Érick Gonçalves Cabral

Data Engineer
Full Stack Developer

# Uma Análise Prática do Desempenho de Virtual Threads em Java

Perfil no LinkedIn

24 de maio de 2025

**Resumo**

Este documento detalha um estudo prático comparando o desempenho de Virtual Threads e Platform Threads em Java sob cargas de trabalho I/O-bound e CPU-bound. O objetivo é fornecer clareza sobre os cenários onde as Virtual Threads oferecem vantagens significativas, utilizando código de exemplo, resultados de teste e uma análise das diferenças observadas.

# Sumário

# 1 Introdução

Com o lançamento do Projeto Loom e a introdução das Virtual Threads como um recurso padrão no Java 21, a comunidade de desenvolvedores tem explorado ativamente seu potencial. As Virtual Threads prometem simplificar o desenvolvimento de aplicações concorrentes de alta vazão, especialmente aquelas que lidam com um grande número de operações bloqueantes (I/O-bound).

Minha inspiração para pesquisar um pouco mais e claro, aprender sobre veio do post do Pedro Munhoz no Linkedin (Perfil do Pedro), para sair do abstrato, realizei um teste prático com o intuito de observar e entender o impacto das Virtual Threads em comparação com as tradicionais Platform Threads. Esta documentação visa compartilhar a metodologia, o código utilizado, os resultados obtidos e as conclusões tiradas desta investigação.

# 2 O Experimento

## 2.1 Configuração do Ambiente

Os testes foram conduzidos no seguinte ambiente:

- **JDK:** Eclipse Adoptium JDK 21.0.4.7-hotspot
- **Processador:** 8 núcleos disponíveis para a JVM
- **Sistema Operacional:** Windows

## 2.2 Código Utilizado

O código Java abaixo foi empregado para simular as tarefas I/O-bound e CPU-bound, e para gerenciar a execução utilizando tanto Platform Threads quanto Virtual Threads através de `ExecutorService`.

```
1  import java.util.concurrent.ExecutorService;
2  import java.util.concurrent.Executors;
3  import java.util.concurrent.TimeUnit;
4
5
6  public class ThreadComparisonDemo {
7
8      // --- Configuracoes para o cen rio I/O-Bound ---
9      static final int IO_TASK_COUNT = 10_000;
10     static final int IO_TASK_SLEEP_MS = 50;
11
12     // --- Configuracoees para o cen rio CPU-Bound ---
13     static final int CPU_CORE_MULTIPLIER = 2;
14     static final int CPU_TASK_COUNT = Runtime.getRuntime().
    availableProcessors() * CPU_CORE_MULTIPLIER;
15     static final long CPU_TASK_ITERATIONS = 30_000_000L;
16
17     static volatile double sharedCpuResultSink = 0;
18
19     static void ioBoundTask(int taskNumber) {
20         try {
```

```
            Thread.sleep(IO_TASK_SLEEP_MS);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println("I/O Task " + taskNumber + " interrupted.");
        }
    }

    static void cpuBoundTask(int taskNumber, long iterations) {
        double result = 0;
        for (long i = 0; i < iterations; i++) {
            result += Math.sin(i) * Math.cos(i) * Math.tan(i / (iterations / 10.0 + 1.0));
        }
        sharedCpuResultSink += result;
    }

    public static void main(String[] args) throws InterruptedException {
        System.out.println("JDK Version: " + System.getProperty("java.version"));
        System.out.println("Available processors: " + Runtime.getRuntime().availableProcessors());
        System.out.println("=======================================================");

        // --- Cenario I/O-Bound ---
        System.out.println("--- I/O-Bound Scenario ---");
        System.out.printf("Number of I/O tasks: %,d | Each task sleeps for: %d ms%n%n", IO_TASK_COUNT, IO_TASK_SLEEP_MS);

        int platformIoThreadPoolSize = 200;
        System.out.printf("Running I/O tasks with Platform Threads (Fixed Pool Size: %d)...%n", platformIoThreadPoolSize);
        ExecutorService platformIoExecutor = Executors.newFixedThreadPool(platformIoThreadPoolSize);
        long startTimePlatformIo = System.currentTimeMillis();
        for (int i = 0; i < IO_TASK_COUNT; i++) {
            final int taskNum = i;
            platformIoExecutor.submit(() -> ioBoundTask(taskNum));
        }
        platformIoExecutor.shutdown();
        platformIoExecutor.awaitTermination(5, TimeUnit.MINUTES);
        long endTimePlatformIo = System.currentTimeMillis();
        System.out.printf("Platform Threads (I/O): Total time = %,d ms%n%n", (endTimePlatformIo - startTimePlatformIo));

        System.out.println("Running I/O tasks with Virtual Threads...");
        ExecutorService virtualIoExecutor = Executors.newVirtualThreadPerTaskExecutor();
        long startTimeVirtualIo = System.currentTimeMillis();
        for (int i = 0; i < IO_TASK_COUNT; i++) {
            final int taskNum = i;
            virtualIoExecutor.submit(() -> ioBoundTask(taskNum));
        }
        virtualIoExecutor.shutdown();
        virtualIoExecutor.awaitTermination(5, TimeUnit.MINUTES);
```

```
67          long endTimeVirtualIo = System.currentTimeMillis();
68          System.out.printf("Virtual Threads (I/O): Total time = %,d ms%n
    %n", (endTimeVirtualIo - startTimeVirtualIo));
69          System.out.println("
    ------------------------------------------------------");
70
71          // --- Cenario CPU-Bound ---
72          System.out.println("\n--- CPU-Bound Scenario ---");
73          System.out.printf("Number of CPU tasks: %d | Iterations per CPU
     task: %,d%n%n", CPU_TASK_COUNT, CPU_TASK_ITERATIONS);
74          sharedCpuResultSink = 0;
75
76          int platformCpuThreadPoolSize = Runtime.getRuntime().
    availableProcessors();
77          System.out.printf("Running CPU tasks with Platform Threads (
    Fixed Pool Size: %d)...%n", platformCpuThreadPoolSize);
78          ExecutorService platformCpuExecutor = Executors.
    newFixedThreadPool(platformCpuThreadPoolSize);
79          long startTimePlatformCpu = System.currentTimeMillis();
80          for (int i = 0; i < CPU_TASK_COUNT; i++) {
81              final int taskNum = i;
82              platformCpuExecutor.submit(() -> cpuBoundTask(taskNum,
    CPU_TASK_ITERATIONS));
83          }
84          platformCpuExecutor.shutdown();
85          platformCpuExecutor.awaitTermination(5, TimeUnit.MINUTES);
86          long endTimePlatformCpu = System.currentTimeMillis();
87          System.out.printf("Platform Threads (CPU): Total time = %,d ms%
    n%n", (endTimePlatformCpu - startTimePlatformCpu));
88          sharedCpuResultSink = 0;
89
90          System.out.println("Running CPU tasks with Virtual Threads...")
    ;
91          ExecutorService virtualCpuExecutor = Executors.
    newVirtualThreadPerTaskExecutor();
92          long startTimeVirtualCpu = System.currentTimeMillis();
93          for (int i = 0; i < CPU_TASK_COUNT; i++) {
94              final int taskNum = i;
95              virtualCpuExecutor.submit(() -> cpuBoundTask(taskNum,
    CPU_TASK_ITERATIONS));
96          }
97          virtualCpuExecutor.shutdown();
98          virtualCpuExecutor.awaitTermination(5, TimeUnit.MINUTES);
99          long endTimeVirtualCpu = System.currentTimeMillis();
100         System.out.printf("Virtual Threads (CPU): Total time = %,d ms%n
    %n", (endTimeVirtualCpu - startTimeVirtualCpu));
101         System.out.println("
    ======================================================");
102      }
103 }
```

Listing 1: Código Java do Experimento (ThreadComparisonDemo.java)

## 2.3 Cenários de Teste

Dois cenários principais foram avaliados:

1. **I/O-Bound:** Simulação de 10.000 tarefas, onde cada tarefa realiza uma pausa de

50 $ms$ (`Thread.sleep(50)`) para mimetizar operações de I/O bloqueantes, como chamadas de rede ou acesso a banco de dados. As Platform Threads foram limitadas a um pool de 200.

2. **CPU-Bound:** Simulação de 16 tarefas (2× número de núcleos da CPU) que executam um loop computacionalmente intensivo com 30.000.000 de iterações. As Platform Threads foram configuradas com um pool de 8 threads (igual ao número de núcleos da CPU).

# 3 Resultados e Análise

A execução do código de teste produziu a seguinte saída no console:

```
JDK Version: 21.0.4
Available processors: 8
=====================================================
--- I/O-Bound Scenario ---
Number of I/O tasks: 10.000 | Each task sleeps for: 50 ms

Running I/O tasks with Platform Threads (Fixed Pool Size: 200)...
Platform Threads (I/O): Total time = 3255 ms

Running I/O tasks with Virtual Threads...
Virtual Threads (I/O): Total time = 506 ms

-----------------------------------------------------

--- CPU-Bound Scenario ---
Number of CPU tasks: 16 | Iterations per CPU task: 30.000.000

Running CPU tasks with Platform Threads (Fixed Pool Size: 8)...
Platform Threads (CPU): Total time = 14.667 ms

Running I/O tasks with Virtual Threads...
Virtual Threads (CPU): Total time = 14.300 ms

=====================================================
```

## 3.1 Cenário I/O-Bound

- **Platform Threads (pool de 200):** Tempo total = 3255 $ms$.
- **Virtual Threads:** Tempo total = 506 $ms$.

Como podemos ver, as Virtual Threads demonstraram uma superioridade relevante. Elas completaram as 10.000 tarefas em aproximadamente **15,55%** do tempo levado pelas Platform Threads, representando uma **redução de 84,45%** no tempo de execução, ou, em outras palavras, foram cerca de **6,43 vezes mais rápidas** ($3255/506 \approx 6,43$).

Isso ocorre porque as Virtual Threads são leves e não monopolizam uma thread do sistema operacional (SO) durante operações bloqueantes. Quando uma virtual thread executa `Thread.sleep()`, ela é "desmontada" da sua carrier thread (uma platform thread), permitindo que a carrier thread execute outras virtual threads. Com um pool de apenas 200 platform threads, cada uma teria que lidar, em média, com 50 tarefas bloqueantes em sequência, levando a um tempo total muito maior. As Virtual Threads, por outro lado, gerenciam essa concorrência massiva de forma eficiente.

Pense assim: essas 'tarefas virtuais' são superleves e espertas. Quando uma delas precisa esperar por algo (como uma pesquisa no google ou simplesmente esperar), ela não fica ocupando um dos 'ajudantes' principais do seu computador à toa. Em vez disso, ela 'dá licença' temporariamente, e o 'ajudante' principal fica livre para atender outras tarefas virtuais que estão "prontas".

Agora, imagine o oposto: se você tivesse poucos 'ajudantes' principais e cada um pegasse uma fila enorme de tarefas que ficam parando e esperando, uma atrás da outra. Tudo ficaria super lento, porque os 'ajudantes' ficariam presos nessas esperas.

Com as 'tarefas virtuais', mesmo que você tenha milhares delas precisando esperar ao mesmo tempo, o sistema dá conta do recado muito melhor. Elas se organizam de um jeito que não travam tudo, permitindo que o computador faça muito mais coisas ao mesmo tempo sem sufocar.

## 3.2 Cenário CPU-Bound

- **Platform Threads (pool de 8):** Tempo total = 14.667 $ms$.
- **Virtual Threads:** Tempo total = 14.300 $ms$.

Aqui, a diferença de desempenho foi mínima. As Virtual Threads foram marginalmente mais rápidas, cerca de **2,50%** $((14667 - 14300)/14667 \times 100\%)$. Essa pequena variação está dentro do esperado para flutuações normais de execução e não indica uma vantagem intrínseca das Virtual Threads para cargas puramente computacionais.

Ambas as abordagens são limitadas pela capacidade de processamento dos 8 núcleos da CPU. As Virtual Threads, embora leves para criar, ainda executam o trabalho computacional em carrier threads, cujo número é, por padrão, igual ao de processadores disponíveis. Portanto, não há um aumento na capacidade de processamento paralelo para este tipo de tarefa, abaixo outro exemplo que comprova o ponto, visto que dessa vez a Plataform Threads superou a Virtual Threads com a mesma amostra.

```
--- CPU-Bound Scenario ---
Number of CPU tasks: 16 | Iterations per CPU task: 30.000.000

Running CPU tasks with Platform Threads (Fixed Pool Size: 8)...
Platform Threads (CPU): Total time = 12.903 ms

Running CPU tasks with Virtual Threads...
Virtual Threads (CPU): Total time = 13.403 ms
```

- **Platform Threads (pool de 8):** Tempo total = 12.903 $ms$.
- **Virtual Threads:** Tempo total = 13.403 $ms$.

# 4 Discussão sobre Escalabilidade e Limites

A melhoria de 6,43 vezes no cenário I/O-bound, observada com 10.000 tarefas, é um indicativo claro do potencial das Virtual Threads. Se aumentássemos o número de tarefas (e.g., para 100.000 ou mais), mantendo o pool de Platform Threads fixo e limitado, é altamente provável que esse fator de ganho das Virtual Threads se tornasse ainda mais expressivo. Isso ocorre porque o tempo de execução das Platform Threads (com pool fixo) tende a degradar-se de forma mais acentuada com o aumento do número de tarefas bloqueantes, enquanto as Virtual Threads são projetadas para lidar com essa escala de concorrência bloqueante de forma muito mais eficiente.

No entanto, é crucial entender que nem as Virtual Threads possuem escalabilidade infinita. Seus limites práticos incluem:

- **Consumo de Memória (Heap):** Embora cada virtual thread seja significativamente mais leve que uma platform thread em termos de consumo de stack, milhões delas ainda consumirão uma quantidade considerável de memória no heap para seus metadados e para as estruturas de dados que utilizam.
- **Overhead de Gerenciamento da JVM:** A JVM necessita de recursos de CPU para criar, agendar, estacionar (park) e liberar (unpark) um grande número de virtual threads. Em volumes extremamente altos, esse overhead de gerenciamento pode se tornar um novo gargalo.
- **Limites de Recursos Externos:** A capacidade da rede, a latência do disco, o número máximo de conexões de um banco de dados ou os limites de taxa de APIs externas continuarão sendo fatores limitantes. As Virtual Threads permitem que sua aplicação sature esses recursos mais facilmente, mas não podem aumentar a capacidade intrínseca desses sistemas externos.
- **Saturação do Pool de Carrier Threads:** Se as tarefas, mesmo sendo predominantemente I/O-bound, tiverem pequenas, mas frequentes, rajadas de trabalho CPU entre as operações de I/O, um número astronômico de tarefas poderia, teoricamente, pressionar o pool de carrier threads (que é limitado, por padrão, ao número de núcleos da CPU).

Apesar desses limites, o ponto fundamental é que as Virtual Threads brilham intensamente em **aplicações I/O-bound** (e isso é realmente muito bacana, ver o ganho me fez sorrir), onde as tarefas passam a maior parte do tempo esperando por respostas externas. Nesses cenários, elas permitem uma utilização de recursos drasticamente melhor e um aumento significativo no throughput, possibilitando que um servidor lide com muito mais requisições concorrentes com o mesmo hardware.

É importante também mencionar algumas **limitações e considerações atuais**. Uma delas é o fenômeno de "pinning" da carrier thread. Se uma virtual thread executa código dentro de um bloco `synchronized` extenso que realiza uma operação bloqueante, ou se executa um método nativo (JNI) que bloqueia, a carrier thread pode ficar "presa" (pinned) à virtual thread. Isso significa que a carrier thread não pode ser liberada para executar outra virtual thread, reduzindo temporariamente os benefícios de escalabilidade para aquela operação específica. Bibliotecas legadas ou código que faz uso intensivo de `synchronized` em torno de operações de I/O podem precisar de atenção ou adaptação para se beneficiarem plenamente das virtual threads. Mas isso com certeza será resolvido pela comunidade!

# 5 Conclusão

Os testes práticos e a discussão subsequente confirmaram os principais aspectos das Virtual Threads em Java e demonstraram o avanço que significa:

1. Para cargas de trabalho **I/O-bound**, as Virtual Threads oferecem um ganho de desempenho e escalabilidade substancial em comparação com Platform Threads tradicionais. Elas simplificam a escrita de código concorrente para I/O, permitindo um estilo de bloqueio sequencial que escala eficientemente.

2. Para cargas de trabalho **CPU-bound**, as Virtual Threads apresentam um desempenho comparável a um pool de Platform Threads devidamente dimensionado. O gargalo continua sendo a capacidade de processamento da CPU.

3. Embora elevem drasticamente o teto de escalabilidade para I/O, as Virtual Threads possuem limites práticos (memória, overhead de gerenciamento, capacidade de recursos externos) e considerações importantes como o "pinning" de carrier threads em conjunto com certas construções de código (como `synchronized` em operações bloqueantes).

Este experimento mostra como é crucial entender a carga de trabalho da aplicação antes de escolher uma estratégia de concorrência. As Virtual Threads são, sem dúvida, uma adição poderosa e transformadora ao ecossistema Java, democratizando o desenvolvimento de aplicações de alta performance e alta concorrência.

# Referências


- JEP 444: Virtual Threads. OpenJDK. Disponível em: https://openjdk.org/jeps/444.

- Oracle. Documentação Oficial sobre Virtual Threads (Exemplo). Disponível em: URL da documentação da Oracle.

- Munhoz, Pedro. Post que me inspirou a escrever sobre o assunto. Disponível em: Perfil de Pedro Munhoz.