UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO
DEPARTAMENTO DE ESTATÍSTICA E INFORMÁTICA
PROGRAMA DE PÓS–GRADUAÇÃO EM BIOMETRIA E ESTATÍSTICA APLICADA

HENRIQUE CORREIA TORRES SANTOS

INSTRUMENTAÇÃO PARA A PARALELIZAÇÃO E DISTRIBUIÇÃO DE SIMULAÇÕES COMPUTACIONAIS SEQUENCIAIS

RECIFE – PE
2023

HENRIQUE CORREIA TORRES SANTOS

INSTRUMENTAÇÃO PARA A PARALELIZAÇÃO E DISTRIBUIÇÃO DE SIMULAÇÕES COMPUTACIONAIS SEQUENCIAIS

Tese submetida à exame de qualificação na Coordenação do Programa de Pós-Graduação em Biometria e Estatística Aplicada do Departamento de Estatística e Informática - DEINFO - Universidade Federal Rural de Pernambuco, como parte dos requisitos necessários para obtenção do grau de Doutor.

ORIENTADOR: Prof. Dr. Tiago Alessandro Espínola Ferreira

RECIFE – PE

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal Rural de Pernambuco
Sistema Integrado de Bibliotecas
Gerada automaticamente, mediante os dados fornecidos pelo(a) autor(a)

S237i Santos, Henrique Correia Torres Santos
Instrumentação para a Paralelização e Distribuição de Simulações Computacionais Sequenciais / Henrique Correia Torres Santos Santos. - 2023.
97 f. : il.

Orientador: Tiago Alessandro Espinola Ferreira.
Inclui referências e apêndice(s).

Tese (Doutorado) - Universidade Federal Rural de Pernambuco, Programa de Pós-Graduação em Biometria e Estatística Aplicada, Recife, 2023.

1. Desktop grid. 2. Programação Paralela. 3. Computação distribuída. 4. Virtualização de contêineres. I. Ferreira, Tiago Alessandro Espinola, orient. II. Título

CDD 519.5

HENRIQUE CORREIA TORRES SANTOS

INSTRUMENTAÇÃO PARA A PARALELIZAÇÃO E DISTRIBUIÇÃO DE SIMULAÇÕES COMPUTACIONAIS SEQUENCIAIS

Tese submetida à exame de qualificação na Coordenação do Programa de Pós-Graduação em Biometria e Estatística Aplicada do Departamento de Estatística e Informática - DEINFO - Universidade Federal Rural de Pernambuco, como parte dos requisitos necessários para obtenção do grau de Doutor.

Aprovada em: ___/___/_____

BANCA EXAMINADORA

Prof. Dr. Tiago Alessandro Espínola Ferreira
(Orientador)
Universidade Federal Rural de Pernambuco
Departamento de Biometria e Estatística Aplicada

Prof. Dr. Borko Stosic
Universidade Federal Rural de Pernambuco
Departamento de Biometria e Estatística Aplicada

Prof. Dr. Victor Wanderley Costa de Medeiros
Universidade Federal Rural de Pernambuco
Departamento de Informática Aplicada

Prof. Dr. Antonio Romaguera
Universidade Federal Rural de Pernambuco
Departamento de Física Aplicada

Prof. Dr. Rômulo César Carvalho de Araújo
Instituto Federal de Pernambuco

Dedico esse trabalho a minha filha que no momento em que mais precisou de forças para superar os desafios que para ela foram impostos, era ela quem me dava forças para me ajudar superar os meus.

Agradecimentos

Aos meus pais, por compreenderem todas as ausências e estarem sempre apoiando minhas empreitadas em busca de minha formação acadêmica.

À minha esposa Kate, por ter me incentivado desde o início e ter me dado o suporte necessário para seguir sempre adiante nesse processo. Por entender também minhas ausências, estresses e momentos de impaciência. Você sempre esteva ali por mim e isso eu nunca vou esquecer.

Ao professor Tiago, por acreditar que valorizar as pessoas é mais importante do que qualquer método ou processo e por me acompanhar nessa jornada mantendo sempre o foco no benefício que poderíamos trazer para os estudantes com esse trabalho.

Ao meu amigo Paulo Guedes, coordenador do curso de Análise e Desenvolvimento de Sistemas do IFPE, por ser meu suporte e apoio em todas as etapas que precisaram de iteração junto ao instituto. Esse apoio foi muito importante para eu chegar até aqui.

Aos meus amigos Eduardo e João, por estarem sempre presente nos momentos mais tensos durante as aulas e por todo o suporte que me deram também.

Aos meus amigos Augusto, Marília, Natália, Marciele, Lucas e Edvaldo por permitirem que eu estivesse junto a eles nessa troca de experiência que passamos juntos nesse período.

À minha psicologa Marta, por me ajudar a me organizar quando tudo parecia estar desmoronando.

E por último mas não menos importante, na verdade sendo a mais importante, agradeço a minha filha Bárbara, por me mostrar que podemos lutar nossas lutas de cabeça erguida, com serenidade e coragem. Aceitando o que não podemos mudar, mas lutando para melhorar naquilo que podemos. Se hoje eu consegui chegar até aqui foi porque você me mostrou o que podemos conquistar. Obrigado por me incentivar a continuar mesmo no momento em que você mais precisou de mim e por entender que a vida deve continuar, mesmo quando as eventos não parecem favoráveis para nós.

Nunca desista do que você realmente quer fazer. A pessoa com grandes sonhos é mais poderosa do que aquela com todos os fatos.

(Albert Einstein)

Resumo

A necessidade de acesso a recursos computacionais cresce à medida que o aumento da complexidade no desenvolvimento de algoritmos computacionais se tornam frequentes em diferentes setores da comunidade científica. A busca por esses recursos tem estimulado o desenvolvimento de diversas plataformas em nuvem, que abstraem a complexidade de uma infraestrutura computacional ao mesmo tempo em que oferecem aos seus usuários acesso aos recursos necessários para suas simulações. O custo de acesso a esses recursos pode limitar o perfil de usuários que podem ter acesso a eles, deixando de lado uma variedade de estudos que poderiam ser realizados em uma infraestrutura mais simples. Com o aumento da complexidade dos problemas a serem resolvidos em atividades de pesquisas, através do desenvolvimento de simulações computacionais, recursos avançados de programação paralela e distribuída se tornam um requisito para que essas simulações sejam executadas em tempo hábil. Com a lacuna existente nas ementas dos cursos de graduação em estatística abre-se espaço para o desenvolvimento de uma solução que permita uma abstração da complexidade desse tipo de programação, permitindo que códigos escritos para serem executados de forma sequencial possam ser executados de forma paralela com ajustes mínimos no código original. Nesta tese, apresentamos o *Parallel Experiment for Sequential Code* (PESC), uma plataforma para distribuição de simulações computacionais sobre computadores disponíveis em uma rede de computadores, empacotando o código do usuário em *contêiners* que abstraem toda a complexidade necessária para configurar um ambiente de execução e permitem que qualquer usuário possa se beneficiar dessa infraestrutura existente. Com um módulo *web* de fácil utilização e um módulo cliente instalado nos computadores que executarão as simulações, é possível executar simulações em diversas linguagens de programação, de *scripts* e *frameworks* (*Python, Java, C, R, PyTorch, Tensorflow*, dentre outros). Os resultados são consolidados através da página do usuário com estatísticas de tempo de execução e mapa de distribuição das simulações pelos computadores. Apresentaremos resultados obtidos em simulações que exigiram mais de 1000 execuções com diferentes parâmetros iniciais e uma variedade de outros estudos que se beneficiaram do uso do PESC.

Palavras-chave: Desktop grid, Programação Paralela, Computação distribuída, Virtualização de contêineres

Abstract

The need for access to computational resources grows as the increasing complexity in the development of computational algorithms becomes frequent in different sectors of the scientific community. The search for these resources has stimulated the development of several cloud platforms, which abstract the complexity of a computational infrastructure while offering its users access to the resources needed for their simulations. However, the cost of accessing these resources can limit the profile of users who can access them, setting aside various studies that could be carried out in a simpler infrastructure. Furthermore, with the complexity increase of the problems to be solved in research activities through the development of computer simulations, advanced features of parallel and distributed programming have become a requirement for these simulations to be executed promptly. The existing gap in the statistics undergraduate courses syllabus opens space for the development of a solution that allows an abstraction of the complexity of this type of programming, allowing written codes to be executed sequentially. Run in parallel with minimal tweaks to the original code. In this thesis, we present the Parallel Experiment for Sequential Code (PESC), a platform for distributing simulations on computers available in a network, packaging the user code in containers that abstract all complexity required to configure an execution environment and allow any user to benefit from this existing infrastructure. With an easy-to-use web module and a client module installed on the computers that will run the simulations, it is possible to run simulations in several programming languages, scripts, and frameworks (Python, Java, C, R, PyTorch, Tensorflow, among others). The results are consolidated through the user's page with runtime statistics and a distribution map of the simulations by the computers. We will present results obtained in simulations that required more than 1000 runs with different initial parameters and various other studies that benefited from using PESC.

Keywords: Desktop grid, Parallel Programing, Distributed Computing, Container Virtualization

Lista de Figuras

Lista de tabelas

Lista de algoritmos

Lista de Siglas

API	*Interface de Programação de Aplicação*
CPU	*Central Processing Unit*
CSV	*Comma Separated Value*
DRF	*Pytorch Distributed RPC Framework*
FUR	*Ranking da Folha Universitária*
GPU	*Graphical Process Unit*
HA	*Alta Disponibilidade*
HPC	*Computação de Alto Desempenho*
HTC	*Computação de Alto Rendimento*
LB	*Balanceamento de Carga*
LCG	*Geradores Congruentes Lineares*
MIMD	*Multiple Instruction Stream Multiple Data Stream*
MISD	*Multiple Instruction Stream Single Data Stream*
MNIST	*Modified National Institute of Standards and Technology*
NVML	*NVIDIA Management Library*
ORM	*Object Relational Mapping*
PESC	*Parallel Experiment for Sequential Code*
REST	*Representational State Transfer*
SIMD	*Single Instruction Stream Multiple Data Stream*
SISD	*Single Instruction Stream Single Data Stream*
VMM	*Virtual Machine Manager*
TI	*Tecnologia da Informação*

Sumário

1 Introdução

O uso de computadores como ferramenta de apoio à pesquisa científica tornou o ensino de linguagens de programação praticamente um atividade obrigatória nos cursos de pós-graduação. Escrever código e executar programas de computador deixam de ser uma atividade predominantemente relacionadas aos cursos de ciências da computação, e outros cursos correlatos, e passam a ser uma atividade comum inclusive nas áreas da saúde e humanas. Com a necessidade de abstrair complexidades relacionadas ao funcionamento interno do processo de execução dos programas, alocação de memória e tipo de dados, novas linguagens de programação e de *scripts* foram desenvolvidas e devido a essas características passaram a ser adotadas de forma mais ampla dentro deste contexto.

Atualmente existem diversos serviços *online* que disponibilizam ambientes pré-configurados e integrados onde qualquer usuário pode acessar e executar códigos em diversas linguagens de programação diferentes. Outra opção é configurar um ambiente de desenvolvimento no computador do usuário de acordo com as suas necessidades, mas devido a complexidade dos problemas que são propostos nas pesquisas científicas é necessário a adoção de recursos computacionais mais especializados e para isso estudantes recorrem aos laboratórios das suas instituições para executarem seus programas, mas esbarram na impossibilidade de configurar esses equipamentos para suas necessidades, devido ao uso compartilhado desses equipamentos por outros usuários.

Com o surgimento de novos recursos especializados, como as Unidades de Processamento Gráfico (GPUS), problemas mais complexos que exigem uma demanda computacional elevada passaram a ser resolvidos em ambiente local através de computadores do tipo *desktops*. Escrever códigos que façam uso desses recursos deixa de ser uma atividade trivial, requerendo uma avaliação entre o ganho de tempo de execução *versus* o tempo necessário para aprender os novos recursos computacionais. Com a popularidade desses novos recursos computacionais, novos arcabouços de software (*frameworks*) foram desenvolvidos criando uma abstração da complexidade computacional, permitindo que sua adoção aconteça de forma mais rápida e fácil pelos pesquisadores.

A dificuldade em configurar os equipamentos disponíveis para necessidades específicas de cada simulação ainda continua a ser um problema devido a quantidade de linguagens de programação, *frameworks* e bibliotecas existentes, onde cada uma pode ter

uma variedade de dependências que podem apresentar conflito entre diferentes versões.

Com a criação de serviços de nuvens públicas, como a Amazon AWS (AMAZON, 2022), Microsoft Azure (MICROSOFT, 2022) e Google Cloud (GOOGLE, 2022), infraestruturas locais foram gradualmente sendo substituídas por soluções nessas plataformas (CHOI et al., 2007) ou foram estendidas com o uso desses recursos (KRAŠOVEC; FILIPČIČ, 2019). Um dos motivos para essa migração está relacionada ao custo de adquirir e manter os equipamentos de uma infraestrutura local, além da preparação das equipes de Tecnologia da Informação (TI) responsáveis pelas devidas manutenções (TAYLOR et al., 2020).

Apesar desses serviços de nuvem pública oferecerem planos gratuitos eles não favorecem a execução de simulações demoradas ou que necessitam de recursos especializados (COLAB, 2022). Dessa forma, o acesso aos recursos oferecidos por essas plataformas pode não ser viável às instituições educacionais públicas, principalmente em países em desenvolvimento, devido à burocracia nas contratações e à falta de investimento em pesquisa e desenvolvimento para contratações desse tipo de serviço e infraestrutura (NJENGA et al., 2019).

1.1 Delimitação do Tema

Uma forma de tornar programas de computador factíveis para a resolução de problemas de alta complexidade é identificar pontos que possibilitem a sua execução de forma paralela. Paralelizar um código permite que recursos como o processador de um computador possa ser utilizado de forma mais otimizada, ou mesmo que vários computadores sejam utilizados de forma simultânea para executar o mesmo programa. Nesse caso, cada parte paralela será distribuída para um computador diferente que a executará e devolverá o resultado ao final do processo. Quando vários computadores são usados em um processo de programação paralela, como definido anteriormente, está sendo configurado um *grid* computacional e nesse trabalho será apresentado a construção de uma solução para essa finalidade. Onde a complexidade da configuração dos equipamentos e a distribuição do código será abstraído através de uma interface simples e intuitiva para que qualquer usuário, independente de formação, possa se beneficiar do resultado apresentado por este trabalho.

1.2 Problema de Pesquisa

Apesar de possível não é simples paralelizar programas de computador que foram originalmente escritos para serem executados de forma sequencial (BHALLA, 2014). Vários formas de resolver esse problema foram desenvolvidos e alguns deles são apresentados no Capítulo 3, mas por apresentarem um foco maior no problema técnico do funcionamento da solução pode existir a necessidade do usuário realizar adaptações em seu código para que ele funcione nessas soluções. Essas alterações nem sempre são adaptações simples, e em alguns casos pode ser necessário reescrever o código totalmente. Dessa forma é apresentada uma solução que facilite sua adoção sem a necessidade de realizar alterações que o usuário não considere apropriadas e não tire a sua liberdade de executar o código no ambiente que achar apropriado.

1.3 Objetivos

A seguir serão apresentados os objetivos geral e específicos que nortearão a condução desta pesquisa.

1.3.1 Objetivo Geral

A presente tese tem como objetivo geral apresentar uma forma de viabilizar o uso de recursos computacionais ociosos, como computadores de um laboratório ou de uma sala de aula, através do desenvolvimento de uma plataforma usada para gerenciar a distribuição da carga de execução de códigos de computador, centrada nas necessidades de um usuário sem formação específica, e não um usuário de ciências da computação ou áreas correlatas.

o desenvolvimento de uma plataforma usada para gerenciar a distribuição da carga de execução de códigos de computador em uma infraestrutura preexistente, como um laboratório ou uma sala de aula, através de uma solução centrada nas necessidades de um usuário sem formação específica, e não um usuário de ciências da computação ou áreas correlatas.

1.3.2 Objetivo Específico

Para atingir o objetivo geral foram definidos os seguintes objetivos específicos:

- Identificar os recursos necessários para a execução do código do usuário em ambiente remoto;
- Configurar uma solução de *contêiner* para a execução do código do usuário em qualquer computador;
- Propor uma Interface de Programação de Aplicação (API) de comunicação entre o servidor e os clientes que permita acompanhar e gerenciar o fluxo de execução;
- Propor um ambiente integrado e intuitivo para o usuário definir os ambientes de execução de seus código e gerenciar e acompanhar os processo de execução solicitados;
- Implementar a solução proposta.

1.4 Justificativa

O uso de recursos ociosos em uma instituição de ensino/pesquisa representa um aproveitamento mais efetivos do investimento realizada para a sua aquisição. Adotar tecnologias complexas pode representar uma baixa adoção por parte da comunidade acadêmica e o desenvolvimento dessa plataforma é no sentido de maximizar a sua adoção no apoio à pesquisa.

1.5 Estrutura do Trabalho

Este trabalho está estruturado da seguinte forma. No Capítulo 2 é apresentado o referencial teórico necessário para o acompanhamento das informações sobre o desenvolvimento da solução proposta e descrevemos soluções computacionais existentes para a distribuição de execução paralela. A seguir, no Capítulo 3 serão apresentados trabalhos identificados na literatura que abordam problemas similares ao apresentado neste trabalho. A seguir, no Capítulo 4 uma visão geral da plataforma, a sua arquitetura e como os seus componentes se comunicam é apresentado. No Capítulo 5 avaliamos a plataforma através de testes controlados e validamos através de casos de uso reais que usaram a plataforma e se beneficiaram da proposta apresentada. Finalmente, o Capítulo 6 conclui o trabalho apresentado.

2 Referencial Teórico

Programas de computador são escritos em várias linguagens de programação diferentes e, do ponto de vista do usuário, o fluxo de execução é um processo relativamente simples, onde dados são apresentados como uma entrada para o programa, que é executado pelo processador (CPU) e fornece um novo conjunto de dados como saída após a conclusão da execução, esse processo é apresentado na Figura 1.

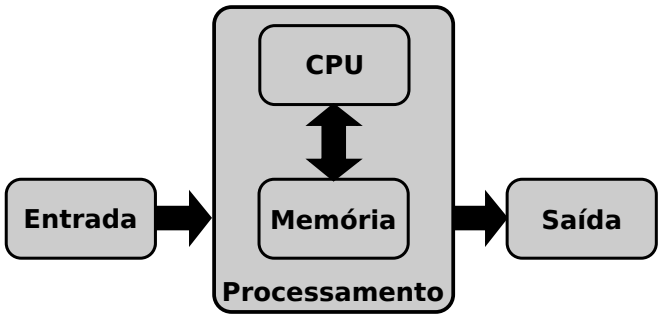


Figura 1 – Fluxo de execução (o autor)

O aumento da complexidade do problema a ser resolvido requisita cada vez mais códigos que provavelmente exigirão mais recursos computacionais para serem executados sem refletir em um aumento expressivo no tempo de execução. Dessa forma, processadores, memórias, discos e outros componentes evoluem para atender a alta demanda por recursos computacionais desses programas.

Em 1965 Gordon Moore fez uma observação com relação a quantidade de transistores em um circuito. Moore afirmou que a quantidade de transistores dobraria a cada dezoito meses, Figura 2, impactando diretamente na capacidade e velocidade de execução dos programas pelos computadores. Mas partindo do princípio que não é possível aumentar indefinidamente a velocidade do processador e a frequência das memórias, novas arquiteturas computacionais foram desenvolvidas para contornar essa limitação e diminuir o tempo de execução de programas complexos.

2.1 Computação Concorrente

Uma das formas de reduzir o tempo de execução de programas complexos é através da divisão do programa em partes menores que estarão em execução ao mesmo tempo.

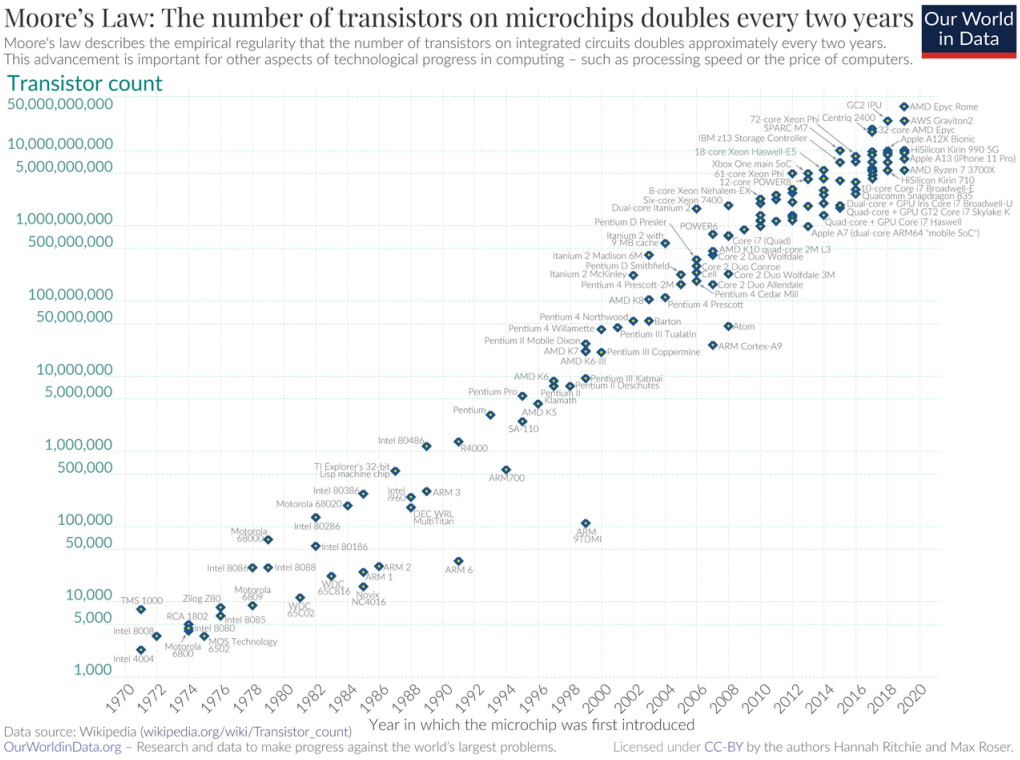
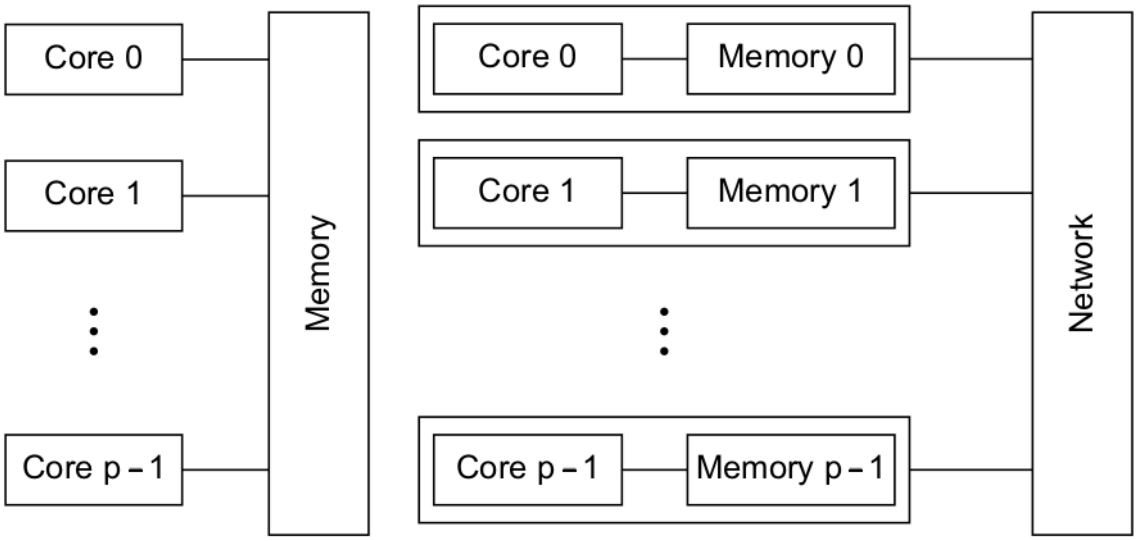


Figura 2 – Lei de Moore (ROSER; RITCHIE, 2022)

Dependendo da forma como cada parte é executada esse processo pode ser classificado da seguinte forma (PACHECO; MALENSEK, 2011):

- Concorrente: É a capacidade de ter mais de um processo em execução em andamento ao mesmo tempo. Essas execuções podem estar sendo realizadas em um ou mais núcleos de processamento diferentes de um mesmo processador.
- Paralelo: Estar executando mais de um processo ao mesmo tempo que devem estar sendo realizadas em núcleos de processamento diferentes, Figura 3(a).
- Distribuído: É um conjunto de processadores conectados por meio de uma rede onde cada processador tem uma memória anexada e acessível apenas por esse processador. Os dados podem ser compartilhados entre os diferentes processadores através de funções de API projetadas para esse tipo de comunicação, Figura 3(b).

Apesar da expressão "programação paralela" ser usada de forma genérica, é possível perceber que ser paralelo nesse contexto é um tipo de concorrência, sendo esse o termo mais geral para esse tipo de execução. A concorrência permite que as partes do processo estejam em execução ao mesmo tempo, mesmo em processadores com um único núcleo de execução, não sendo possível nesse caso ser uma execução paralela. Para isso o sistema operacional faz a troca do processo em execução em intervalos de tempo e essa troca é



(a) Shared memory (b) Distributed memory

Figura 3 – Tipos de sistemas paralelos (PACHECO; MALENSEK, 2011)

chamada de multitarefa preemptiva (ECKERT et al., 2016), Figura 4. Ao final do processo, para o usuário, os processos foram executados ao mesmo tempo dando a impressão que a execução aconteceu de forma paralela. Mas diferente dos programas concorrentes, os programas paralelos e distribuídos podem ter tarefas que estejam realmente em execução de forma simultânea.

Segundo (PACHECO; MALENSEK, 2011), alguns autores classificam programas paralelos como sendo do tipo *shared-memory* e programas distribuídos como *distributed-memory*, embora não exista consenso geral sobre estes termos. Nos programas classificados como *shared-memory* as instâncias das tarefas em execução são chamadas de *threads* e por compartilharem o mesmo *hardware* durante a execução são definidos como fortemente acoplados, nos programas classificados como *distributed-memory* as instâncias são chamadas de processos e por poderem estar em execução em computadores geograficamente separados são definidos como fracamente acoplados.

2.2 Programação Paralela

Atualmente a maioria dos computadores possuem processadores com vários núcleos de processamento e linhas de execução paralelas. Dessa forma, a execução paralela de um programa depende mais da forma como foi construído do que ter uma infraestrutura

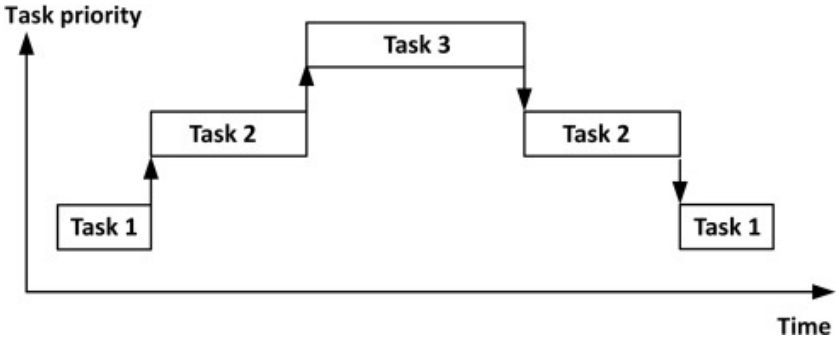


Figura 4 – Multitarefa Preemptiva (IBRAHIM, 2020)

específica para que a sua execução aconteça de forma paralela. Por isso cabe ao programador fazer uso dos recursos da linguagem de programação para aproveitar os recursos disponíveis, como apresentado no exemplo a seguir.

No Algoritmo 2.1 é apresentado um código na linguagem de programação *Python* que executa uma função, que verifica se um número é primo ou não, de forma sequencial. Essa função pode ter seu tempo de execução acentuado dependendo do valor recebido como parâmetro por fazer um laço nessa verificação. Como apresentado na Figura 5 a execução sequencial desse programa envolve apenas a participação de um único núcleo de processamento, o núcleo 3.

```python
# sequencial.py
import time, math

def verificar_primo(n):
    if n <= 1 : return False
    if n == 2: return True
    if n > 2 and n % 2 == 0: return False
    for i in range(3, math.ceil(math.sqrt(n)) + 1, 2):
        if n % i == 0: return False
    return True

if __name__ == "__main__":
    valores = list(range(3, 100000000))
    inicio = time.time()
    for valor in valores:
        verificar_primo(valor)  # Operação demorada
    print(f"Tempo: {time.time()-inicio} segundos")
```

Algoritmo 2.1 – Exemplo de código sequencial em Python

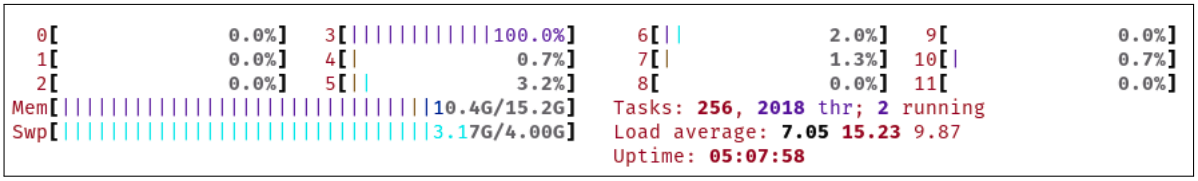


Figura 5 – Núcleos de processamento em uso com a execução sequencial (o autor)

O mesmo problema foi modificado para distribuir a carga de execução com os outros núcleos de processamento existentes, a modificação faz uso dos recursos da linguagem *Python* e está apresentada no Algoritmo 2.2. Nesse exemplo a carga de execução será divida em seis (6) processos que serão executados de forma paralela e como apresentado na Figura 6 é possível perceber a participação de outros núcleos de processamento na execução desse código. Dessa forma, é possível perceber que apenas usando recursos da linguagem foi possível obter um ganho de aproximadamente 300% no tempo de execução desse código, os tempos de execução estão apresentados na Tabela 1.

```python
# paralelo.py
import time, math
from multiprocessing import Pool

def verificar_primo(n):
    # mesma função do exemplo sequencial
    ...

if __name__ == "__main__":
    valores = list(range(3, 100000000))
    inicio = time.time()
    with Pool(processes=6) as pool:          # Criados 6 processos
        pool.map(verificar_primo, valores)  # Operação demorada
    print(f"Tempo: {time.time()-inicio} segundos")
```

Algoritmo 2.2 – Exemplo de código paralelo em Python

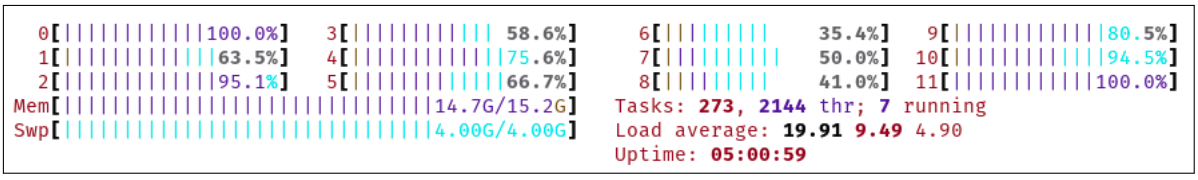


Figura 6 – Núcleos de processamento em uso com a chamada paralela (o autor)

Programa	Tempo em segundos
sequencial.py	724.39
paralelo.py	240.70

Tabela 1 – Tempo de execução do código sequencial e paralelo

2.2.1 Classificação de sistemas paralelos

Paralelismo pode ser classificado de várias formas diferentes quando está relacionado ao hardware ou ao software. Com relação ao hardware, uma forma de classificar sistemas paralelos é através da taxonomia de Flynn (FLYNN, 1966) que leva em consideração o conceito de fluxo de instruções e de dados. O fluxo de instruções é uma sequência de instruções que serão executadas em um processador, já o fluxo de dados é uma sequência de dados necessários às instruções em execução no processador. Dessa forma, Flynn definiu quatro possíveis combinações que são descritas a seguir.

- SISD - *Single Instruction Stream Single Data Stream* (Um fluxo de instrução e um fluxo de dados): Computação sequencial;
- SIMD - *Single Instruction Stream Multiple Data Stream* (Um fluxo de instrução e múltiplos fluxos de dados): Computação em GPU;
- MISD - *Multiple Instruction Stream Single Data Stream* (Múltiplos fluxos de instruções e um fluxo de dados): É um modelo teórico sem implementação prática;
- MIMD - *Multiple Instruction Stream Multiple Data Stream* (Múltiplos fluxos de instruções e múltiplos fluxos de dados): Maioria dos processadores atuais, que apresentam vários núcleos;

Com relação ao software (KUMAR et al., 2003), paralelismo é classificada como implícita e explicita. A classificação implícita ocorre internamente no processador do computador, independente de comandos ou controles no programa do usuário. Já a classificação explicita acontece quando o programador determina em seu código que um certo conjunto de instruções deve ser executado de forma paralela, como foi mostrado no Algoritmo 2.2.

2.2.2 Conversão de código sequencial em paralelo

Um código sequencial pode ser convertido para uma versão que seja executado de forma concorrente. Apesar desta operação ser possível, não é simples paralelizar programas originalmente escritos para serem executados de forma sequencial (BHALLA, 2014). Existem diversas técnicas para realizar essa conversão, mas em (BRESHEARS, 2009) são apresentadas duas formas que normalmente são abordadas e estão listadas a seguir.

- Decomposição de tarefas
- Decomposição de dados

Apesar dessas duas técnicas abordarem parte do código que pode ser paralelizado, ainda segundo (BRESHEARS, 2009), existem outras partes que não podem ser paralelizadas.

- Algoritmos com Estado
- Recorrências
- Variáveis de Indução
- Redução
- Dependência Carregada por Loop

Dessa forma converter um programa sequencial para uma versão paralela pode demandar uma grande quantidade de alterações e o resultado final ser praticamente um programa novo, reescrito com recursos avançados da linguagem de programação usada em sua construção.

2.2.3 Lei de Amdahl

Um programa de computador escrito originalmente para ser executado de forma sequencial pode ser revisado e reescrito para ter parte de seu código paralelizado. Isso acontece porque parte do código não pode ser paralelizado e precisa manter a execução sequencial como em uma leitura de dados, de um arquivo ou na definição de variáveis. A lei de Amdahl demonstra o ganho em velocidade de execução, *speedup*, que pode ocorrer com a paralelização de parte de um programa de computador (AMDAHL, 1967). Ela diz que a otimização de uma única parte de um sistema impacta na melhoria geral de desempenho, enquanto essa parte estiver em execução, e dessa forma não existe uma relação linear entre

o aumento no número de unidades de processamento e o ganho de velocidade na execução do programa, como mostrado na Figura 7.

Para calcular o *speedup* obtido entre uma versão paralela e uma versão sequencial de um mesmo código pode ser usado a expressão:

$$S_{obtido} = \frac{T_{sequencial}}{T_{paralelo}}$$

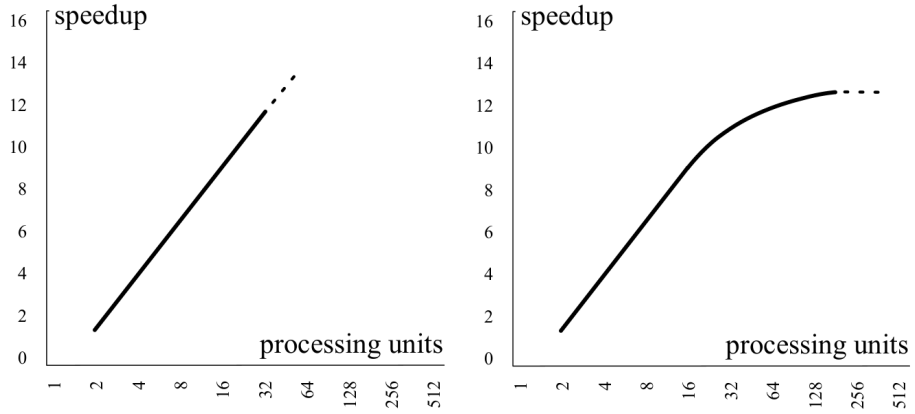
Onde S_{obtido} é o *speedup* obtido, $T_{sequencial}$ é o tempo de execução da versão sequencial e $T_{paralelo}$ e o tempo de execução da versão paralela, mas através da Lei de Amdahl é possível calcular o *speedup* estimado atavés da equação (PACHECO; MALENSEK, 2011):

$$T_{paralelo\_esperado} = (1 - f) * \frac{T_{sequencial}}{N} + f * T_{serial}$$

Sendo f a fração do código que não pode ser paralelizadao devido ao tipo de rotina executada e $1 - f$ a fração do código que será executada de forma parelela e N o número de processos paralelos que serão utilizados na execução da parte paralela do código, temos,

$$S_{esperado} = \frac{T_{sequencial}}{T_{paralelo\_esperado}}$$

$$S_{esperado} = \frac{T_{sequencial}}{(1 - f) * \frac{T_{sequencial}}{N} + f * T_{sequencial}}$$



(a) Expectativa (b) Ganho real

Figura 7 – Expectativa e ganho real de velocidade com o aumento de unidades de processamento (TROBEC et al., 2020)

2.3 Computação Distribuída

Como apresentado na Seção 2.1 a computação distribuída pode ser definida como um conjunto de processadores, cada um com uma memória própria e não compartilhada, que se comunicam através da troca de mensagens por uma rede de computadores (BAL et al., 1989). A finalidade dessa arquitetura é resolver um conjunto de problemas que não podem ser resolvidos por um único computador e dependendo da forma como esses computadores estão organizados ela pode ser classificada como computação *peer-to-peer*, *cluster*, utilitária e *jungle*, como apresentado na Figura 8. Devido a natureza desse trabalho que explora a distribuição da carga de trabalho em computadores de uma infraestrutura local a computação em *cluster* e *grid* serão detalhadas a seguir.

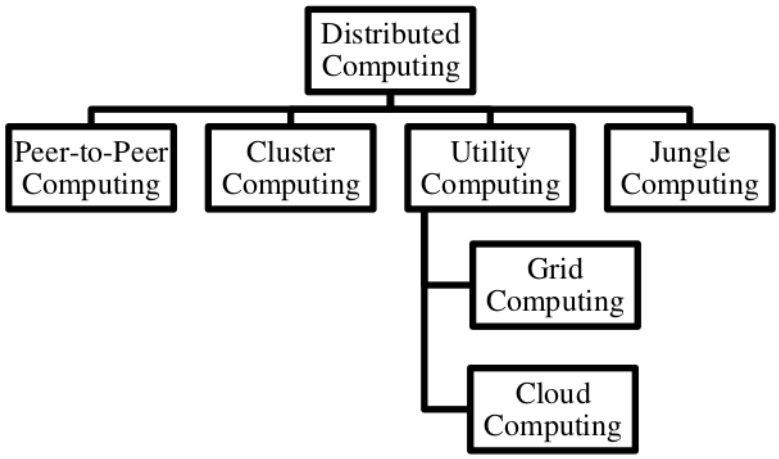


Figura 8 – Classificação de Computação Distribuída (KAHANWAL et al., 2013)

2.3.1 Computação em Cluster

Um *cluster* é um tipo de sistema de computação distribuída que abstrai a existência de um conjunto de computadores autônomos conectados por um rede local de alta velocidade. Esses computadores são apresentados ao usuário ou outras aplicações como um único sistema e podem funcionar de forma dedicada ao *cluster* ou podem ser adicionados e removidos de forma dinâmica, quando forem necessários para outra finalidade.

Os *clusters* foram uma das primeiras abordagens para resolver o problema da necessidade de alto desempenho na execução de algoritmos complexos. Usado tanto em problemas como a previsão do clima, na industria, na medicina e na astronomia os *clusters*,

ou computação de alto desempenho (HPC), definiram uma nova abordagem na resolução de problemas complexos.

Apesar de estarem associados à computação de alto desempenho os *cluster* podem ser classificados também em *clusters* de alta disponibilidade (HA) e *clusters* de balanceamento de carga (LB). Enquanto os HA são usados para garantir que serviços críticos não fiquem indisponível por alguma falha no computador principal os LB são usados para distribuir uma carga de acesso a um determinado serviço pelos computadores do *cluster*.

Apesar da arquitetura da plataforma PESC não ser classificada como um *cluster* é possível manter um conjunto de computadores dedicados a ela e dessa forma classificá-la como um *clusters* de computação de alto desempenho (HPC).

2.3.2 Computação em Grid

Um *grid* computacional é uma infraestrutura composta pela seleção e agregação de diversos recursos computacionais, que podem estar geograficamente distantes, permitindo o compartilhamento de recursos não utilizados através de uma rede de computadores (SINGH, 2019).

O acesso a esses recursos computacionais distribuídos permite que tarefas complexas sejam dividas em partes que serão distribuídas pelos equipamentos configurados no *grid*. Segundo (MISHRA et al., 2017) os *grids* podem ser classificados baseados em:

- Tipo de Aplicação: Comercial, Pesquisa Científica, Acadêmica, Ciências da Saúde, Público e Híbrido;
- Tipo de Funcionalidade: Computação Intensiva, Uso intensivo de dados, Utilitário, Auto-organizado e Tempo real;
- Escala: Pequeno, Médio e Grande;
- Escopo: Global, Internacional, Nacional, Regional e Local

Os *grids* de Computação Intensiva estão diretamente relacionados ao compartilhamento de recursos de *CPU* para o processamento de problemas complexos e podem ser classificados como: *Desktops*, Servidores, Equipamentos ou Instrumentos Eletrônicos, Alto Rendimento e Supercomputação Distribuída. Devido a essa flexibilidade vários desafios foram identificados por (SINGH, 2019) na computação em *grid* e entre elas podemos destacar:

- Por fazer uso de recursos heterogêneos não existe um padrão específico para todas as estruturas que fazem parte do grid;
- Definir um agendamento de tarefas eficiente que identifique os recursos ideais para a execução das tarefas;
- Um mecanismo de tolerância a falhas que permita a execução das tarefas distribuídas no grid;

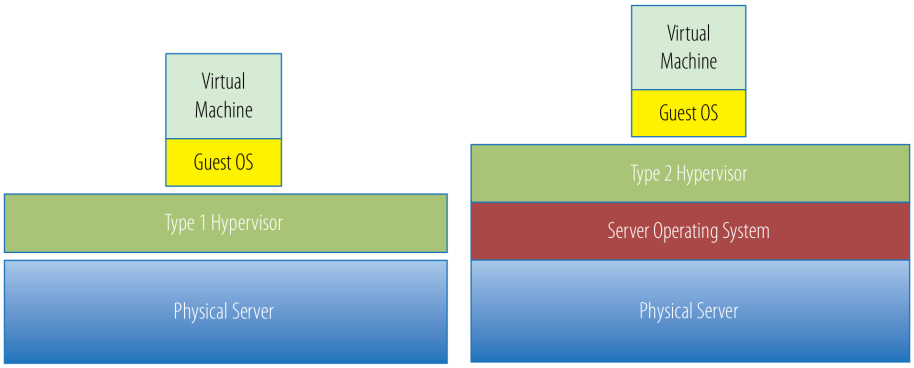
Por permitir o compartilhamento de recursos ociosos de computadores disponíveis em uma rede local a plataforma PESC pode ser definida como um *grid* computacional de computação intensiva, formado por computadores desktops, de pequena escala e escopo local.

2.4 Virtualização

O conceito de virtualização vem da necessidade de executar sistemas que estão sendo desenvolvidos para um determinado equipamento que ainda não está disponível para uso e uma simulação desse equipamento pode ser desenvolvida para a execução desses sistemas (GOLDBERG, 1974). Dessa forma, um equipamento virtual estará disponível para o usuário do computador como qualquer outro sistema instalado no sistema operacional. Quando o equipamento a ser simulado é um computador e não um *hardware* arbitrário, a virtualização passa a ser um processo que permite que um computador físico seja particionado em várias máquinas virtuais. Esse particionamento pode ser realizado no nível do *hardware* e/ou no nível do *software* (BUYYA et al., 2013).

2.4.1 Hypervisor

Na virtualização em nível do *hardware* é fornecido, para as máquinas virtuais que estão em execução no computador, uma camada de abstração criada pelo *Virtual Machine Manager* (VMM) em relação ao hardware. O VMM ou *hypervisor* está classificado em Tipo 1 e Tipo 2, como apresentados na Figura 9. O *hypervisor* Tipo 1 é executado diretamente no *hardware*, sem a necessidade de um sistema operacional instalado no computador e no Tipo 2 o *hypervisor* é instalado no sistema operacional do computador para prover o serviço de virtualização. Independente do tipo de *hypervisor* o *hardware* do



(a) Tipo 1 (b) Tipo 2

Figura 9 – Tipos de hypervisor (PORTNOY, 2016)

computador é apresentado para a máquina virtual onde um sistema operacional completo será configurado e executado, podendo inclusive ser diferente do sistema operacional instalado no computador onde está instalado o *hypervisor*.

2.4.2 Contêiner

Na virtualização em nível do *software* não existe o papel do *hypervisor* ou de máquinas virtuais, mas a configuração de diferentes ambientes de execução isolados um dos outros. Esses ambientes de execução são criados a partir do mesmo sistema operacional, com o compartilhamento do seu *kernel* entre os ambientes de execução que são chamados de contêineres. Por não representar um máquina virtual com um sistema operacional completo, mas que pode ser configurado para executar uma aplicação específica, a virtualização em nível de software é considerada uma opção mais leve de virtualização e dessa forma sendo uma opção viável para configuração de ambientes de execução em qualquer tipo de computador e sistema operacional.

Na Figura 10a é apresentado o modelo de computação baseado na execução de aplicações no sistema operacional instalado no computador físico, sem o uso de recursos de virtualização. Na Figura 10b o modelo de virtualização em nível do hardware através de um *hypervisor* é utilizado, onde as aplicações são instaladas no sistema operacional das máquinas virtuais e na Figura 10c as dependência necessárias para executar uma aplicação são configuradas em contêineres que compartilham recursos do sistema operacional da computador físico, mas não confiltam entre si.

Apesar de existirem várias soluções de virtualização por contêineres, como o Podman,

LXD e o Containerd, o Docker passa a ser uma opção popular por se tornar um padrão no desenvolvimento de software (CITO et al., 2017) será e detalhado a seguir.

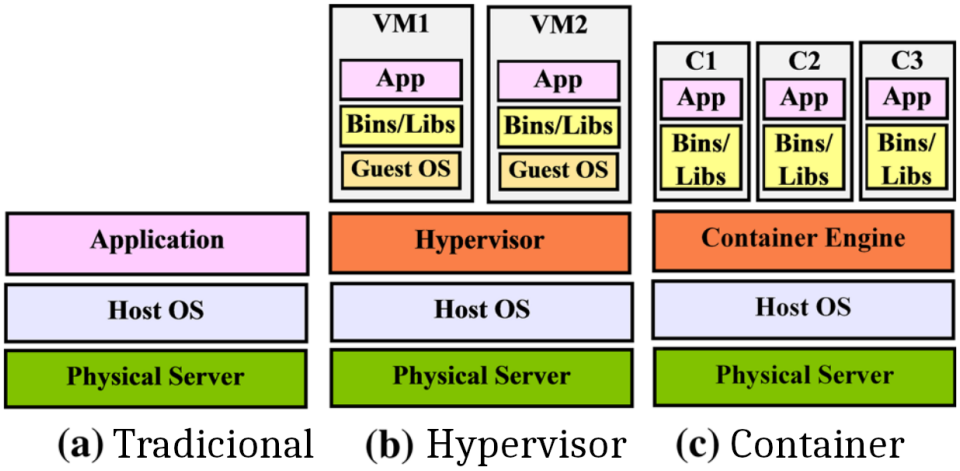


Figura 10 – Comparação entre um ambiente não virtualizado (a), virtualizado no nível do Hardware (b) e no nível do Software (c) (BHARDWAJ; KRISHNA, 2021)

2.4.2.1 Docker

O Docker[1] é um projeto de código aberto que permite o empacotamento de aplicações e serviços atráves de virtualização baseada em contêineres. A arquitetura do Docker é baseada no modelo cliente-servidor e é composta por um conjunto de compontes que estão listados a seguir:

- Docker Daemon: Componente que gerencia as imagens, contêineres, redes e volumes. É o responsável por todo o ciclo de vida na execução de contêineres;
- Docker Client: É uma ferramenta de linha de comando usada para interagir com o *daemon*. Normalmente ela é instalada na mesma máquina que o *daemon*;
- Docker Desktop: É uma aplicação instalada no sistema operacional que permite de forma gráfica gerenciar e interagir com todo o ecossitema criado pelo Docker;
- Docker Registry: É um repositório onde as imagens que são usadas para a criação dos contêineres são armazenadas. O Docker Hub é um *registry* público mantido pela Docker e que o *daemon* usa como padrão na criação de uma imagem;
- Docker Engine: É a combinação do Docker Daemon e do Docker Client.

A criação de contêineres no Docker é realizado a partir de uma imagem, que pode

[1] https://www.docker.com/

estar disponível em um *registry*, ou descrita em um arquivo Dockerfile na mesma máquina onde está instalado o *daemon*. O arquivo Dockerfile contém um conjunto de instruções que será executado durante o processo de construção da imagem, esse conjunto de instruções é formado pelos mesmos comandos que seriam executados em um sistema operacional para preparar o ambiente de execução para a aplicação ou serviço que está sendo empacotado no contêiner. A Figura 11 mostra um exemplo de um arquivo Dockerfile que prepara um ambiente de execução para programas desenvolvidos na linguagem de programação *R* com suporte a liguagem de programa *Python*.

```
FROM r-base 4.1.2

WORKDIR /app

RUN   apt-get update && \
      apt-get upgrade -y && \
      apt-get install -q -y python3 python3-pip nano

RUN pip3 install --upgrade pip

COPY requirements.txt ./

RUN ln -s /usr/bin/python3 /usr/bin/python

RUN pip install --no-cache-dir -r requirements.txt
```

Figura 11 – Exemplo de um Dockerfile (o autor)

A imagem é um objeto imutável e a partir dela os contêineres são criados como uma instância dessa imagem, mas com uma área para a persistência de arquivos separadas, tendo seu ambiente de execução isolado de outros contêineres e do sistema operacional hospedeiro.

Por permitir que apenas o Docker Engine seja instalado em um sistema operacional facilitou a sua adoção pela comunidade no desenvolvimento de software. Os ambientes de desenvolvimento passaram a ter uma proximidade maior com o ambiente de produção, onde os sistemas serão efetivamente disponibilizados, mitigando uma série de problemas com versões de bibliotecas das dependências necessários para cada ambiente.

2.4.3 Orquestração de Contêineres

A utilização de contêineres simplificou a configuração do ambiente de desenvolvimento, mas para o ambiente de produção, em nuvem ou *on-premise*, que

deve compartilhar seus recursos físicos (Discos, processor, memória, etc.) com todos os contêineres implantados foram desenvolvidas ferramentas para a automatização da implantação e o gerenciamento desses contêineres. Essas ferramentas fazem a orquestração de contêineres que automatizam tarefas como:

- Deploy: Implantar os contêineres em ambiente de produção;
- Escalonamento: Provisionamento de infrastrutura necessária para a execução dos contêineres;
- Acesso: Definição de um único de ponto de acesso ao contêiner, independente da quantidade de cópias criada pelo escalonamento;
- Recuperação de falhas: Distribuição do contêiner para outra infraestrutura em caso de falha na atual.

Apesar de existirem várias ferramentas para a orchestração de contêineres, como o Docker Swarm[2] e o Kubernetes[3], elas não foram usadas na solução proposta por este trabalho por adicionarem uma camada de complexidade na arquitetura do sistema que impactaria na sua implantação e manutenção para as equipes de TI. Além disso elas estão mais alinhadas com o gerenciamento de ambientes de produção para serviços com alta demanda de acesso, como servidores *web*, de aplicação e banco de dados. Apesar disso elas forneceram uma base para a criação do mecanismo que faz uma orquestração mais simples, baseada apenas em necessidades específicas simplificando a arquitetura da solução apresentada e sua manutenção.

2.5 Ensino de Programação Paralela em Estatística

Apesar dos ganhos de eficiência mostrados na seção anterior, a programação paralela é um conteúdo que está normalmente inserido na grade curricular dos cursos de ciências da computação ou correlatos. É fundamental para entender a demanda pelo tipo de ferramenta desenvolvida nesse trabalho como as universidades abordam o ensino de programação paralela nos cursos de estatística.

Nessa seção analisamos os currículos dos cursos de estatística das instituições de ensino superior no Brasil, independente de serem públicas ou privadas, com o foco no ensino de programação paralela de computadores. As 24 universidades analisadas,

[2] https://docs.docker.com/engine/swarm/
[3] https://kubernetes.io/

foram escolhidas de acordo com o sistema de pontuação disponível no Guia da Faculdade (Estadão, 2021)[4] e que apresentaram *ranking* mínimo de 4 estrelas (BOM) de um total de 5 estrelas (ÓTIMO). O Guia da Faculdade foi escolhido por permitir uma seleção por curso e realizar uma avaliação por pares como metodologia para avaliação dos cursos.

Também foi levado em consideração o Ranking da Folha Universitária (FUR) (Folha de São Paulo, 2019)[5] que avalia as melhores universidades do país, o *QS World University Rankings* da *QS Quacquarelli Symonds* para a área de "estatística e pesquisa operacional"[6] e o *Times Higher Education Latin America University Rankings*[7] para universidades que oferecem cursos na área de "matemática e estatística" para que ementas de universidades bem avaliadas de forma geral, não apenas pelo curso de estatística, também participassem da análise. Os resultados estão compilados na Tabela 2, onde N/A indica que a universidade não foi listada no resultado da instituição avaliadora.

Tabela 2 – Universidades avaliadas

Sigla	**Instituição**	**Estadão**	**Folha**	**Top Universities**	**Times Higher Education**
UNICAMP	Universidade Estadual de Campinas	1	2	2	2
UFOP	Universidade Federal de Ouro Preto	2	N/A	N/A	N/A
UFPE	Universidade Federal de Pernambuco	3	10	5	N/A
ENCE	Escola Nacional de Ciências Estatísticas	4	N/A	N/A	N/A
UFRJ	Universidade Federal do Rio de Janeiro	5	3	3	7
UEM	Universidade Estadual de Maringá	6	24	N/A	N/A
UFSM	Universidade Federal de Santa Maria	7	21	N/A	N/A

Continua na próxima página

[4] <https://publicacoes.estadao.com.br/guia-da-faculdade/?post_type=faculdades_2021&ano=2021&s=estat%C3%ADstica&tipo=&modalidade=Presencial&estado=&cidade=&classificacao=>

[5] <https://ruf.folha.uol.com.br/2019/ranking-de-universidades/principal/>

[6] <https://www.topuniversities.com/university-rankings/university-subject-rankings/2022/statistics-operational-research>

[7] <https://www.timeshighereducation.com/world-university-rankings/2021/latin-america-university-rankings>

Continuação da página anterior da Tabela 2

Sigla	Instituição	Estadão	Folha	Top Universities	Times Higher Education
UNIR	Fundação Universidade Federal de Rondônia	8	N/A	N/A	N/A
UNESP	Universidade Estadual Paulista Júlio de Mesquita Filho	9	6	N/A	N/A
UFSCAR	Universidade Federal de São Carlos	10	12	N/A	N/A
UFBA	Universidade Federal da Bahia	12	14	N/A	N/A
UFF	Universidade Federal Fluminense	13	17	N/A	N/A
UFU	Universidade Federal de Uberlândia	14	N/A	N/A	N/A
UNB	Universidade de Brasília	15	9	N/A	N/A
UFAM	Universidade Federal do Amazonas	16	N/A	N/A	N/A
USP	Universidade de São Paulo	17	1	1	1
UFG	Universidade Federal de Goiás	18	20	N/A	N/A
UFC	Universidade Federal do Ceará	19	11	N/A	N/A
UFES	Universidade Federal do Espírito Santo	20	N/A	N/A	N/A
UFMT	Universidade Federal de Mato Grosso	21	N/A	N/A	N/A
UFPA	Universidade Federal do Pará	22	N/A	N/A	N/A
UFMG	Universidade Federal de Minas Gerais	23	4	4	3
UFPR	Universidade Federal do Paraná	24	8	N/A	N/A
UFRN	Universidade Federal do Rio Grande do Norte	N/A	22	N/A	N/A

O material usado para a coleta de dados estão disponíveis nos sites das instituições de ensino e foram descartadas aquelas que não disponibilizam a ementa de cada componente curricular. Das universidades avaliadas apenas a UNICAMP, ENCE, UFPE e a UFSCAR mencionam computação paralela no conteúdo programático de algum componente curricular, ou seja, apenas $16,67\%$ da instituições ranqueadas como "BOM" ou "ÓTIMO". A Tabela 3 apresenta a quantidade de cursos de estatística distribuídos por região e desses cursos quantos apresentaram uma referência a programação paralela em sua ementa.

Região	Qtd de Cursos	Qtd Encontrada
Centro-Oeste	9	0
Nordeste	20	1
Norte	13	0
Sudeste	74	3
Sul	21	0

Tabela 3 – Distribuição dos cursos por região

De acordo com os dados levantados, o ensino de programação paralela não está inserido nos currículos dos cursos de graduação de estatística, área que naturalmente trabalha com experimentos estocásticos. Os experimentos estocásticos tendem a gerar resultados diferentes, mesmo se executados nas mesmas condições, e precisam ser executados um certo número de vezes para garantir que estatísticas significativas possam ser calculadas (BROWNLEE, 2018). Usar programação paralela nesses casos pode gerar uma redução no tempo total de execução, mas adotar esse tipo de programação exige uma base de conhecimentos computacionais que não estão inseridos de forma institucional na graduação dos cursos de estatística em geral no Brasil, apesar de poderem estar sendo tratados de forma individual por professores e estudantes.

3 Trabalhos Relacionados

De forma geral, é possível encontrar na literatura várias propostas de sistemas para permitir a execução de programas distribuindo sua carga de execução em computadores que fazem parte de infraestruturas existentes. Alguns trabalhos publicados revisam esses estudos, como em (IVASHKO et al., 2018) e em (KHAN et al., 2017), mas nem todas as soluções revisadas nessas publicações continuam ativas. E diferente da proposta apresentada nesse trabalho, algumas necessitam que bibliotecas específicas sejam instaladas nos clientes para a execução do código do usuário, sendo construídas para uma linguagem de programação específica ou ainda necessitando que o código do usuário seja reescrito. Nessa seção apresentaremos alguns desses estudos que se relacionam com a proposta da plataforma apresentada neste trabalho.

3.1 HTCondor - 1984

Desenvolvido pela University of Wisconsin-Madison o HTCondor (WISCONSIN–MADISON, 2022) é econhecido como um dos principais sistemas de processamento em lote. O HTCondor aproveita o tempo ocioso dos equipamentos em uma arquitetura focada em computação de alto rendimento (HTC), que permite sua execução em ambientes heterogêneos. Também apresenta suporte nativo a contêineres e é altamente escalável. O HTCondor é usado no Open Science Grid e no CERN. Apesar disso é uma solução baseado em linha de comando onde o usuário deve criar um arquivo de submissão usando a linguagem de descrição de tarefas *Classified Advertisements*, como mostrado na 12.

```
# science3.sub -- run 100 instances of science.exe, with
#  unique directories named by the $(Process) macro
executable              = science.exe
arguments               = "infile-A.txt infile-B.txt outfile.txt"
should_transfer_files   = IF_NEEDED
when_to_transfer_output = ON_EXIT
initialdir              = run$(Process)
transfer_input_files    = infile-A.txt,infile-B.txt
log                     = science3.log
queue 100
```

Figura 12 – Exemplo de um arquivo de submissão (o autor)

As diferenças em relação a plataforma PESC, proposta neste trabalho, está na sua interface *web* para usuários e administradores, além de abstrair a complexidade da criação de contêineres e na submissão de tarefas onde não é necessário que o usuário aprenda outras ferramentas/linguagens.

3.2 BOINC - 2004

Desenvolvido na Universidade da Califórnia, Berkeley, o BOINC (BOINC, 2022) é uma plataforma que permite tanto a computação voluntária quanto a computação em *grid* em ambiente local. Nesse tipo de *grid* o BOINC também usa recursos ociosos de computadores existentes para executar trabalhos distribuídos por um servidor. O usuário deve criar um projeto para cada necessidade de execução e configurar seu código como um aplicativo BOINC.

A plataforma apresentada neste trabalho compartilha alguns conceitos da computação em *grid* em ambiente local do BOINC, mas abstrai a complexidade da plataforma para tornar a sua adoção mais fácil para que qualquer usuário possa executar seus códigos e simulações, é fortemente baseado em contêineres, além de não interferir no código do usuário.

3.3 Everest - 2014

Desenvolvido pelo *The Institute for Information Transmission* (IITP RAS) o Everest (SUKHOROSLOV; AFANASIEV, 2014) é uma plataforma *web* que permite a execução de aplicativos em recursos de computação distribuída. As aplicações devem ser portadas para a plataforma que permite a criação de um fluxo de execução de aplicações. O Everest é disponibilizado com um serviço na *web* e não disponibiliza o servidor para instalação em uma infraestrtura local, apenas o cliente que deve ser instalado nas máquinas que estarão conectadas ao servidor do Everest para receber as tarefas que deverão ser executadas.

As diferenças em relação a plataforma PESC está na disponibilidade do módulo servidor para ser instalado nas dependências da instituição, mantendo a privacidade sobre estudos sensíveis ou mesmo confidênciais. Não interferir no código do usuário por ser fortemente baseado em contêineres, além disso não depende de componentes de terceiros nos

módulos essenciais da plataforma e permite a execução de códigos em qualquer linguagem de programação.

3.4 MiCADO - 2016

O MiCADO (KISS et al., 2019) é um *framework* genérico que abstrai a complexidade de detalhes técnicos de baixo nível das tecnologias de nuvens para simplificar o desenvolvilmente de aplicações baseadas em microsserviços. A estrutura genérica do MiCADO é desenvolvida para suportar a orquestração dinâmica de aplicativos em nuvem tanto na implantação quanto no tempo de execução.

A plataforma apresentada nesse trabalho é desenvolvida como uma solução centrada no usuário que faz uma distribuição de carga de execução em infraestrutura existente, mas não determina a forma como o usuário deve escrever seus códigos e não depende de componentes de terceiros nos módulos essenciais.

3.5 OSCAR - 2019

O OSCAR (PÉREZ et al., 2019) é um *framework* open source que oferece suporte à computação sem servidor em plataformas locais para aplicativos de processamento de dados orientados a eventos. O OSCAR é baseado no provisionamento automatizado de um cluster Kubernetes, auto escalonável, usado na implantação automatizada de uma estrutura de função como serviço que são associadas a eventos de uploads de arquivos em um servidor de armazenamento. Os usuários acessam o serviço através de uma interface *web* prática e intuitiva. A solução é criada a partir de um conjunto de outros *frameworks* que realizam as tarefas de escalonamento e para criar as funções sem servidor com Docker e Kubernetes.

A plataforma PESC não depende de componentes de terceiros nos módulos essenciais e por possuir uma arquitetura mais simples facilita o proceso de gerenciamento e implantação para as equipes de TI das instituições.

3.6 DLHub - 2021

O DLHub (LI et al., 2021) é um sistema de aprendizagem que fornece recursos de publicação e serviço de modelos para aprendizado de máquina científica. A infraestrutura de serviço baseia-se no *funcX*, uma plataforma de função como serviço distribuída, desenvolvida especificamente para apoiar a execução remota e distribuída de funções.

A plataforma apresentada neste trabalho tem objetivos parecidos, mas é desenvolvida de uma forma que não determina como o código do usuário deve ser escrito, permitindo que programas legados possam ser executados sem modificações, além de permitir o compartilhamento de recursos ociosos dos computadores da instituição.

3.7 Soluções para problemas específicos

Outros trabalhos apresentam soluções que fazem uso de infraestruturas similares, mas que resolvem problemas específicos, como em (ALFAILAKAWI et al., 2021) onde uma infraestrutura para processamento paralelo e distribuido é construida para a execução do SCA (Sine Cosine Algorithm) aproveitando os recursos nativos de computação em memória do Spark, um mecanismo de processamento de dados em grande escala. Em (BORLEA et al., 2019) é apresentado uma plataforma que permite a execução de algoritmos de agrupamento (Clustering algorithms) de forma paralela e distribuida através do procedimento de MapReduce. Em (WATANABE; FUKUSHI, 2020) é apresentado uma forma de permitir a comunicação assistida por servidor entre processos do OpenMPI, que permite a comunicação entre as máquinas envolvidas na execução mesmo quando não é possível existir a comunição direta entre elas. Por fim em (BAHAREVA et al., 2020) apresenta uma proposta de uma plataforma para a automação de computação distribuída baseado em tecnologias de nuvem e focada no processamento de *big data* de forma colaborativa em *hardware* dedicado.

3.8 Conclusão

Com a evolução da tecnologia é possível revisar a forma como esses sistemas podem ser implementados, mantendo o foco na facilidade da adoção pelo usuário e mantida pelas equiples de TI das instituições. Em (BAHAREVA et al., 2020) é dito que “As

soluções baseadas em Hadoop e produtos relacionados não são fáceis de implantar e gerenciar e exigem um nível significativo de conhecimento e experiência para usá-las em sua própria infraestrutura" onde fica claro que nem sempre as soluções apresentadas levam em consideração esses pontos quando definem os elementos que farão parte de sua arquitetura.

4 Plataforma PESC

A PESC (*Parallel Experience for Sequential Code*) é uma plataforma que permite que seus usuários executem programas de computador nos recursos computacionais ociosos de uma instituição, sem a necessidade de realizar configurações específicas nesses recursos e nem realizar alterações significativas em seus programas. Os programas dos usuários da plataforma podem ser executados um número definido de vezes, em paralelo ou não, de forma transparente para o usuário. A plataforma PESC é acessada através de uma interface *web* intuitiva para que o usuário consiga configurar e executar seus códigos de forma autônoma. O processo de instalação e configuração da plataforma está disponível no Apêndice A.

Instituições de pesquisa que possuem recursos computacionais, computadores *desktop*, distribuídos em salas de aulas e laboratórios podem otimizar o uso desses equipamentos quando estão ociosos. Vários estudos e pesquisas podem depender de uma capacidade computacional que um equipamento desse tipo, de forma isolada, pode não disponibilizar. Pesquisas realizadas nessas instituições podem se beneficiar com o uso da plataforma PESC por ela apresentar os seguintes princípios:

- O usuário tem autonomia para definir ambientes de execução sem depender de modificações realizadas pelas equipes de TI nos equipamentos;
- O usuário pode executar seu código sem modificações impostas pela plataforma;
- Através de ajustes mínimos o usuário pode fazer um código sequencial ser executado de forma paralela.

4.1 Softwares e tecnologias utilizadas

No desenvolvimento dessa proposta várias tecnologias diferentes foram utilizadas sempre com a preocupação na manutenção da garantia do acesso a elas. Desta forma, e para manter o projeto aberto, todas as tecnologias utilizadas são reconhecidas por serem software livre ou de código aberto.

- Python[1]: É a linguagem de programação base de todo o projeto. É uma linguagem interpretada, orientada a objetos e com tipagem dinâmica que enfatiza a facilidade

[1] https://www.python.org/

na leitura do código e consequentemente a redução no custo de manutenção;

- REST API[2]: É uma interface de programação de aplicações (API) que define um conjunto de regras na integração entre aplicações através do protocolo *HTTP* e que estão de acordo com os princípios REST (*Representational state transfer*);
- Tecnologias do módulo gerente:
  - Django[3]: É um *framework Python* para o criação de aplicações *web* que abstrai a complexidade desse tipo de desenvolvimento com a finalidade de aumentar a produtividade das equipes envolvidas;
  - Django Rest Framework[4]: É um *framework* desenvolvido em *Python* que funciona integrado ao *Django* para permitir a criação de chamadas *REST API*;
  - PostgreSQL[5]: É um sistema gerenciador de banco de dados relacionais que apresenta características de segurança, confiabilidade, robustez e desempenho para ser usado em servidores de sistemas com alta demanda de acesso e armazenamento de informações;
  - Vue.js[6]: É um *framework* desenvolvido na linguagem *Javascript* para auxiliar na construção de interfaces de comunicação com o usuário através de páginas *web*;
- Tecnologias do módulo cliente:
  - Flask[7]: É um *framework* desenvolvido em *Python* que de forma similar ao *django rest framework* permite a criação de chamadas *REST API*, mas que funcionam de forma independente necessitando apenas da linguagem *Python* para funcionar;
  - SQLAlchemy [8]: É uma ferramenta que faz o mapeamento objeto relacional para Python e fornece um conjunto completo de padrões de persistência de forma simples e eficiente;
  - Docker[9]: É uma plataforma para a criação e gerenciamento de contêineres que permite separar as aplicações de uma infraestrutura específica para a

[2] https://www.ics.uci.edu/ fielding/pubs/dissertation/rest_arch_style.htm
[3] https://www.djangoproject.com/
[4] https://www.django-rest-framework.org
[5] https://www.postgresql.org/
[6] https://vuejs.org/
[7] https://flask.palletsprojects.com/en/2.2.0/
[8] SQLAlchemy
[9] https://www.docker.com/

sua execução, simplificando dessa forma o processo de execução em outros ambientes;

– SQLite[10]: É uma biblioteca que implementa um banco de dados relacional pequeno, rápido e autônomo para ser usado como suporte a aplicações, inclusive de forma embutida;
– NVIDIA Container Toolkit [11]: É um conjunto de ferramentas que permite a criação de cointainers configurados automaticamente para acessar as *GPUs* da NVIDIA.

4.2 Visão Geral da Plataforma

Como mostrado na Figura 13, a plataforma PESC é um sistema modular dividido em três componentes que são hospedados em uma infraestrutura local, o cliente, o gerente e o *frontend*. Apesar de ser desenvolvido usando tecnologias de código aberto, o código da plataforma PESC ainda não está disponível, dada a fase atual de testes, mas no futuro a plataforma será disponibilizada para a comunidade como uma ferramenta de código aberto.

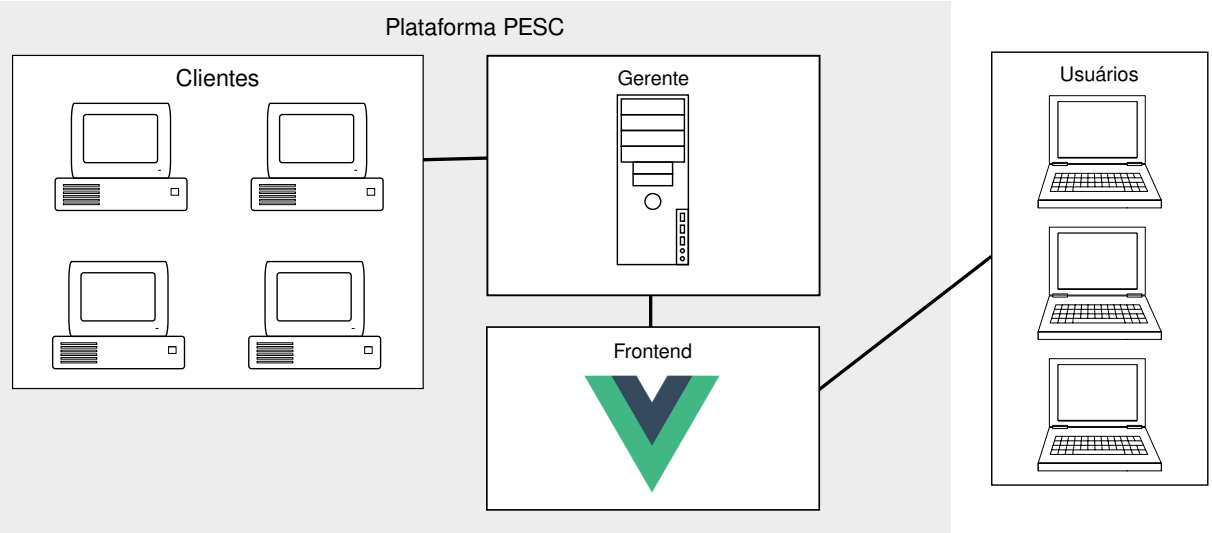


Figura 13 – Visão geral da plataforma (o autor)

O gerente é o módulo instalado em um servidor que centralizará as operações da plataforma e os clientes são os recursos computacionais ociosos que podem estar fisicamente

[10] https://www.sqlite.org/index.html
[11] https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/overview.html

alocados em diferentes ambientes da instituição. Para representar essa distribuição física dos computadores, como mostrado na Figura 14, a plataforma permite a criação de salas. Estas salas podem assumir qualquer tipo de configuração de agrupamento de máquinas, dependendo da necessidade da aplicação. Na plataforma está inicialmente configurada uma sala pública para uso geral (*General*), onde todos os clientes adicionados a ela podem ser acessados por qualquer usuário. A criação de salas deve ser solicitada ao administrador da plataforma, para evitar que configurações impróprias sejam realizadas por outros usuários, e o usuário solicitante será o administrador da sala criada onde poderá adicionar clientes e restringir o seu acesso a apenas um grupo de usuários. Os clientes inseridos em uma sala podem ser movidos para outras salas, mas um usuário só poderá realizar essa operação se ele for o administrador de ambas as salas, ou o administrador da plataforma.

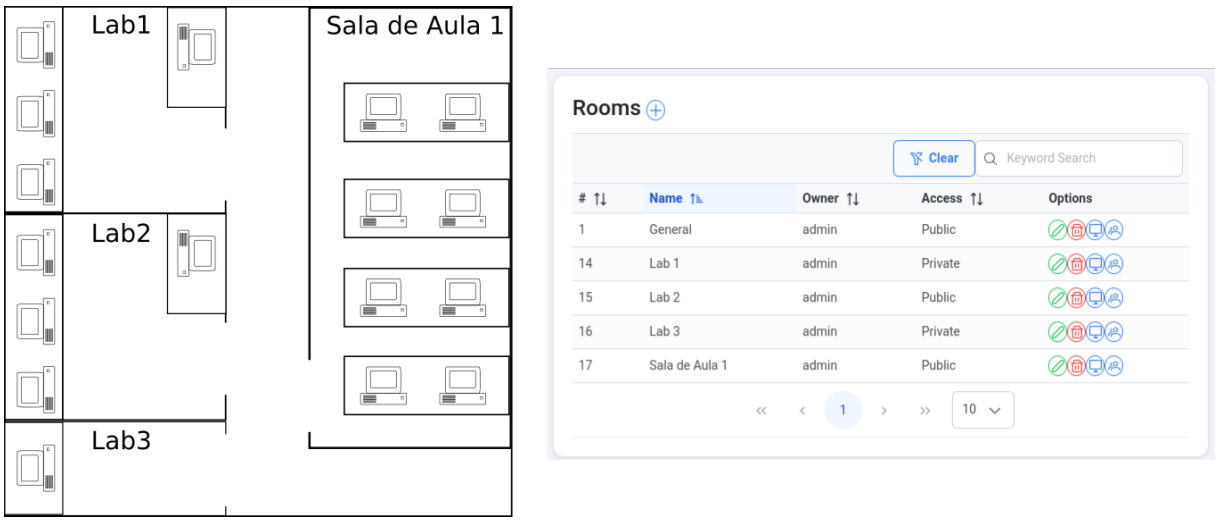


Figura 14 – Organização física dos clientes e sua representação em salas da plataforma (o autor)

Para manter uma compatibilidade com os programas de computador existentes, a platorma PESC é construída para ser uma solução não intrusiva e permitir que seus usuários executem seus códigos sem modificações significativas (ou, em muitos casos, sem alterações). O código do usuário será executado como uma chamada de linha de comando com os parâmetros relacionados à execução atual. Portanto, o código do usuário deve incluir um cabeçalho para usar esses valores de parâmetro enviados no momento da execução, mas pode ignorar esse cabeçalho e executar o código mesmo sem conhecer esses valores. A plataforma PESC disponibiliza o código do cabeçalho em seu site, simplificando esse procedimento de configuração para o usuário. Vale ressaltar que o cabeçalho já está

disponível para diversas linguagens de programação, como *Python*, *Java*, *C/C++*, *R*, entre outras.

O usuário deve ter a liberdade de executar o código com as mudanças que considerar apropriadas e a plataforma não deve interferir com essa liberdade, por isso o cabeçalho apresenta valores padrão definidos e a adição desse cabeçalho ao código não interferirá na sua execução no computador do usuário ou em outro ambiente computacional. Os parâmetros recebidos pelo código do usuário são apresentados na lista abaixo:

- app_dir: O diretório atual onde o código é executado;
- checkpoint_dir: O diretório onde os pontos de recuperação de falhas devem ser armazenados;
- output_dir: O diretório que será retornado ao gerente após a execução do código. O usuário deve armazenar toda a saída desejada de seu código neste diretório;
- rank: O numero de identificação da instância em execução. É um número sequencial que inicia em zero;
- repetitions: O número de repetições solicitadas pelo usuário;
- master_addr: O endereço *IP* do cliente que recebeu rank=0;
- master_port: A porta associada ao endereço *IP* do cliente que recebeu rank=0;
- parameters: Uma lista de valores informados pelo usuário que é recebida como um vetor.

```
1  import argparse,os
2  parser=argparse.ArgumentParser()
3  parser.add_argument("--app_dir","-a",default=f"{os.getcwd()}/")
4  parser.add_argument("--checkpoint_dir","-c",default=f"{os.getcwd()}/checkpoint")
5  parser.add_argument("--output_dir","-o",default=f"{os.getcwd()}/output")
6  parser.add_argument("--rank","-r",type=int, default=0)
7  parser.add_argument("--world_size","-w",type=int, default=1)
8  parser.add_argument("--master_addr","-d",default="127.0.0.1")
9  parser.add_argument("--master_port","-t",type=int, default=9000)
10 parser.add_argument("--parameters","-p",nargs="*",default=[])
11 args=parser.parse_args()
```

Algoritmo 4.1 – Exemplo do cabeçalho para a lingugem *Python*

Se o cabeçalho não estiver no código do usuário, o PESC poderá incluí-lo automaticamente. Essa funcionalidade está disponível para os programas escritos nas linguagens interpretadas, como o *Python* e o *R*, mas em novas versões da plataforma estará disponível também para outras linguagens de programação.

O código do usuário é executado em um contêiner *Docker* e para a configuração do ambiente de execução deste contêiner, aqui chamado de Domínio, é necessário a configuração de um arquivo *Dockerfile* e um arquivo *requirements.txt*, como mostrado nos Algoritmos 4.2 e 4.3. Apesar do arquivo *requirements.txt* ser um mecanismo usado para a instalação de pacotes na linguagem de programação *Python*, esse arquivo é necessário independente da tecnologia usada na criação do código do usuário. Internamente a plataforma usará códigos em *Python* para iniciar a execução do código do usuário e por isso existe a necessidade da criação desse arquivo.

```
1  FROM r-base:4.1.2
2  WORKDIR /app
3  RUN apt-get update && \
4    apt-get upgrade -y && \
5    apt-get install -q -y python3 python3-pip nano
6
7  RUN pip3 install --upgrade pip requests
8  COPY requirements.txt ./
9  RUN ln -s /usr/bin/python3 /usr/bin/python
10 RUN pip install --no-cache-dir -r requirements.txt
```

Algoritmo 4.2 – Arquivo Dockerfile para criar um contêiner para o R v4.1.2

```
1 pandas==1.2.4
2 requests==2.25.1
3 matplotlib==3.2.2
4 matplotlib-inline==0.1.3
5 matplotlib-venn==0.11.6
6 sklearn==0.0
```

Algoritmo 4.3 – Arquivo requirements.txt para instalar bibliotecas em um ambiente Python

Esses arquivos podem ser fornecidos pelo próprio usuário, mas para manter a plataforma simples uma loja que permite a criação de domínios é disponibilizada com um conjunto de tecnologias comumente utilizadas, como mostrado na Figura 15, que podem ser personalizadas com as bibliotecas necessárias para a execução do código do usuário, como mostrado na Figura 16. Essa personalização está disponível apenas para a criação de domínios na linguagem de programação *Python* e será usada para criar o arquivo *requirements.txt*, mas em novas versões da plataforma uma alternativa para esse recurso estará disponível também para outras linguagens de programação. Se não houver na loja uma opção que atenda às necessidades do usuário, ele pode fazer uma solicitação

que será atendida pelo usuário administrador da plataforma ou por outro usuário, e neste caso a opção será disponibilizada para o usuário solicitante após a validação realizada por usuários autorizados pela administração da plataforma.

Os contêineres criados para a execução na plataforma devem seguir uma padronização, como usar o diretório */app* como *root* da aplicação, ter os comandos *python3* e o *pip* configurados e executar a instalação das bibliotecas listadas no arquivo *requirements.txt*. Por isso as imagens não devem ser recuperadas diretamente de fontes externas, como o *docker hub*, e precisam ser validadas antes de inseridas na plataforma. A criação da loja de domínios automatiza esse processo e é uma parte fundamental para manter a plataforma simples e acessível para qualquer usuário.

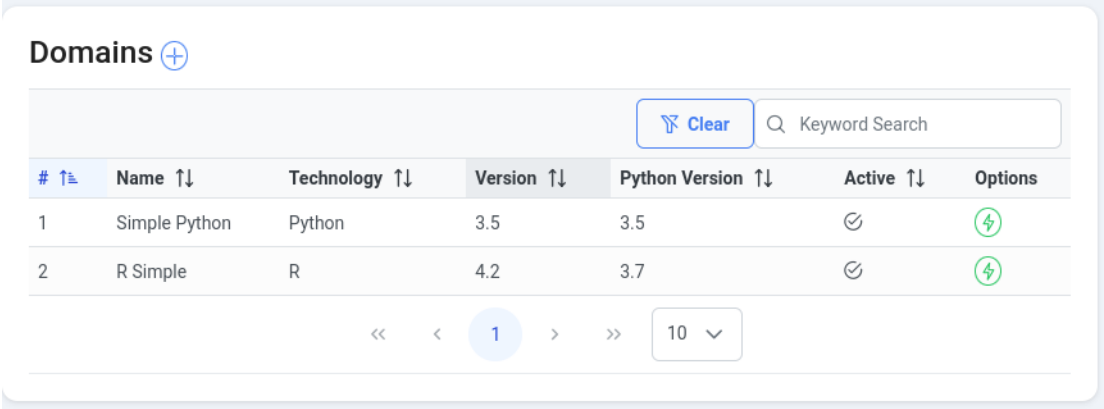


Figura 15 – Domínios disponíveis na loja (o autor)

Após a definição do Domínio, o usuário deve realizar a configuração do processo que será executado através do formulário apresentado na Figura 17. Os processos são os arquivos de código do usuário que serão executados nos contêineres e podem ser fornecidos pelo o usuário em um único arquivo na linguagem *Python* ou em um arquivo compactado no formato *zip*. O arquivo compactado é necessário quando o código está dividido em vários arquivos ou escrito em outras linguagens de programação.

A plataforma PESC permite também ao usuário carregar arquivos compartilhados que poderão ser solicitados pelos clientes no momento da execução do processo do usuário, caso ainda não estejam armazenados localmente no cliente. Esses arquivos permanecem como recursos compartilhados para todas as instâncias dos processos, do mesmo usuário, executando nesse cliente e dessa forma elimina a necessidade de transferir uma nova cópia do arquivo para cada processo que necessitar dela. Para evitar erros que afetem o funcionamento dos contêineres que estejam em execução paralela o acesso concorrente a

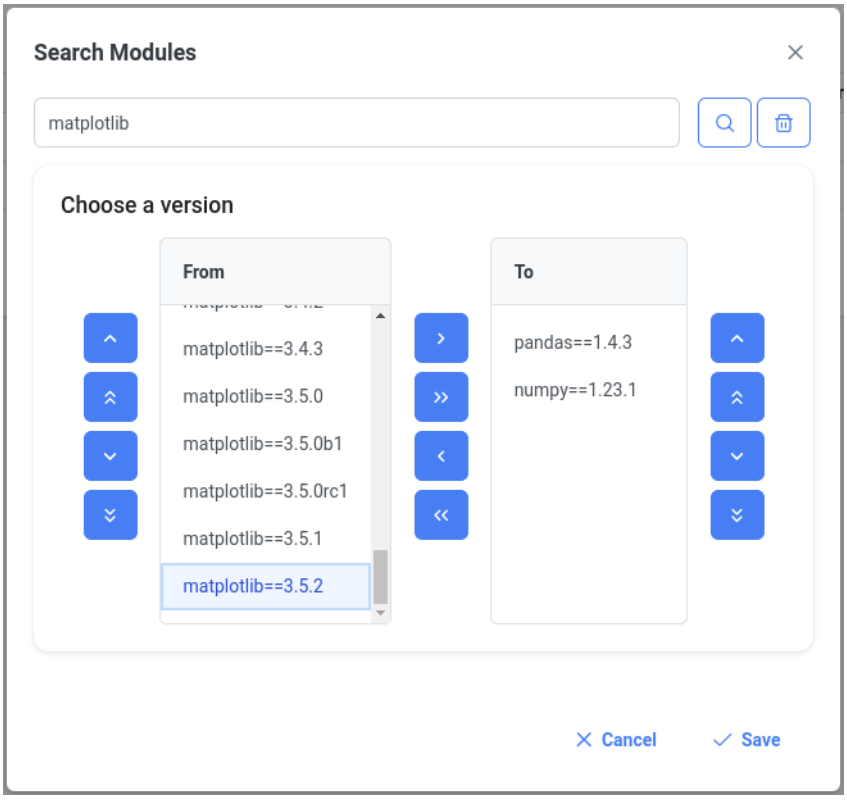


Figura 16 – Personalização das bibliotecas (o autor)

esses arquivos será realizado com a restrição de somente leitura. Um exemplo do uso deste recurso seria uma base de dados, no formato de arquivo *CSV*, que será carregada durante a execução de um processo do usuário que deve ser repetido 20 vezes em uma sala com 2 clientes, com esse recurso cada cliente receberá apenas uma cópia deste arquivo que estará disponível para cada uma das instâncias do processo que é recebida para execução.

Após a configuração do Domínio, do Processo e do carregamento de arquivos compartilhados, se necessário, o usuário deve fazer uma requisição de execução, que chamaremos apenas de requisição. A requisição é adicionada à uma fila de execução do usuário que será atendida quando existir clientes com disponibilidade de recursos para isso. No momento da criação da requisição o usuário deve informar os dados solicitados no formulário de requisição, Figura 18, e que estão descritos a seguir.

- Domínio: A definição do ambiente de execução;
- Processo: O código do usuário a ser executado;
- GPU: Se o processo precisa de *GPU*. Este valor indica que devem ser selecionados apenas clientes que possuam este recurso;
  - Devices Qty: Quantidade mínima de *GPUs* que o cliente deve ter para ser selecionado;

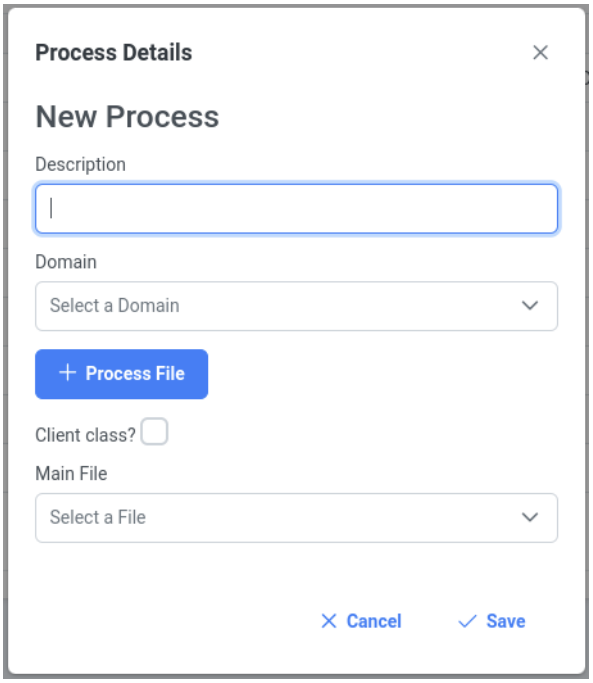


Figura 17 – Formulário para a criação de processos (o autor)

  - Minimum Capability: Número que identifica os recursos suportados pelo hardware da *GPU* da NVIDIA;
- Paralela: Se o código faz uso de recursos paralelos. Este valor informará aos clientes para aguardar a distribuição de todas as cópias solicitadas antes de iniciar a execução;
- Mesmo cliente: O usuário deve informar se todas as instâncias do processo paralelo devem ser executados no mesmo cliente. O código do usuário deve fazer uso do compartilhamento de recursos locais, como memória ou *GPU*;
- Repetições: Número de vezes que o processo deve ser executado. Cada execução será realizada em um contêiner diferente;
- Parâmetros: Uma lista de valores separados por vírgula que serão recebidos como parâmetro pelas instâncias do processo em tempo de execução;
- Arquivos compartilhados: Arquivos que podem ser necessários na execução do processo. O usuário deve ter carregado anteriormente esses arquivos na plataforma;
- Salas: Em que grupo de clientes o processo deve ser executado.

O monitoramento de execução das requisições pode ser realizado pelo usuário, em tempo real, através de um painel de controle (*dashboard*) disponível na interface *web*, como mostrado na Figura 19. Se o processo do usuário estiver integrado com os recursos da plataforma PESC será possível enviar detalhes de execução, como mensagens personalizadas e percentuais de execução, caso contrário somente as mensagens de início e

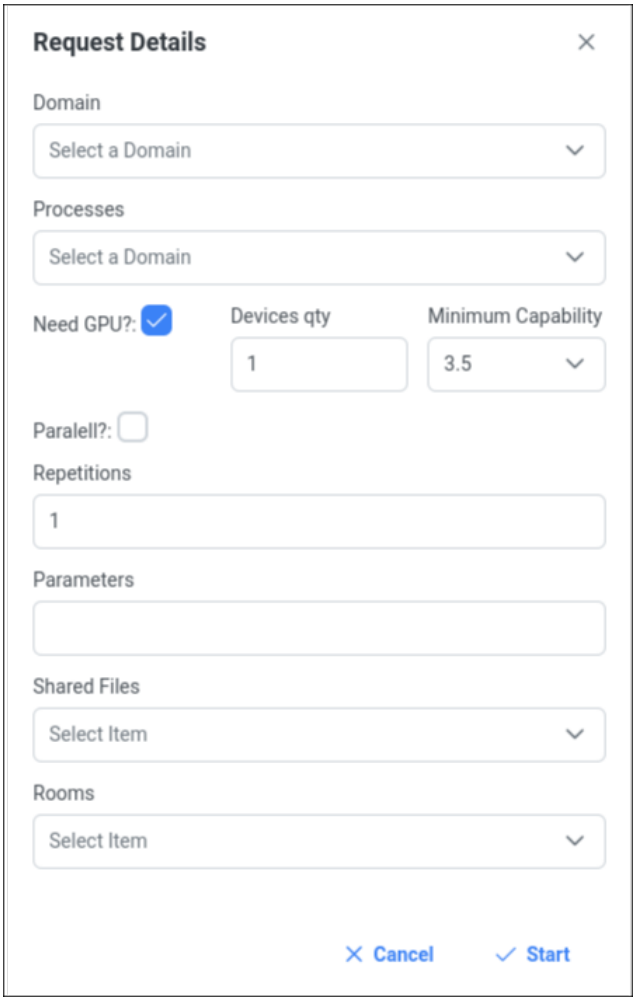


Figura 18 – Formulário de *Request* (o autor)

fim de execução serão exibidas. A integração com os recursos da plataforma é realizada com a implementação do código do usuário através de uma classe *Python* com as definições específicas. Um modelo dessa classe é disponibilizada na interface *web* da plataforma e o seu uso é opcional, considerando que as modificações no código do usuário para ser usado na plataforma podem ser mínimas.

Os clientes recebem as instâncias do processo do usuário definido na requisição e para cada instância é verificado se já existe a imagem *Docker* construída para esse processo, em caso negativo a construção da imagem é iniciado e após a sua finalização o cliente passa a receber outras instâncias para esse mesmo processo. Antes de iniciar a execução do contêiner, o cliente verifica se os arquivos compartilhadas configurados na requisição existem e em caso negativo eles são solicitados ao gerente. Durante a execução de uma requisição o cliente verifica periodicamente com o gerente se o usuário a cancelou e, em caso afirmativo, interrompe a execução de todas as instâncias do processo associadas a ela.

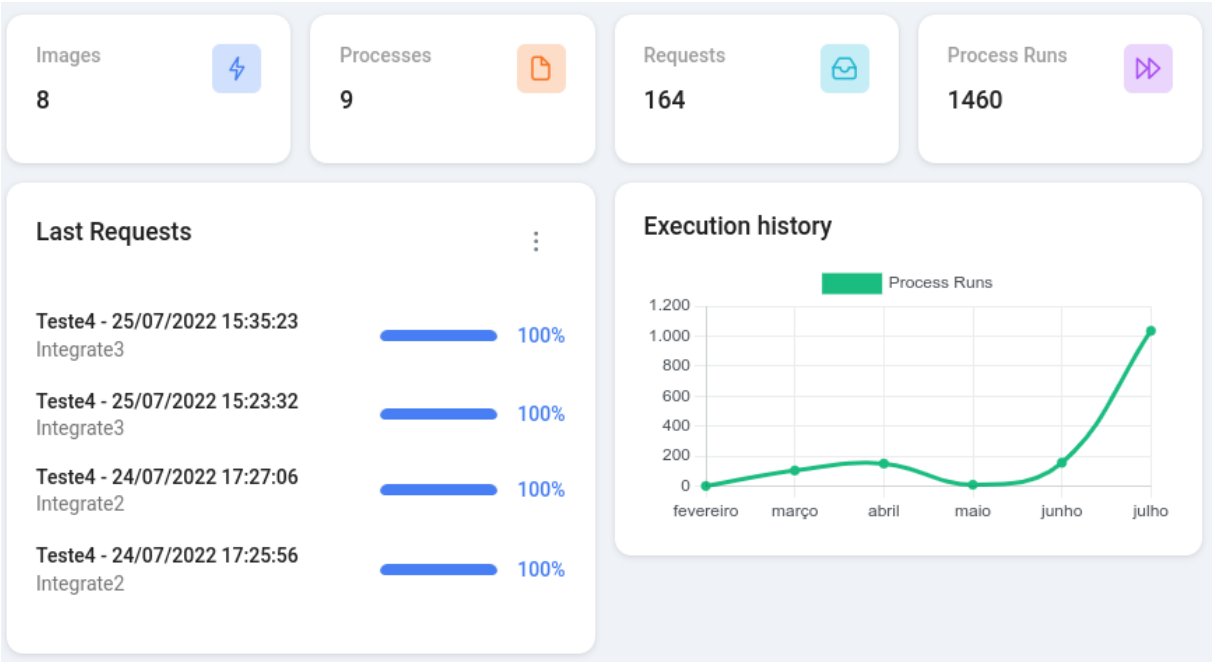


Figura 19 – Tela inicial da plataforma (o autor)

Após o encerramento da execução, o cliente compacta o diretório de *output*, informado no cabeçalho de parâmetros recebidos, e envia de volta para o gerente onde ficará disponível para o usuário que fez a requisição.

Após a execução da requisição o usuário pode solicitar a descarga (*download*) de todos os arquivos de *output* criados por cada cliente envolvido na execução da requisição. Os arquivos de todos os clientes são compactados em um único arquivo que é enviado para a solicitação de *download* do usuário. É possível que todas as saídas apresentadas em tela durante a execução do código do usuário, normalmente através do comando *print*, sejam agrupadas em um único arquivo, para isso é necessário que exista um arquivo de texto com o nome *output.txt* no diretório de *output* do contêiner. A plataforma identifica a existência desse arquivo em cada arquivo compactado retornado pelos clientes e concatena em um novo arquivo, ordenando esse processo pelo *rank* de cada instância executada para facilitar a análise dos resultados pelo usuário, como apresentado na Figura 20.

Todas essas etapas necessárias para usar a plataforma PESC estão simplificadas no processo apresentado no fluxograma na Figura 21, onde os elementos em verde representam as etapas com interação obrigatória do usuário.

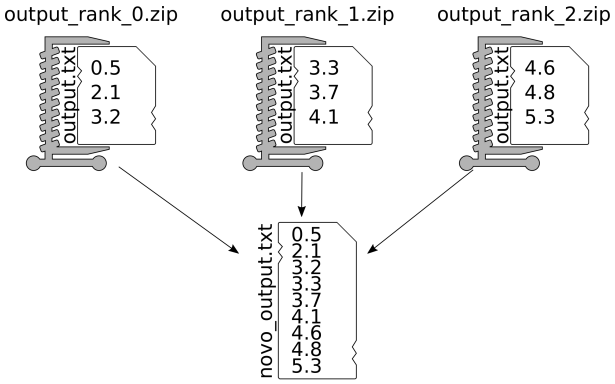


Figura 20 – Criação do novo arquivo agrupando as saídas recebidas (o autor)

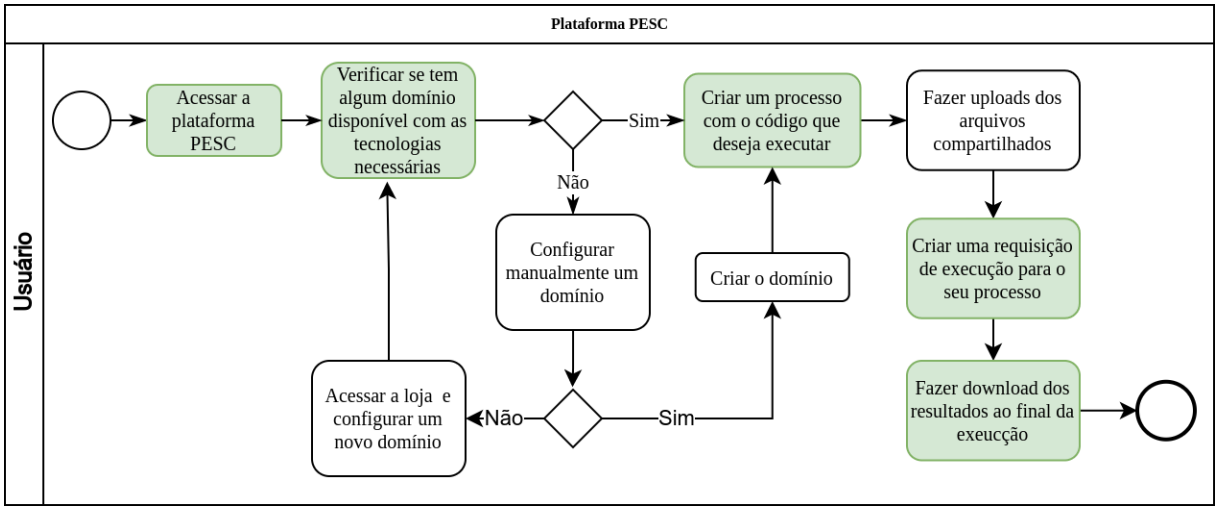


Figura 21 – Fluxo de execução da plataforma PESC (o autor)

4.2.1 Gerenciamento de GPU

Devido a variedade de computadores que podem ser configurados como clientes na plataforma PESC o gerenciamento do uso das *GPUs* dessas máquinas precisou ser definido de uma forma que não impactasse na distribuição das requisições que necessitam desse recurso e nem na sobrecarga desses computadores.

Atualmente a plataforma PESC está configurada para permitir o gerenciamento de placas baseadas em *GPU* fornecidas pela NVIDIA, que dependem do paradigma de programação CUDA, e para que os contêineres *Docker* possam acessar essas *GPUs* é necessário a instalação do *NVIDIA Container Toolkit* fornecido pela própria NVIDIA.

A comunicação com a *GPU*, para fins de monitoramento, pode ser realizado através do *NVIDIA Management Library* (NVML) que é instaldo junto com o driver da *GPU*, e por isso não é listado como uma dependência direta da plataforma. Atráves do NVML

é possível consultar a lista de processos em execução e quanto de memória RAM da *GPU* cada processo está usando. Em drivers mais antigos essa informação não pode ser recuperada, devido a uma limitação do próprio NVML, e o gerenciamento dos processos em execução fica limitado, neste sentido apenas uma única GPU seria utilizada.

A plataforma PESC tenta otimizar o uso dos recursos disponíveis nos clientes e pode compartilhar o uso de tempo da *GPU* entre processos diferentes. Esse processo só pode ser realizado se a versão do *driver* da *GPU* suportar a consulta dos processos em execução. Nos clientes onde isso não é possível a *GPU* será de uso exclusivo do processo que estiver atualmente em execução e o Cliente não aceitará novos processos que dependem desse tipo de recurso.

Para gerenciar o compartilhamento de recusos a plataforma PESC coleta dados de uso de CPU, memória RAM e memória RAM da *GPU*, durante a execução dos processos, e armazena os maiores valores alcançados para cada uma dessas medidas. Dessa forma é possível verificar a disponibilidade de recursos em um cliente para a execução de uma requisição do usuário. A distribuição das instâncias de um processo que necessitam de *GPU* está apresentado na Figura 22.

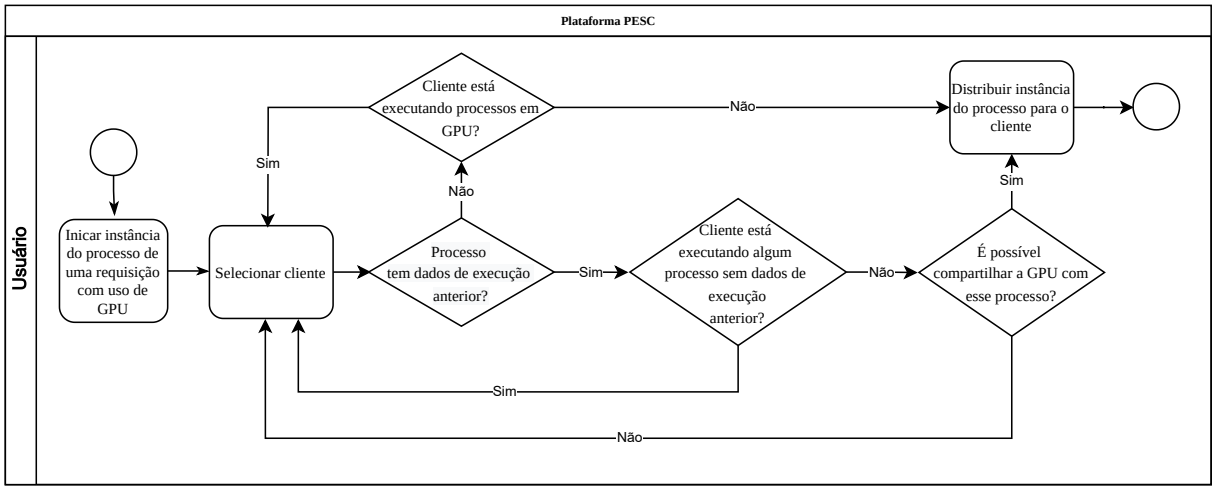


Figura 22 – Fluxo de distribuição baseado em GPU (o autor)

4.3 Arquitetura

Cada módulo, cliente e gerente, é composto por um conjunto de componentes que são detalhados a seguir. O frontend é desenvolvido em *Vue.js* e se comunica com o gerente através de chamadas *REST API* e não será detalhado no escopo deste trabalho.

4.3.1 Arquitetura do Gerente

O gerente, Figura 23, é um servidor *web* desenvolvido com o *framework Django*, que foi escolhido por fornecer um conjunto de recursos para o desenvolvimento desse tipo de sistema. Entre esses recursos é possível destacar a integração com banco de dados, módulo de administração integrado, autenticação do usuário e controle de acesso às funcionalidades do sistema. Integrado ao *Django* foi configurado o *Django Rest Framework* para permitir a criação de *endpoints* de comunicação com os clientes e o *frontend* através de chamadas *REST API*. Os dados são armazenados em um banco de dados *PostgresSQL*, mas devido a abstração que o mapeamento objeto relacional *ORM* do *Django* oferece é possível realizar a substituição por outros sistemas de bancos de dados relacionais sem a necessidade de alterações no código do sistema. Para gerenciar as requisições dos usuários e as comunicações com os clientes um conjunto de monitores foram desenvolvidos, cada monitor é executado em uma *thread* diferente e mantém o estado da aplicação sempre atualizado.

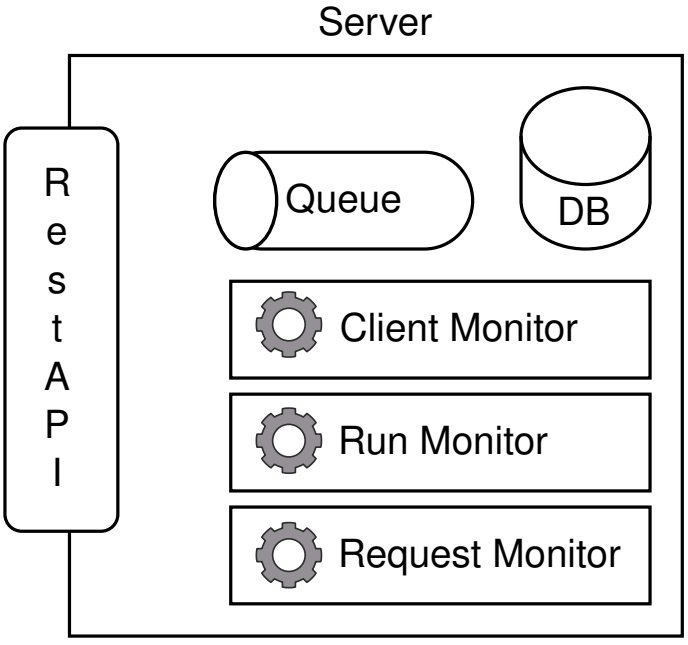


Figura 23 – Arquitetura do Gerente (o autor)

4.3.1.1 Monitor de Clientes

Os clientes atualizam seu status com o gerente através de chamadas realizadas pelos próprios clientes, mas devido à possíveis inconsistências da rede ou um desligamento

inesperado de um dos clientes, o gerente também monitora o status dos clientes conectados a ele. Neste processo, verifica-se se o cliente responde a uma chamada *REST API* e em caso negativo é verificado se o computador onde o cliente está instalado é alcançado através da rede. Caso isso seja possível o Monitor de Cliente verifica a possibilidade de iniciar o serviço do cliente PESC no computador onde ele está instalado. Esta possibilidade de inicialização é configurada em cada cliente através de seu arquivo de configuração.

O arquivo de configuração do cliente é definido pelo administrador do sistema no momento de sua configuração e deve ser criado com as informações apresentadas na Seção A.2 e customizadas para cada cliente.

4.3.1.2 Monitor de Requisições

As requisições são adicionadas a uma fila criada para o usuário que as solicitou. O monitor de solicitação gerencia essas filas e analisa cada requisição. A requisição pode estar configurada para ser executada uma única vez ou para várias repetições e, neste caso, a requisição continuará na fila do usuário até que a quantidade total de instâncias de processos sejam encaminhadas para os clientes disponíveis. A seleção do cliente é baseada nas características da requisição, por exemplo, se ele precisa de uma *GPU* ou não, e na carga de trabalho que já foi distribuída para cada cliente, sendo selecionado o cliente que tenha mais recursos disponíveis para a execução.

4.3.1.3 Monitor de Execução de Processos

Cada instância do processo de uma requisição que é enviada a um cliente é chamada de *Process Run*. O monitor de execução de processos verifica periodicamente com o cliente, que recebeu uma instância de processo para executar, o status de sua execução. *Process Runs* que não podem ser avaliados devem ser movidos para outros clientes e uma notificação de cancelamento é enviada para o cliente atual, os clientes que estão fora-de-ar (*offline*) receberão a notificação de cancelamento na próxima conexão com o gerente e suspenderão a execução. Dessa forma, enquanto um cliente estiver disponível na plataforma, uma instância de processos será direcionado para ele, respeitando os limites definidos no seu arquivo de configuração, e as requisições deverão ser totalmente atendidas.

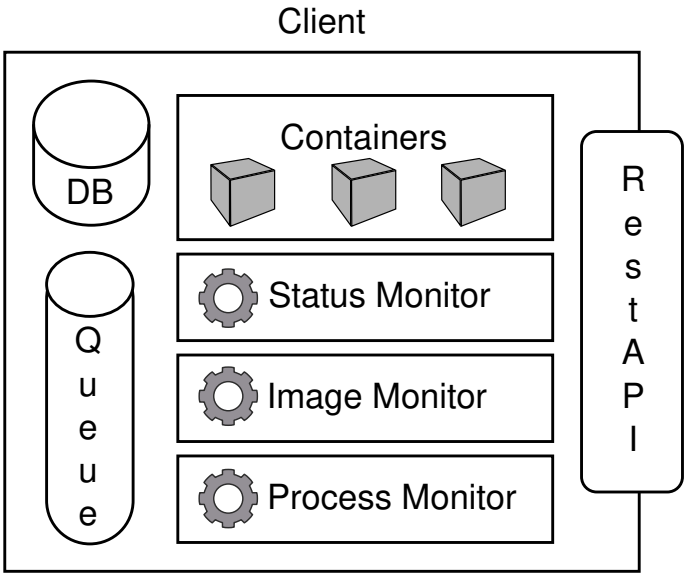


Figura 24 – Arquitetura do Cliente (o autor)

4.3.2 Arquitetura do Cliente

O cliente, Figura 24, é uma aplicação desenvolvida usando o *framework Flask* que permite a criação de *endpoints* de comunicação com o gerente e os contêineres *Docker* em execução através de chamadas *REST API*. Os dados são armazenados em um banco de dados *SQLite3* que é acessado através dos recursos disponíveis no pacote *SQLAlchemy*, que além de fazer o mapeamento entre os objetos do sistema e as tabelas do banco de dados cria uma abstração que permite realizar a troca do banco de dados relacional sem a necessidade de alterações no código do sistema. Para gerenciar as operações de construção de imagens e execução de contêineres, um conjunto de monitores foram desenvolvidos, cada monitor é executado em uma *thread* diferente e mantém o estado da aplicação sempre atualizado.

O cliente é executado como um serviço de segundo plano que é iniciado por um usuário do sistema operacional criado para essa finalidade, ver a Seção A.2. O cliente deve equilibrar o uso dos recursos do computador com outros usuários que podem estar usando o sistema operacional de forma concorrente. Isso é necessário porque os computadores clientes não são exclusivamente dedicadas à plataforma PESC.

4.3.2.1 Monitor de Estado

O cliente deve informar ao gerente sobre as características de sua operação. O monitor de status é responsável por coletar essas informações para garantir ao gerente que o cliente está em condições operacionais e habilitado para executar os processos recebidos. Se o gerente não receber essas informações, ele poderá mover o processo em execução para outros clientes disponíveis. O monitor de estado verifica as seguintes informações:

- Uso simultâneo do computador por outros usuários do sistema operacional;
- A porcentagem de uso de memória *RAM*;
- A porcentagem de uso do processador;
- A porcentagem de uso de memória *RAM* da *GPU*, se houver.

Quando a porcentagem de uso do processador atinge 70% o cliente informa ao gerente que atualmente não é capaz de receber novos processos para executar. Em caso de uso concorrente por outro usuário do sistema operacional os recursos alocados para os contêineres atualmente em execução são reduzidos para 10%. Estas porcentagens são definidas no arquivo de configuração do cliente e podem ser alteradas a qualquer instante.

4.3.2.2 Monitor de imagem

Os contêineres *Docker* são criados com base em uma imagem que deve ser previamente construída nos clientes antes que a instância do processo recebido comece a ser executada. O monitor de imagem é responsável por monitorar a necessidade de construção das imagens e notificar todo o sistema do progresso deste processo. Se a imagem não existir no cliente, a informação necessária para a sua criação é solicitada ao gerente. Qualquer instância de processo que dependa de uma imagem que está sendo construída esperará que a construção seja concluída com sucesso antes de iniciar a execução. Apesar de existir recursos para a realização desse tipo de monitoramento com os contêineres *Docker* a plataforma pode ser configurada para usar outras tecnologias e estar totalmente integrada ao *Docker* representaria um alto custo de manutenção caso outra tecnologia de contêiner venha a ser utilizada.

4.3.2.3 Monitor de Processo

Processos criados pelo usuário são executados em contêineres *Docker* que precisam de uma infraestrutura de arquivos e diretórios para seu correto funcionamento. O monitor de processo é responsável por monitorar o ciclo de vida da instância do processo e preparar a infraestrutura necessária para sua execução. O contêiner é executado em modo desconexo e assim outros contêineres podem ser iniciados simultaneamente, respeitando o limite de consumo de recursos do sistema operacional definido no arquivo de configuração do cliente. A instância do processo em execução no contêiner se comunica com o cliente através de chamadas *REST API*, como apresentado na Figura 25.

Os processos do usuário podem usar o diretório *checkpoint*, informado no cabeçalho de parâmetros recebidos, e em caso de falha na execução do contêiner será verificado a existência de algum ponto de recuperação nesse diretório. Se existir será iniciando uma nova tentativa de execução do contêiner a partir desse ponto de recuperação informado no arquivo.

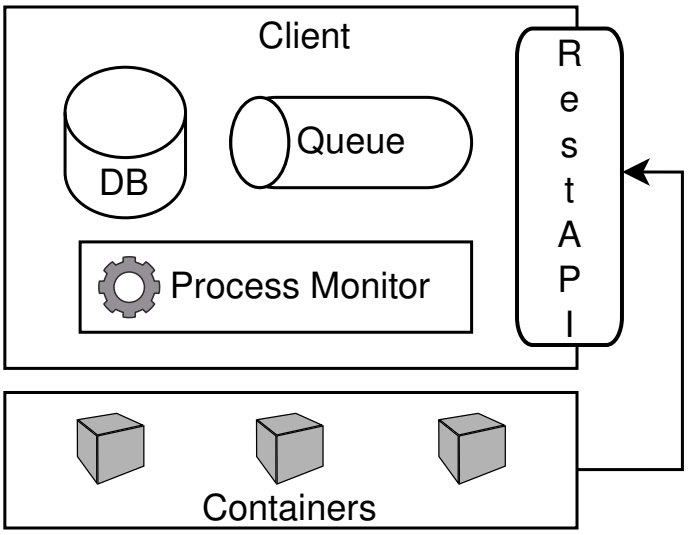


Figura 25 – Monitor de Processos (o autor)

4.4 Exemplos de uso

Para demonstrar como toda a plataforma cria uma abstração na complexidade de execução de uma simulação computacional dois cenários hipotéticos foram criados e suas execuções serão demonstradas passo a passo.

4.4.1 Exemplo 1 - Usabilidade - Gerador de números aleatórios

Passo 1: Código do usuário

Um estudante do curso de estatística desenvolveu um programa para gerar números aleatórios através de uma distribuição *gaussiana*. O código desenvolvido em *Python* realiza uma transformação de *Box-Muller* em números aleatórios gerados a partir de um gerador uniforme para gerar um par de valores normais independentes. O gerador uniforme é baseado em Geradores Congruentes Lineares (LCG) tendo os parâmetros *modulo*, *multiplicador* e *incremento* definidos com os valores da biblioteca *gcc*.

- módulo, m = 2^{31}
- multiplicador, a = 1103515245
- incremento, c = 12345

A versão final do código desenvolvido pelo estudante está apresentado no Algoritmo 4.4.

```
1  import time
2  from math import sqrt, cos, sin, log, pi
3
4  class GeradorAleatorio:
5
6      def __init__(self) -> None:
7          self.x = 0
8
9      # Retorna a parte decimal da função time
10     def gerar_semente(self):
11         t = time.time()
12         return int((t-int(t))*1000000000)
13
14     # Baseado em Geradores congruentes lineares
15     def gerar_uniforme(self, modulo=(2**31), mult=1103515245, c = 12345, seed
       =123456789):
16         if self.x == 0: # Gerar o primeiro valor
17             self.x = (seed*mult+c)%modulo
18         else:
19             self.x = (self.x*mult+c)%modulo
20         U = self.x/modulo
21         return U
22
23     # Rrealiza uma transformação de Box-Muller
24     def gerar_normal(self):
25         U1 = self.gerar_uniforme(seed=self.gerar_semente())
26         U2 = self.gerar_uniforme(seed=self.gerar_semente())
27         Z0 = sqrt(-2*log(U1))*cos(2*pi*U2)
28         Z1 = sqrt(-2*log(U1))*sin(2*pi*U2)
29         return Z0, Z1
30
31 if __name__ == "__main__":
```

```
32      rn = GeradorAleatorio()
33      v1, v2 = rn.gerar_normal()
34      print(f"{v1}, {v2}")
```

Algoritmo 4.4 – Código para gerar números aleatórios

Passo 2: Conversão em uma simulação computacional

A partir do resultado obtido o estudante resolve executar o código para gerar 20.000.000 números aleatórios. Nessa etapa o método *gerar_normal* será chamado várias vezes em um laço e as alterações necessárias para executar essa simulação são realizadas apenas no método *main* e estão apresentadas no Algoritmo 4.5.

```
1   if __name__ == "__main__":
2       rn = GeradorAleatorio()
3       for i in range(10000000): #gerar_normal retorna 2 números aleatórios
4           v1, v2 = rn.gerar_normal()
5           print(f"{v1}, {v2}")
```

Algoritmo 4.5 – Alterações do método main na geração de números aleatórios

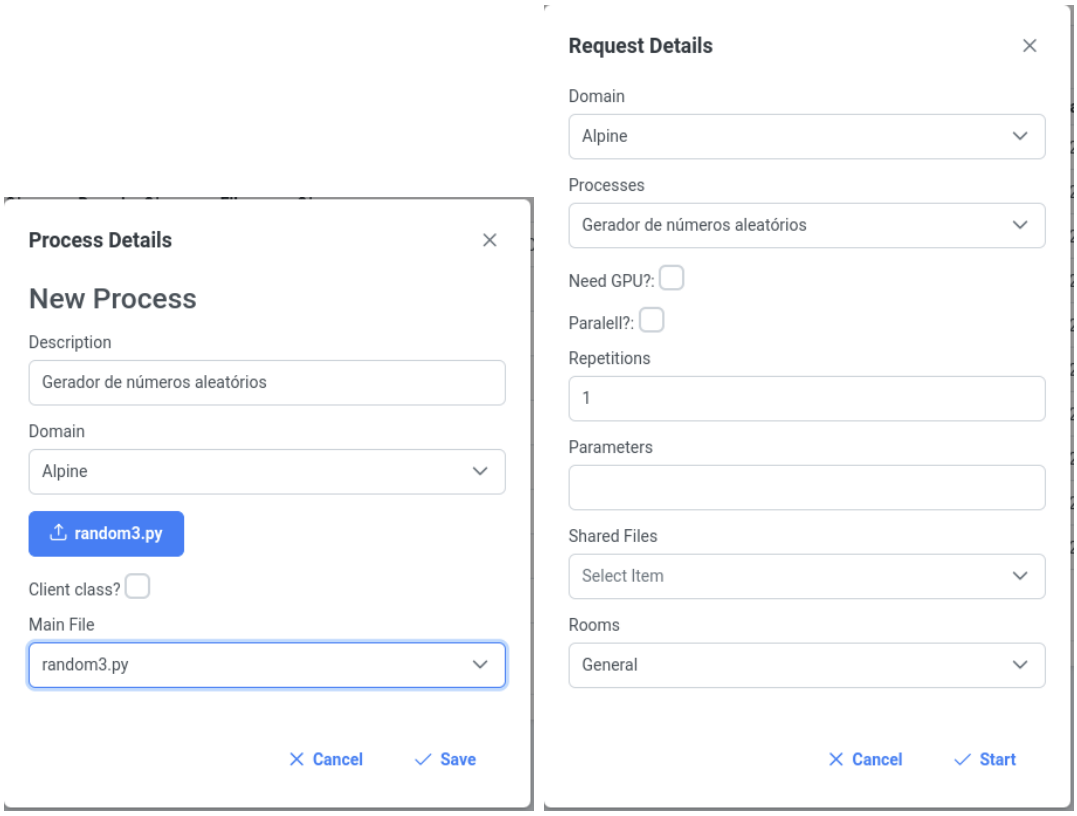
Passo 3: Execução em ambiente compartilhado

Para executar o código no ambiente compartilhado, salas de aula e laboratórios, da instituição de ensino o estudante precisa verificar se será necessário a realização de alterações nesses computadores. Nesse caso o único requisito é a instalação da linguagem de programação *Python* que é um programa nativo do sistema operacional *Linux*, mas precisa ser instalado em computadores com o sistema operacional *Windows*. As alterações nos computadores devem ser solicitadas a equipe de TI que pode atender ou não a solicitação levando em consideração particularidades de cada solicitação. Por não existir uma garantia na possibilidade de realizar essas modificações nos computadores o estudante resolve usar a plataforma PESC para executar a sua simulação.

Passo 4: Execução na plataforma PESC

Com a plataforma PESC instalada e configurada o estudante pode, de forma autônoma, criar o ambiente de execução necessário para a sua simulação e executar seu código nos clientes conectados à plataforma.

Considerando que a linguagem *Python* é padrão na plataforma o estudante apenas precisa criar o Processo e a Requisição, Figura 26(a) e 26(b), e ao final deste processo o estudante consegue de forma autônoma, sem necessidade de realizar alterações no seu código, executar sua simulação em um computador de sua instituição de ensino.



(a) Formulário do Processo (b) Formulário da Requisição

Figura 26 – Processo de execução de geração de números aleatórios (o autor)

Caso o estudante queira gerar uma quantidade maior de números aleatórios com o mesmo código basta informar no formulário de Requisição a quantidade de repetições desejadas. Para cada repetição o código será executado gerando 20.000.000 de números aleatórios que no final do processo são recebidos pelo usuário em um arquivo de texto com as saídas de todas as execuções concatenadas. Em uma simulação para gerar 100.000.000 o código foi executado na máquina do estudante em um tempo total de $730,54$ segundos, criando uma requisição com 5 repetições na plataforma obteve um tempo total de 143 segundos. A distribuição da execução pelos clientes está apresentado na Figura 27.

```
  id   | rank | client_id |    date_time_client_start     |     date_time_client_end      | status
-------+------+-----------+-------------------------------+-------------------------------+--------
 26695 |    0 |        11 | 2022-08-02 02:47:53.057683+00 | 2022-08-02 02:49:36.941747+00 |      3
 26696 |    1 |        12 | 2022-08-02 02:47:55.556123+00 | 2022-08-02 02:49:55.990223+00 |      3
 26697 |    2 |        11 | 2022-08-02 02:48:01.320114+00 | 2022-08-02 02:49:51.641348+00 |      3
 26698 |    3 |        10 | 2022-08-02 02:48:03.725083+00 | 2022-08-02 02:49:39.744633+00 |      3
 26699 |    4 |        12 | 2022-08-02 02:48:15.031234+00 | 2022-08-02 02:50:16.246821+00 |      3
(5 rows)
```

Figura 27 – Exemplo 1 - Execução dos processos (o autor)

Considerando a Lei de Amdahl, como mostrado na Seção 2.2.3, podemos calcular o

speedup esperado entre a implementação paralelizada deste exempo e a sua implementação sequencial, considerando que nesse exemplo 99% do código pode ser paralelizado, como demonstrado a seguir:

$$T_{paralelo\_esperado} = (1 - f) * \frac{T_{sequencial}}{N} + f * T_{sequencial}$$

$$T_{paralelo\_esperado} = 0,99 * \frac{730,54}{5} + 0,01 * 730,54$$

$$T_{paralelo\_esperado} = 151,95$$

logo,

$$S_{esperado} = \frac{T_{sequencial}}{T_{paralelo\_esperado}}$$

$$S_{esperado} = \frac{730,54}{143}$$

$$S_{esperado} = 4,81$$

O speedup obtido por essa execução paralela está calculado a seguir:

$$S_{obtido} = \frac{T_{sequencial}}{T_{paralelo}}$$

$$S_{obtido} = \frac{730,54}{143}$$

$$S_{obtido} = 5,11$$

Considerando que a plataforma PESC distribui os processos em clientes com caracetrísticas diferentes houve uma melhoria do valor do *speedup* obtido em relação ao esperado, e essa melhoria representa um ganho de 80% em relação ao tempo de execução da versão sequencial. O código também foi executado de forma sequêncial nos mesmos clientes que receberam as instâncias do processo para executar através da plataforma PESC. Baseado nesses valores obtidos o tempo de execução paralelo esperado e o ganho de *speedup* esperado foram calculados em relação ao tempo da execução sequêncial na máquina do usuário e estão apresentados na Tabela 4 onde é possível perceber que o valor de *speedup* obtido $(5,11)$ está entre o intervalo do maior $(5,49)$ e menor $(4,94)$ valor esperado.

Cliente ID	Tempo Seq	Tempo Paralelo Esperado	Speedup Esperado
25	710.33	147.75	4.94
26	710.33	147.75	4.94
27	638.77	132.86	5.49

Tabela 4 – Speedup esperado

4.4.2 Exemplo 2 - Desempenho - Calcular a área de uma integral

Passo 1: Código do usuário

Um estudante do curso de estatística quer desenvolver um programa na linguagem de programação *Python* para calcular a integral definida de uma função apesar de existirem diversos *frameworks* que disponibilizam essa funcionalidade. Para realizar o cálculo da integral sem ter que conhecer previamente a antiderivada da função que será integrada o estudante utilizou um método numérico conhecido como regra dos trapézios. A regra dos trapézios consiste em aproximar o gráfico da função a ser integrada por uma quantidade definida de segmentos de retas e calcular a área dos trapézios formada por esse segmento de reta e a respectiva base no eixo X. A soma de todas as áreas dos trapézios entre os pontos a e b será o valor aproximado da integral definida da função entre esses pontos. A função a ser integrada pelo estudante é o $seno(x)^2$ e a representação da aproximação da função por trapézios é apresentada na Figura 28. Para a definição da acurácia foi calculado a diferença entre a soma da área dos retângulos a partir o ponto atual ($index$), e a soma da área do retângulo a partir do próximo ponto ($index + 1$), quando a quantidade de retângulos aumentar a tendência é essa diferença diminuir tendendo para zero.

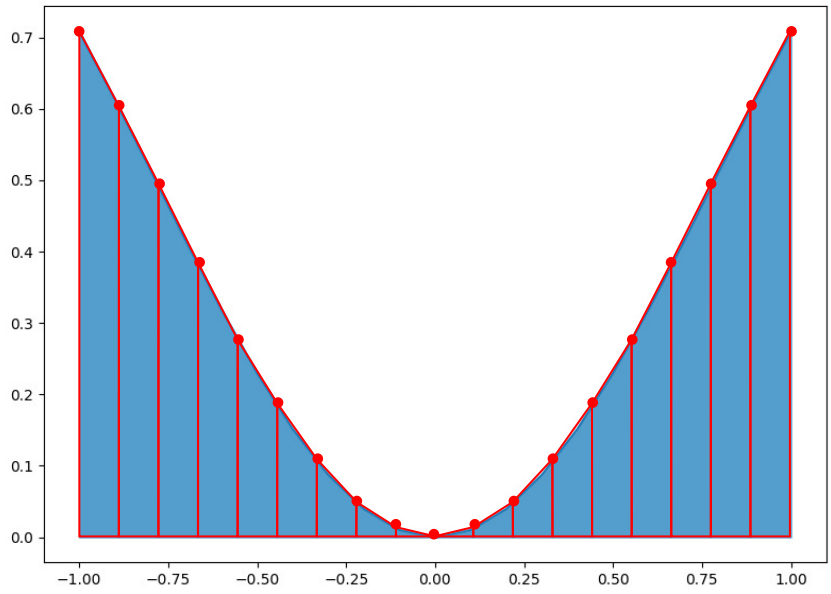


Figura 28 – Aproximação do gráfico da função $seno(x)^2$ por segmentos de retas (o autor)

A versão final do código desenvolvido pelo estudante está apresentado no Algoritmo 4.6.

```
import numpy as np
from numpy import sin
import time

def funcao(x):
    return sin(x)**2

def integrar(funcao, lim_inferior, lim_superior, precisao):
    function = funcao
    integral = erro = soma_area_superior = soma_area_inferior = 0

    if lim_inferior > lim_superior:
        raise ValueError("O limite inferior deve ser maior que o superior")

    pontos = np.linspace(lim_inferior, lim_superior, int(precisao))
    for index in range(len(pontos) - 1):
        delta = pontos[index + 1] - pontos[index]
        y_1 = function(pontos[index])
        y_2 = function(pontos[index + 1])
        area_trapezio = (y_2 + y_1)* delta/2
        integral += area_trapezio
        area_inferior = y_1 * delta
        soma_area_inferior += area_inferior
        area_superior = y_2 * delta
        soma_area_superior += area_superior

    erro = soma_area_superior - soma_area_inferior
    return integral, erro

```

```
30 if __name__ == "__main__":
31     resultado, acuracia = integrar(funcao, −1, 1)
32     print(f"{resultado}, {acuracia}")
```

Algoritmo 4.6 – Código para calcular integral de uma função

Esse código foi desenvolvido e executado em um notebook de uso particular do estudante e a instalação de todos os requisitos para executar o código foram realizados, nesse caso a instalação do *Python* e da biblioteca *Numpy*.

Passo 2: Conversão em uma simulação computacional

A partir do resultado obtido o estudante resolve executar o código para diversos valores usados no parâmetro precisão, que define a quantidade de pontos usados no cálculo das áreas na função integrar. Nessa etapa a função integrar será chamada variando o parâmetro precisão de 1 a 1.000.000 em intervalos de 1000 em 1000. As alterações necessárias para executar essa simulação impactam apenas o método *main* e estão apresentadas no Algoritmo 4.7.

```
1 if __name__ == "__main__":
2     inicio = time.time()
3     for indice, precisao in enumerate(range(1, 1000000, 1000)):
4         resultado, acuracia = integrar(funcao, −1, 1, precisao=precisao)
5         print(f"{indice}, {precisao}, {resultado}, {acuracia}")
6     fim = time.time()
7     print(f"Tempo de execução: {fim−inicio}segs")
```

Algoritmo 4.7 – Alterações no método main

Com essa alteração o código executou com sucesso e apresentou uma lista de valores que representam o índice da simulação e os valores retornados pela função integrar, sendo eles o resultado do cálculo e da acurácia. Além do resultado também foi apresentado ao final da simulação o tempo total de execução de 4333.06 segundos, aproximadamente 72 minutos. Considerando que esse processo vai ser repetido para outras funções e que o tempo de execução impactará na execução de novas simulações o estudante resolve executar o código em um dos computadores de sua instituição de ensino.

Passo 3: Execução em ambiente compartilhado

De forma similar ao exemplo anterior modificações nos computadores podem ser necessárias para a execução do código no ambiente compartilhado da instituição de ensino e devem ser solicitadas a equipe de TI. Nesse caso além do *Python* será necessário a instalação do pacote *Numpy*, que não causa impactos diretos na configuração do sistema

operacional, mas dependendo da versão desejada pode causar conflitos com instalações realizadas previamente.

Passo 4: Preparação do ambiente de execução na plataforma PESC

O código do estudante tem como pré-requisitos para a execução a linguagem de programa *Python* e a biblioteca *Numpy* e para executar esse código na plataforma PESC é necessário preparar um ambiente de execução para ele. O estudante pode verificar se já existe algum domínio que atenda a suas necessidades na lista de domínios ou criar um novo a partir de um ambiente previamente definido (SimplePython) na loja, como apresentado na Figura 29.

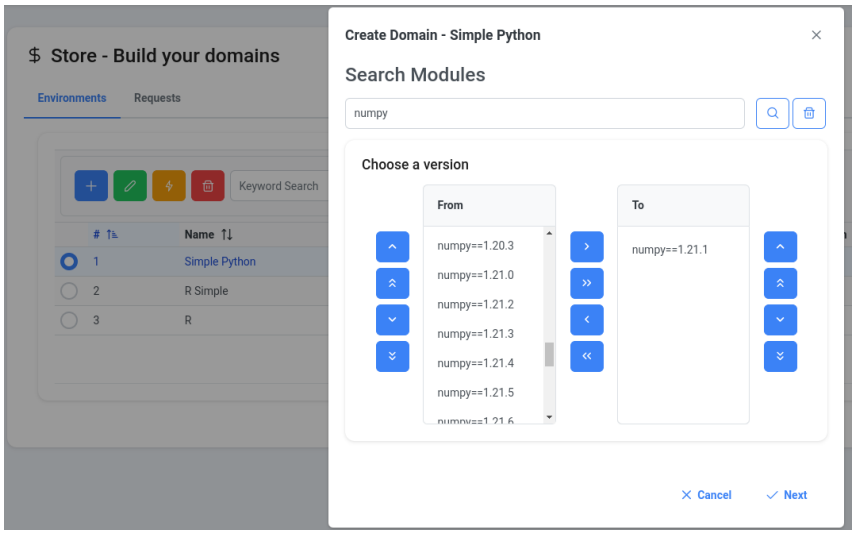


Figura 29 – Criação de um domínio a partir da loja (o autor)

Com o ambiente configurado o estudante pode criar o processo e a requisição de execução de forma similar ao apresentado no passo 4 do exemplo 1. O uso da plataforma PESC dessa forma garante que o programa do estudante será executado em algum dos computadores conectado como cliente, mas as configurações do equipamento que vai efetivamente executar o código do estudante pode ser similar às configurações do notebook pessoal usado no passo 2 e o tempo total da execução poderá não ser muito diferente do resultado obtido anteriormente.

Uma outra possibilidade que pode impactar no tempo de execução é o uso compartilhado do cliente, que recebeu a requisição de execução, por outros usuários o que faz com que o Client Module reduza os recursos computacionais disponível para as execuções da plataforma PESC.

Passo 5: Modificar o código para execução paralela

Entendendo que a plataforma PESC permite que um código sequencial pode ser executado de forma paralela o estudante modifica seu código para tentar diminuir o tempo total de execução. As alterações são relacionadas a quantidade de execuções paralelas que o programa vai realizar e o parâmetro *rank*, recebido pelo código durante o processo de execução, será usado para essa definição. Cada execução individual da função integrar executa relativamente rápido no computador do estudante, entre 0.01 e 3.5 segundos, e para não ter um rank associado a cada chamada da função integrar o estudante ajusta o código para que 10 processos estejam em execução paralela, como apresentado no Algoritmo 4.8.

```
if __name__ == "__main__":

    # Receber os parâmetros na chamada de execução da plataforma
    import argparse
    parser=argparse.ArgumentParser()
    parser.add_argument("--app_dir","-a",default=f"{os.getcwd()}/")
    parser.add_argument("--checkpoint_dir","-c",default=f"{os.getcwd()}/checkpoint")
    parser.add_argument("--output_dir","-o",default=f"{os.getcwd()}/output")
    parser.add_argument("--rank","-r",type=int, default=0)
    parser.add_argument("--world_size","-w",type=int, default=1)
    parser.add_argument("--master_addr","-d",default="127.0.0.1")
    parser.add_argument("--master_port","-t",type=int, default=9000)
    parser.add_argument("--parameters","-p",nargs="*",default=[])
    args=parser.parse_args()

    # Definir os limites superior e inferior baseados no rank, se o rank for 1 executar entre
     100.000 e 200.000
    limite_inferior = args.rank*100000
    limite_superior = limite_inferior+100000

    inicio = time.time()
    # Usar os limites na definição do range das precisões
    for indice, precisao in enumerate(range(limite_inferior, limite_superior, 1000)):
        resultado, acuracia = integrar(funcao, -1, 1, precisao=precisao)
        print(f"{indice}, {precisao}, {resultado}, {acuracia}")
    fim = time.time()
    print(f"Tempo de execução: {fim-inicio}segs")
```

Algoritmo 4.8 – Alterações no método *main* para execução paralela

Após essa alteração e a criação de um novo Processo, o estudante criou uma requisição de execução com 10 repetições. Cada instância da execução recebeu um rank e calculou os limites iferiores e superiores de acordo com a Tabela 5.

rank	limite_inferior	limite_superior
0	0	100000
1	100000	200000
2	200000	300000
3	300000	400000
4	400000	500000
5	500000	600000
6	600000	700000
7	700000	800000
8	800000	900000
9	900000	1000000

Tabela 5 – Ajustes das variáveis de acordo com o rank

Cada instância do processo é executada de forma independente das outras e podem ser distribuídas para clientes diferentes, como apresentado na Figura 30 e dessa forma otimizar a execução do código que anteriormente era executado de forma sequêncial.

```
  id   | rank | client_id |    date_time_client_start     |     date_time_client_end      | status
-------+------+-----------+-------------------------------+-------------------------------+--------
 26660 |    0 |        12 | 2022-07-25 19:11:10.874673+00 | 2022-07-25 19:11:56.614562+00 |      3
 26661 |    1 |        11 | 2022-07-25 19:11:16.657394+00 | 2022-07-25 19:12:38.48694+00  |      3
 26662 |    2 |        10 | 2022-07-25 19:11:20.2988+00   | 2022-07-25 19:13:32.666808+00 |      3
 26663 |    3 |        12 | 2022-07-25 19:11:31.168183+00 | 2022-07-25 19:16:57.979402+00 |      3
 26664 |    4 |        11 | 2022-07-25 19:11:27.538404+00 | 2022-07-25 19:15:32.578954+00 |      3
 26665 |    5 |        10 | 2022-07-25 19:11:31.767298+00 | 2022-07-25 19:16:15.839739+00 |      3
 26666 |    6 |        12 | 2022-07-25 19:11:47.305506+00 | 2022-07-25 19:21:49.972113+00 |      3
 26667 |    7 |        11 | 2022-07-25 19:11:54.598246+00 | 2022-07-25 19:18:54.821882+00 |      3
 26668 |    8 |        10 | 2022-07-25 19:11:58.987657+00 | 2022-07-25 19:19:31.067202+00 |      3
 26669 |    9 |        12 | 2022-07-25 19:12:07.594334+00 | 2022-07-25 19:26:32.998525+00 |      3
(10 rows)
```

Figura 30 – Exemplo 2 - Execução dos processos (o autor)

O tempo total de execução foi reduzido para 15 minutos representando um ganho real de 80% em relação ao tempo de execução da versão sequêncial. Esses ajustes são mínimos e não representaram uma alteração significativa no código do estudante e não foi necessário conhecer os recursos específicos de programação paralela da linguagem na obtenção desse ganho real.

5 Testes e Validação da Plataforma

Avaliar a plataforma PESC é fundamental para garantir a sua correta operação e medir o ganho em tempo de execução com a sua adoção.

5.1 Ambiente de avaliação

Para estar alinhado com os princípios da plataforma PESC, o ambiente de avaliação emula um pequeno laboratório com um servidor e seis computadores clientes. A Tabela 6 apresenta os requisitos de software do ambiente de avaliação, onde todas os computadores utilizam o Ubuntu 18.04.6 LTS (*Bionic Beaver*) como sistema operacional e 500 GiB de espaço de armazenamento em disco. As outras configurações dos computadores são mostradas na Tabela 7.

Tipo	**Banco de dados**	**Pacotes principais**
Gerente	PostgreSQL 12	Django, DRF
Cliente	SQLite 3	Flask, SQLAlchemy

Tabela 6 – Requisitos de software

Cliente ID	**RAM**	**Processador**
Cliente 7 e 8	32 GB	Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
Cliente 9	16 GB	Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz;
Cliente 10 and 11	32 GB	Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
Cliente 12	8 GB	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz

Tabela 7 – Configurações dos computadores clientes

5.2 Execução da avaliação

Para avaliar a plataforma PESC foram criados vários cenários e em cada um a plataforma deve executar as solicitações do usuário e finalizar com os resultados esperados. Os cenários selecionados estão apresentados a seguir:

- Executar com sucesso em um computador cliente;
- Executar com sucesso em mais de um computador cliente;
- Executar com sucesso após falha em um computador cliente;

- Executar com sucesso um processo que usa um *framework* com recursos paralelos nativos;
- Executar e gerenciar com sucesso o compartilhamento de tempo de GPU.

5.2.1 Cenário 1 - Executar com sucesso em um computador cliente

Neste cenário um usuário precisa executar uma simulação, mas não pode fazer alterações no sistema operacional do computador para instalar os softwares necessários. A plataforma PESC será usada como uma abstração para o uso de contêineres, sendo esse o cenário de uso mais básico para ela. O código do usuário é um algoritmo de k-vizinhos mais próximos (k-NN) (CUNNINGHAM; DELANY, 2021) escrito em *Python*, usando os recursos de biblioteca *Scikit-learn* (PEDREGOSA et al., 2011), que realiza a tarefa de classificar o conjunto de dados de dígitos MNIST (DENG, 2012) convertido para o formato CSV. Uma parte do código do usuário é apresentado no Algoritmo 5.1.

```
1  ...
2  # Cabeçalho dos parâmetros
3  ...
4  train=pd.read_csv('./mnist_test.csv')
5  test=pd.read_csv('./mnist_train.csv')
6  X_train=train.drop('label',axis=1)
7  y_train=train['label']
8  X_test=test.drop('label',axis=1)
9  y_test=test['label']
10
11 for rank in range(1,upper_limit+1):
12   model=KNeighborsClassifier(
13     n_neighbors=rank
14   )
15   model.fit(X_train,y_train)
16   accuracy=model.score(
17     X_test,y_test
18   )
19   print(f'k={rank}==>{accuracy}')
```

Algoritmo 5.1 – Cenário 1 - KNN Sequêncial

Os requisitos necessários para este cenário são o *Dockerfile*, o *requirements.txt*, o código de usuário e o arquivo *CSV* com a base de dígitos. O usuário cria o Domínio com os dois primeiros arquivos e o processo com o código do usuário. O banco de dados de teste e treinamento foram criados em um processo anterior a essa execução e serão enviados para a plataforma PESC como arquivos compartilhados, dessa forma eles poderão ser

usados em outros cenários de validação apresentados nessa seção sem a necessidade de transferir o arquivo novamente para os clientes que já participaram da execução deste cenário. A requisição foi realizada com os seguintes parâmetros informados no formulário de requisição:

- Domain: Cenário_1
- Process: Validação_1
- Repetitions: 1
- Parameters: 10
- Shared Files: minst_test, mnist_train
- Rooms: General

O valor 10 passado como parâmetro é usado como limite superior (*upper_limit*) do *loop*, linha 11 do Algoritmo 5.1. O código é executado como esperado e retorna o arquivo de saída com o resultado da acurácia encontrada para cada valor de k e está apresentada na Figura 31:

```
k=1==>0.9416833333333333
k=2==>0.93195
k=3==>0.9428333333333333
k=4==>0.94115
k=5==>0.9425166666666667
k=6==>0.9404333333333333
k=7==>0.9401333333333334
k=8==>0.93905
k=9==>0.9380166666666667
k=10==>0.9371666666666667
```

Figura 31 – Cenário 1 - Arquivo de saída (o autor)

Para este cenário foi validado também a execução de um *script* em *R* e um código Java. Nesses casos o usuário deve informar no momento da criação do processo qual a linguagem de programação utilizada e a plataforma fará a chamada do código do usuário da forma definida para a tecnologia. Para executar códigos escritos em *Java* é necessário que seja feito o *upload* do arquivo no formato *jar* executável no momento da criação da processo. As duas simulações foram executadas e retornaram os arquivos de saída esperados.

5.2.2 Cenário 2 - Executar com sucesso em mais de uma computador cliente

Neste cenário o código usado no cenário anterior será alterado para executar cada valor de k em um contêiner diferente. Desta forma, um código escrito para ser executado

de forma sequencial será executado em paralelo com a realização de alterações mínimas no código do usuário, apresentado no Algoritmo 5.2. Neste caso a plataforma é utilizada de forma mais otimizada que no cenário anterior.

```
...
# Cabeçalho dos parâmetros
...
train=pd.read_csv('./mnist_test.csv')
test=pd.read_csv('./mnist_train.csv')
X_train=train.drop('label',axis=1)
y_train=train['label']
X_test=test.drop('label',axis=1)
y_test=test['label']

#for rank in range(1,upper_limit+1):
model=KNeighborsClassifier(
  n_neighbors=rank #rank is received by parameter
)
model.fit(X_train,y_train)
accuracy=model.score(X_test,y_test)
print(f'k={rank}==>{accuracy}')
```

Algoritmo 5.2 – Cenário 2 - KNN Paralelo

Neste caso o limite superior não será usado para definir um *loop* sequencial, mas o código será executado a quantidade de vezes informada no campo repetições no formulário de requisição. Cada instância do código receberá uma identificação diferente (valor do *rank*) que será usada como o valor de k na classificação, como mostrado na Figura 32. A requisição foi realizada com os seguintes parâmetros informados no formulário de requisição:

- Domínio: Cenário_2
- Process: Validação_2
- Repetitions: 10
- Shared Files: minst_test, mnist_train
- Rooms: General

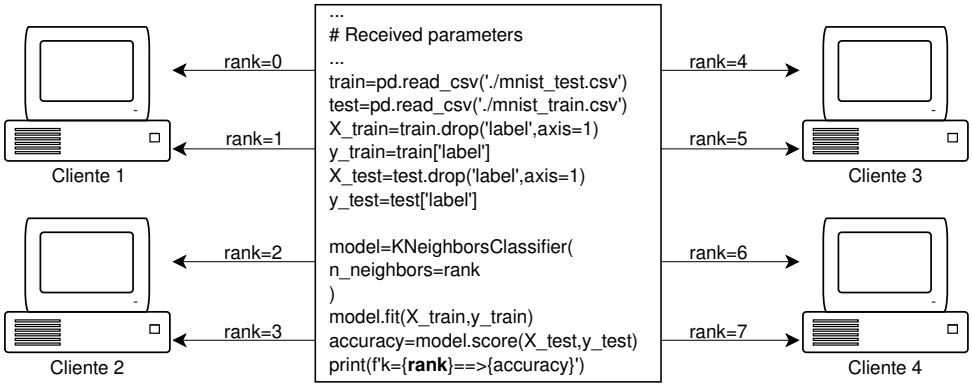


Figura 32 – Cenário 2 - KNN Paralelo (o autor)

O código é executado como esperado e retorna os arquivos de saída de cada cliente com o resultado encontrado para o valor k executado em cada um. Nesse caso os arquivos de saída são concatenados em um único arquivo para facilitar a análise dos resultados pelo usuário que solicitou a execução.

Para validar o ganho usando a plataforma PESC neste cenário, o código dos processos Validação_1 e Validação_2 foram executados várias vezes para os valores de k = 1, 5, 10, 15 e 20 e o tempo de execução foram comparados, como mostrado nas Tabelas 8 e 9.

K	Início	Fim	Segundos
1	13:48:42	13:48:59	17
5	13:50:25	13:51:44	79
10	13:56:19	13:59:00	161
15	14:08:03	14:12:06	243
20	14:13:57	14:19:22	325

Tabela 8 – Tempo de execução - Cenário 1

K	Início	Fim	Segundos
1	14:32:02	14:32:20	18
5	14:33:55	14:34:56	61
10	14:39:23	14:40:47	84
15	14:48:04	14:49:35	91
20	14:53:26	14:54:59	93

Tabela 9 – Tempo de execução - Cenário 2

O tempo para construir a imagem do *Docker* não foi considerado e o ganho com as alterações realizadas no código para este cenário é mostrado na Figura 33. O gráfico apresenta o comportamento esperado pela Lei de Amdahl, quando o aumento da quantidade de núcleos de execução melhora o tempo de execução, mas não de forma linear.

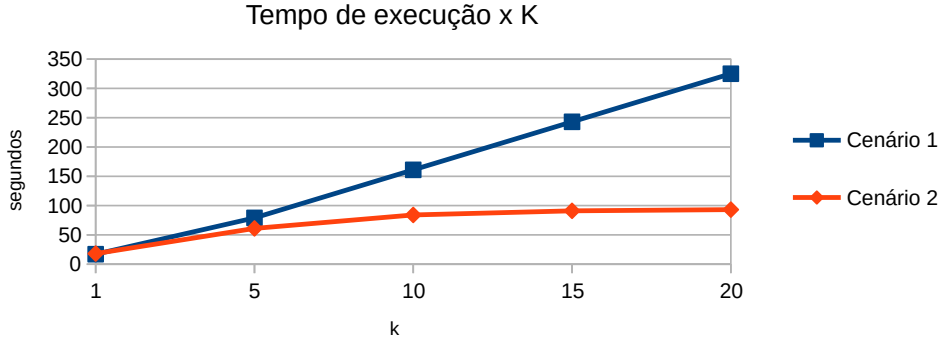


Figura 33 – Comparação entre os tempos de execução dos cenários 1 e 2 (o autor)

5.2.3 Cenário 3 - Executar com sucesso após falha no cliente

Neste cenário a mesma requisição feita no cenário anterior será repetida, mas dois clientes serão desconectados após receberem a instância do processo para executar. Depois que a instância do processo é enviada para os clientes o gerente monitora seu *status* e em caso de falha na comunicação com um dos cliente o processo recebido por ele é cancelado e redistribuído para outro cliente. O código funciona como esperado e os processos redistribuídos podem ser verificados na tabela do banco de dados apresentada na Figura 34. O usuário pode verificar esse comportamento através da interface *web* da plataforma.

Id	rank	client_id	status
23547	0	11	Success
23548	1	12	Success
23549	2	8	Success
23550	3	10	Canceled
23557	3	12	Success
23558	4	8	Success
23551	4	7	Canceled
23552	5	12	Success
23553	6	8	Success
23554	7	7	Canceled
23559	7	11	Success
23555	8	11	Success
23556	9	11	Success

Figura 34 – Cenário 3 - Redistribuição de tarefas (o autor)

É possível perceber que o processo executado com id 23550 e $rank = 3$ foi cancelado no cliente com $id = 10$, mas o mesmo $rank$ concluiu com sucesso no cliente com $id = 12$. O mesmo aconteceu com os processos com ids iguais a 23558 e 23554. Esta é uma garantia de que enquanto existir clientes conectados e com disponibilidade de recursos o código do usuário será executado com sucesso e a requisição será concluída sem qualquer iteração do usuário.

Em caso de falha do gerente os clientes continuam executando as instâncias dos processos que foram recebidas e enviarão o *status* da execução quando o gerente estiver novamente disponível e uma conexão for realizada entre eles. Esse processo não altera o fluxo inicial de execução após a distribuição das instâncias do processo.

5.2.4 Cenário 4 - Executar com sucesso um processo que usa um *framework* com recursos paralelos nativos

Alguns *frameworks* oferecem recursos nativos para a programação paralela, como o *Pytorch Distributed RPC Framework* (DRF) (PYTORCH, 2022). Neste cenário um processo será executado, suas instâncias irão assumir diferentes papéis devido aos requisitos deste recurso. Este cenário foi construído com base no código do tutorial oficial do *PyTorch* (LI, 2002) onde um treinamento distribuído foi construído usando o pacote `torch.distributed.rpc`, neste caso apenas o aprendizado de reforço distribuído usando o código *RPC* e *RRef* foi usado. Neste cenário um cliente desempenhará o papel do agente, que coordena o envio de dados para outros clientes, que desempenharão o papel dos observadores. O agente tem um papel central e os observadores precisam identificar este cliente para que se estabeleça uma comunicação entre eles. A plataforma PESC informa para cada instância do processo o endereço *IP* e a porta que foi criada para o cliente que recebeu o $rank = 0$, dessa forma as outras instâncias, $rank > 0$, podem se comunicar com essa instância e atender aos requisitos do recurso utilizado no *DRF*. No código do usuário é verificado através do parâmetro *rank* qual o *id* da instância em execução e os ajustes necessários são feitos em relação a isso, como mostrado no Algoritmo 5.3. A requisição foi feita com os seguintes parâmetros informados no formulário de requisição:

- Domínio: Cenário_4
- Process: Validação_4
- Parallel: True
- World Size: 3
- Rooms: General

```
...
# Cabeçalho dos parâmetros
...
def run_worker(
    rank,
    world_size,
    gamma,
    log_interval
):
    if rank == 0: # rank0 is the agent
        rpc.init_rpc(
            AGENT_NAME,
            rank=rank,
            world_size=world_size
        )
        agent = Agent(world_size, gamma)
```

```
17          for i_episode in range(1, 100):
18              agent.run_episode()
19              last_reward = (
20                  agent.finish_episode()
21              )
22              ...
23      else:
24          # other ranks are the observers
25          rpc.init_rpc(
26              OBSERVER_NAME.format(rank),
27              rank=rank,
28              world_size=world_size
29          )
30      rpc.shutdown()
31  ...
```

Algoritmo 5.3 – Cenário 4 - Pytorch Distributed RPC Framework

O código é executado como esperado onde o valor de *World Size* passado como parâmetro no formulário da requisição determina a quantidade de instâncias que serão executadas. Para evitar que os processos comecem de forma dessincronizada, também é informado que esta solicitação usará recursos paralelos. Esta informação fará com que todos os clientes que receberam uma instância do processo esperem por um sinal do gerente antes de iniciar a execução. Isso ajuda a minimizar que recursos sejam alocados em contêineres que não estejam efetivamente executando seus processos.

5.2.5 Cenário 5 - Executar e gerenciar com sucesso o compartilhamento de tempo de GPU

Neste cenário um usuário precisa executar uma simulação que faz uso de *GPU* como recurso computacional e desenvolve a simulação na linguagem de programação *C* com uso da biblioteca CUDA. O programa é uma solução para o problema que calcula o caminho com custo mínimo entre os vértices de um grafo, algoritmo de Dijkstra (JAVAID, 2013), e será iniciado várias vezes com uma semente aletória. Para testar o gerenciamento do uso das GPUs disponíveis nos Clientes essa simulação será iniciada 30 vezes nos clientes 7, 8 e 10 que apresentam *GPUs* com características diferentes, como mostrado na Tabela 10. Apesar dos Clientes 7 e 8 possuirem duas *GPUs* a versão atual da plataforma PESC gerencia e monitora apenas uma *GPU* por cliente e por isso elas foram omitidas da Tabela 10.

Por ter uma quantidade maior de memória RAM é esperado neste cenário que

Cliente ID	GPU	RAM
Cliente 7 e 8	GeForce GTX 690	2GB
Cliente 10	TITAN V	12GB

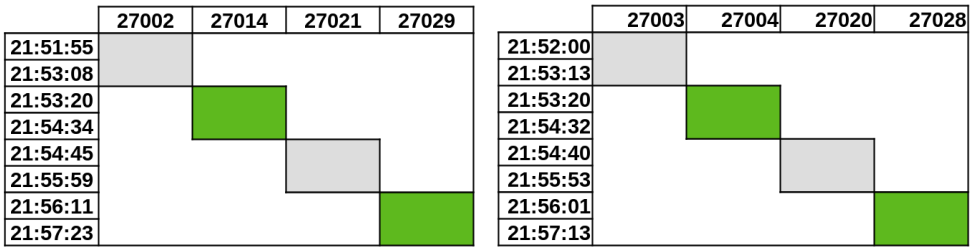
Tabela 10 – Configurações das GPUs dos computadores clientes

o Cliente 10 receba uma quantidade maior de instâncias de processo para executar. A simulação ocupa aproximadamente 1328MB de memória da *GPU* e nos Clientes 7 e 8 não teremos compartilhamento de tempo de *GPU* pelas instâncias do processo, tanto pela quantidade de memória disponível quanto pela versão do *driver* que não permite consultar a quantidade de memória da *GPU* usada por um processo como foi apresentado na Seção 4.2.1. O código é executado como esperado e os processos distribuídos podem ser verificados na tabela do banco de dados apresentada na Figura 35.

id	client_id	process_id	process_request_id	rank	max_gpu_memory	status	start	end
27002	7	27	260	0	0	3	21:51:55.918	21:53:08.096
27014	7	27	260	12	0	3	21:53:20.959	21:54:34.134
27021	7	27	260	19	0	3	21:54:45.941	21:55:59.278
27029	7	27	260	27	0	3	21:56:11.187	21:57:23.041
27003	8	27	260	1	0	3	21:52:00.331	21:53:13.458
27004	8	27	260	2	0	3	21:53:20.442	21:54:32.482
27020	8	27	260	18	0	3	21:54:40.909	21:55:53.099
27028	8	27	260	26	0	3	21:56:01.135	21:57:13.148
27005	10	27	260	3	1393557503	3	21:52:10.558	21:52:38.233
27006	10	27	260	4	1393557503	3	21:52:25.146	21:53:15.507
27007	10	27	260	5	1393557503	3	21:52:35.354	21:53:36.746
27008	10	27	260	6	1393557503	3	21:52:49.538	21:54:02.171
27009	10	27	260	7	1393557503	3	21:53:03.867	21:54:17.628
27010	10	27	260	8	1393557503	3	21:53:20.906	21:54:33.304
27011	10	27	260	9	1393557503	3	21:54:08.189	21:55:18.608
27012	10	27	260	10	1393557503	3	21:53:41.297	21:54:54.392
27013	10	27	260	11	1393557503	3	21:54:29.027	21:55:43.140
27015	10	27	260	13	1393557503	3	21:54:31.813	21:55:44.464
27016	10	27	260	14	1393557503	3	21:55:07.976	21:56:14.087
27017	10	27	260	15	1393557503	3	21:55:30.012	21:56:40.805
27018	10	27	260	16	1393557503	3	21:55:50.887	21:57:14.377
27019	10	27	260	17	1393557503	3	21:55:54.545	21:57:17.093
27022	10	27	260	20	1393557503	3	21:57:26.244	21:58:36.430
27023	10	27	260	21	1393557503	3	21:57:46.940	21:59:02.075
27024	10	27	260	22	1393557503	3	21:58:37.118	21:59:29.973
27025	10	27	260	23	1393557503	3	21:58:41.025	21:59:32.177
27026	10	27	260	24	1393557503	3	21:56:17.823	21:57:34.357
27027	10	27	260	25	1393557503	3	21:56:21.835	21:57:37.880
27030	10	27	260	28	1393557503	3	21:57:22.643	21:58:32.321
27031	10	27	260	29	1393557503	3	21:57:44.019	21:58:59.236

Figura 35 – Cenário 5 - Redistribuição de tarefas para as GPUs (o autor)

É possível perceber que as instâncias do processo executadas nos Clientes 7 e 8 não são executados de forma paralela e esse comportamento está apresentado na Figura 36, já as instâncias executadas no Cliente 10 compartilham o tempo de *GPU* dentro do limite definido no arquivo de configuração do cliente e este comportamento está apresentado na Figura 37 que apresenta a distribuição das 10 primeiras instâncias executadas por esse Cliente.



(a) Cliente 7 (b) Cliente 8

Figura 36 – Execuções sequênciais na GPU (o autor)

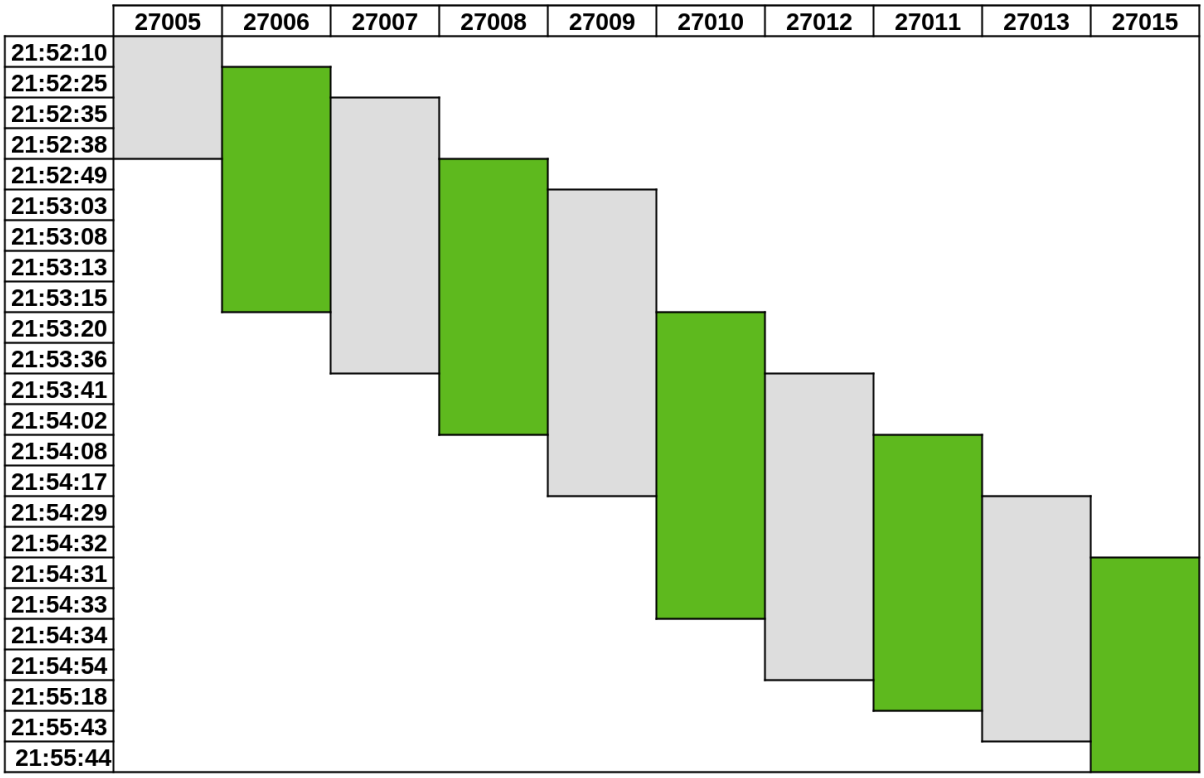


Figura 37 – Cenário 5 - Execuções paralela na GPU (o autor)

5.3 Resultados Experimentais

Embora os testes de validação apresentem os resultados esperados, um caso de uso foi usado para avaliar o valor da plataforma para um usuário em uma situação real com uma demanda de execução que se encaixa no cenário 2, apresentado na Seção 5.2. Neste expermento, uma simulação de passeio aleatório usando algoritmos de computação quântica deve ser executada com uma alta taxa de repetição. Essa taxa de repetição faz parte da natureza desta simulação, que deve gerar uma grande quantidade de dados para análise e confirmação de seus pressupostos. Os experimentos são divididos em três partes e em uma delas são realizadas 1200 simulações. O uso da plataforma simplificou o processo

de execução das simulações, gerenciando o *status* e o ciclo de vida das instâncias do processo, reiniciando instâncias e movendo instâncias para outros clientes nos casos onde algum tipo de falha seja detectada. Dessa forma, foi possível executar 1200 repetições da simulação no mesmo ambiente utilizado para os testes de validação, apresentados na Seção 5.1, mas neste caso apenas quatro clientes cadastrados participaram desta execução. Os clientes não são usados exclusivamente pela plataforma PESC e podem estar competindo por recursos computacionais com outros sistemas, além disso as configurações do cliente não são as mesmas, e portanto o valor mostrado na coluna de duração média na Tabela 11 é diferente para cada cliente, o que afeta a distribuição da quantidade de instâncias recebidas pelos clientes.

Cliente ID	**Duração Média**	**Qtd Ranks**
Cliente 7	01:20:38.42278	207
Cliente 8	01:22:21.366742	202
Cliente 9	00:31:24.892872	567
Cliente 12	01:01:13.74276	224
Total		**1200**

Tabela 11 – Distribuição de instâncias em nós clientes

Considerando que o menor tempo médio para a execução das instâncias foi 00:31:24s, se todas as 1200 repetições sequenciais, formato original do código, fossem executadas nesse cliente o tempo de execução seria de aproximadamente 600 horas. O tempo total da execução na plataforma PESC foi de aproximadamente 12:39:14s, considerando o momento em que a primeira instância inicia sua execução e a última termina, caracterizando um ganho de tempo em relação a execução sequencial de aproximadamente 98%.

6 Considerações Finais

6.1 Conclusões

A necessidade de acesso a recursos computacionais cria uma demanda na utilização de serviços de nuvens públicas, mas esses recursos nem sempre estarão realmente disponíveis para instituições pública de ensino em países em desenvolvimento. Apresentamos uma forma de aproveitar recursos ociosos em computadores *desktops* dessas instituições sem a necessidade de ter equipes de TI especializadas para manter essa solução e sem a necessidade de impor mudanças significativas nos códigos dos usuários que serão executados através dela. A autonomia dada ao usuário através de sua interface *web* permite que os ambientes de execução sejam criados sem a intervenção direta de equipes de TI e que pesquisadores de qualquer área se beneficiem na adoção da plataforma. Também foi demonstrado que usuários sem conhecimento especializado em computação paralela podem se beneficiar através de ajustes mínimos em seu código que mesmo sendo desenvolvidos para serem executados de forma sequencial podem ser executados de forma paralela. Os resultados apresentados mostram um ganho substancial em relação ao tempo que seria dispêndio sem a adoção da plataforma e a simplicidade na instalação e manutenção do serviço permitem sua adoção independente da infraestrutura de TI apresentada o que seria um ganho para as instituições que possuem esses recursos ociosos e podem fazer uso dele de forma mais otimizada.

6.2 Trabalhos Futuros

A plataforma PESC deve continuar sendo desenvolvida continuamente, inclusive em parceria com a Pós-Graduação em Informática Aplicada, onde trabalhos de mestrado podem trazer novos elementos e funcionalidades para ela. Além disso a plataforma PESC deverá ser instalada e configurada para atender a demanda interna do Departamento de Estatísitca e Informática da UFRPE onde será analisada e ajustada de acordo com a demanda e o *feedback* dos usuários. Entre essas melhorias podemos citar:

- O monitoramento e gerenciamento de mais de uma *GPU* por Cliente;
- A criação de um novo nível de gerenciamento no papel de um Subgerente para

distribuir as instâncias dos processos em equipamentos geograficamente distribuídos;
- A possibilidade de criptografia da comunicação entre os módulos Clientes e Gerente;
- Permitir a compilação de programas do usuário na plataforma PESC não limitando ao envio do arquivo binário.

Após atingir um nível de maturidade, onde a plataforma estará estável e em produção, será expandido o seu uso para outras insitutições como o Campus Recife do Instituto Federal de Pernambuco. A partir dessas experiências de implantação e uso novos estudos serão criados e publicados para tornar a Plataforma PESC um produto de referência da Universidade Federal Rural de Pernambuco.

6.3 Produções

O desenvolvimento dessa plataforma como objeto de estudo dessa tese permitiu o desenvolvimento das seguintes produções:

- Publicação do *preprint* do artigo *PESC - Parallel Experiment for Sequential Code* no arxiv (SANTOS et al., 2023);
- Submissão do artigo entitulado *PESC – Parallel Experiment for Sequential Code* para o periódico *Journal of Parallel and Distributed Computing*;
- Registro do software "PESC Client - Parallel Experience for Sequencial Code Client"sob o processo número BR512022002108-0;
- Registro do software "PESC Server API - Parallel Experience for Sequencial Code Server API"sob o processo número BR512022002110-2.
- Produção do conteúdo, textos e vídeos, para a tranferência de tecnologia após implantação da plataforma PESC, como apresentado no Apêndice B.

Referências

ALFAILAKAWI, M. G.; ALJAME, M.; AHMAD, I. Parallel and distributed implementation of sine cosine algorithm on apache spark platform. **IEEE Access**, IEEE, v. 9, p. 77188–77202, 2021.

AMAZON. **Home Page**. 2022. <https://aws.amazon.com>.

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: **Proceedings of the April 18-20, 1967, spring joint computer conference**. [S.l.: s.n.], 1967. p. 483–485.

BAHAREVA, N.; USHAKOV, Y.; USHAKOVA, M.; PARFENOV, D.; LEGASHEV, L.; BOLODURINA, I. Researching a distributed computing automation platform for big data processing. In: IEEE. **2020 International Conference Engineering and Telecommunication (En&T)**. [S.l.], 2020. p. 1–5.

BAL, H. E.; STEINER, J. G.; TANENBAUM, A. S. Programming languages for distributed computing systems. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 21, n. 3, p. 261–322, 1989.

BHALLA, A. Various ways of parallelization of sequential programs. **INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY**, v. 3, n. 1, 2014. ISSN 2278-0181. Disponível em: <https://www.ijert.org/various-ways-of-parallelization-of-sequential-programs>.

BHARDWAJ, A.; KRISHNA, C. R. Virtualization in cloud computing: Moving from hypervisor to containerization—a survey. **Arabian Journal for Science and Engineering**, Springer, v. 46, n. 9, p. 8585–8601, 2021.

BOINC. **User manual**. 2022. <https://boinc.berkeley.edu/wiki/User_manual>.

BORLEA, I.-D.; IERCAN, D.; PRECUP, R.-E.; DRAGAN, F.; BORLEA, A.-B. Implementing a platform to run clustering algorithms using distributed computing. In: IEEE. **2019 IEEE 13th International Symposium on Applied Computational Intelligence and Informatics (SACI)**. [S.l.], 2019. p. 217–222.

BRESHEARS, C. **The art of concurrency: A thread monkey's guide to writing parallel applications**. [S.l.]: "O'Reilly Media, Inc.", 2009.

BROWNLEE, J. **Statistical methods for machine learning: Discover how to transform data into knowledge with Python**. [S.l.]: Machine Learning Mastery, 2018.

BUYYA, R.; VECCHIOLA, C.; SELVI, S. T. Chapter 3 - virtualization. In: BUYYA, R.; VECCHIOLA, C.; SELVI, S. T. (Ed.). **Mastering Cloud Computing**. Boston: Morgan Kaufmann, 2013. p. 71–109. ISBN 978-0-12-411454-8. Disponível em: <https://www.sciencedirect.com/science/article/pii/B9780124114548000036>.

CHOI, S.; KIM, H.; BYUN, E.; BAIK, M.; KIM, S.; PARK, C.; HWANG, C. Characterizing and classifying desktop grid. In: IEEE. **Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)**. [S.l.], 2007. p. 743–748.

CITO, J.; SCHERMANN, G.; WITTERN, J. E.; LEITNER, P.; ZUMBERI, S.; GALL, H. C. An empirical analysis of the docker container ecosystem on github. In: IEEE. **2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)**. [S.l.], 2017. p. 323–333.

COLAB, G. **Google Colab FAQ**. 2022. <https://research.google.com/colaboratory/faq.html>.

CUNNINGHAM, P.; DELANY, S. J. K-nearest neighbour classifiers - a tutorial. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 54, n. 6, jul 2021. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3459665>.

DENG, L. The mnist database of handwritten digit images for machine learning research. **IEEE Signal Processing Magazine**, IEEE, v. 29, n. 6, p. 141–142, 2012.

ECKERT, M.; MEYER, D.; HAASE, J.; KLAUER, B. Operating system concepts for reconfigurable computing: review and survey. **International Journal of Reconfigurable Computing**, Hindawi, v. 2016, 2016.

FLYNN, M. J. Very high-speed computing systems. **Proceedings of the IEEE**, IEEE, v. 54, n. 12, p. 1901–1909, 1966.

GOLDBERG, R. P. Survey of virtual machine research. **Computer**, v. 7, n. 6, p. 34–45, 1974.

GOOGLE. **Home Page**. 2022. <https://cloud.google.com/>.

IBRAHIM, D. **ARM-Based Microcontroller Multitasking Projects: Using the FreeRTOS Multitasking Kernel**. [S.l.]: Newnes, 2020.

IVASHKO, E.; CHERNOV, I.; NIKITINA, N. A survey of desktop grid scheduling. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 29, n. 12, p. 2882–2895, 2018.

JAVAID, A. Understanding dijkstra's algorithm. **Available at SSRN 2340905**, 2013.

KAHANWAL, D.; SINGH, D. T. et al. The distributed computing paradigms: P2p, grid, cluster, cloud, and jungle. **arXiv preprint arXiv:1311.3070**, 2013.

KHAN, M. K.; MAHMOOD, T.; HYDER, S. I. Scheduling in desktop grid systems: Theoretical evaluation of policies & frameworks. **International Journal of Advanced Computer Science and Applications**, The Science and Information Organization, v. 8, n. 1, 2017. Disponível em: <http://dx.doi.org/10.14569/IJACSA.2017.080117>.

KISS, T.; KACSUK, P.; KOVÁCS, J.; RAKOCZI, B.; HAJNAL, Á.; FARKAS, A.; GESMIER, G.; TERSTYANSZKY, G. Micado—microservice-based cloud application-level dynamic orchestrator. **Future Generation Computer Systems**, Elsevier, v. 94, p. 937–946, 2019.

KRAŠOVEC, B.; FILIPČIČ, A. Enhancing the grid with cloud computing. **Journal of Grid Computing**, Springer, v. 17, n. 1, p. 119–135, 2019.

KUMAR, V.; GRAMA, A.; GUPTA, A.; KARYPIS, G. **Introduction to parallel computing**. [S.l.]: Addison-Wesley Reading, MA, USA, 2003. v. 2.

LI, S. **Getting Started with Distributed RPC Framework**. 2002. Disponível em: <https://pytorch.org/tutorials/intermediate/rpc_tutorial.html>.

LI, Z.; CHARD, R.; WARD, L.; CHARD, K.; SKLUZACEK, T. J.; BABUJI, Y.; WOODARD, A.; TUECKE, S.; BLAISZIK, B.; FRANKLIN, M. J. et al. Dlhub: Simplifying publication, discovery, and use of machine learning models in science. **Journal of Parallel and Distributed Computing**, Elsevier, v. 147, p. 64–76, 2021.

MICROSOFT. **Home Page**. 2022. <https://azure.microsoft.com/>.

MISHRA, M. K.; PATEL, Y. S.; GHOSH, M.; MUND, G. A review and classification of grid computing systems. **International Journal of Computational Intelligence Research**, v. 13, n. 3, p. 369–402, 2017.

NJENGA, K.; GARG, L.; BHARDWAJ, A. K.; PRAKASH, V.; BAWA, S. The cloud computing adoption in higher learning institutions in kenya: Hindering factors and recommendations for the way forward. **Telematics and Informatics**, v. 38, p. 225–246, 2019. ISSN 0736-5853. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0736585318306397>.

PACHECO, P.; MALENSEK, M. **An Introduction to Parallel Programming**. [S.l.]: Morgan Kaufmann, 2011.

PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V. et al. Scikit-learn: Machine learning in python. **the Journal of machine Learning research**, JMLR. org, v. 12, p. 2825–2830, 2011.

PÉREZ, A.; RISCO, S.; NARANJO, D. M.; CABALLER, M.; MOLTÓ, G. On-premises serverless computing for event-driven data processing applications. In: IEEE. **2019 IEEE 12th International Conference on Cloud Computing (CLOUD)**. [S.l.], 2019. p. 414–421.

PORTNOY, M. **Virtualization Essentials**. [S.l.]: John Wiley & Sons, 2016.

PYTORCH. **Documentation**. 2022. <https://pytorch.org/docs/stable/rpc.html>.

ROSER, M.; RITCHIE, H. **Technological Change**. 2022. Disponível em: <https://ourworldindata.org/technological-change>.

SANTOS, H. C. T.; SOUZA, L. S. de; CARVALHO, J. H. A. de; FERREIRA, T. A. E. **PESC – Parallel Experiment for Sequential Code**. arXiv, 2023. Disponível em: <https://arxiv.org/abs/2301.05770>.

SINGH, M. An overview of grid computing. In: **2019 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)**. [S.l.: s.n.], 2019. p. 194–198.

SUKHOROSLOV, O. V.; AFANASIEV, A. Everest: A cloud platform for computational web services. In: **CLOSER**. [S.l.: s.n.], 2014. p. 411–416.

TAYLOR, S. J. E.; ANAGNOSTOU, A.; ABUBAKAR, N. T.; KISS, T.; DESLAURIERS, J.; TERSTYANSZKY, G.; KACSUK, P.; KOVACS, J.; KITE, S.; PATTISON, G.; PETRY, J. Innovations in simulation: Experiences with cloud-based simulation experimentation. In: **2020 Winter Simulation Conference (WSC)**. [S.l.: s.n.], 2020. p. 3164–3175.

TROBEC, R.; SLIVNIK, B.; BULIC, P.; ROBIC, B. **Introduction to parallel computing: from algorithms to programming on state-of-the-art platforms**. [S.l.]: Springer, 2020.

WATANABE, Y.; FUKUSHI, M. A parallel volunteer computing based on server assisted communications. In: IEEE. **2020 Eighth International Symposium on Computing and Networking Workshops (CANDARW)**. [S.l.], 2020. p. 242–247.

WISCONSIN–MADISON, U. of. **Documentation**. 2022. <https://research.cs.wisc.edu/htcondor/>.

APÊNDICES

APÊNDICE A – Instalação da Plataforma PESC

A.1 Instalação do Módulo Gerente

O gerente é o módulo da plataforma PESC que deve ser instalado como um servidor, nesse processo de instalação estará sendo utilizado o banco de dados SQLITE3. A implementação atual está trabalhando com as seguintes limitações:

- Sistema operacional Linux baseado no Debian
- Python 3.9 ou maior

Pré -requisitos

- hadolint: Para validar o Dockerfile (https://github.com/hadolint/hadolint)
- hadolint-wraper: Para formatar a saída da holdolint

Processo de instalação

- Configure o Hadolint
  - Download do arquivo binário do Hadolint em https://github.com/hadolint/hadolint/releases
  - $ Copiar o arquivo binário para /usr/local/bin com o nome hadolint e permissão de execução (+x)
- Clone o projeto do repositório no github

  $ git clone https://github.com/hctsantos/pesc_backend.git

  $ cd pesc_backend
- Copie o arquivo .env.example para .env e altere os valores da variável

  $ cp .env.example .env
- Importar o banco de dados

  $ make migrate
- Crie o usuário administrador da aplicação

  $ python manager.py createsuperuser
- Inicie o sistema

  $ make run
- Na interface do administrador, crie o usuário utilizado para a comunicação dos clientes com o gerente
  - http://localhost:8000/admin

A.2 Instalação do Módulo Cliente

O cliente é o modulo que deve estar instalado nos computadores que participarão da plataforma PESC. A implementação atual está trabalhando com as seguintes limitações:

- Sistema operacional Linux baseado no Debian
- Python 3.9 ou maior

Pré-requisitos

- Para computadores com GPU NVIDIA
  - A instalação do driver da GPU NVIDIA pode variar com o modelo da placa de vídeo disponível em cada computador e por isso deve ser verificado os procedimentos de instalaçao na página oficial da fabricante[1]
  - Instalação do NVIDIA Container Toolkit seguindo as orientações na página oficial do produto[2]
- Instalação do Docker

  $ sudo apt-get install apt-transport-https ca-certificates curl gnupg lsb-release

  $ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg –dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

  $ echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

  $ sudo apt-get update

  $ sudo apt-get install docker-ce docker-ce-cli containerd.io

  $ sudo groupadd docker
- Instalação do Make

  $ sudo apt install -y make

Processo de instalação

- Criar um nono usuário chamado pesc no sistema operacional

  $ sudo adduser pesc
- Adicionar o usuário pesc ao grupo docker

  $ sudo usermod -aG docker pesc
- Alterne para o usuário pesc

[1] https://docs.nvidia.com/datacenter/tesla/tesla-installation-notes/index.html
[2] https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html

$ su - pesc

- Clone o repositório do projeto

  $ git clone https://github.com/hctsantos/pesc_client.git

  $ cd pesc_client

- Criar e ativar o ambiente virtual no diretório do projeto, já está adicionado ao arquivo .gitignore

  $ python3 -m venv venv

  $ source ./venv/bin/activate

  $ pip install –upgrade pip

- Instalar os pacotes necessários

  $ pip install -r requirements.txt

- Criar um arquivo .env com as variáveis

  APPSEED_CONFIG_MODE=Debug

- Criar o arquivo de configuração config_client.py no diretório do projeto com os seguinte conteúdo:

```
1
2  #Nome do cliente como será exibido na interface web
3  CLIENT_NAME = 'Client 1'
4
5  #Endereço IP do cliente;
6  IP = '192.168.0.2'
7
8  #Porta usada para comunicação via API com o cliente
9  PORT = '9876'
10
11 #O gerente pode tentar ativar esse cliente, caso esteja offline
12 CAN_BE_ACTIVATED = False
13
14 #Nome do usuário do cliente definido no gerente
15 WEB_USER = 'client'
16
17 #Senha do usuário do cliente definido no gerente
18 WEB_PASSWORD = 'PescClientUser'
19
20 #Nome do usuário do sistema operacional
21 LINUX_USER = 'pesc'
22
23 #Endereço IP do gerente
24 IP_WEB_SERVER = '192.168.0.1'
25
26 #Porta de comunicação com o gerente
27 PORT_WEB_SERVER = '8000'
28
29 #Diretório onde os arquivos de imagens serão armazenados
```

```
30 BUILD_DIR = f'/home/pesc/.pesc/image_builds/'
31
32 #Diretório onde os arquivos de processos serão armazenados
33 RUN_DIR = '/home/pesc/.pesc/process_runs/'
34
35 #Diretório onde os arquivos compartilhados serão armazenados
36 SHARED_FILES_DIR = '/home/pesc/.pesc/shared_files/'
37
38 #Limite de uso do processador por esse serviço. Após esse valor ser alcançado o
       cliente informa ao gerente para não enviar mais processos para serem executados
39 CPU_PERCENT_LIMIT = 70
40
41 #Limite de uso de GPU por esse serviço. Após esse valor ser alcançado o cliente
       informa ao gerente para não enviar mais processos que dependam desse tipo de
       recurso para serem executados
42 GPU_PERCENT_MEMORY_LIMIT = 70
43
44 #Percentual de uso dos recursos alocados para os contêineres atualmente em execuçã
       o
45 CONCURRENT_RESOURCES_PERCENT = 10
46
47 #Número de GPUS disponíveis
48 NUM_GPU = 1
49
50 #Lista dos modelos das GPUs (Valores separados por vírgula)
51 GPUS = ['TITAN V']
52
53 #Capability de cada modelo informado (Valores separados por vírgula);
54 CAPABILITIES = [7]
55
56 #Intervalo de endereços IPs usados na criação dos contêineres;
57 CONTAINER_NETWORK = '172.17.0.0/16'
58
59 #Verificar se existe uso concorrente da máquina com outros usuários do sistema
       operacional
60 CHECK_CONCURRENT_USER = True
```

Algoritmo A.1 – Arquivo de configuração do cliente

- Iniciar o sistema

  $ make run

APÊNDICE B – Transferência de Tecnologia

A plataforma PESC deve ser implantanda e utilizada em infraestruturas diversas e ser uma ferramenta que garanta autonomia para os seus usuários. Uma etapa fundamental nesse processo é a transferência de tecnologia para que o implantandor da plataforma, e seus usuários, se apropriem do conhecimento necessário para sua utilização. Nesse Apêndice serão apresentados as ferramentas de transferência de tecnologia que forma desenvolvidos junto com a plataforma PESC.

B.1 Documentação

A documentação da plataforma PESC está disponível como um portal web que pode ser acessado por qualquer usuário da insituição onde ela foi implantada. A documentação é dividida entre os diferentes públicos alvos da plataforma listados abaixo e está apresentada na Figura 38:

- Administradores: Equipe de TI responsável por instalar e manter a plataforma em funcionamento;
- Operadores: Usuários com acesso avançado que podem criar e modificar elementos que afetam a execução da plataforma, como ambientes de execução, clientes, linguagens, etc;
- Usuários: Usuário que fazem uso das funcionalidades da plataforma para executar as suas simulações computacionais.

B.2 Tutoriais

Para garantir uma autonomia dos usários os tutorias foram divididos em:

- Como fazer?: Textos apresentados em um formato de perguntas e respostas que simplifcam o entendimento e direcionam o usuário para a ação que eles realmente querem executar;
- Vídeos: Vídeo aulas gravadas em um formato mais tradicional onde uma visão geral da plataforma é apresentada e algumas dinâmicas são apresentadas para o usuário entender como usar a plataforma;

Figura 38 – Portal da documentação da plataforma (o autor)

B.3 Documentação do Administrador

Além da documentação apresentada na Seção B.1 o administrador do sistema também tem acesso a uma documentação interna do sistema que apresenta os *endpoinst* RESTAPI que são chamados pelos módulos Cliente e Gerente. Essa documentação é acessada através do menu lateral quando o usuário administrador está acessando o sistema e pode ser visualizado na Figura 39.

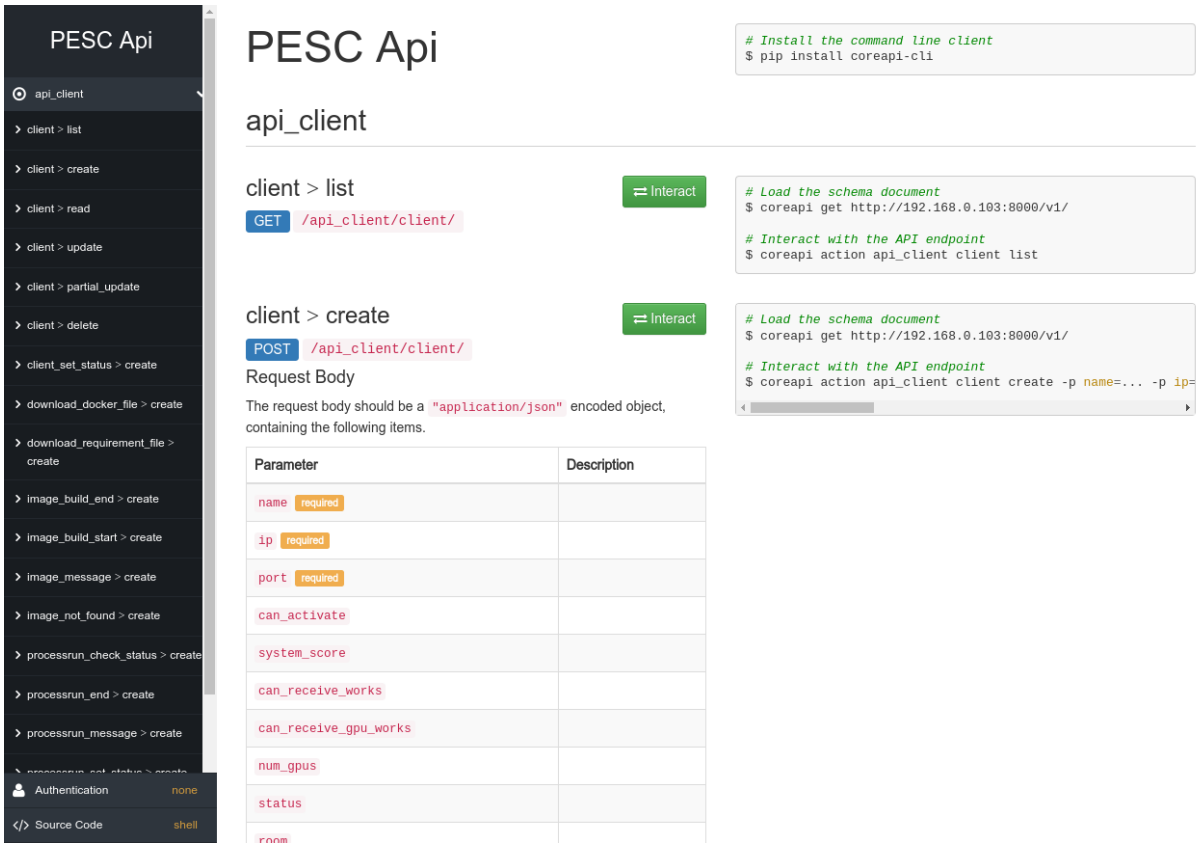


Figura 39 – Documentação das chamadas RESTAPI da plataforma (o autor)