solicita um nó substituto à AWS por meio da ferramenta HPC e permanece bloqueada até que o novo nó esteja operacional. Com a nova instância disponível, o sistema é reiniciado a partir do último *checkpoint* salvo no EBS; e

Abordagem não preemptiva Nesta abordagem, os *ranks* sobressalentes são provisionados em nós já existentes. Para quantificar a degradação de desempenho, foram criados três cenários nos quais as falhas foram injetadas em diferentes estágios da simulação: ao atingir 25%, 50% e 75% do total de iterações.

5.3. Tempo de Execução e Custos

A Figura 5 e a Tabela 1 consolidam os resultados obtidos. A análise dos dados evidencia as vantagens e desvantagens de cada abordagem.

Na estratégia preemptiva, o tempo de recuperação dos nós é um fator crucial. Nos testes realizados com o HPC@Cloud Toolkit, o tempo médio para provisionar uma nova instância *on-demand* foi de 81 segundos, com desvio padrão de 13,39 segundos. Vale destacar que a disponibilidade dessas instâncias difere das instâncias *spot* e pode variar significativamente, tornando o tempo de recuperação imprevisível por natureza.

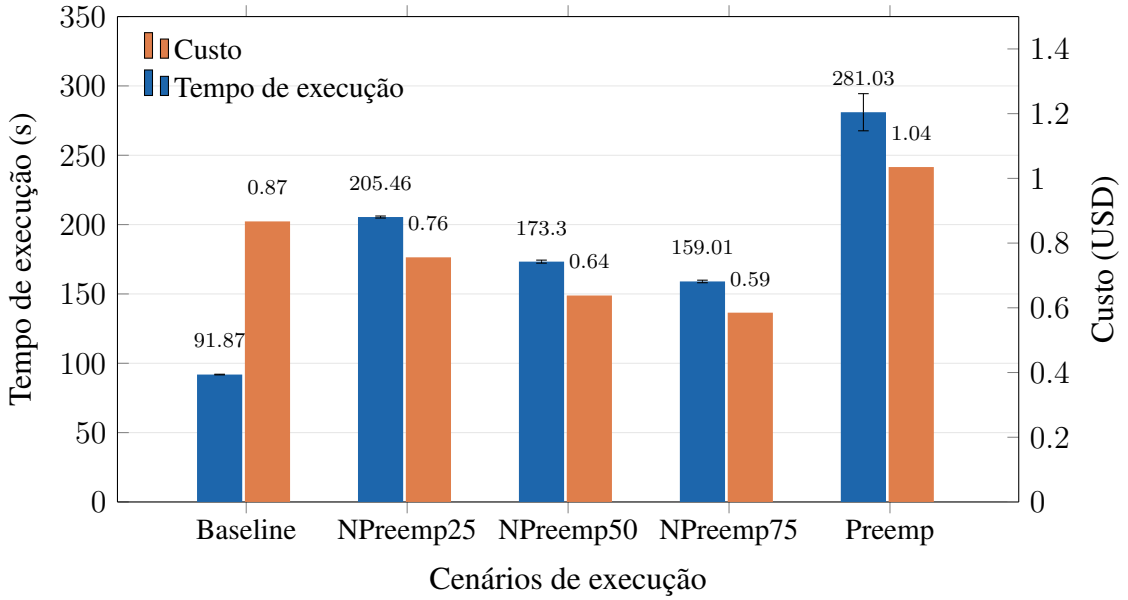

Figura 5. Médias de tempo de execução com desvio padrão e custo estimado.

Tabela 1. Médias de tempo de *checkpoint*, recuperação e sobrecusto por iteração após a falha (em segundos). Os percentuais de 25%, 50% e 75% na abordagem não preemptiva indicam que, respectivamente, 25%, 50% e 75% das iterações haviam sido executadas no momento em que a falha induzida ocorreu.

Cenário	Checkpoint (s)	Recuperação (s)	Sobrecusto[1] (s)	Total (s)
Baseline	-	-	-	91,87
Preemptiva	23,07 (Disco)	81,57	-	281,03
Não Preemptiva (25%)	1,91 (Memória)	10,52	0,94	205,46
Não Preemptiva (50%)	1,96 (Memória)	10,53	0,95	173,29
Não Preemptiva (75%)	1,95 (Memória)	10,53	0,94	159,01

[1] *Sobrecusto por iteração após a ocorrência da falha induzida.*

A análise dos tempos de *checkpoint* revela uma clara distinção de desempenho entre as abordagens. A persistência em memória é, em média, dez vezes mais rápida que a persistência em disco. Em contrapartida, o *checkpoint* em disco oferece maior robustez, flexibilidade e simplicidade de implementação, com duas vantagens principais: a capacidade de ser restaurado em execuções futuras e a tolerância a uma falha total do *cluster*. A abordagem em memória, por sua vez, é efêmera e depende da sobrevivência de ao menos um nó do grupo de replicação para viabilizar a recuperação.

A estratégia de *oversubscribing* introduz um custo adicional constante no tempo de cada iteração. Nos testes, a falha de um dos quatro *ranks* resultou na duplicação do tempo por etapa (de 0,9s para 1,8s, aproximadamente), pois a instância sobrevivente, ao passar a hospedar dois *ranks*, teve seus recursos de CPU e memória compartilhados. Isso impactou diretamente a etapa de cálculo de colisão da simulação LBM, que é computacionalmente intensiva. Portanto, conclui-se que o número de falhas impacta diretamente o custo adicional das iterações subsequentes da aplicação.

A escolha da estratégia ótima, portanto, depende de um balanço entre múltiplos fatores, incluindo:

1. **Progresso da simulação no momento da falha:** Este fator determina por quanto tempo a aplicação executará com desempenho degradado;
2. **Latência de provisionamento de novas instâncias pela AWS:** O tempo necessário para que um novo recurso esteja disponível para a aplicação; e
3. **Estimativa do sobrecusto de desempenho:** Dependendo da quantidade de falhas durante a execução, o desempenho final da aplicação pode se tornar tão custoso ao ponto de inviabilizar a continuidade da execução sem que novas instâncias sejam alocadas.

6. Conclusão

Este artigo avaliou o impacto de duas estratégias de persistência e duas estratégias de recuperação de falhas na execução da aplicação *tightly-coupled* LBM extraída da suíte SPEChpc® 2021. A análise permitiu identificar as vantagens e limitações de cada abordagem, considerando o sobrecusto da versão não preemptiva, o tempo de restauração de instâncias na AWS e os fatores relevantes para configurar adequadamente aplicações HPC em ambientes de nuvem.

Os experimentos mostraram que, mesmo com falhas, o uso de instâncias *spot* aliadas a estratégias de tolerância a falhas pode ser vantajoso em termos de custo financeiro. Em um dos cenários avaliados, houve uma economia de 32% no custo total, mesmo com um aumento de 42% no tempo de execução, após uma falha ocorrida com 75% da simulação concluída. Também foi observado que todas as versões não preemptivas apresentaram redução de custo, ainda que com aumento no tempo de execução.

Conclui-se que instâncias *spot* podem gerar economias significativas para aplicações HPC *tightly-coupled*, como no caso da aplicação LBM. No entanto, os desafios técnicos e o custo da adaptação para um modelo tolerante a falhas exigem cautela. Uma migração gradual, começando por versões mais simples (como a não preemptiva com persistência em disco), pode ser uma alternativa viável antes da adoção de soluções mais complexas e otimizadas.

Como trabalho futuro, propõe-se o desenvolvimento de uma estratégia de tolerância a falhas adaptativa. A ideia é reiniciar imediatamente a aplicação usando a abordagem *oversubscribe* e, em paralelo, avaliar dinamicamente os fatores de custo e desempenho para decidir se é vantajoso solicitar novas instâncias AWS. Essa abordagem visa garantir a continuidade da execução mesmo durante períodos indefinidos de espera pela disponibilidade de novas instâncias *spot*, evitando interrupções prolongadas na simulação.

Agradecimentos

Este trabalho foi financiado parcialmente pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e pela Amazon Web Services (AWS) por meio da chamada CNPq/AWS nº 64/2022 (Créditos em Nuvem para Pesquisa).

Referências

Amoon, M., El-Bahnasawy, N., Sadi, S., and Wagdi, M. (2019). On the design of reactive approach with flexible checkpoint interval to tolerate faults in cloud computing systems. *Journal of Ambient Intelligence and Humanized Computing*, 10(11):4567–4577.

Brum, R., Teylo, L., Arantes, L., and Sens, P. (2023). *Ensuring Application Continuity with Fault Tolerance Techniques*, pages 191–212. Springer International Publishing, Cham.

Calore, E., Gabbana, A., Schifano, S., and Tripiccione, R. (2017). Optimization of lattice boltzmann simulations on heterogeneous computers. *The International Journal of High Performance Computing Applications*, 33(1):124–139.

Chen, S. and Doolen, G. D. (1998). Lattice boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30(Volume 30, 1998):329–364.

Hargrove, P. H. and Duell, J. C. (2006). Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conference Series*, 46(1):494.

Moody, A., Bronevetsky, G., Mohror, K., and Supinski, B. R. d. (2010). Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11.

Munhoz, V., Bonfils, A., Castro, M., and Mendizabal, O. (2023). A performance comparison of hpc workloads on traditional and cloud-based hpc clusters. In *2023 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 108–114.

Munhoz, V. and Castro, M. (2023). Enabling the execution of hpc applications on public clouds with hpc@cloud toolkit. *Concurrency and Computation: Practice and Experience*, 36.

Munhoz, V., Castro, M., and Mendizabal, O. (2022). Strategies for fault-tolerant tightly-coupled hpc workloads running on low-budget spot cloud infrastructures. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 263–272.

Netto, M., Calheiros, R., Rodrigues, E., Cunha, R., and Buyya, R. (2017). Hpc cloud for scientific and business applications: Taxonomy, vision, and research challenges. *ACM Computing Surveys*, 51.

Qu, C., Calheiros, R. N., and Buyya, R. (2016). A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. *Journal of Network and Computer Applications*, 65:167–180.