UNIVERSIDADE DA REGIÃO DE JOINVILLE – UNIVILLE
Bacharelado em Engenharia de Software

ANALISANDO AS THREADS VIRTUAIS DA LINGUAGEM JAVA EM DIFERENTES CENÁRIOS

FELIPE ROSA
WALTER COAN

Joinville - SC
2022

FELIPE ROSA

ANALISANDO AS THREADS VIRTUAIS DA LINGUAGEM JAVA EM DIFERENTES CENÁRIOS

Projeto do Trabalho de Conclusão de Curso apresentado ao curso de Engenharia de Software da Universidade da Região de Joinville – Univille – como requisito parcial para obtenção do grau de Bacharel em Engenharia de Software, sob orientação do Professor Walter Coan.

Joinville - SC

2022

1. RESUMO

Este trabalho busca observar a utilização de *threads* virtuais na linguagem de programação Java e compará-la com as demais formas de utilização já existentes. Dado que algumas linguagens de programação com ampla utilização já oferecem mecanismos consolidados para lidar com concorrência sem dependência direta do sistema operacional e que a linguagem Java se trata de uma das mais importantes dentro do mundo da computação, se constatou ser importante analisar se a nova forma disponibilizada pela linguagem oferece vantagens quando comparado as demais implementações já existentes de concorrência. Para isso, um experimento é realizado por meio da utilização de servidores web que processem as requisições HTTP com os diferentes tipos de *threads* disponíveis em Java. A partir da execução são extraídos alguns indicadores como latência das requisições e uso de memória, poder de processamento e *threads* do sistema operacional. Como resultados preliminares, quando utilizadas *threads* virtuais se observou uma menor latência na resposta das requisições e diminuição na utilização de memória RAM pela JVM.

Palavras-chave: Java, concorrência, *threads* virtuais.

2. ABSTRACT

This work seeks to observe the use of virtual *threads* in the Java programming language and compare it with other existing ways of use. Given that some widely used programming languages already offer consolidated ways to deal with concurrency without direct dependence of the Operating System and that the Java language is one of the most important in the world of computing, it was found to be important to analyze whether the new way made available by the language offer advantages when compared with other existing implementations of concurrency. For this, an experiment is carried out using web servers that process HTTP requests with the different types of *threads* available in Java. Some indicators are extracted from the execution, such as latency of requests and memory usage, CPU use and operating system *threads*. As preliminary results, when using virtual *threads*, there is a lower latency in the response of requests and a decrease in the use of RAM memory by the JVM.

Keywords: Java, concurrency, virtual *threads*.

3. SUMÁRIO

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

GB	Gigabytes
JDK	Java Development Kit
JEP	JDK Enhancement Proposal
JVM	Java Virtual Machine
MB	Megabytes
RAM	Memória de acesso randômico
SO	Sistema Operacional
TIC	Tecnologias da Informação e Comunicação
URI	Identificador de recurso HTTP (Uniform Resource Identifier)
URL	Localizador de recurso HTTP (Uniform Resource Locator)

1 DEFINIÇÃO DO PROJETO

1.1 Tema

Nas últimas décadas têm se observado uma evolução constante dos componentes de hardware disponíveis para processamento computacional, que por consequência acabam demandando novas formas de programação para aproveitar ao máximo todo o processamento disponível.

Este trabalho atuará dentro da linha de pesquisa de Sistemas de Informação, Engenharia de Software e TICs, com concentração em Impacto do avanço da tecnologia na Engenharia de Software. A linha de pesquisa busca investigar as relações causais entre variáveis, analisando os efeitos positivos ou negativos das mudanças nas técnicas de programação. Para isso, faz uso de um ambiente experimental a ser manipulado e observado e preconiza situações em que determinadas variáveis estão sob o controle do pesquisador.

O objeto do trabalho é realizar uma análise comparativa entre as novas implementações feitas na linguagem de programação Java em seu ambiente de execução (JVM), que implementam o conceito de *threads* virtuais para compreender os impactos dessa nova tecnologia no desenvolvimento de aplicações e de sistemas de informação.

1.2 Motivação e Descrição do Problema

Conforme previsto pela declaração de Moore (1965), o aumento do poder computacional nos *chips* evoluiu de maneira exponencial nas últimas décadas, feito inclusive por meio da criação de *chips* com mais de um núcleo disponível para processamento. Com o incremento de hardware é esperado que as linguagens de programação utilizadas pelos desenvolvedores para acessar e gerenciar os recursos do dispositivo ofereçam formas de tirar o máximo de proveito e visando maior eficiência na utilização dos recursos computacionais, como memória e poder de processamento.

Segundo o ranking que mede a popularidade de linguagens de programação (TIOBE Index, 2022), a linguagem Java se destaca pela ampla utilização, sempre estando entre as três mais utilizadas pelos desenvolvedores mundialmente. Todo esse sucesso se deve a um histórico de confiabilidade na linguagem devido a algumas

de suas características chave, destacando-se (Schildt, 2015): a linguagem ser interpretada (gera um bytecode otimizado que é interpretado pelo ambiente de execução da linguagem), neutralidade de arquitetura e portabilidade (o bytecode gerado pode ser utilizado em qualquer sistema operacional que tenha o ambiente de execução), robustez (é uma linguagem fortemente tipada e que executa verificações de tempo de execução), além de oferecer suporte integrado para programação com várias *threads*.

Apesar de tamanha importância no mundo da computação, a linguagem Java ainda não oferece em sua versão de suporte de longo prazo (LTS) um mecanismo de concorrência através de *threads* leves, que serão exploradas posteriormente. Algumas linguagens que estão abaixo do Java no TIOBE Index já implementaram esse mecanismo em suas versões oficiais, como Golang (Klyuev, 2014) e Kotlin (linguagem esta que roda no mesmo ambiente de execução que o Java, na JVM), sendo amplamente adotado pelos desenvolvedores em seus ecossistemas de ferramentas.

Conforme a utilização de *threads* leves foi se expandindo, a linguagem Java recentemente buscou implementar o mesmo mecanismo em sua plataforma através do Projeto Loom, sendo liberada para testes em uma versão prévia, dentro da expedição da versão JDK 19. Dada a relevância que essa técnica trouxe ao ecossistema de outras linguagens, é interessante que esse novo recurso seja comparado com as demais formas de programação com *threads* disponíveis na linguagem Java, buscando entender em que cenários a utilização do novo recurso seria mais indicado.

Portanto, a motivação para este trabalho se dá pela relevância da linguagem Java no mundo da computação e da necessidade de analisar se a forma com que a linguagem implementou o conceito de *threads* leves é de fato mais eficiente do que as formas já existentes de se trabalhar com concorrência dentro da linguagem.

1.3 Objetivos

1.3.1 Objetivo Geral

Analisar o comportamento de *threads* virtuais na linguagem de programação Java, comparando com outras formas de utilização de concorrência já existentes na linguagem.

1.3.2 Objetivos Específicos

- Introduzir conceitos sobre sistemas operacionais, processos e especialmente sobre a utilização de *threads*, além de como funcionam dentro da linguagem Java.
- Verificar a forma indicada pela linguagem para utilização de *threads* virtuais, observando as diferenças com a implementação de *threads* comuns disponíveis na linguagem.
- Definir os cenários em que serão testadas as execuções com *threads* e com *threads* virtuais, a serem posteriormente comparados.
- Executar os programas desenvolvidos com base nos cenários definidos e coletar informações acerca dos recursos computacionais consumidos, como tempo de execução, consumo de memória e de processamento.
- Comparar o comportamento entre as duas implementações de *threads* em cada cenário, buscando identificar em quais cenários, determinada implementação se mostra mais eficiente.

1.4 Justificativa e Contribuições

Nesta pesquisa será analisada a execução das *threads* virtuais disponibilizadas pela linguagem Java, comparando com outras formas de utilização de concorrência já disponíveis na linguagem. Por ser um recurso disponibilizado recentemente para testes da comunidade de usuários da linguagem de programação, entende-se que é importante que sejam feitos testes e comparações que elevem o entendimento acerca da implementação de *threads* leves na plataforma Java, que hoje carece de comparações mais detalhadas.

Após realizada a análise, os desenvolvedores terão melhor conhecimento acerca dos cenários indicados para utilização de *threads* virtuais e poderão aplicá-las de forma mais racional às suas aplicações, visando ganho de performance e consequentemente, redução nos custos de execução das aplicações.

Além disso, a escolha do tema se justifica porque deve unir conhecimento de áreas como Sistemas Operacionais e da própria Engenharia de Software, trazendo relevante carga de conhecimento ao estudante pesquisador.

Devido a linguagem liberar suas versões de forma gratuita, a pesquisa se torna viável ao ter acesso ao ambiente de execução na versão necessária, além de toda a especificação da implementação realizada estar disponível nos documentos construídos pelos times que implementaram a funcionalidade.

1.5 Organização do trabalho

Este trabalho está organizado em quatro capítulos, sendo eles: (1) Definição do projeto; (2) Revisão bibliográfica; (3) Proposta e Procedimentos Metodológicos e (4) Projeto Mínimo Viável.

Na Definição do Projeto é descrito o tema do trabalho, detalhando também a motivação encontrada para escolha do tema e uma breve descrição do problema. Além disso, são definidos o Objetivo Geral e os Objetivos Específicos que irão guiar o trabalho.

Na Revisão Bibliográfica, o tema é contextualizado através da elucidação dos conceitos necessários para entendimento do problema. Ainda nesta seção são citadas fontes que ajudam a compreender o problema e um artigo que tem estreita relação com a análise a ser realizada pelo trabalho.

Em Proposta e Procedimentos Metodológicos é detalhado como a pesquisa foi feita, sob quais condições o experimento foi realizado e quais técnicas foram utilizadas para coletar e analisar os dados do trabalho.

No capítulo reservado ao MVP – Projeto Mínimo Viável, são detalhados os resultados obtidos e após o fim do trabalho são listadas as referências utilizadas para embasar o desenvolvimento do trabalho.

2. REVISÃO BIBLIOGRÁFICA

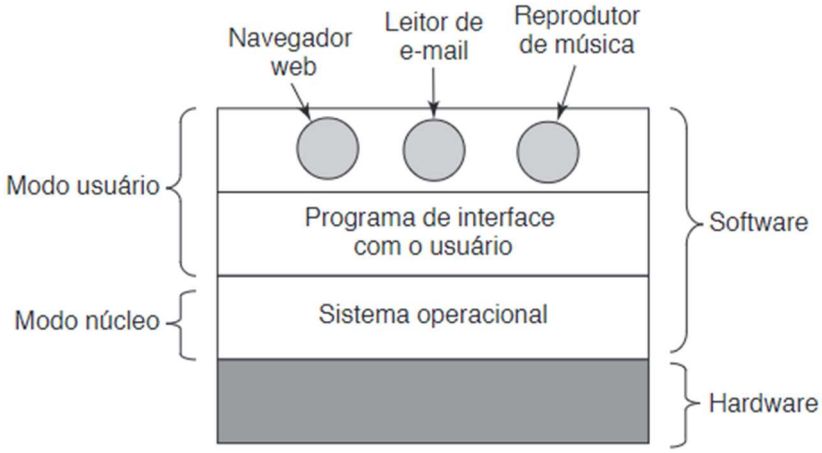
2.1 Conceitos

Para entender as situações em que *threads* são utilizadas e a necessidade de criação das chamadas *threads* virtuais, alguns conceitos devem ser esclarecidos, tendo como base o conceito de sistema operacional.

Segundo Tanenbaum (2016), um sistema operacional serve, dentre outras coisas, para gerenciar complexidade e faz isso principalmente através de abstrações. Isso permite que os usuários não necessitem conhecer cada operação de gerenciamento de memória ou dos detalhes dos registradores da máquina utilizada. Uma perspectiva simplificada pode ser representada dividindo o sistema operacional em três camadas, como retratado na figura 1.

Uma parte é a de hardware, onde se localizam os componentes do computador, como chips, discos e monitor. As outras duas partes indicadas por Tanenbaum são de software: um referente a operação em modo núcleo, que executa a parte do software que tem acesso a todo o hardware e tem permissão para executar qualquer instrução disponibilizada pela máquina; e operação no modo usuário, que pode executar apenas um subconjunto limitado de instruções e onde o usuário utiliza seus programas, como navegador de internet, editores de texto e leitor de e-mails.

Figura 1: onde o sistema operacional se encaixa.



Fonte: Tanenbaum *et al.* (2016).

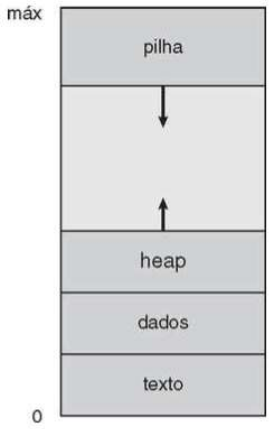
Entende-se então que o sistema operacional atua como intermediário entre os componentes de hardware e as aplicações de usuário, tendo papel essencial em

otimizar os recursos de hardware disponíveis. Para Silberschatz (2016), um dos aspectos mais importantes para a boa execução de um sistema operacional está baseado na capacidade de multiprogramação.

Devido a necessidade do usuário de utilizar várias aplicações ao mesmo tempo, além do próprio sistema operacional ter que controlar inúmeras funções e possíveis interrupções (como um clique, novo dispositivo de entrada/saída etc.) é necessário que o tempo de processamento na CPU seja devidamente compartilhado entre os vários processos em execução, organizando as tarefas de forma que a CPU sempre tenha algo a ser executado. Para que um sistema operacional possa executar multiprogramação, é essencial que as tarefas a serem executadas estejam definidas de alguma forma padronizada, que é onde o conceito de processos e *threads* se mostram necessários.

Um processo, para Silberschatz (2016), é definido de forma simplificada como um programa em execução. Ele considera que diferente de um programa, que por si só é uma entidade passiva (lista de instruções a serem seguidas definidas em algum arquivo na memória), um processo é o programa sendo executado pelo computador. O programa se torna um processo a partir do momento em que o arquivo executável é carregado na memória. Em uma definição mais completa, o processo deve conter: uma pilha com os dados temporários (stack), memória que pode ser alocada dinamicamente durante o tempo em execução (heap), seção de dados contendo as variáveis globais (dados) e ainda, o código do programa para execução (seção de texto), representado na figura 2.

Figura 2: Processo na memória.



Fonte: Silberschatz *et al.* (2016, p. 80).

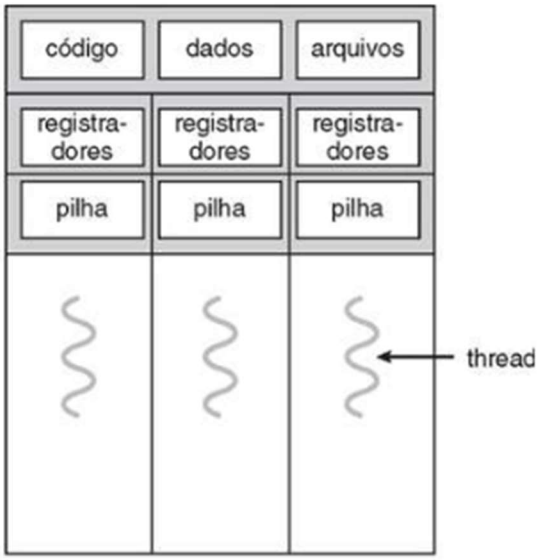
Do ponto de vista do sistema operacional, ele é definido através de um Bloco de Controle do Processo (PCB), que contém informações como o estado do processo (ex.: em execução, bloqueado, esperando), o número (PID), o contador do programa (que indica o endereço da próxima execução), dentre outras propriedades de natureza do processo.

Inicialmente, um processo (programa em execução) seria definido contendo apenas um *thread* de controle (unidade de execução) para um espaço de endereçamento. Para atender a necessidade de se executar mais de um procedimento dentro do mesmo processo, os sistemas operacionais evoluíram permitindo que um processo pudesse ser dividido em mais de uma unidade de execução dentro do mesmo espaço de endereçamento, que contribui para o surgimento e utilização de *threads*.

Segundo Silberschatz, um *thread* é uma unidade básica de utilização da CPU, composta basicamente do ID da *thread*, um contador do programa, um conjunto de registradores e uma pilha, sendo uma estrutura mais simples que um processo. Para ilustrar a necessidade de mais de um *thread* de controle dentro de um mesmo processo, é razoável pensar em um navegador de internet – que é definido no sistema operacional como um processo, mas que necessita executar de forma quase paralela o tráfego de informações dentro da rede em um *thread*, enquanto lê os arquivos de apresentação e apresenta na aba do navegador em outra *thread*.

Quando um mesmo processo agrega mais de um *thread* de controle, alguns recursos podem ser compartilhados enquanto várias execuções são realizadas. O processo divide o acesso a arquivos, aos dados globais e ao código, enquanto cada *thread* mantém suas informações em sua própria pilha e nos registradores, conforme ilustrado na figura 3.

Figura 3: Um processo com múltiplas *threads*.

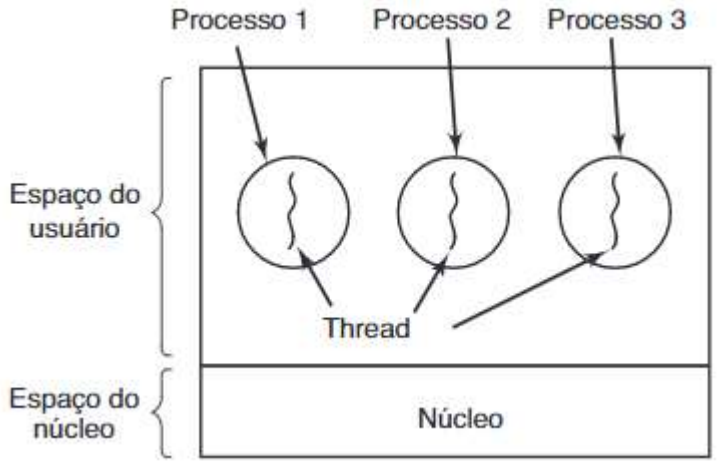


Fonte: Silberschatz *et al.* (2016, p. 118).

Tanenbaum (2016) destaca cinco razões em que a utilização de *threads* pode ser útil: (1) em aplicações com muitas atividades ao mesmo tempo, geralmente interrompidas por bloqueios de algum processamento em andamento ou à espera de uma informação trafegando na rede, simplificado através da decomposição das tarefas em múltiplas *threads* não bloqueantes entre si; (2) maior facilidade e velocidade para criar e destruir quando comparado aos processos, por não ter recursos associados à *thread*; (3) em programas de fluxo intenso de entrada e saída, que podem ganhar velocidade pela possibilidade de sobrepor a execução de atividades sempre que bloqueadas; (4) por serem especialmente úteis em sistemas com múltiplas CPUs.

Ao se constatar o benefício de *threads*, a definição de processo pode ser reconsiderada como uma forma de se agrupar recursos, onde o processo apresenta o espaço de endereçamento com o código e os dados do programa, além do acesso aos outros possíveis recursos, como arquivos abertos e processos filhos. De forma simplificada, enquanto processos agrupam recursos (incluindo as próprias *threads*), *threads* são utilizadas como entidades a serem escalonadas para execução na CPU (Tanenbaum, 2016).

Figura 4: Três processos, cada um com um *thread*.



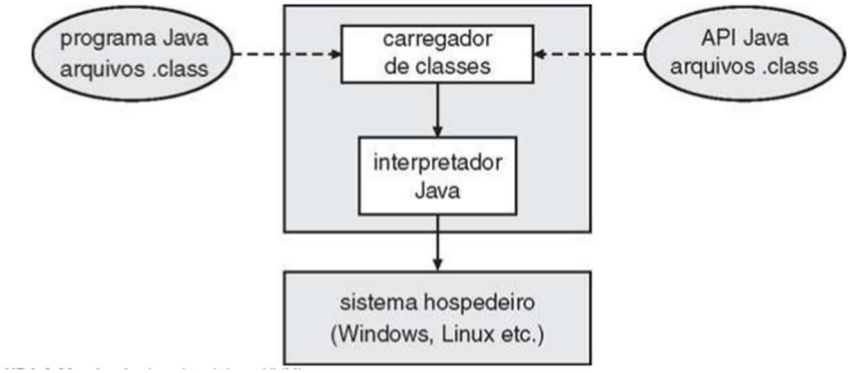
Fonte: Tanenbaum *et al.* (2016, p. 71).

Como introduzido anteriormente, conforme os sistemas operacionais evoluem as linguagens de programação buscam formas de contemplar essas mudanças dentro de suas classes padrão, traçando uma interface para utilizar os recursos do sistema operacional dentro dos padrões da linguagem.

Em Java, tanto processos como *threads* podem ser implementados através de diversas classes que a linguagem disponibiliza. Para *threads* especificamente, existem vários métodos e propriedades para operações como definição de prioridade da *thread*, retornar dados de estado de execução ou fazer chamadas de bloqueio.

Conforme apresentado por Silberschatz (2016), a linguagem Java é independente de arquitetura devido a forma como ela é executada, independente de qual sistema operacional: todo o código estruturado nos arquivos “.java” são compilados para um código de bytes independente de arquitetura (bytecode). O bytecode gerado então pode ser lido pela JVM (Java Virtual Machine), que é composta por um carregador de classes (class loader) e um interpretador Java. O class loader realiza uma inspeção prévia no arquivo “.class” a ser executado e verifica se é um bytecode válido e que não faz acessos indevidos à memória. Após passar pela validação, o interpretador Java lê o bytecode e o executa no sistema operacional da JVM, processo representado na figura 5.

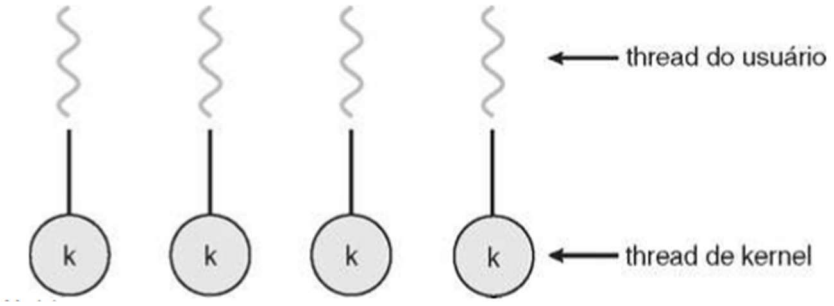
Figura 5: A máquina virtual Java (JVM).



Fonte: Silberschatz *et al.* (2016, p. 62).

Na execução, a máquina virtual da linguagem então implementa o código definido, se adaptando de acordo com cada arquitetura do sistema operacional hospedeiro. A JVM utiliza o modelo de multi*thread*ing um para um nos principais sistemas operacionais (Windows e Linux). Nesse modelo, para cada *thread* criada dentro da JVM, outra *thread* deve ser criada no sistema operacional hospedeiro. O modelo se mostra conveniente principalmente em ambientes com multiprocessadores, já que permite que as múltiplas *threads* possam ser executadas de forma paralela, diminuindo o tempo de processamento. O modelo é representado na figura 6.

Figura 6: Modelo de multi*thread*ing um para um (*thread* JVM para *thread* do SO).



Fonte: Silberschatz *et al.* (2016, p. 120).

Apesar do modelo facilitar a concorrência ao criar uma *thread* no sistema operacional para cada uma da JVM, como consequência pode haver uma redução no desempenho da aplicação ao se implementar uma quantidade expressiva de *threads*,

já que o sistema operacional pode restringir o número de *threads* criadas afim de evitar o esgotamento de recursos do sistema operacional.

Para contornar esses riscos, a própria linguagem Java disponibilizou formas de se evitar a criação de um *thread* para cada tarefa. Uma alternativa para o problema está na utilização de bancos de *threads*. No modelo de bancos de *threads* a ideia é que seja criado um número limitado de *threads* que ficam disponíveis para executarem determinadas tarefas. Quando disponível, executa uma tarefa recebida e após a execução retorna para o banco de *threads*. Segundo Silberschatz (2016), esse modelo oferece dois principais benefícios: (1) atende tarefas com *threads* já existentes, sem necessidade de esperar a criação da *thread* na JVM e no sistema operacional hospedeiro; (2) limita o número de *threads* que podem ser criadas, diminuindo o risco de causar esgotamento de recursos da máquina.

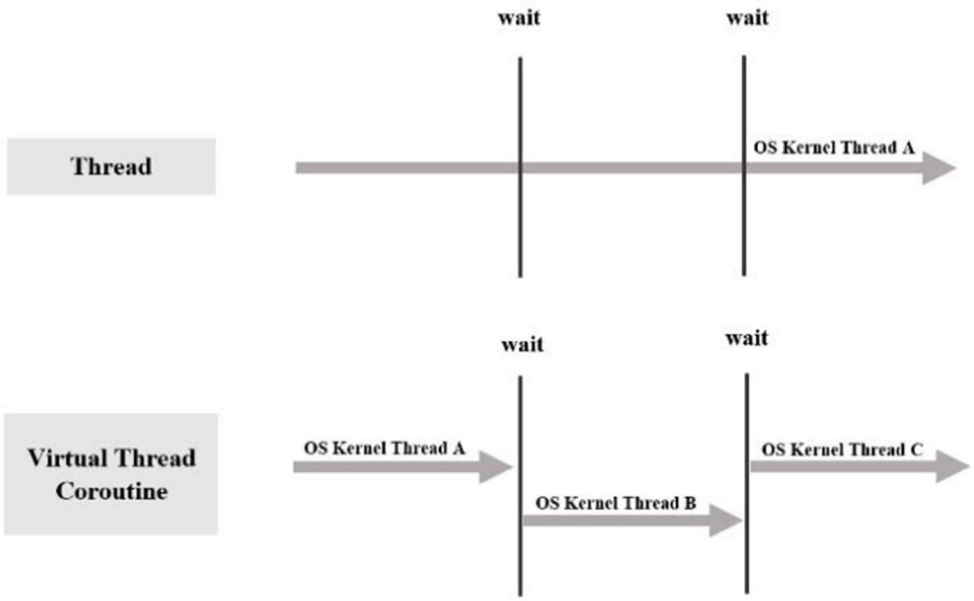
Na API da linguagem Java, pode ser utilizado o conceito de "*Executor*" (uma forma de executar tarefas sem precisar vincular a um *thread* específico). Neste modelo, destacam-se três principais formas de utilização dos bancos de *threads*: (1) banco com apenas um *thread* ("*newSingleThreadExecutor*"); (2) banco de *threads* com um número fixo, que deve ser especificado ("*newFixedThreadPool*") e (3) banco de *threads* ilimitado, mas que mantém o conceito de reutilizar *threads* já criadas para executar as tarefas ("*newCachedThreadPool*").

Ainda que a linguagem tenha disponibilizado uma forma de se reaproveitar *threads*, diminuindo esforço e tempo de processamento para criar e descartar *threads*, o risco de esgotamento de recursos da máquina ainda é grande, principalmente em casos em que não se sabe qual o número ideal de *threads* a serem utilizadas ao criar o banco de *threads*.

O papel das *threads* leves, ou *threads* virtuais, está justamente em diminuir o gasto de recursos computacionais empenhados na utilização de *threads*. No atual modelo, a concorrência se dá pela instância da classe "java.lang.Thread", que por sua vez é executada em uma *thread* correspondente do sistema operacional durante toda a vida útil do código (Pressman, 2021). A proposta que o modelo de *threads* virtuais traz na JEP 425 (forma de inserir mudanças no ambiente de execução da linguagem Java) é de que o código de um *thread* Java continue sendo executado em um *thread* do sistema operacional, mas que não seja exclusiva de uma única *thread* da JVM,

mas utilizada por várias delas. Neste modelo, um *thread* virtual que está executando uma tarefa pode ser executado em diferentes *threads* do sistema operacional, sempre que houver alguma operação que bloqueie ou que esteja esperando por algum resultado que não seja de processamento na CPU, conforme ilustrado na figura 7.

Figura 7: Relação de um *thread* comum e de *thread* virtual executando no *thread* do S.O.



Fonte: Beronić *et al.* (2022, p. 1468).

A partir dessa proposta, são introduzidos novos conceitos: *plarform threads* (modelo tradicional de *threads*, um *thread* da JVM é vinculado em um *thread* do S.O.), *virtual threads* (*thread* que é criada e gerenciada pela JVM, mas sempre que precisar ser executada na CPU, é executada na *thread* do sistema operacional) e *carrier threads* (são *threads* criadas pela JVM no S.O. para executar tarefas das virtual *threads*).

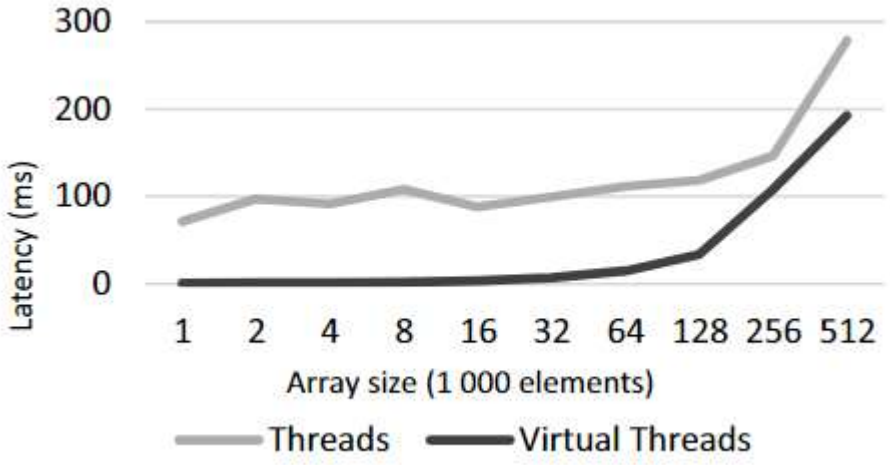
2.2 Comparações existentes

Apesar de *threads* virtuais terem sido recentemente desenvolvidas na linguagem Java, alguns pesquisadores realizaram experimentos comparando uma das versões prévias liberadas pela linguagem e encontraram resultados que indicam a melhoria na performance de *threads* virtuais em relação aos *threads* tradicionais. O teste foi executado em uma máquina com sistema operacional Linux e com uma versão da JVM compatível com os *threads* virtuais (Beronić e*t al*, 2021). Os programas

executados eram algoritmos de ordenação por mistura (merge sort), de ordenação sequencial e de execução paralela. O experimento buscou comparar métricas de velocidade de criação de *threads*, concorrência e nível de eficiência em operações bloqueantes. Devido a forma que a JVM executa e para garantir certa conformidade nos resultados, os pesquisadores executaram os programas duas vezes antes de coletar os dados. Isso evita que os processos como validação do bytecode e compilação realizados no início pela JVM interferissem nos dados.

Nos resultados coletados, observou-se que: referente ao tempo de criação de *threads*, o tempo para criar um milhão de *threads* comuns (platform *threads*) foi 2,23 vezes mais lento que a criação do mesmo número de *threads* virtuais. Quanto a execução, no algoritmo de Merge Sort em um array com 64000 elementos, observou-se que utilizando *threads* virtuais a ordenação foi 30,82% mais rápida do que utilizando os *threads* de plataforma.

Figura 8: Tempo de solução do Merge Sort conforme tamanho do vetor, utilizando *threads* e virtual *threads*.



Fonte: Beronić *et al.* (2021, p. 1688).

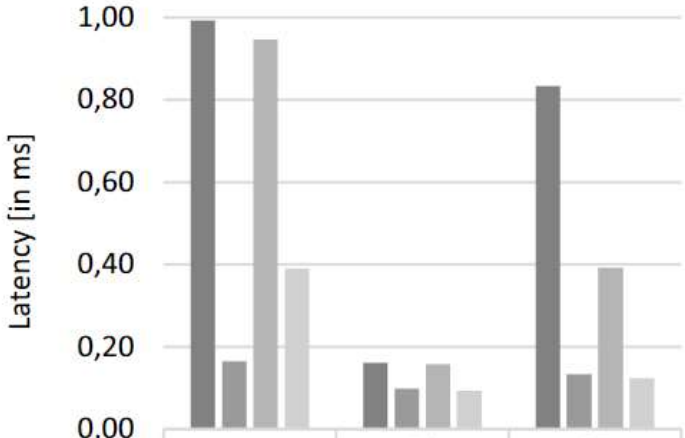
Os pesquisadores concluíram que os *threads* virtuais melhoraram a performance nos cenários testados e que resolveram os problemas de forma mais eficiente, sem precisar adicionar recursos na máquina que executou os testes (Beronić e*t al*).

Em outro artigo (Beronić et al, 2022), os pesquisadores fizeram comparações entre *threads* tradicionais e as virtuais do Java, além das *threads* tradicionais da linguagem Kotlin e as chamadas “coroutines”, nome dado aos *threads* virtuais desta linguagem. A comparação faz sentido visto que a linguagem Kotlin roda no mesmo

ambiente de execução do Java, a JVM. O caso de teste é diferente do artigo anterior: agora foi construído um servidor web que recebe requisições em uma porta da máquina e que a cada requisição, dá início a um número de *threads*, variando o tipo delas de acordo com o teste.

Dentre as métricas colhidas, a latência observada apresenta uma diferença expressiva, sempre em favorecimento aos *threads* virtuais, em ambas as plataformas, sendo menos que a metade da constatada em *threads* da plataforma.

Figura 9: Média de latência de resposta às requisições utilizando diferentes formas de *threads*.



	mean	50	95
Java - threads	0,9921	0,1615	0,8331
Java - virtual threads	0,1642	0,0988	0,1329
Kotlin - threads	0,9464	0,1582	0,3915
Kotlin - coroutines	0,3896	0,0931	0,1234

Fonte: Beronić *et al.* (2022, p. 1470).

Em ambos os testes mencionados os pesquisadores não realizaram testes comparando outras métricas, como consumo de memória e testes utilizando outras soluções de se trabalhar com *threads* em Java, como os bancos de *threads*, mencionados anteriormente neste trabalho.

3 PROPOSTA E PROCEDIMENTOS METODOLÓGICOS

3.1 Caracterização do delineamento da pesquisa

Este trabalho utiliza uma abordagem de pesquisa quantitativa, ou seja, através de experimentações (execução de programas com *threads* Java) busca examinar a relação causal entre variáveis a serem medidas. Quanto aos objetivos, se trata de uma pesquisa explicativa, justamente por buscar explicar determinados comportamentos com base em experimentações. Para isso, faz uso de experimentos de processamento computacional, coleta de dados e informações que apoiem o aprofundamento no entendimento do fenômeno a ser estudado.

3.2 Caracterização da população/ amostra ou caracterização do ambiente experimental ou caracterização do caso

Para execução dos experimentos, é utilizado o ambiente de testes com as seguintes características: um computador com sistema operacional Ubuntu versão 20.04 LTS sendo executado em um hardware de 8GB de memória RAM e processador Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, composto de 8 núcleos virtuais para processamento.

3.3 Técnicas e ferramentas de coleta de dados

O experimento é feito através da execução de algoritmo em linguagem Java que faz uso dos métodos e classes disponibilizados pela linguagem para ativar diferentes servidores WEB que atendem requisições HTTP. Em cada requisição é repassado um número aleatório que após o processamento do programa recebe como resposta uma mensagem indicando se o número aleatório informado é ou não um número primo.

Os servidores web utilizados são definidos através de instâncias da classe “HttpServer” que recebem uma série de informações acerca da execução a ser empregada. Para cada servidor web são definidos: a porta do sistema operacional que será utilizada para receber/responder as requisições (indicado no método “create”); um contexto, que define a rota e qual operação será feita em cada requisição (método “createContext”) e um tipo de “Executor”, ou seja, com que tipo de execução será feito o processamento da requisição.

As variações nos cenários de teste se dão com base no tipo de servidor definido no método “setExecutor”. Para este experimento foram testados com quatro variações: (1) “new*Thread*PerTaskExecutor”: cenário em que para cada requisição recebida a linguagem Java tenta criar um *thread* no SO para executar as operações, representando a forma mais tradicional de execução de *threads* na linguagem; (2) “newSingle*Thread*Executor”: cenário em que é utilizado um banco de *thread* com apenas um único *thread* para executar todas as requisições recebidas, podendo ser reutilizado entre cada requisição; (3) “newCached*Thread*Pool”: cenário em que é utilizado um banco de *threads* que vão sendo criadas conforme a necessidade e a disponibilidade de recursos da máquina, reutilizando os *threads* já criados; (4) “newVirtual*Thread*PerTaskExecutor”: cenário que utiliza a nova forma de execução com *threads* da linguagem Java, onde para cada requisição recebida é criado um *thread* virtual (dentro do ambiente de execução) e esses *threads* virtuais compartilham tempo de execução entre alguns *threads* do sistema operacional.

A operação executada em cada requisição é a mesma independentemente do tipo de *thread* que a processará, consistindo em: ler a URL da requisição e identificar o número a ser verificado; criar um objeto “PrimeNumber” na memória passando para o construtor do objeto o número identificado; escrever uma linha em um arquivo formato “.CSV” informações como o resultado do processamento (se é um número primo ou não), se teve ou não uma interrupção na execução (atraso no processamento com comando “*Thread*.sleep”), a sequência de números que o algoritmo tentou dividir até assumir se o número era ou não primo; retornar a requisição com código de status HTTP como sucesso (200) e a frase indicando o resultado.

Na execução dos experimentos são coletados dados em duas partes distintas: cliente (busca quantificar o tempo de resposta para retornar cada requisição, em cada variação de execução com *threads*) e servidor (quanto de memória, processamento e *threads* do SO foram utilizados para processar as requisições).

Para coleta de dados do cliente é utilizada a ferramenta Vegeta, a mesma utilizada na análise de Beronić (2021). Nesta ferramenta é possível definir através de linha de comando ou em um script de execução para terminal Linux quantas requisições serão lançadas por segundo, a URL de destino a ser testada e se terá variação no URI da URL. Após a execução a ferramenta consolida os resultados das

requisições trazendo em um relatório dados como o número de requisições enviadas, os resultados por tipo de resposta, a quantia de dados trafegados em bytes, média de latência, dentre outros.

Para coleta de informações do lado do servidor é utilizada a ferramenta VisualVM, que coleta e permite exportar informações como uso de CPU da máquina, memória reservada e utilizada pela JVM na execução, além do número de *threads* sendo executadas.

Afim de melhorar o fluxo de execução dos testes, as tarefas que fazem as chamadas do programa Vegeta para cada tipo de teste foram definidas dentro de arquivos ".sh". Neste script também foram definidos intervalos de 120 segundos entre cada execução para evitar que os indicadores da aplicação demonstrados na VisualVM não fossem afetados por resultados do teste anterior. Importante ressaltar que em todos os experimentos foram utilizados os mesmos números aleatórios, evitando variações nos resultados devido testes não uniformes de execução.

3.4 Técnicas e ferramentas para análise de dados

A análise de resultados feita leva em consideração os indicadores gerados por ambas as partes da execução: os dados do lado do cliente (latência média de resposta) e as métricas levantadas pela VisualVM sobre a execução Java. Na análise todos os dados são colocados em uma planilha e dela são extraídos os gráficos que ajudam a visualizar de uma melhor forma os resultados por tipo de execução.

Os dados de latência média são fornecidos pela própria ferramenta Vegeta, que retorna após o fim da execução um consolidado com a média de tempo para resposta de cada requisição que obteve sucesso. Nos quatro cenários de teste executados, todos tiveram 100% de sucesso no retorno das respostas. Com os valores definidos na planilha, é gerado um gráfico de barras que indica por tipo de execução a latência média auferida em milissegundos.

Os indicadores gerados pela ferramenta VisualVM contém dados acerca da aplicação Java, dados esses que são coletados a cada segundo. Essas informações são então exportadas em formato ".CSV" e para serem analisadas são importadas dentro de uma aba na planilha Excel. A partir dos dados gerados por esta ferramenta são analisados três indicadores: percentual de uso de CPU pelo processo da JVM,

que indica o quanto de processamento do computador a JVM de fato exigiu ao executar em cada um dos modelos de concorrência; o tamanho da memória Heap, que indica quanto da memória RAM do computador foi ocupado pela JVM, sendo dividida entre a parte reservada e a parte realmente utilizada dinamicamente pela JVM para alocar os objetos necessários para a execução e por último, quantas *threads* do sistema operacional estavam em uso pelo processo da JVM durante cada execução. Para estes três indicadores, a partir das aferições consolidadas por segundo, são construídos os três gráficos de linha que indicam por tipo de execução as variações observadas.

3.5 Cronograma e Plano de Ação

O quadro abaixo apresenta as etapas a serem cumpridas para a execução do projeto, assim como o período de realização de cada etapa.

Quadro 1 - Cronograma e plano de ação (2022/2023)

Etapas	MAR	NOV	DEZ	JAN	FEV	MAR	ABR	MAI	JUN
Definir linha de pesquisa e tema do projeto.	X								
Escrita das definições do projeto.		X							
Escrita da Revisão Bibliográfica do tema escolhido.		X							
Detalhamento da técnica de pesquisa e do experimento adotado.		X							
Execução do experimento detalhado e extração dos resultados.		X							
Planejamento e reavaliação do trabalho com base nas correções realizadas.			X	X	X				
Construção e execução dos novos experimentos a serem analisados.					X	X	X		
Extração e detalhamentos dos resultados do novo experimento.							X	X	
Escrita do artigo final.								X	X

Fonte: Elaborado pelo autor (2022).

4. MVP – PROJETO MÍNIMO VIÁVEL

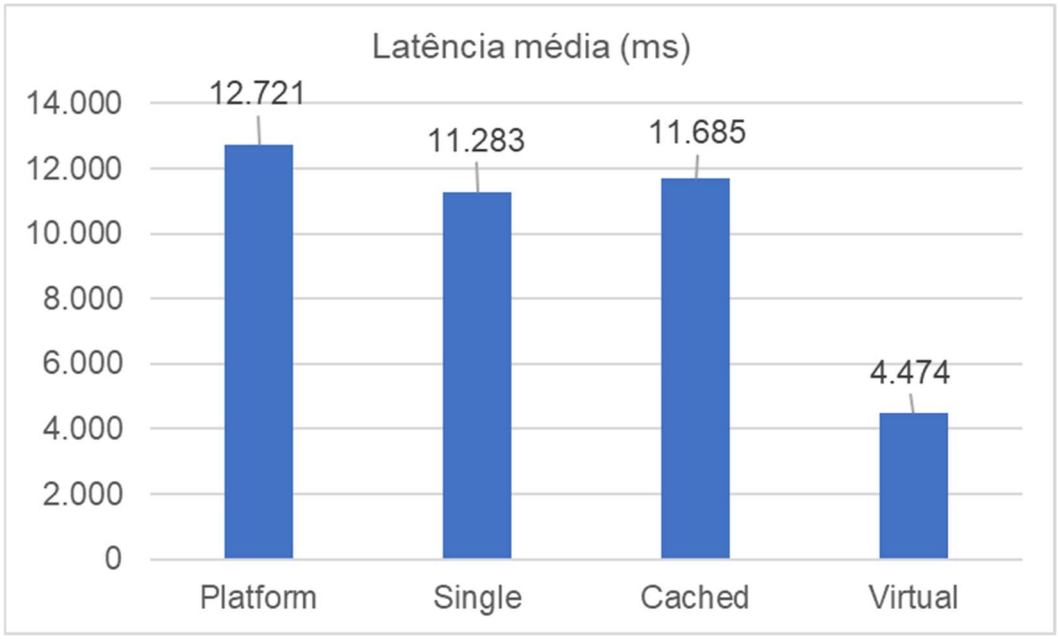
A partir dos métodos e técnicas descritos no capítulo anterior, após as execuções dos programas de teste e da coleta e consolidação dos dados gerados, alguns resultados preliminares já podem ser constatados.

A execução auferida se trata de quatro servidores web, cada um disponível em um socket de rede, que recebem requisições HTTP. No experimento são feitos a cada segundo, 500 requisições para o servidor sendo testado durante 5 segundos seguidos – totalizando 2500 requisições tratadas pelo servidor web em teste.

Cada servidor web trata as requisições recebidas com um tipo de *Executor* diferente, sendo identificados nos gráficos de resultados da seguinte forma: *Platform* (modelo de *threads* tradicionais, onde cada *thread* da JVM deve estar relacionada a um *thread* do SO), *Single* (*Executor* de apenas uma *thread* para todas as execuções), *Cached* (*Executor* onde a JVM cria e elimina os *threads* com base na necessidade constatada para utilização) e *Virtual* (modelo onde várias *threads* da JVM compartilham alguns *threads* do SO). Para cada requisição executada, a ferramenta Vegeta aufere a diferença de tempo entre o envio da requisição e o retorno da resposta (latência). Após todas as requisições serem devidamente tratadas a ferramenta consolida os resultados e calcula a média de tempo em que as requisições tiveram que aguardar para serem respondidas em milissegundos (para comparação, 1000 milissegundos equivalem a 1 segundo), representado no gráfico de barras abaixo.

Quanto a latência, no experimento se observa uma expressiva queda no tempo médio de resposta com a utilização das *Virtual Threads*, comparando com os demais executores testados.

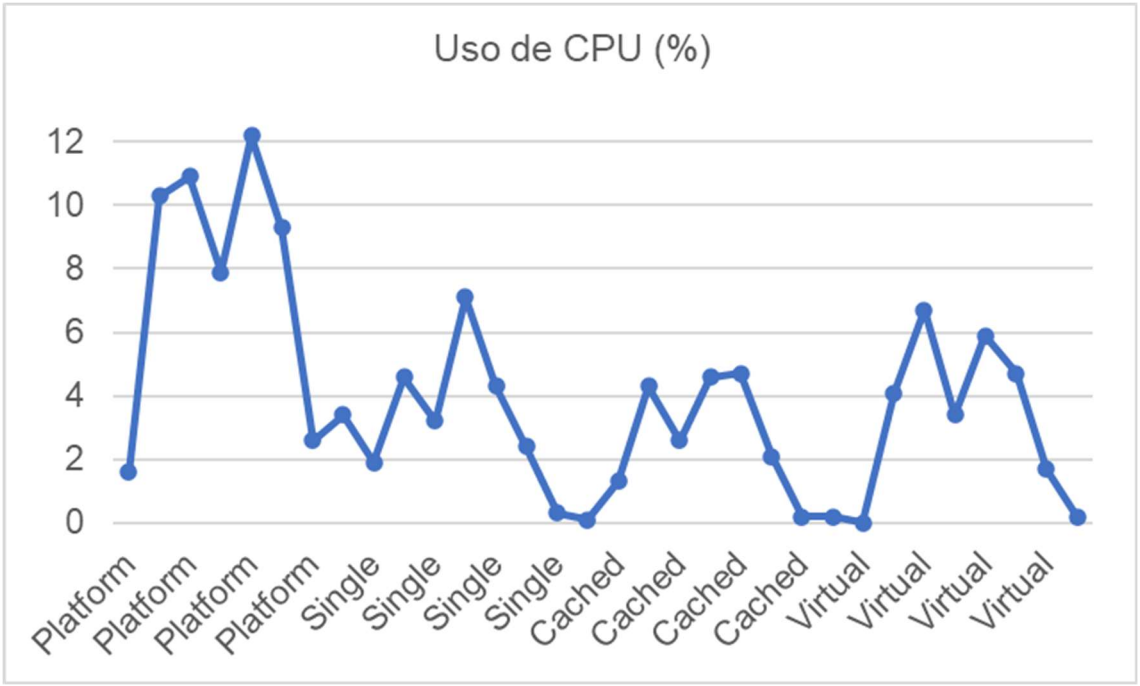
Figura 10: Latência média das requisições utilizando diferentes formas de *threads*.



Fonte: Elaborado pelo autor (2022).

Do ponto de vista do processo da JVM no computador, mais alguns indicadores podem ser extraídos. Quanto ao consumo de CPU pela JVM durante cada tipo de execução, observou-se uma variação pequena. O pico é observado quando executado com *threads* de plataforma, forma mais tradicional de utilização em Java.

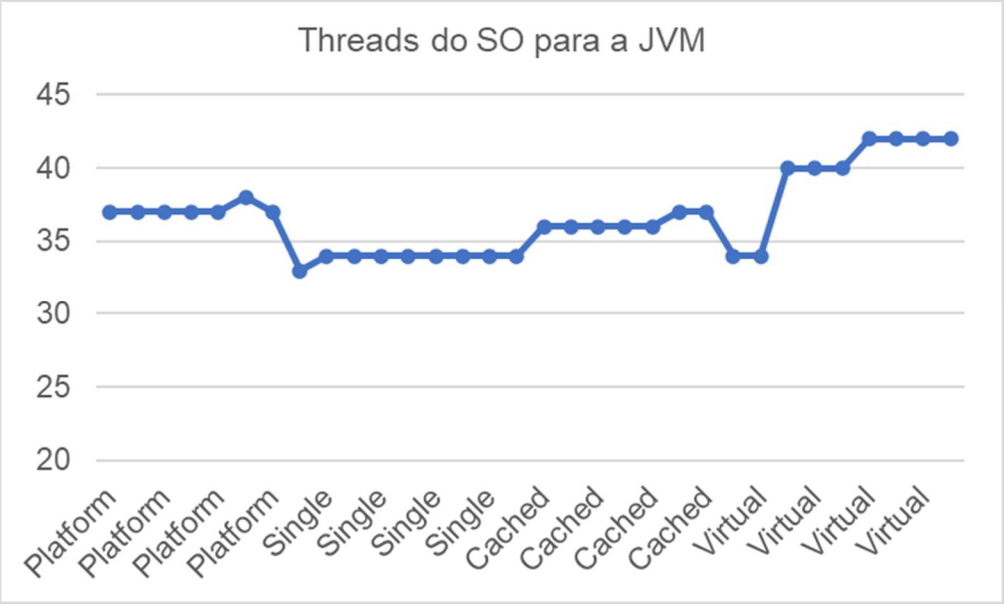
Figura 11: Percentual da CPU em uso pela JVM.



Fonte: Elaborado pelo autor (2022).

Quanto ao número de *threads* do sistema operacional em uso pelo processo da JVM, na execução com *threads* virtuais há um aumento em relação as demais execuções, conforme demostrado na figura abaixo.

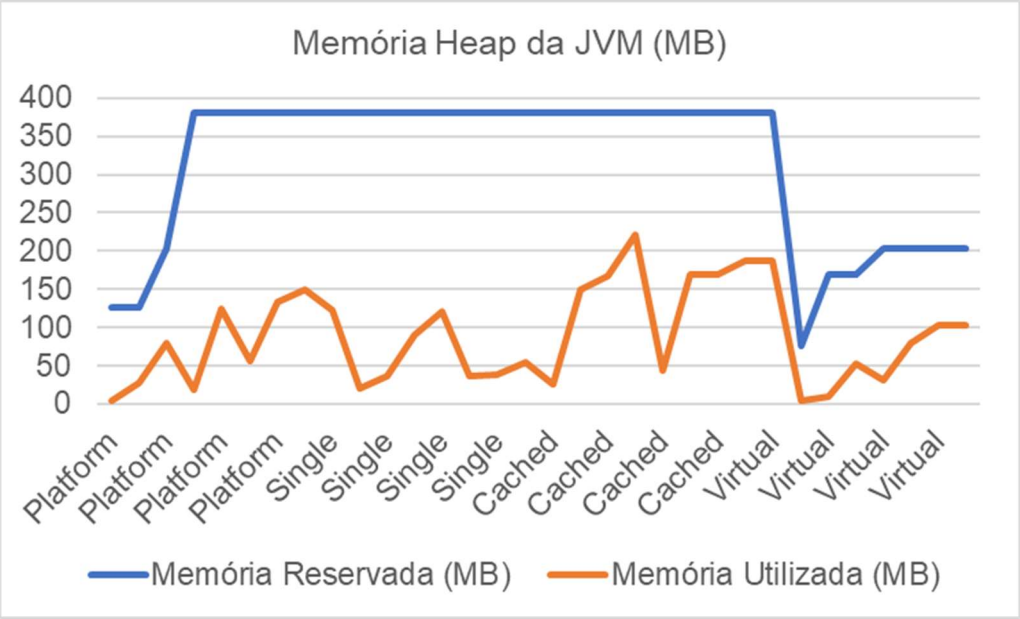
Figura 12: *Threads* do SO em uso pela JVM.



Fonte: Elaborado pelo autor (2022).

Por fim, o último indicador extraído da JVM é referente a utilização de memória RAM do computador pela JVM. Neste indicador se observa uma expressiva queda de memória reservada e utilizada para execução com *threads* virtuais.

Figura 13: Memória RAM em uso pela JVM em megabytes (MB).



Fonte: Elaborado pelo autor (2022).

Portanto, neste primeiro experimento com alguns cenários de utilização de *threads* Java, se constatou que o modelo de *threads* leves (*virtual threads*) tem resultados promissores, destacando-se a menor utilização de memória RAM e diminuição na latência das requisições realizadas no experimento.

É importante ainda destacar que a utilização de *threads* leves na JVM ainda está em fase de testes e desenvolvimento pelos desenvolvedores da linguagem, podendo apresentar melhoras operacionais em futuros experimentos conforme novas versões forem liberadas.

5 REFERÊNCIAS BIBLIOGRÁFICAS

BERONIĆ, D.; PUFEK, B.; MIHALJEVIĆ, B.; RADOVAN, A. **On Analyzing Virtual *Threads* – a Structured Concurrency Model for Scalable Applications on the JVMs**, MIPRO 2021. 01 out 2021.

BERONIĆ, D.; MODRIĆ, L.; MIHALJEVIĆ, B.; RADOVAN, A. **Comparison of Structured Concurrency Constructs in Java and Kotlin – Virtual *Threads* and Coroutines**, MIPRO 2022. 23 maio 2022.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim; BLAIR, Gordon. **Sistemas Distribuídos**. Grupo A, 2013. E-book. ISBN 9788582600542.

DA SILVA, Fabricio Machado; LEITE, Márcia Cristina D.; OLIVEIRA, Diego Bittencourt D. **Paradigmas de programação**. Grupo A, 2019. E-book. ISBN 9788533500426.

KLYUEV, Vitaly; TOGASHI, Naohiro. **Concurrency in Go and Java: Performance Analysis**, 4th IEEE International Conference on Information Science and Technology. 2014.

MOORE, Gordon E. **Cramming More Components onto Integrated Circuits**, Reprinted from Electronics, Volume 38, Number 8, April 19, 1965, Pp.114 Ff." IEEE Solid-State Circuits Society Newsletter 11.3 (2006): 33-35.

PRESSLER, Ron; BATEMAN, Alan. **JEP 425: Virtual *Threads* (Preview)**. Disponível em: https://openjdk.org/jeps/425. Acesso em 05 nov. 2022.

SCHILDT, Herbert. **Java para iniciantes**. 6ª edição. Oracle Press, 2015.

SILBERSCHATZ, Abraham; GALVIN, Peter B.; GAGNE, Greg. **Sistemas Operacionais com Java**. 8ª edição. Elsevier Editora, 2016.

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. 4ª edição. Pearson, 2016.

TIOBE INDEX. **TIOBE Index for October 2022**. Disponível em: https://www.tiobe.com/tiobe-index/. Acesso em 2 nov. 2022.

VEGETA. **Vegeta: HTTP load testing tool and library**. Disponível em: https://github.com/tsenart/vegeta/. Acesso em 5 dez. 2022.

VISUALVM. **VisualVM: All-in-one Java troubleshooting Tool**. Disponível em: https://visualvm.github.io/. Acesso em 5 dez. 2022.