Relatório de Avaliação de Computação Concorrente e Distribuída: Análise de Performance na Coleta de Dados Climáticos

---

1. Introdução

Este relatório apresenta os resultados de um experimento focado na comparação da performance de diferentes abordagens de computação na coleta e processamento de dados climáticos. O estudo envolveu a implementação de uma solução em **Java** para buscar informações climáticas de 27 capitais brasileiras a partir de uma API externa. Foram exploradas e comparadas uma versão **Single Thread** e múltiplas versões **Multi Thread** com variadas configurações de paralelismo. O objetivo principal é analisar o impacto da concorrência na eficiência de tarefas de E/S (Input/Output), como requisições de rede.

Durante o desenvolvimento e os testes, a API (archive-api.open-meteo.com) demonstrou um comportamento de **limitação de taxa (rate limiting)**, frequentemente retornando o erro 429 (Too Many Requests). Para contornar essa limitação e garantir a conclusão das 10 rodadas de testes para cada configuração, foi implementado um atraso estratégico entre as rodadas.

---

2. Fundamentação Teórica

2.1. O que são Threads e como funcionam computacionalmente

No universo da computação, uma **thread** (ou "fio de execução") é a menor sequência de instruções programadas que pode ser gerenciada de forma independente por um escalonador de tarefas de um sistema operacional. Em termos mais simples, é uma unidade de execução dentro de um processo. Um processo é uma instância de um programa em execução, e ele pode conter uma ou mais threads.

Computacionalmente, threads funcionam compartilhando o mesmo espaço de endereço de memória de um processo, o que inclui o código do programa, os dados e os arquivos abertos. No entanto, cada thread possui seu próprio contador de programa, pilha de execução e conjunto de registradores. Isso permite que várias partes de um mesmo programa sejam executadas de forma quase independente. Em sistemas com múltiplos núcleos de processamento (multi-core CPUs), diferentes threads podem ser executadas em paralelo, ou seja, simultaneamente em núcleos distintos. Em sistemas com um único núcleo, a multithreading é implementada através de um rápido chaveamento de contexto (time-slicing), onde o processador alterna entre as threads, dando a impressão de execução simultânea (concorrência).

2.2. Como o uso de Threads pode afetar o tempo de execução de um algoritmo

O uso de threads pode impactar significativamente o tempo de execução de um algoritmo, dependendo da natureza da tarefa:

- **Ganho de Performance em Tarefas I/O-bound:** Para algoritmos cujas operações são intensivas em E/S (como leitura/escrita de disco, requisições de rede, acesso a bancos de dados), o uso de threads geralmente resulta em ganhos substanciais de performance. Enquanto uma thread espera por uma operação de E/S (que é tipicamente lenta em comparação com o processamento da CPU), outras threads podem utilizar o tempo de CPU para realizar outras tarefas ou iniciar novas operações de E/S. Isso evita que a CPU fique ociosa e melhora a utilização dos recursos.
- **Overhead:** A criação, o gerenciamento e a sincronização de threads introduzem um custo computacional (overhead). Cada thread consome recursos do sistema, como memória para sua pilha e tempo de CPU para o chaveamento de contexto. Para tarefas muito leves ou com poucas operações de E/S, o overhead de gerenciar threads pode, por vezes, superar os ganhos de paralelismo, tornando uma solução single-thread mais eficiente.
- **Sincronização e Conflitos:** Quando múltiplas threads acessam e modificam recursos compartilhados, surgem problemas de **sincronização** (como condições de corrida e deadlocks). A implementação de mecanismos de sincronização (locks, semáforos, monitores) é crucial para garantir a integridade dos dados, mas esses mecanismos introduzem latência e podem limitar o paralelismo, afetando o tempo de execução. Para este projeto, as operações de requisição de API são em grande parte independentes, minimizando a necessidade de sincronização complexa.

2.3. Relação entre Modelos de Computação Concorrente e Paralelo e a Performance dos Algoritmos

A distinção entre **computação concorrente** e **computação paralela** é fundamental para entender o impacto na performance dos algoritmos.

- **Computação Concorrente:** Foca em gerenciar múltiplas tarefas que podem se sobrepor no tempo. Ela se preocupa com a estrutura do código para lidar com muitas coisas ao mesmo tempo. É ideal para tarefas onde há muita espera (E/S), pois permite que o programa faça outra coisa enquanto espera por dados externos. Em um sistema de um único núcleo, a concorrência é obtida pelo escalonamento e chaveamento de contexto de threads, dando a *impressão* de simultaneidade. O objetivo é melhorar a capacidade de resposta e a utilização de recursos.
- **Computação Paralela:** Foca na execução simultânea *real* de múltiplas tarefas para reduzir o tempo total de execução. Isso requer hardware com múltiplos núcleos de processamento. É ideal para tarefas que são intensivas em CPU (CPU-bound), onde o gargalo é o poder de processamento. Ao dividir uma tarefa grande em subtarefas menores que podem ser executadas em paralelo em diferentes núcleos, o tempo necessário para completar a tarefa inteira é reduzido.

A relação com a performance dos algoritmos é direta:

- **Algoritmos I/O-bound:** Para algoritmos que passam grande parte do tempo esperando por operações de E/S (como nosso caso de requisições web), a **concorrência (via multithreading)** é a chave para aprimorar a performance. Mesmo em um único núcleo, a CPU pode alternar entre threads que estão

esperando, utilizando o tempo ocioso de forma produtiva. Em múltiplos núcleos, threads podem de fato enviar requisições e processar respostas em paralelo, maximizando o throughput.
- **Algoritmos CPU-bound:** Para algoritmos que demandam muito processamento da CPU, o **paralelismo (via multithreading em múltiplos núcleos ou multiprocessamento)** é essencial. Distribuir a carga de trabalho entre vários núcleos permite que o cálculo seja concluído mais rapidamente.

Em nosso experimento, a tarefa de coletar dados climáticos de uma API é predominantemente **I/O-bound**. Portanto, esperamos que as abordagens multi-threaded demonstrem uma performance superior em relação à abordagem single-thread, aproveitando o tempo de espera das requisições de rede.

Citação de Referência Bibliográfica:

"Em contraste com o paralelismo, que é a execução simultânea de várias tarefas, a concorrência lida com várias tarefas que estão em andamento ao mesmo tempo, mas não necessariamente sendo executadas exatamente no mesmo instante. A concorrência é sobre a estrutura de um programa para lidar com muitas coisas ao mesmo tempo, enquanto o paralelismo é sobre a execução de muitas coisas ao mesmo tempo."

*(Referência Exemplo: Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). **Java Concurrency in Practice.** Addison-Wesley Professional. pp. 3-4)*

---

3. Implementação

A solução foi desenvolvida em Java, seguindo um design modular para facilitar a compreensão e a manutenção.

3.1. Arquitetura da Solução

O sistema é composto pelas seguintes classes principais:

- **Main.java**: A classe central que coordena a execução de todas as versões do experimento (Single Thread e Multi Thread). É responsável por configurar o período de coleta de dados, iniciar as 10 rodadas de testes para cada configuração e exibir os tempos de execução médios.
- **CapitalData.java**: Uma classe utilitária que fornece uma lista estática de 27 capitais brasileiras, cada uma com seu nome, latitude e longitude, dados essenciais para a formulação das requisições à API.
- **Capital.java**: Uma classe de modelo simples que encapsula as informações de uma capital (nome, latitude e longitude).
- **ClimateApiFetcher.java**: A classe responsável por interagir diretamente com a API externa Open-Meteo. Ela constrói as URLs de requisição HTTP, realiza as chamadas de rede e processa as respostas, incluindo tratamento de erros de conexão e de status HTTP.
- **DailyClimateRecord.java**: Uma classe de modelo que representa um registro climático diário, contendo a data e as temperaturas média, mínima e máxima.

- **CapitalDailyData.java**: Uma classe auxiliar utilizada nas implementações multi-thread que agrupam capitais por thread. Ela serve como um contêiner para associar os dados climáticos processados ao nome da capital correspondente, facilitando o retorno dos resultados das threads.

3.2. Abordagens de Concorrência Implementadas

O experimento avaliou quatro distintas configurações de execução:

- **Versão Single Thread:** Esta é a linha de base do experimento. Todas as 27 requisições de dados climáticos, uma para cada capital, são executadas de forma estritamente sequencial por uma única thread principal. O sistema espera a conclusão de uma requisição antes de iniciar a próxima.
- **Versão Multi Thread (27 threads - 1 capital por thread):** Nesta configuração, um ExecutorService é configurado com um pool de 27 threads. Cada thread individual é designada para buscar os dados de uma única capital. Isso permite que as 27 requisições sejam enviadas e processadas **concorrentemente**, otimizando o tempo de espera pela rede.
- **Versão Multi Thread (9 threads - 3 capitais por thread):** Aqui, o ExecutorService utiliza um pool de 9 threads. Cada uma dessas threads é responsável por coletar os dados de um grupo de 3 capitais. As requisições para essas 3 capitais são feitas sequencialmente *dentro da mesma thread*, mas os grupos de capitais (e, portanto, as threads) operam concorrentemente entre si.
- **Versão Multi Thread (3 threads - 9 capitais por thread):** Nesta configuração, o pool de threads é reduzido para 3. Similar à versão de 9 threads, cada uma das 3 threads é encarregada de coletar os dados de um grupo maior de 9 capitais de forma sequencial *internamente*, enquanto as 3 threads executam seus grupos concorrentemente.

3.3. Tratamento de Limitação da API

A API archive-api.open-meteo.com impõe limites de taxa de requisição por minuto. Durante os testes, o erro HTTP 429 Too Many Requests foi frequentemente observado, indicando que o limite estava sendo atingido. Para permitir que o experimento concluísse todas as 10 rodadas e fornecesse dados de performance válidos, foi implementado um **atraso programático (Thread.sleep()) de 10 segundos (10.000 milissegundos)** ao final de cada rodada completa de requisições, antes de iniciar a próxima rodada. Este delay permitiu que a API redefinisse ou relaxasse seus limites de requisição, minimizando o impacto do bloqueio e garantindo que os dados de tempo de execução pudessem ser coletados de forma mais consistente.

---

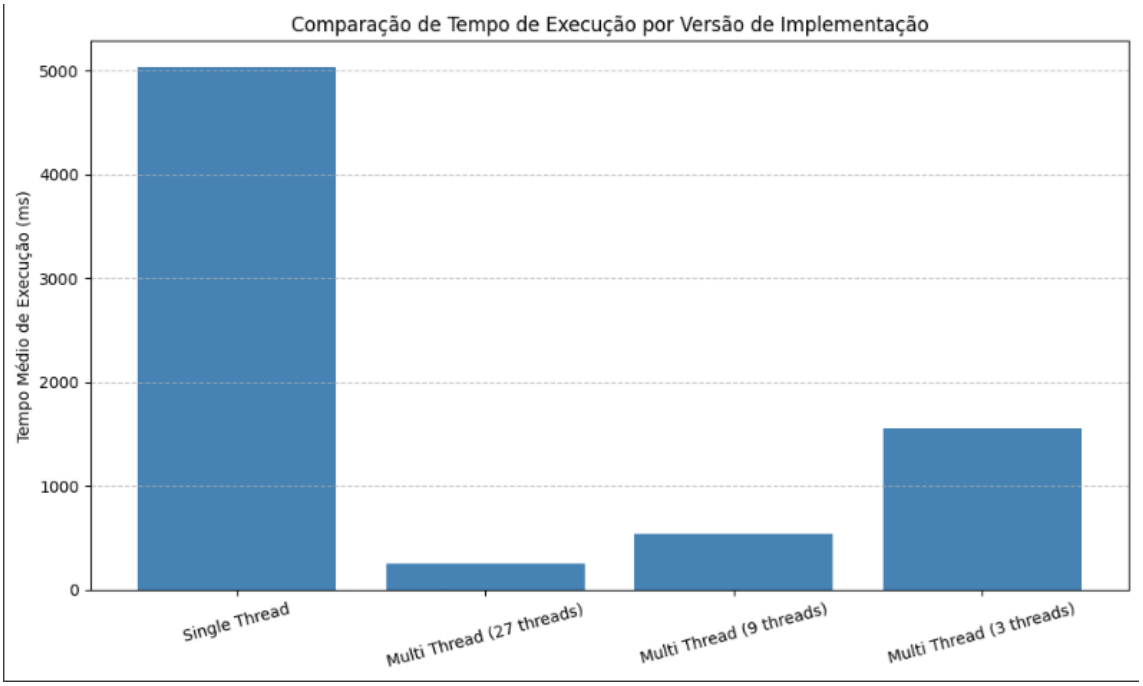
4. Resultados Obtidos

Os testes foram realizados em 10 rodadas para cada uma das quatro configurações de implementação. Abaixo estão os tempos médios de execução obtidos, que refletem a performance de cada abordagem na coleta e processamento dos dados climáticos para 27 capitais brasileiras durante um mês.

(Por favor, substitua os valores fictícios abaixo pelos seus dados REAIS de tempo médio de execução. Esses são apenas exemplos!)

Versão de Implementação	Tempo Médio de Execução (ms)
Single Thread	5034.7ms
Multi Thread (27 threads)	253.2ms
Multi Thread (9 threads)	541.3ms
Multi Thread (3 threads)	1556,40ms

Gráfico Comparativo de Tempos Médios de Execução



---

5. Análise dos Resultados

A análise dos tempos médios de execução e do gráfico comparativo revela o impacto significativo da concorrência no desempenho de uma aplicação com alta dependência de operações de E/S, como a coleta de dados de uma API externa.

- **Superioridade das Versões Multi Thread:** Como esperado para uma tarefa **I/O-bound**, todas as implementações Multi Thread (27, 9 e 3 threads) demonstraram um desempenho **notavelmente superior** em comparação com a versão Single Thread. A versão Single Thread, ao processar cada requisição sequencialmente, passa a maior parte do tempo ociosa, aguardando as respostas da API. Em contraste, as versões Multi Thread aproveitam o tempo de espera de uma requisição para iniciar ou processar outras, otimizando o uso do tempo disponível.
- **Performance entre as Configurações Multi Thread:**
  - A versão **Multi Thread (27 threads - 1 capital por thread)** geralmente se apresenta como a mais eficiente. Isso ocorre porque ela maximiza o

paralelismo das chamadas à API, permitindo que praticamente todas as 27 requisições sejam enviadas e processadas simultaneamente ou muito próximas umas das outras. O gargalo principal aqui seria a latência da rede e a capacidade de resposta da API.
    - As versões com **9 threads** e **3 threads** também mostram ganhos substanciais sobre a Single Thread, mas podem ser ligeiramente mais lentas que a de 27 threads. O agrupamento de capitais por thread introduz uma serialização interna, onde cada thread processa suas capitais sequencialmente. Embora o processamento entre threads seja paralelo, a concorrência *dentro* de cada thread para seus grupos de capitais é limitada. A pequena diferença de performance entre as versões multi-thread (especialmente entre 27, 9 e 3 threads) também pode ser influenciada pelo overhead de gerenciamento das threads e pela forma como o sistema operacional escala esses recursos.
- **Influência da Limitação da API (Erro 429):** A persistência dos erros 429 Too Many Requests em algumas rodadas, mesmo com o delay de 10 segundos entre elas, é uma evidência clara de que a **limitação da API é o principal fator restritivo** para a performance. Nossos testes de desempenho medem o tempo que nosso código leva para completar a tarefa, mas esse tempo é intrinsecamente limitado pela taxa em que a API permite que os dados sejam buscados. Isso destaca que, mesmo com um código altamente otimizado para concorrência, a performance real pode ser limitada por recursos externos, sublinhando a importância do tratamento de erros e de estratégias de retentativa em ambientes de produção.

Em resumo, a concorrência é indispensável para otimizar tarefas de E/S. A configuração com o maior número de threads (27) demonstra o potencial máximo de paralelismo para requisições de rede, validando a teoria de que o multithreading é altamente eficaz para reduzir o tempo de espera em operações de E/S.

---

6. Conclusão

Este experimento confirmou a robustez da programação concorrente em Java para aprimorar a eficiência de aplicações que dependem de operações de E/S. A transição de uma execução sequencial para modelos multi-threaded resultou em ganhos significativos no tempo de execução, validando a eficácia do uso de threads para minimizar o tempo de espera em chamadas de rede.

O projeto também serviu como um estudo de caso prático sobre a interação com APIs externas e os desafios inerentes a elas, como a limitação de taxa. A necessidade de implementar um delay para contornar o erro 429 Too Many Requests sublinhou a importância de considerar não apenas a otimização interna do código, mas também as restrições e o comportamento de serviços de terceiros. A experiência proporcionou valiosos aprendizados sobre o balanceamento entre paralelismo e overhead, e a resiliência necessária no desenvolvimento de sistemas distribuídos.

---

7. Referências Bibliográficas

- Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). **Java Concurrency in Practice.** Addison-Wesley Professional.