UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Vanderlei Munhoz Pereira Filho

HPC@Cloud: A Provider-Agnostic Toolkit to Enable the Execution of HPC Applications on Public Clouds

Florianópolis
2023

Vanderlei Munhoz Pereira Filho

HPC@Cloud: A Provider-Agnostic Toolkit to Enable the Execution of HPC Applications on Public Clouds

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de Mestre em Ciência da Computação.
Supervisor: Prof. Márcio Bastos Castro, Dr.

Florianópolis
2023

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

Pereira Filho, Vanderlei Munhoz
HPC@Cloud: A Provider-Agnostic Toolkit to Enable the Execution of HPC Applications on Public Clouds / Vanderlei Munhoz Pereira Filho ; orientador, Márcio Bastos Castro, 2024.
102 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2024.

Inclui referências.

1. Ciência da Computação. 2. HPC. 3. Computação em Nuvem. 4. MPI. 5. Spot. I. Castro, Márcio Bastos. II. Universidade Federal de Santa Catarina. Programa de Pós Graduação em Ciência da Computação. III. Título.

Vanderlei Munhoz Pereira Filho

HPC@Cloud: A Provider-Agnostic Toolkit to Enable the Execution of HPC Applications on Public Clouds

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Profa. Lúcia Maria de Assumpção Drummond, Dra.
Universidade Federal Fluminense (UFF)

Prof. Tiago Coelho Ferreto, Dr.
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)

Prof. Douglas Dyllon Jeronimo de Macedo, Dr.
Universidade Federal de Santa Catarina (UFSC)

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Mestre em Ciência da Computação.

Documento assinado digitalmente
Marcio Bastos Castro
Data: 27/02/2024 17:30:41-0300
CPF: ***.707.720-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Márcio Bastos Castro, Dr.
Coordenador do Programa

Documento assinado digitalmente
Marcio Bastos Castro
Data: 27/02/2024 17:31:03-0300
CPF: ***.707.720-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Márcio Bastos Castro, Dr.
Orientador

Florianópolis, 2023.

ACKNOWLEDGEMENTS

The authors would like thank the National Laboratory for Scientific Computing (LNCC/MCTI), whose resources have contributed to this research. This work was partially funded by the National Council for Scientific and Technological Development (CNPq) and Amazon Web Services (AWS) through the CNPq/AWS call Nº 64/2022 — Cloud Credits for Research.

RESUMO

O advento da computação em nuvem tornou o acesso à infraestrutura de computação disponível para milhões de pesquisadores e organizações. No contexto da Computação de Alto Desempenho (*High Performance Computing* – HPC), os recursos da nuvem pública emergiram como uma alternativa custo-efetiva aos caros *clusters* locais. No entanto, existem vários desafios e limitações na adoção dessa abordagem. Esta dissertação propõe o *HPC@Cloud*, um conjunto de ferramentas de *software* de código aberto e agnóstico a provedores que facilita a migração, teste e execução de aplicações HPC em plataformas de nuvem pública. A ferramenta aproveita várias tecnologias de tolerância a falhas para permitir o uso de infraestrutura de nuvem efêmera e de baixo custo, comumente conhecida como instâncias "spot" na Amazon Web Services (AWS). Além disso, possui integração com contêineres Singularity, permitindo aos usuários executar aplicações complexas em *clusters* virtuais de HPC de maneira portátil e reprodutível. Executamos uma diversa gama de experimentos para avaliar o desempenho e a eficiência das soluções propostas e integradas ao *HPC@Cloud*, incluindo um estudo de caso de migração de uma aplicação real de simulação física, o *DynEMol*, comparando seu desempenho na AWS e em um *cluster* HPC tradicional. Por fim, nossa ferramenta fornece uma abordagem baseada em dados para estimar os custos da infraestrutura de nuvem quando aplicações são executadas. Os resultados obtidos em dois provedores de nuvem pública (AWS e Vultr) mostraram que: (i) o *HPC@Cloud* pode construir *clusters* virtuais de HPC na nuvem de forma eficiente; (ii) as estratégias de tolerância a falhas propostas são eficazes em ajudar a reduzir custos sem incorrer em penalidades de desempenho relevantes; (iii) nosso estudo de caso de migração do *DynEMol* demonstrou que o uso de uma plataforma de nuvem pública, embora menos eficiente do que o *cluster* tradicional, é economicamente viável; (iv) o uso de contêineres melhora a portabilidade das aplicações HPC com perda de desempenho mínima, embora com complexidade adicional de configuração e comunicação; (v) a abordagem de previsão de custos proposta é capaz de estimar o tempo de execução das aplicações na AWS e Vultr com pequenos erros quadráticos médios, proporcionando informação valiosa para tomada de decisão pelo usuário final.

Palavras-chave: HPC. Computação em Nuvem. Spot. Singularity. MPI.

RESUMO ESTENDIDO

Introdução

A computação em nuvem democratizou o acesso a recursos de infraestrutura, permitindo a organizações e indivíduos utilizarem serviços pela Internet sem contratos de longo prazo, reduzindo custos operacionais e de investimento. Embora ofereça vantagens significativas, a migração e otimização de aplicações para *High Performance Computing (HPC)* em ambientes de nuvem pública apresenta desafios devido à complexidade e aos requisitos de desempenho, tornando a adoção custosa para alguns casos de uso. Os provedores de nuvem oferecem hardware especializado de alto desempenho, mas alugar essas máquinas pode ser mais caro do que manter um *cluster* HPC local. Além disso, a abstração de hardware, apesar de ser uma vantagem da computação em nuvem, dificulta a otimização de aplicações computacionalmente intensivas para HPC. Nesse contexto, nós argumentamos que para uma gama de aplicações HPC, a computação em nuvem ainda pode ser custo-efetiva se os usuários tiverem acesso a um conjunto abrangente de *softwares* e frameworks para migração.

Objetivos

Este trabalho visa avaliar a eficiência de custo e a adequação das tecnologias atuais de computação em nuvem para cargas de trabalho de HPC, variando de benchmarks a aplicações reais, e identificar os principais obstáculos na migração para plataformas de nuvem pública. Desenvolvemos o *HPC@Cloud*, uma ferramenta de código aberto, para facilitar a implantação de aplicações HPC em plataformas de nuvem pública com esforço mínimo, abordando lacunas identificadas em estudos relacionados. Os objetivos específicos incluem: (1) uma análise detalhada da eficiência de custo ao usar máquinas virtuais públicas do tipo persistente e também efêmeras, também conhecidas como instâncias spot, para executar aplicações e benchmarks em dois provedores de nuvem (AWS e Vultr); (2) a proposta de mecanismos de tolerância a falhas configuráveis para restauração de aplicações baseadas em *Message Passing Interface (MPI)*, voltadas para o uso de infrastrutura efêmera na núvem, reduzindo custos; (3) avaliar o impacto de tecnologias de contêinerização em virtualização de hipervisor para cargas de trabalho MPI na nuvem pública; (4) introduzir um método empírico para estimar custos de execução de cargas de trabalho MPI em nuvens públicas; e (5) realizar a migração e avaliação de performance de uma aplicação de simulação física real, o *DynEMol*, para AWS, comparando custos operacionais com um *cluster* tradicional, e detalhando os principais desafios encontrados e como superá-los.

Metodologia

Neste trabalho, empreendemos uma avaliação abrangente visando abordar lacunas de *software* no contexto da computação de alto desempenho em plataformas de nuvem pública, conforme identificado em estudos anteriores. Nossa avaliação da ferramenta *HPC@Cloud*, desenvolvido para facilitar a transição de cargas de trabalho HPC para ambientes em nuvem, seguiu várias dimensões, incluindo escalabilidade, gerenciamento de recursos de nuvem, estratégias de tolerância a falhas, migração de aplicações legadas, execução containerizada, e predição de custos de execução. No âmbito da escalabilidade de *clusters*, tecnologias e gerenciamento de recursos, testamos a escalabilidade da infraestrutura em nuvem da AWS e da Vultr executando uma aplicação benchmark desenvolvida neste tra-

balho, testando dois tamanhos de problema em várias configurações de *clusters*. Isso incluiu a exploração de diferentes tipos de instâncias e tecnologias de nuvem relacionadas a rede e armazenamento, buscando entender seu impacto no desempenho e escalabilidade de aplicações HPC na nuvem. Continuamos analisando as estratégias de tolerância a falhas propostas, avaliando a eficácia de diferentes mecanismos para cargas de trabalho HPC, particularmente no contexto de recursos em nuvem efêmeros, como instâncias spot da AWS. Executamos diversos experimentos comparando tecnologias como *Berkeley Lab Checkpoint-Restart (BLCR)*, *Distributed MultiThreaded CheckPointing (DMTCP)*, *Scalable Checkpoint Restart (SCR)*, e *User-Level Failure Mitigation (ULFM)*, identificando as perdas de desempenho de cada uma delas conforme falhas ocorrem no sistema. Ademais, buscamos realizar uma análise da economia financeira do uso de instâncias spot com aplicações tolerantes a falhas em comparação ao uso de instâncias persistentes com aplicações sem tolerância a falhas. No estudo da migração de aplicações legadas, realizamos um caso de estudo focado na migração e avaliação de desempenho da aplicação de simulação física conhecida como *DynEMol* para a AWS. Testamos várias configurações e tecnologias em nuvem para avaliar sua adequação e desempenho, fornecendo insights sobre os desafios e benefícios da migração para a nuvem. Os resultados de desempenho e custo do *DynEMol* na AWS são então comparados com os resultados obtidos quando executado em um *cluster* HPC tradicional do departamento de Física da UFSC. Na avaliação acerca da execução containerizada, essa pesquisa também investigou o uso da tecnologia Singularity como caso de teste para execução de cargas de trabalho MPI. O objetivo foi entender as implicações de desempenho e complexidades operacionais de implantar aplicações HPC desse tipo quando containerizadas em ambientes de nuvem. São executados diversos experimentos com benchmarks, onde analisamos a diferença de desempenho quando as cargas são executadas diretamente nas instâncias, sem o uso de contêineres. Visando prever o tempo e o custo de execução de uma aplicação, com o objetivo de apoiar o processo de decisão na seleção da infraestrutura, testamos dois modelos de machine learning simples, XGBoost e Regressão Linear, usando dados capturados pelo *HPC@Cloud* ao executar cargas de trabalho. Nosso estudo comparativo sobre previsão de custos demonstrou visa demonstrar a viabilidade do uso de modelos de aprendizado de máquina para esse objetivo, lançando as fundações para trabalhos futuros na área. Cada um desses critérios de avaliação foi meticulosamente projetado para sondar a convergência de HPC e Computação em Nuvem, focando não apenas em desempenho e escalabilidade, mas também nas implicações operacionais e de custo de implantar cargas de trabalho HPC em ambientes de nuvem. Esta metodologia abrangente fornece um robusto framework para avaliar a viabilidade e eficácia das plataformas de nuvem para aplicações HPC, abrindo caminho para futuras pesquisas e desenvolvimento nesta área crítica da tecnologia da computação.

Resultados e Discussão

Obtivemos os seguintes resultados: (a) o método utilizado pelo *HPC@Cloud* para requisição e configuração de *clusters* é extremamente escalável, levando cerca de 1 minuto e meio para preparar um *cluster* de até 8 nós quando utilizando imagens preparadas. Sem o uso de imagens preparadas, o tempo de configuração de um *cluster* depende dos recursos computacionais dos nós, que precisam compilar e instalar dependências. Se uma aplicação muito grande será executada, criar uma imagem preparada para ganhar alguns minutos no tempo de execução pode não valer o esforço; (b) nos experimentos que visam avaliar o desempenho e escalabilidade na execução de benchmarks na AWS e Vultr, detectamos que o tamanho do problema afeta diretamente na escalabilidade do *cluster*, confirmado pelos resultados obtidos que demonstraram uma perda de desempenho significante quando o

cluster é escalonado horizontalmente para aplicações pequenas, enquanto que para aplicações grandes o escalonamento horizontal é vantajoso; (c) ademais, a partir dos experimentos de escalabilidade e desempenho iniciais, concluímos que o uso de instâncias com muitos recursos computacionais em menor quantidade é geralmente mais vantajoso do que o uso de uma grande quantidade de instâncias com poucos recursos; (d) na avaliação das estratégias de tolerância a falhas, concluímos que as estratégias em nível de sistema (BLCR e DMTCP) são mais facilmente integráveis do que as estratégias em nível de aplicação (SCR e ULFM), todavia possuem desempenho geralmente inferior, com o ULFM obtendo os melhores resultados, seguido por SCR, DMTCP e BLCR, respectivamente; (e) em nossos testes, a abordagem e frequência adotada para a execução de checkpoints influenciou drasticamente o tempo total de execução das aplicações, sobretudo quando utilizadas as técnicas a nivel de sistema, causando uma demora aproximadamente 4 vezes maior (BLCR com checkpoint periódico) para completar a execução, em comparação com o caso sem falhas; (f) a técnica de tolerância a falhas mais eficiente foi a execução adaptativa com ULFM, que permite a execução contínua da aplicação sem grandes interrupções durante a ocorrência de falhas, proporcionando um desempenho próximo da aplicação sem falhas; (g) no âmbito de custos, o uso da técnica ULFM com instâncias *spot* na AWS proporcionou uma economia de até 87% comparado aos custos do uso de instâncias normais, enquanto que o uso da técnica BLCR com checkpoints periódicos (totalizando 20 checkpoints durante a execução), custou 42% a mais do que o uso de instâncias comuns, demonstrando que o uso de infraestrutura efêmera não garante economia financeira; (h) mesmo com técnicas não tão eficientes para tolerância a falhas, como as de nível de sistema, com o uso de alarmes da AWS e a execução preemptiva de checkpoints, foi possível reduzir bastante o tempo de execução (diminuindo de 20 para apenas 2 checkpoints realizados), atingindo uma economia de até 73% quando utilizando DMTCP com instâncias *spot*, por exemplo; (i) quando levado em conta os possíveis custos de implementação de uma solução mais eficiente a nível de aplicação, os custos podem acabar sendo maiores do que simplesmente utilizar uma técnica a nível de sistema, que não requer modificações em código; (j) em nosso estudo de caso de migração da aplicação *DynEMol*, confirmamos os mesmos resultados obtidos em outros experimentos, reforçando a importância da configuração minuciosa do tamanho do *cluster*, ou escalonabilidade horizontal, que afeta drasticamente o tempo de execução total; (k) quando comparada a execução virtualizada na nuvem com a execução no *cluster* físico tradicional, os resultados foram similares, entretanto, o *cluster* físico obteve os melhores resultados (speedup de 1.29, comparado com um speedup de 1.22 na nuvem); (l) o uso de tecnologias proprietárias da AWS, como o sistema de armazenamento FSx for Lustre, e a tecnologia de rede Elastic Fabric Adapter (EFA) não demonstraram um ganho significativo de desempenho; (m) nos experimentos envolvendo a execução containerizada, o uso da ferramenta Singularity para a execução de cargas de trabalho MPI resultou em uma perda insignificante de desempenho, demonstrando ser completamente viável para HPC na nuvem; (n) a abordagem de previsão de custos proposta, baseada em XGBoost, foi capaz de estimar o tempo de execução das aplicações na AWS e Vultr com um pequeno erro quadrático médio (7.97s para uma média de 140.81s e mediana de 120.00s), enquanto que a abordagem baseada em regressão linear não obteve resultados tão satisfatórios, sendo incapaz de detectar não-linearidade nos dados utilizados para treinamento do modelo de predição.

Considerações Finais

A ferramenta *HPC@Cloud* representa um passo significativo em direção à convergência de HPC e computação em nuvem. Nossas contribuições neste trabalho não apenas

demonstram as capacidades da ferramenta, mas também fornecem percepções sobre as complexidades do gerenciamento de recursos na nuvem e a tolerância a falhas em HPC na nuvem. Ao continuar construindo sobre essas bases, nosso objetivo é desbloquear o potencial total das plataformas de computação em nuvem públicas para a comunidade HPC. Planejamos implementar uma versão centralizada do *HPC@Cloud*, que será acessada por meio de uma API REST, eliminando a necessidade de instalação de dependências na máquina do usuário, como Terraform, Docker e CLIs de provedores, simplificando o uso por pesquisadores. Além disso, pretendemos adicionar suporte para outros dois grandes provedores de nuvem pública: *Google Cloud Platform (GCP)* e Azure. Com esse suporte, o *HPC@Cloud* pode se tornar verdadeiramente uma solução multiplataforma para HPC na nuvem.

Palavras-chave: HPC. Computação em Nuvem. Spot. Singularity. MPI.

ABSTRACT

The advent of cloud computing has made access to computing infrastructure available to millions of researchers and organizations. In the context of High-Performance Computing (HPC), public cloud resources have emerged as a cost-effective alternative to expensive on-premises clusters. However, there are several challenges and limitations in adopting this approach. This dissertation proposes *HPC@Cloud*, a multi-provider, open-source software toolkit that facilitates the migration, testing, and execution of HPC applications on public cloud platforms. The toolkit leverages various fault tolerance technologies to enable the use of inexpensive ephemeral cloud infrastructure, commonly known as "spot" instances in Amazon Web Services (AWS). Additionally, it features integration with Singularity containers, allowing users to run complex applications on virtual HPC clusters in a portable and reproducible way. We conducted a diverse range of experiments to assess the performance and efficiency of the proposed solutions and integrations within *HPC@Cloud*, including a case study of migrating a real physical simulation application, *DynEMol*, and comparing its performance on AWS to a traditional HPC cluster. Finally, the toolkit provides a data-based approach to estimating cloud infrastructure costs when running applications. The results obtained on two public cloud providers (AWS and Vultr) show that: (i) *HPC@Cloud* can efficiently build virtual HPC clusters on the cloud; (ii) the proposed fault tolerance strategies proved effective in helping reduce costs without incurring relevant performance penalties; (iii) our case study migration of *DynEMol* demonstrated that while the use of a public cloud platform is less efficient than the traditional cluster, it is economically viable; (iv) the use of containers improves the portability of HPC applications with a minimal performance footprint, albeit with added setup and communications complexity; (v) the proposed cost prediction approach can estimate the running time of applications on AWS and Vultr with small round median square errors, providing valuable information for end-user decision-making.

Keywords: HPC. Cloud Computing. Spot. Singularity. MPI.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS AND ACRONYMS

HT HyperThreading. 78

I/O Input and Output interfaces. 36, 62, 63, 77, 81, 82, 89, 91
IaaS Infrastructure as a Service. 38, 39, 40, 41, 47, 51
IaC Infrastructure as Code. 54
IOPs Input and Output Operations per Second. 55
IT Information Technology. 15, 38, 39

JSON JavaScript Object Notation. 55

LTS Long-Term Support. 67

MAE Mean Absolute Error. 84, 86
ML Machine Learning. 43, 86
MPI Message Passing Interface. . 9, 10, 11, 32, 33, 36, 37, 38, 41, 44, 46, 47, 49, 50, 52, 53, 54, 58, 61, 62, 63, 64, 65, 66, 67, 68, 74, 79, 80, 82, 83, 89, 90, 92
MPIF Message Passing Interface Forum. 37

NFS Network File System. 53, 55, 57, 67
NIST National Institute of Standards and Technology. 38
NPB NAS Parallel Benchmarks. 43, 44, 69, 71, 82, 84, 93
NSF National Science Foundation. 62

OS Operating System. 37, 82

PaaS Platform as a Service. 38, 39
PID Process Identification Number. 61

RAM Remote Access Memory. 77, 85
RDMA Remote Direct Memory Access. 78
RMSE Root Mean Square Error. 84, 86
RoCE RDMA over Converged Ethernet. 43

SaaS Software as a Service. 38, 39, 47
SCR Scalable Checkpoint Restart. 10, 21, 32, 44, 62, 63, 64, 76, 90
SOA Service Oriented Architecture. 38
SSH Secure Shell Protocol. 53, 58, 72

TPU Tensor Processing Unit. 43

UFSC University of Santa Catarina. 45, 78
ULFM User-Level Failure Mitigation. 10, 21, 32, 44, 49, 50, 62, 63, 64, 65, 76, 77, 90, 91

vCPU Virtual CPU. 29, 31, 39, 40, 56, 74
VLSI Very Large Scale Integration. 36
VM Virtual Machine. 15, 30, 31, 32, 35, 39, 40, 41, 47, 51, 53, 55, 57, 58, 59, 67, 68, 71, 72, 73, 82, 92
VNA VPC 2.0 Network Adapter. 58
VPC Virtual Private Cloud. 57

YAML Yet Another Markup Language. 55, 59

CONTENTS

1 INTRODUCTION

Public cloud platforms allow anyone to access their services over the Internet without requiring a long-term contract. By leveraging economies of scale, the Cloud Computing paradigm has democratized access to infrastructure resources, making it available to millions of organizations and individuals. The common pay-per-use model adopted by cloud providers significantly lowers the capital barrier required to set up computing infrastructure, reducing investment risk and operational costs (Buyya et al., 2019). Amazon Web Services (AWS), Microsoft Azure, Alibaba Cloud, Google Cloud Platform (GCP), and Huawei are some of the leading public cloud providers in terms of revenue, collectively accounting for 80% of the market share (Gartner, 2022).

The compelling benefits of Cloud Computing, particularly for organizations handling High Performance Computing (HPC) workloads on constrained budgets, make it an attractive option for meeting their computational needs. Nevertheless, as underscored by Netto et al. (2018), the migration of applications to the cloud is far from a straightforward endeavor. The complexity is further exacerbated when attempting to optimize traditional HPC applications, which have stringent performance requirements to function efficiently, specially within the unpredictable shared environments offered by public cloud providers.

Comparable to what currently exists in terms of software tools for enterprise web applications, there are few tools that focus on converging HPC and public cloud platforms, despite the ongoing effort by major industry players and research institutions. As of today, organizations that want to benefit from accessible compute resources have to build, configure, and manage the necessary infrastructure from scratch, on top of handling the entire migration process, making the public cloud adoption for HPC use cases costly and potentially unviable (Coghlan; Yelick, 2011).

Although several mainstream providers offer specialized hardware through on-demand cloud pricing models, renting these kinds of machines can easily get more expensive than buying and maintaining an on-premise HPC cluster, contrary to what may be expected (Emeras; Varrette; Bouvry, 2016). Market forces that regulate large-scale Cloud Computing purchases are also different from small-scale use cases, where users can readily get access to infrastructure without previous notice or long-term contracts. Standard accounts in AWS, for instance, can launch up to 20 instances with up to 64 Virtual CPUs (vCPUs) in total. To go beyond these limits, users must submit a request that needs to be manually reviewed and approved by AWS prior to the creation of additional resources.

Moreover, the hardware abstraction, which is considered one of the biggest improvements brought by the Cloud Computing paradigm, is actually a painful aspect for traditional HPC use cases, where fine-grained knowledge of the underlying hardware is desired to optimize CPU-intensive applications. There are continuous debate and ongoing studies on the viability of executing specialized HPC applications in uncontrolled and

abstracted environments such as public cloud platforms (Richter, 2016).

In this context, we argue that Cloud Computing options can still be cost-effective for a range of HPC applications, given that users have access to a comprehensive software stack and frameworks to migrate existing applications seamlessly.

1.1 Motivation

In the HPC realm, the resources provided by public cloud services present an enticing alternative to costly on-site clusters. While it is true that the performance of public cloud infrastructure might not measure up to traditional physical hardware, they do possess certain compensatory attributes worth exploring. For instance, some providers offer substantial discounts on *ephemeral* Virtual Machines (VMs), a.k.a spot VMs, essentially renting out surplus infrastructure on a temporary basis. These machines are labeled as ephemeral because the provider can, without prior notification, repossess this infrastructure whenever a more lucrative long-term request is made by other clients.

The spot market unveils promising avenues for smaller research groups, enabling them to utilize highly-parallel infrastructures at considerably reduced costs. However, it is crucial for these applications to be fault-tolerant to ensure viable execution within such environments. Moreover, establishing a sustainable HPC cloud platform necessitates a robust software ecosystem. The need to fill software gaps, such as cost advisors, large contract handlers, *DevOps* solutions, automation APIs, and cloud-aware resource managers, is paramount to facilitate a fully operational HPC cloud platform (Netto et al., 2018), and one of the main motivators for our research.

While public cloud platforms offer numerous advantages, they do come with a distinct set of challenges that enterprises are actively studying and addressing. These include the issue of vendor lock-in, where switching platforms becomes difficult due to the unique services and infrastructure of each provider. Additionally, the absence of universal industry standards can complicate interoperability and integration efforts. The lack of transparency surrounding pricing structures can also be a concern, potentially leading to unforeseen cost escalations if resources are not meticulously managed. These accumulated challenges forced organizations to employ dedicated professionals whose primary responsibility is to manage their cloud resources and expenditures. In response to this need, a comprehensive set of management practices has emerged to address the fluctuating expenses associated with cloud resource usage, typically referred to as *FinOps* (Storment; Fuller, 2023).

The democratization of computational resources is set to confer invaluable benefits to scientific exploration and human progress. Consequently, surmounting these inherent challenges becomes essential to empower researchers in their pursuit of groundbreaking studies, spanning fields such as pharmacological discovery, climate modeling, genome mapping, and materials science, among others (Shalf, 2020).

1.2 Limitations of Existing Platforms and Strategies

Over recent years, cloud providers have launched numerous localized products and services, promoting them as HPC solutions. These offerings typically include infrastructure options featuring high-speed interconnections, such as InfiniBand, although they can be expensive. Cloud providers also provide large VM instances equipped with hundreds of vCPUs to cater to applications with substantial processing demands. Since the mid-2010s, the rental of Graphical Processing Unit (GPU)-powered machines has gained traction, particularly for accelerating Artificial Intelligence (AI) model training. Recently, NVIDIA, the largest manufacturer in the field, achieved a monumental milestone by reaching a market valuation of 1 trillion dollars[1]. This significant surge was largely driven by the escalating demand for GPU chips, a phenomenon propelled by the burgeoning interest in generative AI technologies.

Regarding software tooling, the most significant contributions have been centered around task scheduling and virtualization technologies. For infrastructure provisioning, for example, Amazon launched the AWS Parallel Cluster[2] tool in 2016 to assist developers in building clusters using Elastic Compute Cloud (EC2) instances from the command line. Microsoft Azure has Azure CycleCloud[3], a service which is basically a web interface to help users dynamically provision virtualized clusters with tools widely used in HPC preinstalled. Google launched their own Google Cloud HPC Toolkit[4], similar to AWS Parallel Cluster in functionality. However, despite their utility, these products and tools are exclusive to each provider, making migration between solutions challenging due to the absence of industry standards for Cloud Computing and HPC (Netto et al., 2018). There is also the obvious lack of interest from the cloud providers to give means for users to easily jump out of their platforms and join their competitors whenever they increase prices.

When it comes to utilizing ephemeral VMs for HPC, the predominant focus of research has been on bag-of-tasks applications (Malla; Christensen, 2020; Teylo et al., 2023). These are inherently easy to parallelize, and the eviction of an instance tends not to pose significant issues, since only the work performed by the evicted instance needs to be repeated. However, for tightly-coupled applications, instance eviction escalates into a substantial challenge. In these cases, the implementation of fault-tolerance strategies becomes critical, and they must be exceptionally efficient to prevent a substantial increase in runtime and monetary costs. Nevertheless, there is a noticeable scarcity of updated studies addressing the execution of such applications using public cloud infrastructure. Furthermore, current software tools designed to aid users in migrating and executing these applications cost-effectively are virtually non-existent.

[1] https://www.forbes.com/sites/dereksaul/2023/05/30/nvidia-hits-1-trillion-market-value
[2] https://github.com/aws/aws-parallelcluster/releases/tag/1.0.0
[3] https://learn.microsoft.com/en-us/azure/cyclecloud/overview
[4] https://cloud.google.com/blog/products/compute/new-google-cloud-hpc-toolkit

1.3 Goals and Contributions

Addressing the challenges highlighted earlier, the primary objective of this research is to *assess the cost-efficiency of current cutting-edge cloud computing technologies, while appraising their suitability for handling diverse HPC workloads, ranging from benchmarks to real-world applications*. Furthermore, this research aims to pinpoint the primary obstacles that researchers may encounter while transitioning to, or developing applications for, public cloud platforms. Consequently, we devise software solutions to counter these identified setbacks. These solutions are then integrated into *HPC@Cloud*, an open-source software toolkit developed in the context of this research. This toolkit is designed to facilitate the deployment of HPC applications on public cloud platforms with minimum effort, and also fill some software tooling gaps identified by related studies, such as *DevOps* and *FinOps* capabilities.

Overall, we bring the following contributions to state of the art on HPC and Cloud Computing convergence:

1. We produce a detailed cost-efficiency analysis of using standard and ephemeral public cloud VMs to execute a diverse range of Message Passing Interface (MPI) applications and benchmarks across two cloud providers, giving a clear picture of the pros and cons of using such type of infrastructure and a quick view of the main performance bottlenecks faced by these kinds of applications.
2. We propose and evaluate four configurable fault-tolerance mechanisms for Checkpoint/Restart (C/R) tailored for MPI applications running on ephemeral VMs. Each strategy have variations on how checkpointing is executed, with an in-depth analysis of performance and costs trade-offs of running them on a variety of different cluster configurations. The evaluated fault-tolerance mechanisms are:
    a) Berkeley Lab Checkpoint-Restart (BLCR).
    b) Distributed MultiThreaded CheckPointing (DMTCP).
    c) Scalable Checkpoint Restart (SCR).
    d) User-Level Failure Mitigation (ULFM).
3. We evaluate the performance impact and viability of a well-established containerization technology (Singularity) when used on top of hypervisor virtualization for running MPI workloads, testing benchmarks and real-world applications, discussing the main advantages of this technology.
4. We introduce an empirical method to estimate job execution costs on public clouds. This approach entails executing the target application with a small input problem size and collecting execution metrics. These metrics are subsequently utilized to train regression models, enabling the prediction of total execution time. By employing this solution, we aim to enhance the accuracy of estimating job execution costs on public clouds.

5. As a case-study, we perform the migration of *DynEMol*, a real-world physics simulation application, to AWS. We evaluate the application performance on a range of virtualized clusters, both persistent and ephemeral, as well as on a traditional on-premise HPC cluster, analyzing and comparing the operational costs.

The aforementioned scientific contributions were packed into a new open-source toolkit that leverages enterprise-grade technologies to help researchers migrate legacy HPC code to public clouds named *HPC@Cloud*. This toolkit was designed to work with different public cloud providers and is equipped with a versatile command line interface that empowers users to efficiently determine the required computational resources and configurations when configuring HPC infrastructure. It facilitates the seamless operation of MPI applications on the provisioned cluster, employing TCP/IP or InfiniBand (when available) for communication. Additionally, the toolkit provides inherent support for Singularity containers, thus enhancing portability when it comes to transitioning legacy HPC code to the Cloud.

1.4 Publications

The main contributions of this dissertation were published in high quality venues (Munhoz; Castro; Mendizabal, 2022; Munhoz; Castro, 2022b; Munhoz; Castro, 2023; Munhoz; Castro; Rego, 2023; Munhoz et al., 2023; Munhoz; Castro, 2022a; Ferrão; Munhoz; Castro, 2023; Althoff; Munhoz; Castro, 2023). Below is a list of research papers produced during the dissertation:

- **Vanderlei Munhoz**, Márcio Castro, Odorico Mendizabal. *Strategies for Fault-Tolerant Tightly-coupled HPC Workloads Running on Low-Budget Spot Cloud Infrastructures*. International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). Bordeaux, France: IEEE Computer Society, 2022. DOI: 10.1109/SBAC-PAD55451.2022.00037.
- **Vanderlei Munhoz**, Márcio Castro. *HPC@Cloud: A Provider-Agnostic Software Framework for Enabling HPC in Public Cloud Platforms*. Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD). Florianópolis, Brazil: SBC, 2022. DOI: 10.5753/wscad.2022.226528.
- **Vanderlei Munhoz**, Márcio Castro. *Benchmarking the Scalability of MPI-Based Parallel Solvers for Fluid Dynamics in Low-Budget Cloud Infrastructure*. XXII Escola Regional de Alto Desempenho da Região Sul (ERAD-RS). Curitiba, Brazil, 2022. DOI: 10.5753/eradrs.2022.19170.
- Lívia Ferrão, **Vanderlei Munhoz**, Márcio Castro. *Análise do Sobrecusto de Utilização de Contêineres para Execução de Aplicações de HPC na Nuvem*. XXIII Escola Regional de Alto Desempenho da Região Sul (ERAD-RS). Porto Alegre, Brazil, 2023. DOI: 10.5753/eradrs.2023.229787.

- Luiz Fernando Althoff, **Vanderlei Munhoz**, Márcio Castro. *Análise de Viabilidade do Perfilamento de Aplicações de HPC Baseada em Contadores de Hardware na AWS*. XXIII Escola Regional de Alto Desempenho da Região Sul (ERAD-RS). Porto Alegre, Brazil, 2023. DOI: 10.5753/eradrs.2023.230088.
- **Vanderlei Munhoz**, Márcio Castro. *Enabling the Execution of HPC Applications on Public Clouds with HPC@Cloud Toolkit*. Concurrency and Computation: Practice and Experience (CCPE), 2023. DOI: 10.1002/cpe.7976.
- **Vanderlei Munhoz**, Márcio Castro, Luis G. C. Rego. *Evaluating the Parallel Simulation of Dynamics of Electrons in Molecules on AWS Spot Instances*. Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD). Porto Alegre, Brazil: SBC, 2023. DOI: wscad.2023.235765.
- **Vanderlei Munhoz**, Antoine Bonfils, Márcio Castro, Odorico Mendizabal. *A Performance Comparison of HPC Workloads on Traditional and Cloud-based HPC Clusters*. International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW). Porto Alegre, Brazil: IEEE Computer Society, 2023. DOI: 10.1109/SBAC-PADW60351.2023.00026.

1.5 Work Organization

The remainder of this work is organized as follows. In Chapter 2, we cover the required theoretical background on HPC and Cloud Computing concepts. In Chapter 3, we discuss previous work and related research. In Chapter 4, we present our proposed public cloud toolkit for HPC and detail its implementation, as well as all strategies devised and applied to make it cost-effective. In Chapter 5, we present our evaluation method, including evaluated metrics, applications and execution environments, delving into comprehensive discussions about the obtained results. Finally, we draw our main conclusions and present valuable future studies related to our research in Chapter 6.

2 BACKGROUND

This chapter delves into the foundational aspects that underpin this work. We start by providing a comprehensive overview of the HPC context, followed by an in-depth exploration of the Cloud Computing paradigm. Then, we detail the ephemeral VM market within the AWS public cloud platform. Next, we delve into the fundamental virtualization and network technologies that form the core of Cloud Computing, discussing their potential impact on the execution of HPC workloads. Then, we discuss the ongoing HPC and Cloud Computing convergence, highlighting the state of the art and the main challenges that are hindering the full convergence of both fields. Finally, we present a brief overview of the benchmarks and the applications used in this research.

2.1 High Performance Computing

Traditional HPC usually *refers to the utilization of physical computer clusters for parallel processing of large-scale data and algorithms*. The execution of HPC workloads usually entails the collaborative effort of millions of processors, as their completion on a single computer would be impractical within a reasonable timeframe. These computer clusters, commonly known as *supercomputers*, were traditionally constructed using costly and specialized circuitry. Nevertheless, technological advancements in parallel computer architectures have driven their increased adoption in recent decades. This trend is primarily driven by the limitations of heat dissipation when attempting to build Central Processing Units (CPUs) with higher clock speeds. Within the HPC domain, parallelization is indispensable for efficient computation.

HPC applications are typically optimized to run on specific hardware, leveraging specialized processor instructions for specific operations. Various types of processor cores, known as accelerators, such as GPUs and Field-Programmable Gate Arrays (FPGAs), can be employed to enhance performance for specific tasks within large algorithms. This architecture, known as *heterogeneous computing*, has gained significant popularity in fields such as cryptocurrency mining and AI. Another example of an accelerator is the *quantum computer*, capable of leveraging quantum properties of matter to perform certain algorithms exponentially faster than traditional algorithms executed on standard CPUs.

A considerable portion of advancements in science and engineering relies heavily on HPC, as it provides solutions to complex problems such as Deoxyribonucleic Acid (DNA) sequencing, real-world physics simulations, and climate modeling, all of which involve vast amounts of data.

2.1.1 Parallel Computer Architectures

Parallel algorithms commonly utilize *data parallelism*, *task parallelism* or both. Data parallelism entails the concurrent execution of identical tasks on distinct data segments using multiple threads or processes. This approach proves beneficial in scenarios involving grid problems, where data parallelism can be effectively applied. On the other hand, task parallelism involves the simultaneous execution of diverse tasks by multiple threads or processes. Task parallelism finds its utility in signal processing algorithms and similar cases. It is important to note that most real-world applications exist on a spectrum between data and task parallelism, combining elements of both approaches to achieve optimal performance.

At the hardware level, parallelism can be achieved in several ways: from Very Large Scale Integration (VLSI) technology that made possible the fabrication of large number of transistor components into the same chip, to modern multicore and multiprocessor architectures. Multicore systems refer to processor units with multiple cores, which can be integrated into a single die or onto multiple dies, however inside a single chip. In the mid 2010s, the computer industry swiftly pivoted processor architectures from single-core to multicore systems, specially to mitigate overheating problems associated with high operation frequencies. Nowadays, multicore systems are widely present on commodity hardware, and are used across many application domains. Multicore systems are more energy efficient, and with this paradigm shift and technology diffusion at desktop and mobile computers, modern applications started to require parallel and concurrent implementations to efficiently operate (Hill; Marty, 2008).

Multiprocessor systems refer to computers with multiple CPUs sharing the same system bus, memory and Input and Output interfaces (I/O). These kinds of systems allow us to solve problems that cannot fit inside a single CPU, or cannot be solved in a reasonable time by a single processor. Most HPC systems are based on multiprocessor architectures, typically connected through special grade high-speed networks, and possibly having dedicated links between GPUs for even faster processing. Fault tolerance is a must for HPC clusters, as tasks may execute for several days, or even weeks, and CPU failures are bound to happen as the number of components increase up to millions.

In parallel computer architectures, the memory and other resources can be shared or distributed among processor units, and developing applications targeting multiple threads or processes is a complex task. With this technological shift, several programming models and specifications were developed to help programmers write proper parallel code, some of them prevalent in the HPC context, such as MPI. We briefly discuss the main parallel programming models used for HPC in the next section.

2.1.2 Parallel Programming Models

In this section, we briefly discuss some of the most used parallel programming models, specially in the HPC context. These include OpenMP and Message Passing Interface (MPI).

OpenMP is a shared-memory programming model for C, C++, and Fortran launched in 1997 by the OpenMP Architecture Review Board (ARB), a consortium of hardware industry leaders and software vendors. This programming model was created with the assumption of multiple lightweight processing elements managed outside of OpenMP itself, such as Operating System (OS) threads or some other type of processor unit, each one of them with equal-time access to a shared address space (Mattson, 2001). Thus, in the OpenMP model, threads communicate by sharing variables, a data-access pattern subject to unintended sharing of data known as *racing conditions*.

It uses the *fork-join* model of parallel execution: the program begins with a single thread (a.k.a *master thread*), which executes sequentially until the first parallel region construct is encountered. Inside a parallel region, OpenMP automatically creates and manages multiple *worker threads* that execute the code inside the parallel region in parallel. Parallel regions and other basic constructs of OpenMP are specified through the use of compiler directives, which are embedded in C/C++ or Fortran source code. During the compilation process, these directives are used to generate parallel code in a transparent way. Overall, OpenMP features a wide range of directives to parallelize loops, to synchronize threads and to execute parallel tasks with or without task dependencies.

Since OpenMP only works with shared-memory architectures, it can only be used to achieve *intra-node* parallelization in a HPC cluster. Thus, another solution must be employed for *inter-node* parallelization, enabling applications to run on all nodes of an HPC cluster. The MPI is largely the most famous distributed programming model for inter-node parallelization in the HPC domain. The specification of MPI is maintained by the Message Passing Interface Forum (MPIF), an open group of parallel computing experts. There are several implementations of the MPI standard specification, supporting a wide range of operating systems and network standards. OpenMPI[1], MPICH[2], and MVAPICH[3] are popular implementations of MPI, although there are many other open-source and also proprietary distributions.

MPI implementations provide abstractions supporting several features defined by the MPI specification, working like middlewares. Upper-level applications make calls to MPI functions to handle virtual topology, process synchronization and communications through language-independent abstractions, including both point-to-point (communication between two specific processes) and collective communications (communications in-

[1] https://www.open-mpi.org
[2] https://www.mpich.org
[3] http://mvapich.cse.ohio-state.edu

volving all processes in a group). MPI programs are typically composed by processes bound to CPU cores at runtime by the workload agent or job manager, called *mpirun* or *mpiexec*.

Although MPI is the main programming model used in HPC applications since its inception, this programming model can be considered complimentary to shared-memory models such as OpenMP, and are they often used together.

2.2 The Cloud Computing Paradigm

Academic literature defines the concept of Cloud Computing as an *Information Technology (IT) paradigm representing the omnipresent access to configurable shared resources, that are capable of being rapidly provisioned with minimum effort through the Internet* (Buyya; Broberg; Goscinski, 2011). A similar definition was proposed by Mell & Grance (2011), defining the Cloud Computing paradigm as *a pay-per-use model which allows the convenient on-demand access to a configurable group of computing resources (networks, servers, storage, applications, software, and more) in a rapid manner with minimum effort and contact with the provider*. In simple terms, the Cloud Computing paradigm can be described by the extension of the public utility business model to the computing infrastructure world.

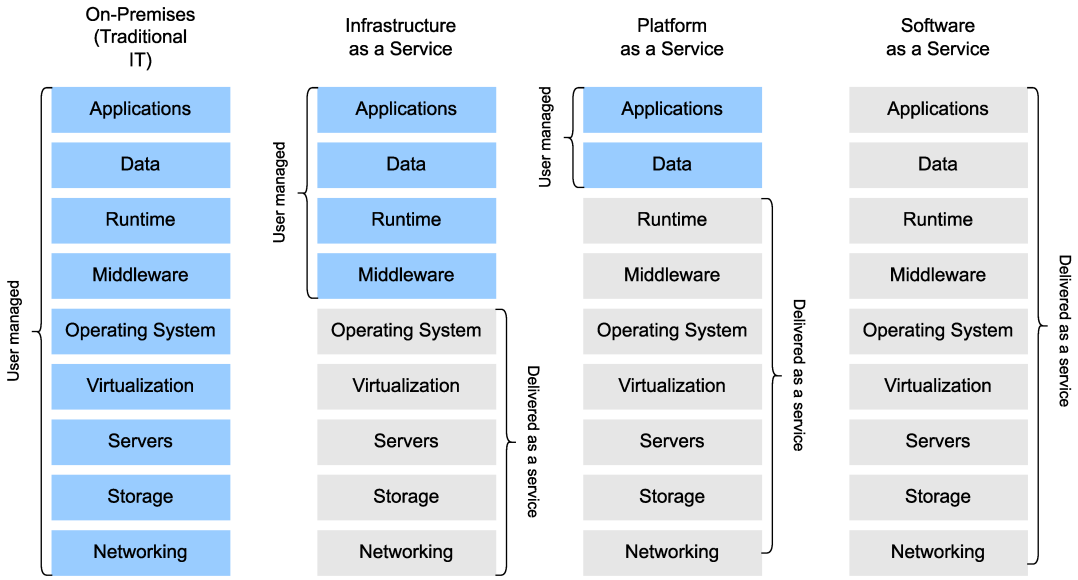
Cloud Computing platforms are typically categorized into three types: *public*, *private*, and *hybrid*. *Public clouds* are designed to be accessible to anyone via the Internet without needing long-term contracts or direct interaction with the provider. In contrast, *private clouds* are typically accessible only to specific organizations or user groups with appropriate access permissions. Finally, *hybrid clouds* allow organizations to connect public and private clouds, enabling them to leverage both benefits while avoiding the risks of relying solely on a single provider or on-premise infrastructures. In this research we are mainly interested on leveraging low-budget public cloud infrastructure resources for HPC.

2.2.1 Cloud Service Models

Most cloud providers offer their products following the Service Oriented Architecture (SOA) model. Services are classified based on three main types, standardized by the American National Institute of Standards and Technology (NIST): Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). These service types are commonly depicted by a stack chart as presented in Figure 1, showing what are the client's and the provider's responsibilities.

IaaS offerings refers to online services that provide Application Programming Interfaces (APIs) for users to spawn and manage compute infrastructure, including low-level details like network, storage, and backups. Cloud providers maintain hardware in several

Figure 1 – Common cloud service models compared to traditional IT.



Source: Adapted from (Al-Roomi et al., 2013)

data centers, giving users access to compute resources in the form of *instances*. A VM is a type of instance virtually allocated over a shared physical infrastructure and is usually the most commonly available and affordable option. Cloud providers offer various types of instances that vary in characteristics such as hypervisor type, CPU architecture, number of vCPUs, amount of memory, and more, allowing customers to choose the instance that best suits their needs. In the context of HPC, certain public cloud platforms offer specialized solutions specifically designed for demanding applications. These solutions include bare metal[4] instances, hardware accelerators, and high-speed communications. However, these services are often available in limited capacities and have a considerably higher price tag than standard VMs.

PaaS products refers to services that the user can use to create and deploy custom software applications using a configurable environment hosted by the cloud provider. Runtime, middleware and software features are abstracted from the user, and managed by the provider. SaaS products are ready-to-use software applications with specific purposes that the provider typically offers through APIs. Users can then use these APIs directly into their applications.

In this research, we focus mostly on IaaS offerings, where the user has the capacity to configure virtual or bare-metal clusters of machines for HPC.

2.2.2 Ephemeral Infrastructure in Public Cloud Platforms

Cloud providers can optimize resource usage by renting out spare infrastructure at significant discounts, commonly known as the *spot market*. Spot machines are ephemeral

[4] Bare metal instances provide users with direct access to the hardware without any virtualization.

instances that cost considerably less per hour than persistent (standard) machines and run on the same type of infrastructure with identical performance and characteristics. As organizations increasingly aim to minimize infrastructure expenses, most leading cloud providers have introduced a variant of spot market for IaaS. Nonetheless, there is a caveat: the cloud provider reserves the right to reclaim spot machines on short notice to accommodate persistent instances, which necessitates the user to proactively safeguard data and prepare for application recovery in the event of instance repossessions.

Each cloud provider operates under its own unique rules and pricing mechanisms. In this dissertation, we focus on discussing the AWS spot market. When requesting a spot machine on this platform, users must bid the maximum price they are willing to pay per hour for the requested resources. The provider will only fulfill requests if the specified spare capacity is available at a price lower or equal to the bid value. This model differs significantly from on-demand pricing, where an instance request is met instantly.

AWS provides historical spot pricing data for the public, although only during a limited past timeframe. Supply and demand changes are opaque to the cloud consumers during runtime, however AWS spot prices are quite steady and predictable. Machines with large numbers of vCPUs or accelerators (GPUs) typically have few spot instances available at a time, being the smaller standard x86 and ARM machines the commonly used spot instances. AWS also provides a messaging system that alerts the consumer within a short notice of two minutes before a spot instance repossession occurs. Such alerts are helpful for preemptively preparing a recovery strategy, even without a complex failure prediction model. Users are also able to define expiration times for their spot requests (the maximum wait time before canceling a bid and giving up). This feature is also explored in our cluster creation strategies.

It is essential to note that spot requests have no guarantee from the provider to be met, making this characteristic a crucial detail in contexts where workloads have a defined deadline for completion. Furthermore, spot prices for each instance type often fluctuate dynamically based on market demand in each availability zone, although this may vary depending on the provider. For example, Oracle offers a fixed discount for their spot machines, regardless of the available capacity.

2.2.3 Low-Budget Providers: Vultr Cloud

Vultr is an alternative cloud provider known for low prices when compared to major market players, and their target audience are individual software developers and startups. Unfortunately, Vultr does not offer spot VMs at the time of this research. Nevertheless, we conduct some experiments using non-transient infrastructure from Vultr, as this provider is also a popular low-budget infrastructure option and have substantial discounts compared to other cloud platforms. Vultr also lacks some features oriented to HPC which are present at AWS, such as InfiniBand support, hardware accelerators, and

high-performance storage solutions such as Lustre. Moreover, Vultr instances are billed in hourly increments, in contrast to AWS which bills VMs in seconds increments.

2.3 HPC and Containerization

Containerization has become essential in modern software development due to its ability to accelerate and streamline application development and deployment processes (Casalicchio; Iannucci, 2020). Containers are widely used in Cloud Computing environments, enabling fast and scalable application deployment. Moreover, containerization tools like Docker and Kubernetes have become fundamental tools, helping organizations develop applications more efficiently (Kithulwatta et al., 2022). Although containers are lightweight compared to VMs, they can still impact performance, which may be a concern for some HPC applications. Nonetheless, the benefits of containerization for most other use cases far outweigh the potential drawbacks.

Over the years, containerization tools tailored for HPC have been developed to address the performance drawbacks of traditional containerization solutions. Sarus (Benedicic et al., 2019) and Singularity (Kurtzer; Sochat; Bauer, 2017) are two such tools that have gained popularity in the scientific community. Singularity, in particular, was designed specifically for HPC environments and has been proven to have minimal performance footprint (Zhang; Lu; Panda, 2017), given that Singularity containers use fewer resources and have lower latency than containerization technologies like Docker. Singularity allows users to bundle the entire software environment, including the operating system, libraries, and dependencies into a single portable container that can be run on different HPC clusters.

Singularity offers several key features that make it an excellent fit for HPC environments, including seamless integration with tools such as job schedulers, MPI, InfiniBand networks, and GPUs. It is also easy to use, even for users unfamiliar with containerization or HPC systems, and is widely adopted for research computing, scientific simulations, and other types of workloads.

2.4 HPC and Cloud Convergence Challenges

In the context of HPC, the public-utility pricing model of IaaS offerings provides an appealing alternative for budget-constrained researchers and organizations. This is because it eliminates the need for expensive on-premise specialized hardware and substantially reduces the capital and technological entry barriers for HPC. Public cloud platforms have evolved to support the rapid allocation and deallocation of virtualized resources, allowing users to scale infrastructure dynamically based on their needs. As a result, most companies use cloud resources to ensure service availability and disaster recovery capabilities for their enterprise systems. In most cases, these companies do not require low-level knowledge of the underlying hardware or high-speed communica-

tion between nodes, both of which are typical requirements for HPC workload (Santos; Cavalheiro, 2020). Instead, for most enterprise applications, the primary requirements are resiliency and fault tolerance, which can be achieved through redundancy: replicas of the same service are executed in different geographical locations to avoid single points of failure. The HPC use case, however, tends to be the opposite as nodes are allocated as close as possible to each other to reduce communication latency.

Although spot markets and alternative low-cost cloud providers may be appealing options for small organizations, traditional HPC still has a long way to go before effectively utilizing cheap cloud resources. Due to the possibility of spot instance repossessions during job execution, fault tolerance strategies are imperative for executing stateful applications, which is typical for most HPC workloads. Furthermore, since available capacity varies between different instance types, careful consideration is necessary when selecting instance flavors. The number of instances repossessions can be high and significantly slow down execution, even when using low-overhead fault tolerance mechanisms. Additionally, estimating optimal bids for minimum costs is a challenging problem due to the large optimization space and the need for more available information from most cloud providers. These providers often lack transparency regarding pricing dynamics and historical data, further complicating cost optimization efforts.

Unfortunately, the software ecosystem required to establish a viable HPC cloud platform is not yet fully developed (Netto et al., 2018). There is a deficiency in cost advisors, large contract handlers, *DevOps* solutions, automation APIs, and HPC-aware resource managers. For example, cost advisors commonly used in standard cloud environments must not only be capable of predicting the duration of a user's HPC jobs but also estimate how long an experiment with an unknown number of jobs will take to execute entirely. Additionally, it is necessary to have HPC-aware resource managers that can handle specific HPC workflows and guarantee high levels of performance, which typical cloud resource managers do not offer. Furthermore, large contract handlers and automation APIs that can manage thousands of machines simultaneously are needed to enable HPC workflows at scale. Such solutions are necessary for adopting HPC on cloud platforms and necessitate new solutions to overcome these challenges.

Existing studies suggest that the traditional pricing models for public cloud services are not tailored to the unique requirements of HPC workloads (Al-Roomi et al., 2013). Since public cloud resources are shared among multiple users, the performance of HPC workloads can vary substantially each time they are tested, which makes *DevOps* for HPC a complex and challenging task. Moreover, the lack of specialized tools for managing HPC workloads in the cloud, such as HPC-aware resource managers and automation APIs, makes optimizing the performance and cost of these workloads in the cloud environment challenging. However, state of the art in Cloud HPC is characterized by rapid innovation, with cloud providers continually adding new capabilities and features to support the needs of HPC users. The availability of high-speed interconnects

such as InfiniBand and RDMA over Converged Ethernet (RoCE) on the cloud has led to improved performance for parallel computing workloads by allowing low-latency communication between compute nodes. Containerization technology, such as Singularity, is also being used to simplify the deployment and management of HPC applications on the cloud, increasing portability and agility.

Additionally, HPC is being used as a foundation for Machine Learning (ML) and AI workloads, with specialized hardware such as Tensor Processing Units (TPUs) offered by cloud providers to accelerate Deep Learning models. Many organizations are adopting hybrid cloud architectures that combine on-premises infrastructure with cloud resources to achieve optimal cost and performance for their HPC workloads. Integrating these cutting-edge technologies propels innovation in HPC and expands its reach to a broader audience. Nevertheless, it is crucial to acknowledge that many researchers still face challenges due to software and technical limitations when attempting to transition from traditional infrastructure to cloud-based computing.

2.5 Applications and Benchmarks

In this section, we give a brief overview of the applications and benchmarks used in our experimental evaluation.

2.5.1 NASA Advanced Supercomputing Parallel Benchmarks (NPB)

The NAS Parallel Benchmarks (NPB), are a suite of programs developed by NASA to help evaluate the performance of parallel supercomputers. These benchmarks are derived from Computational Fluid Dynamics (CFD) applications and are intended to assist in the performance analysis of parallel architectures, parallel processing compilers, and parallel algorithms. They act as a standard measure, providing a vast number of insights into different aspects of parallel computing. Over the years, these benchmarks have been widely used within the HPC community to evaluate the performance of parallel processors and environments.

Among the various workloads provided by NPB, we focused on three notable ones (EP, FT, and IS), which provide unique challenges to the system under test and showcase different facets of parallel computing performance:

1. **EP (Embarrassingly Parallel)**: is designed to generate pairs of Gaussian random deviates, emphasizing the floating-point performance of a system without the complications of inter-process communication.
2. **FT (Fast Fourier Transform)**: focuses on the computation of a three-dimensional discrete Fourier transform, which involves intense floating-point operations and all-to-all communications between processes.

3. **IS (Integer Sort)**: is centered around an integer sort operation using the bucket sort algorithm. It emphasizes integer computation and tests the communication capability of the system, particularly the all-to-all communication pattern.

In this research, we used NPB version 3.4.2, with workloads implemented and C and Fortran using MPI and OpenMP. NPB source code can be found at https://www.nas.nasa.gov/software/npb.html.

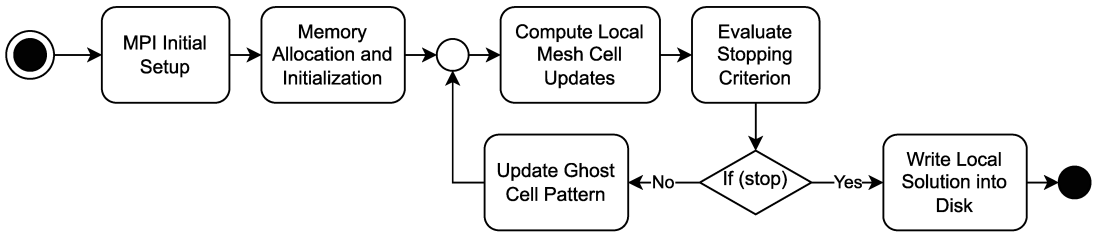
2.5.2 Parallel Implementation of 2D-Heat Diffusion Equations (HEAT)

We developed our own parallel solver implementation for Laplace's heat diffusion equations based on the iterative Jacobi method (Hadjidimos, 2000), implemented in C. This application is named *HEAT*, and its main objective is to be used as a simple template for testing application-level fault tolerance methods, such as SCR and ULFM, both detailed in Section 4.3.

We consider a 2-dimensional square region with non-periodic borders as the physical domain to be simulated. This domain is then broken into subdivisions and distributed to each MPI rank for iterative computation, resulting in a Cartesian topology of MPI processes. Subdivisions are made following a one-dimensional decomposition strategy, and global boundaries have a constant temperature value. Each subdivision needs information from its neighbors to compute each iteration at the internal borders. To reduce messaging, we employ a ghost-cell pattern updated at the end of each iteration (Kjolstad; Snir, 2010). Thus, given an initial state, the application will iteratively solve the discrete heat diffusion recurrence formula in each mesh point until a convergence criterion or a defined maximum number of iterations is reached. An action diagram of the parallel algorithm is presented in Figure 2.

At the initial stages, the MPI world is initialized and memory is allocated locally for each rank. The workload distribution is made through a Cartesian MPI process topology employing a slab decomposition strategy, where each process will compute an equal chunk of the total mesh. Local updates are computed applying the heat diffusion recurrence equation at each mesh cell. After updating the entire local grid, we

Figure 2 – Action diagram of the *HEAT* application logic.



Source: Produced by the author.

Algorithm 1 – *HEAT* application iterative logic.

```
1: MPI initial setup
2: memory allocation and initialization
3: repeat
4:     update ghost cell pattern
5:     compute local mesh cell updates
6:     all-reduce residual error with all ranks
7:     if criterion (residual or iterations) reached then
8:         converged = true
9: until converged
10: write solution into disk
```

check a stopping criteria (maximum number of iterations reached, for example) and then write the solution to disk if completed. In case the simulation did not reach the solution, we update the ghost cell pattern by exchanging border cells information between neighbors and start a new iteration. To get information on each neighbor process we implemented a `get_neighbors()` function which returns the Cartesian coordinates and rank of all neighbors of the current process. To get the Cartesian coordinates we use the `MPI_CART_SHIFT` function[5]. In our implementation we use `MPI_SENDRECV` to update the ghost cells between neighbor ranks. Algorithm 1 shows the iterative *HEAT* application logic as pseudo-code. The source code of the *HEAT* application is available at: https://github.com/vanderlei-filho/jacobi-method.

2.5.3 DynEMol Simulation Software

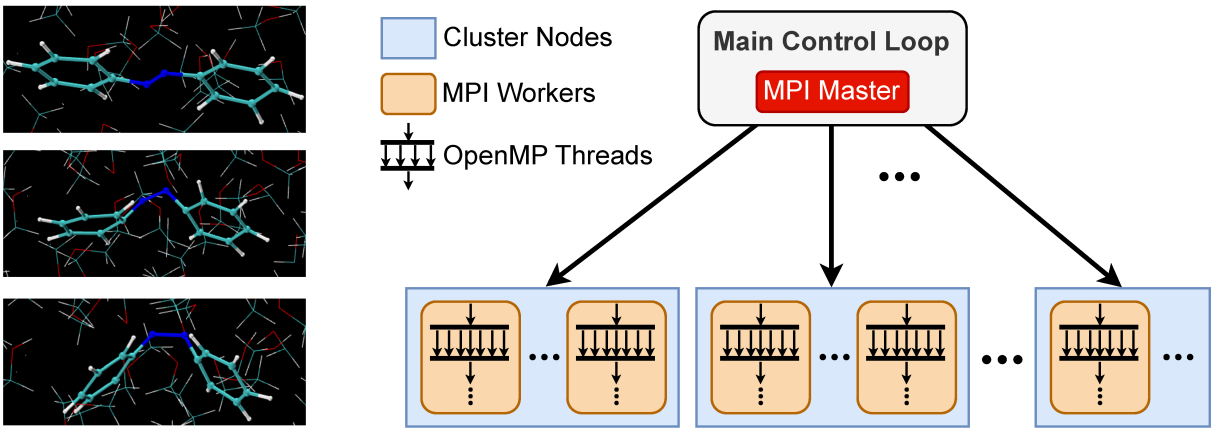
Dynamics of Electrons in Molecules (*DynEMol*[6]) is a sophisticated simulation tool for accurately representing how molecules behave when they are attached to large solid surfaces and during processes where electrical charge is transferred. It performs simulations of the excited-state dynamics of molecular systems in the solid and liquid phases. This software package is used as case study in this research, in partnership with the Department of Physics of University of Santa Catarina (UFSC).

DynEMol's simulation method combines tight-biding Quantum Mechanical (QM) with classical Molecular Mechanics (MM) formalisms in a semi-empirical hybrid quantum-classical model capable of simulating the non-adiabatic dynamics of large atomistic models at the lowest computational cost. This method provides the tools for studying a variety of photo-induced effects, including the charge and energy transfer dynamics in large molecular and nanostructured materials subject to complex structural deformations (Torres et al., 2018; Abraham; Rego; Gundlach, 2019). Simulations are carried out within the framework of the self-consistent Ehrenfest method and the Coherent Switching with

[5] https://www.open-mpi.org/doc/current/man3/MPI_Cart_shift.3.php
[6] https://luisrego.sites.ufsc.br/

Figure 3 – Overview of DynEMol simulation results and parallelization.

(a) DynEMol Output. (b) DynEMol Parallelization Strategy.



Source: Produced by the author.

Decay-of-Mixing (CSDM) method (Shu et al., 2020).

DynEMol is written mainly in Fortran and leverages OpenMP and MPI to execute compute-intensive tasks in parallel. The dynamics simulation consists of a main loop (the *time loop*) that executes fifty to a hundred thousand time-steps. Parallel computations are restricted to interaction calculations in a given time slice. A master MPI process (*rank 0*) undergoes the entire time evolution of the simulation and exchanges data with MPI worker processes distributed across the cluster. It also executes a few compute-intensive tasks, such as matrix diagonalizations. Most MPI worker processes are set to dwell in specific subroutines, particularly the ones that calculate the forces on the atoms. A smaller number of MPI worker processes are used for eigenvalue and eigenvector calculations as well as matrix inversions.

Figure 3(a) shows an example of a simulation output by *DynEMol* (in this case, an azobenzene molecule in ethanol), whereas Figure 3(b) shows an overview of the hybrid MPI/OpenMP parallelization strategy adopted in *DynEMol*. Parameters of the parallel solution include the number of: (i) HPC cluster nodes; (ii) MPI worker processes per cluster node; and (iii) OpenMP threads per MPI worker process. A variety of communications are used to send tasks and receive solutions to/from MPI processes. *DynEMol* achieves better performance when the number of MPI workers and OpenMP threads are fine-tuned to the target system based on its atomic size. Linear algebra calculations, such as matrix diagonalization and matrix inversion, benefit from OpenMP threads, whereas the atomic force calculations are efficiently parallelized via MPI processes.

DynEMol also employs a data fault tolerance mechanism that preserves the state of simulations in a set of files. The frequency of these backups, determined by the number of iterations, can be adjusted to suit the needs of the simulation.

3 RELATED WORK

Most research and industry efforts regarding HPC and Cloud Computing is focused on understanding the cost-benefit of moving legacy applications from on-premise clusters to a public cloud platform. Research in the field is also largely focused on how to extract the best performance possible from unknown underlying hardware, connected through a much slower and unreliable network.

Given the broad range of topics involved in our work, we organize the related works in separate sections.

3.1 HPC and Cloud Computing Convergence

The utilization of public cloud resources for HPC has garnered significant attention in the past years. Numerous studies have delved into the challenges and opportunities of leveraging cloud resources for HPC workloads, addressing performance, cost, and security concerns. For instance, a recent study by Guidi et al. (2021) showed that contemporary high-end Cloud Computing could provide HPC-competitive performance at moderate scales. Aljamal, El-Mousa & Jubair (2018) produced an in-depth survey of existing HPC offerings in four major public cloud providers, although this study is quite outdated by now given the fast evolution of cloud platforms in the past years.

Malla & Christensen (2020) evaluates the usage of Function as a Service (FaaS) and IaaS offerings at GCP for an embarrassingly parallel application, evaluating the cost-effectiveness of each type of compute service for this particular workload. Although we do not consider FaaS services in our research, given that we focus on tightly-coupled MPI applications which are not suited for parallel execution with self-contained functions, we produce a more in-depth analysis of VM usage for a more diverse range of HPC applications and cloud providers, with additional fault tolerance strategies for reliable execution with ephemeral instances.

Peña-Monferrer, Manson-Sawko & Elisseev (2021) proposed a framework for executing CFD workloads based on an hybrid cloud architecture. The majority of the computational load is executed in a traditional on-premises HPC cluster, while the data processing of the results is done using public cloud infrastructure. Our research has a different scope and infrastructure, since we focus on bringing HPC workloads entirely to the public cloud.

Other studies concentrate on providing access to applications on public cloud platforms rather than developing and running them directly in the cloud. Wong & Goscinski (2013) introduced a framework for deploying and offering HPC applications as SaaS solutions in public clouds. While their research addresses the provisioning of HPC applications, our study investigates the feasibility of executing HPC workloads using public

cloud infrastructure. In addition, our study delves deeper into the cost-effectiveness of running these workloads on ephemeral instances at AWS.

Despite the existence of provider-specific tools, such as *AWS ParallelCluster*[1], there is still a need for provider-agnostic software that can make it easier for the HPC community to build HPC clusters with public cloud resources with minimal effort. In our work, we propose *HPC@Cloud*, a software framework that addresses some of the challenges and limitations of using public cloud resources for HPC workloads. The key differences from previous technology such as *AWS ParallelCluster* lies in the built-in support for spot cluster restoration, costs prediction capabilities, and a provider-agnostic architecture. We demonstrate the effectiveness of *HPC@Cloud* through an experimental analysis on two public cloud platforms: AWS and Vultr cloud. While our research also employs microbenchmarks to assess performance gaps, it primarily focuses on low-budget instances lacking high-speed memory systems and interconnects featured in prior studies. Additionally, we delve deeper into the actual costs of deploying HPC applications to the cloud, a complex task that may ultimately constitute the most significant expense associated with cloud-based application execution.

3.2 Ephemeral Cloud Infrastructure for HPC

Several studies explore the feasibility of using ephemeral instances for HPC applications, specially AWS spot instances. Teylo et al. (2023) comprehensively assessed the use of AWS spot instances for scheduling bag of tasks applications. We consider their research complementary to ours, offering valuable insights into employing spot instances. Our study extends their work by introducing a generalized software toolset for migrating tightly-coupled HPC workloads to public cloud platforms. Zhou et al. (2022) proposed a framework for optimizing costs with spot instances. Again, their research is complementary to ours, as their framework is focused solely on predicting prices in the AWS spot market without considering the use of AWS on-demand instances (or other provider's infrastructure) to minimize costs and the actual execution of HPC workloads on these instances with fault-tolerant mechanisms.

Dancheva, Alonso & Bartoň (2023), and Fernandez (2022), present a broad analysis of IaaS from public cloud providers, using micro and macro benchmarks to assess the performance of MPI operations across various vendors and architectures. Our research differs in several key aspects: (i) we evaluate real-world HPC applications, such as *DyNEmol*, which is a distinct departure from the use of microbenchmarks in the referenced studies; (ii) we address the challenges of migrating and managing such HPC applications in a public cloud environment, which is not explicitly addressed by the aforementioned studies; (iii) we focus on strategies for further reducing costs associated with cloud-based

[1] Available at: https://aws.amazon.com/hpc/parallelcluster/

HPC, such as leveraging transient virtual machines like AWS spot instances, rather than just relying on on-demand pricing models; and (iv) our study also delves into a practical application of a fault tolerance technique tailored for AWS spot clusters, taking advantage of a eviction notification system to minimize checkpoint frequency and overhead. Therefore, our research offers a more applied perspective on cloud-based HPC, addressing both the performance and operational challenges associated with such an approach.

Sena et al. (2023) presented a comprehensive exploration of the potential advantages users might extract from the diversity of instances and contract models offered by public cloud providers, aiming to reduce financial expenditure. Their research delineates a methodology for dynamically scheduling applications subject to deadline constraints across both spot and persistent instances. While their focus on spot instances aligns with ours, our research takes a different approach. Rather than emphasizing scheduling strategies and pricing model analyses, we concentrate on the practical nuances of running real-world applications using the aforementioned infrastructure. Thus, our work offers valuable insights into the application side of leveraging such resources.

Marathe et al. (2014) researched redundancy for the cost-effectiveness of time-constrained HPC applications on AWS, and Marathe et al. (2014) explored techniques to minimize costs for running time-constrained applications on AWS spot machines making use of redundancy and checkpointing for fault tolerance. Nevertheless, both studies, although valuable, are not applicable anymore due to the overhaul of the AWS spot market bidding and pricing schema in 2018[2].

Within the scope of cloud costs forecasting, Zhou et al. (2022) proposed *FarSpot*, a framework that centers on forecasting AWS infrastructure expenses using Machine Learning ensemble methods, such as Xgboost. While aligning with the broader theme of cost optimization, our study serves a distinct purpose and offers a complementary perspective. Rather than developing a generalized machine learning model for predicting spot instances costs, we explore the application of an empirical solution based on sample workload execution, which can be applied over any type of cloud infrastructure.

Gong, He & Zhou (2015) provided a detailed analytical model that estimates the costs of using fault-tolerant techniques in the AWS spot market. This study observed that checkpoints help reduce the cost of failures while replicated execution reduce the risk of failures. Similar to our work, they used the BLCR library for a rollback-recovery approach for fault tolerance. We expanded the analysis with experiments considering multiple cloud providers and different strategies, such as application-level rollback restart with ULFM. We also advanced the state-of-the-art within HPC in the cloud with an empirical cost prediction strategy for tightly-coupled MPI applications, with a detailed cost-effectiveness evaluation.

Furthermore, in the case of tightly-coupled applications, fault tolerance is impera-

[2] Available at: https://aws.amazon.com/blogs/compute/new-amazon-ec2-Spot-pricing/

tive for efficient execution and there is a research overlap between ephemeral infrastructure and fault tolerance.

3.3 Fault Tolerance for MPI Applications

We integrated into *HPC@Cloud* existing fault tolerance strategies to address the unreliability of spot instances (Munhoz; Castro; Mendizabal, 2022). These strategies are built upon existing technologies such as BLCR and the innovative ULFM, which is currently under development by the MPI Forum. Although numerous studies have explored fault tolerance using BLCR and other well-established technologies, such as (Hargrove; Duell, 2006; Wang et al., 2007; Hariyale et al., 2012), they do not comprehensively analyze the advantages and disadvantages of utilizing more affordable, less reliable cloud instances compared to high-end options in a public cloud and HPC context. We also propose and evaluate new adaptive fault tolerance strategies based on ULFM that does not stop the MPI application when AWS revokes spot instances, thus minimizing idle infrastructure and providing better performance than blocking fault tolerance strategies.

Brum et al. (2023) offers a comprehensive review of the fault tolerance techniques most commonly employed by applications operating within cloud and HPC environments. Their focus primarily revolves around checkpoint-rollback and replication strategies, in addition to exploring fault detection approaches and existing reliable storage solutions within the cloud. In our study, we empirically examine several adapted checkpoint-rollback methods for ephemeral cloud infrastructure. We consider our work complementary, as our analysis contributes theoretical and practical insights to the existing landscape, enhancing our understanding of how these fault tolerance techniques perform in real-world Cloud Computing scenarios.

Amoon et al. (2019) suggested a design method that centers around reactive checkpointing. This approach dynamically tailors the checkpoint frequency to the changing circumstances of the cloud environment. While their work sets an important theoretical groundwork, our study provides a unique and complementary angle. We delve into a real-world fault-tolerant HPC application that leverages reactive checkpointing, introducing a method intended to decrease checkpoint overhead by harnessing AWS spot instances' eviction notification systems. Characterized by a practical approach, our research is supported by empirical data, demonstrating the practical benefits of reducing checkpointing frequency in real-world applications.

3.4 Containerization for HPC

Numerous other studies focused on developing workflows for HPC in public cloud environments, including the work of Vaillancourt et al. (2020), centering on HPC applications portability. Their research employs Docker and Terraform to build clusters on

AWS and test multi-VM MPI applications, a method similar to ours. However, this dissertation expands upon this by assessing containerization technologies designed for HPC, such as Singularity, and utilizing cost-effective AWS spot instances to further reduce expenses. Furthermore, we introduce and develop a software toolkit that enables users to seamlessly customize their infrastructure according to their requirements, eliminating the need for prior cloud-specific technical expertise.

Sindi & Williams (2019) shared insights from their implementation of a containerized HPC environment based on Docker for running multi-VM MPI workloads, investigating container migration techniques for fault tolerance. Their primary approach involved using Checkpoint/Restore In Userspace (CRIU) technology to migrate containers to different hosts in anticipation of potential failures. Although their research also addresses the containerization of HPC applications, it does not focus on executing them on public cloud platforms, neither analyzing the associated workflow costs.

Overall, there is a vast overlap between research areas that are important for enabling HPC in public cloud platforms, and our work tackle most of these issues. We provide an end-to-end analysis and solutions for the main hurdles and challenges faced by scientists and organizations that wish to tap cheap compute resources.

3.5 Discussion

In Table 1, we conduct a comprehensive comparison between the topics addressed and the contributions made in this dissertation with those covered in the related work discussed in this chapter. Our evaluation criteria encompass a diverse range of aspects, including: (1) studies on cloud computing and their contributions towards enhancing HPC through the use of **public** cloud infrastructure; (2) innovative solutions and significant contributions in the realm of IaaS management, alongside the development of specialized software toolkits in this domain; (3) in-depth analyses on task scheduling methodologies; (4) empirical research utilizing spot or ephemeral instances on public cloud platforms; (5) exploration of Burstable instances within AWS; (6) examination of system-level fault tolerance technologies and strategies; (7) investigation into application-level fault tolerance technologies and strategies; (8) predictive modeling for cost management in public cloud environments; and (9) the application of containerization technologies within HPC contexts.

This dissertation delivers significant advancements across the presented areas, except in burstable instances exploration and task scheduling, which we acknowledge as complementary to our research scope. Notably, our work distinguishes itself in the realm of application-level fault tolerance for MPI applications, particularly by focusing on tightly-coupled workloads, as opposed to the bag-of-tasks model commonly explored in existing studies. In the fault tolerance domain, we uniquely incorporate provider-specific services, such as AWS spot eviction alarms, into our fault tolerance strategies.

Table 1 – Related work contributions comparison.

	Public Cloud Studies	IaaS Management	Task Scheduling	Spot Instances	Burstable Instances	System-Level Fault Tolerance	Application-Level Fault-Tolerance	Costs Prediction	Containers for HPC
Aljamal, El-Mousa & Jubair (2018)	✓								
Malla & Christensen (2020)	✓	✓							
Peña-Monferrer, Manson-Sawko & Elisseev (2021)		✓							
Wong & Goscinski (2013)	✓								
Teylo et al. (2023)	✓		✓	✓	✓	✓			
Zhou et al. (2022)	✓		✓	✓				✓	
Dancheva, Alonso & Bartoň (2023)	✓					✓			
Fernandez (2022)	✓								
Sena et al. (2023)	✓		✓	✓		✓		✓	
Marathe et al. (2014)	✓			✓		✓		✓	
Gong, He & Zhou (2015)	✓			✓		✓		✓	
Brum et al. (2023)						✓	✓		
Amoon et al. (2019)	✓					✓			
Vaillancourt et al. (2020)	✓	✓							✓
Sindi & Williams (2019)						✓			✓
THIS WORK	✓	✓		✓		✓	✓	✓	✓

This integration represents a novel contribution to the field, enhancing the resilience of cloud-based applications against failures.

We offer additional insights into the utilization of containerization technologies within the HPC context. Our investigation is concentrated on assessing current methods for containerizing MPI workloads and analyzing the associated overhead. This evaluation serves as a valuable reference point for optimizing containerized HPC environments in public cloud shared infrastructure. Furthermore, in terms of costs prediction, this dissertation lays the foundational steps towards the development of an automated toolkit designed for forecasting the expenses associated with running MPI workloads on public cloud clusters. This secondary contribution sets the stage for future research aimed at enabling more cost-effective cloud computing solutions for HPC applications.

4 THE HPC@CLOUD SOFTWARE TOOLKIT

This chapter introduces our proposed software toolkit, *HPC@Cloud*, designed to utilize public cloud resources for running HPC applications. While the initial version supports only two cloud providers (Amazon AWS and Vultr Cloud), *HPC@Cloud* is readily extensible to accommodate additional providers. Our software contributions are open-source, and accessible at https://github.com/lapesd/hpcac-toolkit.

4.1 HPC@Cloud Software Architecture Overview

To facilitate the migration of HPC applications to public cloud platforms, our proposed toolkit offers a Command-Line Interface (CLI) that can be executed on the user's machine. The CLI enable users to configure cloud infrastructure, execute jobs, monitor performance, predict costs, and interact with cloud platforms in an automated and provider-agnostic manner.

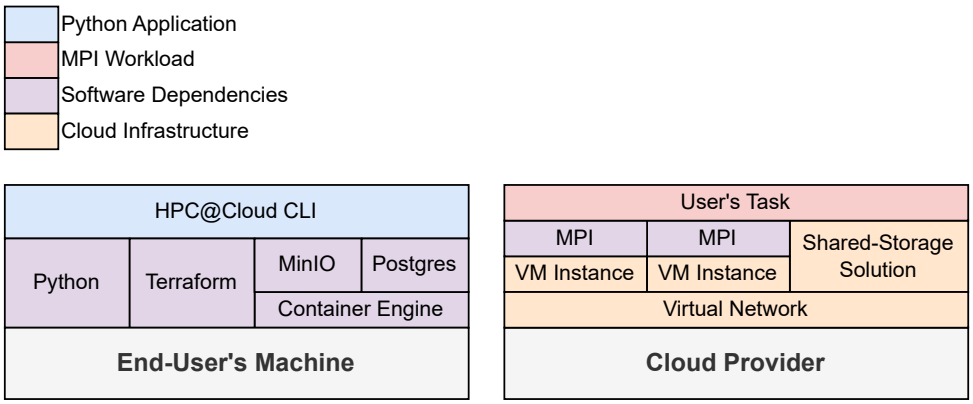
The usual workflow is based on defining a configuration file with the desired cluster specifications, such as: target provider, regions, Availability Zones (AZs), number of nodes, types of instances, storage setup, and more. With the configuration file ready, the user can spawn the desired cluster using a CLI command, and then access it through Secure Shell Protocol (SSH) when the infrastructure is ready.

We also provide provider-specific VM images already configured for MPI and shared storage solutions supported by our software suite, such as Network File System (NFS) and FSx for Lustre (AWS). *HPC@Cloud* is developed using mostly Python and HashiCorp Configuration Language (HCL).

Overall, *HPC@Cloud*'s software architecture is based on four main components, each serving a distinct purpose:

- **Command-Line Interface (CLI)**, a Python package that can be installed with *pip*, from which the end user can submit commands to manage infrastructure and run tasks;
- **Terraform**, a third-party solution for managing the state of cloud resources, facilitating the integration of different cloud providers with the *HPC@Cloud* CLI;
- **Postgres Database**, a relational database system for storing cluster configurations and experiment execution data. It offers flexibility in deployment, allowing for cloud hosting or, alternatively, local containerized setup on the end-user's machine;
- **MinIO**, a high-performance object storage solution utilized by HPC@Cloud for file storage. Similar to the Postgres database, it provides deployment flexibility, accommodating various hosting environments.

Figure 4 – Overview of software dependencies.



Source: Produced by the author.

HPC@Cloud is shipped with configuration files for the whole containerized environment with Postgres and MinIO, allowing it to be executed entirely in the end-user's machine. Figure 4 depicts a high-level overview of the existing dependencies between *HPC@Cloud* and third-party tools, also depicting the main cloud resources involved in running MPI tasks.

In the following sections, we will delve into the operational processes and interactions of *HPC@Cloud* with its previously mentioned dependencies. We outline the steps involved in creating clusters and executing tasks.

4.2 Managing Multicloud Infrastructure

Despite standardization efforts, configuring and managing cloud resources frequently requires provider-specific knowledge about the platform and its APIs, which may be subject to arbitrary changes by the providers. Repeatable infrastructure management is a sought-after feature in both the HPC and enterprise domains. According to Rahman, Mahdavi-Hezaveh & Williams (2019), the current best practice for infrastructure management centers around the concept of Infrastructure as Code (IaC). IaC refers to the practice of configuring computing resources via machine-readable definition files instead of relying on interactive configuration tools.

We have integrated the concept of IaC into the *HPC@Cloud* toolkit for creating, monitoring, and destroying cloud resources. To bridge the gap between Cloud Computing and HPC, our proposed toolkit aims to utilize pre-existing open-source tools whenever feasible. For infrastructure configuration, we chose Terraform[1], a tool that adopts a declarative approach to defining infrastructure. Terraform allows us to create virtual clusters on numerous cloud providers, including AWS and Vultr Cloud — two cloud platforms evaluated in this dissertation.

[1] http://terraform.io

Table 2 – Configurable parameters in *HPC@Cloud*.

Name	Description
`cluster_label`	A label, or name tag for the cluster
`region`	The region code of a specific provider (example: `us-east-1`)
`provider`	The target cloud provider tag (currently supporting `aws` and `vultr`)
`availability_zone`	The AZ code of a specific provider
`private_rsa_key_path`	System path to a private RSA key to be used by the cluster
`public_rsa_key_path`	System path to a public RSA key to be used by clients
`public_key_name`	A label, or name tag for the RSA key pair
`node_count`	The desired number of nodes in the cluster
`instance_username`	The username that needs to be used to login into the selected VM image
`machine_image_id`	The identifier code of a custom VM image to be used
`node_instance_type`	Instance type code (example: `t3.2xlarge`)
`init_commands`	A list of shell commands to be executed in every node after creation
`use_spot`	Flag variable to define if ephemeral infrastructure should be used
`spot_maximum_rate`	Maximum value in USD per hour to pay for spot instances
`spot_maximum_timeout`	Maximum spot request wait time in seconds
`node_rbs_size`	Size of the root block storage for each node, in Gigabytes
`node_rbs_type`	Type of root block storage
`node_rbs_iops`	Root block storage Input and Output Operations per Second (IOPs) for each node
`use_efa`	Flag variable to define if Elastic Fabric Adapter (EFA) should be used
`use_fsx`	Flag variable to define if FSx for Lustre (FSx) should be used
`use_efs`	Flag variable to define if an NFS shared storage system should be created

To interact with multiple public cloud platforms, we use prepared Terraform recipes with reasonable defaults for HPC clusters written in HCL, a configuration language visually resembling JavaScript Object Notation (JSON). In essence, we implemented functions responsible for mapping standardized user-defined parameters into provider-specific Terraform recipes. To support a new cloud provider, these Terraform recipes must be created using the same variable naming conventions as those adopted by *HPC@Cloud*. The configurable cluster variables are shown in Table 2, and are passed to the system in Yet Another Markup Language (YAML) format.

When an user defines a cluster configuration, *HPC@Cloud* generates and saves Terraform files in a MinIO[2] bucket, a high performance object storage solution for modern data lakes. Terraform can then read the files and spawn infrastructure when the user runs the `create_cluster` command. When applying changes or creating infrastructure for the first time, Terraform checks the state of the desired cluster and defines which resources are stale, missing, or needing to be re-created with different configurations.

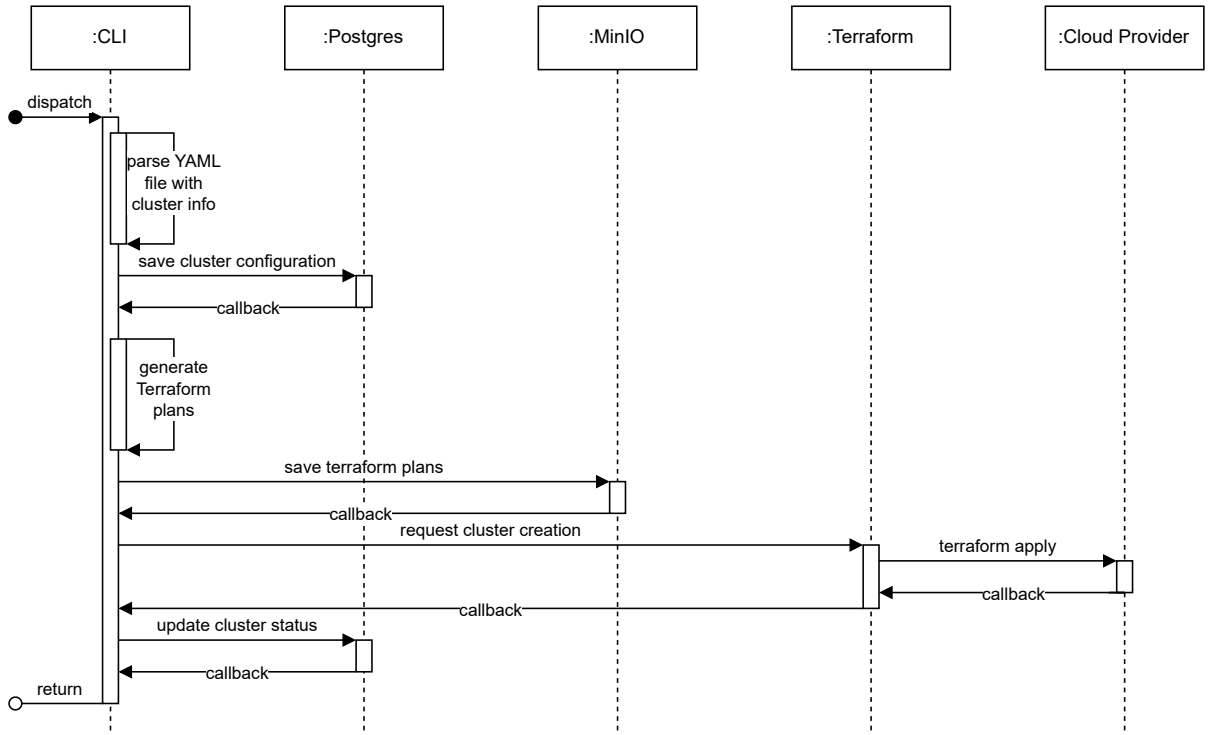
Figure 5 depicts the sequence of actions involved between *HPC@Cloud* and its dependencies when the user dispatches a CLI command to create a cluster.

4.2.1 Cluster Topology, Storage and Networking Devices

The Terraform plans generated by *HPC@Cloud* follow a standard recipe, which has some common elements regardless of provider. In this section, we detail the proposed HPC cluster architecture built on top of AWS and Vultr using *HPC@Cloud*. AWS and

[2] https://min.io/

Figure 5 – Sequence diagram for cluster creation with *HPC@Cloud*.



Source: Produced by the author.

Vultr provide a diverse spectrum of instances, each differing in attributes such as hypervisor type, CPU design, number of vCPUs, memory capacity, and more, enabling users to pinpoint the ideal configuration for their unique requirements.

4.2.1.1 Cluster Topology

We propose a cluster architecture blueprint akin to a classical on-premise homogeneous HPC cluster. Given the single-use character of the clusters – being instantiated solely for a singular task before termination – there is no distinction between master and worker nodes. As such, every node will be dedicated to workload execution. In AWS we can leverage spot instances to build ephemeral clusters, which are not available in Vultr.

The cluster topology is based on a *shared storage topology*, where multiple nodes access a shared storage system, ensuring they can all read and write data to the same set of files concurrently. A centralized storage approach simplifies data management and provides a consistent data view to all nodes. The performance, however, relies heavily on the network, depending on the type of device used to implement the shared storage system. In this dissertation, we discuss three storage options: Elastic File System (EFS), Elastic Block Storage (EBS), and FSx for Lustre. FSx for Lustre and EFS are only available at AWS.

4.2.1.2 Cloud Storage Technologies

EBS is a ubiquitous storage service present in most cloud providers, including AWS and Vultr. At a low level, EBS volumes operate like raw, unformatted block devices, which can be individually attached to the cluster's nodes. Once attached, they are formatted with a file system. EBS volumes can also be configured in terms of Input and Output Operations per Second (IOPS), and we test a variety of setups in our experiments. Our cluster design allocates a distinct EBS volume to each node, maintaining a consistent IOPS configuration across all volumes.

Amazon provides the Elastic File System (EFS) service, a cloud-based file storage service designed to provide scalable, elastic, shared file storage that is accessible to multiple Amazon EC2 instances. Unlike traditional block storage (EBS), which offers block-level storage exclusively to a single EC2 instance, EFS serves as a managed file system that can be simultaneously mounted and accessed by multiple instances, facilitating easier data sharing across applications and services. To establish a shared file system, *HPC@Cloud* can utilize the EFS service from AWS, or build a raw Linux NFS when working with providers that do not offer managed shared file system services, such as Vultr.

Lustre is a shared file system tailored for rapid processing, prevalent in HPC settings. For our cloud-based clusters, we can also utilize Amazon FSx for Lustre, a managed version of the Lustre file system. Unlike EBS volumes, which can only be attached to a single EC2 instance at once, FSx for Lustre supports concurrent attachment to multiple instances, providing a ready-to-use, high-performance shared file system. A notable distinction between using raw Block Storage devices instead of services like EFS or FSx for Lustre, beyond the financial implications, is that the latter options are equipped with its own dedicated computational infrastructure, not requiring the allocation of extra resources for a file system server.

4.2.1.3 Cloud Networking Technologies

Both Vultr and AWS provide robust and scalable network infrastructure for VM instances. A Virtual Private Cloud (VPC) is a logically isolated section of the underlying cloud infrastructure where resources, including VM instances, can be launched. It enables configurations of resources like subnets, route tables, and network gateways, mimicking conventional network elements. We have opted to deploy a VPC in a single AZ for our cloud cluster architecture. This ensures all nodes are housed within the same physical data center, minimizing geographic spread and, consequently, reducing latency.

Moreover, when launching multiple EC2 instances in AWS, they are strategically positioned to ensure a spread distribution across the underlying hardware, mitigating the risk of simultaneous failures. Although ideal for enterprise web applications focused on

Table 3 – Technology features available in each provider integrated into *HPC@Cloud*.

Cloud Provider	Ephemeral Instances	ENA/VNA	Elastic Fabric Adapter (EFA)	Elastic Block Storage (EBS)	Elastic File System	FSx for Lustre
AWS	✓	✓	✓	✓	✓	✓
Vultr	×	✓	×	×	✓	×

fault tolerance through redundancy, this default behavior results in high network latency, damping the performance of tightly-integrated node-to-node communications commonly seen in HPC applications. To improve this, we configured the EC2 service to pack instances closely together with the AWS placement groups[3] feature, thus further improving the network performance. In Vultr there are no configurable setting equivalent to placement groups in AWS.

The basic network interface provided by AWS is called Elastic Network Adapter (ENA), which provides bandwidth capabilities above 100 Gbps for some specific instances. AWS also offers an improved network interface to be attached to EC2 instances named EFA. EFA provides all of the functionality of an ENA but is specifically designed to boost network performance by allowing instances to bypass the typical TCP/IP stack when communicating with each other, resulting in lower network latency. EFA is an optional EC2 networking feature that can be enabled at no additional cost. A downside is that a limited number of operating systems and EC2 instance types are compatible with EFA adapters. In Vultr we have only the standard VPC 2.0 Network Adapter (VNA), that is configured by default on all VMs.

Table 3 highlights the availability of each integrated feature from AWS and Vultr into *HPC@Cloud*.

4.3 Managing Fault-Tolerant Workloads

Workloads are managed using ubiquitous tools in the HPC world, such as `mpiexec` for MPI applications. This approach enables users familiar with these tools to seamlessly deploy their applications in the cloud, thus simplifying the migration process. Once a cluster is created, the user can transfer application files to the cluster's shared file system and execute them using `mpiexec` over an SSH connection. Besides using SSH and accessing the clusters directly, users can also use the `run-tasks` command from *HPC@Cloud* to run multiple MPI applications. *HPC@Cloud* has its own SSH client implemented with the *paramiko* Python package, used to communicate remotely with provisioned cloud infrastructure.

To handle failures, *HPC@Cloud* is embedded with a routine script that periodically checks cluster health. When a node eviction is detected during task execution, a

[3] Available at: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html

new node is automatically requested by re-applying the generated Terraform plans. It is important to note that the provider might lack sufficient spare infrastructure to fulfill new spot requests. To circumvent indefinite waiting for requests to be met, *HPC@Cloud* will opt for an on-demand instance after the `spot_maximum_timeout` threshold is exceeded. If this parameter is set to zero, or if there are no available on-demand infrastructure as well, *HPC@Cloud* will terminate the cluster in the absence of available spare infrastructure.

Some providers offer mechanisms to predict imminent ephemeral instance evictions, saving additional time when requesting new VMs and also when checkpointing applications. Specifically for AWS, by default we enable spot repossession alarms, which provide a two-minute warning before a spot instance is reclaimed. When an ephemeral AWS cluster is created with *HPC@Cloud*, the routine script also monitors the AWS eviction notices to checkpoint and create new spot bids before a failure occur.

When running tasks with the *HPC@Cloud* `run-tasks` command, the execution loop will keep running in the background and restarting tasks when a failure or spot eviction occur. If no checkpoints are available, execution is restarted from the beginning. *HPC@Cloud* supports **system-level** fault tolerance by allowing the restart of failed tasks from checkpoint files created by previous executions, or using specific shell commands provided by the user in the tasks configuration YAML file.

Workloads that employ fault tolerance in the **application-level** are not tracked by *HPC@Cloud*. In these cases, the user is responsible for coding a custom fault tolerance mechanism that can or cannot be based on checkpoint-restart, thus *HPC@Cloud* assumes the fault tolerance is being handled correctly by the application.

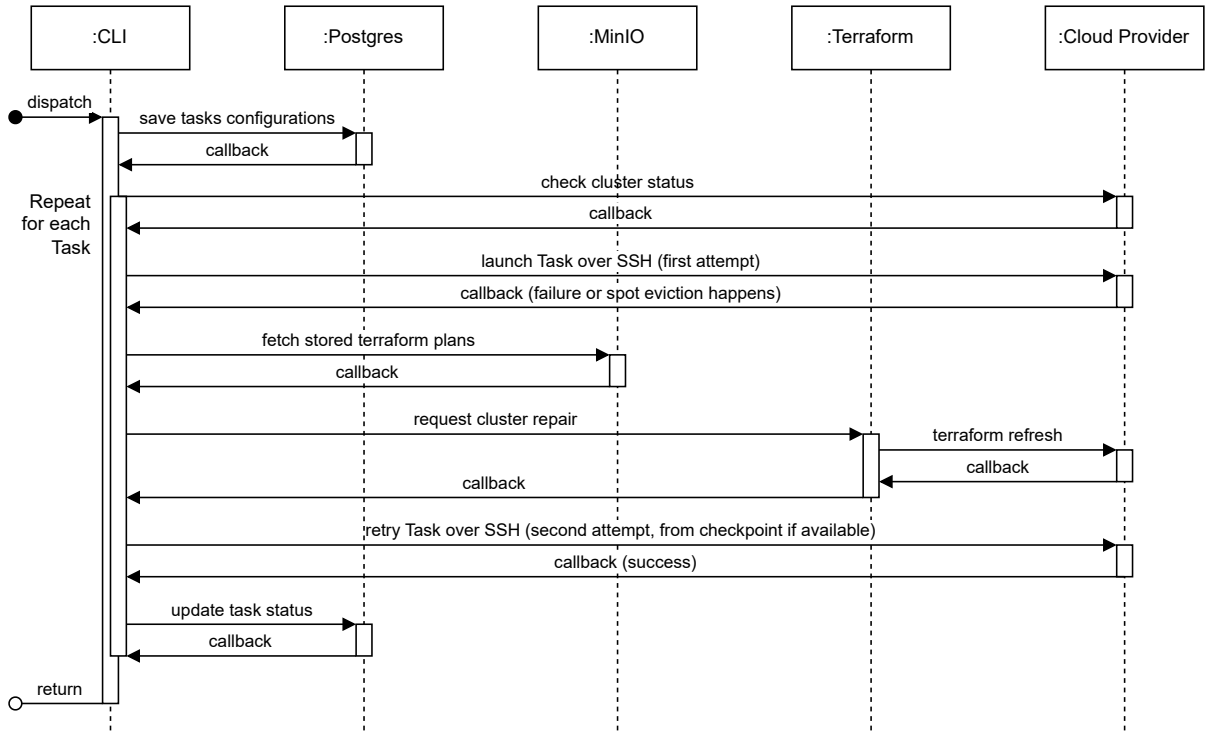
The configurable task variables are shown in Table 4, and are parsed in YAML format by *HPC@Cloud*.

Figure 6 depicts the sequence of actions involved between *HPC@Cloud* and its dependencies when the user dispatches a CLI command to run tasks in a provisioned cluster, highlighting the case where a failure occurs and the repair process.

Table 4 – Task configurable parameters in *HPC@Cloud*.

Name	Description
`delete_cluster_after_execution`	Self-explanatory flag variable
`overwrite_tasks`	Define if tasks with the same label should be overwritten in Postgres
`task_tag`	Label to help identify tasks
`setup_command`	Shell command to setup task execution, usually compiling
`run_command`	Shell command to start the task execution
`ckpt_command`	Command to create system-level checkpoint files
`restart_command`	Command to be executed when retrying executions after failures
`fault_tolerance_technology_label`	Label to help identify tasks
`checkpoint_strategy_label`	Label to help identify tasks
`retries_before_aborting`	Maximum number of retries before aborting execution
`remote_outputs_dir`	A remote folder to be saved locally after execution

Figure 6 – Sequence diagram for tasks execution with *HPC@Cloud*.



Source: Produced by the author.

4.3.1 Failure System Model

We assume a distributed system composed of interconnected processes running on top of a cloud infrastructure. The system is asynchronous, *i.e.*, there is no bound on message delays and on relative process speeds. The underlying infrastructure comprises instance nodes that can be allocated or deallocated on the fly. Computational nodes may differ in computing, memory, and network capacities.

The unbounded set of processes $P = \{p_1, p_2, \dots\}$ is automatically mapped to the unbounded set of nodes $N = \{n_1, n_2, \dots\}$, where each process $p_i \in P$ is mapped to a node n_j and all nodes $n_j \in N$ host at least one process. Computational nodes are classified as *spot* or *on-demand* instances. We assume there is no resource's scarcity for *on-demand* instances. Thus, although spot instances might be unavailable, *on-demand* instances are available whenever they are requested.

Spot instances may fail, or they can be revoked by the cloud provider, while the *on-demand* instances never fail or are revoked. Any process executing on a faulty or revoked node eventually fails. We assume the *fail-stop* failure model and exclude malicious and arbitrary behavior (e.g., no Byzantine failures). This means that a faulty processes running on spot instances are detected by correct processes in the system. We assume the existence of a notification system that alerts system processes before a spot instance revocation occurs. Typically, these alerts are notified with short notice, but processes can

miss the alert. Processes are equipped with a volatile memory, a local storage, and they share a persistent storage. Upon a failure, a process loses its volatile memory and local storage content, but the information written in stable storage survives the failure.

4.3.2 System-Level Fault Tolerance

We integrated into *HPC@Cloud* two system-level checkpointing technologies: BLCR and DMTCP. The BLCR package (Hargrove; Duell, 2006) approach was previously proposed and extensively tested in (Munhoz; Castro; Mendizabal, 2022). We also proposed two checkpointing strategies (`periodic` and `preemptive`) that can be employed with both of these technologies.

The following subsections briefly present these technologies and discuss their pros and cons.

4.3.2.1 Berkeley Labs Checkpoint-Restart (BLCR)

Utilizing the BLCR method for fault tolerance necessitates no modifications to the MPI application code. The BLCR command-line tool is invoked by *HPC@Cloud* to create and restore a checkpoint, using the Process Identification Number (PID) of the launched MPI application. To leverage spot repossession alerts from AWS, *HPC@Cloud* can proactively orchestrate checkpoints, initiating new spot requests and monitoring spot eviction notifications from the cloud provider. While periodical checkpoints are initiated at predefined time intervals, preemptive ones are activated exclusively upon AWS notifications. It is worth noting that some applications might require a longer duration than a brief alarm to finalize a checkpoint, potentially making the preemptive approach ineffective.

In summary, the BLCR approach has the following limitations:

1. **Outdated Solution**: BLCR requires specific versions of old Linux kernel headers to work, which are difficult to install on cloud instances and have severe unpatched security issues—the latest version of BLCR (0.8.6) is tested with CentOS Linux release `7.3.1611`, kernel version `3.10.0-514.el7`.
2. **Dynamically Linked Applications Unsupported**: Executables must be compiled as a static binary for BLCR to be able to checkpoint them.
3. **Support for TCP/IP Only**: BLCR only supports MPI over TCP/IP connections; thus, it cannot use InfiniBand and other types of high-speed networks, such as the improved EFA from AWS for example.
4. **MPI Libraries Unsupported**: MVAPICH2 is the only MPI distribution with embedded support for BLCR, limiting options for the users.
5. **Inefficient Operations**: BLCR checkpoint save and restoration actions are completely blocking, wasting idle cloud resources.

Despite these issues, BLCR is simple enough that self-contained MPI applications can be made fault-tolerant without requiring any source code changes.

4.3.2.2 Distributed MultiThreaded CheckPointing (DMTCP)

In comparison to BLCR, DMTCP[4] is a much newer solution, not having the major compatibility downsides that BLCR has. DMTCP is partially funded by Intel Corporation and the National Science Foundation (NSF).

In contrast to BLCR, DMTCP has the following advantages:

1. **Modern and Updated**: DMTCP is a newer solution compared to BLCR, supporting a wider range of system configurations without being bound to older Linux kernel headers.
2. **Flexibility with Networking**: While BLCR is limited to TCP/IP networking, DMTCP supports a broader array of network protocols, including InfiniBand.
3. **Dynamically Linked Applications Supported**: DMTCP does not require applications to be static binaries.
4. **MPI Libraries Supported**: DMTCP works with all major MPI distributions, such as OpenMPI, IntelMPI, MVAPICH2 and MPICH.

Although DMTCP is a much more versatile solution for checkpointing than BLCR, it still has the same disadvantages that file-based system-level solutions have. For example, DMTCP requires a potentially large amount of storage space for checkpoint files, as well as requiring a considerable amount of time for I/O operations.

Despite the issues, DMTCP is the most versatile system-level checkpointing solution currently available, also requiring no changes to existing application code.

4.3.3 Application-Level Fault Tolerance

In this dissertation, we have also tested two major frameworks for application-level fault tolerance: SCR and ULFM. They are briefly described in the next subsections.

4.3.3.1 Scalable Checkpoint Restart (SCR)

SCR is a high-level library for caching checkpoint data in local storage on compute nodes, or remote and parallel file systems. It provides a fast and scalable checkpoint restart capability for MPI applications (Moody et al., 2010).

In contrast to system-level solutions such as BLCR and DMTCP, SCR requires code changes in the target application. Integrating SCR into an application at the code

[4] Accessible at: https://dmtcp.sourceforge.io/

level typically involves linking with the SCR library and inserting API calls at strategic points within the codebase. Initially, the application's configuration file must specify parameters like checkpoint intervals and storage hierarchy details. In the application, the developer initiates the checkpointing process with a call to the SCR's `Start_checkpoint` API, followed by the actual data saving routines, which might involve serializing data structures and writing them to designated local storage. Once the checkpoint data is prepared, a call to `Complete_checkpoint` finalizes the process. If a failure is detected upon application restart, SCR provides mechanisms to identify the most recent valid checkpoint with functions like `Have_restart`, facilitating the restoration process. To aid in this, SCR handles the retrieval of checkpoint data from either local storage or more remote, reliable storage mediums, depending on the configuration and circumstances of the failure. Throughout this process, it is imperative to ensure thread safety and manage potential I/O bottlenecks, which SCR optimizes through asynchronous data transfers and by minimizing data movement.

We implemented a fault-tolerant version of *HEAT* (Section 2.5.2) using SCR, adding C/R capabilities to the main Jacobi loop. We use SCR API calls to write and read checkpoints from a storage system that can be local EBS storage or a shared FSx for Lustre system – depending on the cluster setup. With the `periodical` C/R strategy set, we save checkpoint files based on the checkpoint frequency configured. If the `preemptive` C/R strategy is set, we save a checkpoint file after polling an eviction notice from AWS. In case of an eviction alarm, we promptly exit the application after the checkpoint is saved, and *HPC@Cloud* is then responsible for restoring the cluster and restarting *HEAT*.

Checkpoint files for *HEAT* contain array data with the mesh cell values and some flow control variables. The pseudo-code with our implementation is presented in Algorithm 2.

4.3.3.2 User-Level Failure Mitigation (ULFM)

ULFM offers a solution to manage failures by providing a set of extensions to the MPI standard, enabling application developers to handle errors. These extensions allow the detection, reporting, and recovery from process failures within the MPI applications (Bland et al., 2013).

Unlike other fault tolerance mechanisms that focus on checkpointing or replication techniques, ULFM allows applications to adapt their behavior and continue execution in the presence of process failures, making adaptive restoration possible. However, this requires developers to design and implement their own solutions, tailoring them to their applications' specific requirements and characteristics. ULFM in MPI is a framework that equips developers with the tools necessary to build resiliency into parallel applications in an adaptive manner.

In summary, the ULFM approach has the following main limitations:

Algorithm 2 – *HEAT* application iterative logic, with C/R added with SCR.

```
1: MPI initial setup
2: memory allocation and initialization
3: if checkpoint is available then
4:     get data from last checkpoint
5:     update mesh cells with checkpoint data
6: repeat
7:     if periodic checkpoint then
8:         save state checkpoint
9:     if spot alarm notifies an eviction then
10:        save state checkpoint
11:        exit application                ▷ HEAT will be restarted by HPC@Cloud
12:    update ghost cell pattern
13:    compute local mesh cell updates
14:    all-reduce residual error with all ranks
15:    if criterion (residual or iterations) reached then
16:        converged = true
17: until converged
18: write solution into disk
```

1. **More Complexity**: In contrast to BLCR, DMTCP and SCR, ULFM does not provide a ready-to-use solution; instead, users must design and implement a fault tolerance strategy of their own, requiring significant source-code modifications to the MPI application, making migration time-consuming and costly;
2. **Some MPI Libraries Unsupported**: Not all MPI library distributions currently offer comprehensive support for ULFM, though the most popular ones either provide limited support or are working towards implementing it. Among the available options, the development branch of OpenMPI currently delivers the most robust ULFM support;
3. **Limited to C and Fortran**: At present, ULFM APIs for MPI are exclusively available for C and Fortran programs.

Despite its limitations, employing ULFM significantly enhances checkpointing efficiency compared to system-level methods, such as BLCR and DMTCP, and offers inventive methods for maintaining application continuity, different from the basic checkpointing provided by SCR APIs.

We implemented routines to add C/R capabilities into the parallel *HEAT* application (Section 2.5.2), First, routines to allocate and update an array with global mesh checkpoint data were added. Secondly, we set a custom error handler routine for the application. By default, MPI errors are handled by `MPI_ERRORS_ARE_FATAL`, which terminates the `MPI_COMM_WORLD` and stops the entire application. With the ULFM API, we are able to set a new `MPI_ERRORS_RETURN` handler, which will return an error code in case of failure (Bland et al., 2012).

Algorithm 3 – *HEAT* application iterative logic, with C/R added with ULFM.

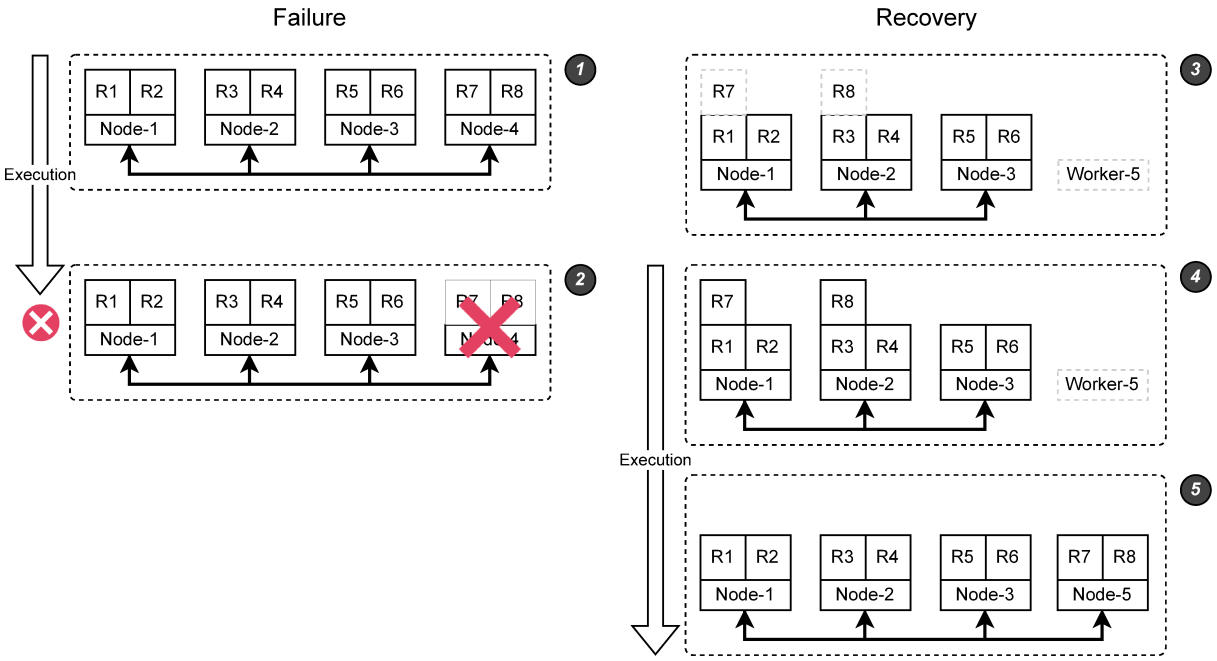
```
1: function ERROR_HANDLER(MPI_Comm, ...)
2:     replace failed communicators
3:     re-order communicators to original rank ordering
4:     longjmp() to recovery
5:
6: MPI initial setup
7: set error_handler(MPI_comm, ...) as the error handler
8: memory allocation and initialization
9: setjmp() for recovery
10: if recovering then
11:     get data from last checkpoint
12:     update arrays with checkpoint data
13:     longjmp() to computation
14: repeat
15:     if periodic checkpoint then
16:         save state checkpoint
17:     if spot alarm notifies an eviction then
18:         save state checkpoint
19:         request ephemeral cluster to be repaired          ▷ API call to Terraform
20:     setjmp() for computation
21:     compute local mesh cell updates
22:     all-reduce residual error with all ranks
23:     if criterion (residual or iterations) reached then
24:         converged = true
25: until converged
26: write solution into disk
```

After a spot eviction, all MPI processes currently hosted at that instance are unusable. By setting a custom `MPI_ERRORS_RETURN` handler, we define a set of routine calls to restore the MPI world and resume execution from the last in-memory checkpoint. To remove failed processes and stop current operations after a failure, we call the `MPIX_Comm_shrink` routine to create a new communicator by excluding all known failed processes from the parent communicator. For fully recovering our application, we need to restore our MPI world to the same previous size while also maintaining the same exact rank mapping. To re-spawn missing processes preserving rank mapping we use the `MPI_Comm_spawn` routine. Algorithm 3 shows the application logic to fully recover execution as pseudo-code.

All other presented fault tolerance methods relied on blocking restoration (i.e., the application is suspended temporarily when the cloud provider revokes any spot instance). With our ULFM approach, the *HEAT* application continue executing without waiting for new spot instances to be created.

Figure 7 describes how this mechanism works. Let us consider that an MPI ap-

Figure 7 – MPI oversubscription approach for ULFM-based fault tolerance.



Source: Produced by the author.

plication is running with eight MPI ranks (*R1–R8*) on four spot nodes (*Node-1–Node-4*), and there are two MPI ranks per node (①). Assuming that the cloud provider revokes *Node-4* (②), the *HPC@Cloud*'s *Infrastructure Lifecycle Manager* will immediately request a replacement node (*Node-5*), and all failed MPI ranks will be oversubscribed to healthy nodes (③), resuming the work from the latest checkpoint (④). This allows the application to keep running but with a damped performance. When eventually *Node-5* is restored, the application is again rebalanced to its original number of MPI ranks per node by killing the oversubscribed MPI processes and respawning them on the new instance (⑤). This strategy eliminates idle time waiting for spot requests to be handled by the provider.

Table 5 compares all fault tolerance technologies discussed and tested in this research, highlighting their main features and advantages.

Table 5 – Features and advantages of each fault tolerance technology.

	BLCR	**DMTCP**	**SCR**	**ULFM**
System-Level Integration	✓	✓	×	×
Application-Level Integration	×	×	✓	✓
Out-of-the-box Checkpoint Restart Solution	✓	✓	✓	×
Flexibility to Implement Different Fault Tolerance Strategies	×	×	×	✓
Compatible with libfabric and InfiniBand support	×	✓	✓	✓
Latest Linux Kernels Support	×	✓	✓	✓
C and Fortran Support	✓	✓	✓	✓
C++ and Python Support	✓	✓	✓	×
Dynamically Linked Applications Support	×	✓	✓	✓

4.4 Images, Communication, and Containerization Support

Users can select basic operating system images when spawning VMs on public cloud platforms, which are usually the Long-Term Support (LTS) releases of popular Linux distributions. Most cloud providers also allow customers to save snapshots of modified machine images. As using a snapshot to create a new machine is much faster than installing everything from scratch every time, *HPC@Cloud* leverages this feature to reduce cluster setup time. Users can use pre-defined machine images with the latest packages and libraries needed by *HPC@Cloud* or provide his own machine image.

To execute installations that depend on the configuration of other machines, such as setting up a NFS server for shared storage and connecting clients to it, users of *HPC@Cloud* can define shell scripts to be executed immediately after a VM is instantiated. To minimize makespan, dependencies must be installed beforehand and copied as static binaries or a custom machine image must be utilized.

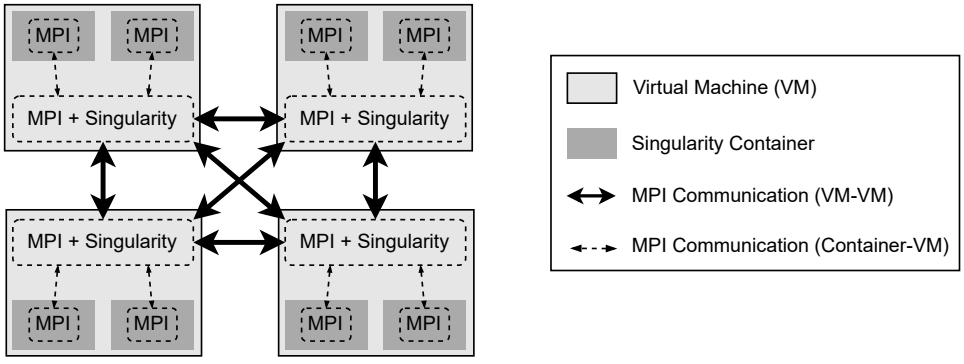
HPC@Cloud provides a prepared VM image with the latest Singularity[5] release installed from source, incorporating a compatible distribution of the MPI library (MPICH, OpenMPI, or MVAPICH2). To allow applications inside Singularity containers to communicate with each other, we have implemented a hybrid approach wherein an MPI distribution is installed inside the containers as well, allowing it to communicate seamlessly with the MPI process from the host VM.

The hybrid model for MPI communications between Singularity containers operates by establishing a bridge between containers utilizing the MPI distribution present on the VM hosts. This implies that for one container to communicate with another, it must first engage with the host MPI, followed by host-to-host communication. Subsequently, the host containing the target container forwards the information to the container MPI process. This communication hierarchy is illustrated in Figure 8.

At this point, processes within the container function as they would typically do

[5] https://sylabs.io

Figure 8 – MPI hybrid model: message passing communications between VM hosts and containers.



Source: Produced by the author.

directly on the host VM. The benefits of this workflow include seamless integration with resource managers and simplicity, as the process closely mirrors natively running MPI applications. However, the drawbacks of this workflow involve the need for compatibility between the MPI version within the container and the host's MPI version, as well as the necessity for careful configuration of the container's MPI implementation to ensure optimal hardware usage, especially when performance is crucial. Hursey (2020) presents other approaches for running MPI applications using containers.

4.5 Forecasting Costs

Predicting Cloud Computing costs has become a significant concern for companies and organizations. Although the initial information technology investments can be avoided, over time, Cloud Computing expenses may exceed those of operating a physical data center, depending on the scale and types of services utilized (Li et al., 2021). As Netto et al. (2018) highlighted, accurate cost predictions remain a significant challenge in developing a fully realized cloud HPC platform.

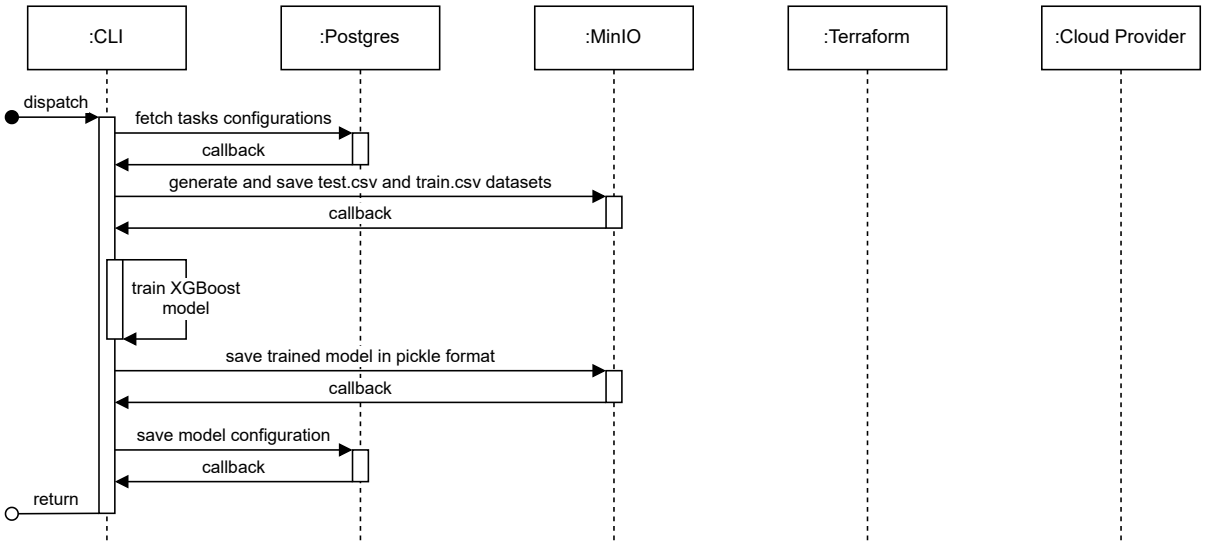
For multi-VM MPI applications, the total cluster makespan constitutes the most significant cost, as storage and network expenses are negligible compared to compute time prices per hour. Makespan costs are billed by various increments, such as months, hours, minutes, or even seconds, depending on the cloud provider, with most providers offering discounts for long-term purchases. In the case of applications running on AWS, EC2, and storage services have been billed in seconds since 2017 (Amazon Web Services, 2017). Elastic GPUs are also billed by seconds, albeit with a 1-minute minimum.

Estimating costs thus becomes a matter of determining the workload's runtime, which is a complex problem due to the numerous variables involved. For spot instances, despite their variable pricing, the smoother price fluctuations since 2017 have made time series prediction a viable option for accurately predicting the spot price of specific instances (Chhetri et al., 2017). However, public Cloud Computing resources are inherently shared, resulting in unreliable performance. Identical VMs may behave differently, as the underlying hardware state is unknown.

HPC@Cloud allow users to benefit from a quick costs estimation before committing resources to execute tasks. We embedded a simple machine learning workflow for training and scoring task duration based on a XGBoost model, which is trained using historical task execution data stored by *HPC@Cloud* in the Postgres database. The XGBoost model is implemented with the `xgboost` Python library. To initiate model training, the user must execute a specific command via the CLI. Subsequently, when a new task is launched, the XGBoost model utilizes preliminary task information, including infrastructure and workload specifics, to predict the total execution time. This prediction is then used to calculate the estimated infrastructure costs during the task execution.

Figure 9 depicts the machine learning training workflow in *HPC@Cloud*. The

Figure 9 – Machine Learning training workflow in *HPC@Cloud*.



Source: Produced by the author.

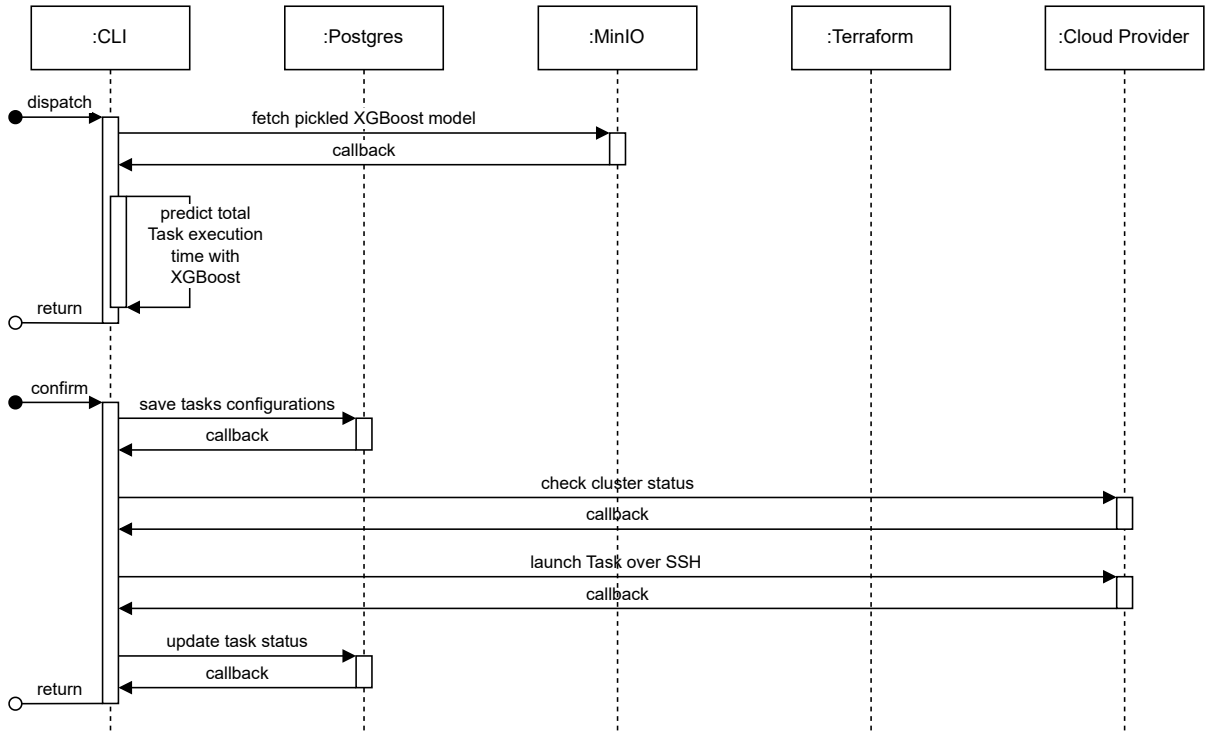
model training is triggered by a CLI command, and the resulting model artifact is pickled and stored in MinIO for later use.

Figure 10 showcases the machine learning scoring workflow in *HPC@Cloud*. Before finalizing the task launch, users will be provided with a cost forecast generated by the XGBoost model. This allows them to make an informed decision regarding the allocation of resources for the task.

We have chosen the following variables for training our XGBoost model, due to their significance in influencing the execution time and overall task performance:

1. **Total execution time in seconds**: This is our target variable, an integer indicating how long the experiment will take in seconds.
2. **Application class**: This is a variable representing the scale or complexity of the application. This concept is akin to the classification of application "classes" in many benchmarks, including the NPB.
3. **Number of nodes**: A numerical variable that captures the scale of parallelism. More nodes often mean better distributed computing capabilities, but can also introduce complexities related to communication overhead.
4. **vCPUs per node**: A numerical variable that translates to more computational power in each cluster node. Applications that are CPU-bound will be directly influenced by the number of vCPUs available.
5. **RAM memory per node**: Given as a numerical value, it denotes the memory capacity available to each node. For memory-intensive applications, having adequate RAM is crucial to avoid bottlenecks related to swapping and paging.
6. **Measured node-to-node network bandwidth**: A numerical representation of the communication speed between nodes. This is essential for applications that

Figure 10 – Machine Learning scoring workflow in *HPC@Cloud*.



Source: Produced by the author.

require frequent data exchanges between nodes, as a higher bandwidth can reduce inter-node communication delays.

7. **Measured IOPS when writing to shared-storage**: This numerical metric is critical for applications with high disk I/O operations, indicating the speed at which data can be written to or read from shared storage systems.

Being a tree-based model, XGBoost is inherently capable of handling a broad range of data distributions. Numerical variables like **vCPUs per node**, and **Measured node-to-node network bandwidth** require no specific transformation before training. The variables are stored in the *HPC@Cloud* Postgres database and can be directly accessed via the CLI command during the training phase of the machine learning model. Once the training is complete, the XGBoost model is serialized into a binary format (pickled) and stored as an object in MinIO.

When running tasks with the *HPC@Cloud* CLI, the user can enable the costs forecasting if a previously trained XGBoost model is available. The task and cluster configurations are then passed as input for scoring in the machine learning model to get an estimated time for completion. With this estimated time, we calculate the total costs by multiplying this duration with the per-second cluster cost. To determine the minimum anticipated expenses, we factor in spot discounts, assuming no failures. As of now, *HPC@Cloud* lacks a feature to predict spot evictions, making it a promising field for future research.

5 EXPERIMENTAL RESULTS

This chapter presents our evaluation methodology to assess *HPC@Cloud* and our proposed fault tolerance strategies, as well as the obtained results when testing our toolkit with real public cloud infrastructure. We split our experiments into five branches:

- **Resource Management Efficiency (Section 5.1):** We assess the resource management efficiency of *HPC@Cloud* by measuring the time required to launch VM instances across virtual clusters of varying node counts.
- **Cluster Scalability (Section 5.2):** This section delves into the scalability of clusters across varying sizes running the *HEAT* application. We mainly assess the pros and cons of using many small instances versus few fat instances for cluster creation.
- **Fault Tolerance Strategies (Section 5.3):** We analyze the cost-effectiveness and performance of a wide range of fault tolerance approaches applied to the *HEAT* application running on ephemeral clusters in AWS, comparing them with on-demand options from both AWS and Vultr.
- ***DynEMol* Migration Analysis (Section 5.4):** This sectio brings an in-depth study case of a real-world HPC application (*DynEMol*) migration to the public cloud.
- **Containerized Execution (Section 5.5):** We analyze the performance overhead of the containerization layer added by Singularity, using the *HEAT* application and NPB as test workloads.
- **Costs Forecasting (Section 5.6):** We analyze the accuracy of *HPC@Cloud*'s costs forecasting method, also using the *HEAT* application and NPB as test workloads.

5.1 Resource Management Efficiency

Our goal is to determine the cost overhead solely related to the infrastructure setup, which incurs charges from the provider even when no tasks are executed. To achieve this, we initiated clusters featuring different node counts and instance types. We deployed variations of homogeneous clusters with 1, 2, 4, and 8 nodes, utilizing a range of instance types in AWS. We specifically measured the duration of the following steps involved in the cluster setup process:

1. **Instance spawn time**, comprising the period from the instance request to when it becomes accessible via the SSH protocol.
2. **Cluster initialization**, comprising the aggregate time consumed by executing all user-defined initialization commands specified in the `cluster_config.yaml` file.

3. **Shared file system configuration**, comprising the total time dedicated to configuring the shared file system.

We define the total setup time as the cumulative duration encompassing the aforementioned processes. During the cluster initialization process, we execute commands to update the machine's operating system, install, and compile OpenMPI along with its dependencies from source. This also includes altering configuration files to enable passwordless SSH connections between nodes and access to a shared file system. Dependencies are installed and updated with `yum`, the standard Amazon Linux 2023 package manager. AWS VMs are created in the `us-east-1a` AZ, using the base Amazon Machine Image (AMI) for Amazon Linux 2023, with code `ami-0230bd60aa48260c6`. Listing 1 presents the shell script used to setup instances.

We select exclusively CPU-only instances with shared-infrastructure, representing the majority of available instances in public cloud platforms, which can be readily used without long-term contracts. Details of the instances used in the infrastructure management experiments are shown in Table 6, and Figure 11 presents the obtained results. Each experiment was repeated 3 times.

Based on the results, we notice that using pre-built machine images is essential to reduce the time needed to launch clusters and reduce costs, as highlighted in Figure 11 by the much larger time spent running initialization commands.

Instances within the *Compute Optimized* category, including `c5.2xlarge` and

Listing 1 – Shell script for instance setup.

```
#!/bin/bash
sudo yum -y update
sudo yum groupinstall -y 'Development Tools'
sudo yum install -y git wget nfs-utils
sudo yum clean all sudo rm -rf /var/cache/yum
sudo mkdir /var/nfs_dir
sudo chown -R nobody:nobody /var/nfs_dir
sudo chown -R ec2-user:nobody /var/nfs_dir
sudo chmod -R 775 /var/nfs_dir
sudo rm /etc/ssh/ssh_config
sudo touch /etc/ssh/ssh_config
sudo sh -c "echo 'Host *' >> /etc/ssh/ssh_config"
sudo sh -c "echo 'StrictHostKeyChecking no' >> /etc/ssh/ssh_config"
sudo sh -c "echo 'UserKnownHostsFile=/dev/null' >> /etc/ssh/ssh_config"
sudo sh -c "echo '10.0.0.10' >> /root/.ssh/authorized_keys"
git clone --recursive https://github.com/open-mpi/ompi.git && cd ompi
sudo ./autogen.pl
sudo ./configure --disable-io-romio CFLAGS='-O0 -g'
sudo make all install
sudo git clean -fdx
sudo rm -rf 3rd-party
```

Source: Produced by the author.

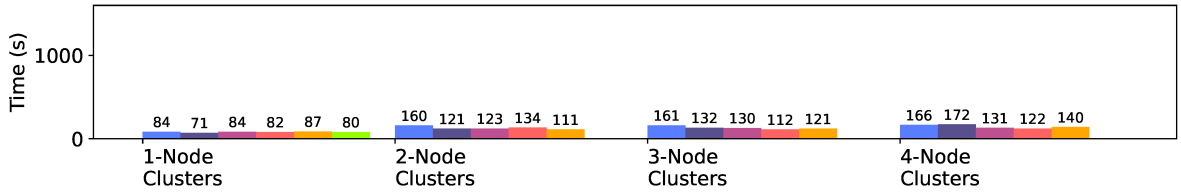
Table 6 – Instance types evaluated in resource management efficiency tests.

Platform	Instance Name	Instance Category	Infrastructure Type	Cores*	On-Demand Price (hourly)
AWS	`t2.micro`	General Purpose	VM – On-Demand	1	0.0058
	`t3.2xlarge`	General Purpose	VM – On-Demand	8	0.3328
	`c5.2xlarge`	Compute Optimized	VM – On-Demand	8	0.3400
	`c6i.2xlarge`	Compute Optimized	VM – On-Demand	8	0.3400
	`i3.2xlarge`	Storage Optimized	VM – On-Demand	8	0.6240
	`r6idn.16xlarge`	HPC	VM – On-Demand	64	6.2525

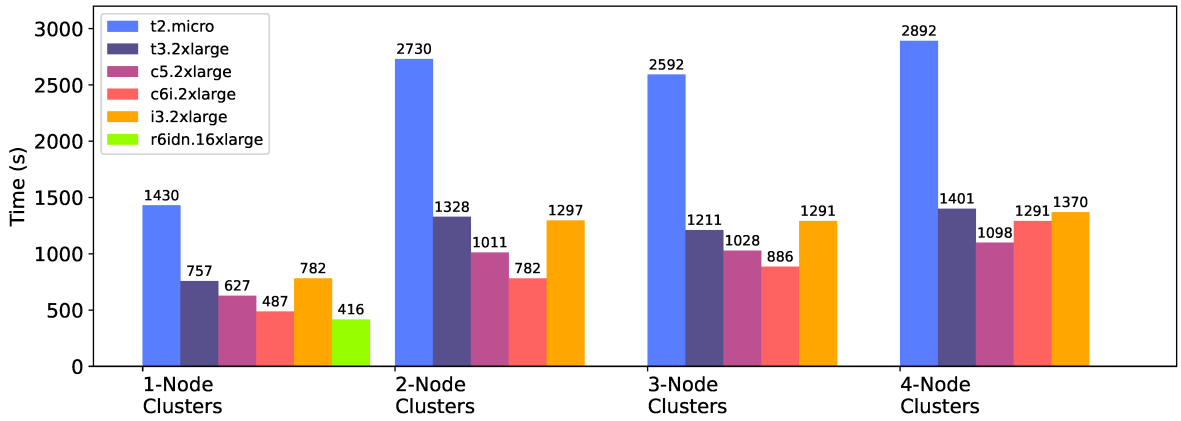
*Virtual (logical) cores with HyperThreading enabled.

Figure 11 – Resource management efficiency results.

(a) Cluster spawn times.



(b) Initialization and shared file system setup times.



Source: Produced by the author.

`c6i.2xlarge`, demonstrated faster setup times. This improvement is primarily attributed to the accelerated compile times, a direct consequence of the enhanced processing capabilities these types of instances offer. Instances within the *High-Performance Computing* category, such as `r6idn.16xlarge`, obtained the fastest setup times, however their spawn time is basically the same as the other instances.

Smaller instances of the *General Purpose* category, such as `t2.micro`, can take up to 40 minutes just to install OpenMPI from sources, versus a 2-minute setup time when using a prepared AMI. Results paint a clear trend that more powerful machines get set up quicker than cheaper and more common instances, specially when installing dependencies on the fly. Nevertheless, this difference is negligible when using AMIs or VM snapshots.

Our tests reveal that cluster setup times remain remarkably consistent, regardless of whether 2 or 8 nodes are created, indicating good scalability. It is evident that the attained high scalability for resource creation and setup is attributed to the parallel creation of cluster nodes, being limited only by the provider's capacity. Single-node clusters can be created much faster, given that there is no need for setting up a shared storage system.

Furthermore, the benefit of manually creating AMIs may not justify the effort and time involved, particularly for large workload sizes which will execute for much longer than the cluster spawn and setup times. AMIs also incur storage costs to remain available in the cloud, and there is a cap on how many images one can generate. Amazon restricts the creation of public images to a default maximum of five. Additionally, at the time of our experiments, Vultr Cloud did not support machine image snapshots, indicating that the feasibility of utilizing this feature is specific to the utilized cloud provider.

5.2 Cluster Scalability

In this section, we conduct a thorough assessment of the performance and scalability of cloud computing clusters with HPC applications of differing sizes. Our analysis focuses on how effectively these clusters can scale by integrating additional nodes. For the practical experiments we select a range of cluster configurations using AWS and Vultr. We focus our experiments on the readily available CPU-only machine types, including the `r6idn.16xlarge` instance, which is marketed by Amazon for high-performace tasks. We also test one of the cheapest baremetal infrastructure options in the market, the Vultr `vbm-6c-32gb` machine with Intel E-2286G processors. AWS machines are created in the `us-east-1a` AZ, located in Washington DC (USA), and Vultr machines are created in the `dfw` AZ located in Dallas (USA). We prepared a series of cluster configurations with sizes varying between 1, 2, 4, and 8 nodes, adopting a one MPI rank per vCPU assignment mapping for all executions. Our clusters use the standard EBS for shared storage, and also the standard network adapters for node-to-node communication. Detailed information on the instance types employed in these experiments is provided in Table 7.

Table 7 – Instance types evaluated with *HEAT*.

Platform	Instance Name	Instance Category	Cores*	On-Demand Price (hourly)	Spot Price (hourly)	Discount
AWS	`t3.2xlarge`	General Purpose	8	0.3328	0.1277	61.6%
	`c5.2xlarge`	Compute Optimized	8	0.3400	0.1147	66.1%
	`i3.2xlarge`	Storage Optimized	8	0.6240	0.1913	69.3%
	`r6idn.16xlarge`	HPC	64	6.2525	0.7024	88.8%
Vultr	`vhp-8c-16gb`	Compute Optimized	8	0.1430	×	×
	`vbm-6c-32gb`	Baremetal	6	0.2750	×	×

*Virtual (logical) cores with HyperThreading enabled.

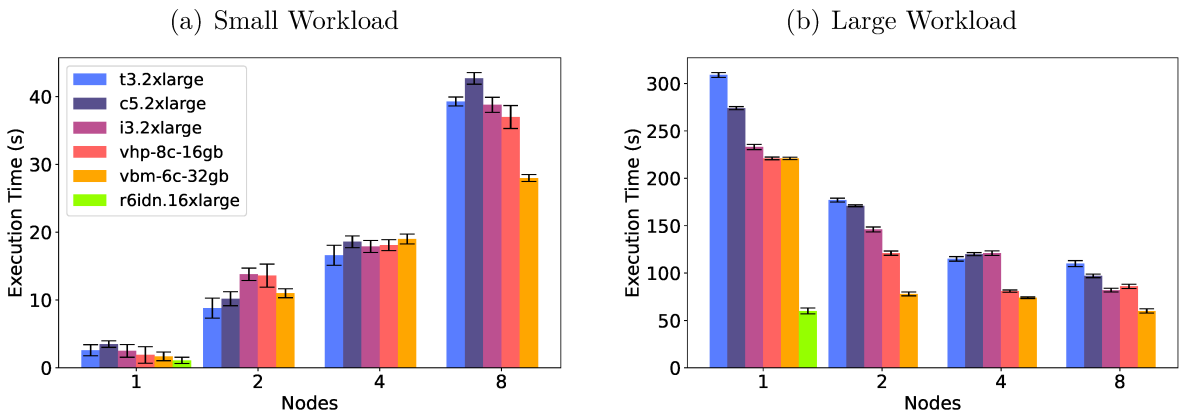
To evaluate our clusters, we run the *HEAT* application using two distinct workload sizes, conducting 200 iterations across two different matrix dimensions: the `small` configuration, featuring a 128x128 mesh grid, and the `large` configuration, which utilizes a more extensive 2048x2048 mesh grid. For now, no failures are induced when executing workloads, as our failure system model assumes no errors are possible with on-demand infrastructure. Base results for *HEAT* without fault tolerance are shown in Figure 12.

The findings from Subfigure 12(a) emphasize that *HEAT* (small) struggles with efficient horizontal scaling. This limitation is primarily attributed to the increased communication latency which becomes counterproductive when CPU processing time is minimal. In such scenarios, it is evidently more advantageous to pursue vertical scaling and circumvent potential network bottlenecks. Subfigure 12(b) illustrates that large workloads are better poised to capitalize on parallelism. However, at some point a limit is reached and the addition of nodes merely escalates costs without providing a meaningful boost in performance or speedup, as it can be noted when doubling the amount of nodes (from 4 to 8).

When considering costs, even though the use of Vultr's `vbm-6c-32gb` baremetal machines results in a much faster execution, because they are billed in hourly increments they are overkill and not worth for *HEAT*, which executes in mere minutes. Nevertheless, Vultr instances are an excellent option for long term use where adding fault tolerance to an existing application is not viable.

The best performance (63 seconds) was attained by the single-node cluster comprised of one `r6idn.16xlarge` instance, which has 64 logical cores. However, this cluster is 2.3 times more expensive than a 8-node cluster comprised of `c5.2xlarge` instances, which can also attain a decent performance (98 seconds). We didn't run tests with multiple `r6idn.16xlarge` instances given the default 128 vCPU limit imposed by AWS.

Figure 12 – *HEAT* execution times and horizontal scaling (no fault tolerance).



Source: Produced by the author.

5.3 Fault Tolerance Strategies

To assess the effectiveness of the proposed fault tolerance strategies, we conducted experiments comparing the failure-prone and fault-tolerant versions of the *HEAT* application outlined in Section 4.3. These experiments included variant implementations utilizing BLCR, DMTCP, ULFM, and SCR to explore the performance footprint and incurred costs of each proposed fault tolerance technology and mechanism.

The different implementations of *HEAT* were evaluated with clusters comprised of AWS instances only, the same used for the scalability tests, and described in Table 7. We run 200 iterations of the heat `large` configuration on 8-node (`t3.2xlarge`, `c5.2xlarge`, and `i3.2xlarge`), and 1-node (`r6idn.16xlarge`) clusters. Spot evictions are emulated using a helper script designed to terminate a single instance at specific points during execution, namely at one-third and two-thirds of the total iteration count. When a failure happens, *HPC@Cloud* automatically requests a new spot instance. We limited our analysis to instances where the spot infrastructure was immediately available, excluding any results from instances when the provider was unable to meet our request — a situation that occurred only once during our experiments.

The periodic C/R approaches are all configured to save checkpoints at fixed intervals — every 10 iterations of *HEAT*. The metrics used for analysis are essentially the total execution times and incurred cloud infrastructure costs. Unfortunately, it wasn't possible to use a single MPI distribution or operating system for our tests, given the technological restrictions of each fault tolerance mechanism. The evaluated scenarios are detailed in Table 8.

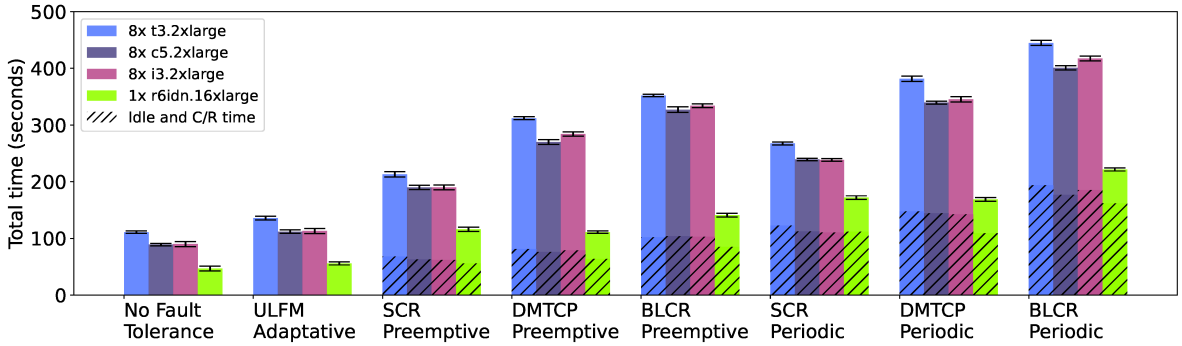
Figure 13 showcases the total execution time of the `large` *HEAT* workload when applying each fault tolerance and checkpointing strategy. The results shown in Subfigure 13(a) demonstrate the advantage of less frequent checkpointing in reducing overall execution time, particularly when comparing periodic and preemptive strategies. However, reducing checkpointing frequency can lead to increased re-work when an application needs to be restarted. Therefore, it is advisable for users to opt for more frequent pe-

Table 8 – Test scenarios for fault tolerance evaluation.

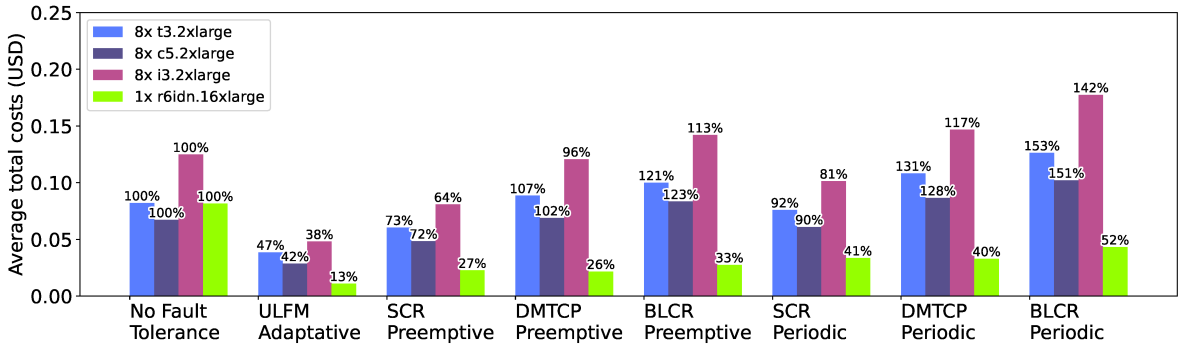
Fault Tolerance Technology	Checkpoint Strategy	Total Checkpoints	Total Failures (Restores)	Operating System	Kernel Version	MPI Library Release
—	—	0	0	AL 2023	v6.1.58	OpenMPI 5.0.0
BLCR	Periodic	20	2	CentOS 7.3.1611	v3.10.0	MVAPICH2 2.3.7
BLCR	Preemptive	2	2	CentOS 7.3.1611	v3.10.0	MVAPICH2 2.3.7
DMTCP	Periodic	20	2	AL 2023	v6.1.58	MVAPICH2 2.3.7
DMTCP	Preemptive	2	2	AL 2023	v6.1.58	MVAPICH2 2.3.7
SCR	Periodic	20	2	AL 2023	v6.1.58	OpenMPI 5.0.0
SCR	Preemptive	2	2	AL 2023	v6.1.58	OpenMPI 5.0.0
ULFM	—	0	2	AL 2023	v6.1.58	OpenMPI 5.0.0

Figure 13 – Evaluation of the fault tolerance strategies over different clusters.

(a) Total execution times for HEAT (large) with each fault tolerance approach.



(b) Costs for fault tolerance experiments, with percentages relative to "no fault tolerance" costs.



Source: Produced by the author.

riodic checkpointing when dealing with spot instance types that are prone to failure. Determining the optimal checkpointing frequency, however, can be challenging. Our findings indicate that the preemptive C/R strategy offers an extreme advantage, especially when using system-level methods that require disk I/O, such as BLCR and DMTCP. Unfortunately, not all providers offer support for spot eviction notices, making this strategy provider-dependent.

Regarding the fault tolerance techniques, the use of ULFM for adaptive execution is shown to be the most cost-effective. This method avoids downtime while waiting for replacement infrastructure after spot instance evictions, and benefits from faster data access due to storing checkpoint data in Remote Access Memory (RAM). Despite these advantages, implementing ULFM-based fault tolerance is more complex, especially in comparison to DMTCP and BLCR, which require no changes to the application.

Focusing on costs, the use of a single `r6idn.16xlarge` spot instance with an 88% discount leads to significant savings, as shown in Subfigure 13(b). Savings ranged from 48% with BLCR periodic checkpointing to 87% with ULFM adaptive execution, showing that the ULFM fault tolerance overhead slightly increases the cost footprint by about only 1% in our tests.

When we look at the types of instances used in the clusters, the more expensive

`i3.2xlarge` did not improve performance over the `t3.2xlarge` and `c5.2xlarge` instances. This is mainly because the workload does not rely heavily on storage speed, and thus the higher cost of the storage-optimized `i3.2xlarge` machine, which includes a faster but underused SSD, does not pay off. Similarly, `t3.2xlarge` has more memory but a slower processor and network, resulting in longer execution times. This highlights the importance of choosing the right infrastructure for the workload type. The right choice is just as crucial as a lightweight fault tolerance mechanism.

Finally, it is also important to note that for short tasks which only take a few minutes — like the *HEAT* workload tested in this study — the time it takes to provision a replacement spot instance and restore execution from a checkpoint file is relatively long (depicted in Figure 13 as "Idle and C/R time"). However, for much larger tasks, like those simulated by *DynEMol*, which can take days or weeks, this waiting time becomes insignificant. In such cases, system-level checkpointing methods like BLCR and DMTCP could be more cost-effective than they seem from the results shown in Subfigure 13, specially when considering the software development and migration expenses associated with the application-level methods.

In the next section we continue our experimental analysis, detailing the migration of *DynEMol* to AWS.

5.4 *DynEMol* Migration Analysis

DynEMol's migration evaluation is mostly focused on comparing the performance of this real-world HPC application running in the cloud versus the performance when running in a traditional on-premise HPC cluster. The Department of Physics at UFSC houses an HPC cluster named *FISICA-01*. This cluster is composed by four homogeneous nodes, each powered by Intel Xeon E5–2687W CPUs equipped with 16 physical cores with Hyper-Threading (HT) enabled (32 virtual cores). The nodes are interconnected via a high-speed InfiniBand network, with Remote Direct Memory Access (RDMA) capabilities, achieving high-speed data transfers. The cluster operating system is based on the Linux Ubuntu 20.04.4 LTS distribution.

Our evaluation aims to assess *DynEMol*'s performance using public cloud infrastructure and the minimum amount of resources required to attain performance similar to *FISICA-01*'s output. We also assessed a variety of different AWS cluster configurations, mirroring the setup of the *FISICA-01* cluster. Given that *DynEMol* has its own fault tolerance method embedded into the application, we considered clusters composed of both spot and on-demand instances in AWS. Table 9 shows the cluster configurations used in *DynEMol*'s experimental evaluation.

All AWS clusters are composed of `c5n.9xlarge` instances due to its close resemblance to the *FISICA-01* nodes. We executed preliminary tests varying the number of MPI ranks per cluster node and number of OpenMP threads per rank, which revealed

Table 9 – Cluster configurations evaluated with *DynEMol*.

Platform	Nodes	Cores* per Node	Infrastructure Type	Storage	Network Adapter
AWS	4	36	VM – On-demand	EBS	ENA (TCP/IP)
	4	36	VM – Spot	EBS	ENA (TCP/IP)
	4	36	VM – Spot	EBS	EFA
	4	36	VM – Spot	FSx	EFA
FISICA-01	4	32	Baremetal	SSD	InfiniBand

*Virtual (logical) cores with hyperthreading enabled.

that *DynEMol* attained the best performance when executed with four MPI ranks and four OpenMP threads per cluster node. We thus kept this configuration as the standard one for all of the remaining experiments. To investigate the performance implications of *DynEMol*'s checkpointing mechanism, we measured the time required to save checkpoint files and restart execution from a checkpoint.

We carried out simulations using *DynEMol* that describe the vibrational relaxation of photoexcited molecular systems, whereby a photon excites the molecular system from the ground quantum-state to an unoccupied quantum-state of higher energy, thus leaving the ground state unoccupied (with an excitation known as a "hole"). As the simulation progresses, the photo-excited high-energy electron decays back to the ground state and eventually annihilates the hole. In these simulations, electronic decay occurs through the generation of vibrational states on the molecular arrangement. Therefore, electronic energy is converted into vibrational energy of the molecular system.

We executed three different molecular simulations using the parameters described in Table 10. Only one isolated molecule is in the gas phase in the small system. The medium and large systems comprise a molecular dye attached (anchored) to the surface of a titanium dioxide (TiO2) anatase cluster. In these cases, in addition to generating vibrations in the molecule and the cluster arrangements, the photoexcited electron is transferred from the molecular dye into the TiO2 cluster (the process is called interfacial electron transfer). This type of inorganic substrate sensitized with molecular dyes is used in photoelectrochemical fuel cells.

We pursue the following primary objectives when running *DynEMol* across diverse cluster architectures in AWS (on-demand and spot) as well as on the *FISICA-01* HPC cluster: (i) assess both the performance enhancements and cost-effectiveness of FSx for Lustre relative to EBS; (ii) investigate the performance benefits derived from employing ENA and EFA to optimize network communications; and (iii) gauge the per-

Table 10 – *DynEMol* workload sizes tested.

Workload Size	Quantum Atoms	Orbitals	Force Pairs	Time-Steps (t_s)	Simulated Time
Small	35	95	595	$100,000$	1 ps
Medium	451	$2,351$	$4,656$	100	10^{-3} ps
Large	628	$3,412$	$4,005$	100	10^{-3} ps

formance overhead associated with *DynEMol*'s fault tolerance mechanism when a variable number of simulated spot evictions occur during the execution of the application.

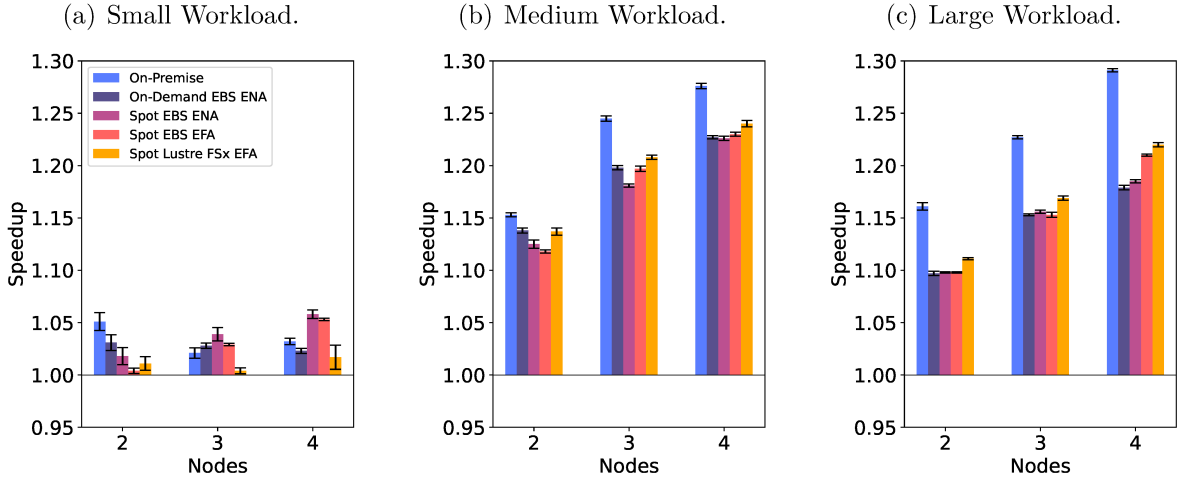
For each scenario, we compute the speedup achieved by *DynEMol* when multiple nodes of the cluster are used. These speedups are relative to the parallel execution of *DynEMol* using a single node of the cluster. All speedups of spot clusters are relative to the non-optimized spot cluster variant (EBS/ENA). To evaluate the cost-effectiveness of each cluster configuration, we vary the number of nodes and induce spot instance evictions, analyzing the number of executed time-steps per USD spent.

The frequency of checkpoints, denoted by f, is determined for each spot cluster according to the time-steps t_s, as detailed in Table 10. The frequency is calculated using the function $f(t_s) = \frac{t_s}{10}$. We do not make checkpoints when running *DynEMol* in *FISICA-01* and on-demand clusters. Figure 14 showcases the results obtained from the three workload sizes (Table 10) across the five cluster configurations (Table 9). This initial analysis does not consider the impact of spot instance evictions.

As noted in Subfigure 14(a), horizontal scaling performed poorly for small workloads, producing a negligible speedup on all clusters. The enhanced networking and storage solutions on the spot cluster (EFA and FSx for Lustre) did not yield significant improvements for small workloads either. Results in Figure 14 present a better horizontal scaling with medium and large workloads for all cluster configurations, achieving speedups of up to 1.22× (FSx for Lustre) and 1.29× (on-demand clusters) over 4-node clusters. The advantages of using EFA are more apparent for larger workloads, producing up to 9% increase in performance compared to ENA. The results further demonstrate that EFA can offer scalable performance for MPI applications that approaches the levels achieved with the InfiniBand connection available in the *FISICA-01* cluster.

Nevertheless, all speedup results look disappointing, even on bare-metal hard-

Figure 14 – *DynEMol* speedups achieved with different cluster configurations.



Source: Produced by the author.

ware. This leads us to consider two potential explanations: (i) the parallel version of Dynemol is not optimized effectively, or (ii) the "large" workload may not be large enough to fully leverage the advantages of parallel processing.

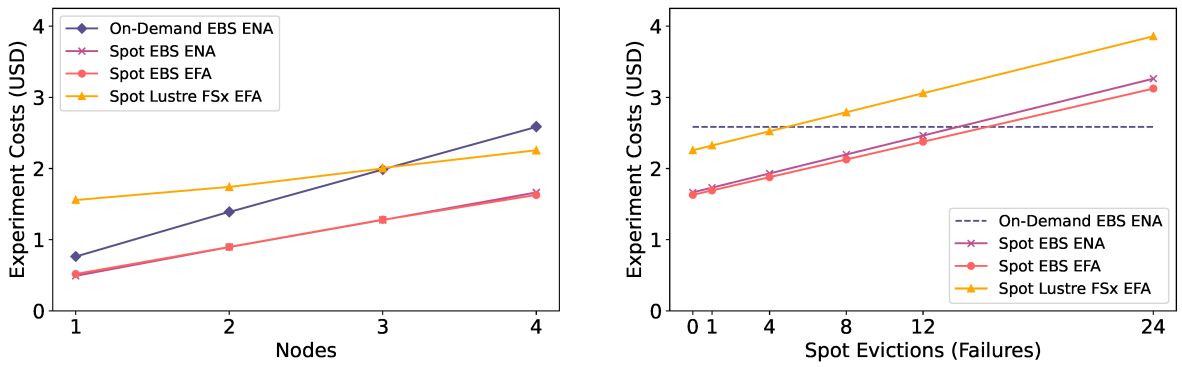
Figure 15 illustrates the monetary costs (in US dollars) of running *DynEMol* on AWS with the large workload across a variable number of nodes with no spot evictions (Subfigure 15(a)) and with four nodes and a variable number of spot evictions (Subfigure 15(b)). Moreover, based on the results in Subfigure 15(a), we can conclude that FSx for Lustre becomes cost-effective when running larger workloads in large clusters. In our tests with four-node clusters, using FSx for Lustre with spot instances was cheaper than a standard on-demand cluster with EBS for storage. Although spot instances can be much cheaper than on-demand infrastructure, the performance impact of the fault tolerance strategy can easily translate to higher infrastructure costs (Subfigure 15(b)). The checkpoint restoration process employed in *DynEMol* is CPU-intensive, demanding little I/O throughput for performance. This is reflected in the steep cost increase for experiments with FSx for Lustre (Subfigure 15(b)), indicating that experiencing more than four spot evictions could lead to costs higher than simply opting for on-demand instances.

Figure 16 presents the ratio between executed time-steps and each US dollar spent on cloud resources, highlighting the computational cost-efficiency that was extracted from each cluster configuration. Overall, spot clusters offer a much more cost-effective alternative than on-demand clusters when there are no spot evictions (Subfigure 16(a)), executing up to 54% more time-steps for the same price. However, the cost-efficiency of spot clusters undergoes a steep decline with a rising number of failures (Subfigure 16(b)), potentially rendering the utilization of spot instances less advantageous. Still, spot clusters with EBS were cheaper than on-demand clusters with up to 12 failures.

Finally, we noticed that FSx for Lustre is only better than EBS when employed in larger clusters (four nodes or more). Given that FSx for Lustre is approximately 233%

Figure 15 – *DynEMol* execution costs (large workload).

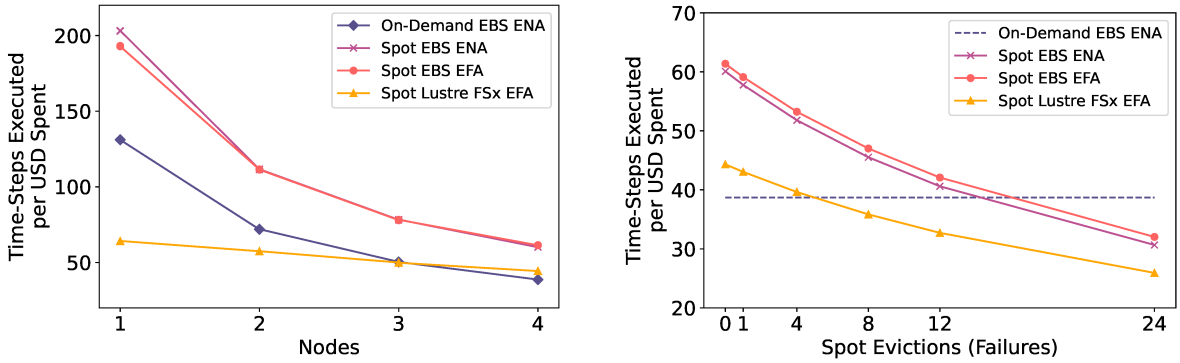
(a) Increasing Nodes (0 failures). (b) Increasing Failures (4 nodes).



Source: Produced by the author.

Figure 16 – *DynEMol* time-steps per USD spent (large workload).

(a) Increasing Nodes (0 failures). (b) Increasing Failures (4 nodes).



Source: Produced by the author.

costlier than EBS, it may only be worth for heavy I/O applications, which is not the case of *DynEMol*.

In the next section we continue our experimental analysis, showcasing the performance and costs of using Singularity containers to run workloads with *HPC@Cloud*.

5.5 Containerized Execution

To assess the impact of containerization technology in shared public cloud environments, we evaluated the performance of *HEAT* and NPB benchmarks on AWS clusters, comparing native execution against running within Singularity containers.

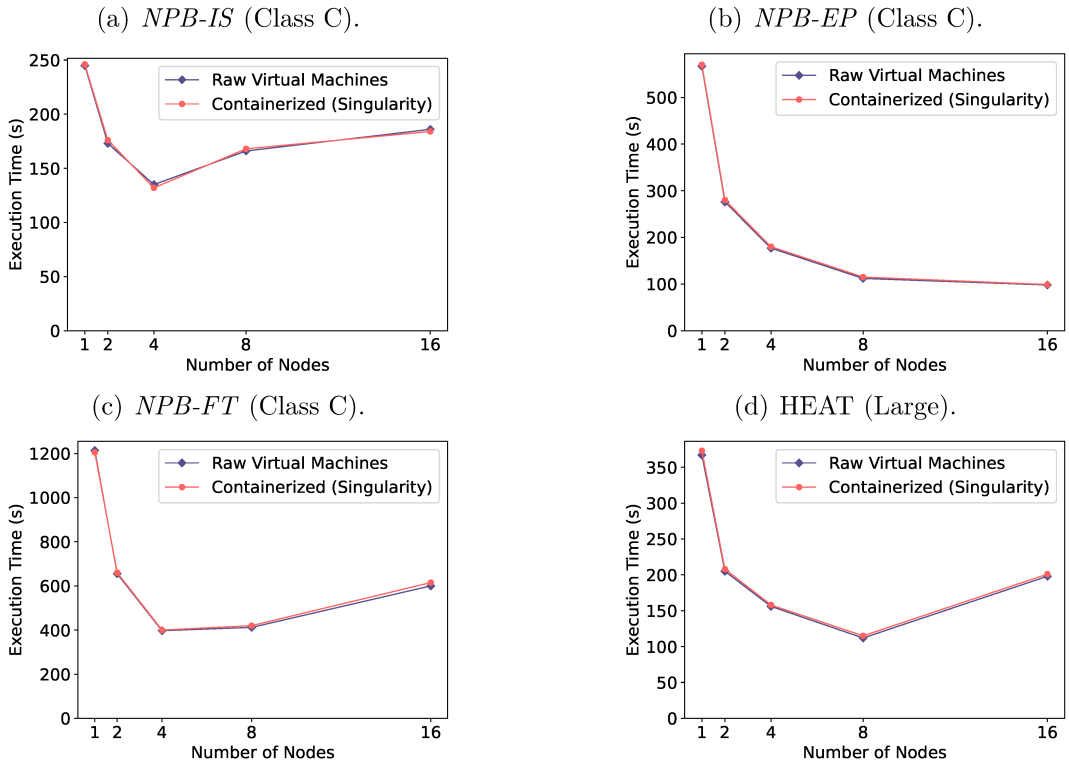
We test Singularity using 1, 2, 4, 8 and 16-node AWS clusters comprised of `c6i.2xlarge` instances, considering the average results of 5 repetitions for each scenario. For the image OS we use Amazon Linux 2023 with the v6.1.58 Kernel. Moreover, we select the community version of Singularity (v4.0.1) for testing. The MPI library implementation used is MVAPICH2 2.3.7, both on the VMs and inside the Singularity containers. We do not test failure scenarios for containerized executions, as support for it is limited and still under research and development in *HPC@Cloud*.

We executed the workloads detailed in Table 11 with and without Singularity running a one container per MPI rank mapping. Figure 17 showcases the results obtained, emphasizing the comparative performance analysis of the applications when executed on

Table 11 – Workloads considered for containerization experiments.

Application	Workload Label	Input Size
NPB-EP	`Class C`	$n = 2^{32}$
NPB-FT	`Class C`	$n = 512^3$
NPB-IS	`Class C`	$n = 2^{31}$
HEAT	`large`	$n = 2,048$

Figure 17 – Containerized execution performance evaluation.



Source: Produced by the author.

AWS with and without containers.

Overall, no significant discrepancies were observed in the execution times when applications were executed with or without containers. Consequently, a preliminary conclusion can be drawn that the overhead cost of utilizing Singularity containers and the hybrid execution approach of MPI has negligible performance footprint. As anticipated, due to its inherently parallel nature, the *NPB-EP* application displayed favorable scalability. Conversely, the *NPB-IS* and *NPB-FT* applications encountered scalability challenges as the number of nodes increased.

The adoption of container technology introduces some incremental costs. Primarily, the preparation of a machine image containing Singularity and the MPI distribution significantly reduces cluster spawn time. However, this convenience incurs a storage fee charged by the cloud provider. For instance, Amazon charges around 1.4 USD monthly for storing the AMI utilized in our experiments. Additionally, to expedite workload setup, leveraging a pre-built container image from an image registry, rather than assembling it from the ground up, is preferred. This approach, however, leads to further expenses associated with infrastructure and bandwidth, costing approximately 0.12 USD per container image each month with Amazon Elastic Container Registry (ECR).

In the next section, we conclude our experimental analysis with results regarding costs forecasting in *HPC@Cloud*.

5.6 Costs Forecasting

Our goal is to explore the potential of leveraging machine learning to enhance decision-making in the planning of cloud infrastructure for large-scale workloads. To initiate this exploration, we trained two simple models using a constrained dataset, laying the groundwork for more extensive research in the future.

5.6.1 Adopted Evaluation Method

To assess the precision of our experimental cost prediction methodology, we evaluate the efficacy of the XGBoost model described in Section 4.5. We additionally test a simple and cheaper Linear Regression model trained using the same *HPC@Cloud* workflows. The model training process involved the utilization of a dataset derived from executing the NPB benchmarks workloads — *NPB-EP*, *NPB-FT*, and *NPB-IS* of the standard test problems: Classes A, B, and C. Each class of problems has around a 4× size increase going from one class to the next. The diversity of workload patterns in the NPB benchmarks ensures that the training data encapsulates a wide variety of computational and data access patterns, which is paramount for a holistic evaluation.

In a departure from prior methodologies used in Munhoz & Castro (2023), we refined our costs prediction evaluation strategy to enhance the validity of the predictive models. This was achieved by implementing a cross-application assessment where the models, trained on NPB benchmarks, were applied to predict the execution costs of an entirely separate application, the *HEAT* program. This approach circumvents the potential overfitting to a single application scenario, thereby providing a more generalized and robust assessment of model performance.

For a comprehensive evaluation, we calculated prediction errors by comparing the models' cost predictions against actual observed execution times. We specifically employed the Root Mean Square Error (RMSE) and Mean Absolute Error (MAE) metrics, alongside a residual plot analysis, to provide a nuanced understanding of the models' accuracy.

5.6.2 Model Training

The training dataset used to build our model is comprised by 1 target variable, `w_time`, representing the total execution time in seconds, and 6 features, described in Section 4.5 and further detailed in Table 12. The assembled dataset has 384 rows, with sample data shown in Listing 2.

Table 12 – Variables used for model training.

Variable Name	Variable Type	Variable Role	Description
w_time	numerical	Target	Total workload execution time
w_class	categorical	Feature	Value provided by the user (workload-specific)
c_nodes	numerical	Feature	Number of cluster nodes
c_vcpus_pn	numerical	Feature	Number of vCPUs available per cluster node
c_ram_pn	numerical	Feature	Amount of RAM per cluster node
c_bandwidth	numerical	Feature	Node-to-node measured bandwidth
c_iops	numerical	Feature	Instructions per second when writing to shared storage

Listing 2 – Raw dataset sample for training (in CSV format).

```
1  w_time,w_class,c_nodes,c_vcpus_pn,c_ram_pn,c_bandwidth,c_iops
2  1212,npb-ft-C,1,8,21.0,22.9,2262
3  657,npb-ft-C,2,8,21.0,23.1,2312
4  399,npb-ft-C,4,8,21.0,22.2,2332
5  412,npb-ft-C,8,8,21.0,21.3,2295
6  587,npb-ft-C,16,8,21.0,22.2,2254
```

Source: Produced by the author.

5.6.2.1 Linear Regression

To train the Linear Regression model, which accepts only numerical variables, we apply the following transformations to the raw dataset partially shown in Listing 2:

1. **One-Hot Encoding**: Because `w_class` is a categorical variable representing different classes of workloads, it should be converted into numerical form using one-hot encoding. This process creates binary columns for each category of the variable, ensuring that the model can interpret and use this information correctly.
2. **Feature Scaling**: Also known as normalization, this transformation is important for ensuring that all features contribute equally to the model training, especially since linear regression is sensitive to the scale of input features. For example, `c_ram_pn`, `c_bandwidth`, and `c_iops` have different scales, which could affect the model's performance. We apply a Min-Max scaling to convert values to be in the range of 0 to 1, by subtracting the minimum value and dividing by the range of the data.

5.6.2.2 XGBoost

For the XGBoost model, our transformation approach is slightly different due to its inherent capability to handle a broader range of data distributions. Numerical variables require no specific transformation in this case, as XGBoost, being a tree-based model, is less sensitive to variable scale. We simply one-hot encode the `w_class` variable for training XGBoost.

5.6.3 Model Scoring and Evaluation

The selected error metrics (RMSE and MAE) offer a dual perspective: the RMSE emphasizes the impact of large errors, while the MAE provides a straightforward average error magnitude. The residual plot in Figure 19 further allows us to visualize the variance in prediction errors across the range of costs, giving us deeper insights into the predictive quality and potential biases of our models. In our tests, the main objective is not merely to predict costs but to gauge the accuracy of these predictions vis-à-vis actual incurred costs. This distinction is pivotal since the value of a prediction model lies in its ability to approximate real-world outcomes.

The blind total execution time forecasting results for *HEAT* when using Linear Regression and XGBoost models are showcased in terms of the RMSE and MAE error metrics in Figure 18. Forecasted and measured values are shown in execution time (seconds). Cost estimations are then computed from the total cluster costs (including storage and networking) multiplied by the predicted execution times, and results are presented in Figure 19.

Subfigure 18(a) elucidates the performance of the Linear Regression model, which incurs a relatively high RMSE of 26.20s and MAE of 18.25s. This indicates that while Linear Regression provides a baseline model, it struggles to handle the complexity inherent in the dataset, potentially due to its assumptions of linearity.

In stark contrast, Subfigure 18(b) presents the XGBoost model as significantly more precise, boasting an RMSE of only 7.97s and MAE of 7.28s. The enhanced accuracy of the XGBoost model can be attributed to its advanced ML algorithm that effectively captures complex non-linear patterns and interactions among variables. This is particularly beneficial for the tackled problem of costs prediction with heterogeneous datasets.

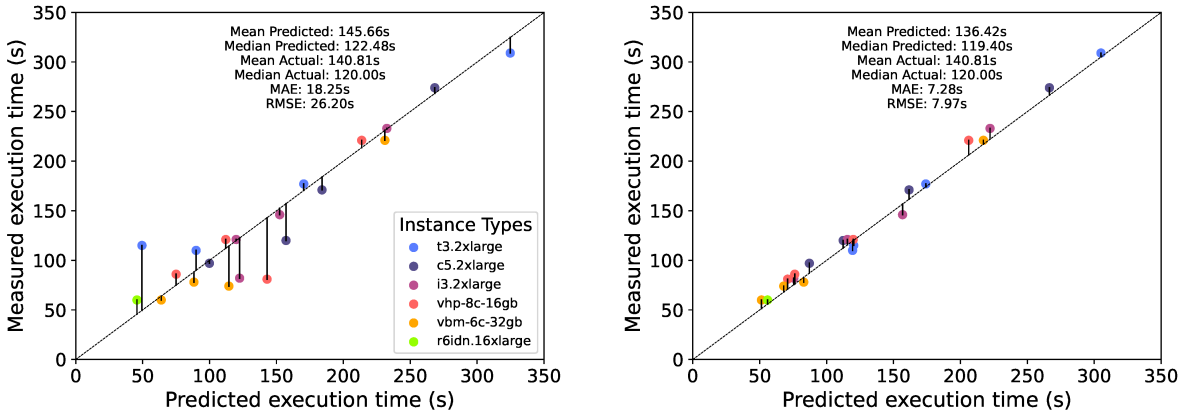
When we compute the total costs from the predicted execution times, given that our workloads are relatively small, the error is virtually negligible. When rounding to cents (the minimum billable amount in AWS), Linear Regression and XGBoost boast an RMSE of 0.02\$ and 0.01\$ respectively, for workloads with a mean actual cost of 0.10\$, or 10 times the RMSE.

In the context of transient infrastructures, it is important to recognize that a multitude of factors affect the aggregate execution costs. These encompass the overhead associated with checkpointing and the fluctuating availability of instances. Moreover, while AWS hourly spot rates have become more stable, as reported by Amazon Web Services (2018), they continue to be a variable difficult to predict, which influences the overall expenses. Additionally, the absence of a predictive mechanism within *HPC@Cloud* for forecasting spot instance evictions constitutes a significant limitation in providing accurate cost estimates for spot infrastructure usage.

The unpredictability of spot instance eviction frequency and the absence of publicly accessible provider data make it infeasible to develop a predictive dataset econom-

Figure 18 – Predicted execution time residuals plot.

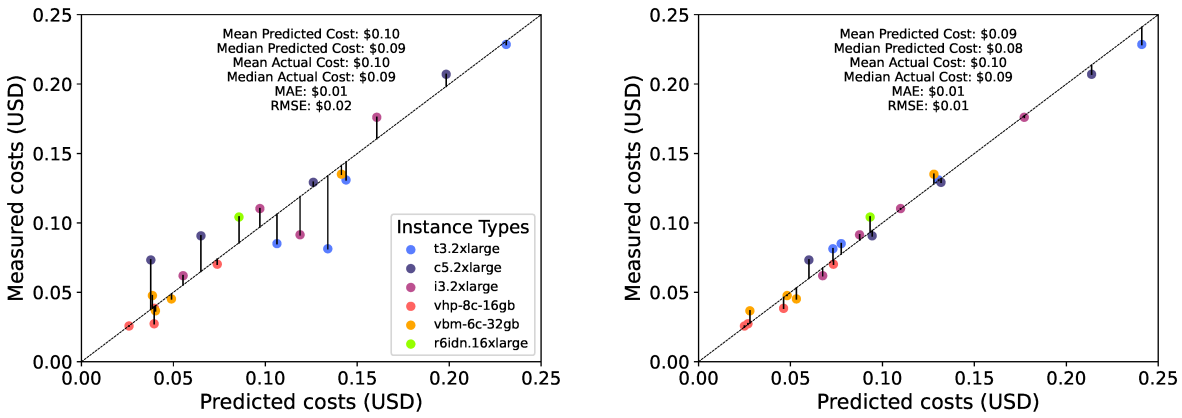
(a) Linear Regression. (b) XGBoost Model.



Source: Produced by the author.

Figure 19 – Computed costs predictions residuals plot.

(a) Linear Regression. (b) XGBoost Model.



Source: Produced by the author.

ically. However, when employing fault-tolerant techniques with negligible overhead or dealing with long-duration applications, the impact of these limitations diminishes. Consequently, the cost estimations for such scenarios remain significantly aligned with those for on-demand instances, asserting the utility of our predictive models.

In conclusion, the discrepancy in the quality of the models underlines the importance of selecting the right algorithm for cost prediction tasks. While Linear Regression serves as a simple, interpretable model, XGBoost's advanced features and adaptability provide a more reliable and accurate tool for handling the complexities of cloud cost prediction, both for shorter and long-running workloads.

6 CONCLUSION AND FUTURE RESEARCH

In this work, we tackled several of the HPC and Cloud Computing convergence software gaps identified by Netto et al. (2018), introducing the *HPC@Cloud* toolkit to facilitate the migration of HPC workloads to public cloud platforms. Although we focused on tightly-coupled MPI applications, our contributions extend to other types of workloads, such as bag-of-tasks. *HPC@Cloud* includes software modules for automated virtual environment configuration, cloud infrastructure management, and costs forecasting. We have demonstrated its usefulness and efficacy by running multiple types of workloads and benchmarks, including a real-world HPC application (*DynEMol*), over different cluster configurations in real public cloud platforms (AWS and Vultr).

In the following sections, we draw our main conclusions about each one of the main contributions of this work, as well as future research.

6.1 Cluster Scalability, Technologies, and Resource Management

We evaluated cloud scalability by executing both small and large classes of the *HEAT* application across various cluster sizes. The results indicated a clear pattern: large applications gain significantly from horizontal scaling, whereas smaller applications are best served by single-instance execution. The effectiveness of using large clusters with resource-intensive nodes in public cloud settings is notably constrained by the resource limits imposed by most providers. This limitation somewhat undermines the very principle of public cloud platforms, as it forces users to manually request resources and await non-automated approval from the provider.

In this dissertation, we investigated a variety of cloud computing technologies related to networking and storage, such as FSx for Lustre, EBS, EFS, ENA, EFA, among others. Yet, numerous other avenues remain unexplored. For instance, future work will consider testing cluster topologies that forego centralized shared storage, potentially enhancing workload execution speeds by eliminating potential I/O bottlenecks. Such an approach would necessitate a shift in both application-level strategies and the interaction model within *HPC@Cloud*, adapting to a scenario where files are not uniformly accessible across all cluster nodes.

Moreover, one of the most important features of our toolkit, which is the fault tolerance support with eviction alarms from AWS CloudWatch, would need some code adaptations to work in a transparent manner with a different cloud provider, such as Microsoft Azure. The adaptations required, besides the addition of Terraform recipes for clusters using the new provider, include the implementation of several helper functions to fetch node IP addresses, check node state, and identify eviction alarms (if available). The integration of a second ephemeral infrastructure provider is one of the highest priorities

in *HPC@Cloud*'s roadmap.

The integration of hardware accelerators, particularly GPUs, into the supported computing infrastructure is also critically important. Modern HPC applications are increasingly designed to leverage the computational power of GPUs, achieving substantial performance enhancements as a result. GPUs, with their highly parallel structure, are exceptionally well-suited for the complex mathematical and simulation tasks that are typical in HPC environments. This compatibility allows for significant speedups in processing times, enabling more efficient and faster computations compared to traditional CPU-only setups. Although *HPC@Cloud* can spawn and manage instance types with GPUs, we did not conduct empirical tests with them, being another important area for future research.

6.2 Fault Tolerance Strategies and Ephemeral Instances

To be able to explore ephemeral cloud infrastructure such as AWS spot instances, we proposed and evaluated a combination of four different fault tolerance technologies for MPI workloads: (a) Berkeley Labs Checkpoint Restart (BLCR), (b) Distributed MultiThreaded CheckPointing (DMTCP), (c) Scalable Checkpoint Restart (SCR), and (d) User-Level Failure Mitigation (ULFM). We also employed three restoration strategies: (a) Periodic Checkpointing, (b) Preemptive Checkpointing, and (c) Adaptative Restoration (with ULFM only).

Results in Figure 13 corroborate a stark decrease in infrastructure costs when using AWS spot instances, specially when applying lightweight application-level fault tolerance technologies such as SCR and ULFM. System-level fault tolerance with DMTCP and BLCR also proved useful for quickly making use of cheaper infrastructure without needing to change source code, although BLCR support ended with Linux Kernel 3.10.0. Collected results from our experiments also indicate that application performance can be heavily impacted by spot instance evictions, which often lead to idle times while awaiting provider resource provisioning. To alleviate this, we make use of provider-specific notification systems like AWS CloudWatch to preemptively checkpoint and manage resources, thus reducing idle times due to spot instance failures. When these alarm systems are not available, fault tolerant applications can still fallback to periodic checkpointing, albeit with a higher performance footprint. Using the ULFM APIs, we managed to completely nullify the idle time when running the *HEAT* application with spot instances by oversubscribing MPI ranks to existing nodes during failures, which resulted in higher performance and lower infrastructure costs, as shown in Figure 13.

Nevertheless, our cost analysis suggests that spot resource utilization by itself does not always result in savings, as evidenced by the comparable cost-effectiveness of persistent Vultr instances — persistent 8-node `vhpc-8c-16gb` clusters attained almost the same performance as the ephemeral 8-node `t3.2xlarge` clusters when running *HEAT*-large (Figure 12), costing around 0.4 USD less per hour. This comparison highlights the

importance of careful instance selection based on your specific workload requirements. Moreover, our investigation into high-end instances reveals that while they can be more cost-effective when spot options are available, reliance on a single-node setup increases risk, as revocations necessitate fallback to expensive persistent storage solutions.

Future directions include the research into simplifying the adoption of application-level fault tolerance, by devising automatic code injection methods or other mechanism to facilitate the development of simple C/R capabilities that could quickly enable the use of ephemeral infrastructure. We also plan to investigate the feasibility of integrating ULFM with *Charm++*[1] and *StarPU*[2], two distributed programming models that leverage MPI for inter-process communication that can potentially provide fault tolerance with ULFM in a transparent way.

6.3 Migrating Legacy Applications

We migrated the *DynEMol* simulation tool to AWS using the *HPC@Cloud* software toolkit, mainly to assess its usefulness in a real-world scenario. We tested various cluster configurations, evaluating different available technologies, such as spot and on-demand instances for computational infrastructure; EBS and FSx for storage; and EFA and ENA for networking.

Our experimental results suggest that while cloud-based resources configured optimally exhibit slightly inferior scalability compared to the on-premise infrastructure tested, they still offer substantial benefits. The existing fault tolerance mechanism in *DynEMol*, specifically its checkpointing feature, is sufficiently lightweight and its performance footprint is virtually undetectable for medium to large workloads. This feature enables us to increase checkpointing frequency, thereby circumventing unnecessary rework when restoring from these checkpoints. Moreover, limiting the number of instances enhances the cost-effectiveness of spot infrastructure due to the associated reduction in failure probability. Utilizing spot instances and maintaining a finely-tuned parallel composition for the workload is crucial for achieving cost-effectiveness. Our results corroborate the conclusions of related studies, discussed in Chapter 3. These suggest that fewer fat instances, equipped with hundreds of cores, may be more suitable than a large number of small nodes. The downside for larger instances rests on their usual unavailability and inexistence in most providers, especially low-cost ones.

Regarding the storage technologies available in AWS, although FSx can be costly, the benefits outweigh the costs when a high number of nodes (at least four) perform parallel I/O. Furthermore, advances in networking technologies have paved the way for high-bandwidth, low-latency communications in the public cloud. When testing networking technologies, we observed better performance when using EFA, attesting to this fact.

[1] https://charmplusplus.org/
[2] https://starpu.gitlabpages.inria.fr/

Furthermore, EFA can be enabled in AWS at no additional cost, further improving the speedup efficiency.

Future research includes a deeper analysis of *DynEMol*'s parallelization methods to further improve the obtainable speedups, both through code improvements and fine-tuning of the number of MPI ranks and OpenMP threads. We also intend to test the GPU-accelerated version of *DynEMol* using cloud resources, a different range of instance types, and cloud providers. Future objectives also include the development of a cloud-native version of *DynEMol* for easy deployment by any researcher in the field.

6.4 Containerized Execution

Through integrating Singularity into *HPC@Cloud*, we enabled containerized execution, which yielded performance comparable to native VM execution, as clearly shown in Figure 17. This finding underpins the potential of container technology in cloud-based HPC applications. In conclusion, the practicality of deploying containers in a single-purpose cluster designed to run only a single workload before decommissioning is debatable, as it potentially introduces unnecessary operational complexity.

The design of *HPC@Cloud* is primarily guided towards execution of workloads on single-purpose clusters, yet it is versatile enough to facilitate the creation of long-lived clusters serving multiple users. On the other hand, within a persistent, multi-user shared cluster hosted on a public cloud platform, the utilization of containers becomes significantly more beneficial due to the isolation they provide. Workload managers like SLURM can be seamlessly integrated by specifying initialization commands within *HPC@Cloud*.

In such multi-user environments, containerized execution emerges as a particularly relevant approach, albeit the absence of a scheduler to efficiently manage containers across distributed nodes presents a challenge. Incorporating this functionality would inevitably increase operational complexity and consume valuable computational resources that could otherwise be dedicated to running workloads directly. Therefore, due to these considerations, we have decided set aside further developments in containerization within *HPC@Cloud*. Our focus will remain on optimizing the single-purpose cluster model for public cloud environments.

Nevertheless, a significant area of research that could enhance *HPC@Cloud* involves integrating fault tolerance techniques for containers. This would facilitate the utilization of spot instances, offering a robust solution for cost-efficient computing resources.

6.5 Costs Forecasting

In this dissertation, we established the foundation for integrating machine learning models into *HPC@Cloud*, aimed at predicting execution time and costs. This ini-

tiative is designed to support decision-making processes in selecting the optimal cluster infrastructure for handling large workloads, allowing the end-user to make an informed decision before committing resources. We test two straightforward models — XGBoost and Linear Regression — using data captured by *HPC@Cloud* when running *HEAT* and NPB workloads. Our comparative study on cost forecasting unambiguously demonstrates the superiority of XGBoost over Linear Regression. XGBoost's robust handling of non-linear complexities renders it a valuable asset for precise forecasting, which is crucial in the dynamic and often unpredictable landscape of cloud resource pricing and allocation.

For future enhancements, we consider the integration of real-time market data for spot instances, potentially refining the accuracy of our predictive models by uncovering more nuanced relationships impacting costs. Furthermore, the development or integration of spot instance eviction predictors is another promising avenue. Creating a predictive model capable of estimating the likelihood and timing of spot instance evictions could significantly improve cost estimations for ephemeral infrastructures. However, it is crucial to note that such a model would be provider-specific and may not seamlessly integrate into *HPC@Cloud* existing framework.

Finally, to accommodate the trend towards multi-cloud environments, the models could be trained with data from various cloud providers. This cross-platform approach to cost forecasting could significantly aid researchers and practitioners in making well-informed decisions regarding multi-cloud strategies, ensuring cost efficiency and optimization across different cloud platforms.

6.6 Other Future Directions for *HPC@Cloud*

The *HPC@Cloud* toolkit represents a significant step towards the seamless adaptation of legacy HPC applications to the cloud. Our contributions in this work not only demonstrate the toolkit's capabilities but also provide insights into the complexities of cloud resource management and fault tolerance in cloud HPC. By continuing to build upon these foundations, we aim to unlock the full potential of Cloud Computing for the HPC community.

We plan to deploy a centralized version of *HPC@Cloud*, which will then be accessed through a REST API, eliminating the need for dependencies to be installed in the user's machine, such as Terraform, Docker and provider's CLIs, simplifying use by researchers.

We also intend to add support for other two big public cloud providers: GCP and Azure. With this support, *HPC@Cloud* can truly become a multi-platform solution for cloud HPC.

BIBLIOGRAPHY

ABRAHAM, B.; REGO, L. G. C.; GUNDLACH, L. Electronic-Vibrational Coupling and Electron Transfer. **The Journal of Physical Chemistry C**, American Chemical Society, v. 123, n. 39, p. 23760–23772, 2019.

AL-ROOMI, M. et al. Cloud Computing Pricing Models: A Survey. **International Journal of Grid and Distributed Computing**, Science & Engineering Research Support soCiety (SERSC), Daedoek-Gu, Republic of Korea, v. 6, p. 93–106, 2013. ISSN 2005-4262.

ALJAMAL, R.; EL-MOUSA, A.; JUBAIR, F. A Comparative Review of High-Performance Computing Major Cloud Service Providers. In: **Proceedings of the 9th International Conference on Information and Communication Systems**. London, United Kingdom: Association for Computing Machinery, 2018. p. 181–186.

ALTHOFF, L.; MUNHOZ, V.; CASTRO, M. Análise de Viabilidade do Perfilamento de Aplicações de HPC Baseada em Contadores de Hardware na AWS. In: **Anais da XXIII Escola Regional de Alto Desempenho da Região Sul**. Porto Alegre, Brazil: Sociedade Brasileira de Computação (SBC), 2023. p. 45–48. ISSN 2595-4164.

Amazon Web Services. **New Amazon EC2 Billing Model**. 2017. Accessed at: 2023-11-06. Available from Internet: https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes/.

Amazon Web Services. **New Amazon EC2 Spot Pricing**. 2018. Accessed at: 2023-11-06. Available from Internet: https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/.

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**, Springer International Publishing, v. 10, p. 4567–4577, 2019.

BENEDICIC, L. et al. Sarus: Highly Scalable Docker Containers for HPC Systems. In: **ISC High Performance International Workshops**. Frankfurt, Germany: Springer International Publishing, 2019. p. 46–60.

BLAND, W. et al. An Evaluation of User-Level Failure Mitigation Support in MPI. In: **Recent Advances in the Message Passing Interface**. Berlin, Heidelberg, Germany: Springer International Publishing, 2012. v. 95, p. 193–203.

BLAND, W. et al. Post-Failure Recovery of MPI Communication Capability: Design and Rationale. **The International Journal of High Performance Computing Applications**, Sage Publications, v. 27, n. 3, p. 244–254, 2013.

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**. Berlin, Heidelberg, Germany: Springer International Publishing, 2023. p. 191–212. ISBN 978-3-031-29769-4.

BUYYA, R.; BROBERG, J.; GOSCINSKI, A. **Cloud Computing: Principles and Paradigms**. Hoboken, New Jersey, USA: John Wiley & Sons, 2011. ISBN 978-0-470-88799-8.

BUYYA, R. et al. A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade. **ACM Computing Surveys**, Association for Computing Machinery, London, United Kingdom, v. 51, n. 5, 2019. ISSN 0360-0300. Available from Internet: https://doi.org/10.1145/3241737.

CASALICCHIO, E.; IANNUCCI, S. The State-of-the-Art in Container Technologies: Application, Orchestration and Security. **Concurrency and Computation: Practice and Experience**, John Wiley & Sons, Hoboken, New Jersey, USA, v. 32, n. 17, p. 56–68, 2020.

CHHETRI, M. B. et al. On Estimating Bids for Amazon EC2 Spot Instances Using Time Series Forecasting. In: **Proceedings of the IEEE International Conference on Services Computing**. Honolulu, HI, USA: IEEE Computer Society, 2017. p. 44–51.

COGHLAN, S.; YELICK, K. **The Magellan Final Report on Cloud Computing**. Berkeley, USA: Lawrence Berkeley National Laboratory, 2011. 50–54 p. Available from Internet: https://www.osti.gov/biblio/1076794.

DANCHEVA, T.; ALONSO, U.; BARTOŇ, M. Cloud Benchmarking and Performance Analysis of an HPC Application in Amazon EC2. **Cluster Computing**, Springer International Publishing, p. 1–18, 2023.

EMERAS, J.; VARRETTE, S.; BOUVRY, P. Amazon Elastic Compute Cloud (EC2) vs. In-House HPC Platform: A Cost Analysis. In: **Proceedings of the IEEE International Conference on Cloud Computing**. San Francisco, USA: IEEE Computer Society, 2016. p. 284–293.

FERNANDEZ, A. Evaluation of the Performance of Tightly Coupled Parallel Solvers and MPI Communications in IaaS From the Public Cloud. **IEEE Transactions on Cloud Computing**, v. 10, n. 4, p. 2613–2622, Oct 2022. ISSN 2168-7161.

FERRÃO, L.; MUNHOZ, V.; CASTRO, M. Análise do Sobrecusto de Utilização de Contêineres para Execução de Aplicações de HPC na Nuvem. In: **Anais da XXIII Escola Regional de Alto Desempenho da Região Sul**. Porto Alegre, Brazil: Sociedade Brasileira de Computação (SBC), 2023. p. 37–40. ISSN 2595-4164.

GARTNER. **Worldwide Public Cloud Services Markets**. 2022. Accessed at: 2023-11-06. Available from Internet: https://www.gartner.com/en/newsroom/press-releases/2022-06-02-gartner-says-worldwide-iaas-public-cloud-services-market-grew-41-percent-in-2021.

GONG, Y.; HE, B.; ZHOU, A. C. Monetary Cost Optimizations for MPI-based HPC Applications on Amazon Clouds: Checkpoints and Replicated Execution. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. Austin, Texas, USA: Association for Computing Machinery, 2015. p. 1–12.

GUIDI, G. et al. 10 Years Later: Cloud Computing is Closing the Performance Gap. In: **Proceedings of the International Conference on Performance**

Engineering. Virtual Event: Association for Computing Machinery, 2021. p. 41–48. ISBN 978-1-450-38331-8.

HADJIDIMOS, A. Successive Overrelaxation (SOR) and Related Methods. **Journal of Computational and Applied Mathematics**, Elsevier Science Publishers B.V, Amsterdam, Netherlands, v. 123, n. 1, p. 177–199, 2000.

HARGROVE, P.; DUELL, J. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. **Journal of Physics: Conference Series**, IOP Publishing, Bristol, United Kingdom, v. 46, p. 494, 2006.

HARIYALE, H. et al. Load Balancing in Cluster Using BLCR Checkpoint/Restart. In: **Advances in Computing and Information Technology**. Berlin, Heidelberg, Germany: Springer International Publishing, 2012. p. 729–737. ISBN 978-3-642-31513-8.

HILL, M. D.; MARTY, M. R. Amdahl's Law in the Multicore Era. **Computer**, IEEE Computer Society, v. 41, n. 7, p. 33–38, 2008.

HURSEY, J. Design Considerations for Building and Running Containerized MPI Applications. In: **Proceedings of the 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC**. Atlanta, GA, USA: IEEE Computer Society, 2020. p. 35–44.

KITHULWATTA, W. M. C. J. T. et al. Adoption of Docker Containers as an Infrastructure for Deploying Software Applications: A Review. In: **Advances on Smart and Soft Computing**. Berlin, Heidelberg, Germany: Springer International Publishing, 2022. p. 247–259. ISBN 978-981-16-5559-3.

KJOLSTAD, F. B.; SNIR, M. Ghost Cell Pattern. In: **Workshop on Parallel Programming Patterns (ParaPLoP)**. Carefree, Arizona, USA: Association for Computing Machinery, 2010. ISBN 978-1-450-30127-5.

KURTZER, G. M.; SOCHAT, V.; BAUER, M. W. Singularity: Scientific Containers for Mobility of Compute. **PLOS ONE**, Public Library of Science, San Francisco, California, USA, v. 12, n. 5, p. 1–20, 2017.

LI, Z. et al. Efficiency or Innovation: The Long-Run Payoff of Cloud Computing. **Journal of Global Information Management**, IGI Global International Academic Publisher, Hershey, Pennsylvania, USA, v. 29, n. 6, p. 1–23, 2021.

MALLA, S.; CHRISTENSEN, K. HPC in the Cloud: Performance Comparison of Function as a Service (FaaS) vs Infrastructure as a Service (IaaS). **Internet Technology Letters**, John Wiley & Sons, Hoboken, New Jersey, USA, v. 3, n. 1, p. 131–137, 2020.

MARATHE, A. et al. Exploiting Redundancy for Cost-Effective, Time-Constrained Execution of HPC Applications on Amazon EC2. In: **Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing**. Vancouver, BC, Canada: Association for Computing Machinery, 2014. p. 279–290. ISBN 978-1-450-32749-7.

MATTSON, T. An Introduction to OpenMP. In: **Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid**. London, United Kingdom: Association for Computing Machinery, 2001. p. 3–3. ISBN 0-7695-1010-8.

MELL, P. M.; GRANCE, T. **The NIST Definition of Cloud Computing**. Gaithersburg, USA, 2011.

MOODY, A. et al. Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. New Orleans, Louisiana, USA: IEEE Computer Society, 2010.

MUNHOZ, V. et al. A Performance Comparison of HPC Workloads on Traditional and Cloud-Based HPC Clusters. In: **Proceedings of the International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)**. Porto Alegre, Brazil: IEEE Computer Society, 2023. p. 108–114.

MUNHOZ, V.; CASTRO, M. Benchmarking the Scalability of MPI-Based Parallel Solvers for Fluid Dynamics in Low-Budget Cloud Infrastructure. In: **Anais da XXII Escola Regional de Alto Desempenho da Região Sul**. Curitiba, Brazil: Sociedade Brasileira de Computação (SBC), 2022. p. 77–78. ISSN 2595-4164.

MUNHOZ, V.; CASTRO, M. HPC@Cloud: A Provider-Agnostic Software Framework for Enabling HPC in Public Cloud Platforms. In: **Anais do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)**. Florianópolis, Brazil: Brazilian Computer Society, 2022. p. 157–168.

MUNHOZ, V.; CASTRO, M. Enabling the Execution of HPC Applications on Public Clouds with HPC@Cloud Toolkit. **Concurrency and Computation: Practice and Experience**, John Wiley & Sons, Hoboken, New Jersey, USA, 2023.

MUNHOZ, V.; CASTRO, M.; MENDIZABAL, O. Strategies for Fault-Tolerant Tightly-coupled HPC Workloads Running on Low-Budget Spot Cloud Infrastructures. In: **International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. Bordeaux, France: IEEE Computer Society, 2022. p. 263–272.

MUNHOZ, V.; CASTRO, M.; REGO, L. G. C. Evaluating the Parallel Simulation of Dynamics of Electrons in Molecules on AWS Spot Instances. In: **Anais do XXIV Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)**. Porto Alegre, Brazil: Sociedade Brasileira de Computação (SBC), 2023.

NETTO, M. et al. HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges. **ACM Computing Surveys**, Association for Computing Machinery, London, United Kingdom, v. 51, n. 8, p. 1–29, 2018.

PEÑA-MONFERRER, C.; MANSON-SAWKO, R.; ELISSEEV, V. HPC-Cloud Native Framework for Concurrent Simulation, Analysis and Visualization of CFD Workflows. **Future Generation Computer Systems**, Elsevier Science Publishers B.V, Amsterdam, Netherlands, v. 123, p. 14–23, 2021. ISSN 0167-739X.

RAHMAN, A.; MAHDAVI-HEZAVEH, R.; WILLIAMS, L. A Systematic Mapping Study of Infrastructure as Code Research. **Information and Software Technology**, Elsevier Science Publishers B.V, Amsterdam, Netherlands, v. 108, p. 65–77, 2019. ISSN 0950-5849.

RICHTER, H. About the Suitability of Clouds in High-Performance Computing. In: **Computer Science & Information Technology**. Mumbai, India: Academy & Industry Research Collaboration Center (AIRCC), 2016.

SANTOS, M. d.; CAVALHEIRO, G. G. H. Cloud Infrastructure for HPC Investment Analysis. **Revista de Informática Teórica e Aplicada**, Federal University of Rio Grande do Sul, Porto Alegre, Brazil, v. 27, n. 4, p. 45–62, 2020.

SENA, A. C. et al. Harnessing Low-Cost Virtual Machines on the Spot. In: **High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment**. Berlin, Heidelberg, Germany: Springer International Publishing, 2023. p. 163–189. ISBN 978-3-031-29769-4.

SHALF, J. The Future of Computing Beyond Moore's Law. **Philosophical Transactions of the Royal Society A**, The Royal Society Publishing, v. 378, n. 2166, 2020.

SHU, Y. et al. Time-Derivative Couplings for Self-Consistent Electronically Nonadiabatic Dynamics. **Journal of Chemical Theory and Computation**, American Chemical Society, v. 16, n. 7, p. 4098–4106, 2020.

SINDI, M.; WILLIAMS, J. R. Using Container Migration for HPC Workloads Resilience. In: **Proceedings of the High Performance Extreme Computing Conference**. Waltham, MA, USA: IEEE Computer Society, 2019. p. 1–10.

STORMENT, J.; FULLER, M. **Cloud FinOps**. Sebastopol, California, USA: O'Reilly Media, Inc., 2023. ISBN 978-1-492-05457-3.

TEYLO, L. et al. Scheduling Bag-of-Tasks in Clouds using Spot and Burstable Virtual Machines. **IEEE Transactions on Cloud Computing**, IEEE Computer Society, v. 11, n. 1, p. 984–996, 2023.

TORRES, A. et al. Charge Transfer Driven Structural Relaxation in a Push-Pull Azobenzene Dye-Semiconductor Complex. **The Journal of Physical Chemistry Letters**, American Chemical Society, v. 9, n. 20, p. 5926–5933, 2018.

VAILLANCOURT, P. et al. Reproducible and Portable Workflows for Scientific Computing and HPC in the Cloud. In: **Practice and Experience in Advanced Research Computing**. Portland, OR, USA: Association for Computing Machinery, 2020. p. 311–320. ISBN 978-1-450-36689-2.

WANG, C. et al. A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance. In: **Proceedings of the International Parallel and Distributed Processing Symposium**. Long Beach, California USA: IEEE Computer Society, 2007. p. 1–10.

WONG, A. K.; GOSCINSKI, A. M. A Unified Framework for the Deployment, Exposure and Access of HPC Applications as Services in Clouds. **Future Generation Computer Systems**, v. 29, n. 6, p. 1333–1344, 2013. ISSN 0167-739X.

ZHANG, J.; LU, X.; PANDA, D. K. Is Singularity-Based Container Technology Ready for Running MPI Applications on HPC Clouds? In: **Proceedings of the International Conference on Utility and Cloud Computing**. Austin, Texas, USA: Association for Computing Machinery, 2017. p. 151–160. ISBN 978-1-450-35149-2.

ZHOU, A. C. et al. FarSpot: Optimizing Monetary Cost for HPC Applications in the Cloud Spot Market. **Transactions on Parallel and Distributed Systems**, IEEE Computer Society, v. 33, n. 11, p. 2955–2967, 2022.