

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIA DA COMPUTAÇÃO

Rafael Luis Sol Veit Vargas

**Análise de Viabilidade de Aplicações HPC na Nuvem: Implementação de
tolerância a falhas no método Lattice Boltzmann para execução resiliente em
instâncias efêmeras**

Florianópolis
2025

Rafael Luis Sol Veit Vargas

**Análise de Viabilidade de Aplicações HPC na Nuvem:
Implementação de tolerância a falhas no método Lattice
Boltzmann para execução resiliente em instâncias efêmeras**

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Ciência da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Márcio Bastos Castro, Dr.

Florianópolis

2025

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Vargas, Rafael Luis Sol Veit
Análise de Viabilidade de Aplicações HPC na Nuvem:
Implementação de tolerância a falhas no método Lattice
Boltzmann para execução resiliente em instâncias efêmeras /
Rafael Luis Sol Veit Vargas ; orientador, Márcio Bastos
Castro, 2025.
52 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Centro Tecnológico,
Graduação em Ciências da Computação, Florianópolis, 2025.

Inclui referências.

1. Ciências da Computação. 2. HPC em Nuvem. 3. Instâncias
Spot. 4. MPI e ULFM. 5. Lattice Boltzmann. I. Castro,
Márcio Bastos. II. Universidade Federal de Santa Catarina.
Graduação em Ciências da Computação. III. Título.

Rafael Luis Sol Veit Vargas

Análise de Viabilidade de Aplicações HPC na Nuvem: Implementação de tolerância a falhas no método Lattice Boltzmann para execução resiliente em instâncias efêmeras

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo curso de Graduação em Ciência da Computação.

Florianópolis, 5 de dezembro de 2025.

Banca Examinadora:

Prof. Márcio Bastos Castro, Dr.
Orientador
Universidade Federal de Santa Catarina

Prof. Odorico Machado Mendizabal, Dr.
Avaliador
Universidade Federal de Santa Catarina

Prof. Eduardo Camilo Inacio, Dr.
Avaliador
Universidade Federal de Santa Catarina

A Deus,
a minha família em Cuiabá,
e a minha esposa, Isabel Akemi.

AGRADECIMENTOS

Gostaria de agradecer primeiramente à minha esposa, Isabel Akemi, que esteve comigo pela maior parte da minha graduação e sempre me apoiou para que eu continuasse nos estudos.

Também gostaria de agradecer à minha família, aos meus pais, Alex Vargas e minha mãe, Laura Carolina, por me ensinarem os caminhos corretos a serem seguidos, me corrigindo e me guiando para que eu me tornasse o homem que sou hoje. Também agradeço ao meu irmão, Gabriel Vargas, por ser meu parceiro nos momentos de dificuldade na minha juventude.

Agradeço ainda aos meus avós, Carlos Vargas, Tanha Vargas e Natália Sol, por também me guiarem e me apoiarem para me manter na escola desde o ensino fundamental.

Aqui em Florianópolis tenho muitos amigos para agradecer, primeiramente aos meus colegas de curso: Rodrigo Carvalho, João Volpato, João Bonomo, Thiago, Fillipi, Leonardo, Arthur, Caio, Enzo e Rafael Oliveira, por estarem comigo durante as cadeiras que fizemos aqui na UFSC; por todos os almoços ou jantas no RU e pelas discussões de Org que tivemos na fila enquanto tínhamos a prova do grande LC. Também agradeço pelas ajudas em todos os trabalhos da faculdade e por toda a rede de apoio profissional que montamos, com diversas recomendações e o pula-pula de empresa para empresa.

Agradeço ao Leonardo por me chamar para trabalhar na Camerge, onde fui apresentado pela primeira vez ao verdadeiro mundo corporativo e onde me moldei como desenvolvedor de software.

Por fim, acima de tudo isso, agradeço a Deus pela sabedoria que me deu e pela capacidade de levantar todos os dias da cama e enfrentar as dificuldades que minha carne possui. Que Deus abençoe a todos que estão lendo este agradecimento e lembrem-se:

Porque os meus pensamentos não são os vossos pensamentos, nem os vossos caminhos os meus caminhos, diz o Senhor. Porque, assim como os céus são mais altos do que a terra, assim são os meus caminhos mais altos do que os vossos caminhos, e os meus pensamentos mais altos do que os vossos pensamentos.

Isaías 55:8–9

RESUMO

A Computação em Nuvem consolidou-se como uma solução eficiente e de baixo custo frente aos *clusters* tradicionais, especialmente com a introdução das Instâncias Efêmeras, conhecidas como instâncias *Spot*, que podem proporcionar economias de até 90% no aluguel de infraestrutura. Contudo, a volatilidade dessas instâncias - que podem ser revogadas a qualquer momento pelo provedor - impõe desafios críticos para aplicações de Computação de Alto Desempenho (HPC), que exigem alta disponibilidade e baixa latência. A migração de aplicações HPC legadas para a nuvem também é dificultada pela abstração de hardware e pela necessidade de migração do paradigma do código-fonte. Estudos recentes indicam que abordagens tradicionais de tolerância a falhas, como o uso de *checkpoints* externos à aplicação, como o *Distributed MultiThreaded CheckPointing* (DMTCP) ou *Berkeley Lab Checkpoint/Restart* (BLCR), podem ser ineficientes e gerar custos elevados em cenários com alta taxa de falhas. Como alternativa, a extensão *User-Level Failure Mitigation* (ULFM) do padrão *Message Passing Interface* (MPI) apresenta-se como uma solução eficiente, permitindo a recuperação dinâmica e adaptativa da aplicação sem a necessidade de aguardar indefinidamente a disponibilização de uma nova instância. Este trabalho propõe a adaptação do benchmark *Lattice Boltzmann Method* (LBM) da suíte SPEChpc® 2021 para incorporar a extensão ULFM, avaliando sua viabilidade técnica e econômica em ambientes de nuvem baseados em instâncias *Spot*. Os experimentos foram conduzidos na plataforma *Amazon Web Services* (AWS), comparando o tempo de execução da aplicação em diferentes versões de tolerância a falhas em instâncias do tipo *On-Demand* da AWS. Os resultados demonstram que o uso de instâncias *Spot* é viável para aplicações HPC, permitindo até 32% de economia mesmo com aumento de 42% no tempo de execução. A abordagem adaptativa, baseada em ULFM e checkpoints em disco, mostrou-se a mais eficaz ao anular o sobrecusto em execuções longas, viabilizando o uso de recursos efêmeros sem comprometer a continuidade da simulação.

Palavras-chave: Computação em Nuvem. HPC. Instâncias Spot. MPI. ULFM. Lattice Boltzmann.

ABSTRACT

Cloud computing has established itself as an efficient and cost-effective solution compared to traditional clusters, especially with the introduction of ephemeral instances, known as Spot instances, which can provide savings of up to 90% on infrastructure costs. However, the volatility of these instances—which can be revoked by the provider at any time—poses critical challenges for HPC applications that demand high availability and low latency. Furthermore, migrating legacy HPC applications to the cloud is hindered by hardware abstraction and the need to shift the source code paradigm. Recent studies indicate that traditional fault tolerance approaches, such as application-external checkpoints like DMTCP or BLCR, can be inefficient and incur high costs in scenarios with high failure rates. Alternatively, the ULFM extension of the MPI standard emerges as an efficient solution, enabling dynamic and adaptive application recovery without the need to wait indefinitely for a new instance to become available. This work proposes adapting the LBM benchmark from the SPEChpc® 2021 suite to incorporate the ULFM extension, evaluating its technical and economic viability in cloud environments based on spot instances. Experiments were conducted on the AWS platform, comparing application execution times across different fault tolerance versions on AWS on-demand instances. The results demonstrate that using spot instances is viable for HPC applications, allowing for up to 32% savings even with a 42% increase in execution time. The adaptive approach, based on ULFM and disk checkpoints, proved to be the most effective by offsetting overhead in long executions, enabling the use of ephemeral resources without compromising simulation continuity.

Keywords: Cloud Computing. HPC. Spot. MPI. ULFM. Lattice Boltzmann.

LISTA DE FIGURAS

Figura 1 – Troca de massa entre os <i>ranks</i> em uma grade 3x3 na aplicação LBM do SPEChpc® 2021	32
Figura 2 – Estrutura de componentes da aplicação e fluxo de comunicação para requisição de nova instância	39
Figura 3 – Fluxograma de execução da aplicação LBM com <i>checkpoints</i> periódicos.	41
Figura 4 – Fluxo de dados no algoritmo de <i>checkpointing</i> em disco (Amazon EBS)	41
Figura 5 – Fluxo de dados no algoritmo de <i>checkpointing</i> em memória para 6 <i>ranks</i> em 3 instâncias <i>spot</i>	42
Figura 6 – Operação de <i>oversubscribing</i> de novos processos em nós pré-existentes após falha.	45
Figura 7 – Fluxo de recuperação para versão adaptativa	46
Figura 8 – Médias de tempo de execução com desvio padrão e custo estimado. . .	55

LISTA DE TABELAS

Tabela 1 – Médias de tempo de <i>checkpoint</i> , 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.	54
--	----

LISTA DE ABREVIATURAS E SIGLAS

API	<i>Application Programming Interface.</i>	25, 30, 38
AWS	<i>Amazon Web Services.</i>	9, 11, 22, 23, 39, 40, 45, 46, 51, 52, 53, 55, 59
BLCR	<i>Berkeley Lab Checkpoint/Restart.</i>	9, 11, 21, 29, 30, 37
CR	<i>Checkpoint/Restart.</i>	29, 37
DCC	<i>Dynamic Change Checkpoint.</i>	37
DMTCP	<i>Distributed MultiThreaded CheckPointing.</i>	9, 11, 21, 30
EBS	<i>Elastic Block Store.</i>	37, 38, 41, 51, 52
HPC	Computação de Alto Desempenho.	9, 11, 21, 22, 23, 24, 25, 28, 29, 32, 37, 38, 39, 63
IaaS	Infraestrutura como Serviço.	21, 24, 37
LBM	<i>Lattice Boltzmann Method.</i>	9, 11, 13, 22, 23, 31, 32, 38, 39, 41, 51, 54, 57, 58
MPI	<i>Message Passing Interface.</i>	9, 11, 21, 22, 23, 24, 25, 30, 32, 33, 34, 40, 42, 44, 45, 53, 57
OpenMP	<i>Open Multi-Processing.</i>	25, 39
SCR	<i>Scalable Checkpoint/Restart.</i>	30, 38
SEU	<i>Single Event Upset.</i>	28
SPEC	<i>Standard Performance Evaluation Corporation.</i>	22, 32, 33
ULFM	<i>User-Level Failure Mitigation.</i>	9, 11, 21, 22, 25, 30, 31, 37, 39, 40, 42, 43, 44, 45, 57, 58
VMs	Máquinas Virtuais.	23, 37

SUMÁRIO

1	INTRODUÇÃO	21
1.1	CONTEXTUALIZAÇÃO	21
1.2	OBJETIVO GERAL	22
1.2.1	Objetivos Específicos	22
1.3	CONTRIBUIÇÕES	22
2	FUNDAMENTAÇÃO TEÓRICA	23
2.1	COMPUTAÇÃO EM NUVEM	23
2.2	COMPUTAÇÃO DE ALTO DESEMPENHO (HPC)	24
2.2.1	Message Passing Interface (MPI)	25
2.3	TOLERÂNCIA A FALHAS	26
2.3.1	Estratégias de Tolerância a Falhas	27
2.3.2	Estratégias de Checkpoint em Aplicações HPC	29
2.4	O MÉTODO LATTICE BOLTZMANN (LBM)	31
2.4.1	O Benchmark LBM da Suíte SPEChpc® 2021	32
2.4.2	Empacotamento dos dados	33
2.4.3	Comunicação assíncrona	34
2.4.4	Propagação, Colisão e Sincronização Global	34
3	TRABALHOS RELACIONADOS	37
3.1	CONTRIBUIÇÕES	38
4	PROPOSTA DE PESQUISA	39
4.1	CHECKPOINTS	40
4.2	IDENTIFICAÇÃO DE FALHAS	42
4.3	RECUPERAÇÃO DE FALHAS	42
4.4	VERSÕES PROPOSTAS	44
4.4.1	C/R Preemptivo	44
4.4.2	C/R Não Preemptivo: <i>Oversubscribe</i>	45
4.4.3	C/R Adaptativo	46
5	RESULTADOS	51
5.1	AMBIENTE EXPERIMENTAL	51
5.2	ORQUESTRADOR	51
5.3	CENÁRIOS	52
5.4	EXECUÇÃO	53
5.5	RESULTADOS	54
6	CONCLUSÃO	57

6.1	SÍNTESE DAS CONTRIBUIÇÕES TÉCNICAS	57
6.2	ANÁLISE COMPARATIVA DAS ESTRATÉGIAS	57
6.3	VIABILIDADE ECONÔMICA	58
6.4	CONSIDERAÇÕES FINAIS	58
6.5	TRABALHOS FUTUROS	59
	REFERÊNCIAS	61
A	CÓDIGO COMUM ÀS IMPLEMENTAÇÕES	77
B	IMPLEMENTAÇÃO: VERSÃO ADAPTATIVA (C/R ADAP- TATIVO)	99
C	IMPLEMENTAÇÃO: VERSÃO OVERSUBSCRIBE (TOLE- RÂNCIA A FALHAS)	113
D	IMPLEMENTAÇÃO: VERSÃO WITHOUTTOLERANCE (SEM TOLERÂNCIA A FALHAS)	131

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

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 viável aos *clusters* locais tradicionais, com serviços como Infraestrutura como Serviço (IaaS) concentrados em poucos provedores, que detêm cerca de 80% do mercado (Netto et al., 2017).

A introdução de Instâncias Efêmeras, comumente conhecidas como *Spot*, ampliou ainda mais o potencial de economia no aluguel de infraestrutura, oferecendo reduções de custo de até 90% (Munhoz; Castro, 2023a). Em contrapartida, essa modalidade concede ao provedor o direito de revogar a instância a qualquer momento para atender a picos de demanda. Contudo, estudos como o de Qu, Calheiros e Buyya (2016) demonstram que a implementação de mecanismos de tolerância a falhas, baseados na utilização de instâncias heterogêneas, pode tornar o uso de instâncias *Spot* economicamente viável e confiável, mesmo para aplicações com requisitos críticos de disponibilidade, como as aplicações *web*.

Apesar das vantagens, 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, que, conforme apontado por Munhoz, Castro e Mendizabal (2022) e Munhoz et al. (2023), não foram originalmente projetadas para um ambiente fundamentalmente distinto. Adicionalmente, Netto et al. (2017) enfatiza que 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.

Além disso, as opções de instâncias em nuvem com desempenho comparável ao de *clusters* HPC tradicionais, como aquelas aceleradas por GPUs ou FPGAs, possuem um custo elevado que, segundo Munhoz, Castro e Mendizabal (2022), pode superar o de adquirir e manter um *cluster* próprio. Este fator, somado à inerente suscetibilidade a falhas no ambiente de nuvem — especialmente ao utilizar Instâncias Efêmeras —, torna imprescindível a investigação de mecanismos de tolerância a falhas que sejam economicamente eficientes.

A extensão ULFM para o padrão de comunicação MPI representa uma abordagem superior em comparação a outras técnicas baseadas em *checkpoint*, destacada em Munhoz e Castro (2023b) por sua capacidade de restauração adaptativa e não bloqueante. Diferente de métodos bloqueantes como o BLCR, ou até mesmo versões mais recentes como o DMTCP, o ULFM permite que a aplicação continue executando após a falha de uma

instância, realocando dinamicamente as tarefas para os nós ativos e eliminando o tempo ocioso de espera por recursos do provedor. Essa resiliência é complementada por vantagens técnicas cruciais, como o suporte a redes de alta velocidade (e.g., *InfiniBand*). Embora exija modificações no código-fonte, essa abordagem confere à aplicação controle granular sobre a recuperação, garantindo maior eficiência e continuidade.

1.2 OBJETIVO GERAL

Este trabalho tem como objetivo geral analisar a viabilidade técnica e econômica da execução de aplicações de HPC em Instâncias Efêmeras na nuvem. Para isso, foi desenvolvido diferentes mecanismos de tolerância a falhas com a extensão ULFM no *benchmark* LBM da suíte SPEChpc® 2021, a fim de avaliar e comparar o impacto da resiliência nos custos e no tempo de execução em um ambiente sujeito a falhas, validando a eficácia da abordagem proposta.

1.2.1 Objetivos Específicos

- Caracterizar o modelo de programação paralela com o padrão MPI, detalhando as primitivas da extensão ULFM necessárias para a implementação de tolerância a falhas;
- Adaptar a aplicação de referência LBM, do *benchmark* SPEChpc® 2021 da *Standard Performance Evaluation Corporation* (SPEC), para operar de forma resiliente por meio da implementação de mecanismos de tolerância a falhas com ULFM;
- Realizar experimentos comparativos, executando versões tolerantes e não tolerantes a falhas, em diferentes casos de ocorrência de falhas e de tipos de instâncias na nuvem AWS, incluindo Instâncias Efêmeras (*Spot*) e sob demanda (*On-Demand*);
- Analisar comparativamente os resultados de desempenho (tempo de execução) e custo, a fim de quantificar a viabilidade e a eficiência da abordagem resiliente em diferentes cenários de ocorrência de falhas.

1.3 CONTRIBUIÇÕES

Parte das contribuições e resultados apresentados neste trabalho foram publicados e apresentados no *XXVI Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD 2025)*, realizado em Bonito, MS, Brasil (Vargas; Munhoz; Castro, 2025).

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo estabelece a fundamentação teórica necessária para a compreensão da pesquisa, abordando os conceitos-chave que definem seu escopo e metodologia. Inicialmente, discute-se a Computação em Nuvem como o paradigma consolidado de infraestrutura elástica. Em seguida, apresenta-se a Computação de Alto Desempenho (HPC), detalhando o perfil das aplicações que demandam tal poder computacional. Para elucidar o paralelismo nessas aplicações, explora-se a especificação Message Passing Interface (MPI), o padrão de fato para este domínio. O desafio da confiabilidade é então abordado, partindo da definição de falhas de sistema para, em seguida, apresentar as estratégias de tolerância a falhas mais consolidadas para HPC. Por fim, para contextualizar a aplicação prática de todos esses conceitos, o capítulo descreve o método Lattice Boltzmann (LBM) e detalha sua implementação no *benchmark* da suíte SPEChpc® 2021, que constitui o objeto de estudo deste trabalho.

2.1 COMPUTAÇÃO EM NUVEM

A Computação em Nuvem, ou *Cloud Computing*, representa uma profunda transformação no modo como recursos computacionais são consumidos e gerenciados. Trata-se de um paradigma que viabiliza o acesso, via Internet, a um conjunto compartilhado de recursos configuráveis — como servidores, armazenamento, redes e aplicações — que podem ser rapidamente provisionados e liberados com mínimo esforço de gerenciamento. A principal proposta de valor do modelo reside na sua elasticidade e escalabilidade, permitindo que as organizações ajustem dinamicamente a capacidade computacional às suas necessidades. Isso contrasta diretamente com o modelo tradicional *On-Premise*, caracterizado por altos investimentos iniciais em hardware e pela complexidade de manutenção e atualização da infraestrutura física.

O pilar tecnológico que sustenta a Computação em Nuvem moderna é a virtualização. Essa tecnologia permite a criação de uma representação virtual de um recurso físico, como um servidor, desacoplando o software (sistema operacional e aplicações) do hardware subjacente. Por meio de um software chamado *hypervisor*, um único servidor físico pode ser particionado para hospedar múltiplas Máquinas Virtuais (VMs) independentes e isoladas umas das outras.

No contexto da nuvem, as VMs são usualmente denominadas “instâncias”. A virtualização é fundamental para a eficiência do modelo, pois permite o compartilhamento e a otimização do uso dos recursos físicos, além de garantir a agilidade necessária para criar e destruir ambientes computacionais em questão de minutos, o que é um pré-requisito para a elasticidade prometida pela nuvem.

O ecossistema de Computação em Nuvem é dominado por grandes provedores globais, como AWS, Microsoft Azure e Google Cloud, que operam e mantêm *data centers*

massivos e geograficamente distribuídos. Esses provedores estruturam seus serviços em diferentes modelos, sendo o mais fundamental o de Infraestrutura como Serviço (IaaS).

No modelo IaaS, o cliente aluga a infraestrutura de TI básica — servidores (instâncias), redes e armazenamento — e é responsável por gerenciar o sistema operacional e as aplicações. Esse modelo oferece o mais alto nível de flexibilidade e controle sobre os recursos. Para atender a diversas demandas, os provedores oferecem um vasto catálogo de tipos de instâncias, incluindo opções otimizadas para computação intensiva e equipadas com aceleradores como GPUs e redes de alta velocidade, tornando a nuvem um ambiente cada vez mais atraente para cargas de trabalho de HPC.

2.2 COMPUTAÇÃO DE ALTO DESEMPENHO (HPC)

A Computação de Alto Desempenho (HPC), ou *High-Performance Computing*, refere-se ao uso de supercomputadores e *clusters* para resolver problemas computacionais de alta complexidade, que seriam impraticáveis em computadores convencionais devido ao tempo de processamento exigido. Conforme definido por (Munhoz; Castro, 2023a), a HPC envolve o processamento paralelo de dados e algoritmos em larga escala. Tradicionalmente, esses sistemas são compostos por *clusters* de computadores caracterizados por alto poder de processamento e conexões de rede rápidas (Netto et al., 2017).

O principal objetivo da HPC é impulsionar a inovação e a descoberta científica em diversas áreas, como ciências da vida, modelagem climática, mapeamento de genomas, finanças e engenharia aeroespacial, além de acelerar as mais recentes aplicações de big data e inteligência artificial.

Os sistemas HPC são projetados para extrair o mais alto desempenho possível da infraestrutura (Netto et al., 2017). Para isso, suas aplicações têm requisitos de desempenho rigorosos e são frequentemente otimizadas para rodar em hardware específico, aproveitando instruções especializadas do processador (Munhoz; Castro, 2023a). Uma característica central de muitas aplicações HPC, especialmente as chamadas *tightly coupled*, é a necessidade de alta comunicação entre processadores.

Para minimizar a latência da rede, os nós computacionais são fisicamente próximos e interligados por tecnologias de alta velocidade e custo elevado, como *InfiniBand*, em vez de redes *Ethernet* tradicionais. A programação desses sistemas é predominantemente realizada com o padrão *de facto* para paralelismo em memória distribuída, a biblioteca MPI.

A tolerância a falhas é uma necessidade fundamental em HPC. Como as simulações podem durar dias ou semanas, a probabilidade de falha em um dos milhares de componentes do sistema torna-se alta. Losada et al. (2020) apontam que, à medida que os sistemas avançam para a era *exascale* (10^{18} operações por segundo), com milhões de núcleos, o tempo médio entre falhas diminui drasticamente, tornando as técnicas de tolerância a falhas essenciais. A especificação MPI, em sua forma tradicional, não possui

suporte nativo à resiliência; seu comportamento padrão é abortar toda a aplicação caso um único processo falhe. A abordagem clássica para contornar isso é o processo de *checkpoint/restart*, onde o estado da aplicação é salvo periodicamente.

Contudo, esse método gera uma sobrecarga considerável no sistema de arquivos. Para superar essa limitação, propostas como a extensão ULFM surgem para incluir capacidades resilientes no próprio padrão MPI, permitindo que a aplicação detecte a falha e continue sua execução de forma adaptativa, sem a necessidade de reiniciar do zero.

2.2.1 Message Passing Interface (MPI)

A MPI constitui-se como uma especificação de API e não uma biblioteca em si, estabelecendo o padrão predominante para a programação de aplicações paralelas em sistemas de memória distribuída. Dentre as implementações efetivas do MPI, destacam-se bibliotecas como *OpenMPI*, *MPICH* e *Intel MPI*, que fornecem as ferramentas práticas para comunicação e sincronização entre processos. Conforme afirmam Kohl et al. (2017) e Losada et al. (2020), o MPI é o modelo de comunicação mais comum e popular no domínio da HPC. Sua principal função é orquestrar a troca de dados e a sincronização entre processos paralelos. Na prática, é frequentemente utilizado em modelos híbridos, combinando-o com *Open Multi-Processing* (OpenMP) para a paralelização em memória compartilhada dentro de um mesmo nó computacional.

Com a ascensão da nuvem, a execução de aplicações MPI em infraestruturas virtualizadas tornou-se um campo de pesquisa ativo, com ferramentas como o *HPC@Cloud Toolkit*, proposto por Munhoz e Castro (2023a), que facilitam a orquestração de contêineres para a execução de aplicações MPI, utilizando tecnologias como o *Singularity* para garantir a portabilidade, com impacto insignificante no desempenho.

O ciclo de vida de uma aplicação paralela com MPI começa não por uma chamada direta ao executável, mas por meio de um comando lançador, como `mpirun` ou `mpiexec`. Esse comando é responsável por instanciar múltiplas cópias do mesmo programa em diferentes nós de um *cluster*, operando sob o modelo SPMD (*Single Program, Multiple Data*).

Embora todos os processos executem o mesmo código, cada processo pode ter um comportamento diferenciado com uso de instruções *if-else*. Para isso, o MPI atribui um identificador único para cada processo (denominado *rank*), o qual pode ser utilizado para realizar desvios no código de forma que cada processo execute um conjunto diferente de instruções. Da mesma forma, o *rank* pode ser utilizado para definir um endereço de memória que será lido/escrito pelo processo, um subintervalo no espaço de iterações de um laço (*for*) a ser paralelizado de forma que cada processo realize operações em índices diferentes de um vetor, entre outras estratégias possíveis.

O ambiente de execução é formalmente estabelecido pela chamada à função `MPI_Init` e deve ser concluído com `MPI_Finalize` antes do término do programa. Den-

tro deste escopo, um padrão de projeto comum envolve um processo mestre (tipicamente o de *rank* 0) que distribui porções de um problema ou dados para os demais processos utilizando operações coletivas, como `MPI_Bcast` ou `MPI_Scatter`.

Subsequentemente, cada processo executa sua tarefa computacional em paralelo, trocando informações com processos vizinhos quando necessário, por meio de operações ponto a ponto, como `MPI_Send` e `MPI_Recv`. Ao final da computação, os resultados parciais são frequentemente consolidados de volta no processo mestre através de operações de coleta, como `MPI_Gather` ou `MPI_Reduce`, para a composição da solução final.

2.3 TOLERÂNCIA A FALHAS

Todo software ou sistema desenvolvido pode apresentar comportamentos inesperados, resultando em resultados não determinísticos e insatisfatórios. No campo de sistemas computacionais, uma falha é compreendida como a causa fundamental, julgada ou hipotética, que pode levar a um comportamento incorreto do sistema. Conforme a taxonomia seminal de Avizienis et al. (2004), uma falha permanece em estado dormente até ser ativada, momento em que produz um erro. Essas causas podem ser classificadas de diversas formas, por exemplo, podem ser naturais, como defeitos de produção, ou de origem humana, sendo estas maliciosas ou não. De forma mais concreta, Dotti, Mendizabal e Santos (2005) modelam o comportamento de falhas como transições de estado específicas, exemplificando-as como *crash* (parada de um processo) ou *omission* (falha em enviar ou receber mensagens). No contexto de sistemas distribuídos complexos, Castro (2001) aborda as falhas Bizantinas, onde os componentes do sistema podem se comportar de maneira totalmente arbitrária, representando uma das classes de falha mais desafiadoras de se tolerar.

Nesse sentido, Schneider (1990) expande a discussão ao classificar o comportamento de componentes falhos em duas categorias representativas: Falhas Bizantinas e Falhas de Parada (*Fail-stop*). Enquanto a primeira assume a suposição mais fraca — onde componentes podem agir maliciosamente ou em conluio, exigindo $2t + 1$ réplicas para tolerar t falhas —, a segunda pressupõe que o componente apenas interrompe seu funcionamento de forma detectável. Para a maioria das aplicações, assumir falhas *fail-stop* é suficiente e computacionalmente mais eficiente, pois requer apenas $t + 1$ réplicas para garantir a disponibilidade do serviço.

Um erro é a manifestação de uma falha ativa, definido por Avizienis et al. (2004) como um desvio no estado interno do sistema em relação ao seu estado correto. Essencialmente, o erro é a consequência direta da ativação de uma falha, mas sua existência não garante um impacto externo. Muitos erros podem ser corrigidos ou contidos internamente, nunca se propagando para a interface de serviço. Um erro que é sinalizado por uma mensagem ou indicador é considerado detectado, enquanto um erro presente, mas não indicado, é classificado como latente. A ponte entre a causa (falha) e a consequência

(erro) é reforçada por Castro (2001), que conectam “*bugs* de software” — uma forma de falha — à ocorrência de “erros de software”, que são as manifestações desses *bugs* durante a execução. A detecção de um erro, como uma mensagem de *timeout* resultante de um processo que sofreu uma falha de *crash*, é o primeiro passo para que um sistema possa tentar se recuperar.

Por fim, a cadeia de propagação pode alcançar o *defeito*. O defeito — termo frequentemente traduzido de *failure* — manifesta-se quando o erro atinge a interface do usuário ou o comportamento observável do sistema, resultando em degradação da qualidade do serviço prestado Avizienis et al. (2004).

Assim, um defeito de software, popularmente conhecido como *bug*, corresponde à materialização visível de uma falha presente no código-fonte, seja ela decorrente de erros de lógica, implementação ou projeto.

Os autores, em especial Avizienis et al. (2004), estabelecem uma clara ordem causal que diferencia os conceitos e suas ocorrências. A sequência correta de eventos é:

- **Falha (*fault*):** É a origem de tudo, a causa raiz. Pode ser um defeito no código, uma falha de hardware, um erro humano na configuração ou uma interferência externa. Ela pode existir de forma latente no sistema por um longo período;
- **Erro (*error*):** Ocorre quando uma falha é ativada por alguma entrada ou condição específica. O erro é a consequência interna, um estado incorreto no sistema (ex: uma variável com valor errado, um ponteiro nulo). Nem todo erro é fatal; o sistema pode ser capaz de detectá-lo e corrigi-lo;
- **Defeito (*failure*):** Acontece quando um erro se propaga até a interface externa do sistema e o serviço entregue ao usuário desvia-se do comportamento esperado e especificado. Esta é a manifestação externa e observável do problema.

Em resumo, a falha é a causa, o erro é o estado incorreto interno, e o defeito é a consequência externa observável pelo usuário. Um defeito, como um *bug*, é simplesmente o fator observável após a ocorrência de toda essa cadeia de eventos.

A Computação em Nuvem introduz novas classes de falhas ligadas diretamente ao seu modelo de operação com múltiplos componentes. Além disso, essa natureza multi-localitária da infraestrutura pode causar falhas de serviço por degradação de desempenho em situações adversas de latência da rede global. Devido a isso, as chances de casos de omissão e quebra de componentes individuais de uma aplicação aumentam drasticamente.

2.3.1 Estratégias de Tolerância a Falhas

No contexto do desenvolvimento de sistemas, um conceito amplamente explorado é o de dependabilidade, que se refere à capacidade de um sistema em fornecer um serviço confiável e justificadamente seguro. Conforme Avizienis et al. (2004), os métodos para alcançar essa dependabilidade podem ser agrupados em quatro categorias principais:

- Prevenção de Falhas;
- Tolerância a Falhas;
- Remoção de Falhas;
- Previsão de Falhas.

Falhas são inevitáveis e podem surgir por diversas causas, desde eventos físicos, como quedas de servidores, até fenômenos naturais, como a incidência de radiação em componentes eletrônicos, que podem causar erros conhecidos como *Single Event Upset* (SEU). Além disso, com a redução do tempo médio entre falhas, torna-se cada vez mais importante que os sistemas sejam capazes de se adaptar e continuar operando corretamente mesmo após a ocorrência dessas falhas — ou seja, que sejam tolerantes a falhas.

Segundo Dotti, Mendizabal e Santos (2005), estratégias como o uso de *checkpoints* são fundamentais para garantir a consistência dos dados perante falhas em sistemas distribuídos, como em bancos de dados. Essas estratégias também são cruciais para aplicações *tightly coupled* que realizam balanceamento de carga entre múltiplos processos, um cenário comum em aplicações de alto desempenho computacional (HPC). A garantia de que simulações complexas possam continuar operando corretamente na presença de falhas é ainda mais importante ao considerarmos que essas simulações podem executar por vários dias.

Assim, podemos citar as seguintes metodologias para lidar com a ocorrência de falhas em sistemas:

Checkpoint e Recuperação A estratégia mais difundida e tradicionalmente aplicada para a tolerância a falhas em sistemas distribuídos é o *checkpoint* e recuperação. Como explicado por Dotti, Mendizabal e Santos (2005), a técnica consiste em salvar periodicamente um *snapshot* consistente do estado global do sistema em um armazenamento estável. Em caso de falha, o sistema é revertido para o último *checkpoint* válido e a execução é retomada a partir daquele ponto. As etapas de criação de *checkpoints* podem ser caracterizadas da seguinte forma:

- **Não Coordenadas:** Cada componente/processo do sistema cria seu *checkpoint* independentemente, gerando menor sobrecarga de sincronização ao custo de um processo de recuperação mais complexo, que gera um “efeito dominó” de *rollbacks*;
- **Coordenadas:** Requerem uma sincronização entre todos os processos para salvar completamente o estado do sistema, simplificando o processo de recuperação;
- **Incrementais:** Salvam as modificações entre o último *checkpoint* e o atual, reduzindo o volume de dados salvo em detrimento da complexidade de recuperação;

- **Adaptativas:** Utilizam metodologias para analisar o histórico de ocorrência de falhas para prever a próxima falha e, assim alterar a frequência de *checkpoints*, diminuindo o tempo ocioso utilizado na criação de *checkpoints* sem aumentar o trabalho perdido.

Redundância Essa estratégia é amplamente utilizada em sistemas físicos nos quais não é possível realizar operações como *rollback* para um estado anterior e correto do sistema. Um exemplo clássico são replicações físicas de componentes, como sensores redundantes em veículos aéreos, que garantem a continuidade e confiabilidade da operação mesmo diante da falha de um dos sensores. Além disso, essa abordagem não se limita apenas ao nível físico. Também existem exemplos de redundância lógica, como o uso do código de Hamming. Esse código adiciona *bits* extras à informação transmitida, criando uma redundância que permite detectar e corrigir erros ocorridos durante o transporte dos dados. Dessa forma, mesmo que haja falhas na comunicação, o sistema pode identificar e recuperar a informação correta, aumentando a confiabilidade global do mesmo.

2.3.2 Estratégias de Checkpoint em Aplicações HPC

A técnica de *Checkpoint/Restart (CR)* é a abordagem mais consolidada e utilizada para prover tolerância a falhas em aplicações de longa duração, como as de HPC. A sua premissa fundamental é a criação de *checkpoints* que salvam o estado da aplicação em intervalos regulares. Em caso de falha, em vez de reiniciar a computação do zero, a aplicação retorna ao último *checkpoint* válido e retoma a execução, recalculando apenas o trabalho perdido desde aquele ponto. Conforme detalhado por Munhoz, Castro e Mendizabal (2022), no contexto de nuvem, essa abordagem é adaptada de maneiras distintas, principalmente através das estratégias de nível de sistema e de nível de aplicação. A seguir é apresentada uma breve descrição de cada estratégia:

Berkeley Lab Checkpoint/Restart (BLCR) O BLCR é uma biblioteca que implementa o *checkpointing* no nível do sistema operacional. Sua operação é orquestrada por um gerenciador de *job* externo, que dispara a criação de *checkpoints* de forma periódica ou preemptiva, conforme apresentado por Buyya, Broberg e Goscinski (2011). Quando uma falha ocorre, o gerenciador é responsável por provisionar novos nós para substituir os que falharam e reiniciar toda a aplicação a partir do último *checkpoint*, que fica armazenado em um sistema de arquivos persistente e compartilhado. A principal vantagem do BLCR é ser transparente para a aplicação, não exigindo modificações no código-fonte. Isso simplifica enormemente a migração de aplicações HPC legadas para a nuvem.

No entanto, suas desvantagens são significativas em termos de desempenho. A abordagem tem uma sobrecarga substancial, pois precisa salvar o estado completo

do processo em disco. Segundo Munhoz, Castro e Mendizabal (2022), o BLCR apresenta um desempenho inferior e não escala bem com um número crescente de falhas, podendo, dentro de um contexto de comparação de Instâncias Efêmeras e permanentes, em certos cenários, tornar a execução mais custosa do que o uso de instâncias *On-Demand*.

Distributed MultiThreaded CheckPointing (DMTCP) O DMTCP é uma ferramenta de *checkpointing* que também opera no nível de sistema, Ansel, Arya e Cooperman (2009), de forma similar ao BLCR, porém apresenta-se como uma solução mais moderna e flexível. Diferente do BLCR, que possui limitações com versões de kernel e redes específicas, o DMTCP suporta configurações de sistema mais recentes, redes de alta velocidade como *InfiniBand*, e é compatível com as principais distribuições MPI.

Essa versatilidade o torna uma alternativa transparente mais robusta. Contudo, por ser uma solução baseada em arquivos, compartilha a desvantagem fundamental de outras ferramentas de nível de sistema: a potencial sobrecarga de E/S e a necessidade de grande espaço de armazenamento para os *checkpoints*.

Scalable Checkpoint/Restart (SCR) O *Scalable Checkpoint/Restart* (SCR) é uma biblioteca projetada para mitigar o principal gargalo das estratégias de *checkpointing* tradicionais: o alto custo de *E/S* para o sistema de arquivos paralelo. Sua principal inovação, conforme apresentado por Moody et al. (2010), é a implementação de um sistema de *checkpointing* multinível.

Em vez de escrever cada *checkpoint* diretamente no armazenamento persistente e lento, o SCR utiliza uma hierarquia de armazenamento, priorizando a escrita em mídias locais e rápidas, como a memória RAM ou discos SSD dos próprios nós computacionais. Essa abordagem reduz drasticamente a sobrecarga, pois a maioria das falhas pode ser recuperada a partir desses *checkpoints* locais, que são ordens de magnitude mais rápidos. A transferência para o sistema de arquivos paralelo, mais lento, ocorre de forma menos frequente ou assíncrona. A principal desvantagem do SCR é que, ao contrário do BLCR e DMTCP, ele não é totalmente transparente e exige que o desenvolvedor modifique o código-fonte da aplicação para inserir chamadas à sua API, tornando o processo de integração mais complexo.

User-Level Failure Mitigation (ULFM) O ULFM é uma extensão da especificação MPI que provê uma API para que a própria aplicação gerencie as falhas. Assim como o SCR, a lógica de tolerância a falhas é implementada diretamente no código-fonte. Uma estratégia comum é o armazenamento do *checkpoint* em memória, onde cada processo mantém uma cópia do estado necessário para a recuperação. Em caso de falha, a aplicação utiliza rotinas do ULFM, como `MPIX_Comm_shrink`, para reconfigurar o comunicador, removendo os processos falhos e continuando a execução a partir do último estado salvo em memória, sem a necessidade de reiniciar todo o trabalho.

A grande vantagem do ULFM é o desempenho superior e o menor custo de infraestrutura, resultado dos *checkpoints* em memória e da recuperação ágil, que leva milissegundos. Ele escala bem mesmo com muitas falhas e oferece flexibilidade para o desenvolvedor. A sua principal desvantagem, contudo, é a necessidade de modificações consideráveis no código da aplicação, tornando o processo de migração mais complexo e custoso. Adicionalmente, o *checkpoint* em memória aumenta significativamente o consumo de RAM, e a implementação do ULFM no *OpenMPI* ainda é considerada experimental.

2.4 O MÉTODO LATTICE BOLTZMANN (LBM)

Segundo Chen e Doolen (1998), o Lattice Boltzmann (LBM) é uma técnica computacional poderosa e cada vez mais popular para simular o comportamento de fluidos. Em vez de resolver as equações tradicionais da fluidodinâmica, que descrevem o fluido como um material contínuo, o LBM adota uma perspectiva diferente. O método considera o fluido não como um todo, mas como uma vasta coleção de “pacotes” de partículas fictícias que se movem e colidem em uma grade de N dimensões. Conforme os autores, sua grande vantagem reside na facilidade com que consegue lidar com cenários complexos, como o fluxo de sangue em artérias, o movimento de petróleo através de rochas porosas ou o escoamento de ar ao redor de geometrias intrincadas, como a de um avião.

O LBM foi desenvolvido para ser uma alternativa eficiente para simular fenômenos que são um desafio para métodos convencionais. Isso inclui desde a turbulência de fluidos e a mistura de líquidos diferentes até a interação de partículas suspensas em um líquido, como a areia na água. A ideia central do método é permitir a operação em um nível mesoscópico, um meio-termo entre o mundo microscópico das moléculas individuais e o mundo macroscópico do fluxo visível. Em vez de rastrear cada partícula — o que seria computacionalmente custoso — o LBM simplifica o problema. Em cada ponto da sua grade, ele rastreia a probabilidade de encontrar pacotes de partículas se movendo em um pequeno conjunto de direções predefinidas. Ao fazer isso, ele captura a física essencial do movimento e das interações das partículas de uma forma estatisticamente precisa, mas muito mais simples de calcular.

O algoritmo do LBM funciona em um ciclo contínuo de dois passos simples que se repetem em toda a grade, um processo que Calore et al. (2017) descreve como altamente paralelizável:

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

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 .

A aplicação permite configurar a quantidade de nós participantes da simulação. Essa informação é utilizada para particionar a matriz global, atribuindo a cada nó uma matriz local que será manipulada individualmente durante as iterações.

Além disso, o funcionamento em iterações da aplicação permite a verificação da integridade da execução. Essa consistência pode ser confirmada por meio de operações coletivas, como a `MPI_Gather`, que realiza a soma da massa total da malha, assegurando que a principal propriedade da simulação — a conservação da massa — seja mantida ao longo da execução.

2.4.2 Empacotamento dos dados

Antes que qualquer comunicação possa ocorrer, cada *rank* precisa copiar para *buffers* auxiliares os valores das suas bordas que serão enviados aos vizinhos. Essa etapa é fundamental no LBM, pois o método depende do envio das populações de partículas das células de fronteira para preservar a correta propagação das distribuições.

O código abaixo ilustra essa etapa de empacotamento, utilizando funções específicas como `pack_top_border`, `pack_bot_border`, `pack_right_border` e `pack_left_border`. Essas funções percorrem a estrutura de dados local (`f2_soa_h`) e copiam os valores das células na periferia para estruturas lineares, que serão posteriormente transmitidas via MPI. Esse processo reduz a complexidade da comunicação e garante que apenas os valores minimamente necessários sejam transmitidos.

Listing 2.1 – Empacotamento dos dados das bordas antes da comunicação

```
data_t *topSndBuf = NULL, *topRxBuf = NULL, *botSndBuf = NULL, *botRxBuf = NULL;
data_t *leftSndBuf = NULL, *leftRxBuf = NULL, *rightSndBuf = NULL, *rightRxBuf = NULL;

// Para quem tem um vizinho no topo
if (mpi_rank_y < (n_mpi_y - 1))
{
    pack_top_border(topSndBuf, f2_soa_h, Q1);
}

// Para quem tem um vizinho embaixo
if (mpi_rank_y > 0)
{
    pack_bot_border(botSndBuf, f2_soa_h, Q2);
}

pack_right_border(leftSndBuf, f2_soa_h, Q1);
pack_left_border(rightSndBuf, f2_soa_h, Q2);
```

Essa fase conclui a preparação dos dados que serão enviados aos vizinhos. A seguir, inicia-se a etapa de comunicação assíncrona entre os *ranks*.

2.4.3 Comunicação assíncrona

Após empacotar os dados, a aplicação inicia a comunicação entre os *ranks* utilizando chamadas não bloqueantes do MPI. O uso de *MPI_Irecv* permite registrar previamente múltiplas requisições de recepção, que serão completadas de forma assíncrona à medida que os dados forem chegando. Em paralelo, *MPI_Isend* transmite os *buffers* empacotados aos vizinhos.

Esse padrão é característico de aplicações de HPC que buscam sobrepor comunicação e computação. Como o LBM permite operar no *bulk* (parte interna da matriz) independentemente das bordas, é possível iniciar a computação interna enquanto as mensagens ainda estão em trânsito. Ao final, as chamadas *MPI_Wait* garantem que as transferências foram concluídas antes do uso dos dados recebidos, evitando condições de corrida.

Listing 2.2 – Comunicação não bloqueante entre ranks e espera pela conclusão

```
// MPI Requests
MPI_Request reqRxTopHalo, reqTxTopBorder, reqRxBotHalo, reqTxBotBorder;
MPI_Request reqRxLeftHalo, reqTxRightBorder, reqRxRightHalo, reqTxLeftHalo;

// MPI-Comm Right-Border
MPI_Irecv(leftRxBuf, (15 + 8 + 3) * NY, MPI_DOUBLE,
mpi_peer_left, tagLEFT, MPI_COMM_WORLD, &reqRxLeftHalo);
MPI_Isend(leftSndBuf, (15 + 8 + 3) * NY, MPI_DOUBLE,
mpi_peer_right, tagLEFT, MPI_COMM_WORLD, &reqTxRightBorder);

// MPI-Comm Left-Border
MPI_Irecv(rightRxBuf, (15 + 8 + 3) * NY, MPI_DOUBLE,
mpi_peer_right, tagRIGHT, MPI_COMM_WORLD, &reqRxRightHalo);
MPI_Isend(rightSndBuf, (15 + 8 + 3) * NY, MPI_DOUBLE,
mpi_peer_left, tagRIGHT, MPI_COMM_WORLD, &reqTxLeftHalo);

// Espera pela comunicacao da borda direita
MPI_Wait(&reqTxRightBorder, MPI_STATUS_IGNORE);
MPI_Wait(&reqRxLeftHalo, MPI_STATUS_IGNORE);
unpack_left_halo(f2_soa_h, leftRxBuf, Q1);

// Espera pela comunicacao da borda esquerda
MPI_Wait(&reqTxLeftHalo, MPI_STATUS_IGNORE);
MPI_Wait(&reqRxRightHalo, MPI_STATUS_IGNORE);
```

Essa etapa garante que todas as trocas de halos foram concluídas e que os dados já estão disponíveis para uso no modelo físico da simulação. Com as fronteiras atualizadas, a aplicação pode prosseguir para as fases físicas do LBM.

2.4.4 Propagação, Colisão e Sincronização Global

A terceira etapa ocorre após a chegada dos dados das bordas e compreende as operações físicas do LBM: propagação, colisão, aplicação das condições de contorno e, finalmente, o sincronismo global entre os processos.

A propagação transfere distribuições entre células vizinhas, enquanto a colisão ajusta as distribuições para que tendam ao estado de equilíbrio conforme o modelo de Boltzmann discretizado. Em regiões de fronteira, entram em ação funções específicas como *bcTop* e *bcBottom*, responsáveis por aplicar condições de contorno físicas.

Logo após essas operações, o código realiza o *swap* entre os ponteiros das matrizes de estados *f1* e *f2*, técnica comum em LBM para evitar cópias de memória. Por fim, é chamada uma barreira global com *MPI_Barrier*, garantindo que todos os *ranks* concluam a iteração antes de avançar, mantendo a integridade temporal do método.

Listing 2.3 – Propagação, colisão, condições de contorno e sincronização

```
// Bottom Border
if (mpi_rank_y == 0)
{
    propagate(f1_soa_h, f2_soa_h, (HX), (HX + LSIZEEX),
              (HY), (HY + hy + 1), Q4);
    bcBottom(f1_soa_h, f2_soa_h, param, (HX), (HX + LSIZEEX), Q4);

    COLLIDE(f1_soa_h, f1_soa_h, (HX), (HX + LSIZEEX),
            (HY), (HY + hy + 1), yoff, Q4);
}
else
{
    PROPAGATECOLLIDE(f1_soa_h, f2_soa_h, (HX), (HX + LSIZEEX),
                    (HY), (HY + hy + 1), yoff, Q4);
}

// Top Border
if (mpi_rank_y == (n_mpi_y - 1))
{
    propagate(f1_soa_h, f2_soa_h, (HX), (HX + LSIZEEX),
              (HY + LSIZEY - hy - 1), (HY + LSIZEY), Q5);
    bcTop(f1_soa_h, f2_soa_h, param, (HX), (HX + LSIZEEX), Q5);

    COLLIDE(f1_soa_h, f1_soa_h, (HX), (HX + LSIZEEX),
            (HY + LSIZEY - hy - 1), (HY + LSIZEY), yoff, Q5);
}
else
{
    PROPAGATECOLLIDE(f1_soa_h, f2_soa_h, (HX), (HX + LSIZEEX),
                    (HY + LSIZEY - hy - 1), (HY + LSIZEY), yoff, Q5);
}

// Swap pointers
f_tmp_h = f1_soa_h;
f1_soa_h = f2_soa_h;
f2_soa_h = f_tmp_h;

// Sincronizacao global
MPI_Barrier(MPI_COMM_WORLD);

interactionEndTime = MPI_Wtime();
interactionExecTime = interactionEndTime - interactionStartTime;
```

Essa etapa finaliza a iteração do método LBM. Após a barreira, todos os processos

estão novamente alinhados temporalmente e podem prosseguir para a próxima iteração do *loop* principal.

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, facilitada por softwares de gerenciamento de instâncias, e pela significativa redução de custos proporcionada pelas Instâncias Efêmeras, popularmente conhecidas como instâncias *Spot*. Contudo, a natureza volátil dessas instâncias impõe um desafio fundamental: **garantir a continuidade da execução das aplicações após revogações de instâncias 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 CR.

As abordagens tradicionais, como o BLCR apresentado por Hargrove e Duell (2006), que opera no nível de sistema operacional possui a grande vantagem de ser transparente para a aplicação, não necessitando de alterações no código-fonte. Entretanto, conforme apontado por Munhoz, Castro e Mendizabal (2022), essa abordagem pode se tornar ineficiente e onerosa. A ineficiência se manifesta em cenários com alta frequência de falhas, onde o alto custo para criar *checkpoints*, sob uma perspectiva externa à aplicação, pode anular os benefícios econômicos das instâncias *Spot*, resultando em menor desempenho e maior custo total de aquisição de VMs.

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

Paralelamente à otimização do mecanismo, a otimização da estratégia de temporização dos *checkpoints* também é explorada. O trabalho de Amoon et al. (2019), por exemplo, aborda a ineficiência dos intervalos fixos e propõe o *Dynamic Change Checkpoint (DCC)*, uma abordagem reativa onde o intervalo entre *checkpoints* é ajustado dinamicamente com base na taxa de falhas observada no ambiente. Os resultados experimentais confirmaram que essa estratégia adaptativa melhora significativamente o tempo de resposta, o consumo de energia e o custo monetário, reforçando que a eficiência da tolerância a falhas na nuvem depende tanto do mecanismo de recuperação quanto da inteligência de sua aplicação.

Em sua revisão sobre técnicas de tolerância a falhas, (Brum et al., 2023) analisa 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 *overhead* 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.

Nesse mesmo sentido, para resolver o gargalo de *E/S* do *checkpointing* tradicional em sistemas HPC, o trabalho de Moody et al. (2010) propõe a biblioteca SCR, baseada em um sistema de *checkpointing* multinível. Essa 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. O estudo demonstra que a maioria das falhas pode ser recuperada localmente, o que aumenta drasticamente a eficiência do sistema e reduz a carga na rede. Embora exija modificações no código da aplicação via 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.

3.1 CONTRIBUIÇÕES

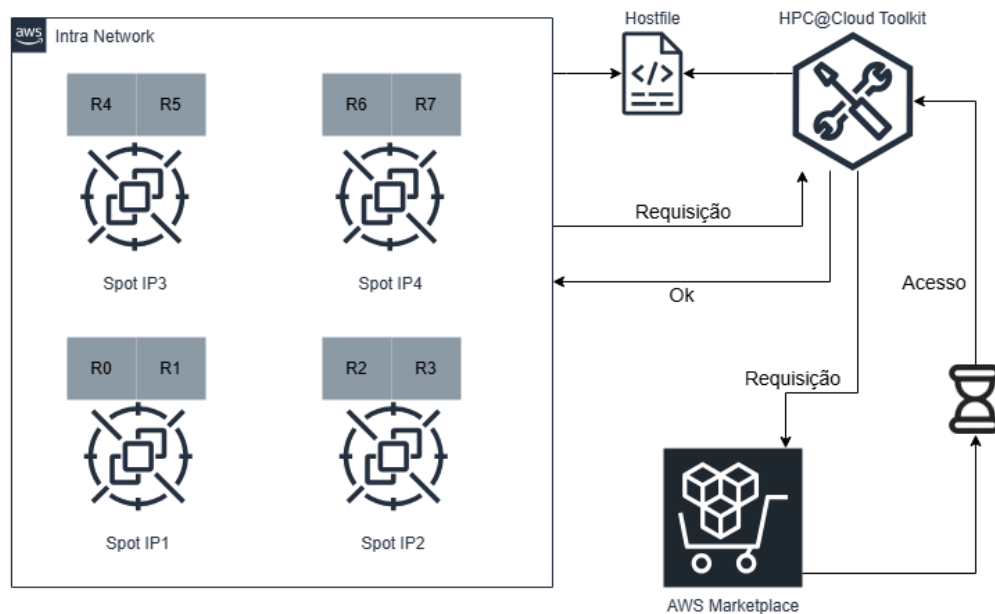
Este trabalho se diferencia do estado da arte ao investigar o comportamento do custo de desenvolvimento e execução no método Lattice Boltzmann (LBM), considerando distintas estratégias de tolerância a falhas. A pesquisa analisou em profundidade as versões preemptiva e não preemptiva da aplicação durante o processo de recuperação. Além disso, foram implementadas duas abordagens de *checkpointing* — em disco e em memória — sendo que esta última permite ajustar dinamicamente o número de falhas simultâneas suportadas e recuperáveis. O objetivo é otimizar e potencialmente reduzir o custo de criação dos *checkpoints*, oferecendo uma análise mais abrangente da viabilidade e eficiência do LBM em ambientes de nuvem voláteis.

4 PROPOSTA DE PESQUISA

Considerando o cenário atual, este trabalho tem como objetivo avaliar a viabilidade técnica e econômica da migração e execução de aplicações HPC desenvolvidas originalmente para serem executadas em clusters *On-Demand* para Instâncias Efêmeras. Para isso, foi alterado o código-fonte do *benchmark* LBM da suíte SPEChpc® 2021, que já estava implementado em *OpenMPI* e OpenMP, para diferentes versões tolerantes a falhas, utilizando a extensão ULFM.

A execução e o gerenciamento das instâncias *Spot* foram realizados por meio da ferramenta *HPC@Cloud Toolkit*, que facilitou a orquestração dos recursos computacionais na infraestrutura da AWS. O *HPC@Cloud Toolkit* é uma ferramenta de software de código aberto e agnóstico a provedores, projetada para simplificar e democratizar o uso da Computação em Nuvem para cargas de trabalho de HPC. Proposto por Munhoz e Castro (2023a), ele é capaz de abstrair a complexidade inerente à migração, configuração, execução e monitoramento de aplicações HPC em nuvens públicas. Ao facilitar o gerenciamento de múltiplas instâncias, a ferramenta viabiliza o acesso a infraestruturas massivamente paralelas para grupos de pesquisa e empresas que não possuem capacidade. O *toolkit* integra nativamente o suporte a contêineres *Singularity*, o que resolve o problema de portabilidade ao permitir que aplicações e suas dependências sejam empacotadas em um ambiente único e reproduzível, que pode ser executado em qualquer nuvem sem modificações.

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



Fonte: Autor

Conforme ilustrado na Figura 2, cada processo — ou *rank* — foi alocado em ins-

tâncias *Spot*, adquiridas por meio do *HPC@Cloud Toolkit* no Marketplace de instâncias *Spot* da AWS. Além disso, uma mesma instância pôde ser compartilhada por múltiplos processos em execução simultânea. Essa configuração, porém, exigiu atenção às características de hardware da instância, como número de núcleos e capacidade de memória, de modo a garantir o desempenho adequado e evitar sobrecarga nos recursos computacionais.

Além disso, a Figura 2 representa a requisição feita comunicação *Requisição* feita ao *HPC@Cloud Toolkit*, a qual é repassada para a AWS, assim como o tempo indeterminado de resposta por parte da provedora, representado pela ampulheta na figura. Ainda sobre esta figura, o hostfile

Durante os experimentos, as falhas foram simuladas manualmente por meio da exclusão de instâncias *Spot*, o que causou a interrupção abrupta dos *ranks* que estavam em execução nessas instâncias. Esse procedimento forçou os demais *ranks* a realizar a recuperação da falha e a prosseguir com a execução.

Foram conduzidos diversos experimentos sob diferentes cenários de execução, abrangendo versões com e sem tolerância a falhas. Todos os testes foram realizados em instâncias *on-demand* da AWS, a fim de garantir estabilidade e repetibilidade durante a coleta de resultados. Entretanto, no cálculo do custo total, as versões tolerantes a falhas consideraram o valor equivalente das instâncias *spot*, refletindo de forma mais realista o contexto de uso proposto em ambientes voláteis de nuvem. Essa abordagem permitiu uma comparação justa entre as estratégias, preservando a consistência experimental sem comprometer a análise econômica.

Além disso, foi avaliada a viabilidade técnica da refatoração do código-fonte original para adaptar a aplicação à metodologia ULFM, com foco na complexidade e no esforço de desenvolvimento necessários para tornar a aplicação tolerante a falhas.

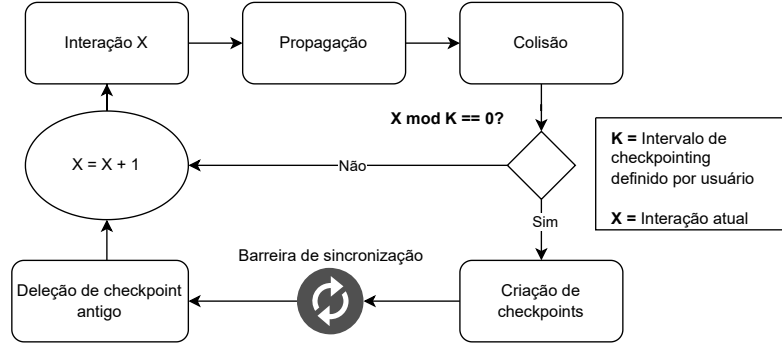
4.1 CHECKPOINTS

Conforme especificado pela extensão ULFM, a responsabilidade pela implementação da estratégia de tolerância a falhas é delegada ao desenvolvedor, incluindo a criação e o gerenciamento dos *checkpoints*. A frequência de criação desses *checkpoints* é parametrizável, podendo ser ajustada pelo usuário para ocorrer, por exemplo, a cada X iterações.

Os processos de checkpointing neste trabalho utilizam mecanismos de sincronização nativos do MPI, como o `MPI_Barrier`, que garantem que o processamento seja concluído com sucesso. Assim, caso ocorra uma falha durante o processo de *checkpoint*, todos os *ranks* reagem e iniciam o procedimento de recuperação. Conforme ilustrado na Figura 3, os algoritmos asseguram a liberação da memória associada ao *checkpoint* anterior somente após a confirmação de que o novo processo de salvamento foi finalizado com êxito em todos os *ranks*.

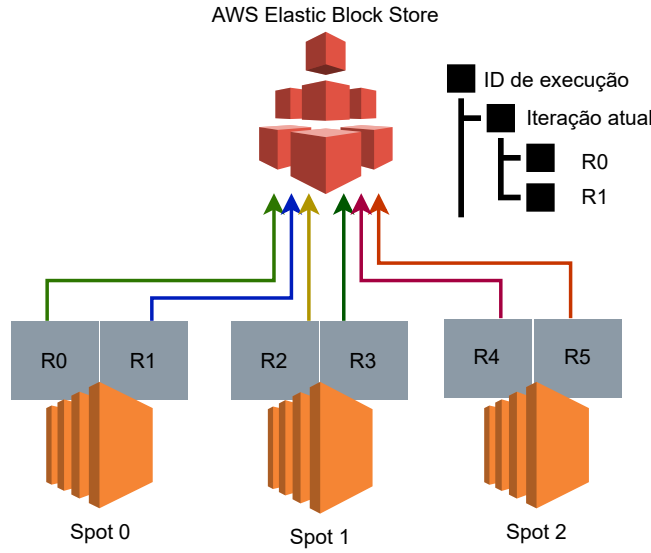
Este trabalho propôs duas abordagens distintas de *checkpoint*:

Figura 3 – Fluxograma de execução da aplicação LBM com *checkpoints* periódicos.



Fonte: Autor

Figura 4 – Fluxo de dados no algoritmo de *checkpointing* em disco (Amazon EBS)



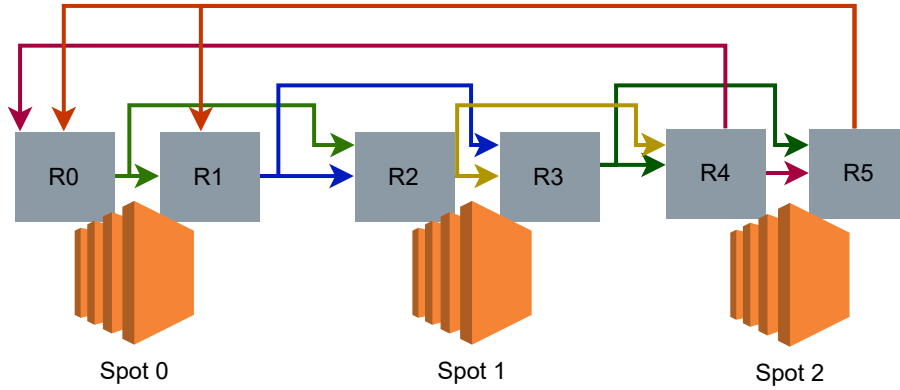
Fonte: Autor

Em disco Essa abordagem permite que a execução seja recuperada mesmo em casos de falha total dos *ranks*, pois o estado da aplicação é armazenado em memória não volátil. A Figura 4 ilustra como esse mecanismo foi implementado neste trabalho. Nessa configuração, todos os *ranks* salvam seus dados em um volume EBS compartilhado entre todas as instâncias.

Em memória Utilizando 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, ou seja, $\{R_{(i-1) \bmod n}, \dots, R_{(i-x) \bmod n}\}$. O valor x , definido pelo usuário, indica o número de *ranks* cujos dados são replicados, representando assim a quantidade de falhas simultâneas que a aplicação é capaz de suportar. A Figura 5 mostra o fluxo de dados para $x = 2$ e $n = 6$.

Como ilustrado na Figura 5, em uma configuração tolerante a duas falhas simultâneas, cada processo envia seu estado atual (*checkpoint*) aos dois próximos *ranks*, seguindo uma organização sequencial de 0 a N . No exemplo, o processo R_0 envia seu *checkpoint* para R_1 e R_2 ; R_1 envia para R_2 e R_3 ; e assim sucessivamente, até que o

Figura 5 – Fluxo de dados no algoritmo de *checkpointing* em memória para 6 *ranks* em 3 instâncias *spot*



Fonte: Autor

último *rank* R_5 envie seus dados de *checkpoint* de volta para R_0 e R_1 , completando um ciclo contínuo.

Essa abordagem oferece ao usuário a possibilidade de ajustar o nível de resiliência da simulação conforme o perfil de falhas esperado no ambiente de execução. Dessa forma, é possível equilibrar a robustez da aplicação e o custo computacional, otimizando o tempo de criação dos *checkpoints* e minimizando o *overhead* de comunicação de acordo com o nível de tolerância a falhas desejado.

4.2 IDENTIFICAÇÃO DE FALHAS

A identificação de falhas é um recurso nativo da extensão ULFM, que permite detectar automaticamente falhas de processos durante a execução paralela. No MPI, os processos trocam mensagens dentro de um comunicador, que é uma entidade lógica responsável por organizar e agrupar os *ranks* participantes da aplicação. Sempre que ocorre uma falha em um dos *ranks*, os *ranks* que mantêm comunicação direta com o *rank* falho são imediatamente notificados pela ULFM.

Neste trabalho, a detecção de falhas foi configurada para redirecionar a execução para a função `handle_exception` assim que uma falha é detectada. Nessa função, os processos executam operações coletivas específicas da ULFM, como `MPiX_Comm_revoke` e `MPiX_Comm_shrink`. Essas operações permitem que todos os processos sobreviventes abandonem o comunicador atual e formem um novo comunicador que exclui os processos falhos. No escopo desse novo comunicador, os processos devem então coordenar-se para realizar a recuperação da falha e retomar a execução da simulação.

4.3 RECUPERAÇÃO DE FALHAS

Neste trabalho, considera-se que a recuperação de falhas é bem-sucedida quando o trabalho originalmente atribuído a um processo que falhou pode ser continuado por outro

processo ativo. Isso inclui a retomada das trocas de partículas entre vizinhos, o cálculo das colisões e a participação nas operações coletivas necessárias para a continuidade da simulação

Para que a execução possa prosseguir, é necessário recriar os *ranks* perdidos e integrá-los a um novo comunicador, garantindo que eles recuperem o estado anterior de execução. Dessa forma, esses novos processos podem retomar o processamento das regiões do domínio anteriormente atribuídas aos processos falhos.

O ULFM possibilita uma recuperação **transparente**, em que os processos sobreviventes não são finalizados após uma falha e podem se reorganizar automaticamente, sem a necessidade de reiniciar a aplicação com o comando `mpirun`. Em contrapartida, a maioria das aplicações utilizam abordagens **não transparentes**, nas quais a execução é interrompida e deve ser reiniciada manualmente, com base em um *checkpoint* previamente salvo.

Quanto à continuidade da execução durante a recuperação neste trabalho, adotam-se duas classificações:

Recuperação preemptiva: A execução da simulação é suspensa até que uma nova instância *spot* esteja disponível para substituir os *ranks* perdidos.

Recuperação não-preemptiva: A simulação continua nas instâncias sobreviventes, ainda que com desempenho reduzido, sem depender da alocação imediata de novos recursos pela provedora.

Embora a versão mais recente do *OpenMPI* (5.0.8) já integre suporte nativo ao ULFM, **a implementação atual ainda não oferece tolerância a falhas em nível de nó de forma plenamente funcional**. Durante os experimentos, observou-se que, quando ocorre uma falha no ambiente de execução paralelo remoto — o *PRTE* (*Parallel Runtime Environment*) —, todos os nós participantes são encerrados abruptamente, independentemente das configurações de recuperação adotadas pelo ULFM. Assim, a falha de um único nó físico ou virtual leva à terminação completa do sistema distribuído, inviabilizando a recuperação transparente.

Em contrapartida, verificou-se que o mecanismo de tolerância a falhas comporta-se corretamente apenas quando o erro é restrito ao nível de processo, permitindo que os processos sobreviventes detectem a falha, reconstruam o comunicador e prossigam com a execução, conforme previsto pelas especificações do ULFM. No entanto, considerando que este trabalho aborda falhas em nível de nó — devido às instâncias efêmeras —, tal limitação torna a versão *open-source* atual do *OpenMPI* inadequada para o uso pleno do ULFM nos cenários propostos.

Consequentemente, o mecanismo de *checkpoint* em memória implementado, embora funcional em falhas simuladas, não pode ser utilizado de forma eficaz diante de falhas reais de nó. Isso ocorre porque, quando o *runtime* falha, todos os processos, inclusive os responsáveis por armazenar o estado em memória volátil, são encerrados. Dessa forma,

a aplicação perde completamente o estado de execução e torna-se incapaz de retomar o processamento. Para que a aplicação seja realmente tolerante a falhas reais, é indispensável empregar *checkpoints* persistentes em disco. Essa estratégia assegura que o estado global da simulação possa ser restaurado mesmo após a perda total de um nó, garantindo continuidade e efetividade na recuperação. Portanto, somente a solução de *checkpoint* em disco foi adotada para as versões Preemptiva e Adaptativa.

4.4 VERSÕES PROPOSTAS

Este trabalho propõe três versões distintas de implementação de tolerância a falhas, cada uma explorando diferentes estratégias de recuperação e redistribuição de carga após a ocorrência de falhas em ambientes de execução distribuída. As versões apresentadas a seguir — *Preemptiva*, *Não Preemptiva (Oversubscribe)* e *Adaptativa* — buscam avaliar o impacto de cada abordagem sobre o tempo de recuperação, a imprevisibilidade da provedora e o desempenho da aplicação.

O processo de adaptação de uma aplicação MPI legada para utilizar a extensão ULFM revelou-se significativamente mais complexo do que o esperado. A natureza da extensão — desenhada de forma intencionalmente aberta para se adequar a diferentes padrões — exige que o desenvolvedor compreenda profundamente o fluxo de dados e de controle da aplicação, uma vez que as decisões de recuperação, reconfiguração e redistribuição dos dados recaem completamente sobre a aplicação, e não sobre a biblioteca.

Dessa forma, não é suficiente realizar modificações pontuais: é necessário entender como os dados circulam entre os processos, em que momento e com que frequência *checkpoints* devem ser realizados, quais informações precisam ser salvas durante o processo de *checkpointing* e como garantir a consistência entre os processos sobreviventes e os reiniciados. Além disso, a própria estrutura de chamadas da ULFM pode levar a redundâncias funcionais — por exemplo, a necessidade de criar novos comunicadores, testar falhas explicitamente, reconstruir grupos e, por fim, fundir esses grupos — o que torna o código menos intuitivo para novos usuários da biblioteca.

Outro fator limitante identificado durante o desenvolvimento foi a escassez de materiais de apoio. Como a extensão ULFM ainda está em processo de padronização e não é amplamente adotada, há poucos exemplos de código disponíveis, seja em artigos, repositórios ou no fórum oficial do projeto. Essa ausência de referências impõe uma curva de aprendizado íngreme para desenvolvedores que desejam explorar as funcionalidades da biblioteca.

4.4.1 C/R Preemptivo

Na abordagem preemptiva, uma vez restabelecida a comunicação entre os processos sobreviventes por meio da criação de um novo comunicador, a aplicação solicita

ao *HPC@Cloud Toolkit* que a AWS disponibilize uma nova instância para hospedar os processos substitutos correspondentes aos que falharam.

Durante esse processo, a execução da aplicação permanece bloqueada até que o *HPC@Cloud Toolkit* confirme a disponibilidade da nova instância. Após a confirmação, o *HPC@Cloud Toolkit* procede com a atualização do arquivo *hostfile* — que contém a lista de instâncias ativas e seus respectivos endereços IP —, incorporando as informações da instância recém-criada.

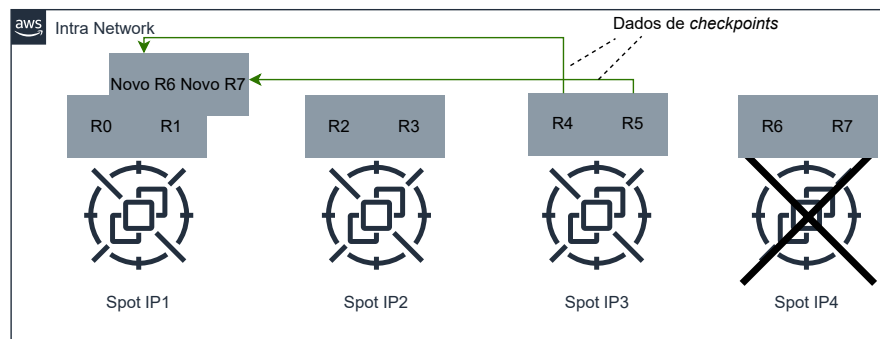
A partir dessa atualização, a aplicação acessa o novo *hostfile* e o utiliza como parâmetro na diretiva `MPI_Comm_Spawn`, responsável por iniciar os novos processos nos nós corretos, restabelecendo a configuração original de distribuição da carga de trabalho entre as instâncias.

A principal vantagem dessa abordagem é que o balanceamento de carga é completamente restaurado, mantendo-se idêntico ao existente antes da falha. No entanto, essa estratégia apresenta uma limitação significativa: o tempo necessário para o provisionamento de uma nova instância pela AWS é incerto, tornando imprevisível o momento exato em que a execução poderá ser retomada. Essa imprevisibilidade pode comprometer o desempenho global da aplicação em cenários de alta frequência de falhas.

4.4.2 C/R Não Preemptivo: *Oversubscribe*

No contexto do ULFM, o termo *oversubscribing* refere-se à prática de alocar múltiplos processos MPI em um mesmo nó de computação, compartilhando os recursos disponíveis entre eles. Essa estratégia é habilitada no momento da execução da aplicação por meio do parâmetro `-oversubscribe` no comando `mpirun`, o que permite que novos processos sejam criados em nós já em uso.

Figura 6 – Operação de *oversubscribing* de novos processos em nós pré-existent após falha.



Fonte: Autor

Conforme ilustrado na Figura 6, após a falha do nó identificado como *IP4*, os *ranks* R_6 e R_7 são reinicializados em outro nó ativo, como o *IP1*. Nesse cenário, o MPI gerencia automaticamente a redistribuição dos novos *ranks* entre os nós sobreviventes, sem a necessidade de intervenção manual ou atualização do arquivo *hostfile*.

A principal vantagem dessa abordagem é a agilidade na recuperação: não é necessário aguardar o provisionamento de uma nova instância nem realizar reconfigurações adicionais, o que possibilita a retomada imediata da execução. Entretanto, essa estratégia acarreta degradação de desempenho, uma vez que os recursos computacionais de um nó passam a ser compartilhados entre um número maior de processos. Dessa forma, há aumento na contenção de CPU, memória e banda de rede, afetando o tempo total de execução da simulação.

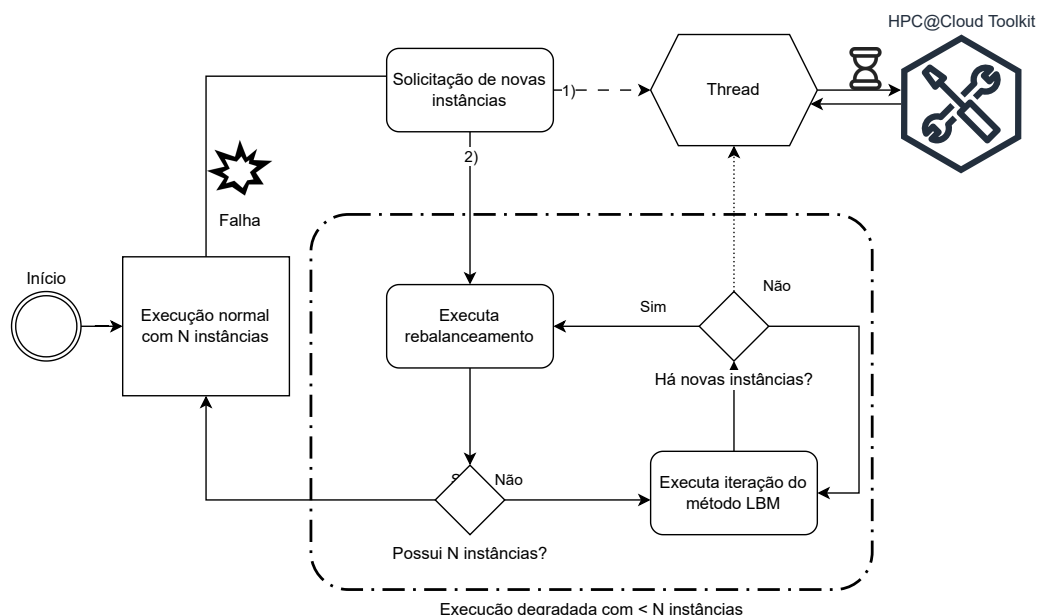
Além disso, em situações nas quais a indisponibilidade da instância perdida seja temporária, a aplicação continuará operando com desempenho reduzido até o término da execução, mesmo que novos nós possam ser alocados posteriormente. Assim, embora ofereça recuperação imediata, essa versão não restabelece o desempenho original da aplicação.

4.4.3 C/R Adaptativo

A versão adaptativa busca combinar as vantagens das abordagens preemptiva e não preemptiva, reduzindo o tempo de inatividade após falhas, ao mesmo tempo em que permite a restauração completa do desempenho da aplicação assim que novos recursos computacionais se tornem disponíveis. Trata-se, portanto, de um modelo híbrido que prioriza continuidade, resiliência e eficiência global.

Conforme ilustrado na Figura 7, logo após a restauração da comunicação entre os processos sobreviventes, a aplicação solicita ao *HPC@Cloud Toolkit* o provisionamento de novas instâncias na AWS para substituir aquelas que falharam. Entretanto, de maneira semelhante à abordagem não preemptiva, a execução não é interrompida: a aplicação

Figura 7 – Fluxo de recuperação para versão adaptativa



Fonte: Autor

continua o processamento usando a diretiva `-oversubscribe`, realocando os processos substitutos nos nós remanescentes. Esse procedimento garante continuidade imediata da execução, ainda que com desempenho temporariamente degradado devido à sobrecarga nos recursos disponíveis.

Simultaneamente, uma *thread* auxiliar é iniciada de forma assíncrona. Essa *thread* é responsável por monitorar periodicamente, via *HPC@Cloud Toolkit*, o estado das instâncias solicitadas. Assim que a nova instância for completamente provisionada, a aplicação executa uma redistribuição dinâmica dos *ranks*, incluindo o nó recém-criado no comunicador e reequilibrando a carga de trabalho.

Essa estratégia reduz significativamente o tempo ocioso associado à espera por novos recursos, mantendo a aplicação ativa mesmo diante de falhas. Quando os novos nós finalmente são integrados ao sistema, o desempenho original é automaticamente restabelecido, sem intervenção manual e sem necessidade de reinicialização da simulação.

Adicionalmente, em cenários com múltiplas revogações de instâncias no início da execução, a abordagem adaptativa evita que o sistema permaneça indefinidamente em estado degradado. O sistema retorna ao equilíbrio assim que os recursos se tornam novamente disponíveis, garantindo um compromisso eficiente entre resiliência, performance e continuidade operacional.

No entanto, devido a limitações da biblioteca OpenMPI — discutidas em seção posterior —, a implementação prática desta abordagem utilizou *checkpoint* em disco. Para isso, foi necessário empregar um orquestrador externo responsável por conduzir o processo de recuperação e coordenar a reintegração da nova instância.

Após a aplicação ser lançada, caso ocorra uma falha, o orquestrador identifica o problema porque a chamada de sistema:

```
system(mpirun ...)
```

retorna um erro. A partir desse momento inicia-se formalmente o tempo de recuperação, como mostrado a seguir:

Listing 4.1 – Criação da *thread* responsável por solicitar uma nova instância no C/R Adaptativo pós-falha.

```
system(mpirun --hostfile hostfile --with-ft ulfm --map-by ppr:1:node -x OMP_NUM_THREADS=8
      -x OMP_PROC_BIND=true -np 4 ./main);

// O tempo de recuperacao começa AGORA, no momento em que o orquestrador detecta o erro.
// A Falha foi detectada
clock_gettime(CLOCK_REALTIME, &failure_time);

// ---- FASE 3: RECUPERACAO PARALELA E EXECUCAO OVERSUBSCRIBED ----
pid_t recovery_pid = fork();

if (recovery_pid == 0) {
Codigo do filho...
}
else {
Codigo do pai...
```

```
}
```

A chamada `fork()` cria duas execuções paralelas:

1. **Processo filho:** responsável por solicitar a nova instância à AWS.
2. **Processo pai:** responsável por reiniciar a aplicação em modo *oversubscribed* para manter a execução ativa enquanto a recuperação está em curso.

O processo filho entra em um laço de tentativas periódicas para solicitar o provisionamento da instância perdida. Assim que o nó é restaurado, o processo registra o evento e notifica o processo principal por meio de um arquivo local:

Listing 4.2 – Algoritmo da *thread* que solicita uma nova instância à AWS

```
while (1) {
    status = system("./toolkit cluster spawn --cluster-id CLUSTER_ID -y);
    if (WIFEXITED(status) && WEXITSTATUS(status) == 0) {
        printf([FILHO-RECUPERACAO]: No recuperado com sucesso!);

        clock_gettime(CLOCK_REALTIME, &node_restored_time);
        FILE* f = fopen(node_restored_timestamp.log, w);
        fclose(f);
        break;
    } else {
        sleep(5);
    }
    // Solicita que a execucao oversubscribed finalize criando um novo checkpoint
    FILE* f = fopen("./checkpoint_and_kill", "w");

    exit(0);
}
```

Quando o arquivo `checkpoint_and_kill` é criado, a aplicação em execução detecta o sinal e gera um novo *checkpoint*. Em seguida, ela encerra sua execução, permitindo que a migração para a nova instância possa ocorrer.

Enquanto o processo filho aguarda o provisionamento da instância, o processo pai relança a aplicação imediatamente utilizando `-oversubscribe`, garantindo continuidade da simulação:

Listing 4.3 – Algoritmo da *thread* que solicita uma nova instância à AWS

```
clock_gettime(CLOCK_REALTIME, &recovery_start_oversubscribe_time);

// Lanca a simulacao com oversubscribe, usando o novo hostfile.
// Esta chamada e bloqueante e so retornara quando o mpirun terminar.
system(mpirun --oversubscribe -map-by node --hostfile hostfiledegraded --display-map -x
    OMP_NUM_THREADS=8 -x OMP_PROC_BIND=true -np 4 ./main --should-create-checkpoint 0 --
    load-checkpoint);

printf([PAI-ORQUESTADOR]: Execucao oversubscribed concluida (sinal 'kill' recebido).\n);

waitpid(recovery_pid, NULL, 0);
```

```
clock_gettime(CLOCK_REALTIME, &relaunch_start_time);  
  
system(mpirun --hostfile hostfile --map-by ppr:1:node --with-ft ulfm -x OMP_NUM_THREADS=8  
-x OMP_PROC_BIND=true -np 4 ./main --load-checkpoint);
```

Após a finalização da execução degradada, o orquestrador relança a aplicação normalmente — desta vez utilizando:

- todos os nós ativos,
- o checkpoint recém-criado
- e sem a diretiva `-oversubscribe`.

Assim, o desempenho original da aplicação é restaurado.

5 RESULTADOS

Neste capítulo, são apresentadas as principais dificuldades encontradas durante o desenvolvimento da tolerância a falhas e os resultados dos experimentos que avaliam três abordagens de recuperação de falhas: (i) preemptiva com *checkpoint* em disco; (ii) não preemptiva com *checkpoint* em memória; (iii) e adaptativo com *checkpoint* em disco.

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*. Para o cálculo do valor total de cada simulação, foi feita a multiplicação do tempo total de simulação pelo valor específico de cada cenário.

A utilização de instâncias *On-Demand* para garantir maior estabilidade baseia-se na premissa fornecida pela própria AWS: os recursos e o desempenho das instâncias são equivalentes nas modalidades *Spot* e *On-Demand*. Dessa forma, é possível concluir que a adoção de instâncias *On-Demand* não compromete a qualidade nem a validade dos resultados apresentados neste estudo.

5.2 ORQUESTRADOR

Conforme discutido anteriormente, a integração entre a biblioteca OpenMPI e a extensão ULFM apresentou limitações que impediram o suporte nativo a falhas de nó em nossa implementação, forçando o encerramento dos processos mediante tais ocorrências.

No caso da estratégia Não Preemptiva, como não há o requisito de provisionar novas instâncias após uma falha, foi possível manter o tratamento do erro estritamente a nível de processo. Isso permitiu que todos os *ranks* identificassem a falha e se reestruturassem sem interromper a execução. Consequentemente, esta foi a única estratégia onde o *checkpoint* em memória se mostrou viável.

Por outro lado, as estratégias Preemptiva e Adaptativa exigem o provisionamento de novas instâncias. Esse requisito implica a revogação completa do nó falho, o que leva,

inevitavelmente, ao encerramento dos processos. Diante disso, para estas estratégias, adotou-se o *checkpoint* em disco e a implementação de um Orquestrador. Este componente atua como um *wrapper* desenvolvido em C, responsável por executar as diretivas *mpirun*, enviar requisições de provisionamento para a AWS e rebalancear a estrutura após a recuperação.

Dessa forma, a lógica de controle foi externalizada para o orquestrador, contornando as limitações técnicas e viabilizando a execução dos testes necessários para este estudo.

5.3 CENÁRIOS

Os cenários de teste foram planejados para permitir uma comparação equilibrada entre as estratégias de *checkpointing* e de recuperação de falhas propostas. As falhas foram deliberadamente induzidas durante a execução das simulações, por meio do encerramento forçado de uma das instâncias ativas.

Com o intuito de evitar que o elevado custo de escrita de *checkpoints* em disco interferisse de forma desproporcional na comparação dos resultados, foram definidos intervalos distintos de salvamento para cada abordagem. Para o *checkpoint* em disco, adotou-se um intervalo de 20 iterações, enquanto para o *checkpoint* em memória, estabeleceu-se um intervalo de 10 iterações. Essa diferença visa compensar a disparidade de desempenho entre os dois métodos, garantindo que a análise se concentre principalmente na eficiência das estratégias de recuperação e não nos custos intrínsecos de E/S .

As falhas foram induzidas de forma controlada. Na adaptativa, elas ocorrem ao atingir cerca de 25% do progresso, e a simulação continua com desempenho reduzido até que o novo nó seja provisionado e o sistema se reequilibre. Nas preemptivas, a falha também é injetada em torno de 25%, mas a aplicação permanece bloqueada até concluir a recuperação, tornando o instante da falha irrelevante para o tempo total. Já na não preemptiva, o ponto da falha é determinante, pois a execução segue degradada até o fim. Por isso, foram definidos três cenários com falhas em 25%, 50% e 75% do progresso, a fim de mensurar o impacto da sobrecarga em diferentes estágios da execução.

As análises foram conduzidas sob os seguintes cenários:

Linha de base (sem falhas) [*Baseline.*] Execução integral da simulação sem injeção de falhas, utilizada como referência para mensuração de tempo e desempenho.

Abordagem preemptiva [*Preemp.*] Em caso de falha, a aplicação solicita, via *HPC@Cloud Toolkit*, o provisionamento de uma nova instância pela AWS e permanece bloqueada até que o nó substituto esteja disponível. A execução é então retomada a partir do último *checkpoint* persistido no EBS. Essa abordagem prioriza a restauração completa do estado anterior, ao custo de maior tempo de recuperação.

Abordagem não preemptiva [*NP25, NP50, NP75.*] Os *ranks* falhos são redistribuídos em nós ativos, ativando o modo *oversubscribe*. As falhas são injetadas aos 25%, 50% e 75% do progresso para avaliar o impacto da degradação de desempenho. Essa abordagem utiliza *checkpoints* em memória.

Abordagem adaptativa [*Adapt.*] Após a falha, a aplicação solicita uma nova instância à AWS e continua executando de forma degradada até sua disponibilização. Quando o nó é incorporado, ocorre o reequilíbrio automático dos *ranks*, restaurando o desempenho inicial. Utiliza *checkpoints* em disco, favorecendo recuperação mais ágil.

5.4 EXECUÇÃO

A execução definitiva dos testes ocorreu conforme descrito a seguir:

Linha de base e Não Preemptiva Nessas abordagens, a aplicação foi executada diretamente nas instâncias utilizando a diretiva `mpirun`, sem intervenções externas até a conclusão da execução. No caso da abordagem Não Preemptiva, a recuperação de falhas é totalmente transparente, dispensando a necessidade de reiniciar o comando `mpirun` após uma falha.

Preemptiva e Adaptativa Para essas abordagens, foi desenvolvida uma aplicação externa responsável por orquestrar todo o processo de recuperação, conforme ilustrado na Figura 7. Após 25% do progresso da execução, o orquestrador desliga propositalmente uma das instâncias do *cluster*, forçando a interrupção da aplicação. Em seguida, ele inicia uma nova *thread* encarregada de solicitar o provisionamento de uma nova instância na AWS, enquanto relança a execução com o `mpirun`, agora redistribuindo os quatro *ranks* apenas entre as instâncias sobreviventes.

Assim que a nova instância é provisionada, a *thread* notifica a aplicação, que realiza imediatamente um *checkpoint* em disco para salvar o progresso alcançado durante o período de recuperação e, então, encerra. Ao detectar o término desse processo, o orquestrador principal executa novamente a aplicação com o `mpirun`, desta vez utilizando todas as instâncias disponíveis.

Os tempos apresentados neste capítulo foram coletados automaticamente pela aplicação durante a execução, utilizando tanto diretivas de temporização no orquestrador — por meio de `clock_gettime` — quanto a função `MPI_Wtime()` da API MPI. Com essas medições registradas nos pontos adequados, a aplicação consegue, posteriormente, calcular a duração de cada etapa específica do processo.

O tempo de sobrecusto, por sua vez, é obtido pela diferença entre o tempo médio das iterações executadas antes da recuperação e o tempo médio das iterações realizadas após esse evento. Os tempos individuais de cada iteração são armazenados em

arquivos, e a média é calculada posteriormente, permitindo então determinar o valor do sobrecusto de forma precisa.

5.5 RESULTADOS

A Figura 8 e a Tabela 1 apresentam os resultados consolidados dos experimentos, permitindo observar com clareza as vantagens e limitações de cada abordagem avaliada.

Na estratégia preemptiva, o tempo de recuperação dos nós mostrou-se um fator determinante. Nos testes realizados com o HPC@Cloud Toolkit, o provisionamento de uma nova instância *on-demand* levou, em média, 81 segundos, com desvio padrão de 13,39 segundos. É importante destacar que a disponibilidade dessas instâncias difere das do tipo *spot*, podendo variar significativamente conforme a demanda, o que torna o tempo de recuperação intrinsecamente imprevisível.

A análise dos tempos de *checkpoint* evidencia uma diferença marcante entre as abordagens: a persistência em memória mostrou-se, em média, dez vezes mais rápida que a em disco. Em contrapartida, o *checkpoint* em disco oferece maior robustez e praticidade, por permitir a restauração do estado em execuções futuras e garantir tolerância a falhas totais do *cluster*. Além disso, conforme discutido anteriormente, na versão mais recente do *OpenMPI*, essa é a única opção plenamente viável para o cenário proposto neste trabalho.

As abordagens Adaptativa e Não Preemptiva, que utilizam *oversubscribe*, introduzem um custo adicional constante no tempo de cada iteração devido à competição por recursos. Os testes demonstraram que, após a falha de um dos quatro nós, o tempo médio por iteração praticamente dobrou (de cerca de 0,9s para 1,8s), resultado direto do compartilhamento de CPU e memória entre dois *ranks* em uma mesma instância. Esse efeito impacta especialmente a etapa de colisão da simulação LBM, de alta demanda computacional. Assim, conclui-se que o número de falhas tem relação direta com o aumento do custo por iteração nas execuções subsequentes.

A abordagem adaptativa apresentou desempenho semelhante à Não Preemptiva

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)
<i>Baseline</i>	-	-	-	91,87
Preemptiva (25%)	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
Adaptativa (25%)	23,31 (Disco)	(80,84 + 25,69) ²	0,87 ³	208,87

¹ Sobrecusto por iteração após a ocorrência da falha induzida.

² Provisionamento pela AWS + Reestruturação da aplicação.

³ Sobrecusto por iteração a durante etapa de recuperação.

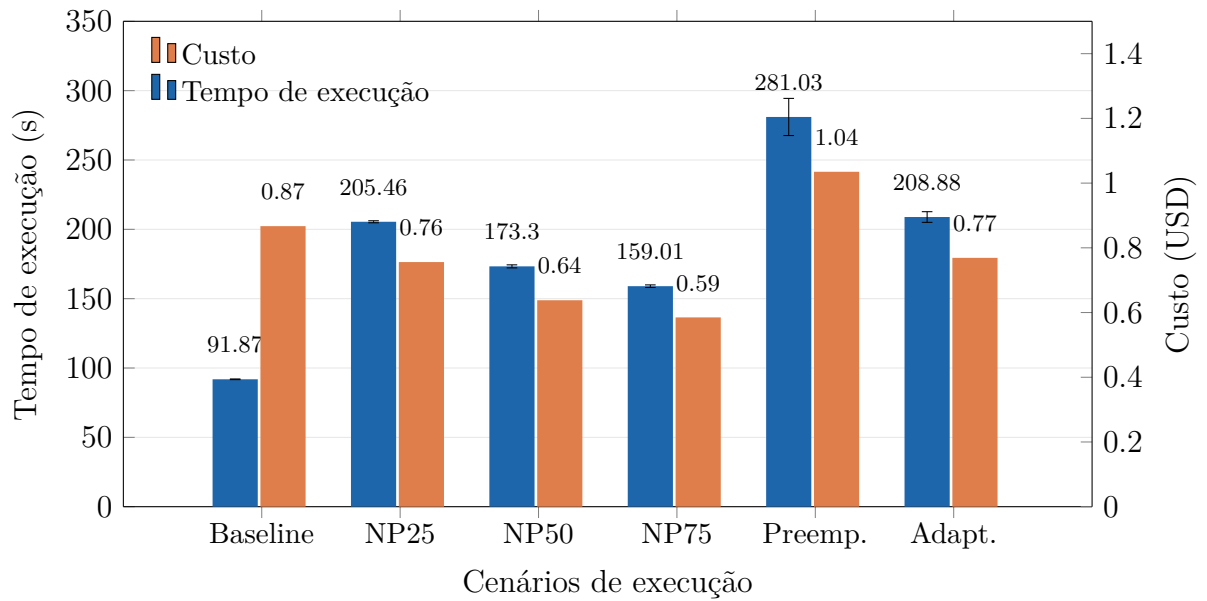


Figura 8 – Médias de tempo de execução com desvio padrão e custo estimado.

no cenário de 25% de progresso, porém com uma diferença fundamental: nas iterações finais, o sistema já havia recuperado o desempenho original, eliminando o sobrecusto, enquanto nas abordagens não preemptivas o tempo degradado se mantém até o término da execução.

Dessa forma, observa-se que a escolha da melhor estratégia de recuperação de falhas requer um equilíbrio entre múltiplos fatores, incluindo:

1. **Progresso da simulação no momento da falha:** define por quanto tempo a aplicação operará com desempenho reduzido;
2. **Latência de provisionamento de novas instâncias pela AWS:** determina o intervalo necessário para restaurar a capacidade total do sistema; e
3. **Custo adicional de desempenho:** em execuções com múltiplas falhas, o aumento acumulado do tempo pode tornar inviável prosseguir sem o provisionamento de novos nós.

6 CONCLUSÃO

Este trabalho avaliou a viabilidade técnica e econômica de executar aplicações fortemente acopladas de *High-Performance Computing* (HPC) em instâncias efêmeras (*spot*) de nuvens públicas, utilizando o método LBM da suíte SPEChpc® 2021 como caso de estudo. O foco consistiu em analisar o impacto da recuperação de falhas em ambientes sujeitos a revogações e comparar as estratégias Preemptiva, Não Preemptiva e Adaptativa sob os critérios de desempenho e custo.

A investigação envolveu a adaptação do código legado para o padrão MPI com a extensão ULFM, a construção de uma infraestrutura experimental na AWS e o desenvolvimento de um orquestrador externo capaz de contornar limitações críticas da implementação atual do OpenMPI.

6.1 SÍNTESE DAS CONTRIBUIÇÕES TÉCNICAS

A principal contribuição deste estudo consiste em demonstrar, na prática, os limites e potencialidades da tolerância a falhas baseada em ULFM no contexto de nuvens públicas instáveis.

Constatou-se que a versão atual do OpenMPI (v5.0.8), embora implemente a especificação ULFM, não suporta recuperações completas diante da queda de um nó físico. Nesses casos, o *runtime* PRTE força o encerramento dos processos, impossibilitando a recuperação nativa desejada. Assim, apenas falhas de processo são completamente tratáveis pela implementação existente.

Para viabilizar os experimentos, foi desenvolvido um Orquestrador externo, responsável por:

1. detectar a falha do nó,
2. solicitar o provisionamento de uma nova instância,
3. reinicializar a aplicação a partir do último *checkpoint*,
4. e coordenar o reequilíbrio dos *ranks* após a recuperação.

Essa solução permitiu avaliar de forma realista o comportamento de diferentes estratégias de recuperação. Os resultados mostraram ainda que o *checkpoint* em memória é cerca de 10× mais rápido que o *checkpoint* em disco, mas totalmente inviável diante de falhas de nó, reforçando a importância de mecanismos persistentes em ambientes de nuvem.

6.2 ANÁLISE COMPARATIVA DAS ESTRATÉGIAS

Os experimentos evidenciaram um conjunto claro de *trade-offs* entre tempo de recuperação, custo e estabilidade do desempenho:

- **Estratégia Não Preemptiva (*Oversubscribe*):** A redistribuição dos *ranks* sobreviventes elimina a espera por novas instâncias, mas dobra o tempo médio por iteração devido à contenção de recursos. Essa estratégia é vantajosa apenas **quando a falha ocorre nas etapas finais** (75% da execução). Em falhas iniciais, o sobre-custo acumulado anula a economia obtida com instâncias *spot*.
- **Estratégia Preemptiva:** Garante a restauração completa do desempenho inicial, pois reinicia a aplicação com quatro nós íntegros. Entretanto, o tempo de provisionamento da AWS — cerca de 81 segundos em média — introduz grande imprevisibilidade no tempo total da simulação.
- **Estratégia Adaptativa:** Combina as vantagens das duas anteriores. A aplicação continua executando em modo degradado enquanto aguarda o novo nó, e retorna automaticamente ao desempenho pleno após o reequilíbrio dos *ranks*. Os resultados mostraram que, em aplicações longas, o sobre-custo temporário é diluído, produzindo **um tempo total e custo próximo ao da execução sem falhas**, mas com o benefício da resiliência.

6.3 VIABILIDADE ECONÔMICA

A análise de custos comprovou que o uso de instâncias *spot* permanece financeiramente vantajoso mesmo após considerar:

1. o sobre-custo de desempenho das estratégias,
2. o tempo de provisionamento,
3. e o custo operacional do orquestrador e *checkpoints*.

Em especial, a estratégia Não Preemptiva com falha tardia resultou em cerca de 32% de economia em comparação ao cenário *on-demand*, ainda que tenha aumentado o tempo total em 42%. Para aplicações HPC onde o prazo de entrega é flexível, o modelo de execução resiliente com instâncias *spot* mostrou-se economicamente atrativo.

6.4 CONSIDERAÇÕES FINAIS

Conclui-se, portanto, que a execução de aplicações fortemente acopladas como o LBM em instâncias instáveis não é apenas possível, mas economicamente atraente, desde que o desenvolvedor assuma a responsabilidade pela resiliência. O modelo "faça você mesmo" imposto pelo ULFM exige um conhecimento profundo do fluxo de dados da aplicação, mas oferece em troca uma granularidade de controle que soluções genéricas de *checkpoint* de sistema (como BLCR ou DMTCP) não conseguem igualar em termos de eficiência.

Para a adoção em larga escala, recomenda-se a utilização da abordagem Adaptativa com *checkpoints* em disco - considerando principalmente a limitação do OpenMPI -,

pois esta combinação oferece a robustez necessária para lidar com a volatilidade da nuvem sem sacrificar excessivamente o desempenho da aplicação.

6.5 TRABALHOS FUTUROS

A partir das limitações e descobertas deste estudo, sugerem-se as seguintes linhas de investigação para trabalhos futuros:

1. **Integração com Alarmes de Revogação:** Utilizar os sinais de aviso de interrupção da AWS (que oferecem uma janela de 2 minutos antes do desligamento) para realizar *checkpoints* apenas quando necessário, eliminando o *overhead* dos salvamentos periódicos.
2. **Otimização de I/O:** Implementar o armazenamento de *checkpoints* em serviços de objetos como o Amazon S3 ou sistemas de arquivos paralelos de alto desempenho (e.g., Amazon FSx for Lustre), visando reduzir o gargalo de escrita observado nos testes com EBS.
3. **Decisão Autônoma:** Aprimorar o Orquestrador com algoritmos de tomada de decisão baseados em custo-benefício em tempo real, permitindo que o sistema decida automaticamente, no momento da falha, se vale a pena esperar por uma nova instância ou terminar a execução em modo degradado, baseando-se no percentual de progresso restante.
4. **Reavaliação do OpenMPI:** Repetir os experimentos assim que as correções para falhas de nó no *runtime* PRTE do OpenMPI forem estabilizadas, o que permitirá validar o potencial total dos *checkpoints* em memória em ambientes distribuídos.

REFERÊNCIAS

- AMOON, M. et al. On the design of reactive approach with flexible checkpoint interval to tolerate faults in cloud computing systems. **Journal of Ambient Intelligence and Humanized Computing**, v. 10, n. 11, p. 4567–4577, Nov 2019. ISSN 1868-5145. Disponível em: <https://doi.org/10.1007/s12652-018-1139-y>.
- ANSEL, J.; ARYA, K.; COOPERMAN, G. Dmtpc: Transparent checkpointing for cluster computations and the desktop. In: **2009 IEEE International Symposium on Parallel Distributed Processing**. [S.l.: s.n.], 2009. p. 1–12.
- AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, v. 1, n. 1, p. 11–33, 2004.
- BRUM, R. et al. Ensuring application continuity with fault tolerance techniques. In: _____. **High Performance Computing in Clouds : Moving HPC Applications to a Scalable and Cost-Effective Environment**. Cham: Springer International Publishing, 2023. p. 191–212. ISBN 978-3-031-29769-4. Disponível em: https://doi.org/10.1007/978-3-031-29769-4_10.
- BUYYA, R.; BROBERG, J.; GOSCINSKI, A. (Ed.). **Cloud Computing: Principles and Paradigms**. Hoboken, NJ: Wiley, 2011.
- CALORE, E. et al. Optimization of lattice boltzmann simulations on heterogeneous computers. **The International Journal of High Performance Computing Applications**, SAGE Publications, v. 33, n. 1, p. 124–139, abr. 2017. ISSN 1741-2846. Disponível em: <http://dx.doi.org/10.1177/1094342017703771>.
- CASTRO, M. Practical byzantine fault tolerance. 04 2001.
- CHEN, S.; DOOLEN, G. D. Lattice boltzmann method for fluid flows. **Annual Review of Fluid Mechanics**, Annual Reviews, v. 30, n. Volume 30, 1998, p. 329–364, 1998. ISSN 1545-4479. Disponível em: <https://www.annualreviews.org/content/journals/10.1146/annurev.fluid.30.1.329>.
- DOTTI, F. L.; MENDIZABAL, O. M.; SANTOS, O. M. dos. Verifying fault-tolerant distributed systems using object-based graph grammars. In: MAZIERO, C. A. et al. (Ed.). **Dependable Computing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 80–100. ISBN 978-3-540-32092-0.
- HARGROVE, P. H.; DUELL, J. C. Berkeley lab checkpoint/restart (blcr) for linux clusters. **Journal of Physics: Conference Series**, v. 46, n. 1, p. 494, sep 2006. Disponível em: <https://dx.doi.org/10.1088/1742-6596/46/1/067>.
- KOHL, N. et al. A scalable and extensible checkpointing scheme for massively parallel simulations. **arXiv:1708.08286 [cs.DC]**, v. 33, 08 2017.
- LOSADA, N. et al. Fault tolerance of mpi applications in exascale systems: The ulfm solution. **Future Generation Computer Systems**, v. 106, 05 2020.

MOODY, A. et al. 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**. [S.l.: s.n.], 2010. p. 1–11.

MUNHOZ, V. et al. 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)**. [S.l.: s.n.], 2023. p. 108–114.

MUNHOZ, V.; CASTRO, M. Enabling the execution of hpc applications on public clouds with hpc@cloud toolkit. **Concurrency and Computation: Practice and Experience**, v. 36, 12 2023.

MUNHOZ, V.; CASTRO, M. Enabling the execution of hpc applications on public clouds with hpc@cloud toolkit. **Concurrency and Computation: Practice and Experience**, v. 36, 12 2023.

MUNHOZ, V.; CASTRO, M.; MENDIZABAL, O. 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)**. [S.l.: s.n.], 2022. p. 263–272.

NETTO, M. et al. Hpc cloud for scientific and business applications: Taxonomy, vision, and research challenges. **ACM Computing Surveys**, v. 51, 10 2017.

QU, C.; CALHEIROS, R. N.; BUYYA, R. A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. **Journal of Network and Computer Applications**, v. 65, p. 167–180, 2016. ISSN 1084-8045. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1084804516300078>.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 22, n. 4, p. 299–319, dez. 1990. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/98163.98167>.

VARGAS, R.; MUNHOZ, V.; CASTRO, M. Implementação de tolerância a falhas no método lattice boltzmann para execução resiliente em instâncias efêmeras da aws. In: **Anais do Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)**. Porto Alegre, RS, Brasil: SBC, 2025. p. 37–48. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/sscad/article/view/37885>.

GLOSSÁRIO

- benchmark** Teste padronizado utilizado para avaliar o desempenho de sistemas computacionais, *softwares* ou componentes, permitindo comparações objetivas entre diferentes configurações ou implementações. 22, 23, 32, 39
- cluster** Conjunto de computadores interconectados que trabalham de forma colaborativa como se fossem um único sistema, com o objetivo de obter maior capacidade de processamento, desempenho ou disponibilidade. 9, 21, 24, 25, 38, 64
- Computação em Nuvem** Modelo de fornecimento de recursos computacionais, como servidores, armazenamento, bancos de dados, redes, *software* e análise, pela *internet* (“nuvem”), permitindo acesso sob demanda, escalabilidade e pagamento conforme o uso. Facilita a disponibilização rápida de recursos computacionais, reduz custos e oferece flexibilidade para aplicações de diferentes portes e áreas.. 9, 21, 23, 27, 37, 39
- E/S** Abreviação do termo em inglês *Input/Output*, que significa entrada e saída de dados em um sistema computacional. 30, 38, 52
- Ethernet** Tecnologia padrão para redes locais (LANs), que utiliza protocolos de comunicação para interligar dispositivos em uma rede com velocidades que variam de 10 Mbps a 100 Gbps, amplamente utilizada em *datacenters* e redes corporativas. 24
- HPC@Cloud Toolkit** Ferramenta desenvolvida para facilitar a execução e o gerenciamento de aplicações de Computação de Alto Desempenho (HPC) em ambientes de Computação em Nuvem. O *HPC@Cloud Toolkit* automatiza a orquestração de recursos, a configuração de instâncias, o provisionamento de infraestrutura e o monitoramento de falhas, oferecendo suporte tanto para instâncias sob demanda quanto para instâncias efêmeras, como as instâncias *spot*. Essa ferramenta visa simplificar o uso da nuvem para aplicações HPC, proporcionando maior flexibilidade, escalabilidade e otimização de custos.. 25, 39, 40, 45, 46, 47, 52
- InfiniBand** Arquitetura de rede de alta velocidade utilizada em *clusters* HPC, que proporciona baixa latência e alta largura de banda para comunicação entre nós computacionais. 22, 24, 30
- Instâncias Efêmeras** Tipo de instância em ambientes de computação em nuvem que pode ser revogada pelo provedor a qualquer momento, geralmente oferecida a um custo reduzido. Também conhecidas como instâncias *Spot*, são indicadas para aplicações tolerantes a falhas devido à sua volatilidade, permitindo economia significativa em comparação com instâncias sob demanda.. 9, 21, 22, 30, 37, 39
- On-Demand** Modelo de fornecimento de recursos computacionais baseado em nuvem, no qual os recursos (como CPU, memória e armazenamento) são disponibilizados

sob demanda e pagos conforme o uso, permitindo escalabilidade e flexibilidade sem a necessidade de infraestrutura local. 9, 22, 30, 39

On-Premise Modelo de infraestrutura onde os servidores, sistemas e recursos computacionais são instalados e mantidos fisicamente dentro da própria organização, diferentemente de soluções baseadas em nuvem. 23

OpenMPI Implementação de código aberto do padrão *Message Passing Interface* (MPI), amplamente utilizada em computação de alto desempenho (HPC) para permitir comunicação eficiente entre processos distribuídos em um *cluster*. O *OpenMPI* integra recursos de diversas implementações anteriores e oferece suporte a diferentes arquiteturas, redes e bibliotecas, sendo uma das principais opções para execução de aplicações paralelas baseadas em MPI. 25, 31, 39, 43, 54

RAM Sigla para *Random Access Memory*, ou memória de acesso aleatório, utilizada para armazenar temporariamente dados e instruções enquanto um programa está em execução. É volátil, ou seja, seus dados são perdidos ao desligar o computador. 30, 31, 38

SPEChpc® 2021 Suíte de *benchmarks* desenvolvida pela SPEC para avaliar o desempenho e a escalabilidade de aplicações paralelas em ambientes de computação de alto desempenho (HPC). 9, 11, 13, 22, 23, 32, 39

Spot Nome de instâncias efêmeras disponibilizadas pela provedora AWS. 9, 21, 22, 37, 39, 40, 63

SSD Sigla para *Solid-State Drive*, dispositivo de armazenamento de dados baseado em memória *flash*, mais rápido e resistente que discos rígidos tradicionais (HDs). 30, 38

tightly coupled Refere-se a sistemas, componentes ou processos que possuem alta dependência e comunicação frequente entre si, exigindo sincronização rigorosa para funcionar corretamente. Em computação paralela, um sistema *tightly coupled* tipicamente possui múltiplos processadores que compartilham memória e dependem uns dos outros para executar tarefas simultâneas com baixa latência. Essa arquitetura contrasta com sistemas *loosely coupled*, nos quais os componentes operam de forma mais independente e se comunicam de maneira menos frequente.. 24, 28

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

Rafael Luis Sol Veit Vargas¹, Vanderlei Munhoz^{1,2}, Márcio Castro¹

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

²University 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,

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.

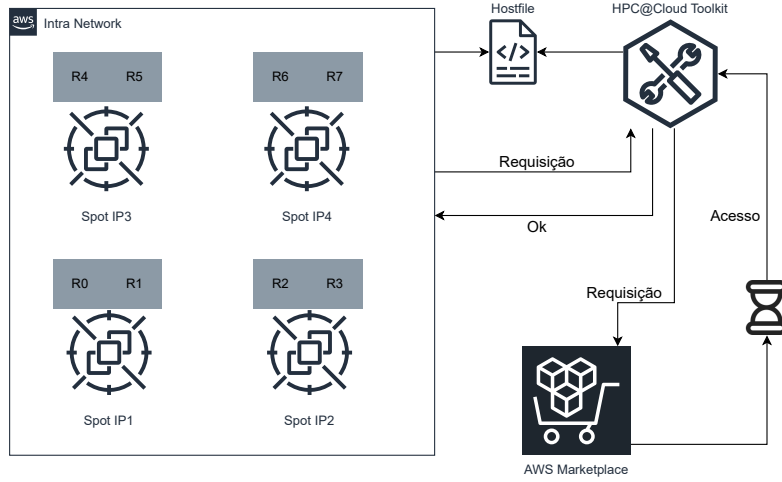


Figura 2. Estrutura de componentes da aplicação e fluxo de comunicação para requisição de nova instância.

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, $\{R_{(i-1) \bmod n}, \dots, R_{(i-x) \bmod n}\}$, 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

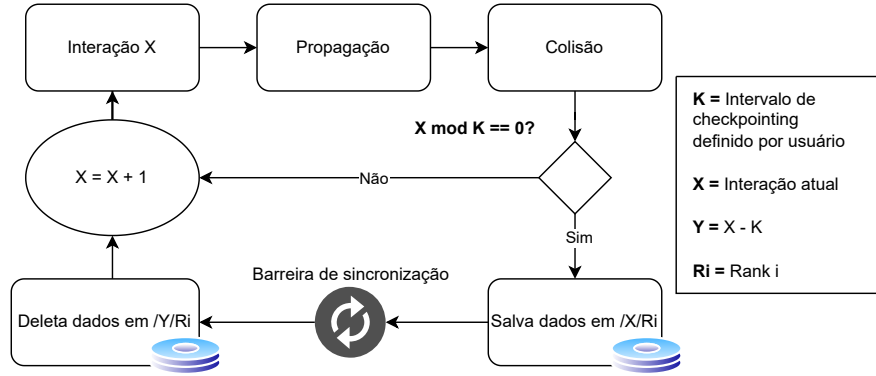


Figura 3. Algoritmo de *checkpoint* em disco (Amazon EBS) executando em *rank Ri*.

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.

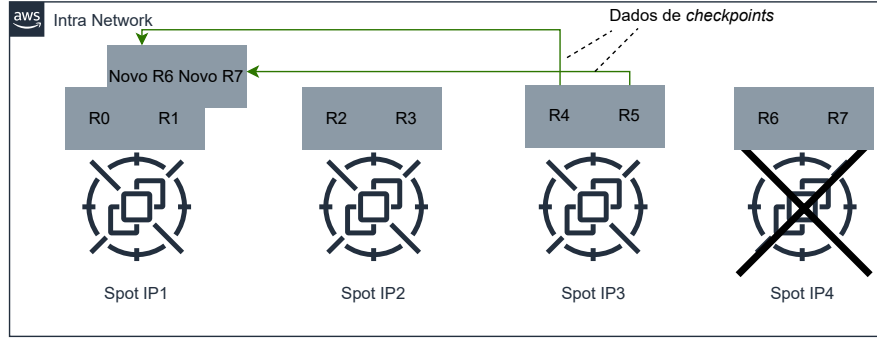


Figura 4. Realocação de *ranks* em nós existentes por meio de *oversubscribing*.

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 análises 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.

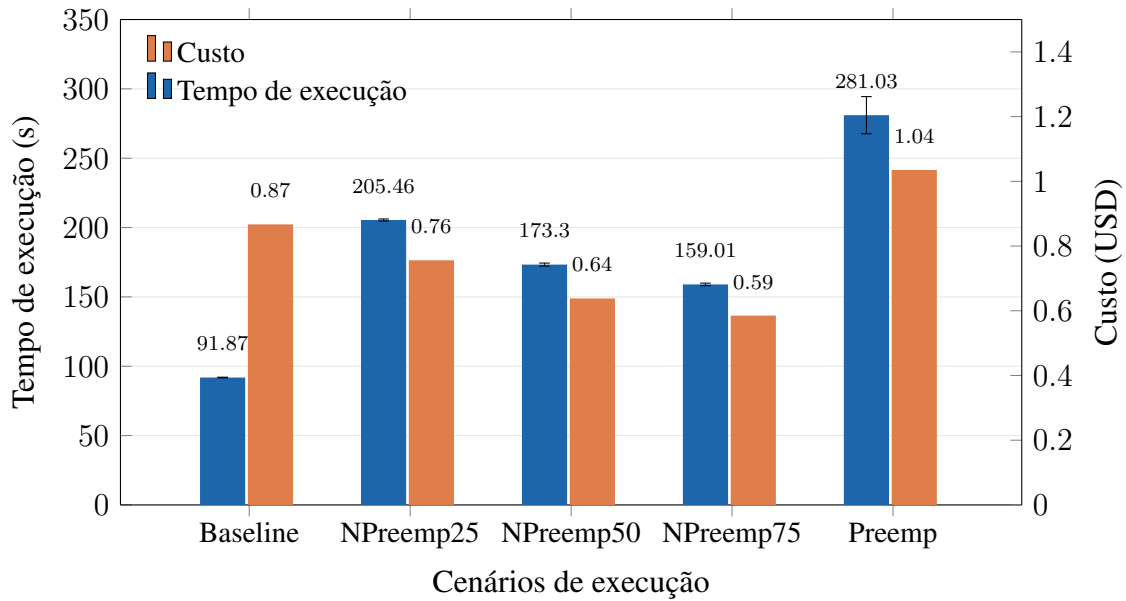


Figura 5. Médias de tempo de execução com desvio padrão e custo estimado.

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)
<i>Baseline</i>	-	-	-	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.

A CÓDIGO COMUM ÀS IMPLEMENTAÇÕES

ARQUIVOS DO DIRETÓRIO COMMON

Listing A.1 – common.h

```

1  /*****
2  *
3  * Copyright (c) 2018, University of Ferrara, University of Rome and INFN. All rights
   reserved.
4  *
5  *
6  * Authors listed in alphabetic order:
7  * -----
8  * Luca Biferale (University of Rome "Tor Vergata" and INFN)
9  * Enrico Calore (University of Ferrara and INFN)
10 * Alessandro Gabbana (University of Ferrara and INFN)
11 * Mauro Sbragaglia (University of Rome "Tor Vergata" and INFN)
12 * Andrea Scagliarini (CNR-IAC, Roma)
13 * Sebastiano Fabio Schifano (University of Ferrara and INFN)(*)
14 * Raffaele Tripiccione (University of Ferrara and INFN)
15 *
16 * (*) corresponding author, sebastiano.schifano __at__ unife.it
17 *
18 * We also thanks the following students:
19 * -----
20 * Elisa Pellegrini (University of Ferrara)
21 * Marco Zanella (University of Ferrara)
22 *
23 *
24 * Relevant publications:
25 * -----
26 *
27 * - Sbragaglia, M., Benzi, R., Biferale, L., Chen, H., Shan, X., Succi, S.
28 *   Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria
29 *   (2009) Journal of Fluid Mechanics, 628, pp. 299-309. Cited 66 times.
30 *   https://www.scopus.com/inward/record.uri?eid=2-s2.0-67651027859&doi=10.1017%2
   fS002211200900665X&partnerID=40&md5=1c57f2de4c0abc5aacaf5e30e9d566d7
31 *   DOI: 10.1017/S002211200900665X
32 *
33 * - Scagliarini, A., Biferale, L., Sbragaglia, M., Sugiyama, K., Toschi, F.
34 *   Lattice Boltzmann methods for thermal flows: Continuum limit and applications to
   compressible
35 * Rayleigh-Taylor systems (2010) Physics of Fluids, 22 (5), art. no. 019004PHF, pp. 1-21.
   Cited 51
36 * times.
37 *   https://www.scopus.com/inward/record.uri?eid=2-s2.0-77955618379&doi=10.1063%2f1
   .3392774&partnerID=40&md5=8b47e1b6c7ffd95583e2dc30d36a66f2
38 *   DOI: 10.1063/1.3392774
39 *
40 * - Calore, E., Gabbana, A., Kraus, J., Pellegrini, E., Schifano, S.F., Tripiccione, R.
41 *   Massively parallel lattice Boltzmann codes on large GPU clusters
42 *   (2016) Parallel Computing, 58, pp. 1-24.
43 *   https://www.scopus.com/inward/record.uri?eid=2-s2.0-84983616316&doi=10.1016%2fj.
   parco.2016.08.005&partnerID=40&md5=a9b40226e16f6e24bc8bc7138b789f90
44 *   DOI: 10.1016/j.parco.2016.08.005
45 *
46 * - Calore, E., Gabbana, A., Kraus, J., Schifano, S.F., Tripiccione, R.

```

```

47 * Performance and portability of accelerated lattice Boltzmann applications with
    OpenACC
48 * (2016) Concurrency Computation, 28 (12), pp. 3485-3502.
49 * https://www.scopus.com/inward/record.uri?eid=2-s2.0-84971215878&doi=10.1002%2fcpe
    .3862&partnerID=40&md5=fe286342abec002fa287da5967b9d2c2
50 * DOI: 10.1002/cpe.3862
51 *
52 *
53 * Redistribution and use in source and binary forms, with or without
54 * modification, are permitted provided that the following conditions
55 * are met:
56 *
57 * * Redistributions of source code must retain the above copyright
58 *   notice, this list of conditions and the following disclaimer.
59 *
60 * * Redistributions in binary form must reproduce the above copyright
61 *   notice, this list of conditions and the following disclaimer in the
62 *   documentation and/or other materials provided with the distribution.
63 *
64 * * Neither the name of the above Institutions nor the names of their
65 *   contributors may be used to endorse or promote products derived
66 *   from this software without specific prior written permission.
67 *
68 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND ANY
69 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
70 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
71 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
72 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
73 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
74 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
75 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
76 * OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
77 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
78 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
79 *
80 *****/
81
82 #ifndef _COMMON_H
83 #define _COMMON_H
84
85 #include <stdio.h>
86 #include <stdlib.h>
87
88 #include "physvar.h"
89
90 //////////////////////////////////////
91 #undef USE_TARGET
92 //////////////////////////////////////
93
94 #define USE_OPENMP 1
95
96 #ifndef USE_DEBUG
97 #define NO_USE_DEBUG 1
98 #endif
99
100 #ifdef SPEC_Y_MAJOR_ORDER
101 #define Y_MAJOR_ORDER 1
102 #else
103 #define X_MAJOR_ORDER 1
104 #endif

```

```

105 |
106 | ///////////////////////////////////////////////////////////////////
107 |
108 | #define CHECK_MASS 1 // enable mass check
109 |
110 | ///////////////////////////////////////////////////////////////////
111 |
112 | #define NPOP 37 // NUmber of Population.
113 |
114 | #define HX 3 // Dimension of horizontal halos.
115 | #define HY 16 // Dimension of vertical halos.
116 |
117 | #define hx 3 // Dimension of horizontal borders.
118 | #define hy 3 // Dimension of vertical borders.
119 |
120 | ///////////////////////////////////////////////////////////////////
121 |
122 | // #define __FLOAT__ // For single-precision code.
123 | #define __DOUBLE__ // For double-precision code.
124 |
125 | #ifdef __FLOAT__
126 | #define data_t float
127 | #undef __DOUBLE__
128 | #endif
129 |
130 | #ifdef __DOUBLE__
131 | #define data_t double
132 | #undef __FLOAT__
133 | #endif
134 |
135 | typedef int queue_t;
136 |
137 | ///////////////////////////////////////////////////////////////////
138 |
139 | #define MAX_FAILURE_SUPPORTED 1
140 | #define TIMEOUT_WAITING_SYNC 3 // Timeout em segundos
141 | #define MPI_ERR_TIMEOUT -1 // C digo de erro para timeout
142 |
143 | #define TAG_BUFF_SIZE 10
144 | #define TAG_STRUCT_DATA 20
145 | #define TAG_BUFF_DATA 30
146 |
147 | typedef struct {
148 |     data_t* buffer; // Buffer de dados
149 |     int LSIZEX; // Tamanho local no eixo X
150 |     int LSIZEY; // Tamanho local no eixo Y
151 |     uint64_t NX; // Tamanho da malha em X (incluindo halos)
152 |     uint64_t NY; // Tamanho da malha em Y (incluindo halos)
153 |     int LENGHT; // Tamanho total do buffer (NX * NY * NPOP)
154 |     int allocated; // Caso o buffer tenha sido alocado ou n o
155 |     int source_rank;
156 | } CheckpointData;
157 |
158 | typedef struct {
159 |     CheckpointData own_data;
160 |     CheckpointData* neighbor_data; // Array de checkpoints de vizinhos
161 |     int available_checkpoints; // N mero de checkpoints dispon veis
162 |     int checkpoint_creation_result; // 0 => Success | 1 => Error
163 | } MultiCheckpointData;
164 |

```

```

165
166 // Type used to represent a population.
167 typedef struct
168 {
169     data_t p[NPOP];
170 } pop_t;
171
172 // Type used to define physical attributes (velocity, density, temperature).
173 typedef struct
174 {
175     data_t u;
176     data_t v;
177     data_t rho;
178     data_t temp;
179 } uvrt_t;
180
181 // Tyoe use to define weights attributes.
182 typedef struct
183 {
184     data_t ww[NPOP];
185     data_t cx[NPOP];
186     data_t cy[NPOP];
187     data_t H0[NPOP];
188     data_t H1x[NPOP];
189     data_t H1y[NPOP];
190     data_t H2xx[NPOP];
191     data_t H2xy[NPOP];
192     data_t H2yy[NPOP];
193     data_t H3xxx[NPOP];
194     data_t H3xxy[NPOP];
195     data_t H3xyy[NPOP];
196     data_t H3yyy[NPOP];
197     data_t H4xxxx[NPOP];
198     data_t H4xxxy[NPOP];
199     data_t H4xxyy[NPOP];
200     data_t H4xyyy[NPOP];
201     data_t H4yyyy[NPOP];
202 } par_t;
203 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
204 // Constants
205
206 #define VZERO 0.0
207 #define VHALF 0.5
208 #define VONES 1.0
209 #define VTWOS 2.0
210 #define VTHRE 3.0
211 #define VFOUR 4.0
212 #define VSIXS 6.0
213 #define VEIGH 8.0
214 #define VTW04 24.0
215 #define _02 (1.0 / 2.0)
216 #define _04 (1.0 / 4.0)
217 #define _06 (1.0 / 6.0)
218 #define _08 (1.0 / 8.0)
219 #define _24 (1.0 / 24.0)
220 #define VCONST1 CONST1
221 #define VCONST2 CONST2
222 #define VCONST3 CONST3
223 #define VTAU1B TAU1B
224 #define VTAU2B TAU2B

```

```

225 #define VTAU3B TAU3B
226 #define forcex (FORCEPOIS * DELTAT)
227
228 //////////////////////////////////////
229
230 #endif /* _COMMON_H */

```

Listing A.2 – physvar.h

```

1  /*****
2  *
3  * Copyright (c) 2018, University of Ferrara, University of Rome and INFN. All rights
   reserved.
4  *
5  *
6  * Authors listed in alphabetic order:
7  * -----
8  * Luca Biferale (University of Rome "Tor Vergata" and INFN)
9  * Enrico Calore (University of Ferrara and INFN)
10 * Alessandro Gabbana (University of Ferrara and INFN)
11 * Mauro Sbragaglia (University of Rome "Tor Vergata" and INFN)
12 * Andrea Scagliarini (CNR-IAC, Roma)
13 * Sebastiano Fabio Schifano (University of Ferrara and INFN)(*)
14 * Raffaele Tripiccione (University of Ferrara and INFN)
15 *
16 * (*) corresponding author, sebastiano.schifano __at__ unife.it
17 *
18 * We also thanks the following students:
19 * -----
20 * Elisa Pellegrini (University of Ferrara)
21 * Marco Zanella (University of Ferrara)
22 *
23 *
24 * Relevant publications:
25 * -----
26 *
27 * - Sbragaglia, M., Benzi, R., Biferale, L., Chen, H., Shan, X., Succi, S.
28 *   Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria
29 *   (2009) Journal of Fluid Mechanics, 628, pp. 299-309. Cited 66 times.
30 *   https://www.scopus.com/inward/record.uri?eid=2-s2.0-67651027859&doi=10.1017%2
   fS002211200900665X&partnerID=40&md5=1c57f2de4c0abc5aacaf5e30e9d566d7
31 *   DOI: 10.1017/S002211200900665X
32 *
33 * - Scagliarini, A., Biferale, L., Sbragaglia, M., Sugiyama, K., Toschi, F.
34 *   Lattice Boltzmann methods for thermal flows: Continuum limit and applications to
   compressible
35 * Rayleigh-Taylor systems (2010) Physics of Fluids, 22 (5), art. no. 019004PHF, pp. 1-21.
   Cited 51
36 * times.
37 *   https://www.scopus.com/inward/record.uri?eid=2-s2.0-77955618379&doi=10.1063%2f1
   .3392774&partnerID=40&md5=8b47e1b6c7ffd95583e2dc30d36a66f2
38 *   DOI: 10.1063/1.3392774
39 *
40 * - Calore, E., Gabbana, A., Kraus, J., Pellegrini, E., Schifano, S.F., Tripiccione, R.
41 *   Massively parallel lattice Boltzmann codes on large GPU clusters
42 *   (2016) Parallel Computing, 58, pp. 1-24.
43 *   https://www.scopus.com/inward/record.uri?eid=2-s2.0-84983616316&doi=10.1016%2fj.
   parco.2016.08.005&partnerID=40&md5=a9b40226e16f6e24bc8bc7138b789f90
44 *   DOI: 10.1016/j.parco.2016.08.005
45 *

```

```

46 * - Calore, E., Gabbana, A., Kraus, J., Schifano, S.F., Tripiccione, R.
47 * Performance and portability of accelerated lattice Boltzmann applications with
    OpenACC
48 * (2016) Concurrency Computation, 28 (12), pp. 3485-3502.
49 * https://www.scopus.com/inward/record.uri?eid=2-s2.0-84971215878&doi=10.1002%2fcpe
    .3862&partnerID=40&md5=fe286342abec002fa287da5967b9d2c2
50 * DOI: 10.1002/cpe.3862
51 *
52 *
53 * Redistribution and use in source and binary forms, with or without
54 * modification, are permitted provided that the following conditions
55 * are met:
56 *
57 * * Redistributions of source code must retain the above copyright
58 * notice, this list of conditions and the following disclaimer.
59 *
60 * * Redistributions in binary form must reproduce the above copyright
61 * notice, this list of conditions and the following disclaimer in the
62 * documentation and/or other materials provided with the distribution.
63 *
64 * * Neither the name of the above Institutions nor the names of their
65 * contributors may be used to endorse or promote products derived
66 * from this software without specific prior written permission.
67 *
68 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND ANY
69 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
70 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
71 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
72 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
73 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
74 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
75 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
76 * OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
77 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
78 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
79 *
80 *****/
81
82 #ifndef _PHYSVAR_H
83 #define _PHYSVAR_H
84
85 //////////////////////////////////////
86
87 #define INIT_RANDOM_PHASE
88 #define ADIABATIC
89
90 #define TO 1.0
91 #define DT 0.05
92 #define IDT (1 / DT)
93 #define GRAVITY ((-8.33e-5) / 32.)
94 #define CONST1 (0.001)
95 #define CONST2 (0.001)
96 #define CONST3 (0.5)
97 #define K_MIN (1) // Used only in INIT_RANDOM_PHASE and INIT_RT.
98 #define K_MAX (LSIZEX / 2) // Used only in INIT_RANDOM_PHASE and INIT_RT.
99 #define EPS 10.0
100 #define WIDTH (2.0 * EPS) // Used only in INIT_RANDOM_PHASE and INIT_RT.
101
102 // Following parameters are 'historically stable'.
103 // WARNING: not obvious that the programs still

```

```

104 //      works correctly if any of these is changed!!!
105
106 #define TEMPWALLUP (T0 - DT)
107 #define TEMPWALLDOWN (T0 + DT)
108 #define RHOM 1.0
109
110 #define UNIT 1.19697977039307435897239 // 1.0
111 #define ENNE 2
112 #define DELTAT 1. / (UNIT)
113
114 #define FORCEPOIS 0.0
115
116 #define TAU1B (1.0)
117 #define TAU2B (1.0)
118 #define TAU3B (1.0)
119
120 #define FORCEX 0
121 #define FORCEY -1.0 * GRAVITY
122
123 #define RHOWDOWN1 0.8
124 #define RHOWDOWN2 0.8
125 #define RHOWDOWN3 0.8
126 #define RHOWUP1 1.2
127 #define RHOWUP2 1.2
128 #define RHOWUP3 1.2
129
130 #define UWALLUP 0.0
131 #define UWALLDOWN 0.0
132
133 #define VWALLUP 0.0
134 #define VWALLDOWN 0.0
135
136 ///////////////////////////////////////////////////////////////////
137
138 #define PI 3.141592654
139
140 #endif // _PHYSVAR_H

```

Listing A.3 – propagate_kernel.h

```

1  /*****
2  *
3  * Copyright (c) 2018, University of Ferrara, University of Rome and INFN. All rights
   reserved.
4  *
5  *
6  * Authors listed in alphabetic order:
7  * -----
8  * Luca Biferale (University of Rome "Tor Vergata" and INFN)
9  * Enrico Calore (University of Ferrara and INFN)
10 * Alessandro Gabbana (University of Ferrara and INFN)
11 * Mauro Sbragaglia (University of Rome "Tor Vergata" and INFN)
12 * Andrea Scagliarini (CNR-IAC, Roma)
13 * Sebastiano Fabio Schifano (University of Ferrara and INFN)(*)
14 * Raffaele Tripiccone (University of Ferrara and INFN)
15 *
16 * (*) corresponding author, sebastiano.schifano __at__ unife.it
17 *
18 * We also thanks the following students:
19 * -----

```

```

20 * Elisa Pellegrini (University of Ferrara)
21 * Marco Zanella (University of Ferrara)
22 *
23 *
24 * Relevant publications:
25 * -----
26 *
27 * - Sbragaglia, M., Benzi, R., Biferale, L., Chen, H., Shan, X., Succi, S.
28 *   Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria
29 *   (2009) Journal of Fluid Mechanics, 628, pp. 299-309. Cited 66 times.
30 *   https://www.scopus.com/inward/record.uri?eid=2-s2.0-67651027859&doi=10.1017%2
31 *   fS002211200900665X&partnerID=40&md5=1c57f2de4c0abc5aacaf5e30e9d566d7
32 *   DOI: 10.1017/S002211200900665X
33 * - Scagliarini, A., Biferale, L., Sbragaglia, M., Sugiyama, K., Toschi, F.
34 *   Lattice Boltzmann methods for thermal flows: Continuum limit and applications to
35 *   Rayleigh-Taylor systems (2010) Physics of Fluids, 22 (5), art. no. 019004PHF, pp. 1-21.
36 *   Cited 51
37 *   https://www.scopus.com/inward/record.uri?eid=2-s2.0-77955618379&doi=10.1063%2f1
38 *   .3392774&partnerID=40&md5=8b47e1b6c7ffd95583e2dc30d36a66f2
39 *   DOI: 10.1063/1.3392774
40 * - Calore, E., Gabbana, A., Kraus, J., Pellegrini, E., Schifano, S.F., Tripiccione, R.
41 *   Massively parallel lattice Boltzmann codes on large GPU clusters
42 *   (2016) Parallel Computing, 58, pp. 1-24.
43 *   https://www.scopus.com/inward/record.uri?eid=2-s2.0-84983616316&doi=10.1016%2fj.
44 *   parco.2016.08.005&partnerID=40&md5=a9b40226e16f6e24bc8bc7138b789f90
45 *   DOI: 10.1016/j.parco.2016.08.005
46 * - Calore, E., Gabbana, A., Kraus, J., Schifano, S.F., Tripiccione, R.
47 *   Performance and portability of accelerated lattice Boltzmann applications with
48 *   OpenACC
49 *   (2016) Concurrency Computation, 28 (12), pp. 3485-3502.
50 *   https://www.scopus.com/inward/record.uri?eid=2-s2.0-84971215878&doi=10.1002%2fcpe
51 *   .3862&partnerID=40&md5=fe286342abec002fa287da5967b9d2c2
52 *   DOI: 10.1002/cpe.3862
53 *   Redistribution and use in source and binary forms, with or without
54 *   modification, are permitted provided that the following conditions
55 *   are met:
56 *
57 *   * Redistributions of source code must retain the above copyright
58 *   notice, this list of conditions and the following disclaimer.
59 *
60 *   * Redistributions in binary form must reproduce the above copyright
61 *   notice, this list of conditions and the following disclaimer in the
62 *   documentation and/or other materials provided with the distribution.
63 *
64 *   * Neither the name of the above Institutions nor the names of their
65 *   contributors may be used to endorse or promote products derived
66 *   from this software without specific prior written permission.
67 *
68 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND ANY
69 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
70 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
71 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
72 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,

```

```

73  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
74  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
75  * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
76  * OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
77  * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
78  * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
79  *
80  *****/
81
82 #ifndef _PROPAGATE_H
83 #define _PROPAGATE_H
84
85 ///////////////////////////////////////////////////////////////////
86
87 void propagate(data_t* const __restrict__ nxt, const data_t* const __restrict__ prv,
88             const int startX, const int endX, const int startY, const int endY)
89 {
90
91     int ix, iy;
92     #ifdef USE_OPENMP
93         #pragma omp parallel for collapse(2) private(ix, iy)
94     #endif
95     for (ix = startX; ix < endX; ix++)
96     {
97         for (iy = startY; iy < endY; iy++)
98         {
99             nxt[ix * NY + iy] = prv[ix * NY + iy + -3 * NY + 1];
100             nxt[ix * NY + iy + NX * NY] = prv[ix * NY + iy + NX * NY - 3 * NY];
101             nxt[ix * NY + iy + 2 * NX * NY] = prv[ix * NY + iy + 2 * NX * NY - 3 * NY -
102                 1];
103             nxt[ix * NY + iy + 3 * NX * NY] = prv[ix * NY + iy + 3 * NX * NY - 2 * NY +
104                 2];
105             nxt[ix * NY + iy + 4 * NX * NY] = prv[ix * NY + iy + 4 * NX * NY - 2 * NY +
106                 1];
107             nxt[ix * NY + iy + 5 * NX * NY] = prv[ix * NY + iy + 5 * NX * NY - 2 * NY];
108             nxt[ix * NY + iy + 6 * NX * NY] = prv[ix * NY + iy + 6 * NX * NY - 2 * NY -
109                 1];
110             nxt[ix * NY + iy + 7 * NX * NY] = prv[ix * NY + iy + 7 * NX * NY - 2 * NY -
111                 2];
112             nxt[ix * NY + iy + 8 * NX * NY] = prv[ix * NY + iy + 8 * NX * NY - NY + 3];
113             nxt[ix * NY + iy + 9 * NX * NY] = prv[ix * NY + iy + 9 * NX * NY - NY + 2];
114             nxt[ix * NY + iy + 10 * NX * NY] = prv[ix * NY + iy + 10 * NX * NY - NY + 1];
115             nxt[ix * NY + iy + 11 * NX * NY] = prv[ix * NY + iy + 11 * NX * NY - NY];
116             nxt[ix * NY + iy + 12 * NX * NY] = prv[ix * NY + iy + 12 * NX * NY - NY - 1];
117             nxt[ix * NY + iy + 13 * NX * NY] = prv[ix * NY + iy + 13 * NX * NY - NY - 2];
118             nxt[ix * NY + iy + 14 * NX * NY] = prv[ix * NY + iy + 14 * NX * NY - NY - 3];
119             nxt[ix * NY + iy + 15 * NX * NY] = prv[ix * NY + iy + 15 * NX * NY + 3];
120             nxt[ix * NY + iy + 16 * NX * NY] = prv[ix * NY + iy + 16 * NX * NY + 2];
121             nxt[ix * NY + iy + 17 * NX * NY] = prv[ix * NY + iy + 17 * NX * NY + 1];
122             nxt[ix * NY + iy + 18 * NX * NY] = prv[ix * NY + iy + 18 * NX * NY];
123             nxt[ix * NY + iy + 19 * NX * NY] = prv[ix * NY + iy + 19 * NX * NY - 1];
124             nxt[ix * NY + iy + 20 * NX * NY] = prv[ix * NY + iy + 20 * NX * NY - 2];
125             nxt[ix * NY + iy + 21 * NX * NY] = prv[ix * NY + iy + 21 * NX * NY - 3];
126             nxt[ix * NY + iy + 22 * NX * NY] = prv[ix * NY + iy + 22 * NX * NY + NY + 3];
127             nxt[ix * NY + iy + 23 * NX * NY] = prv[ix * NY + iy + 23 * NX * NY + NY + 2];
128             nxt[ix * NY + iy + 24 * NX * NY] = prv[ix * NY + iy + 24 * NX * NY + NY + 1];
129             nxt[ix * NY + iy + 25 * NX * NY] = prv[ix * NY + iy + 25 * NX * NY + NY];
130             nxt[ix * NY + iy + 26 * NX * NY] = prv[ix * NY + iy + 26 * NX * NY + NY - 1];
131             nxt[ix * NY + iy + 27 * NX * NY] = prv[ix * NY + iy + 27 * NX * NY + NY - 2];
132             nxt[ix * NY + iy + 28 * NX * NY] = prv[ix * NY + iy + 28 * NX * NY + NY - 3];

```

```

128     nxt[ix * NY + iy + 29 * NX * NY] = prv[ix * NY + iy + 29 * NX * NY + 2 * NY +
129         2];
130     nxt[ix * NY + iy + 30 * NX * NY] = prv[ix * NY + iy + 30 * NX * NY + 2 * NY +
131         1];
132     nxt[ix * NY + iy + 31 * NX * NY] = prv[ix * NY + iy + 31 * NX * NY + 2 * NY];
133     nxt[ix * NY + iy + 32 * NX * NY] = prv[ix * NY + iy + 32 * NX * NY + 2 * NY -
134         1];
135     nxt[ix * NY + iy + 33 * NX * NY] = prv[ix * NY + iy + 33 * NX * NY + 2 * NY -
136         2];
137     nxt[ix * NY + iy + 34 * NX * NY] = prv[ix * NY + iy + 34 * NX * NY + 3 * NY +
138         1];
139     nxt[ix * NY + iy + 35 * NX * NY] = prv[ix * NY + iy + 35 * NX * NY + 3 * NY];
140     nxt[ix * NY + iy + 36 * NX * NY] = prv[ix * NY + iy + 36 * NX * NY + 3 * NY -
141         1];
142 }
143 }
144 }
145 //////////////////////////////////////////////////
146 #endif /* _PROPAGATE_H */

```

Listing A.4 – utils.h

```

1  /*****
2  *
3  * Copyright (c) 2018, University of Ferrara, University of Rome and INFN. All
4  *rights reserved.
5  *
6  *
7  * Authors listed in alphabetic order:
8  * -----
9  * Luca Biferale (University of Rome "Tor Vergata" and INFN)
10 * Enrico Calore (University of Ferrara and INFN)
11 * Alessandro Gabbana (University of Ferrara and INFN)
12 * Mauro Sbragaglia (University of Rome "Tor Vergata" and INFN)
13 * Andrea Scagliarini (CNR-IAC, Roma)
14 * Sebastiano Fabio Schifano (University of Ferrara and INFN)(*)
15 * Raffaele Tripiccione (University of Ferrara and INFN)
16 *
17 * (*) corresponding author, sebastiano.schifano __at__ unife.it
18 *
19 * We also thanks the following students:
20 * -----
21 * Elisa Pellegrini (University of Ferrara)
22 * Marco Zanella (University of Ferrara)
23 *
24 *
25 * Relevant publications:
26 * -----
27 *
28 * - Sbragaglia, M., Benzi, R., Biferale, L., Chen, H., Shan, X., Succi, S.
29 *   Lattice Boltzmann method with self-consistent thermo-hydrodynamic
30 *   equilibria (2009) Journal of Fluid Mechanics, 628, pp. 299-309. Cited 66
31 *   times.
32 *   https://www.scopus.com/inward/record.uri?eid=2-s2.0-67651027859&doi=10.1017%2
33 *   fS002211200900665X&partnerID=40&md5=1c57f2de4c0abc5aacaf5e30e9d566d7
34 *   DOI: 10.1017/S002211200900665X
35 *
36 * - Scagliarini, A., Biferale, L., Sbragaglia, M., Sugiyama, K., Toschi, F.

```

```

36 * Lattice Boltzmann methods for thermal flows: Continuum limit and
37 * applications to compressible Rayleigh-Taylor systems (2010) Physics of Fluids,
38 * 22 (5), art. no. 019004PHF, pp. 1-21. Cited 51 times.
39 * https://www.scopus.com/inward/record.uri?eid=2-s2.0-77955618379&doi=10.1063%2f1
    .3392774&partnerID=40&md5=8b47e1b6c7ffd95583e2dc30d36a66f2
40 * DOI: 10.1063/1.3392774
41 *
42 * - Calore, E., Gabbana, A., Kraus, J., Pellegrini, E., Schifano, S.F.,
43 * Tripiccione, R. Massively parallel lattice Boltzmann codes on large GPU
44 * clusters (2016) Parallel Computing, 58, pp. 1-24.
45 * https://www.scopus.com/inward/record.uri?eid=2-s2.0-84983616316&doi=10.1016%2fj.
    parco.2016.08.005&partnerID=40&md5=a9b40226e16f6e24bc8bc7138b789f90
46 * DOI: 10.1016/j.parco.2016.08.005
47 *
48 * - Calore, E., Gabbana, A., Kraus, J., Schifano, S.F., Tripiccione, R.
49 * Performance and portability of accelerated lattice Boltzmann applications
50 * with OpenACC (2016) Concurrency Computation, 28 (12), pp. 3485-3502.
51 * https://www.scopus.com/inward/record.uri?eid=2-s2.0-84971215878&doi=10.1002%2fcpe
    .3862&partnerID=40&md5=fe286342abec002fa287da5967b9d2c2
52 * DOI: 10.1002/cpe.3862
53 *
54 *
55 * Redistribution and use in source and binary forms, with or without
56 * modification, are permitted provided that the following conditions
57 * are met:
58 *
59 * * Redistributions of source code must retain the above copyright
60 * notice, this list of conditions and the following disclaimer.
61 *
62 * * Redistributions in binary form must reproduce the above copyright
63 * notice, this list of conditions and the following disclaimer in the
64 * documentation and/or other materials provided with the distribution.
65 *
66 * * Neither the name of the above Institutions nor the names of their
67 * contributors may be used to endorse or promote products derived
68 * from this software without specific prior written permission.
69 *
70 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND ANY
71 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
72 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
73 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
74 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
75 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
76 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
77 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
78 * OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
79 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
80 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
81 *
82 *****/
83
84 #ifndef _UTILS_M_H
85 #define _UTILS_M_H
86
87 #include <stdint.h>
88 #include <stdio.h>
89 #include <stdlib.h>
90 #include <unistd.h>
91
92 ////////////////////////////////////////

```

```

93
94 // SoAtoAoS -> Estrutura de LBM para estrutura de print
95 void SoAtoAoSOld(data_t* array, pop_t* AoS, int nx_to_use, int ny_to_use)
96 {
97     int ii, jj;
98     uint64_t size;
99
100     size = NX * NY;
101     #ifdef USE_OPENMP
102         #pragma omp parallel for private(jj)
103     #endif
104     for (ii = 0; ii < size; ii++)
105     {
106         for (jj = 0; jj < NPOP; jj++)
107         {
108             array[jj * size + ii] = AoS[ii].p[jj];
109         }
110     }
111 }
112
113 void export_execution_time(const char* filename, double time_in_seconds) {
114     FILE* logfile = fopen(filename, "a");
115     if (logfile != NULL) {
116         fprintf(logfile, "%f\n", time_in_seconds);
117         fclose(logfile);
118     } else {
119         fprintf(stderr, "[ERROR] Failed to write time to %s\n", filename);
120     }
121 }
122
123 ///////////////////////////////////////////////////////////////////
124
125 void SoAtoAoS(data_t* array, pop_t* AoS)
126 {
127     int ii, jj;
128     uint64_t size;
129
130     size = NX * NY;
131     #ifdef USE_OPENMP
132         #pragma omp parallel for private(jj)
133     #endif
134     for (ii = 0; ii < size; ii++)
135     {
136         for (jj = 0; jj < NPOP; jj++)
137         {
138             AoS[ii].p[jj] = array[jj * size + ii];
139         }
140     }
141 }
142
143 void AoStoSoA(pop_t* AoS, CheckpointData* data)
144 {
145     int ii, jj;
146     uint64_t size = data->NX * data->NY;
147
148     <<<<<< HEAD:utils.h
149     for (ii = 0; ii < size; ii++)
150     =====
151     mass = 0.0;
152     #ifdef USE_OPENMP

```

```

153     #pragma omp parallel for private(jj, kk) reduction(+ : mass)
154     #endif
155     for (ii = HX; ii < HX + LSIZEEX; ii++)
156 >>>>>> 3dcea156beb2a0543eeb0c6af0af87711a5d0e16:src/utils.h
157     {
158         for (jj = 0; jj < NPOP; jj++)
159         {
160             data->buffer[jj * size + ii] = AoS[ii].p[jj];
161         }
162     }
163 }
164
165 double somar_doubles(const double *buffer, int quantidade) {
166     double soma = 0.0;
167     for (int i = 0; i < quantidade; i++) {
168         soma += buffer[i];
169     }
170     return soma;
171 }
172
173 /// 0 -> HX + LSIZEEX + HX + HY + LSIZEY + HY (NX * NY)
174 /// 1 -> 3 + (2/3) + 3 + 16 + (1) + 16
175 /// 2 ->
176 /// ...->
177 /// 36 ->
178 void sum_buffers_new_two(CheckpointData* buffer1, CheckpointData* buffer2, int debug)
179 {
180     // Inicializa buffers intermediarios para receberem os dados no formato AoS
181     pop_t* B1 = NULL;
182     pop_t* B2 = NULL;
183
184     posix_memalign((void**)&B1, 4096, buffer1->NX * buffer1->NY * sizeof(pop_t));
185
186     if (B1 == NULL)
187     {
188         fprintf(stderr, "[main]_ERROR:_B1_alloc_failed\n");
189         MPI_Abort(MPI_COMM_WORLD, -1);
190     }
191
192     posix_memalign((void**)&B2, 4096, buffer2->NX * buffer2->NY * sizeof(pop_t));
193
194     if (B2 == NULL)
195     {
196         fprintf(stderr, "[main]_ERROR:_B2_alloc_failed\n");
197         MPI_Abort(MPI_COMM_WORLD, -1);
198     }
199
200     // Converte os dados de SoA para os buffers intermediarios
201     SoAtoAoS0ld(buffer1->buffer, B1, buffer1->NX, buffer1->NY);
202     SoAtoAoS0ld(buffer2->buffer, B2, buffer2->NX, buffer2->NY);
203
204     if (debug)
205     {
206         printf("Come ando_execu o_compar metros\n"
207             "B1:_{\n"
208             "    LSIZEEX:_%d,\n"
209             "    LSIZEY:_%d,\n"
210             "    NX:_%ld,\n"
211             "    NY:_%ld\n"
212             "},\n"

```

```

213         "B2:\n"
214         "    LSIZEEX:\n"
215         "    LSIZEY:\n"
216         "    NX:\n"
217         "    NY:\n"
218         "}\n",
219         buffer1->LSIZEEX, buffer1->LSIZEY, buffer1->NX, buffer1->NY, buffer2->
                LSIZEEX, buffer2->LSIZEY, buffer2->NX, buffer2->NY);
220     }
221
222     int    access        = 0;
223     data_t transportedMass = 0;
224
225     //int b1MaxII = 2 * HX + buffer1->LSIZEEX - 1;
226     //int b1MaxJJ = HY + buffer1->LSIZEY + HX - 1;
227     int b1MaxII = HX + buffer1->LSIZEEX - 1;
228     int b1MaxJJ = HY + buffer1->LSIZEY - 1;
229
230     // Passa por todas as posi es de arrays do buffer 2 e calcula a posi o
        respectiva no buffer 1
231     for (int ii = HX; ii < HX + buffer2->LSIZEEX; ii++)
232     //for (int ii = 0; ii < 2 * HX + buffer2->LSIZEEX; ii++)
233     //for (int ii = 0; ii < NX; ii++)
234     {
235         for (int jj = HY; jj < HY + buffer2->LSIZEY; jj++)
236         //for (int jj = HY - HX; jj < HY + buffer2->LSIZEY + HX; jj++)
237         //for (int jj = 0; jj < NY; jj++)
238         {
239             for (int kk = 0; kk < NPOP; kk++)
240             {
241                 int b1ii = ii > b1MaxII ? b1MaxII : ii;
242                 int b1jj = jj > b1MaxJJ ? b1MaxJJ : jj;
243
244                 size_t b1Index = (b1ii * buffer1->NY + b1jj) * NPOP + kk;
245                 size_t b2Index = (ii * buffer2->NY + jj) * NPOP + kk;
246
247                 B1->p[b1Index] += B2->p[b2Index];
248
249                 if (debug)
250                 {
251                     data_t prevValue = B1->p[b1Index];
252                     access++;
253                     transportedMass += B2->p[b2Index];
254                     //printf("B2 [%d][%d][%d] => (%ld) para B1 [%d][%d][%d] => (%ld). %f
                        para %f\n", ii, jj, kk, b2Index, b1ii, b1jj, kk, b1Index,
255                             //
                                prevValue, B1->p[b1Index]);
256                 }
257             }
258         }
259     }
260
261     if (debug)
262     {
263         printf("Finalizando soma, foi transportado %.10e de massa com %d acessos\n",
                transportedMass, access);
264     }
265
266     AoStoSoA(B1, buffer1);
267 }
268

```

```

269 ///////////////////////////////////////////////////////////////////
270
271 void SoAtoAoS(CheckpointData* data, pop_t* AoS)
272 {
273     int ii, jj;
274     // Essa verifica o acontece pois na opera o de Checkpoint pode ocorrer de um
275     // receber um CheckpointData que tenha dados em um buffer maior que o AoS local,
276     // feito um truncamento dos dados salvos localmente
277     uint64_t remote_size = data->NX * data->NY;
278     uint64_t local_size = NX * NY;
279
280     for (ii = 0; ii < remote_size; ii++)
281     {
282         int local_index = (ii < local_size) ? ii : local_size - 1;
283
284         for (jj = 0; jj < NPOP; jj++)
285         {
286             double g = data->buffer[jj * remote_size + ii];
287             AoS[local_index].p[jj] = g;
288         }
289     }
290 }
291 ///////////////////////////////////////////////////////////////////
292
293 data_t dumpMass(CheckpointData* data, pop_t* f)
294 {
295     int ii, jj, kk;
296     double mass = 0.0;
297
298     for (ii = HX; ii < HX + data->LSIZEX; ii++)
299     {
300         for (jj = HY; jj < HY + data->LSIZEY; jj++)
301         {
302             for (kk = 0; kk < NPOP; kk++)
303             {
304                 double g = f[ii * data->NY + jj].p[kk];
305
306                 mass += g;
307             }
308         }
309     }
310
311     return (data_t)mass;
312 }
313
314 data_t dumpMassPrint(CheckpointData* data, pop_t* f, int debug)
315 {
316     int ii, jj, kk;
317     double mass = 0.0;
318
319     if (debug)
320     {
321         printf("|||||||||||||||||||||_Matriz_de_somatorios_(kk_somado):_
322             |||||||||||||||||||\\n");
323     }
324
325     for (jj = 0; jj < 2 * HY + data->LSIZEY; jj++)
326     {

```

```

326     for (ii = 0; ii < 2 * HX + data->LSIZEX; ii++)
327     {
328         double soma_local = 0.0;
329
330         for (kk = 0; kk < NPOP; kk++)
331         {
332             double g = f[ii * data->NY + jj].p[kk];
333             soma_local += g;
334             mass += g;
335         }
336
337         if (debug)
338         {
339             printf("%6.3f", soma_local);
340         }
341     }
342
343     if (debug)
344     {
345         printf("\n");
346     }
347 }
348
349 if (debug)
350 {
351     printf("|||||||||||||||||||||||_-----_
352         |||||||||||||||||||||||\n");
353 }
354
355 return (data_t)mass;
356 }
357
358 void free_checkpoint_store(MultiCheckpointData* store)
359 {
360     if (!store)
361         return;
362
363     if (store->own_data.allocated && store->own_data.buffer)
364     {
365         free(store->own_data.buffer);
366     }
367
368     if (store->neighbor_data)
369     {
370         for (int i = 0; i < store->available_checkpoints; i++)
371         {
372             if (store->neighbor_data[i].allocated && store->neighbor_data[i].buffer)
373             {
374                 free(store->neighbor_data[i].buffer);
375             }
376         }
377         free(store->neighbor_data);
378     }
379
380     free(store);
381 }
382
383 void copy_data_from_to(CheckpointData* from, CheckpointData* to)
384 {

```

```

385     if (from == NULL || to == NULL || from->buffer == NULL || to->buffer == NULL)
386     {
387         fprintf(stderr, "Erro: Um dos buffers ou estruturas é NULL.\n");
388         return;
389     }
390
391     to->NX      = from->NX;
392     to->NY      = from->NY;
393     to->LSIZEX  = from->LSIZEX;
394     to->LSIZEY  = from->LSIZEY;
395     to->LENGHT  = from->LENGHT;
396
397     // Verifica se os buffers têm o mesmo tamanho
398     size_t min_length = (from->LENGHT < to->LENGHT) ? from->LENGHT : to->LENGHT;
399
400     // Copia apenas a o menor tamanho
401     memcpy(to->buffer, from->buffer, min_length * sizeof(data_t));
402 }
403
404 bool has_mpi_communication_errors(MPI_Comm* comm)
405 {
406     int mpi_finalized;
407     MPI_Finalized(&mpi_finalized);
408     if (mpi_finalized)
409     {
410         printf("Comunicador finalizado!!!\n");
411         return true; // MPI já finalizado - situação de erro
412     }
413
414     // Verifica se o comunicador é válido
415     int comm_valid;
416     MPIX_Comm_is_revoked(*comm, &comm_valid);
417     if (comm_valid)
418     {
419         printf("Comunicador revogado!!!\n");
420         return true; // Comunicador revogado - erro de comunicação
421     }
422
423     // Verifica a ocorrência de processos falhos
424     MPI_Group group_f;
425     int nf;
426
427     MPIX_Comm_failure_ack(*comm);
428     MPIX_Comm_failure_get_acked(*comm, &group_f);
429     MPI_Group_size(group_f, &nf);
430     MPI_Group_free(&group_f);
431
432     return (nf > 0);
433 }
434
435 // Função auxiliar para verificar falhas durante operações MPI
436 int check_mpi_operation(int error_code, MPI_Comm comm, int* recovery_needed)
437 {
438     *recovery_needed = 0;
439
440     if (error_code == MPI_SUCCESS)
441         return 0;
442
443     char error_string[MPI_MAX_ERROR_STRING];
444     int error_len;

```

```

445 MPI_Error_string(error_code, error_string, &error_len);
446
447 // Verifica se o erro recuperavel
448 if (error_code == MPI_ERR_PROC_FAILED || error_code == MPI_ERR_REVOKED)
449 {
450     *recovery_needed = 1;
451     return 1;
452 }
453
454 fprintf(stderr, "Erro MPI irreparavel: %s\n", error_string);
455 return -1;
456 }
457
458 #include <mpi.h>
459 #include <stdio.h>
460
461 int print_failures_found(MPI_Comm* comm, int mpi_rank, int id, long int interaction)
462 {
463     MPI_Group group_f, group_c;
464     int nf;
465
466     MPIX_Comm_failure_ack(*comm);
467     MPIX_Comm_failure_get_acked(*comm, &group_f);
468     MPI_Group_size(group_f, &nf);
469
470     //printf("[TIMEOUT(R=%d) (ID=%d) (I=%ld)] Timeout aguardando requisico! Quantidade
471     //de Ranks falhos: (%d)\n", mpi_rank, id, interaction, nf);
472 }
473
474 int wait_for_mpi_requests_with_timeout(MPI_Request* requests, int num_requests, MPI_Comm*
475 comm, int mpi_rank, int debug, int id, long int iteration)
476 {
477     int* completed_flags = (int*)malloc(num_requests * sizeof(int));
478     if (!completed_flags)
479     {
480         fprintf(stderr, "Erro ao alocar memoria para os flags de requisico.\n");
481         MPI_Abort(*comm, MPI_ERR_OTHER);
482     }
483
484     for (int i = 0; i < num_requests; i++)
485     {
486         completed_flags[i] = 0;
487     }
488
489     double start_time = MPI_Wtime();
490
491     while (1)
492     {
493         int all_completed = 1; // Assume que todas as requisicoes estao concluidas
494
495         for (int i = 0; i < num_requests; i++)
496         {
497             if (!completed_flags[i])
498             { // Se ainda nao foi concluido
499                 MPI_Test(&requests[i], &completed_flags[i], MPI_STATUS_IGNORE);
500                 if (!completed_flags[i])
501                 {
502                     all_completed = 0; // Pelo menos uma requisicoe ainda est
503                     //pendente
504                 }
505             }
506         }
507     }

```

```

502     }
503 }
504
505 if (all_completed)
506 {
507     free(completed_flags);
508     return MPI_SUCCESS; // Todas as requisicoes foram concluidas com sucesso
509 }
510
511 // Verifica timeout ou erro de comunicacao
512 if (MPI_Wtime() - start_time > TIMEOUT_WAITING_SYNC ||
513     has_mpi_communication_errors(comm))
514 {
515     free(completed_flags);
516     if (debug)
517     {
518         MPI_Group group_f, group_c;
519         int nf;
520
521         MPIX_Comm_failure_ack(*comm);
522         MPIX_Comm_failure_get_acked(*comm, &group_f);
523         MPI_Group_size(group_f, &nf);
524
525         //printf("[TIMEOUT(R=%d) (ID=%d) (I=%ld)] Timeout aguardando requisicao
526             ! Quantidade de Ranks falhos: (%d)\n", mpi_rank, id, iteration, nf);
527     }
528     return MPI_ERR_TIMEOUT; // Timeout atingido
529 }
530 }
531
532 #include <mpi.h>
533 #include <stdbool.h>
534
535 /**
536  * @brief Barreira tolerante a falhas
537  * @param comm Ponteiro para o comunicador MPI (pode ser modificado se houver falhas)
538  */
539 int Fault_Tolerante_Barrier(MPI_Comm* comm)
540 {
541     int initial_size, current_size;
542     MPI_Comm_size(*comm, &initial_size);
543
544     // Primeiro verifica se o comunicador ainda estavelido
545     int is_revoked;
546     MPIX_Comm_is_revoked(*comm, &is_revoked);
547     if (is_revoked)
548     {
549         return false;
550     }
551
552     // Tenta uma barreira normal primeiro (caso comum sem falhas)
553     int barrier_error = MPI_Barrier(*comm);
554
555     if (barrier_error != MPI_SUCCESS)
556     {
557         // MPIX_Comm_failure_ack(*comm);
558         // MPIX_Comm_revoke(*comm);
559     }

```

```

560
561     return barrier_error;
562 }
563
564 int establish_simple_consensus(MPI_Comm* comm, int mpi_rank, int mpi_size, int root_rank,
    int numbers_failed)
565 {
566     #define CONSENSUS_READY_TAG 1003
567
568     int consensus_reached = 0;
569     MPI_Status status;
570
571     printf("[%d] Iniciando processo de consen o...\n", mpi_rank);
572     if (mpi_rank == root_rank)
573     {
574         // ROOT: espera receber mensagem de todos os ranks sobreviventes
575         int* survivors = (int*)calloc(mpi_size, sizeof(int));
576         int received = 0;
577
578         int waiting_tags = (mpi_size - numbers_failed - 1);
579
580         while (received < waiting_tags)
581         { // -1 porque n o conta a si mesmo
582             int incoming_rank;
583             MPI_Recv(&incoming_rank, 1, MPI_INT, MPI_ANY_SOURCE, CONSENSUS_READY_TAG, *
                comm, &status);
584
585             survivors[status.MPI_SOURCE] = 1;
586             received++;
587
588             printf("***** PROCESSO DE CONSEN O ROOT ***** -> Recebido %d/%d
                tags\n", received, waiting_tags);
589         }
590
591         // Envia confirma o para todos os ranks que reportaram
592         for (int rank = 0; rank < mpi_size; rank++)
593         {
594             if (survivors[rank] && rank != root_rank)
595             {
596                 MPI_Send(&consensus_reached, 1, MPI_INT, rank, CONSENSUS_READY_TAG, *comm
                    );
597             }
598         }
599
600         free(survivors);
601         consensus_reached = 1;
602
603         printf("Finalizado processo de consen o.\n");
604     }
605     else
606     {
607         printf("[%d] Enviando tag...\n", mpi_rank);
608         // Ranks n o -ROOT: envia seu rank e aguarda confirma o
609         MPI_Send(&mpi_rank, 1, MPI_INT, 0, CONSENSUS_READY_TAG, *comm);
610
611         printf("[%d] Esperando confirma o...\n", mpi_rank);
612
613         MPI_Recv(&consensus_reached, 1, MPI_INT, root_rank, CONSENSUS_READY_TAG, *comm, &
            status);
614     }

```

```
615  
616     return 1;  
617 }  
618  
619 #endif // _UTILS_M_H
```


B IMPLEMENTAÇÃO: VERSÃO ADAPTATIVA (C/R ADAPTATIVO)

Listing B.1 – main.c – versão Adaptativa

```

1 // Header file inclusion.
2 #include <mpi.h>
3 #include <mpi-ext.h>
4 #include <stdint.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <sys/time.h>
8
9 #include <sys/types.h>
10 #include <sys/stat.h>
11 #include <unistd.h>
12 #include <signal.h>
13
14 #include <time.h>
15 #include "common.h"
16 #include "disk_utils.h"
17
18 #ifdef USE_OPENMP
19 #include <omp.h>
20 #endif
21
22 // Debug flag to track process type
23 static int IS SPAWNED_PROCESS = 0;
24
25 char** gargv = NULL;
26
27 int interaction, maxInteractions, ckpt_interaction; // Iteration
28 // MultiCheckpointData* MultiCheckpoint; // REMOVIDO
29 // CheckpointData* OwnCheckpoint; // REMOVIDO
30
31 MPI_Comm comm;
32
33 data_t i_mass;
34
35 par_t* param;
36 pop_t* f_aos_h;
37
38 int mpi_rank; // Rank of MPI node.
39 int mpi_size; // Size of MPI world communicator.
40
41 int mpi_grid_dims[2]; // Sizes of MPI ranks grid of world communicator.
42
43 int n_mpi; // total number of MPI ranks == n_mpi_x * n_mpi_y == mpi_size
44
45 int n_mpi_x; // number of MPI ranks along X direction
46 int n_mpi_y; // number of MPI ranks along Y direction
47
48 int mpi_rank_x; // MPI X coordinate.
49 int mpi_rank_y; // MPI Y coordinate.
50
51 int mpi_peer_left, mpi_peer_right, mpi_peer_top,
52     mpi_peer_bot; // Rank of neighboring MPI ranks
53
54 char* mpi_order; // MPI ranks order string
55

```

```

56     int MAX_FAILURE_SUPPORTED, SHOULD_CREATE_CHECKPOINT, CKPT_STEP, SHOULD_FAIL,
        FAILURE_ITERATION;
57     int GSIZE_X, GSIZE_Y; // Global Size X and Y of lattice
58     int LSIZE_X, LSIZE_Y; // Local Size X and Y of lattice
59     uint64_t NX, NY;      // X-length and Y-length of local lattice in memory
60
61     long int SEED; // seed for random generator
62
63     long int execution_id; // NOVO: ID nico da execu o
        para o path do checkpoint
64     int load_from_checkpoint_flag = 0; // NOVO: Flag para indicar o
        carregamento
65     static CheckpointData f1_data, f2_data;
66
67     const size_t OFF0[NPOP] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
68
69     size_t OFF[NPOP];
70
71     const int bits = 8 * sizeof(int) - 1;
72
73     ////////////////////////////////////////
74
75     #include "init.h"
76
77     ////////////////////////////////////////
78     // Lattice-Boltzmann Method.
79
80     #ifdef __cplusplus
81     extern "C"
82     {
83     #endif
84         int lbmMain();
85         CheckpointData receive_data_from_parent();
86         void initialize_lbm_buffers(int maxInteractions, MPI_Comm comm);
87         void calculate_neighbours();
88         void validate_grid(int n_mpi_x, int n_mpi_y, int n_mpi);
89         void initialize_offset();
90         void read_data_from_input_file();
91         int handle_mpi_failure_new(MPI_Comm* pcomm, int* errcode, ...);
92         void export_execution_time(const char* filename, double time_in_seconds);
93         int execute_lbm_spawned();
94     #ifdef __cplusplus
95     }
96     #endif
97
98     void blow_everything(MPI_Comm* comm)
99     {
100         MPI_Comm_set_errhandler(*comm, MPI_ERRORS_RETURN);
101
102         MPI_Barrier(*comm);
103
104         raise(SIGKILL);
105     }
106
107     int main(int argc, char* argv[])
108     {
109         f_aos_h = NULL;
110
111         int namelen;

```

```

112
113     MAX_FAILURE_SUPPORTED      = 1;
114     SHOULD_CREATE_CHECKPOINT = 1;
115     SHOULD_FAIL                = 1;
116     FAILURE_ITERATION          = 9;
117     CKPT_STEP                  = 2;
118     const int root             = 0;
119     char      processor_name[MPI_MAX_PROCESSOR_NAME];
120     gargv = argv;
121     param = NULL;
122
123     int should_create_checkpoint_override_value = 0;
124     int should_create_checkpoint_overridden = 0;
125
126     for (int i = 1; i < argc; i++)
127     {
128         if (strcmp(argv[i], "--load-checkpoint") == 0)
129         {
130             load_from_checkpoint_flag = 1;
131
132             // Verifica se um ID foi fornecido ap s a flag
133             // e se n o      outra flag (n o come a com '-')
134             if (i + 1 < argc && argv[i + 1][0] != '-')
135             {
136                 // Um ID foi fornecido, usa-o.
137                 execution_id = atol(argv[i + 1]);
138                 argv[i]      = "";
139                 argv[i + 1] = "";
140                 i++; // Pula o argumento do ID na pr xima itera o do loop
141             }
142             else
143             {
144                 // Nenhum ID foi fornecido, ativa o modo de detec o autom tica.
145                 // Usamos -1 como um sinalizador para o modo autom tico.
146                 execution_id = -1;
147                 argv[i]      = "";
148             }
149             break;
150         }
151         else if (strcmp(argv[i], "--should-create-checkpoint") == 0)
152         {
153             if (i + 1 < argc)
154             {
155                 int value = atoi(argv[i + 1]);
156                 if (value == 0 || value == 1)
157                 {
158                     should_create_checkpoint_override_value = value;
159                     should_create_checkpoint_overridden = 1;
160                     if (value == 0)
161                         printf("[ARG] Checkpoint desativado via linha de comando.\n");
162                     else
163                         printf("[ARG] Checkpoint for ado via linha de comando.\n");
164                 }
165                 else
166                 {
167                     fprintf(stderr, "[WARN] Valor inv lido para --should-create-checkpoint (use 0 ou 1).\n");
168                 }
169                 i++;
170             }

```

```

171     }
172 }
173
174 int    provided;
175 pid_t  pid = getpid();
176 char   hostname[256];
177 gethostname(hostname, sizeof(hostname));
178
179 printf("Process_%d_starting_in_host_%s\n", pid, hostname);
180 fflush(stdout);
181
182 MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
183 if (provided < MPI_THREAD_MULTIPLE)
184 {
185     fprintf(stderr, "Threading_support_level_is_lower_than_the_requested_one.\n");
186     MPI_Abort(MPI_COMM_WORLD, -1);
187 }
188
189 double starttime, endtime;
190 starttime = MPI_Wtime();
191
192 MPI_Comm parent;
193 MPI_Comm_get_parent(&parent);
194 int isChildProcess = parent != MPI_COMM_NULL;
195
196 // This is a new process, created by the ROOT to replace a failed one
197 if (isChildProcess)
198 {
199     execute_lbm_spawned();
200
201     return 0;
202 }
203
204 MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
205 MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
206 MPI_Get_processor_name(processor_name, &namelen);
207
208 // Cria ID de execu o ou
209 if (load_from_checkpoint_flag)
210 {
211     MPI_Bcast(&execution_id, 1, MPI_LONG, root, MPI_COMM_WORLD);
212 }
213 else
214 {
215     if (mpi_rank == root)
216     {
217         srand(time(NULL));
218         execution_id = rand();
219         printf("*****\n");
220         printf("Iniciando_nova_execu o_com_ID:%ld\n", execution_id);
221         printf("Para_reiniciar_do_checkpoint, use: --load-checkpoint_%ld\n",
222             execution_id);
223         printf("*****\n");
224     }
225
226     MPI_Bcast(&execution_id, 1, MPI_LONG, root, MPI_COMM_WORLD);
227
228     if (mpi_rank == root)
229     {
230         char checkpoint_dir[256];

```

```

230     snprintf(checkpoint_dir, sizeof(checkpoint_dir), "./checkpoints");
231     mkdir(checkpoint_dir, 0755);
232     snprintf(checkpoint_dir, sizeof(checkpoint_dir), "./checkpoints/%ld",
                execution_id);
233     mkdir(checkpoint_dir, 0755);
234 }
235 }
236
237 MPI_Barrier(MPI_COMM_WORLD);
238
239 if (mpi_rank == 0)
240 {
241     read_data_from_input_file();
242 }
243
244 if (should_create_checkpoint_overridden) {
245     SHOULD_CREATE_CHECKPOINT = should_create_checkpoint_override_value;
246 }
247
248 MPI_Bcast(&GSIZE_X, 1, MPI_INT, root, MPI_COMM_WORLD);
249 MPI_Bcast(&GSIZE_Y, 1, MPI_INT, root, MPI_COMM_WORLD);
250 MPI_Bcast(&maxInteractions, 1, MPI_INT, root, MPI_COMM_WORLD);
251 MPI_Bcast(&SEED, 1, MPI_LONG, root, MPI_COMM_WORLD);
252 MPI_Bcast(&MAX_FAILURE_SUPPORTED, 1, MPI_INT, root, MPI_COMM_WORLD);
253 MPI_Bcast(&SHOULD_CREATE_CHECKPOINT, 1, MPI_INT, root, MPI_COMM_WORLD);
254 MPI_Bcast(&CKPT_STEP, 1, MPI_INT, root, MPI_COMM_WORLD);
255 MPI_Bcast(&SHOULD_FAIL, 1, MPI_INT, root, MPI_COMM_WORLD);
256 MPI_Bcast(&FAILURE_ITERATION, 1, MPI_INT, root, MPI_COMM_WORLD);
257
258 printf("SHOULD_CREATE_CHECKPOINT: %d\n", SHOULD_CREATE_CHECKPOINT);
259
260 if (load_from_checkpoint_flag)
261 {
262     if (execution_id == -1)
263     {
264         if (mpi_rank == root)
265         {
266             printf("ID de execu o o fornecido. Procurando pelo checkpoint mais
                recente...\n");
267             execution_id = find_latest_checkpoint_execution_id("./checkpoints");
268         }
269         // Rank 0 transmite o ID encontrado (ou -1 se nada foi achado) para todos.
270         MPI_Bcast(&execution_id, 1, MPI_LONG, root, MPI_COMM_WORLD);
271
272         if (execution_id == -1)
273         {
274             if (mpi_rank == root)
275             {
276                 fprintf(stderr, "ERRO: Nenhum checkpoint encontrado para carregar
                automaticamente.\n");
277             }
278             MPI_Abort(MPI_COMM_WORLD, -1);
279         }
280         else
281         {
282             if (mpi_rank == root)
283             {
284                 printf("Checkpoint mais recente encontrado. Carregando ID de
                execu o : %ld\n", execution_id);
285             }

```

```

286     }
287 }
288
289 if (mpi_rank == root)
290 {
291     printf("Tentando carregar do checkpoint para o ID de execu  o %ld\n",
292            execution_id);
293 }
294
295 CheckpointData loaded_meta;
296 interaction = load_checkpoint_from_disk(execution_id, mpi_rank, &loaded_meta, &
297 f_aos_h, &param, &f1_data, &f2_data);
298
299 if (interaction < 0)
300 {
301     if (mpi_rank == root)
302     {
303         fprintf(stderr, "Falha ao carregar o checkpoint. Abortando.\n");
304     }
305
306     MPI_Abort(MPI_COMM_WORLD, -1);
307 }
308
309 // Atualiza as vari veis globais com os metadados carregados
310 GSIZEEX = loaded_meta.GSIZEEX;
311 GSIZEY = loaded_meta.GSIZEY;
312 SEED = loaded_meta.SEED;
313 maxInteractions = loaded_meta.maxInteractions;
314 i_mass = loaded_meta.i_mass;
315 n_mpi_x = loaded_meta.n_mpi_x;
316 n_mpi_y = loaded_meta.n_mpi_y;
317 LSIZEEX = loaded_meta.LSIZEEX;
318 LSIZEY = loaded_meta.LSIZEY;
319 NX = loaded_meta.NX;
320 NY = loaded_meta.NY;
321 ckpt_interaction = interaction;
322
323 if (mpi_rank == root)
324 {
325     printf("Estado carregado com sucesso da itera  o %d. Retomando...\n",
326            interaction);
327
328     FILE *fp = fopen("recovery_timestamp.log", "w");
329     if (fp != NULL) {
330         time_t now = time(NULL);
331         fprintf(fp, "%ld", now); // Escreve o tempo em segundos desde a poca
332             Unix
333         fclose(fp);
334     }
335 }
336
337 // Cria o do manipulador de erros
338 MPI_Errhandler errh;
339 MPI_Comm_create_errhandler((MPI_Comm_errhandler_function*)handle_mpi_failure_new, &
340 errh);
341 MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);
342
343 n_mpi = mpi_size;
344
345 // create a cartesian grid

```

```

341 MPI_Dims_create(n_mpi, 2, mpi_grid_dims);
342
343 n_mpi_x = mpi_grid_dims[0];
344 n_mpi_y = mpi_grid_dims[1];
345
346 calculate_neighbours();
347
348 validate_grid(n_mpi_x, n_mpi_y, n_mpi);
349
350 for (int i = 0; i < n_mpi; i++)
351 {
352     if (mpi_rank == i)
353     {
354         printf("Rank_%d/%d (PID=%d) in Host %s => Coordenada (%d,%d)\n", mpi_rank,
355             n_mpi - 1, pid, hostname, mpi_rank_x, mpi_rank_y);
356         fflush(stdout);
357     }
358 }
359
360 for (int i = 0; i < n_mpi; i++)
361 {
362     if (mpi_rank == i)
363     {
364         printf("Rank_%d/%d => Vizinhos T/R/B/L => %d,%d,%d,%d\n", mpi_rank, n_mpi
365             - 1, mpi_peer_top, mpi_peer_right, mpi_peer_bot,
366             mpi_peer_left);
367         fflush(stdout);
368     }
369 }
370
371 MPI_Barrier(MPI_COMM_WORLD);
372
373 if (!load_from_checkpoint_flag)
374 {
375     LSIZE_X = (GSIZE_X / n_mpi_x); // Local lattice size.
376     LSIZE_Y = (GSIZE_Y / n_mpi_y); // Local lattice size.
377
378     // add reminder to right-most ranks of grid if GSIZE_X is not multiple of n_mpi_x
379     if (mpi_rank_x == (n_mpi_x - 1))
380     {
381         LSIZE_X += (GSIZE_X % n_mpi_x);
382     }
383
384     // add reminder to top-most ranks of grid if GSIZE_Y is not multiple of n_mpi_y
385     if (mpi_rank_y == (n_mpi_y - 1))
386     {
387         LSIZE_Y += (GSIZE_Y % n_mpi_y);
388     }
389
390     NX = (HX + LSIZE_X + HX); // X-length of lattice in memory.
391     NY = (HY + LSIZE_Y + HY); // X-length of lattice in memory.
392
393     // Initialize the OFF array
394     initialize_offset();
395
396     printf("[MPI%04d]: GSIZE_X: %d, GSIZE_Y: %d (0 GB) NMPIX: %d NMPIY: %d LSIZE_X: %d
397         LSIZE_Y: %d NX: %ld NY: %ld\n", mpi_rank, GSIZE_X, GSIZE_Y,
398         n_mpi_x, n_mpi_y, LSIZE_X, LSIZE_Y, NX, NY);
399
400 MPI_Barrier(MPI_COMM_WORLD);

```

```

398
399 // Population lattice is allocated on the host.
400 posix_memalign((void**)&f_aos_h, 4096, NX * NY * sizeof(pop_t));
401
402 if (f_aos_h == NULL)
403 {
404     fprintf(stderr, "[main]_ERROR:_p_mm_malloc_failed\n");
405     MPI_Abort(MPI_COMM_WORLD, -1);
406 }
407
408 posix_memalign((void**)&param, 4096, sizeof(par_t));
409
410 if (param == NULL)
411 {
412     fprintf(stderr, "[main]_ERROR:_param_mm_malloc_failed\n");
413     MPI_Abort(MPI_COMM_WORLD, -1);
414 }
415
416 memset(f_aos_h, 0, NX * NY * sizeof(pop_t));
417
418 // Lattice initialization.
419 init(param, f_aos_h, SEED);
420
421 ckpt_interaction = 0;
422 interaction      = 0;
423 }
424 else
425 {
426     initialize_offset();
427 }
428
429 // Initialize lbm
430 initialize_lbm_buffers(maxInteractions, MPI_COMM_WORLD);
431
432 // Lattice-Boltzmann Method function is called.
433 lbmMain();
434
435 endtime = MPI_Wtime();
436
437 if (mpi_rank == 0)
438 {
439     export_execution_time("exec_times.log", endtime - starttime);
440 }
441
442 blow_everything(&comm);
443
444 // Progressive finalization
445 printf("Rank%d/%d on host %s shut down!\n", mpi_rank, n_mpi - 1, hostname);
446
447 return 0;
448 }
449
450 int execute_lbm_spawned()
451 {
452
453     pid_t pid = getpid();
454     char hostname[256];
455     gethostname(hostname, sizeof(hostname));
456
457     // Mark as spawned process for debug

```

```
1  #include <stdio.h>
```

```

2  #include <stdlib.h>
3  #include <time.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <sys/wait.h>
7
8  // --- FUNÇÃO AUXILIAR PARA MEDIÇÃO DE TEMPO DE ALTA PRECISÃO ---
9  // Calcula a duração em segundos (com casas decimais) entre dois pontos no tempo.
10 double calculate_duration(struct timespec start, struct timespec end) {
11     long seconds = end.tv_sec - start.tv_sec;
12     long nanoseconds = end.tv_nsec - start.tv_nsec;
13     // Ajusta o cálculo caso os nanossegundos "virem" o segundo
14     if (nanoseconds < 0) {
15         seconds--;
16         nanoseconds += 1000000000;
17     }
18     return (double)seconds + (double)nanoseconds / 1.0e9;
19 }
20
21 int main(int argc, char* argv[])
22 {
23     struct timespec total_start_time, failure_time, total_end_time;
24     struct timespec recovery_start_from_log; // Para o timestamp lido do arquivo
25     struct timespec recovery_start_oversubscribe_time;
26     struct timespec node_restored_time;
27     struct timespec relaunch_start_time;
28
29     printf("ORQUESTRADOR: Iniciando teste de resiliência...\n");
30
31     // Compila e aplica o MPI antes de começar
32     system("mpicc -fopenmp -DUSE_OPENMP -g main.c lbm.c disk_utils.c -lm -o main");
33
34     // Garante que não haja logs de recuperação antigos
35     remove("recovery_timestamp.log");
36     remove("checkpoint_times.log");
37     remove("./checkpoint_and_kill");
38     system("rm -rf checkpoints");
39     system("mkdir checkpoints");
40
41     // Marca o início do cenário completo
42     clock_gettime(CLOCK_REALTIME, &total_start_time);
43
44     printf("ORQUESTRADOR: Criando processo filho para induzir falha assíncrona...\n");
45
46     pid_t pid = fork();
47
48     if (pid < 0) {
49         // Erro ao criar o processo filho
50         perror("fork falhou");
51         exit(1);
52     }
53     else if (pid == 0)
54     {
55         // --- CÍDIGO DO PROCESSO FILHO ---
56         // A única responsabilidade deste processo é esperar e causar a falha.
57
58         printf("PROCESSO FILHO (PID: %d): Aguardando 20s para induzir falha...\n",
59             getpid());
60         sleep(47);

```

```

60
61     printf("PROCESSO_FILHO: Desligando o n 10.0.0.13...\n");
62     system("./toolkit_cluster_test-failure--cluster-id_fofdCulh9SGtxVn9--node-
        private-ip_10.0.0.13-y");
63
64     printf("PROCESSO_FILHO: Falha induzida. Encerrando.\n");
65     exit(0); // O filho termina seu trabalho aqui.
66 }
67 else
68 {
69     // --- C DIGO DO PROCESSO PAI (ORQUESTRADOR PRINCIPAL) ---
70     printf("PROCESSO_PAI(PID: %d): Lanando e simula o MPI (bloqueante) o filho (
        PID: %d) ir induzir a falha.\n", getpid(), pid);
71
72     system("mpirun--hostfile_hostfile--with-ft_ulfm--map-by_ppr:1:node-x
        OMP_NUM_THREADS=8-x_OMP_PROC_BIND=true-np4./main");
73
74     // O tempo de recupera o come a AGORA, no momento em que o orquestrador
        detecta o erro.
75     clock_gettime(CLOCK_REALTIME, &failure_time);
76     printf("ORQUESTRADOR: Falha detectada (PRTE)! A aplica o MPI terminou.\n");
77     waitpid(pid, NULL, 0);
78
79     // ---- FASE 3: RECUPERA O PARALELA E EXECU O OVERSUBSCRIBED ----
80     printf("ORQUESTRADOR: Iniciando recupera o de n e execu o com
        oversubscribe_EM_PARALELO.\n");
81
82     pid_t recovery_pid = fork();
83     if (recovery_pid < 0) {
84         perror("fork para recupera o falhou");
85         exit(1);
86     }
87     else if (recovery_pid == 0)
88     {
89         sleep(40);
90
91         // --- C DIGO DO PROCESSO FILHO (TASK A: RECUPERA O DO N ) ---
92         printf("[FILHO-RECUPERA O (PID: %d)]: Iniciando processo de recupera o
            do n ... \n", getpid());
93         int status;
94
95         while (1) {
96             printf("[FILHO-RECUPERA O]: Tentando restaurar o n 10.0.0.13...\n");
97             status = system("./toolkit_cluster_spawn--cluster-id_fofdCulh9SGtxVn9-y
                ");
98
99             if (WIFEXITED(status) && WEXITSTATUS(status) == 0) {
100                 printf("[FILHO-RECUPERA O]: N recuperado com sucesso!\n");
101
102                 clock_gettime(CLOCK_REALTIME, &node_restored_time);
103                 FILE* f = fopen("node_restored_timestamp.log", "w");
104                 if (f) {
105                     fprintf(f, "%ld", node_restored_time.tv_sec);
106                     fclose(f);
107                 }
108
109                 break;
110             } else {
111                 int exit_status = WIFEXITED(status) ? WEXITSTATUS(status) : -1;
112                 fprintf(stderr, "[FILHO-RECUPERA O]: Falha ao restaurar o n (

```

```

113         status: %d). Tentando em 5s...\n", exit_status);
114         sleep(5);
115     }
116 }
117
118 // N  recuperado. Agora, sinaliza para a execu  o 'oversubscribed' parar.
119 printf("[FILHO-RECUPERA  0]: Criando arquivo './checkpoint_and_kill' para
120         finalizar a execu  o oversubscribed.\n");
121 FILE* f = fopen("./checkpoint_and_kill", "w");
122 if (f) {
123     fprintf(f, "stop");
124     fclose(f);
125 } else {
126     perror("[FILHO-RECUPERA  0]: ERRO ao criar arquivo kill");
127 }
128
129 printf("[FILHO-RECUPERA  0]: Encerrando.\n");
130 exit(0);
131 }
132 else
133 {
134     sleep(5);
135
136     // --- C DIGO DO PROCESSO PAI (TASK B: EXECU  O OVERSUBSCRIBED) ---
137     printf("[PAI-ORQUESTADOR(PID: %d)]: Lan ando simula  o MPI com --
138             oversubscribe (filho recupera  o: %d).\n", getpid(), recovery_pid);
139
140     printf("[PAI-ORQUESTADOR]: Lan ando mpirun com hostfile degraded...\n");
141
142     system("cat hostfile degraded");
143
144     clock_gettime(CLOCK_REALTIME, &recovery_start_oversubscribe_time);
145     // Lan a a simula  o com oversubscribe, USANDO O NOVO HOSTFILE DEGRADADO.
146     // Esta chamada bloqueante e s  retornar quando o mpirun terminar.
147     system("mpirun --oversubscribe --map-by node --hostfile hostfile degraded --
148             display-map -x OMP_NUM_THREADS=8 -x OMP_PROC_BIND=true -np 4 ./main --
149             should-create-checkpoint 0 --load-checkpoint");
150
151     printf("[PAI-ORQUESTADOR]: Execu  o oversubscribed conclu da (sinal 'kill
152             ' recebido).\n");
153
154     // Espera pelo filho (recupera  o do n ) para garantir que ele terminou
155     waitpid(recovery_pid, NULL, 0);
156     printf("[PAI-ORQUESTADOR]: Processo de recupera  o do n  confirmado como
157             conclu do.\n");
158
159     // O arquivo 'checkpoint_and_kill' deve ser removido pela pr pria
160     // aplica  o MPI,
161     // mas removemos aqui para garantir.
162     remove("./checkpoint_and_kill");
163     remove("hostfile.degraded");
164 }
165
166 // ---- FASE 4: RELAN AMENTO E CONTINUA  O (P S -OVERSUBSCRIBE) ----
167 printf("ORQUESTADOR: Lan ando a simula  o MPI novamente (sem oversubscribe)
168         para recuperar do checkpoint final...\n");
169 printf("ORQUESTADOR: Aguardando 5 segundos antes de lan ar mpirun final...\n");
170
171 clock_gettime(CLOCK_REALTIME, &relaunch_start_time);
172 // Lan a a simula  o final, com a flag --load-checkpoint

```

```

164 system("mpirun--hostfile_hostfile--map-by_ppr:1:node--with-ft_ulfm-x_
OMP_NUM_THREADS=8-x_OMP_PROC_BIND=true-np4./main--load-checkpoint");
165
166 printf("ORQUESTADOR: A simulacao recuperada foi concluida.\n");
167
168 // ---- FASE 5: CALCULO DOS RESULTADOS ----
169 clock_gettime(CLOCK_REALTIME, &total_end_time);
170
171 FILE* fp = fopen("recovery_timestamp.log", "r");
172 if (fp == NULL) {
173     fprintf(stderr, "ORQUESTADOR: ERRO! Nao foi possivel ler o arquivo de log
'recovery_timestamp.log'.\n");
174     return 1;
175 }
176
177 FILE* fnode = fopen("node_restored_timestamp.log", "r");
178 if (fnode) {
179     fscanf(fnode, "%ld", &node_restored_time.tv_sec);
180     node_restored_time.tv_nsec = 0;
181     fclose(fnode);
182 } else {
183     fprintf(stderr, "[WARN] Nao foi possivel ler node_restored_timestamp.log\n"
);
184 }
185
186 // Ler o timestamp (em segundos) que a applicacao MPI escreveu
187 fscanf(fp, "%ld", &recovery_start_from_log.tv_sec);
188 recovery_start_from_log.tv_nsec = 0; // Zera os nanossegundos para consistencia
189 fclose(fp);
190
191 // Calcula duracoes com alta precisao
192 double total_duration = calculate_duration(total_start_time, total_end_time);
193 double recovery_duration = calculate_duration(failure_time,
recovery_start_from_log);
194 double t1_desfalcado = calculate_duration(failure_time,
recovery_start_oversubscribe_time);
195 double t2_aws = calculate_duration(recovery_start_oversubscribe_time,
node_restored_time);
196 double t3_reintegracao = calculate_duration(node_restored_time,
relaunch_start_time);
197
198 // Formata os timestamps para uma exibicao legivel
199 char start_str[30], failure_str[30], recovery_str[30], end_str[30];
200 strftime(start_str, sizeof(start_str), "%Y-%m-%d_%H:%M:%S", localtime(&
total_start_time.tv_sec));
201 strftime(failure_str, sizeof(failure_str), "%Y-%m-%d_%H:%M:%S", localtime(&
failure_time.tv_sec));
202 strftime(recovery_str, sizeof(recovery_str), "%Y-%m-%d_%H:%M:%S", localtime(&
recovery_start_from_log.tv_sec));
203 strftime(end_str, sizeof(end_str), "%Y-%m-%d_%H:%M:%S", localtime(&total_end_time
.tv_sec));
204
205 printf("\n\n===== \n");
206 printf("RELATÓRIO DO TESTE DE RESILIÊNCIA\n");
207 printf("----- \n");
208 printf("Início do cenrio.....: %s\n", start_str);
209 printf("Momento da falha (PRTE): %s\n", failure_str);
210 printf("Simulacao retomada em.....: %s\n", recovery_str);
211 printf("Fim do cenrio.....: %s\n", end_str);
212 printf("T1 - Tempo at o voltar ao estado normal.....: %.4f s\n", t1_desfalcado);

```

```

    );
213 printf("UU>T2-Tempo at AWS devolver inst ncia.....:%.4f\us\n", t2_aws);
214 printf("UU>T3-Tempo de re integra o normal.....:%.4f\us\n",
    t3_reintegracao);
215 printf("UU>T_total(falha fim).....:%.4f\us\n",
    total_duration);
216 printf("-----\n");
217 printf("UU>Tempo total de execu o do cen rio...:%.4f\us segundos\n",
    total_duration);
218 printf("UU>Tempo de recupera o (da falha retomada):%.4f\us segundos\n",
    recovery_duration);
219 printf("UU>Tempo de checkpoint:\n");
220 system("python3 average.py ./checkpoint_times.log");
221 printf("=====\n");
222 }
223
224 return 0;
225 }

```

C IMPLEMENTAÇÃO: VERSÃO OVERSUBSCRIBE (TOLERÂNCIA A FALHAS)

Listing C.1 – lbm.c – versão Oversubscribe

```

1  #include <math.h>
2  #include <mpi.h>
3  #include <mpi-ext.h>
4  #include <signal.h>
5  #include <stdint.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <sys/time.h>
9  #include <unistd.h>
10 #include "common.h"
11 #include <setjmp.h>
12
13 // Variável global para o contexto de salto
14 static jmp_buf recovery_point;
15 static int verbose = 1;
16
17 extern int mpi_rank; // Rank of MPI node.
18
19 extern int n_mpi; // total number of MPI ranks == n_mpi_x * n_mpi_y == mpi_size
20
21 extern int n_mpi_x; // number of MPI ranks along X direction
22 extern int n_mpi_y; // number of MPI ranks along Y direction
23
24 extern int mpi_rank_x; // MPI X coordinate.
25 extern int mpi_rank_y; // MPI Y coordinate.
26
27 extern int mpi_rank; // Rank of MPI node.
28 extern int mpi_size; // Size of MPI world communicator.
29
30 extern int n_mpi; // total number of MPI ranks == n_mpi_x * n_mpi_y == mpi_size
31
32 extern int n_mpi_x; // number of MPI ranks along X direction
33 extern int n_mpi_y; // number of MPI ranks along Y direction
34
35 extern int mpi_rank_x; // MPI X coordinate.
36 extern int mpi_rank_y; // MPI Y coordinate.
37
38 extern int mpi_peer_left, mpi_peer_right, mpi_peer_top,
39         mpi_peer_bot; // Rank of neighboring MPI ranks
40
41 extern char* mpi_order; // MPI ranks order string
42
43 extern data_t i_mass;
44
45 extern int          interaction, maxInteractions, ckpt_interaction; // Iteration
46 extern MultiCheckpointData* MultiCheckpoint;
47 extern CheckpointData*      OwnCheckpoint;
48
49 // Communicators
50 extern MPI_Comm comm;
51 extern char**   gargv; // Program and args
52
53 // Buffers to neighbours communication
54 static data_t *topSndBuf = NULL, *topRxBuf = NULL, *botSndBuf = NULL, *botRxBuf = NULL;

```

```

55 static data_t *leftSndBuf = NULL, *leftRxBuf = NULL, *rightSndBuf = NULL, *rightRxBuf =
    NULL;
56
57 // Buffers with all data
58 static CheckpointData f1_data, f2_data;
59
60 // Parameters of execution
61 extern par_t* param;
62 extern pop_t* f_aos_h;
63
64 extern int      MAX_FAILURE_SUPPORTED;
65 extern int      SHOULD_CREATE_CHECKPOINT;
66 extern int      SHOULD_FAIL;
67 extern int      FAILURE_ITERATION;
68 extern int      CKPT_STEP;
69 extern int      GSIZE_X, GSIZE_Y; // global lattice size_x and size_y
70 extern int      LSIZE_X, LSIZE_Y; // local lattice size_x and size_y
71 extern uint64_t NX, NY;           // x-length and y-length of lattice in memory.
72
73 extern long int SEED; // seed for random generator
74
75 extern const size_t OFF0[37];
76
77 extern size_t OFF[NPOP];
78
79 extern const int bits;
80
81 static int ROOT_RANK      = 0;
82 static int HAS_RECOVERED = 0;
83
84 int      lbmMain();
85 void      check_mass(CheckpointData* f2_data, pop_t* f_aos_h, int checkID, int
    mpi_rank, data_t i_mass, MPI_Comm* comm, int debug);
86 void      create_checkpoint_data_type(MPI_Datatype* checkpoint_type);
87 MultiCheckpointData* save_checkpoint_distributed(CheckpointData* own_data, int mpi_rank,
    int mpi_size, MPI_Comm comm);
88 void      initialize_lbm_buffers(int niter, MPI_Comm comm);
89 void      communicate_with_neighbours(int interaction);
90 int      handle_mpi_failure_new(MPI_Comm* pcomm, int* errcode, ...);
91
92 #include "utils.h"
93 #include "bc_kernel.h"
94 #include "collide_kernel.h"
95 #include "packunpack.h"
96 #include "propagate_kernel.h"
97
98 int lbmMain()
99 {
100     // Ponto a ser restaurado ap s recupera o de falha
101     if (setjmp(recovery_point) != 0)
102     {
103         //printf("[RECUPERA 0][R%d] - Conclu da!\n", mpi_rank);
104     }
105
106     double interactionStartTime, interactionEndTime, interactionExecTime;
107     interactionStartTime = MPI_Wtime();
108     while (interaction < maxInteractions)
109     {
110         interactionStartTime = MPI_Wtime();
111         interaction++;

```

```

112
113     int real_rank;
114     MPI_Comm_rank(comm, &real_rank);
115     pid_t pid = getpid();
116
117     if (mpi_rank == 0)
118     {
119         printf("[R=%d-RR=%d-PID=%d]-Starting iteration: %d. Last took %f\n",
120                mpi_rank, real_rank, pid, interaction, interactionExecTime);
121     }
122
123     if (!HAS_RECOVERED && (mpi_rank == mpi_size - 1) && interaction ==
124         FAILURE_ITERATION && SHOULD_FAIL)
125     {
126         printf("Rank %d se matando na itera o %d\n", mpi_rank, interaction);
127         raise(SIGKILL);
128     }
129
130     communicate_with_neighbours(interaction);
131
132     // Create checkpoints if required
133     if (interaction % CKPT_STEP == 0 && mpi_size > 1 && SHOULD_CREATE_CHECKPOINT)
134     {
135         double starttime, endtime;
136         starttime = MPI_Wtime();
137
138         // Cria novo checkpoint distribu do
139         MultiCheckpointData* new_checkpoint = save_checkpoint_distributed(&f2_data,
140                                 mpi_rank, mpi_size, comm);
141
142         // Check if everything is ok again
143         establish_simple_consensus(&comm, mpi_rank, mpi_size, ROOT_RANK, 0, 0);
144
145         if (mpi_rank == ROOT_RANK)
146         {
147             printf("*****Checkpoint criado com sucesso*****\n");
148         }
149
150         // Libera o checkpoint anterior se existir
151         if (MultiCheckpoint != NULL)
152         {
153             free_checkpoint_store(MultiCheckpoint);
154         }
155
156         // Atualiza o checkpoint atual
157         MultiCheckpoint = new_checkpoint;
158         ckpt_interaction = interaction;
159
160         endtime = MPI_Wtime();
161
162         if (mpi_rank == 0) {
163             export_execution_time("checkpoint_times.log", endtime - starttime);
164         }
165     }
166     MPI_Barrier(comm);
167
168     interactionEndTime = MPI_Wtime();
169     interactionExecTime = interactionEndTime - interactionStartTime;
170 }

```

```

169 // Print mass
170 check_mass(&f2_data, f_aos_h, -1, mpi_rank, i_mass, &comm, 1);
171
172 // Libera memoria
173 free(f1_data.buffer);
174 free(f2_data.buffer);
175 free(f_aos_h);
176 free(param);
177 free(topSndBuf);
178 free(topRxBuf);
179 free(botSndBuf);
180 free(botRxBuf);
181 free(leftSndBuf);
182 free(leftRxBuf);
183 free(rightSndBuf);
184 free(rightRxBuf);
185
186 return 0;
187 }
188
189 void communicate_with_neighbours(int interaction)
190 {
191     MPI_Request reqRxTopHalo, reqTxTopBorder, reqRxBotHalo, reqTxBotBorder;
192     MPI_Request reqRxLeftHalo, reqTxRightBorder, reqRxRightHalo, reqTxLeftHalo;
193     int tagUP = 1, tagDOWN = 2, tagLEFT = 3, tagRIGHT = 4;
194     long yoff = mpi_rank_y * (GSIYEY / n_mpi_y);
195
196     int has_multiple_rows = n_mpi_y > 1;
197     int has_top_neighbor = (mpi_rank_y < (n_mpi_y - 1));
198     int has_bot_neighbor = (mpi_rank_y > 0);
199
200     int communicate_with_top = has_top_neighbor && mpi_peer_top != mpi_rank;
201     int communicate_with_bot = has_bot_neighbor && mpi_peer_bot != mpi_rank;
202
203     if (has_multiple_rows)
204     {
205         if (communicate_with_top)
206         {
207             pack_top_border(topSndBuf, f2_data.buffer);
208         }
209
210         if (communicate_with_bot)
211         {
212             pack_bot_border(botSndBuf, f2_data.buffer);
213         }
214
215         if (communicate_with_top)
216         {
217
218             MPI_Irecv(topRxBuf, (15 + 8 + 3) * LSIZEY, MPI_DOUBLE, mpi_peer_top, tagDOWN,
219                     comm, &reqRxTopHalo);
220             MPI_Isend(topSndBuf, (15 + 8 + 3) * LSIZEY, MPI_DOUBLE, mpi_peer_top, tagUP,
221                     comm, &reqTxTopBorder);
222         }
223
224         if (communicate_with_bot)
225         {
226
227             MPI_Irecv(botRxBuf, (15 + 8 + 3) * LSIZEY, MPI_DOUBLE, mpi_peer_bot, tagUP,
228                     comm, &reqRxBotHalo);

```

```

226         MPI_Isend(botSndBuf, (15 + 8 + 3) * LSIZE_X, MPI_DOUBLE, mpi_peer_bot, tagDOWN
227             , comm, &reqTxBotBorder);
228     }
229     if (communicate_with_top)
230     {
231         MPI_Request requests[2] = {reqRxTopHalo, reqTxTopBorder};
232         int status = wait_for_mpi_requests_with_timeout(requests, 2, &
233             comm, mpi_rank, 1, 1, interaction);
234         if (status == MPI_ERR_TIMEOUT)
235         {
236             printf("R%d-Timeout_atingido\n", mpi_rank);
237         }
238         unpack_top_halo(f2_data.buffer, topRxBuf);
239     }
240
241     if (communicate_with_bot)
242     {
243         MPI_Request requests[2] = {reqRxBotHalo, reqTxBotBorder};
244         int status = wait_for_mpi_requests_with_timeout(requests, 2, &
245             comm, mpi_rank, 1, 2, interaction);
246         if (status == MPI_ERR_TIMEOUT)
247         {
248             printf("R%d-Timeout_atingido\n", mpi_rank);
249         }
250         unpack_bot_halo(f2_data.buffer, botRxBuf);
251     }
252 }
253
254 pack_right_border(leftSndBuf, f2_data.buffer, 0);
255 pack_left_border(rightSndBuf, f2_data.buffer);
256
257 // Run propagate and collide on Bulk
258 PROPAGATECOLLIDE(f1_data.buffer, f2_data.buffer, (HX + hx), (HX + LSIZE_X - hx), (HY +
259     hy + 1), (HY + LSIZE_Y - hy - 1), yoff);
260
261 // MPI-Comm Right-Border
262 MPI_Irecv(leftRxBuf, (15 + 8 + 3) * NY, MPI_DOUBLE, mpi_peer_left, tagLEFT, comm, &
263     reqRxLeftHalo);
264 MPI_Isend(leftSndBuf, (15 + 8 + 3) * NY, MPI_DOUBLE, mpi_peer_right, tagLEFT, comm, &
265     reqTxRightBorder);
266
267 // MPI-Comm Left-Border
268 MPI_Irecv(rightRxBuf, (15 + 8 + 3) * NY, MPI_DOUBLE, mpi_peer_right, tagRIGHT, comm,
269     &reqRxRightHalo);
270 MPI_Isend(rightSndBuf, (15 + 8 + 3) * NY, MPI_DOUBLE, mpi_peer_left, tagRIGHT, comm,
271     &reqTxLeftHalo);
272
273 MPI_Request requests[4] = {reqTxRightBorder, reqRxLeftHalo, reqRxRightHalo,
274     reqTxLeftHalo};
275 int status = wait_for_mpi_requests_with_timeout(requests, 4, &comm,
276     mpi_rank, 1, 3, interaction);
277 if (status == MPI_ERR_TIMEOUT)
278 {
279     printf("R%d-Timeout_atingido\n", mpi_rank);
280 }
281
282 // Wait right-border Comm and transfer back to device

```

```

276     unpack_left_halo(f2_data.buffer, leftRxBuf);
277
278     // Wait left-border Comm and transfer back to device
279     unpack_right_halo(f2_data.buffer, rightRxBuf);
280
281     // Propagate and collide on Left border
282     PROPAGATECOLLIDE(f1_data.buffer, f2_data.buffer, (HX), (HX + hx), (HY + hy + 1), (HY
        + LSIZEY - hy - 1), yoff);
283
284     // Propagate and collide on Right border
285     PROPAGATECOLLIDE(f1_data.buffer, f2_data.buffer, (HX + LSIZEX - hx), (HX + LSIZEX), (
        HY + hy + 1), (HY + LSIZEY - hy - 1), yoff);
286
287     // Rank is the bottom one
288     if (mpi_rank_y == 0)
289     {
290         propagate(f1_data.buffer, f2_data.buffer, (HX), (HX + LSIZEX), (HY), (HY + hy +
            1));
291         bcBottom(f1_data.buffer, f2_data.buffer, param, (HX), (HX + LSIZEX));
292         COLLIDE(f1_data.buffer, f1_data.buffer, (HX), (HX + LSIZEX), (HY), (HY + hy + 1),
            yoff);
293     }
294     else
295     {
296         PROPAGATECOLLIDE(f1_data.buffer, f2_data.buffer, (HX), (HX + LSIZEX), (HY), (HY +
            hy + 1), yoff);
297     }
298
299     // Rank is the top one
300     if (mpi_rank_y == (n_mpi_y - 1))
301     {
302         propagate(f1_data.buffer, f2_data.buffer, (HX), (HX + LSIZEX), (HY + LSIZEY - hy
            - 1), (HY + LSIZEY));
303         bcTop(f1_data.buffer, f2_data.buffer, param, (HX), (HX + LSIZEX));
304         COLLIDE(f1_data.buffer, f1_data.buffer, (HX), (HX + LSIZEX), (HY + LSIZEY - hy -
            1), (HY + LSIZEY), yoff);
305     }
306     else
307     {
308         PROPAGATECOLLIDE(f1_data.buffer, f2_data.buffer, (HX), (HX + LSIZEX), (HY +
            LSIZEY - hy - 1), (HY + LSIZEY), yoff);
309     }
310
311     // Swap pointers
312     data_t* tmp = f1_data.buffer;
313     f1_data.buffer = f2_data.buffer;
314     f2_data.buffer = tmp;
315 }
316
317 int handle_mpi_failure_new(MPI_Comm* dcomm, int* errcode, ...)
318 {
319     double starttime, endtime;
320     starttime = MPI_Wtime();
321     int debug = 0;
322
323     char errstr[MPI_MAX_ERROR_STRING];
324     int len;
325     MPI_Error_string(*errcode, errstr, &len);
326
327     pid_t pid = getpid();

```

```

328
329 MPI_Comm scomm, // Intercomm dos vivos
330 icomm,          // The intercomm between old and new processes
331 mcomm;          // The intracom merged from icomm
332 MPI_Group group_f, group_c;
333 int             nf;
334 int rc, flag, rflag, i, nc, ns, nd, crank, srnk, drank, agreed;
335
336 sleep(10);
337
338 redo:
339     sleep(0.5);
340     MPI_Errhandler derrh;
341     MPI_Comm_get_errhandler(*dcomm, &derrh);
342
343     // Set the error handler
344     MPI_Comm_set_errhandler(*dcomm, MPI_ERRORS_RETURN);
345     MPIX_Comm_failure_ack(*dcomm);
346     MPIX_Comm_failure_get_acked(*dcomm, &group_f);
347     MPI_Group_size(group_f, &nf);
348
349     // Check if checkpoints are available
350     verify_checkpoints_integrity(nf);
351
352     printf("[RANK=%d PID=%d] Handling (%d) Failures: %s.\n", mpi_rank, pid, nf, errstr);
353
354     // Execute simple consensus between the alives
355     //establish_simple_consensus(dcomm, mpi_rank, mpi_size, ROOT_RANK, nf, 1);
356
357     // Identifica os ranks falhos
358     int* ranks_gf = (int*)malloc(nf * sizeof(int));
359     int* ranks_gc = (int*)malloc(nf * sizeof(int));
360     MPI_Comm_group(*dcomm, &group_c);
361     for (int i = 0; i < nf; i++)
362     {
363         ranks_gf[i] = i;
364     }
365     MPI_Group_translate_ranks(group_f, nf, ranks_gf, group_c, ranks_gc);
366
367     // Check if checkpoints are available
368     verify_checkpoints_integrity(nf);
369
370     // printf("[RECUPERA 0][%d] - %d ranks falharam. Iniciando recupera o...\n",
371         mpi_rank, nf);
372
373     MPI_Info info = build_spawn_info();
374
375     MPIX_Comm_revoke(*dcomm);
376
377     // Create comm SComm locally with the alives
378     rc = MPIX_Comm_shrink(*dcomm, &scomm);
379     agreed = agree_communications(&scomm, rc, 0);
380     if (!agreed)
381     {
382         if (rc == MPI_SUCCESS) MPIX_Comm_revoke(scomm);
383         if (verbose) fprintf(stderr, "%04d: Failed agree in Shrink, redo\n", mpi_rank);
384         goto redo;
385     }
386

```

```

387     MPI_Comm_free(dcomm);
388
389     // Cria novos ranks para substituir os falhos
390     rc = MPI_Comm_spawn(gargv[0], &gargv[1], nf, info, ROOT_RANK, scomm, &icomm,
        MPI_ERRCODES_IGNORE);
391
392     agreed = agree_communications(&scomm, rc, 1);
393     if (!agreed)
394     {
395         if (rc == MPI_SUCCESS) {
396             MPIX_Comm_revoke(icomm);
397             MPI_Comm_free(&icomm);
398         }
399         MPI_Comm_free(&scomm);
400         if (verbose) fprintf(stderr, "%04d: Failed to agree in Spawn, redo\n", mpi_rank);
401
402         goto redo;
403     }
404
405     // ROOT RANK vai enviar o rank antigo para cada sobrevivente
406     if (mpi_rank == ROOT_RANK)
407     {
408         for (int i = 0; i < nf; i++)
409         {
410             int failed_rank = ranks_gc[i];
411
412             //printf("ROOT enviando rank %d para novo processo\n", failed_rank);
413
414             // 1. Envia o rank (failed_rank) que o novo processo (i) deve assumir
415             MPI_Send(&failed_rank, 1, MPI_INT, i, TAG_RANK_ASSIGN, icomm);
416         }
417     }
418
419     // D free no antigo err handler de dcomm
420     MPI_Errhandler_free(&derrh);
421
422     // Faz merge dos comunicadores com os child colocando os ranks parent no inicio
423     MPI_Barrier(icomm);
424
425     rc = MPI_Intercomm_merge(icomm, 0, &mcomm);
426
427     agreed = agree_communications(&scomm, rc, -1);
428
429     MPIX_Comm_revoke(scomm);
430     MPI_Comm_free(&scomm);
431     MPIX_Comm_revoke(icomm);
432     MPI_Comm_free(&icomm);
433
434     if (!agreed) {
435         if( MPI_SUCCESS == rc ) {
436             //MPI_Comm_free(&mcomm);
437         }
438         if( verbose ) fprintf(stderr, "%04d: Intercomm_merge failed, redo\n", mpi_rank);
439         goto redo;
440     }
441
442     //printf("Splitting parent comm\n");
443
444     // Reorder mcomm based on its original rank ordering in old comm
445     rc = MPI_Comm_split(mcomm, 1, mpi_rank, &comm);

```

```

446 agreed = agree_communications(&mcomm, rc, 2);
447 if (!agreed)
448 {
449     if (rc == MPI_SUCCESS) {
450         MPI_Comm_free(&comm);
451     }
452     if (verbose) fprintf(stderr, "%04d: Comm split failed, redo\n", mpi_rank);
453
454     goto redo;
455 }
456
457 MPIX_Comm_revoke(mcomm);
458 MPI_Comm_free(&mcomm);
459
460 // Para cada rank falho, determina qual rank sobrevivente deve restaurar os dados
461 for (int i = 0; i < nf; i++)
462 {
463     int failed_rank = ranks_gc[i];
464     int responsible_rank = -1;
465
466     // Encontra o rank respons vel pela recupera o (o mais pr ximo antes do
467     // falho)
468     int checkpointsCreated = MAX_FAILURE_SUPPORTED < (mpi_size - 1) ?
469     MAX_FAILURE_SUPPORTED : (mpi_size - 1);
470     for (int j = 1; j <= checkpointsCreated; j++)
471     {
472         int candidate = (failed_rank + j) % mpi_size; // Considera tamanho original
473
474         // Verifica se o candidato est vivo
475         int is_alive = 1;
476         for (int k = 0; k < nf; k++)
477         {
478             if (candidate == ranks_gc[k])
479             {
480                 is_alive = 0;
481                 break;
482             }
483         }
484
485         if (is_alive)
486         {
487             responsible_rank = candidate;
488             break;
489         }
490     }
491
492     if (responsible_rank == -1)
493     {
494         fprintf(stderr, "N o foi poss vel encontrar rank respons vel pela
495         recupera o do rank %d\n", failed_rank);
496         MPI_Abort(comm, -1);
497     }
498
499     // Se este o rank respons vel, faz a recupera o
500     if (mpi_rank == responsible_rank)
501     {
502         // Encontra o ndice no checkpoint_store onde est o os dados do rank falho
503         int ckpt_index = (responsible_rank - failed_rank + mpi_size) % mpi_size - 1;
504
505         if (ckpt_index >= 0 && ckpt_index < MultiCheckpoint->available_checkpoints)

```

```

503 {
504     // Obt m os dados do checkpoint do rank falho
505     CheckpointData* failed_data = &MultiCheckpoint->neighbor_data[ckpt_index
506         ];
507     if (debug) printf("[RECUPERA 0]-Rank%denviando metadados para%d
508         ...\n", mpi_rank, i);
509     // Cria tipo MPI personalizado
510     MPI_Datatype checkpoint_type;
511     create_checkpoint_data_type(&checkpoint_type);
512     // 2. Envia os metadados
513     MPI_Send(failed_data, 1, checkpoint_type, failed_rank, TAG_STRUCT_DATA,
514         comm);
515     MPI_Type_free(&checkpoint_type);
516     if (debug) printf("[RECUPERA 0]-Rank%denviando buffer...\n",
517         mpi_rank);
518     // 3. Envia o buffer principal
519     MPI_Send(failed_data->buffer, failed_data->LENGHT, MPI_DOUBLE,
520         failed_rank, TAG_BUFF_DATA, comm);
521     // 4. Envia o buffer f_aos_h
522     size_t f_aos_h_size = failed_data->NX * failed_data->NY * sizeof(pop_t);
523     MPI_Send(failed_data->f_aos_h_buffer, f_aos_h_size, MPI_BYTE, failed_rank
524         , TAG_AOS_DATA, comm);
525     if (debug) printf("[RECUPERA 0]-Rank%denviando params...\n",
526         mpi_rank);
527     // 5. Envia os par metros
528     MPI_Send(&failed_data->param_buffer, sizeof(par_t), MPI_BYTE, failed_rank
529         , TAG_PARAM_DATA, comm);
530     if (debug) printf("[RECUPERA 0]-Rank%denviando MAX_SUP...\n",
531         mpi_rank);
532     // 6. Envia o m ximo de failure
533     MPI_Send(&MAX_FAILURE_SUPPORTED, 1, MPI_INT, failed_rank,
534         TAG_MAX_SUPPORTED_FAILURES, comm);
535     // 7. Envia o m ximo de failure
536     MPI_Send(&CKPT_STEP, 1, MPI_INT, failed_rank, TAG_CKPT_STEP, comm);
537     // 8. Envia o m ximo de failure
538     MPI_Send(&SHOULD_CREATE_CHECKPOINT, 1, MPI_INT, failed_rank,
539         TAG_SHOULD_CREATE_CHECKPOINT, comm);
540     // 9. Envia o m ximo de failure
541     MPI_Send(&SHOULD_FAIL, 1, MPI_INT, failed_rank, TAG_SHOULD_FAIL, comm);
542     // 10. Envia o m ximo de failure
543     MPI_Send(&FAILURE_ITERATION, 1, MPI_INT, failed_rank,
544         TAG_FAILURE_ITERATION, comm);
545     if (debug) printf("[RECUPERA 0]-Rank%denviou dados para o novo
546         rank%d(substituindo rank%d)\n", mpi_rank, i, failed_rank);
547 }

```

```

550         else
551         {
552             fprintf(stderr, "R[%d] ndice de checkpoint inválido para rank %d\n", mpi_rank, failed_rank);
553             MPI_Abort(comm, -1);
554         }
555     }
556 }
557
558 // Todos os ranks recuperam o estado a partir do checkpoint local
559 copy_data_from_to(&MultiCheckpoint->own_data, &f2_data);
560
561 // Reset the interaction
562 interaction = ckpt_interaction;
563
564 // Clean up
565 free(ranks_gc);
566 free(ranks_gf);
567
568 MPI_Errhandler derrh2;
569 MPI_Comm_get_errhandler(comm, &derrh2);
570 MPI_Errhandler_free(&derrh2);
571
572 // Cria o manipulador de erros
573 MPI_Errhandler errh;
574 MPI_Comm_create_errhandler((MPI_Comm_errhandler_function*)handle_mpi_failure_new, &errh);
575 MPI_Comm_set_errhandler(comm, errh);
576
577 HAS_RECOVERED = 1;
578
579 // Establish consensus before continuing execution
580 establish_simple_consensus(&comm, mpi_rank, mpi_size, ROOT_RANK, 0, 0);
581
582 endtime = MPI_Wtime();
583
584 if (mpi_rank == 0) {
585     export_execution_time("recovery_times.log", endtime - starttime);
586 }
587
588 // Salta de volta para o ponto de recuperação em lbmMain()
589 longjmp(recovery_point, 1);
590 }
591
592 CheckpointData receive_data_from_parent()
593 {
594     MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
595
596     MPI_Status status;
597     MPI_Comm icomm, mcomm;
598     int flag, rflag, rc, agreed;
599     int debug = 1;
600
601 redo_child:
602     MPI_Comm_get_parent(&icomm);
603     // Recebe o rank que deve assumir
604     MPI_Recv(&mpi_rank, 1, MPI_INT, MPI_ANY_SOURCE, TAG_RANK_ASSIGN, icomm, &status);
605
606     MPI_Barrier(icomm);
607     if (debug) printf("Substituto para rank %d merging communicator...\n", mpi_rank);

```

```

608 // Merge with parent communicator
609
610 rc = MPI_Intercomm_merge(icomm, 1, &mcomm);
611
612 flag = (rc == MPI_SUCCESS);
613 MPIX_Comm_revoke(icomm);
614 MPI_Comm_free(&icomm);
615
616 if(!(flag)) {
617     if( MPI_SUCCESS == rc ) {
618         MPI_Comm_free(&mcomm);
619     }
620     if( verbose ) fprintf(stderr, "%04d: Intercomm_merge_failed, redo_child\n",
621         mpi_rank);
622     goto redo_child;
623 }
624
625 // Reorder
626 if (debug) printf("Child_Splitting_comm\n");
627 MPI_Comm_split(mcomm, 1, mpi_rank, &comm);
628 agreed = agree_communications(&mcomm, rc, 2);
629 if (!agreed)
630 {
631     if (rc == MPI_SUCCESS) {
632         MPI_Comm_free(&comm);
633     }
634     if (verbose) fprintf(stderr, "%04d: Comm_split_failed, redo_child\n", mpi_rank);
635     goto redo_child;
636 }
637
638 MPIX_Comm_revoke(mcomm);
639 MPI_Comm_free(&mcomm);
640
641 // Cria o tipo MPI para receber a estrutura
642 MPI_Datatype checkpoint_type;
643 create_checkpoint_data_type(&checkpoint_type);
644
645 pid_t pid = getpid();
646 if (debug) printf("PID(%d) recebeu rank %d do rank %d\n", pid, mpi_rank, status.
647     MPI_SOURCE);
648
649 // Recebe os metadados
650 CheckpointData received_data;
651 MPI_Recv(&received_data, 1, checkpoint_type, MPI_ANY_SOURCE, TAG_STRUCT_DATA, comm, &
652     status);
653
654 if (debug) printf("PID(%d) recebeu metadados\n", pid);
655
656 // Atualiza variaveis globais com os dados recebidos
657 GSIZE_X = received_data.GSIZE_X;
658 GSIZE_Y = received_data.GSIZE_Y;
659 SEED = received_data.SEED;
660 interaction = received_data.interaction;
661 maxInteractions = received_data.maxInteractions;
662 i_mass = received_data.i_mass;
663 n_mpi_x = received_data.n_mpi_x;
664 n_mpi_y = received_data.n_mpi_y;
665
666 // Atualiza as dimens es locais

```

```

665     LSIZE_X = received_data.LSIZE_X;
666     LSIZE_Y = received_data.LSIZE_Y;
667     NX      = received_data.NX;
668     NY      = received_data.NY;
669
670     // Aloca mem ria para os buffers
671     posix_memalign((void**)&f2_data.buffer, 4096, received_data.LENGHT * sizeof(data_t));
672     posix_memalign((void**)&f_aos_h, 4096, NX * NY * sizeof(pop_t));
673     posix_memalign((void**)&param, 4096, sizeof(par_t));
674
675     // Recebe o buffer principal
676     MPI_Recv(f2_data.buffer, received_data.LENGHT, MPI_DOUBLE, MPI_ANY_SOURCE,
        TAG_BUFF_DATA, comm, &status);
677
678     if (debug) printf("PID_(%d)_recebeu_buffer._Status_=_%d\n", pid, status.MPI_ERROR);
679
680     // Recebe o buffer f_aos_h
681     size_t f_aos_h_size = NX * NY * sizeof(pop_t);
682     MPI_Recv(f_aos_h, f_aos_h_size, MPI_BYTE, MPI_ANY_SOURCE, TAG_AOS_DATA, comm, &status
        );
683
684     if (debug) printf("PID_(%d)_recebeu_f_aos_h\n", pid);
685
686     // Recebe os par metros
687     MPI_Recv(param, sizeof(par_t), MPI_BYTE, MPI_ANY_SOURCE, TAG_PARAM_DATA, comm, &
        status);
688
689     if (debug) printf("PID_(%d)_recebeu_params\n", pid);
690
691     // Recebe o MAX FAILURE SUPOORTED
692     MPI_Recv(&MAX_FAILURE_SUPPORTED, 1, MPI_INT, MPI_ANY_SOURCE,
        TAG_MAX_SUPPORTED_FAILURES, comm, &status);
693
694     MPI_Recv(&CKPT_STEP, 1, MPI_INT, MPI_ANY_SOURCE, TAG_CKPT_STEP, comm, &status);
695
696     MPI_Recv(&SHOULD_CREATE_CHECKPOINT, 1, MPI_INT, MPI_ANY_SOURCE,
        TAG_SHOULD_CREATE_CHECKPOINT, comm, &status);
697
698     MPI_Recv(&SHOULD_FAIL, 1, MPI_INT, MPI_ANY_SOURCE, TAG_SHOULD_FAIL, comm, &status);
699
700     MPI_Recv(&FAILURE_ITERATION, 1, MPI_INT, MPI_ANY_SOURCE, TAG_FAILURE_ITERATION, comm,
        &status);
701
702     if (debug) printf("Novo_pid_%d_recebeu_os_par metros\n", pid);
703
704     // Configura a estrutura f2_data
705     f2_data.LSIZE_X = LSIZE_X;
706     f2_data.LSIZE_Y = LSIZE_Y;
707     f2_data.NX      = NX;
708     f2_data.NY      = NY;
709     f2_data.LENGHT  = received_data.LENGHT;
710     f2_data.allocated = 1;
711
712     f1_data          = f2_data;
713     f1_data.LSIZE_X = LSIZE_X;
714     f1_data.LSIZE_Y = LSIZE_Y;
715     f1_data.NX      = NX;
716     f1_data.NY      = NY;
717     f1_data.LENGHT  = received_data.LENGHT;
718     f1_data.allocated = 1;

```

```

719
720 // Cria c pia de f2_data para f1_data
721 posix_memalign((void*)&f1_data.buffer, 4096, f1_data.LENGHT * sizeof(data_t));
722 memcpy(f1_data.buffer, f2_data.buffer, f1_data.LENGHT * sizeof(data_t));
723
724 // Libera o tipo MPI
725 MPI_Type_free(&checkpoint_type);
726
727 // Aloca buffers de comunica o se n o foram alocados ainda
728 if (topSndBuf == NULL)
729 {
730     posix_memalign((void*)&topSndBuf, 4096, (15 + 8 + 3) * LSIZE * sizeof(data_t));
731     posix_memalign((void*)&topRxBuf, 4096, (15 + 8 + 3) * LSIZE * sizeof(data_t));
732     posix_memalign((void*)&botSndBuf, 4096, (15 + 8 + 3) * LSIZE * sizeof(data_t));
733     posix_memalign((void*)&botRxBuf, 4096, (15 + 8 + 3) * LSIZE * sizeof(data_t));
734     posix_memalign((void*)&leftSndBuf, 4096, (15 + 8 + 3) * NY * sizeof(data_t));
735     posix_memalign((void*)&leftRxBuf, 4096, (15 + 8 + 3) * NY * sizeof(data_t));
736     posix_memalign((void*)&rightSndBuf, 4096, (15 + 8 + 3) * NY * sizeof(data_t));
737     posix_memalign((void*)&rightRxBuf, 4096, (15 + 8 + 3) * NY * sizeof(data_t));
738 }
739
740 // Verifica aloca es
741 if ((topSndBuf == NULL) || (topRxBuf == NULL) || (botSndBuf == NULL) || (botRxBuf ==
742     NULL) || (leftSndBuf == NULL) || (leftRxBuf == NULL) ||
743     (rightSndBuf == NULL) || (rightRxBuf == NULL) || (f1_data.buffer == NULL) || (
744         f2_data.buffer == NULL))
745 {
746     fprintf(stderr, "ERROR: allocation of buffers failed\n");
747     MPI_Abort(comm, -1);
748 }
749
750 HAS_RECOVERED = 1;
751
752 // Establish consensus before continuing execution
753 establish_simple_consensus(&comm, mpi_rank, mpi_size, ROOT_RANK, 0, 0);
754
755 return received_data;
756 }
757
758 MultiCheckpointData* save_checkpoint_distributed(CheckpointData* own_data, int mpi_rank,
759     int mpi_size, MPI_Comm comm)
760 {
761     // Aloca a estrutura principal
762     MultiCheckpointData* checkpoint_store = (MultiCheckpointData*)malloc(sizeof(
763         MultiCheckpointData));
764     if (!checkpoint_store)
765     {
766         fprintf(stderr, "[%d] Erro ao alocar MultiCheckpointData\n", mpi_rank);
767         return NULL;
768     }
769
770 // Inicializa os campos
771 checkpoint_store->available_checkpoints = 0;
772 checkpoint_store->checkpoint_creation_result = 0; // Success
773 checkpoint_store->neighbor_data = NULL;
774
775 // Aloca espao para os ponteiros dos checkpoints de vizinhos (apenas se houver
776     mltiplos processos)
777 if (mpi_size > 1)
778 {

```

```

774     int checkpointsCreated      = MAX_FAILURE_SUPPORTED < (mpi_size - 1) ?
       MAX_FAILURE_SUPPORTED : (mpi_size - 1);
775     checkpoint_store->neighbor_data = (CheckpointData*)malloc(checkpointsCreated *
       sizeof(CheckpointData));
776     if (!checkpoint_store->neighbor_data)
777     {
778         fprintf(stderr, "[%d] Erro ao alocar array de CheckpointData\n", mpi_rank);
779         free(checkpoint_store);
780         return NULL;
781     }
782
783     // Inicializa os checkpoints de vizinhos
784     for (int i = 0; i < checkpointsCreated; i++)
785     {
786         checkpoint_store->neighbor_data[i].buffer = NULL;
787         checkpoint_store->neighbor_data[i].allocated = 0;
788     }
789 }
790
791 // Inicializa os campos
792 checkpoint_store->own_data.allocated = 1;
793 checkpoint_store->own_data.source_rank = mpi_rank;
794
795 // Preenche os campos adicionais das variaveis globais
796 checkpoint_store->own_data.mpi_rank = mpi_rank;
797 checkpoint_store->own_data.mpi_size = mpi_size;
798 checkpoint_store->own_data.GSIZEX = GSIZEX;
799 checkpoint_store->own_data.GSIZEY = GSIZEY;
800 checkpoint_store->own_data.SEED = SEED;
801 checkpoint_store->own_data.interaction = interaction;
802 checkpoint_store->own_data.maxInteractions = maxInteractions;
803 checkpoint_store->own_data.i_mass = i_mass;
804 checkpoint_store->own_data.n_mpi_x = n_mpi_x;
805 checkpoint_store->own_data.n_mpi_y = n_mpi_y;
806 checkpoint_store->own_data.LSIZEX = own_data->LSIZEX;
807 checkpoint_store->own_data.LSIZEY = own_data->LSIZEY;
808 checkpoint_store->own_data.NX = own_data->NX;
809 checkpoint_store->own_data.NY = own_data->NY;
810 checkpoint_store->own_data.LENGHT = own_data->LENGHT;
811 checkpoint_store->own_data.allocated = 1;
812
813 // Copia os parâmetros (tamanho fixo)
814 memcpy(&checkpoint_store->own_data.param_buffer, param, sizeof(par_t));
815
816 // Aloca e copia o buffer f_aos_h (tamanho calculado de NX * NY)
817 size_t f_aos_h_size = NX * NY;
818 posix_memalign((void**)&checkpoint_store->own_data.f_aos_h_buffer, 4096, f_aos_h_size
       * sizeof(pop_t));
819 memcpy(checkpoint_store->own_data.f_aos_h_buffer, f_aos_h, f_aos_h_size * sizeof(
       pop_t));
820
821 if (posix_memalign((void**)&checkpoint_store->own_data.buffer, 4096, own_data->LENGHT
       * sizeof(data_t)) != 0)
822 {
823     fprintf(stderr, "[%d] Erro ao alocar buffer para own_data\n", mpi_rank);
824     free_checkpoint_store(checkpoint_store);
825     return NULL;
826 }
827
828 copy_data_from_to(own_data, &checkpoint_store->own_data);

```

```

829
830 // Em caso de nico processo, apenas salva os dados locais
831 if (mpi_size == 1)
832 {
833     return checkpoint_store;
834 }
835
836 // Cria tipo MPI personalizado
837 MPI_Datatype checkpoint_type;
838 create_checkpoint_data_type(&checkpoint_type);
839
840 // Para cada n vel de redund ncia
841 int checkpointsToBeSaved = MAX_FAILURE_SUPPORTED < (mpi_size - 1) ?
    MAX_FAILURE_SUPPORTED : (mpi_size - 1);
842 for (int i = 0; i < checkpointsToBeSaved; i++)
843 {
844     int target_rank = (mpi_rank + i + 1) % mpi_size;
845     int source_rank = (mpi_rank - i - 1 + mpi_size) % mpi_size;
846
847     // 1. Troca dos tamanhos dos buffers
848     int send_buffer_size = own_data->LENGHT;
849     int recv_buffer_size;
850
851     int sendrecv_error = MPI_Sendrecv(&send_buffer_size, 1, MPI_INT, target_rank,
        TAG_BUFF_SIZE, &recv_buffer_size, 1, MPI_INT, source_rank,
852        TAG_BUFF_SIZE, comm, MPI_STATUS_IGNORE);
853
854     if (sendrecv_error != MPI_SUCCESS)
855     {
856         char error_string[MPI_MAX_ERROR_STRING];
857         int error_string_length;
858         MPI_Error_string(sendrecv_error, error_string, &error_string_length);
859         printf("[%d]- Erro ao comunicar tamanhos (P1-%d): %s\n", mpi_rank, i,
            error_string);
860         MPI_Type_free(&checkpoint_type);
861         free_checkpoint_store(checkpoint_store);
862         return NULL;
863     }
864
865     // Prepara a estrutura para receber dados
866     checkpoint_store->neighbor_data[i].LSIZEX = 0;
867     checkpoint_store->neighbor_data[i].LSIZEY = 0;
868     checkpoint_store->neighbor_data[i].NX = 0;
869     checkpoint_store->neighbor_data[i].NY = 0;
870     checkpoint_store->neighbor_data[i].LENGHT = recv_buffer_size;
871     checkpoint_store->neighbor_data[i].source_rank = source_rank;
872     checkpoint_store->neighbor_data[i].allocated = 1;
873
874     if (posix_memalign((void*)&checkpoint_store->neighbor_data[i].buffer, 4096,
        recv_buffer_size * sizeof(data_t)) != 0)
875     {
876         fprintf(stderr, "[%d]- Erro ao alocar buffer para neighbor_data [%d]\n",
            mpi_rank, i);
877         MPI_Type_free(&checkpoint_type);
878         free_checkpoint_store(checkpoint_store);
879         return NULL;
880     }
881
882     // 2. Troca dos metadados

```

```

883     MPI_Sendrecv(&checkpoint_store->own_data, 1, checkpoint_type, target_rank,
884                 TAG_STRUCT_DATA, &checkpoint_store->neighbor_data[i], 1,
885                 checkpoint_type, source_rank, TAG_STRUCT_DATA, comm,
886                 MPI_STATUS_IGNORE);
887
888     // 3. Aloca o f_aos_h com base nos metadados recebidos
889     size_t neighbor_f_aos_h_size = checkpoint_store->neighbor_data[i].NX *
890     checkpoint_store->neighbor_data[i].NY * sizeof(pop_t);
891     posix_memalign((void**)&checkpoint_store->neighbor_data[i].f_aos_h_buffer, 4096,
892     neighbor_f_aos_h_size);
893
894     // 4. Troca do buffer principal
895     MPI_Sendrecv(own_data->buffer, own_data->LENGHT, MPI_DOUBLE, target_rank,
896     TAG_BUFF_DATA, checkpoint_store->neighbor_data[i].buffer,
897     checkpoint_store->neighbor_data[i].LENGHT, MPI_DOUBLE, source_rank,
898     TAG_BUFF_DATA, comm, MPI_STATUS_IGNORE);
899
900     // 5. Troca do buffer f_aos_h
901     MPI_Sendrecv(f_aos_h, NX * NY, MPI_BYTE, target_rank, TAG_AOS_DATA,
902     checkpoint_store->neighbor_data[i].f_aos_h_buffer,
903     checkpoint_store->neighbor_data[i].NX * checkpoint_store->
904     neighbor_data[i].NY, MPI_BYTE, source_rank, TAG_AOS_DATA, comm,
905     MPI_STATUS_IGNORE);
906
907     // 6. Troca do buffer param (tamanho fixo)
908     MPI_Sendrecv(&checkpoint_store->own_data.param_buffer, sizeof(par_t), MPI_BYTE,
909     target_rank, TAG_PARAM_DATA,
910     &checkpoint_store->neighbor_data[i].param_buffer, sizeof(par_t),
911     MPI_BYTE, source_rank, TAG_PARAM_DATA, comm, MPI_STATUS_IGNORE);
912
913     checkpoint_store->available_checkpoints++;
914 }
915
916 MPI_Type_free(&checkpoint_type);
917
918 return checkpoint_store;
919 }

```


D IMPLEMENTAÇÃO: VERSÃO WITHOUTTOLERANCE (SEM TOLE- RÂNCIA A FALHAS)

Listing D.1 – lbm.c – versão WithoutTolerance

```

1 /*****
2  *
3  * Copyright (c) 2018, University of Ferrara, University of Rome and INFN. All
4  *rights reserved.
5  *
6  *
7  * Authors listed in alphabetic order:
8  * -----
9  * Luca Biferale (University of Rome "Tor Vergata" and INFN)
10 * Enrico Calore (University of Ferrara and INFN)
11 * Alessandro Gabbana (University of Ferrara and INFN)
12 * Mauro Sbragaglia (University of Rome "Tor Vergata" and INFN)
13 * Andrea Scagliarini (CNR-IAC, Roma)
14 * Sebastiano Fabio Schifano (University of Ferrara and INFN)(*)
15 * Raffaele Tripiccione (University of Ferrara and INFN)
16 *
17 * (*) corresponding author, sebastiano.schifano __at__ unife.it
18 *
19 * We also thanks the following students:
20 * -----
21 * Elisa Pellegrini (University of Ferrara)
22 * Marco Zanella (University of Ferrara)
23 *
24 *
25 * Relevant publications:
26 * -----
27 *
28 * - Sbragaglia, M., Benzi, R., Biferale, L., Chen, H., Shan, X., Succi, S.
29 * Lattice Boltzmann method with self-consistent thermo-hydrodynamic
30 *equilibria (2009) Journal of Fluid Mechanics, 628, pp. 299-309. Cited 66
31 *times.
32 * https://www.scopus.com/inward/record.uri?eid=2-s2.0-67651027859&doi=10.1017%2
33 fS002211200900665X&partnerID=40&md5=1c57f2de4c0abc5aacaf5e30e9d566d7
34 * DOI: 10.1017/S002211200900665X
35 *
36 * - Scagliarini, A., Biferale, L., Sbragaglia, M., Sugiyama, K., Toschi, F.
37 * Lattice Boltzmann methods for thermal flows: Continuum limit and
38 *applications to compressible Rayleigh-Taylor systems (2010) Physics of Fluids,
39 *22 (5), art. no. 019004PHF, pp. 1-21. Cited 51 times.
40 * https://www.scopus.com/inward/record.uri?eid=2-s2.0-77955618379&doi=10.1063%2f1
41 .3392774&partnerID=40&md5=8b47e1b6c7ffd95583e2dc30d36a66f2
42 * DOI: 10.1063/1.3392774
43 *
44 * - Calore, E., Gabbana, A., Kraus, J., Pellegrini, E., Schifano, S.F.,
45 *Tripiccione, R. Massively parallel lattice Boltzmann codes on large GPU
46 *clusters (2016) Parallel Computing, 58, pp. 1-24.
47 * https://www.scopus.com/inward/record.uri?eid=2-s2.0-84983616316&doi=10.1016%2fj.
48 parco.2016.08.005&partnerID=40&md5=a9b40226e16f6e24bc8bc7138b789f90
49 * DOI: 10.1016/j.parco.2016.08.005
50 *
51 * - Calore, E., Gabbana, A., Kraus, J., Schifano, S.F., Tripiccione, R.
52 * Performance and portability of accelerated lattice Boltzmann applications
53 *with OpenACC (2016) Concurrency Computation, 28 (12), pp. 3485-3502.

```

```

51 *   https://www.scopus.com/inward/record.uri?eid=2-s2.0-84971215878&doi=10.1002%2fcpe
   .3862&partnerID=40&md5=fe286342abec002fa287da5967b9d2c2
52 *   DOI: 10.1002/cpe.3862
53 *
54 *
55 * Redistribution and use in source and binary forms, with or without
56 * modification, are permitted provided that the following conditions
57 * are met:
58 *
59 *   * Redistributions of source code must retain the above copyright
60 *   notice, this list of conditions and the following disclaimer.
61 *
62 *   * Redistributions in binary form must reproduce the above copyright
63 *   notice, this list of conditions and the following disclaimer in the
64 *   documentation and/or other materials provided with the distribution.
65 *
66 *   * Neither the name of the above Institutions nor the names of their
67 *   contributors may be used to endorse or promote products derived
68 *   from this software without specific prior written permission.
69 *
70 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND ANY
71 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
72 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
73 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
74 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
75 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
76 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
77 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
78 * OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
79 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
80 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
81 *
82 *****/
83
84 #include <math.h>
85 #include <mpi.h>
86 #include <stdint.h>
87 #include <stdlib.h>
88 #include <string.h>
89 #include <sys/time.h>
90
91 #include "common.h"
92
93 extern int mpi_rank; // Rank of MPI node.
94
95 extern int n_mpi; // total number of MPI ranks == n_mpi_x * n_mpi_y == mpi_size
96
97 extern int n_mpi_x; // number of MPI ranks along X direction
98 extern int n_mpi_y; // number of MPI ranks along Y direction
99
100 extern int mpi_rank_x; // MPI X coordinate.
101 extern int mpi_rank_y; // MPI Y coordinate.
102
103 extern int mpi_rank; // Rank of MPI node.
104 extern int mpi_size; // Size of MPI world communicator.
105
106 extern int n_mpi; // total number of MPI ranks == n_mpi_x * n_mpi_y == mpi_size
107
108 extern int n_mpi_x; // number of MPI ranks along X direction
109 extern int n_mpi_y; // number of MPI ranks along Y direction

```

```

110
111 extern int mpi_rank_x; // MPI X coordinate.
112 extern int mpi_rank_y; // MPI Y coordinate.
113
114 extern int mpi_peer_left, mpi_peer_right, mpi_peer_top,
115         mpi_peer_bot; // Rank of neighboring MPI ranks
116
117 extern char* mpi_order; // MPI ranks order string
118
119 extern int      GSIZE_X, GSIZE_Y; // global lattice sizex and sizey
120 extern int      LSIZE_X, LSIZE_Y; // local lattice sizex and sizey
121 extern uint64_t NX, NY;           // x-length and y-length of lattice in memory.
122
123 extern long int SEED; // seed for random generator
124
125 extern const size_t OFF0[37];
126
127 extern size_t OFF[NPOP];
128
129 extern const int bits;
130
131 #include "utils.h"
132
133 #include "bc_kernel.h"
134 #include "collide_kernel.h"
135 #include "packunpack.h"
136 #include "propagate_kernel.h"
137
138 ///////////////////////////////////////////////////////////////////
139
140 #define NFLOP 6613
141
142 #define Q1 (1)
143 #define Q2 (2)
144 #define Q3 (3)
145 #define Q4 (4)
146 #define Q5 (5)
147
148 ///////////////////////////////////////////////////////////////////
149 // Function used to launch kernels.
150
151 void lbm(int niter, pop_t* f_aos_h, par_t* param)
152 {
153     long interaction, yoff; // ip, ix not used;
154
155     data_t *f1_soa_h = NULL, *f2_soa_h = NULL, *f_tmp_h = NULL;
156
157     data_t *topSndBuf = NULL, *topRxBuf = NULL, *botSndBuf = NULL, *botRxBuf = NULL;
158     data_t *leftSndBuf = NULL, *leftRxBuf = NULL, *rightSndBuf = NULL, *rightRxBuf = NULL
159         ;
160
161     data_t mass, i_mass, g_mass;
162
163     // MPI_Status status;
164
165     MPI_Request reqRxTopHalo, reqTxTopBorder, reqRxBotHalo, reqTxBotBorder;
166     MPI_Request reqRxLeftHalo, reqTxRightBorder, reqRxRightHalo, reqTxLeftHalo;
167
168     int tagUP = 1, tagDOWN = 2, tagLEFT = 3, tagRIGHT = 4;

```

```

169 // double mpimin_dt_tot, mpimax_dt_tot;
170 // double mpimax_dt_swap_nc, mpimin_dt_swap_nc, double mpimax_dt_swap_c,
171 // mpimin_dt_swap_c;
172
173 double mpimax_dt_tot, mpimax_dt_swap_nc, mpimax_dt_swap_c;
174 double t1, t2;
175 // double latticesize; //, bw, p;
176 double mlups;
177
178 ///////////////////////////////////////////////////
179
180 posix_memalign((void*)&topSndBuf, 4096, (15 + 8 + 3) * LSIZE * sizeof(data_t));
181 posix_memalign((void*)&topRxBuf, 4096, (15 + 8 + 3) * LSIZE * sizeof(data_t));
182 posix_memalign((void*)&botSndBuf, 4096, (15 + 8 + 3) * LSIZE * sizeof(data_t));
183 posix_memalign((void*)&botRxBuf, 4096, (15 + 8 + 3) * LSIZE * sizeof(data_t));
184
185 posix_memalign((void*)&leftSndBuf, 4096, (15 + 8 + 3) * NY * sizeof(data_t));
186 posix_memalign((void*)&leftRxBuf, 4096, (15 + 8 + 3) * NY * sizeof(data_t));
187 posix_memalign((void*)&rightSndBuf, 4096, (15 + 8 + 3) * NY * sizeof(data_t));
188 posix_memalign((void*)&rightRxBuf, 4096, (15 + 8 + 3) * NY * sizeof(data_t));
189
190 if ((topSndBuf == NULL) || (topRxBuf == NULL) || (botSndBuf == NULL) || (botRxBuf ==
    NULL) ||
191     (leftSndBuf == NULL) || (leftRxBuf == NULL) || (rightSndBuf == NULL) ||
192     (rightRxBuf == NULL))
193 {
194     fprintf(stderr, "ERROR: allocation of MPI buffers failed\n");
195     exit(-1);
196 }
197
198 ///////////////////////////////////////////////////
199 unsigned long nx = NX;
200 unsigned long ny = NY;
201 unsigned long npop = NPOP;
202
203 unsigned long nelem = nx * ny * npop;
204 unsigned long size = nelem * sizeof(data_t);
205
206 posix_memalign((void*)&f1_soa_h, 4096, size);
207
208 if (f1_soa_h == NULL)
209 {
210     fprintf(stderr, "ERROR: f1_soa_h posix_memalign failed\n");
211     exit(-1);
212 }
213
214 posix_memalign((void*)&f2_soa_h, 4096, size);
215 if (f2_soa_h == NULL)
216 {
217     fprintf(stderr, "ERROR: f2_soa_h posix_memalign failed\n");
218     exit(-1);
219 }
220
221 memset(f1_soa_h, 0x0, size);
222 memset(f2_soa_h, 0x0, size);
223 ///////////////////////////////////////////////////
224
225 AoStoSoA(f_aos_h, f2_soa_h); // f_aos_h --> f2_soa_h
226 memcpy(f1_soa_h, f2_soa_h, size);
227

```

```

228     yoff = mpi_rank_y * (GSIZEY / n_mpi_y);
229
230 #ifdef CHECK_MASS
231     mass = dumpMass(f_aos_h);
232     MPI_Reduce(&mass, &i_mass, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
233 #endif
234
235     MPI_Barrier(MPI_COMM_WORLD);
236
237
238     double interactionStartTime, interactionEndTime, interactionExecTime;
239     interactionStartTime = MPI_Wtime();
240     for (interaction = 1; interaction <= niter; interaction++)
241     {
242         interactionStartTime = MPI_Wtime();
243
244         if (mpi_rank == 0)
245         {
246             printf("[R=%d] - Starting iteration: %ld. Last took %f seconds\n", mpi_rank,
247                 interaction, interactionExecTime);
248         }
249
250         if (n_mpi_y > 1)
251         {
252             // update top and bottom halos
253             // 1) We start by packing the top and the bottom borders which
254             // need to be
255             // exchanged among the appropriate MPI-ranks
256
257             // Para quem tem um vizinho no topo
258             if (mpi_rank_y < (n_mpi_y - 1))
259             {
260                 pack_top_border(topSndBuf, f2_soa_h, Q1);
261             }
262
263             // Para quem tem um vizinho embaixo
264             if (mpi_rank_y > 0)
265             {
266                 pack_bot_border(botSndBuf, f2_soa_h, Q2);
267             }
268
269             // Para quem tem vizinho no topo
270             if (mpi_rank_y < (n_mpi_y - 1))
271             {
272                 MPI_Irecv(topRxBuf, (15 + 8 + 3) * LSIZE, MPI_DOUBLE, mpi_peer_top,
273                     tagDOWN, MPI_COMM_WORLD, &reqRxTopHalo);
274
275                 MPI_Isend(topSndBuf, (15 + 8 + 3) * LSIZE, MPI_DOUBLE, mpi_peer_top,
276                     tagUP, MPI_COMM_WORLD, &reqTxTopBorder);
277             }
278
279             // 3) Para quem tem vizinho em baixo
280             if (mpi_rank_y > 0)
281             {
282                 MPI_Irecv(botRxBuf, (15 + 8 + 3) * LSIZE, MPI_DOUBLE, mpi_peer_bot,
283                     tagUP, MPI_COMM_WORLD, &reqRxBotHalo);
284
285                 MPI_Isend(botSndBuf, (15 + 8 + 3) * LSIZE, MPI_DOUBLE, mpi_peer_bot,
286                     tagDOWN, MPI_COMM_WORLD, &reqTxBotBorder);
287             }
288         }
289     }

```

```

283
284 // Para quem tem vizinho no topo
285 if (mpi_rank_y < (n_mpi_y - 1))
286 {
287     // Aguarda que a troca de informa es se completem
288     // Copia de volta o topo-halo na mem ria
289     MPI_Wait(&reqTxTopBorder, MPI_STATUS_IGNORE);
290     MPI_Wait(&reqRxTopHalo, MPI_STATUS_IGNORE);
291
292     unpack_top_halo(f2_soa_h, topRxBuf, Q1);
293 }
294
295 // Para quem tem vizinho em baixo
296 if (mpi_rank_y > 0)
297 {
298     // Aguarda que a troca de informa es se completem
299     // Copia de volta o topo-halo na mem ria
300     MPI_Wait(&reqTxBotBorder, MPI_STATUS_IGNORE);
301     MPI_Wait(&reqRxBotHalo, MPI_STATUS_IGNORE);
302
303     unpack_bot_halo(f2_soa_h, botRxBuf, Q2);
304 }
305
306 } // Para quem pode ter vizinhos em cima ou embaixo
307 // That's it for the 2D-domain partitioning case
308
309
310 // update left and right halos ...
311 pack_right_border(leftSndBuf, f2_soa_h, Q1);
312 pack_left_border(rightSndBuf, f2_soa_h, Q2);
313
314 // Run propagate and collide on Bulk
315 PROPAGATECOLLIDE(f1_soa_h, f2_soa_h, (HX + hx), (HX + LSIZEX - hx), (HY + hy + 1)
316 ,
317                 (HY + LSIZEY - hy - 1), yoff, Q3);
318
319 // MPI-Comm Right-Border
320 MPI_Irecv(leftRxBuf, (15 + 8 + 3) * NY, MPI_DOUBLE, mpi_peer_left, tagLEFT,
321           MPI_COMM_WORLD, &reqRxLeftHalo);
322 MPI_Isend(leftSndBuf, (15 + 8 + 3) * NY, MPI_DOUBLE, mpi_peer_right, tagLEFT,
323           MPI_COMM_WORLD, &reqTxRightBorder);
324
325 // MPI-Comm Left-Border
326 MPI_Irecv(rightRxBuf, (15 + 8 + 3) * NY, MPI_DOUBLE, mpi_peer_right, tagRIGHT,
327           MPI_COMM_WORLD, &reqRxRightHalo);
328 MPI_Isend(rightSndBuf, (15 + 8 + 3) * NY, MPI_DOUBLE, mpi_peer_left, tagRIGHT,
329           MPI_COMM_WORLD, &reqTxLeftHalo);
330
331 // Wait right-border Comm and tranfer back to device
332 MPI_Wait(&reqTxRightBorder, MPI_STATUS_IGNORE);
333 MPI_Wait(&reqRxLeftHalo, MPI_STATUS_IGNORE);
334
335 unpack_left_halo(f2_soa_h, leftRxBuf, Q1);
336
337 // Wait left-border Comm and tranfer back to device
338 MPI_Wait(&reqTxLeftHalo, MPI_STATUS_IGNORE);
339 MPI_Wait(&reqRxRightHalo, MPI_STATUS_IGNORE);
340
341 unpack_right_halo(f2_soa_h, rightRxBuf, Q2);

```

```

338 // Wait END of left and right unpack kernels: processing of TOP and
339 // BOTTOM borders require that unpack of left and right halos has been
340 // fully completed
341
342 // propagate and collide on Left border
343 PROPAGATECOLLIDE(f1_soa_h, f2_soa_h, (HX), (HX + hx), (HY + hy + 1), (HY + LSIZEY
    - hy - 1), yoff, Q1);
344
345 // propagate and collide on Right border
346 PROPAGATECOLLIDE(f1_soa_h, f2_soa_h, (HX + LSIZEX - hx), (HX + LSIZEX), (HY + hy
    + 1), (HY + LSIZEY - hy - 1), yoff, Q2);
347
348 // Bottom Border
349 if (mpi_rank_y == 0)
350 {
351     propagate(f1_soa_h, f2_soa_h, (HX), (HX + LSIZEX), (HY), (HY + hy + 1), Q4);
352
353     bcBottom(f1_soa_h, f2_soa_h, param, (HX), (HX + LSIZEX), Q4);
354
355     COLLIDE(f1_soa_h, f1_soa_h, (HX), (HX + LSIZEX), (HY), (HY + hy + 1), yoff,
        Q4);
356 }
357 else
358 {
359     PROPAGATECOLLIDE(f1_soa_h, f2_soa_h, (HX), (HX + LSIZEX), (HY), (HY + hy + 1)
        , yoff, Q4);
360 }
361
362 // Top Border
363 if (mpi_rank_y == (n_mpi_y - 1))
364 {
365     propagate(f1_soa_h, f2_soa_h, (HX), (HX + LSIZEX), (HY + LSIZEY - hy - 1), (
        HY + LSIZEY), Q5);
366
367     bcTop(f1_soa_h, f2_soa_h, param, (HX), (HX + LSIZEX), Q5);
368
369     COLLIDE(f1_soa_h, f1_soa_h, (HX), (HX + LSIZEX), (HY + LSIZEY - hy - 1), (HY
        + LSIZEY), yoff, Q5);
370 }
371 else
372 {
373     PROPAGATECOLLIDE(f1_soa_h, f2_soa_h, (HX), (HX + LSIZEX), (HY + LSIZEY - hy -
        1), (HY + LSIZEY), yoff, Q5);
374 }
375
376 // Swap pointers
377 f_tmp_h = f1_soa_h;
378 f1_soa_h = f2_soa_h;
379 f2_soa_h = f_tmp_h;
380
381 MPI_Barrier(MPI_COMM_WORLD);
382
383 interactionEndTime = MPI_Wtime();
384 interactionExecTime = interactionEndTime - interactionStartTime;
385 }
386
387 //////////////////////////////////////////////////
388 // End of LBM iterations.
389
390 // mass check

```

```

391 #ifdef CHECK_MASS
392     // Lattice is copied back to the host and converted to AoS
393     SoAtoAoS(f2_soa_h, f_aos_h); // f2_soa_h --> f_aos_h
394     mass = dumpMass(f_aos_h);
395     MPI_Barrier(MPI_COMM_WORLD);
396
397     MPI_Reduce(&mass, &g_mass, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
398     if (mpi_rank == 0)
399     {
400         if (fabs((i_mass - g_mass) / i_mass) < 1.e-10)
401         {
402             fprintf(stdout, "Mass check: PASSED %.10e == %.10e (Er: %e).\n", i_mass,
403                 g_mass,
404                 fabs((i_mass - g_mass)) / i_mass);
405         }
406         else
407         {
408             fprintf(stdout, "Mass check: FAILED %.10e <> %.10e (Er: %e).\n", i_mass,
409                 g_mass,
410                 fabs((i_mass - g_mass)) / i_mass);
411         }
412         fflush(stdout);
413     }
414 #endif
415
416     MPI_Barrier(MPI_COMM_WORLD);
417 }

```