**UniCEUB - CENTRO UNIVERSITÁRIO DE BRASÍLIA**

**Faculdade de Tecnologia e Ciências Sociais Aplicadas**

**CURSO: Ciência da Computação**

**DISCIPLINA: Arquitetura de Sistemas Distribuídos**

**PROFESSOR: Msc. Fabiano Mariath D´Oliveira**

**Sistema Distribuído para Processamento por meio de Computação Distribuída Voluntária**

**ALUNOS**

**CECÍLIA NEVES VIEIRA**

**GUILHERME LOPES COSTA PEREIRA**

**IVAN DE MEDEIROS YARED**

**LEONARDO CANTIERI TAUBE FAGUNDES**

**VITOR CASTRO DE OLIVEIRA**

Brasília, 20 de Junho de 2024

# 1 - Introdução

## Problema de Pesquisa

Causada por vírus e transmitida pelo mosquito *Aedes aegypti*, a dengue é uma doença há muito tempo presente no Brasil. Segundo o Ministério da Saúde, “a primeira epidemia documentada clínica e laboratorialmente ocorreu em 1981-1982, em Boa Vista (RR)” [1]. Desta data em diante, a dengue ocorre de forma endêmica, com ocorrências de epidemias no país; devido à diversos fatores, como expansão do habitat do mosquito vetor, introdução ou alteração de sorotipos novos ou predominantes, crescimento desordenado da população, precariedade do sistema de saneamento básico e fatores climáticos.

São conhecidos quatro sorotipos da dengue, que podem gerar formas assintomáticas da doença, sintomas leves ou graves e até mesmo ser fatal. A frequência de casos graves é elevada após a segunda ou terceira infecção, independente do sorotipo da doença anterior. Como sintomas, tem-se febre, dor de cabeça, dores pelo corpo, náuseas, manchas vermelhas na pele; e em casos mais graves, sangramentos, dor abdominal intensa e contínua e vômitos [2].

Esta doença encontra-se, atualmente, disseminada por praticamente todo o território nacional, segundo matéria do Uol, em abril de 2024, já está em 95% das cidades brasileiras, incluindo cidades que até 2023 não haviam registrado nenhum caso de dengue [3]. De acordo com a reportagem, em 2004, apenas 36% das cidades do país haviam apontado algum caso de dengue.

Já de acordo com a matéria de André Biernath da BBC [4], de janeiro até final de março de 2024 o Brasil já havia registrado mais de 2,3 milhões de casos prováveis de dengue, superando em mais de 500 mil casos todos os diagnósticos do ano de 2023, e sendo a pior crise sanitária da doença desde o início da série histórica do Ministério da Saúde, em 2000. Esse aumento explosivo de casos está relacionado aos fatores citados anteriormente, como mudança climática, grandes parcelas da população vivendo em áreas urbanas e falhas de políticas públicas adequadas ao combate da dengue.

Em relação ao Distrito Federal, segundo o Ministério da Saúde [5], em fevereiro de 2024, apresentava o maior número de casos por 100 mil habitantes entre todas as unidades da federação. No primeiro trimestre de 2024, 205 mortes foram causadas pela dengue no DF, possuindo uma quantidade de óbitos 10 vezes maior que o número total do ano de 2023; apresentando também um índice de letalidade em casos prováveis quase três vezes maior que a média nacional (0,10 no DF e 0,04 no Brasil) [6].

Retomando a reportagem da BBC [4], há uma explicação sobre o padrão de infecção da doença. Caso uma pessoa seja contaminada por um subtipo da dengue, fica imune a este subtipo mas pode desenvolver a doença com os outros sorotipos. “Essa característica da dengue cria uma dinâmica específica de transmissão, cujos padrões se repetem mais ou menos a cada cinco anos” [4], ou seja, uma população que foi acometida por uma variante, em algum tempo, pode ocorrer a disseminação de outro subtipo e os casos terão um aumento.

Essa alternância do vírus circulante está demonstrada em matéria da Folha de São Paulo [7], que relata sobre casos de dengue tipo 3 no Rio de Janeiro, variante que não circulava no estado desde 2007.

Dado esse cenário e a perspectiva de epidemias piores no futuro, devido ao aumento dos problemas climáticos e urbanos, nota-se a importância de criação de modelos que simulem a transmissão e efeitos da dengue no país, para auxiliar em estratégias de combate e distribuição de recursos financeiros e vacinas.

Porém, o volume de dados utilizados para esses estudos requer grande capacidade computacional de processamento, o que pode ser solucionado utilizando processamento ocioso de diversas máquinas por meio de sistemas distribuídos.

## Alternativas de Solução

Diversos projetos de computação distribuída voluntária já foram desenvolvidos para as mais diversas problemáticas. Durante a recente pandemia de Covid-19, o projeto Folding@home do departamento de Química da Universidade de Stanford se tornou o maior projeto do gênero, atingindo em abril de 2020 um processamento de 2,4 exaflops [8], com mais de 700 mil usuários inscritos na proposta e em média 30 mil pessoas rodando o software por dia [9].

Esse projeto consiste em utilizar o processamento ocioso de computadores ao redor do mundo, incluindo computadores domésticos, para realizar simulações de enrolamento de proteínas, para estudar a estrutura molecular de doenças como câncer, ELA, Influenza e Covid-19.

Outro programa de computação distribuída voluntária é o BOINC (*Berkeley Open Infrastructure for Network Computing*), desenvolvido na Universidade da Califórnia, com objetivo de facilitar que pesquisadores acessem recursos computacionais, por meio do processamento em computadores e dispositivos em todo o mundo. Essa plataforma possui aplicações em diversas áreas, como medicina, matemática, clima, astrofísica, dentre outros; destacam-se com mais de um milhão de voluntários, o SETI@home, projeto lançado em 1999 e já finalizado em que analisava frequências de rádio para pesquisa de sinais de vida extraterrestre , atingiu 1.808.938 usuários [10]; o projeto Rosetta@home, lançado em 2005 e ainda em operação, similar ao Folding@home com pesquisas de estruturas proteicas de doenças, atingiu 1.379.820 usuários [11]; e o Einstein@home, também lançado em 2005 e que continua ativo, utiliza sinais de rádio e dados de ondas gravitacionais para procurar por sinais de estrelas de nêutron em rotação que apresentam pulsos de radiação (pulsares), tendo alcançado 1.049.998 participantes [12].

Relacionado ao tema particular da dengue, o Africa@home, projeto público em parceria com o *Swiss Tropical Institute*, a Universidade de Genebra, a Organização Europeia para a Investigação Nuclear (CERN) e outras organizações, teve por objetivo utilizar a programação distribuída para contribuir para causa humanitárias na África.

Um de seus projetos foi o Malaria Control Project (MCP, Projeto de Controle de Malária, em tradução livre), criado para simular dinâmicas de transmissão e efeitos da malária na saúde. O programa atuou de dezembro de 2006 a junho de 2016, tendo que ser encerrado por restrições financeiras, sendo o primeiro projeto a utilizar computação voluntária para modelar doenças. O modelo buscava simular individualmente a infecção por malária em 50.000 a 100.000 pessoas ao longo da vida, simulando a frequência com que cada indivíduo é picado, infectado e consegue combater a infecção, além de dados como idade da pessoa, saúde, variação do número de parasitas no sangue e nível imunológico. Essas informações eram atualizadas a cada 5 dias ao longo da vida de uma população e o refinamento delas necessitou de diversos testes, com isso foram executadas simulações milhares de vezes, o que exigiu o uso de diversos computadores [13].

No estudo de Gagliardi [14], uma solução de grades computacionais para a simulação da dinâmica do espalhamento da dengue foi empregada. Essa opção utiliza sistemas distribuídos de alto desempenho ou recursos ociosos de computadores para gerar poder computacional, podendo também gerenciar e prover acesso a dados. São redes heterogêneas, escaláveis, com controle distribuído e podem conectar múltiplos domínios administrativos.

## Método de Pesquisa

Inicialmente buscou-se entender os métodos utilizados para a computação distribuída voluntária. Nosso trabalho envolve o desenvolvimento de um sistema baseado em computação voluntária, em que computadores disponibilizados por usuários utilizam sua capacidade ociosa para auxiliar um servidor central no processamento de dados.

Para a construção do referencial teórico e elaboração do sistema foram utilizados livros que introduzem os conhecimentos de computação em grade, como o *Fundamentals of Grid Computing* [15] e o *Grid Computing* [16]. O estudo realizado por Sarmenta em seu doutorado no Massachusetts Institute of Technology [17] possui informações sobre computação voluntária que também serão aproveitadas para este trabalho.

Após esta pesquisa bibliográfica inicial, será realizado o desenvolvimento do código utilizado nas aplicações que possibilitará a comunicação entre as máquinas do sistema e o processamento nos computadores voluntários.

Ao final será realizado um protótipo em pequena escala, simulando o processamento de uma parcela de dados e observando-se o tempo utilizado, desempenho e opções de escalabilidade do sistema para uma maior eficiência. Para isso, utilizaremos computadores conectados à rede local do próprio Ceub, em que uma máquina será o servidor central e as demais serão os computadores que fornecerão os recursos para este servidor.

## Objetivo do Trabalho

O objetivo geral do presente trabalho é a criação de um sistema distribuído de computação voluntária que possibilite o processamento de grandes volumes de dados para uma possível modelagem de transmissão e efeitos da dengue no Brasil, que poderá ser empregado inclusive para outras doenças.

Buscando garantir a eficácia e a clareza dos objetivos deste trabalho, utilizaremos a metodologia SMART, que permite definir metas que são Específicas, Mensuráveis, Atingíveis, Relevantes e Temporais. Essa abordagem auxilia que o desenvolvimento seja bem definido, focando em metas concretas e realistas.

* Específico:

Nosso objetivo é desenvolver um sistema distribuído voluntário que seja capaz de utilizar a capacidade ociosa de computadores para processamento de grandes volumes de dados, observando capacidade de escalabilidade e de melhora no tempo de processamento; auxiliando no desenvolvimento de uma infraestrutura robusta capaz de processar dados epidemiológicos para contribuir em pesquisas futuras e desenvolvimento de estratégias de controle e prevenção de doenças.

* Mensurável:

O objetivo será mensurado pela capacidade de gerar métricas de tempo de processamento e eficiência em relação ao número de máquinas voluntárias envolvidas, dado uma mesma quantidade de dados em cada teste. Por exemplo, se utilizarmos 3 máquinas voluntárias, resultando em um tempo de processamento x; ao utilizarmos 6 máquinas, espera-se um tempo de processamento próximo de x/2; indicando uma relação inversamente proporcional entre a quantidade de máquinas envolvidas e o tempo de processamento.

* Atingível:

A implementação desse sistema distribuído é viável com base na disponibilidade de tecnologias e plataformas existentes para computação distribuída voluntária.

* Relevante:

O sistema é relevante pois a criação de uma base para o processamento de dados em grande escala é de extrema importância para modelagem epidemiológica, auxiliando em pesquisas e desenvolvimento de estratégias, podendo impactar na gestão da saúde pública do nosso país.

* Temporal:

Esse sistema será desenvolvido, implementado e testado neste semestre, a fim de cumprir com o prazo final para o presente trabalho.

# 2 - Referencial Teórico

A computação em grelha refere-se a um modelo de computação distribuída em que diversas máquinas operam em conjunto para realizar tarefas que exigem alto desempenho ou alta capacidade de memória. Para o sistema distribuído ser considerado uma grelha, “o compartilhamento não é primariamente uma troca de arquivos, mas o acesso direto aos recursos computacionais, software, dispositivos de armazenamento e outros recursos [da máquina], com uma regra de compartilhamento altamente controlada que define claramente o que deve ser compartilhado, quem tem permissão de compartilhar e quais são as condições nas quais o compartilhamento deve ocorrer” (MAGOULÈS, 2009, p. 1. Tradução livre) [15].

O conceito de grelha computacional surgiu em meados da década de 1990 com o crescimento e desenvolvimento de redes de computadores de alta velocidade e a Internet. Com estes avanços, computadores distribuídos em diversas áreas geográficas puderam ser utilizados em conjunto para fornecer recursos nas áreas de pesquisa científica, engenharia e finanças. A característica principal da grelha computacional é a capacidade de permitir a colaboração e compartilhamento de recursos de diversas máquinas para resolver problemas que antes não eram possíveis por falta de capacidade computacional. A grelha computacional também permite o uso mais eficiente de computadores, utilizando núcleos do processador e memória normalmente ociosos [16].

Os sistemas de computação em grelha são divididos em dois tipos: grelha computacional e grelha de dados. A grelha computacional se refere a sistemas que se aproveitam de máquinas em um domínio para possuir maior capacidade computacional do que qualquer máquina individual no sistema. A grelha de dados descreve sistemas que provêem capacidade de armazenamento para gerenciamento de dados em larga escala distribuídos em uma WAN (Wide Area Network ou rede de grande extensão). Uma terceira categorização, proposta por Krauter, define a grelha de serviços, que se refere a sistemas que fornecem serviços que não são providos por nenhuma máquina individual [15].

A grelha computacional possui uma arquitetura aberta, extensível e dividida em camadas, que facilitam a identificação e classificação de componentes em relação às suas funções, ainda que estes possam depender de outros componentes em camadas mais baixas. De acordo com Magoulès [15], existem os seguintes tipos de camadas:

* **Camada trama** **(tradução livre, do inglês *fabric layer*):** a camada trama é responsável por fornecer controle local de recursos computacionais, de armazenamento, de rede, repositórios de código e catálogos. Os componentes desta camada têm como função monitorar e gerenciar o acesso a estes recursos, que incluem físicos e virtuais.
* **Camada de conectividade (*connectivity layer*):** a camada responsável pelos protocolos de comunicação e autenticação necessários para a transmissão de dados entre recursos da camada trama.
* **Camada de recursos (*resource layer*):** a camada de recursos define protocolos para o uso seguro, monitoramento e controle de recursos individuais.
* **Camada de coletividade (*collective layer*):** camada que define os protocolos e serviços que não estão associados a nenhum recurso específico. Estes protocolos controlam interações entre coleções de recursos e componentes globalmente.
* **Camada de aplicação (*application layer*):** a camada que contém as aplicações utilizadas pelo usuário.

|  |
| --- |
| **Figura 1 - Magoulès, 2009** [15] |

Uma das aplicações da computação em grelha é denominada computação voluntária. A computação voluntária é uma forma de computação em grelha que torna mais fácil e intuitiva a instalação e configuração de máquinas para colaborarem em um projeto, permitindo que usuários comuns, em diversas regiões do globo, sem qualquer conhecimento técnico, juntem-se à rede de forma rápida e consigam utilizar os recursos ociosos de sua máquina para resolver problemas computacionais. Essa possibilidade expande o leque de máquinas disponíveis substancialmente e em um curto espaço de tempo [17].

Exemplos de projetos de computação voluntária, onde qualquer usuário pode participar a qualquer momento, incluem:

* **Folding@home (medicina e biologia) [18]:** simulação de enovelamento de proteínas para compreender doenças relacionadas à falha deste processo, como doença de Alzheimer e fibrose cística.
* **PrimeGrid (matemática) [19]:** descoberta de novos números primos e resolução de problemas matemáticos
* **Asteroids@home (astrofísica) [20]:** pesquisa e descoberta de propriedades físicas de asteroides.

# 3 - Desenvolvimento

## 3.1. Escolha da Linguagem de Programação e Protocolo

Para criar um sistema distribuído que aproveita a capacidade ociosa de computadores voluntários para o processamento de dados, optou-se pela comunicação lógica via *sockets* UDP (User Datagram Protocol). A escolha do UDP foi motivada por sua simplicidade e eficiência em termos de desempenho. Diferentemente do TCP (*Transmission Control Protocol*), o UDP é um protocolo não orientado à conexão, o que resulta em menor *overhead* e latência, tornando-o adequado para a aplicação onde a velocidade de transmissão é importante e a perda ocasional de pacotes pode ser tolerada.

Além disso, foi utilizada a linguagem de programação *Python* devido à sua simplicidade, legibilidade e a ampla gama de bibliotecas disponíveis, o que facilita o desenvolvimento de aplicações distribuídas. A linguagem *Python*, além de ser amplamente utilizada para aplicações em ciência de dados e *machine learning*, possui bibliotecas que suportam o desenvolvimento de sistemas distribuídos, como *socket* para comunicação em rede e *threading* para gerenciamento de *threads*.

Por questões de segurança e simplicidade, o servidor estará hospedado e conectado à rede interna do CEUB, exigindo que qualquer cliente que queira conectar-se esteja também conectado a esta rede. A rede interna do CEUB possui arquitetura voltada para o uso da rede por desconhecidos e fornece serviços robustos de segurança que mitigam invasões, mantendo a integridade dos computadores utilizados.

## 3.2. Estruturação do Código

O desenvolvimento do sistema foi estruturado em duas partes principais: o servidor e o cliente.

O servidor UDP é responsável por gerenciar conexões de múltiplos clientes utilizando *threads*. Para cada novo cliente que se conecta, o servidor realiza um processo de "*handshake*" para validar a conexão. Este processo assegura que o cliente está autorizado a se comunicar com o servidor. Após a validação, o servidor mantém um registro dos clientes conectados e fica disponível para receber e processar as mensagens enviadas por eles. O uso de *threads* permite que o servidor gerencie várias conexões simultaneamente de maneira eficiente, garantindo que cada cliente receba a devida atenção sem que o desempenho geral do sistema seja comprometido.

O cliente, por sua vez, é responsável por estabelecer a conexão inicial com o servidor e processar as tarefas atribuídas após a confirmação do "*handshake*". A partir do momento em que a conexão é estabelecida, o cliente pode enviar mensagens ao servidor e aguardar as respostas correspondentes. Este processo de comunicação continua até que o cliente decida se desconectar, momento em que uma mensagem de "*disconnect*" é enviada ao servidor para encerrar a conexão de forma ordenada.

Foi desenvolvido um código inicial para estabelecer a comunicação entre as duas partes, implementando as funcionalidades básicas descritas acima. Este código inclui a lógica para o processo de "*handshake*", o gerenciamento de conexões usando threads no servidor, e a capacidade do cliente de enviar e receber mensagens de maneira eficiente.

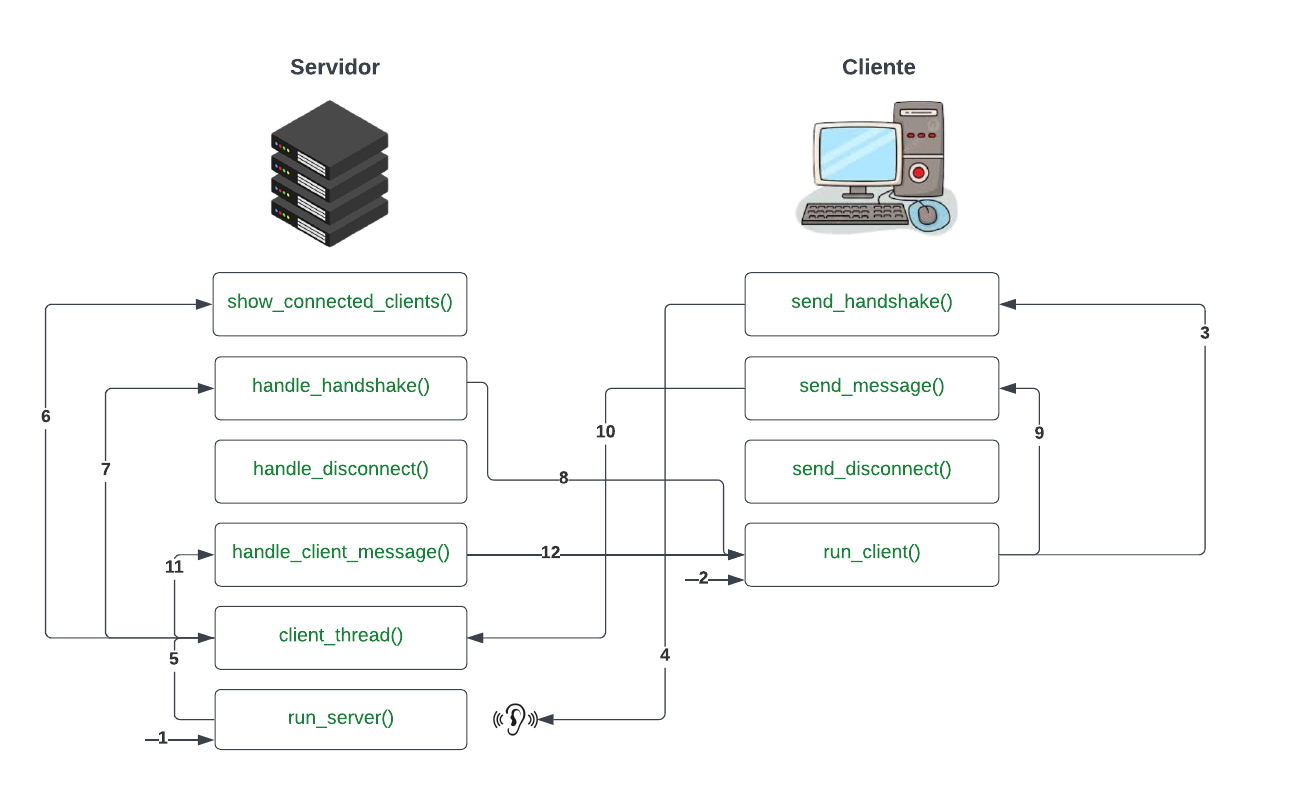
|  |  |
| --- | --- |
| **Funções do servidor** | **Descrição da função** |
| log() | Exibe mensagens de log com timestamp. |
| show\_connected\_clients() | Exibe a lista de clientes conectados. |
| handle\_handshake() | Realiza o handshake com novos clientes. |
| handle\_disconnect() | Desconecta os clientes e atualiza a lista. |
| handle\_client\_message() | Processa mensagens de clientes e envia respostas. |
| client\_thread() | Gerencia a comunicação com cada cliente em uma thread separada. |
| run\_server() | Inicia o servidor UDP, espera conexões e cria threads para clientes. |

**Tabela 1 - Funções presentes no código do servidor.** Autoria própria.

|  |  |
| --- | --- |
| **Funções do cliente** | **Descrição da função** |
| send\_handshake() | Estabelece conexão com o servidor. |
| send\_message() | Envia mensagens ao servidor e recebe respostas. |
| send\_disconnect() | Envia mensagem de desconexão ao servidor. |
| run\_client() | Conecta ao servidor, envia mensagens e gerencia desconexão. |

**Tabela 2 - Funções presentes no código do cliente.** Autoria própria.

A imagem abaixo representa o fluxo das ações de interação entre as funções do cliente e as funções do servidor de forma enumerada.



**Figura 2 - Representação da interação entre as funções do cliente com as do servidor.** Autoria própria.

### 3.2.1. Código do Servidor

def show\_connected\_clients(*clients*: set) -> None:

"""

Exibe na tela os clientes conectados.

Args:

clients (set): Conjunto de tuplas contendo endereço IP e porta dos clientes conectados.

"""

if *clients*:

log(f"| {'Clientes conectados':^30} |", *showTime*=False)

log(f"| {'Client\_IP':^15} | {'Client\_PORT':^15} |", *showTime*=False)

for client in *clients*:

ip, port = client

log(f"| {ip:^15} | {port:^15} |", *showTime*=False)

log()

else:

log("Nenhum cliente conectado no momento.")

def handle\_handshake(*UDPServerSocket*: socket.socket, *address*: tuple, *connected\_clients*: set) -> None:

"""

Realiza o handshake com um novo cliente e o adiciona à lista de clientes conectados.

Args:

UDPServerSocket (socket.socket): Socket do servidor UDP.

address (tuple): Tupla contendo endereço IP e porta do cliente.

connected\_clients (set): Conjunto de tuplas contendo endereço IP e porta dos clientes conectados.

"""

handshakeAck = "Handshake ACK"

*UDPServerSocket*.sendto(str.encode(handshakeAck), *address*)

*connected\_clients*.add(*address*)

log(f"[handshake] Cliente {*address*} conectado.\n")

def handle\_disconnect(*address*: tuple, *connected\_clients*: set) -> None:

"""

Desconecta um cliente e o remove da lista de clientes conectados.

Args:

address (tuple): Tupla contendo endereço IP e porta do cliente.

connected\_clients (set): Conjunto de tuplas contendo endereço IP e porta dos clientes conectados.

"""

if *address* in *connected\_clients*:

*connected\_clients*.remove(*address*)

log(f"[disconnect] Cliente {*address*} desconectado.\n")

show\_connected\_clients(list(*connected\_clients*))

def handle\_client\_message(*UDPServerSocket*: socket.socket, *message*: bytes, *address*: tuple) -> None:

"""

Processa a mensagem recebida de um cliente e envia uma resposta. Foi feito um teste que ao receber uma mensagem “AA” do cliente por meio de input no terminal o servidor envia uma resposta de acerto, caso contrário o servidor envia uma resposta de mensagem incorreta.

Args:

UDPServerSocket (socket.socket): Socket do servidor UDP.

message (bytes): Mensagem recebida do cliente.

address (tuple): Tupla contendo endereço IP e porta do cliente.

"""

client\_ip, client\_port = *address*

clientIP = f"IP: {client\_ip} | Port: {client\_port}"

clientMsg = f"Mensagem do Cliente: {*message*.decode()}\n"

log(clientIP)

log(clientMsg)

if *message*.decode() == "AA":

response = "Resposta correta!"

else:

response = "Resposta incorreta!"

*UDPServerSocket*.sendto(str.encode(response), *address*)

def client\_thread(*UDPServerSocket*: socket.socket, *message*: bytes, *address*: tuple, *connected\_clients*: set) -> None:

"""

Thread para lidar com as mensagens de um cliente.

Args:

UDPServerSocket (socket.socket): Socket do servidor UDP.

message (bytes): Mensagem recebida do cliente.

address (tuple): Tupla contendo endereço IP e porta do cliente.

connected\_clients (set): Conjunto de tuplas contendo endereço IP e porta dos clientes conectados.

"""

if *address* not in *connected\_clients*:

if *message*.decode() == "handshake":

handle\_handshake(*UDPServerSocket*, *address*, *connected\_clients*)

show\_connected\_clients(list(*connected\_clients*))

return

if *message*.decode() == "disconnect":

handle\_disconnect(*UDPServerSocket*, *address*, *connected\_clients*)

return

handle\_client\_message(*UDPServerSocket*, *message*, *address*)

def run\_server(*localIP*: str, *localPort*: int, *bufferSize*: int) -> None:

"""

Inicia o servidor UDP e aguarda conexões de clientes.

Args:

localIP (str): Endereço IP local do servidor.

localPort (int): Porta local do servidor.

bufferSize (int): Tamanho do buffer para recebimento de mensagens.

"""

UDPServerSocket = socket.socket(*family*=socket.AF\_INET, *type*=socket.SOCK\_DGRAM) # AF\_INET é uma constante que indica que estamos criando um socket para comunicação na internet, utilizando endereços IPv4. | SOCK\_DGRAM é uma constante que indica que estamos criando um socket para o protocolo UDP

UDPServerSocket.bind((*localIP*, *localPort*)) # Associa um IP e uma Porta ao socket

log("Servidor UDP ativo e ouvindo...\n")

connected\_clients = set()

while True:

bytesAddressPair = UDPServerSocket.recvfrom(*bufferSize*) # Espera uma mensagem de um cliente

message = bytesAddressPair[0]

address = bytesAddressPair[1]

thread = threading.Thread(*target*=client\_thread, *args*=(UDPServerSocket, message, address, connected\_clients))

thread.start()

if \_\_name\_\_ == "\_\_main\_\_":

LOCAL\_IP = "0.0.0.0"

LOCAL\_PORT = 20001

BUFFER\_SIZE = 1024

run\_server(LOCAL\_IP, LOCAL\_PORT, BUFFER\_SIZE)

### 

### 3.2.2. Código do Cliente

def send\_handshake(*UDPClientSocket*: socket.socket, *serverAddressPort*: tuple, *bufferSize*: int) -> bool:

"""

Envia uma mensagem de handshake para o servidor e aguarda a confirmação.

Args:

UDPClientSocket (socket.socket): Socket do cliente UDP.

serverAddressPort (tuple): Tupla contendo endereço IP e porta do servidor.

bufferSize (int): Tamanho do buffer para recebimento de mensagens.

Returns:

bool: True se o handshake for bem-sucedido, False caso contrário.

"""

handshake\_message = "handshake"

*UDPClientSocket*.sendto(str.encode(handshake\_message), *serverAddressPort*)

msgFromServer, \_ = *UDPClientSocket*.recvfrom(*bufferSize*)

return msgFromServer.decode() == "Handshake ACK"

def send\_message(*UDPClientSocket*: socket.socket, *serverAddressPort*: tuple, *message*: str, *bufferSize*: int) -> str:

"""

Envia uma mensagem para o servidor e aguarda a resposta.

Args:

UDPClientSocket (socket.socket): Socket do cliente UDP.

serverAddressPort (tuple): Tupla contendo endereço IP e porta do servidor.

message (str): Mensagem a ser enviada para o servidor.

bufferSize (int): Tamanho do buffer para recebimento de mensagens.

Returns:

str: Resposta recebida do servidor.

"""

*UDPClientSocket*.sendto(str.encode(*message*), *serverAddressPort*)

msgFromServer, \_ = *UDPClientSocket*.recvfrom(*bufferSize*)

return msgFromServer.decode()

def send\_disconnect(*UDPClientSocket*: socket.socket, *serverAddressPort*: tuple, *bufferSize*: int) -> None:

"""

Envia uma mensagem de desconexão para o servidor.

Args:

UDPClientSocket (socket.socket): Socket do cliente UDP.

serverAddressPort (tuple): Tupla contendo endereço IP e porta do servidor.

bufferSize (int): Tamanho do buffer para recebimento de mensagens.

"""

disconnect\_message = "disconnect"

*UDPClientSocket*.sendto(str.encode(disconnect\_message), *serverAddressPort*)

def run\_client(*serverAddressPort*: tuple, *bufferSize*: int) -> None:

"""

Executa o cliente UDP, estabelece conexão com o servidor e permite o envio de mensagens.

Args:

serverAddressPort (tuple): Tupla contendo endereço IP e porta do servidor.

bufferSize (int): Tamanho do buffer para recebimento de mensagens.

"""

UDPClientSocket = socket.socket(*family*=socket.AF\_INET, *type*=socket.SOCK\_DGRAM) # AF\_INET é uma constante que indica que estamos criando um socket para comunicação na internet, utilizando endereços IPv4. | SOCK\_DGRAM é uma constante que indica que estamos criando um socket para o protocolo UDP

if send\_handshake(UDPClientSocket, *serverAddressPort*, *bufferSize*):

print("Conexão estabelecida com o servidor.\n")

try:

while True:

user\_input = input("Digite uma mensagem ('sair' para encerrar): ")

if user\_input.lower() == "sair":

break

response = send\_message(UDPClientSocket, *serverAddressPort*, user\_input, *bufferSize*)

print(f"Mensagem do Servidor: {response}\n")

finally:

send\_disconnect(UDPClientSocket, *serverAddressPort*, *bufferSize*)

print("Desconectado do servidor.")

else:

print("Falha ao estabelecer conexão com o servidor.")

UDPClientSocket.close()

if \_\_name\_\_ == "\_\_main\_\_":

SERVER\_ADDRESS\_PORT = ("192.168.15.11", 20001)

BUFFER\_SIZE = 1024

run\_client(SERVER\_ADDRESS\_PORT, BUFFER\_SIZE)

## 3.3. Testes e Validação

### 3.3.1. Teste em uma Máquina

Para validar a funcionalidade básica do sistema, foram realizados testes iniciais em uma única máquina. Tanto o servidor quanto o cliente foram executados localmente, garantindo que a comunicação UDP estava funcionando corretamente.

### 3.3.2. Teste em duas Máquinas

O passo seguinte envolveu testar o servidor e o cliente em duas máquinas diferentes dentro da rede do CEUB. Esse teste confirmou que a comunicação UDP entre máquinas distintas estava operando conforme esperado com as duas máquinas conectadas na rede cabeada. Durante os testes, observou-se a capacidade do sistema em gerenciar múltiplas conexões e processar mensagens corretamente.

## 3.4. Implementação do algoritmo de estimativa do valor de Pi

Para validar o funcionamento da troca de mensagens em uma situação de computação voluntária, foi implementada uma função que estima o valor de Pi por meio do método de Monte Carlo. A técnica de Monte Carlo consiste em executar simulações aleatórias a fim de estimar um resultado específico [21]. Apesar de sua invenção ser anterior ao desenvolvimento do computador moderno, o poder de processamento e capacidade de memória das máquinas atuais permitiram o uso da técnica em diversas aplicações nas áreas científica (especialmente física e matemática) e financeira [21]. Em problemas científicos que envolvem várias dimensões, “os cálculos realizados em métodos de Monte Carlo são computacionalmente mais eficientes se comparados com métodos determinísticos” (KALOS e WHITLOCK, 2008, p. 3. Tradução livre) [22].

Por necessitar de uma grande quantidade de simulações aleatórias para alcançar estimativas precisas, os métodos de Monte Carlo são beneficiados pelo alto poder de processamento e capacidade de memória de sistemas distribuídos. Modelos mais complexos, como os que envolvem pesquisas científicas na área das ciências exatas, necessitam de uma quantidade massiva de poder computacional para alcançar o resultado desejado, nem sempre disponível no local. Estas pesquisas beneficiam-se da computação voluntária. Um exemplo de simulação de Monte Carlo que utiliza este método de sistema distribuído é o ATLAS@Home, experimento de física de partículas da equipe responsável pelo LHC, acelerador de partículas da CERN (Organização Europeia para a Investigação Nuclear) [23].

A estimação do valor de Pi por meio de Monte Carlo é feita através da geração de diversas coordenadas aleatórias em um plano (x,y) que contém um círculo e um quadrado, ambos centrados em (0, 0) e com o diâmetro do círculo igual ao lado do quadrado [21].

Sabe-se que a área do círculo é expressa por , onde é o raio deste círculo. Já a área do quadrado circunscrito a este círculo é dada pela expressão , uma vez que cada lado do quadrado é igual a 2r, como é possível observar na figura a seguir:

|  |
| --- |
| **Figura 3 - Representação gráfica de um círculo e um quadrado concêntricos.** Autoria própria. |

Com isso, tem-se que a razão da área do círculo e da área do quadrado é dada pela expressão:

Dada a razão entre as áreas, pode-se também entender que, ao ser multiplicada por 4, obtém-se o valor de Pi:

; então 4 \*

Porém, áreas possuem uma grandeza infinita de pontos, o que torna impossível gerar coordenadas de pontos que representem a sua totalidade. Com isso, é necessário trabalhar com uma aproximação dada por coordenadas aleatórias, onde a maior quantidade de pontos possível geral será uma representação mais fiel da área das figuras.

Assim, após a distribuição das coordenadas dentro da área do quadrado, são obtidos o número de pontos que se encontram dentro da área do círculo e o número total de pontos distribuídos por toda a área do quadrado, gerando a equivalência:

Esta última expressão traz que a razão entre o número de coordenadas geradas dentro da área do círculo e o total de coordenadas geradas é uma aproximação da razão entre as duas áreas [21].

Ao substituir o valor dos pontos na expressão em que se obtém o valor de Pi, tem-se:

Logo, ao utilizar esse método e definir uma quantidade total de pontos, é possível gerar computacionalmente as diversas coordenadas aleatórias e estimar o valor de Pi. Uma vez que uma maior quantidade de pontos totais representa uma estimativa melhor deste valor, tem-se um grande custo computacional, o que torna interessante o uso de computação distribuída.

### 3.4.1. Funções do algoritmo

A implementação do algoritmo consistiu em desenvolver duas novas funções. A primeira, parte do cliente, é responsável por gerar as coordenadas aleatórias (sempre entre -1 e 1) e enviar para o servidor, onde a segunda função, responsável por realizar os cálculos necessários para a estimativa, é executada. Em ambos os casos, as funções de recebimento e envio de mensagens já presentes foram modificadas a fim de adequar-se à execução do método.

Optou-se por essa distribuição de tarefas entre as entidades do sistema, geração de coordenadas no cliente e cálculo se este ponto se encontra dentro da área do círculo no servidor, pois em casos reais de utilização da computação voluntária o servidor apresenta uma grande robustez de processamento e armazenamento, enquanto os clientes voluntários são computadores pessoais doando capacidade ociosa. Por esse perfil, foi definido que o processamento mais complexo estaria no código do servidor, mesmo em situações em que todas as máquinas envolvidas no sistema possuam capacidades similares.

import random

from numpy import random

def monte\_carlo():

'''

Cria uma coordenada de um ponto aleatório

'''

x = random.rand()

print(x)

y = random.rand()

print(y)

return (x, y)

def run\_client(serverAddressPort: tuple, bufferSize: int) -> None:

UDPClientSocket = socket.socket(family=socket.AF\_INET, type=socket.SOCK\_DGRAM)

if send\_handshake(UDPClientSocket, serverAddressPort, bufferSize):

print("Conexão estabelecida com o servidor.\n")

try:

while True:

tupla = monte\_carlo()

response = send\_message(UDPClientSocket, serverAddressPort, tupla, bufferSize)

print(f"Mensagem do Servidor: {response}\n")

if response == "Limite de pontos alcancado":

break

finally:

send\_disconnect(UDPClientSocket, serverAddressPort, bufferSize)

print("Desconectado do servidor.")

else:

print("Falha ao estabelecer conexão com o servidor.")

UDPClientSocket.close()

A função de geração de coordenadas consiste em dois geradores de números aleatórios. Estes números são inseridos em uma tupla. A função de comunicação com o servidor envia a tupla e recebe uma mensagem de resposta caso bem sucedida. Quando o limite de coordenadas previamente determinado pelo servidor é alcançado, o cliente é desconectado e a execução é encerrada. O funcionamento da comunicação descrito na seção 3.2 continua o mesmo.

def handle\_client\_monte\_carlo(UDPServerSocket: socket.socket, message: bytes, address: tuple):

'''

Lida com a mensagem de um cliente para a simulação de Monte Carlo para estimar o valor de π.

Esta função processa a mensagem recebida de um cliente, que contém um ponto (x, y) em formato de string.

O ponto é usado em uma simulação de Monte Carlo para estimar o valor de π. Se o número total de pontos

estiver abaixo de um limite predefinido, o ponto é avaliado para verificar se está dentro do círculo unitário.

As contagens globais de pontos totais e pontos dentro do círculo são atualizadas conforme necessário. Uma

resposta é enviada de volta ao cliente indicando se os dados foram adicionados ou se o limite de pontos foi

alcançado.

Parâmetros:

UDPServerSocket (socket.socket): O objeto socket para comunicação.

message (bytes): A mensagem recebida do cliente, que deve ser uma string codificada em bytes representando um ponto (x, y).

address (tuple): O endereço do cliente no formato (IP, porta).

'''

client\_ip, client\_port = address

clientIP = f"IP: {client\_ip} | Port: {client\_port}"

clientMsg = f"Tupla do Cliente: {message.decode()}\n"

log(clientIP)

log(clientMsg)

tupla = message.decode().split(",")

global total\_points

global points\_in\_circle

if total\_points < limit:

x = Decimal(tupla[0])

print(x)

y = Decimal(tupla[1])

print(y)

total\_points += 1

if ((x\*x) + (y\*y)) <= 1:

points\_in\_circle += 1

print("Pontos dentro do circulo: ", points\_in\_circle, " | ", "Total de pontos: ", total\_points, " | ", "Valor de pi: ", round(Decimal(4\*(points\_in\_circle / total\_points)), 20), "\n")

response = "Dados adicionados"

else:

response = "Limite de pontos alcancado"

UDPServerSocket.sendto(str.encode(response), address)

def client\_thread(UDPServerSocket: socket.socket, message: bytes, address: tuple, connected\_clients: set) -> None:

"""

Thread para lidar com as mensagens de um cliente.

Args:

UDPServerSocket (socket.socket): Socket do servidor UDP.

message (bytes): Mensagem recebida do cliente.

address (tuple): Tupla contendo endereço IP e porta do cliente.

connected\_clients (set): Conjunto de tuplas contendo endereço IP e porta dos clientes conectados.

"""

if address not in connected\_clients:

if message.decode() == "handshake":

handle\_handshake(UDPServerSocket, address, connected\_clients)

show\_connected\_clients(list(connected\_clients))

return

if message.decode() == "disconnect":

handle\_disconnect(UDPServerSocket, address, connected\_clients)

return

handle\_client\_monte\_carlo(UDPServerSocket, message, address)

A função do servidor é responsável por receber as coordenadas geradas pelos clientes e realizar o cálculo de estimativa do valor de Pi. O servidor mantém a execução até o limite de pontos pré-determinado ser alcançado. Neste momento, o servidor encerra as conexões e a execução é finalizada. A função “client\_thread” foi meramente modificada para adaptar-se à função de Monte Carlo, mantendo o funcionamento da comunicação desenvolvida anteriormente.

## 3.5. Testes utilizando o algoritmo de Monte Carlo

Foram realizados diversos testes utilizando o algoritmo de Monte Carlo em um laboratório do CEUB, em que foi possível utilizar a conexão em rede local das máquinas para observar desempenho de processamento e escalabilidade.

Utilizou-se diversos números de máquinas e de quantidades de pontos de simulação aleatória para estimativa do valor de Pi. É importante destacar as propriedades das máquinas clientes do sistema: elas possuem 16 GB de memória RAM e processador Intel Core i7-6700.

Além disso, o servidor precisou ser um computador pessoal de um membro da equipe. Os computadores não realizam o *handshake* quando o código servidor está operando em uma máquina do CEUB.

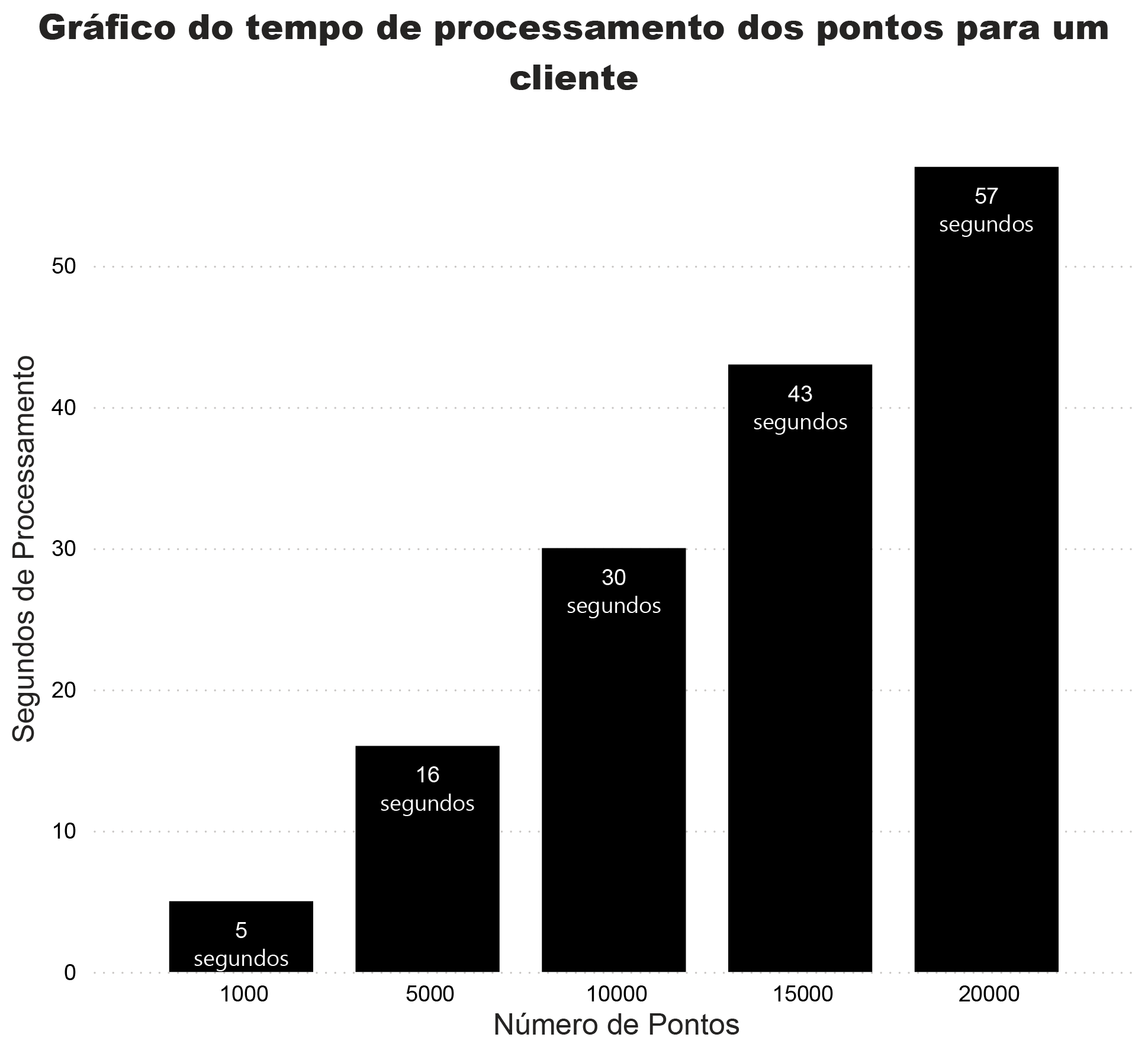
Abaixo estão descritos os cenários de testes que foram utilizados.

### 3.5.1. Cenário 01 - Um cliente

Inicialmente, foi produzido o cenário em que apenas um cliente auxiliou no processamento do algoritmo. Foram utilizadas diversas quantidades de pontos de simulação aleatória, sendo as seguintes com seus respectivos resultados:

* 1.000 pontos - tempo total de processamento: 5 segundos
* 5.000 pontos - tempo total de processamento: 16 segundos
* 10.000 pontos - tempo total de processamento: 30 segundos
* 15.000 pontos - tempo total de processamento: 43 segundos
* 20.000 pontos - tempo total de processamento: 57 segundos

Estes dados estão também disponíveis na visualização abaixo em formato de gráfico de barras, que demonstra com mais facilidade a proporção do crescimento do tempo de processamento ao aumentar o número de pontos:



**Gráfico 1 -** Autoria própria.

Nota-se que há um crescimento praticamente linear do tempo de processamento, com isso foi decidido utilizar ferramentas de previsão e de tendência do PowerBI e um último teste com um cliente para observar se o processamento para 50.000 pontos se manteria constante. Abaixo há o gráfico gerado no PowerBI, sendo a linha preta tracejada a linha de tendência, a linha vermelha a linha real da variação do tempo para os pontos e, por fim, a linha em laranja é a linha de previsão:

**Gráfico 2 -** Autoria própria.

Ao ser realizado o teste prático com 50.000 pontos, obteve-se o tempo de processamento de 139 segundos, valor muito próximo à previsão obtida; indicando um crescimento linear do tempo de processamento à medida que mais pontos são adicionados ao método de Monte Carlo.

### 3.5.2. Cenário 02 - Quatro clientes

Ao adicionar clientes para testes, a quantidade de pontos selecionados foi alterada, devido à escala de processamento. Para 4 clientes, obteve-se os seguintes resultados:

* 10.000 pontos - tempo total de processamento: 24 segundos
* 50.000 pontos - tempo total de processamento: 99 segundos
* 100.000 pontos - tempo total de processamento: 233 segundos

Pode-se observar que o tempo de processamento para 10.000 pontos em relação à utilização de um cliente caiu 6 segundos, o que parece um valor pequeno; porém para 50.000 pontos a diferença foi de 40 segundos, valor muito expressivo, principalmente em termos computacionais.

### 3.5.3. Cenário 03 - Oito clientes

Para esse cenário utilizou-se a mesma quantidade de pontos do Cenário 02, a fim de testar o aumento da capacidade de processamento, com oito computadores clientes. Abaixo os resultados:

* 10.000 pontos - tempo total de processamento: 22 segundos
* 50.000 pontos - tempo total de processamento: 109 segundos
* 100.000 pontos - tempo total de processamento: 242 segundos

Nota-se que os dados obtidos chamam atenção, pois o esperado é que ocorresse como no caso de 10.000 pontos, em que houve uma redução do tempo; porém nos demais cenários o tempo de processamento aumentou aproximadamente 10 segundos.

### 3.5.4. Cenário 04 - Doze clientes

Por fim, o teste do algoritmo de Monte Carlo foi realizado utilizando o processamento ocioso de doze clientes. Os resultados obtidos foram os seguintes:

* 10.000 pontos - tempo total de processamento: 22 segundos
* 50.000 pontos - tempo total de processamento: 109 segundos
* 100.000 pontos - tempo total de processamento: 236 segundos

Observa-se que o tempo de processamento para 10.000 e 50.000 pontos foi exatamente igual ao tempo de processamento do Cenário 03, mas o tempo para 100.000 pontos diminuiu novamente. Essas pequenas oscilações ou a manutenção do tempo de processamento, mesmo com aumento de clientes, levaram ao levantamento de algumas hipóteses para motivos de não ter ocorrido uma melhora no processamento como acontecido entre os Cenários 01 e 02.

Tem-se a possibilidade de gargalo no *switch* da rede local, uma vez que todos os computadores utilizam a capacidade de rede do laboratório do CEUB; ou limitações na interface de rede do computador servidor; ou até mesmo falha na implementação do código, como limitações na biblioteca utilizada ou tamanhos de *buffer*, que podem acarretar em problemas para receber tantas requisições simultâneas.

### 3.5.5. Cenário 05 - Teste de requisições simultâneas

A fim de testar o problema das requisições simultâneas que podem ocorrer devido à biblioteca ou tamanho de *buffer*, optou-se por realizar um Cenário de teste que trouxesse essa problemática. Como as bibliotecas empregadas no código são comumente utilizadas pela comunidade e em diversas aplicações, realizou-se um teste aumentando o *buffer size* do servidor. O *socket* UDP possui um tamanho de *buffer* pré definido no código de 1024 bytes. Cada cliente envia uma mensagem de 144 bytes para o servidor, o que pode ter causado o gargalo observado, causando a perda de eficiência de processamento e de mensagens.

Quando foi aumentado este tamanho, o servidor foi capaz de gravar mais coordenadas, diminuindo o gargalo de recebimento das mensagens dos clientes; sendo apto a armazenar e processar mais pontos gerados.

Inicialmente, o teste foi novamente executado em quatro clientes, aumentando o *buffer size* para 2048 bytes. Os resultados foram os seguintes:

* 10.000 pontos - tempo total de processamento: 21 segundos
* 50.000 pontos - tempo total de processamento: 92 segundos
* 100.000 pontos - tempo total de processamento: 184 segundos

Após este teste, realizou-se o experimento com 8 clientes e *buffer size* ainda de 2048 bytes, obtendo os seguintes tempos:

* 10.000 pontos - tempo total de processamento: 22 segundos
* 50.000 pontos - tempo total de processamento: 94 segundos
* 100.000 pontos - tempo total de processamento: 177 segundos

Nota-se que apenas na situação com 100.000 pontos houve uma mudança significativa no tempo de processamento, uma vez que o tamanho do *buffer* sendo maior e o fluxo sendo mantido, a quantidade de coordenadas que precisam aguardar no *buffer* é menor, o que leva a um processamento mais rápido e a consequente finalização do programa.

### 3.5..

### 6. Cenário 06 - Inversão de tarefas

Neste cenário, foi realizada uma simulação em que a função de geração de coordenadas aleatórias e o cálculo para determinar se este ponto foi gerado dentro do círculo ficarão presentes no código do cliente, enquanto o servidor está responsável apenas pelo somatório de coordenadas em cada área e pela função final .

Serão realizados testes com 4 clientes, mantendo os números anteriores de pontos e o *buffer* de 1024:

* 10.000 pontos - tempo total de processamento: 13 segundos
* 50.000 pontos - tempo total de processamento: 52 segundos
* 100.000 pontos - tempo total de processamento: 102 segundos

Observa-se neste cenário que o tempo de processamento diminuiu bastante em relação aos anteriores. As tarefas mais complexas no caso presente, em que o servidor e os clientes possuem capacidades similares, ao serem deslocadas para os clientes possibilitam uma distribuição da complexidade para estes equipamentos, o que diminui a carga de processamento no servidor, agilizando a finalização da tarefa.

### 3.5.7. Cenário 07 - Inversão de tarefas e retirada da função print

Com o intuito de testar mais uma possibilidade de melhoria no código, além das alterações executadas no Cenário 06, neste cenário também foi retirada a função *print* que imprime os logs do servidor e do cliente durante a execução do processamento.

Serão mantidos os testes com 4 clientes, mantendo também os números anteriores de pontos e o *buffer* de 1024, atingindo os seguintes resultados:

* 10.000 pontos - tempo total de processamento: 2 segundos
* 50.000 pontos - tempo total de processamento: 12 segundos
* 100.000 pontos - tempo total de processamento: 24 segundos

É possível observar que a redução de tempo de processamento foi drástica. A implementação deslocando a função de maior complexidade para os clientes em união com a retirada de processos de *I/O* (entrada/saída), que imprimiam mensagens a cada interação, apresentaram resultados muito mais eficazes que os iniciais. Por exemplo, na simulação realizada no Cenário 02, o tempo de processamento para uma amostra de 100.000 pontos foi de 233 segundos, enquanto neste cenário o tempo necessário para realizar esta tarefa foi de apenas 24 segundos. Desta forma, esta simulação apresentou um desempenho 90% melhor em relação ao Cenário 02.

4 - Conclusão

A proposta do trabalho de elaborar um sistema de computação voluntária distribuída para auxiliar no processamento de grandes quantidades de dados, como é o caso de pesquisas nas áreas de epidemiologia, à semelhança da problemática enfrentada ultimamente no país em relação à dengue; apresentou resultados interessantes.

Algumas adversidades foram observadas ao longo do desenvolvimento deste trabalho. Por exemplo, por questões de segurança a conexão de clientes e servidores por meio de redes com domínios diferentes, em que um servidor estaria em uma rede doméstica, poderia expor e comprometer a integridade dos sistemas da residência. Com isso, optou-se por realizar a conexão no ambiente do CEUB, porém não se tem conhecimento sobre as particularidades e configurações da rede local, o que se mostrou um complicador.

Também por essas restrições no sistema da universidade, o servidor não pôde ficar hospedado em uma máquina do laboratório, uma vez que não aceita receber pacotes desconhecidos, mesmo com uma fonte conhecida. Necessitou-se da utilização de um laptop pessoal para realizar esse papel.

Outro contratempo encontrado foi em relação à transparência de localização. Como não havia a garantia de um endereço IP fixo, em cada sessão de testes foi necessário alterar manualmente no código do cliente o endereço do servidor. Um servidor DNS facilitaria nesta questão, uma vez que seria necessária a mudança apenas no servidor, porém não houve sucesso nas tentativas de implementação deste modelo de servidor.

Os objetivos apresentados de escalabilidade e melhora no tempo de processamento foram atingidos com a aplicação, como demonstrado nos testes realizados em laboratório. Pôde-se demonstrar esses resultados com a adição de computadores clientes sem prejudicar o funcionamento e de maneira simples, em que apenas a execução do código cliente incluía a máquina na rede voluntária; e também com a melhora no processamento do algoritmo de Monte Carlo para estimativa de Pi.

Nestes testes, gargalos foram observados. As hipóteses levantadas anteriormente em relação à limitação de rede da sala do CEUB ou da placa de rede do servidor se mostraram equivocadas; uma vez que a conexão do CEUB é de alta capacidade, como observado tanto na estrutura física de cabo e *switch*, quanto na interface de rede do computador servidor.

Com isso, a hipótese de limitação dada pelo código se tornou objeto de estudo e um novo teste foi realizado, demonstrando que um fator limitador era o tamanho do *buffer*. O problema de gargalo foi reduzido, porém não é possível realizar um teste que conte com máquinas suficientes para saturar a capacidade de armazenamento do servidor novamente.

Por fim, os dois últimos testes realizados apresentaram excelentes resultados, demonstrando outros fatores existentes no código que deveriam ser aprimorados. A escolha inicial de deixar o processamento mais complexo no servidor, aos moldes de aplicações reais de computação distribuída voluntária, não se mostrou a mais eficiente na presente situação. Além disso, a retirada de funções *print* durante a execução também oferece uma melhora no desempenho do processamento de dados em relação ao código inicial, por diminuir sobrecarga de *I/O* (entrada/saída), uma vez que essas operações são significativamente mais lentas que operações de processamento puramente computacionais.

Dados estes resultados, demonstrou-se que a computação voluntária distribuída é um sistema viável e de acessível implementação. Em que a escalabilidade é facilmente obtida e, ao usar recursos ociosos dos computadores clientes para alcançar um objetivo específico, há uma melhora considerável na capacidade de processamento.

# 5 - Bibliografia

|  |  |
| --- | --- |
| 1. | MINISTÉRIO DA SAÚDE. Dengue. **Ministério da Saúde**. Disponível em: <https://www.gov.br/saude/pt-br/assuntos/saude-de-a-a-z/d/dengue>. Acesso em: 08 abril 2024. |
| 2. | MINISTÉRIO DA SAÚDE. Dengue. **Biblioteca Virtual em Saúde**. Disponível em: <https://bvsms.saude.gov.br/dengue-16/>. Acesso em: 08 abril 2024. |
| 3. | FIORATTI, G. Dengue explode no Sul e já alcança 95% das cidades brasileiras. **TAB UOL**, 06 abril 2024. Disponível em: <https://tab.uol.com.br/noticias/redacao/2024/04/06/dengue-explode-no-sul-e-ja-alcanca-95-das-cidades-brasileiras.htm >. Acesso em: 08 abril 2024. |
| 4. | BIENARTH, A. A 'tempestade perfeita' que explica explosão de casos de dengue no Brasil. **BBC News Brasil**, 27 março 2024. Disponível em: <https://www.bbc.com/portuguese/articles/ce7x6yn0r90o>. Acesso em: 08 abril 2024. |
| 5. | MINISTÉRIO DA SAÚDE. Governo Federal inicia mobilização contra a dengue nas escolas. **Ministério da Saúde**, 21 fevereiro 2024. Disponível em: <https://www.gov.br/saude/pt-br/assuntos/noticias/2024/fevereiro/governo-federal-inicia-mobilizacao-contra-a-dengue-nas-escolas>. Acesso em: 08 abril 2024. |
| 6. | INOUE, G. Em 3 meses, DF tem 10 vezes mais mortes por dengue do que em todo ano de 2023. **R7 Brasília**, 07 abril 2024. Disponível em: <https://noticias.r7.com/brasilia/em-3-meses-df-ja-teve-mais-de-10-vezes-o-numero-de-mortes-por-dengue-que-todo-o-ano-de-2023-07042024>. Acesso em: 08 abril 2024. |
| 7. | EIRAS, Y. RJ registra casos de dengue tipo 3, que não circulava no estado desde 2007. **Folha de S.Paulo**, 06 abril 2024. Disponível em: <https://www1.folha.uol.com.br/equilibrioesaude/2024/04/rj-registra-casos-de-dengue-tipo-3-que-nao-circulava-no-estado-desde-2007.shtm>. Acesso em: 08 abril 2024. |
| 8. | FOLDING@HOME. Stats. **Folding@home (via Archive.today)**, 2020. Disponível em: <https://archive.ph/20200412111010/https://stats.foldingathome.org/os>. Acesso em: 2024 maio 08. |
| 9. | RIGUES, R. Membros do projeto Folding@Home criam o maior supercomputador do mundo. **Olhar Digital**, 17 abril 2020. Disponível em: <https://olhardigital.com.br/2020/04/17/coronavirus/membros-do-projeto-folding-home-criam-o-maior-supercomputador-do-mundo/>. Acesso em: 13 abril 2024. |
| 10. | BOINC. Detailed stats - SETI@home. **Boinc Stats**. Disponível em: <https://www.boincstats.com/stats/0/project/detail/>. Acesso em: 13 abril 2024. |
| 11. | ROSETTA@HOME. Status do Projeto. **Rosetta@home**, 13 abril 2024. Disponível em: <https://boinc.bakerlab.org/rosetta/server\_status.php>. Acesso em: 13 abril 2024. |
| 12. | EINSTEIN@HOME. Einstein@Home server status page. **Einstein@Home**, 13 abril 2024. Disponível em: <https://einsteinathome.org/server\_status.php>. Acesso em: 13 abril 2024. |
| 13. | PEARSON, H. Wanted: computers for a humanitarian cause. **Nature (via Web Archive)**, 20 julho 2006. Disponível em: <https://web.archive.org/web/20160826162943/http://www.nature.com/news/2006/060717/full/news060717-15.html>. Acesso em: 13 abril 2024. |
| 14. | GAGLIARDI, H. F. Integramodel : um serviço de simulação de epidemias em grades computacionais. **Universidade Católica de Santos**, 15 junho 2007. Disponível em: <https://tede.unisantos.br/handle/tede/597>. Acesso em: 04 abril 2024. |
| 15. | MAGOULÈS, F. **Fundamentals of Grid Computing**. [S.l.]: Chapman & Hall/CRC, 2009. |
| 16. | WILKINSON, B. **Grid Computing:** Techniques ands Applications. [S.l.]: Chapman & Hall/CRC, 2009. |
| 17. | SARMENTA, L. F. G. **Volunteer Computing**. Massachusetts: Massachusetts Institute of Technology, 2001. |
| 18. | **Folding@home**. Disponível em: <https://foldingathome.org/>. Acesso em: 16 maio 2024. |
| 19. | **PrimeGrid**. Disponível em: <https://www.primegrid.com/>. Acesso em: 16 maio 2024. |
| 20. | O que é Asteroids@home? **Asteroids@home**. Disponível em: <https://asteroidsathome.net/boinc/>. Acesso em: 16 maio 2024. |
| 21. | STRBAC-SAVIC, S.; MILETIC, A.; STEFANOVIC, H. **The estimation of Pi using Monte Carlo technique with interactive animations**, Belgrado, Outubro 2015. Disponível em: <https://www.researchgate.net/publication/282909462\_The\_estimation\_of\_Pi\_using\_Monte\_Carlo\_technique\_with\_interactive\_animations>. Acesso em: 05 junho 2024. |
| 22. | KALOS, MALVIN H.; WHITLOCK, PAULA A. **Monte Carlo Methods**. [S.l.]: Wiley-VCH, 2008. |
| 23. | **Atlas@Home**. Disponível em: <https://atlas.cern/Resources/Atlasathome>. Acesso em: 05 junho 2024. |
| 24. | ALMEIDA, D. Número de mortes por dengue no Distrito Federal sobe para 77. **Agência Brasil**, 02 março 2024. Disponível em: <https://agenciabrasil.ebc.com.br/geral/noticia/2024-03/numero-de-mortes-por-dengue-no-df-sobe-para-77>. Acesso em: 08 abril 2024. |
| 25. | MINISTÉRIO DA SAÚDE. DistritoFederal recebe R$ 4 milhões para a assistência farmacêutica como reforço no combate à dengue. **Ministério da Saúde**, 25 março 2024. Disponível em: <https://www.gov.br/saude/pt-br/assuntos/noticias-para-os-estados/distrito-federal/2024/marco/distrito-federal-recebe-r-4-milhoes-para-a-assistencia-farmaceutica-como-reforco-no-combate-a-dengue>. Acesso em: 08 abril 2024. |
| 26. | MINISTÉRIO DA SAÚDE. “É momento de prevenir e cuidar”, reforça ministra da Saúde sobre dengue. **Ministério da Saúde**, 30 janeiro 2024. Disponível em: <https://www.gov.br/saude/pt-br/assuntos/noticias/2024/janeiro/201ce-momento-de-prevenir-e-cuidar201d-reforca-ministra-da-saude-sobre-dengue>. Acesso em: 08 abril 2024. |
| 27. | CASTRO, G. Aumento histórico de temperatura leva à disseminação da dengue em todo o Brasil. **Portal do Butantan**, 07 novembro 2023. Disponível em: <https://butantan.gov.br/noticias/aumento-historico-de-temperatura-leva-a-disseminacao-da-dengue-em-todo-o-brasil>. Acesso em: 08 abril 2024. |
| 28. | WESTIN, R. Dengue: clima, água parada e falhas do poder público causaram explosão de casos. **Agência Senado**, 16 fevereiro 2024. Disponível em: <https://www12.senado.leg.br/noticias/infomaterias/2024/02/dengue-clima-agua-parada-e-falhas-do-poder-publico-causaram-explosao-de-casos>. Acesso em: 08 abril 2024. |
| 29. | L. BARRETO , ; TEIXEIRA , M. G. Dengue no Brasil: situação epidemiológica e contribuições para uma agenda de pesquisa. **Scielo**, 10 outubro 2008. Disponível em: <https://www.scielo.br/j/ea/a/7FKpQj7MLZ7WbcGtfccxZrd/>. Acesso em: 08 abril 2024. |
| 30. | HOYOS, W.; AGUILAR, J.; TORO, M. An autonomous cycle of data analysis tasks for the clinical management. **Helyon**, outubro 2022. Disponível em: <http://crossmark.crossref.org/dialog/?doi=10.1016/j.heliyon.2022.e10846&domain=pdf>. Acesso em: 04 abril 2024. |
| 31. | REDAÇÃO NATIONAL GEOGRAPHIC BRASIL. Entenda o que são os pulsares, o fenômeno mais impactante das estrelas. **National Geographic**, 08 setembro 2023. Disponível em: <https://www.nationalgeographicbrasil.com/espaco/2023/09/entenda-o-que-sao-os-pulsares-o-fenomeno-mais-impactante-das-estrelas>. Acesso em: 13 abril 2024. |
| 32. | WHO is AFRICA@home? **Africa@Home (via Web Archive)**. Disponível em: <https://web.archive.org/web/20141030171807/https://africa-at-home.web.cern.ch/africa-at-home/presentation.htm>. Acesso em: 13 abril 2024. |
| 33. | LOVGREN, S. Malaria Battlers Enlist Power of Your PC. **National Geographic News (via Web Archive)**, 08 agosto 2006. Disponível em: <https://web.archive.org/web/20140902224121/http://news.nationalgeographic.com/news/2006/08/060808-malaria.html>. Acesso em: 13 abril 2024. |
| 34. | FOLDING@HOME. About. **Folding@home**. Disponível em: <https://foldingathome.org/about-2/>. Acesso em: 13 abril 2024. |
| 35. | BERKELEY Open Infrastructure for Network Computing. **Wikipedia**. Disponível em: <https://en.wikipedia.org/wiki/Berkeley\_Open\_Infrastructure\_for\_Network\_Computing>. Acesso em: 13 abril 2024. |
| 36. | THINK TWICE. Approximating Pi ( Monte Carlo integration ) | animation. **Youtube**, 2018. Disponível em: <https://www.youtube.com/watch?v=ELetCV\_wX\_c>. Acesso em: 10 junho 2024. |