UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

HIARLY FERNANDES DE SOUTO

UMA ANÁLISE COMPARATIVA ENTRE THREADS E GREEN THREADS NO JAVA

CAMPINA GRANDE - PB

2024

HIARLY FERNANDES DE SOUTO

UMA ANÁLISE COMPARATIVA ENTRE THREADS E GREEN THREADS NO JAVA

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador : Thiago Emmanuel Pereira

CAMPINA GRANDE - PB

2024

HIARLY FERNANDES DE SOUTO

UMA ANÁLISE COMPARATIVA ENTRE THREADS E GREEN THREADS NO JAVA

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Thiago Emmanuel Pereira

Orientador – UASC/CEEI/UFCG

Franklin de Souza Ramalho

Examinador – UASC/CEEI/UFCG

Francisco Vilar Brasileiro

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 15 de Maio de 2024

CAMPINA GRANDE - PB

RESUMO

Green thread é um modelo de thread no qual o escalonamento é realizado por runtimes ao invés de ser realizado pelo sistema operacional. Essa abordagem necessita de menos recursos, em termos de memória e ciclos de CPU, do que projetos tradicionais de threads. Recentemente, a linguagem Java, em sua versão 19, introduziu uma implementação de Green threads, chamada de Virtual Threads. Por ser ainda pouco utilizada, conhecemos pouco da eficiência desta implementação. Neste trabalho, avaliamos o desempenho da implementação de green threads de Java. Comparamos os resultados desta implementação com a implementação padrão já disponível na linguagem. Ao término deste trabalho, foi evidenciado que as Virtual Threads apresentam uma significativa melhoria de desempenho em comparação com a abordagem convencional de Threads em Java, sendo superior em todos os testes realizados.Os resultados deste estudo destacaram a disparidade de desempenho entre diferentes abordagens de manipulação de threads, com as Virtual Threads mostrando um desempenho excepcional em comparação com as threads convencionais em Java. Ficou claro que sistemas que possuem um grande número de threads, podem obter melhorias significativas de desempenho ao implementar as Virtual Threads em sua estrutura.

A COMPARATIVE ANALYSIS BETWEEN THREADS AND GREEN THREADS IN JAVA

ABSTRACT

Green thread is a thread model in which scheduling is done by runtimes instead of the operating system. This approach requires fewer resources in terms of memory and CPU cycles than traditional thread designs. Recently, the Java language, in its version 19, introduced an implementation of Green threads called Virtual Threads. Because it is still not widely used, we know little about the efficiency of this implementation. In this work, we evaluated the performance of Java's green thread implementation. We compared the results of this implementation with the standard implementation already available in the language. At the end of this work, it was evident that Virtual Threads show a significant performance improvement compared to the conventional Threads approach in Java, being superior in all tests conducted. The results of this study highlighted the performance disparity between different thread handling approaches, with Virtual Threads showing exceptional performance compared to conventional threads in Java. It became clear that systems with a large number of threads can achieve significant performance improvements by implementing Virtual Threads in their structure.

Uma análise comparativa entre Threads e Green Threads no Java

Hiarly Fernandes de Souto
Universidade Federal de Campina Grande
hiarly.souto@ccc.ufcg.edu.br

Thiago Emmanuel Pereira
Universidade Federal de Campina Grande
temmanuel@computacao.ufcg.edu.br

RESUMO

Green thread é um modelo de thread no qual o escalonamento é realizado por runtimes ao invés de ser realizado pelo sistema operacional. Essa abordagem necessita de menos recursos, em termos de memória e ciclos de CPU, do que projetos tradicionais de threads. Recentemente, a linguagem Java, em sua versão 19, introduziu uma implementação de Green threads, chamada de Virtual Threads. Por ser ainda pouco utilizada, conhecemos pouco da eficiência desta implementação. Neste trabalho, avaliamos o desempenho da implementação de green threads de Java. Comparamos os resultados desta implementação com a implementação padrão já disponível na linguagem. Ao término deste trabalho, foi evidenciado que as Virtual Threads apresentam uma significativa melhoria de desempenho em comparação com a abordagem convencional de Threads em Java, sendo superior em todos os testes realizados.Os resultados deste estudo destacaram a disparidade de desempenho entre diferentes abordagens de manipulação de threads, com as Virtual Threads mostrando um desempenho excepcional em comparação com as threads convencionais em Java. Ficou claro que sistemas que possuem um grande número de threads, podem obter melhorias significativas de desempenho ao implementar as Virtual Threads em sua estrutura.

Keywords

Virtual Threads, Java, Green Thread, Thread

1. INTRODUÇÃO

A linguagem de programação Java é bastante utilizada. Por exemplo, em 2022, Java aparece como 3° colocada no ranking de popularidade de acordo com uma análise do Github [1].

Por ser amplamente utilizada, várias funcionalidades vêm sendo acrescentadas ao longo dos lançamentos das versões do Java. Nesse sentido, recentemente, em sua versão 19, o Java introduziu uma implementação de Green Threads, chamada de Virtual Threads. Green threads é um design de threads no qual o escalonamento é feito pela própria aplicação por meio de uma biblioteca ou framework. Isso pode resultar em um escalonamento mais eficiente para determinados tipos de cargas de trabalho, pois a aplicação tem mais controle sobre como as threads são distribuídas e pode otimizar a alocação de recursos conforme necessário. Essa abordagem se contrapõe ao modelo tradicional de threads, onde o escalonamento é realizado pelo Sistema Operacional. No modelo tradicional, o escalonador do sistema operacional é responsável por decidir quando cada thread é executada, com base em políticas de escalonamento como FIFO (First-In, First-Out) ou Round Robin. Isso geralmente envolve trocas de contexto de kernel, o que pode ter um custo maior em termos de desempenho.

Por ser uma tecnologia nova no contexto da linguagem Java e, portanto, ainda pouco utilizada e difundida entre os programadores, é natural que se tenha poucas informações acerca da tecnologia de Virtual Threads. Diante dessa problemática, é fundamental que os desenvolvedores detenham informações sobre o desempenho dessa nova funcionalidade a fim de tomar decisões conscientes sobre sua aplicação.

Com base nisso, o objetivo deste trabalho é realizar uma análise de desempenho da implementação de green threads em Java. Para alcançar esse objetivo, será apresentada uma fundamentação sobre Threads e Green Threads, com uma explicação breve de como cada abordagem funciona e suas diferenças.

Além disso, serão conduzidos testes de Criação, Inicialização, Junção e Troca de Contexto usando as abordagens mencionadas para trabalhar com threads, com o intuito de medir qual das duas metodologias é mais eficiente. Por fim, será demonstrado que Virtual Threads apresentam uma melhoria significativa em relação à abordagem convencional de trabalho com Threads na linguagem Java.

Ao concluir este trabalho, será demonstrado que as Virtual Threads apresentam melhorias significativas em comparação com os Threads convencionais do Java, superando-as em todos os testes realizados. Isso evidenciará que sistemas com um grande número de threads podem aproveitar as melhorias dessa nova abordagem de gerenciamento de Threads, resultando em ganhos substanciais de desempenho para suas aplicações.

Esse artigo está organizado da seguinte forma: a Seção 2 apresenta o background, a Seção 3 descreve a avaliação enquanto que a Seção 4 apresenta os resultados. Já a Seção 5 discute os resultados, enquanto que as seções seguintes apresentam os trabalhos relacionados e as conclusões.

2. FUNDAMENTAÇÃO TEÓRICA

Esta seção irá discutir conceitos teóricos de Threads e Green Threads e sua aplicação na linguagem de programação Java

2.1. Threads

O conceito de threads, em um sistema operacional, refere-se a um fluxo sequencial de execução de instruções, seja de código das aplicações ou de sistema operacional. Um processo pode ter múltiplas threads, cada uma representando uma sequência independente de execução [2].

Threads diferentes em um processo não são independentes quanto processos diferentes [2]. Todos os threads têm exatamente o mesmo espaço de endereçamento, assim compartil

ham memória, o que pode criar condições de corrida. Por outro lado, processos distintos possuem espaços de memória separados o que, em alguns casos, dificulta a escrita de sistemas que precisam promover a colaboração entre os fluxos de execução.

Como um processo tradicional (isto é, um processo apenas com uma thread), uma thread pode estar em qualquer um de vários estados: em execução, bloqueado, pronto, ou concluído. Um thread em execução tem a CPU naquele momento e está ativo. Em comparação, um thread bloqueado está esperando por algum evento para desbloqueá-lo, um thread pronto está programado para ser executado e o será tão logo chegue a sua vez. [2]

2.2. Green Threads

Green threads têm suas raízes na necessidade de tornar a programação concorrente mais eficiente e portátil. O conceito surgiu em oposição às threads nativas, que são gerenciadas pelo sistema operacional. Enquanto as threads nativas são eficazes em termos de paralelismo real, elas também são caras em termos de recursos e podem ser difíceis de usar em algumas situações de programação.

A capacidade de criar e gerenciar threads leves em nível de usuário dá aos desenvolvedores mais controle sobre como suas aplicações concorrentes são executadas, ao mesmo tempo em que mantêm a portabilidade e a eficiência.

Em um esforço contínuo para melhorar a eficiência e a simplicidade da programação concorrente em Java, o conceito de "Virtual Threads" surgiu como uma inovação significativa. O Java introduziu Virtual Threads na versão 19, como parte do projeto Loom [3], com o objetivo de aprimorar o modelo de programação concorrente da linguagem e abordar as limitações dos tradicionais threads leves (Green Threads) e threads nativas.

Virtual Threads são mais leves que as threads do kernel em termos de uso de memória, e a sobrecarga de troca de contexto entre elas é muito baixa. Milhões de Virtual Threads podem ser criadas em uma única instância de JVM [3].

3. METODOLOGIA

Com o propósito de estabelecer uma comparação entre Virtual Threads e Threads convencionais no contexto Java, utilizou-se o artigo de referência "Comparative Performance Evaluation of Java Threads for Embedded Applications: Linux Thread vs. Green Thread" [4]. Neste estudo, executou-se um microbenchmark comparativo, focado nos métodos compartilhados entre Virtual Threads e Threads.

Cabe ressaltar que alguns dos métodos avaliados no artigo original podem não refletir as condições atuais pois estão depreciados nas versões atuais do java. Levando isso em consideração, optou-se por conduzir as seguintes medições: instanciação de uma thread, tempo para iniciar uma thread, tempo para aguardar pela conclusão das threads e tempo para alterar contexto de execução de uma thread.

Para facilitar a coleta de métricas, recorreu-se à biblioteca org.openjdk.jmh [5]. Esta biblioteca de benchmarking aborda de maneira simples aspectos secundários inerentes a microbenchmarks em Java, tais como a fase de aquecimento (warm-up) e automações de código. Isso possibilita que o usuário isole de forma eficaz e descomplicada o código que deseja testar, concentrando-se nas métricas essenciais para uma análise precisa do desempenho.

A máquina empregada nos testes tem as seguintes especificações: processador AMD® Ryzen 7 3700u, 20GB de memória RAM, disco com capacidade de 256GB, executando o sistema operacional Ubuntu 22.04.03 LTS.

Cada método do microbenchmark a seguir foi executado com 30 repetições, com uma execução inicial adicional como fase de aquecimento. O valor final apresentado após as execuções representa a média total em segundos.

3.1. Instanciar Threads

Para executar o microbenchmark da instanciação de threads, criou-se 100.000 threads e mediu–se o tempo de execução para a criação desses objetos.

Devido às otimizações inerentes do compilador Java, ao implementar apenas a criação da thread usando new Thread(), o compilador ignoraria a instanciação do objeto, pois ele não seria utilizado em nenhum local. Diante dessa problemática, a biblioteca JMH (Java Microbenchmarking Harness) fornece uma solução chamada BlackHole para lidar com esses cenários.

Assim, é suficiente instanciar a thread dentro de um consumidor BlackHole, uma abstração que não realiza nenhuma operação, mas impede que o compilador Java exclua o código da criação da thread. Com essa abordagem, foi possível alcançar o objetivo de medir o tempo de criação das threads de forma mais precisa.

3.2. Iniciar Threads

Para realizar o microbenchmark da instanciação de threads, criou-se 100.000 threads e mediu–se o tempo de execução para a criação desses objetos.

Como indicamos anteriormente acerca das otimizações do compilador Java, utilizou-se a abstração BlackHole para conduzir o teste de iniciar threads.

3.3 Join Threads

Com o propósito de avaliar o tempo necessário para concluir a execução de um conjunto de threads, iniciou-se um total de 10.000 threads que não realizavam operações específicas. Estas threads foram devidamente armazenadas em uma lista para posterior utilização no processo de benchmarking.

Uma vez que as threads foram criadas, o método empregado no benchmarking consistia em iniciar as threads utilizando o método .start() e, posteriormente, aguardar a conclusão de todas elas. Esse processo era realizado por meio de um laço, que invocava o método .join() de cada thread, garantindo que o programa principal aguardasse a finalização de todas as threads antes de prosseguir.

Este método proporcionou uma abordagem eficiente para medir o desempenho do sistema em relação à execução simultânea de um grande número de threads, permitindo uma análise detalhada do tempo total necessário para a conclusão do conjunto de threads em questão.

3.4. Mudar Contexto (Yield) de Threads

O método Thread.yield(), de acordo com a documentação oficial do java, fornece um mecanismo para informar ao escalonador que a thread atual está disposta a renunciar ao seu uso atual do processador, mas gostaria de ser reagendada assim que possível.

Para avaliar esse comportamento, foram criadas inicialmente 100.000 (cem mil) threads, que foram então armazenadas em uma lista. Cada thread aguarda por 100 milissegundos, invoca o método Thread.yield(), e em seguida, aguarda por mais 100 milissegundos. Esse procedimento tem o objetivo de simular uma operação bloqueante em uma thread (uma operação de I/O ou um cálculo matemático "demorado") e a troca de contexto entre threads.

Após a instanciação das threads com o comportamento descrito, um laço foi percorrido para invocar o método .start(), permitindo assim verificar o tempo necessário para que as threads fossem executadas.

Por fim, realizou-se o agrupamento das threads com o método Thread.join(), seguido pela limpeza da lista que continha as threads por meio da invocação do método .clear() da interface List<> do Java. Esse procedimento foi encapsulado por meio da anotação @TearDown da biblioteca JHM, evitando que esse procedimento fosse levado em consideração no benchmarking.

4. RESULTADOS

Todos os benchmarking levaram cerca de 2 horas para serem executados, cujo resultados serão explicados a seguir.

4.1. Instanciar Threads

Método	Tempo médio (s)	Desvio padrão (s)
Thread	0.178s	0.006s
Virtual Thread	0.001s	0.001s

Tabela 1: Desempenho na instanciação de Threads

Os resultados mostrados na Tabela 1, na Figura 1 e na Figura 2 revelam uma notável disparidade de desempenho entre a biblioteca Thread e as Virtual Threads. Enquanto a criação de threads na biblioteca Thread leva aproximadamente 0.178 segundos, as Virtual Threads demonstram uma eficiência significativamente superior, exigindo apenas 0.001 segundos.

Diante dessa discrepância, conduzimos um teste t de Student, com um nível de significância de 0,05, para investigar se existe uma diferença significativa nos tempos médios de execução entre Virtual Threads e Threads. A hipótese nula sugerindo que não há diferença significativa entre os métodos, e a alternativa indicando uma disparidade significativa.

Os resultados do teste revelam um valor de p próximo a 0, indicando uma diferença estatisticamente significativa entre os tempos médios de execução dos métodos. Portanto, rejeitamos a hipótese nula e concluímos que há uma diferença substancial nos tempos médios de execução entre Virtual Threads e Threads.

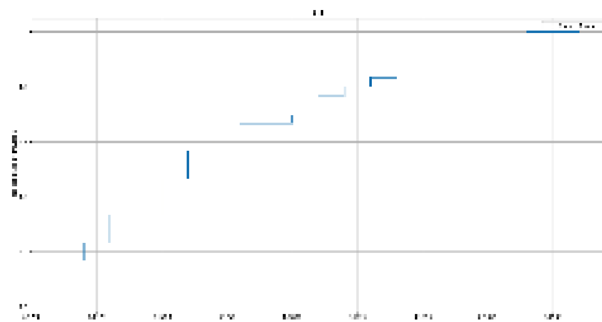

Figura 1: ECDF dos resultados associados a criação de Threads

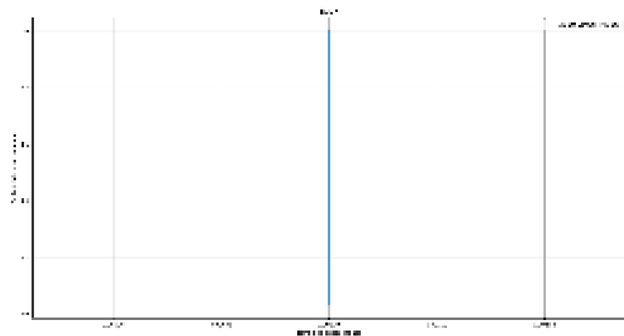

Figura 2: ECDF dos resultados associados a criação de Virtual Threads

4.2. Iniciar Threads

Método	Tempo médio (s)	Desvio padrão (s)
Thread	1.099s	0.008s
Virtual Thread	0.002s	0.001s

Tabela 2: Desempenho na inicialização de Threads

Os resultados evidenciados na Tabela 2, na Figura 3 e na Figura 4 destacam uma significativa discrepância de desempenho entre a biblioteca Thread e as Virtual Threads. Enquanto a inicialização de threads na biblioteca Thread demanda aproximadamente 1.099 segundos, as Virtual Threads demonstram uma eficiência notavelmente superior, exigindo apenas 0.002 segundos.

Diante dessa discrepância, realizamos um teste t de Student, com um nível de significância de 0,05, para examinar se há uma diferença significativa nos tempos médios de inicialização entre Virtual Threads e Threads. A hipótese nula sugere que não há diferença significativa entre os métodos, enquanto a alternativa indica uma disparidade significativa.

Os resultados do teste revelam um valor de p próximo a 0, indicando uma diferença estatisticamente significativa entre os tempos médios de execução dos métodos. Portanto, rejeitamos a hipótese nula e concluímos que há uma diferença substancial nos tempos médios de inicialização entre Virtual Threads e Threads.

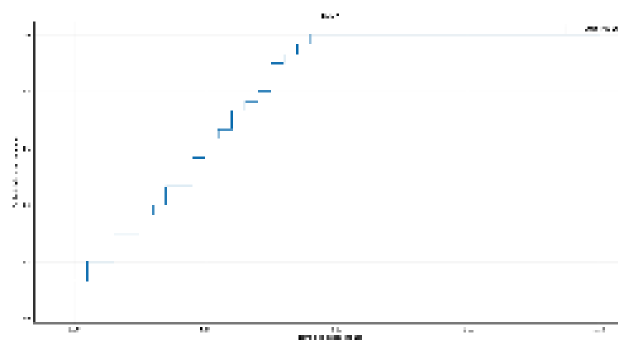

Figura 3: ECDF dos resultados associados a inicialização de Threads

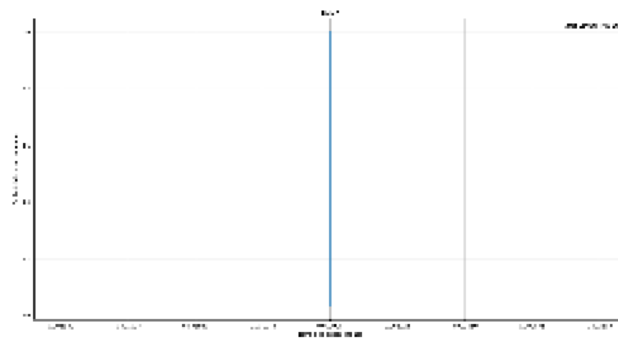

Figura 4: ECDF dos resultados associados a inicialização de Virtual Threads

4.3 Join Threads

Método	Tempo médio (s)	Desvio padrão (s)
Thread	1.093s	0.005s
Virtual Thread	0.002	0.001s

Tabela 3: Desempenho na junção de Threads

Os resultados mostrados na Tabela 3, na Figura 5 e na Figura 6 evidenciam uma discrepância de desempenho entre a biblioteca Thread e as Virtual Threads. Enquanto a junção de threads na biblioteca Thread demanda aproximadamente 1,093 segundos, as Virtual Threads demonstram uma eficácia superior, necessitando de apenas 0.002 segundos.

Diante dessa discrepância, realizamos um teste t de Student, com um nível de significância de 0,05, para investigar se há uma diferença significativa nos tempos médios de junção entre Virtual Threads e Threads. A hipótese nula sugere que não há diferença significativa entre os métodos, enquanto a alternativa indica uma disparidade significativa.

Os resultados do teste revelam um valor de p próximo a 0, indicando uma diferença estatisticamente significativa entre os tempos médios de execução dos métodos. Portanto, rejeitamos a hipótese nula e concluímos que há uma diferença substancial nos tempos médios de junção entre Virtual Threads e Threads.

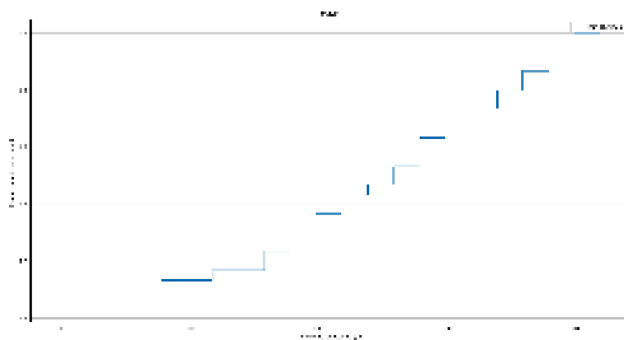

Figura 5: ECDF dos resultados associados a junção de Threads

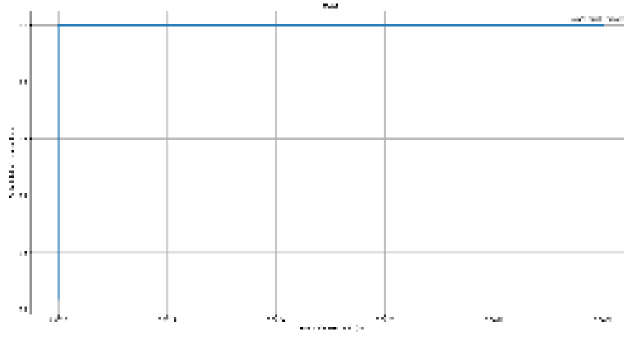

Figura 6: ECDF dos resultados associados a junção de Virtual Threads

4.4 Mudar Contexto (Yield) de Threads

Método	Tempo médio (s)	Desvio padrão (s)
Thread	14.216s	0.030s
Virtual Thread	0.019s	0.002s

Tabela 4: Desempenho na mudança de contexto de Threads

A operação de troca de threads, conhecida como YieldThreads, ressalta uma diferença significativa nas abordagens de manipulação de Threads, conforme evidenciado na Tabela 4, na Figura 7 e na Figura 8. Enquanto a troca de threads na biblioteca Thread demanda aproximadamente 14,216 segundos, as Virtual Threads demonstram uma eficácia superior, requerendo apenas 0.019 segundos.

Diante dessa discrepância, conduzimos um teste t de Student, com um nível de significância de 0,05, para investigar se há uma diferença significativa nos tempos médios de troca de contexto entre Virtual Threads e Threads. A hipótese nula sugere que não há diferença significativa entre os métodos, enquanto a alternativa indica uma disparidade significativa.

Os resultados do teste revelam um valor de p próximo a 0, indicando uma diferença estatisticamente significativa entre os tempos médios de execução dos métodos. Portanto, rejeitamos a hipótese nula e concluímos que há uma diferença substancial nos tempos médios de troca de contexto entre Virtual Threads e Threads.

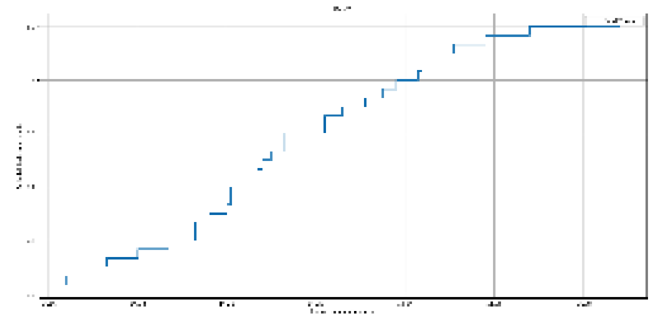

Figura 7: ECDF dos resultados associados a mudança de contexto de Threads

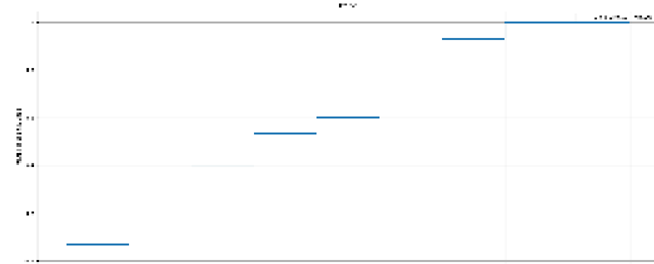

Figura 8: ECDF dos resultados associados a mudança de contexto de Virtual Threads

4.5 Ameaças à Validade

O estudo focalizou especificamente em um modelo de máquina, submetido a uma carga de estresse particular. É crucial observar que, caso essa máquina fosse exposta a uma carga de trabalho mais intensa, os resultados poderiam variar, uma vez que o sistema poderia ficar sobrecarregado e não funcionar corretamente. Por outro lado, seria possível determinar o limite de operação do sistema e coletar dados para otimizar o desempenho da máquina. Além disso, uma carga de estresse mais baixa poderia resultar em resultados semelhantes entre Threads e Virtual Threads, levando a conclusões errôneas sobre as diferenças entre essas duas abordagens.

Outro aspecto crucial a ser considerado é a ausência de diversidade nas configurações de infraestrutura, como máquinas com variadas especificações. A realização dos testes do estudo em ambientes distintos poderia levar a resultados divergentes, uma vez que a JVM (Java Virtual Machine) poderia realizar otimizações específicas de acordo com o hardware da máquina utilizada. Essa variação na performance poderia surgir devido a diferenças em processadores, quantidade de memória RAM, velocidade de disco, entre outros elementos físicos que impactam diretamente a execução dos processos.

5. TRABALHOS RELACIONADOS

Sung [4] realizou um estudo comparativo entre Threads e Virtual Threads utilizando uma abordagem de Green Threads

implementada por uma biblioteca Java. No entanto, diferentemente do experimento atual, as Green Threads foram implementadas nativamente na SDK do Java, tornando-as mais otimizadas em comparação com a versão criada pela biblioteca. Ambos os estudos concluíram que as Green Threads são, em geral, mais eficientes do que as Threads nos métodos avaliados.

Gu, Lee e Cai [6] conduziram um estudo utilizando um benchmark para investigar questões de desempenho na execução de threads em duas plataformas multithread distintas: Windows NT e Solaris. Este estudo difere um pouco do experimento atual, pois focou exclusivamente na análise de performance de Threads Nativas, sem abordar as Green Threads. Apesar disso, pode servir como ponto de partida para compreender a eficiência das Threads em contextos variados, o que possibilita uma estimativa do desempenho das Virtual Threads em diferentes cenários.

Chen, Chang e Hou [7] examinaram os benchmarks de desempenho com diferentes números de núcleos de processador e threads de aplicativo para entender a escalabilidade. Em seguida, apresentou análises detalhadas de desempenho e escalabilidade correlacionando dados de desempenho de hardware de baixo nível com threads da JVM e componentes do sistema, incluindo eventos de espera de hardware e latências do sistema de memória. Esse estudo oferece dados valiosos que contribuem para uma compreensão mais profunda dos custos associados à utilização de Threads. Embora não trate de Green Threads, ele complementa o estudo atual ao fornecer métricas relevantes.

6. CONCLUSÃO E TRABALHOS FUTUROS

A análise dos resultados provenientes deste experimento revela de forma a disparidade entre as distintas abordagens de manipulação de threads. Destacou-se, sem dúvida, o melhor desempenho das Virtual Threads em todos os testes, quando contrastadas com a abordagem convencional de threads em Java.

Ao considerarmos os valores obtidos no benchmarking, torna-se evidente que sistemas com uma elevada quantidade de threads, na ordem de milhares, podem experimentar ganhos substanciais de desempenho ao adotar as Virtual Threads em sua arquitetura. Para estudos subsequentes, é plausível a realização de testes adicionais para avaliar o desempenho e as limitações da utilização de green threads em sistemas concorrentes Java. Isso inclui a portabilidade de sistemas concorrentes que fazem uso intensivo de threads em Java para green threads, e a posterior análise comparativa de desempenho em ambientes reais de uso, em vez de utilizar microbenchmarks. Essa abordagem permitirá uma compreensão mais precisa de como as green threads se comportam em situações práticas, proporcionando insights valiosos para a aplicação dessas tecnologias em ambientes de produção.

Outro aspecto relevante a ser explorado é a avaliação das possíveis limitações das green threads no contexto de sistemas que realizam operações de entrada e saída (E/S) de forma intensiva. Isso pode ser realizado por meio da comparação direta do desempenho entre sistemas que utilizam green threads e sistemas que empregam threads convencionais Java em cenários de aplicação que demandam grande quantidade de operações de E/S. Essa análise aprofundada permitirá verificar se as green threads conseguem oferecer um desempenho superior em ambientes de aplicação que lidam predominantemente com E/S, identificando suas vantagens e possíveis desafios nesse contexto específico.

7. AGRADECIMENTOS

Gostaria de expressar minha gratidão a Deus pela dádiva da vida. Agradeço profundamente aos meus pais por terem investido e confiado em mim durante toda a minha jornada, além do imenso amor e carinho que eles me fornecem todos os dias. Também agradeço a mim mesmo pelo esforço e dedicação que coloquei em cada etapa, superando desafios e fazendo escolhas que trouxeram mudanças significativas para minha vida nos últimos anos, especialmente na escolha do curso que tanto impactou positivamente meu caminho.

Expresso minha gratidão ao meu orientador pela paciência e orientação durante a construção do TCC. Por fim, não posso deixar de agradecer aos meus amigos - Jonatas, Marcos, Vitinho, Kleber, Ítalo, Luiz Gustavo, Bernard (e tantos outros que são parte fundamental da minha jornada) - pelas incontáveis memórias construídas ao longo do curso que serão lembradas com muita paixão pelo resto da vida. Em memória de Adriano, eterno didico que ficará guardado em nossos corações.

8. REFERÊNCIAS

[1] The top programming languages: https://octoverse.github.com/2022/top-programming-languages

[2] TANENBAUM, A. S. Sistemas operacionais modernos. São Paulo (SP): Pearson Universidades, 2015.

[3] Loom Proposal.md. Disponível em: <https://cr.openjdk.org/~rpressler/loom/Loom-Proposal.html>.

[4] SUNG, M. et al. Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread. Information Processing Letters, v. 84, n. 4, p. 221–225, nov. 2002.

[5] Java Microbenchmark Harness (JMH). Disponível em: <https://github.com/openjdk/jmh>.

[6] GU, Y.; LEE, B. S.; CAI, W. Evaluation of Java thread performance on two different multithreaded kernels. ACM SIGOPS Operating Systems Review, v. 33, n. 1, p. 34–46, jan. 1999.

[7] K. -Y. Chen, J. M. Chang and T. -W. Hou, "Multithreading in Java: Performance and Scalability on Multicore Systems," in IEEE Transactions on Computers, vol. 60, no. 11, pp. 1521-1534, Nov. 2011