UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

INF01008 - Programação Distribuída e Paralela

Prof. Claudio Geyer

Avaliação de APIs de Sincronização

Java Threads e Kotlin Coroutines

Grupo:

Felipe Zorzo Pereira - **00261760**

William Wilbert Vargas - **00274692**

---

1. Resumo

Neste trabalho procurou-se avaliar as APIs de programação concorrente em Java (*Threads*) e em Kotlin (*coroutines*), comparando-as. Para tanto, foram implementados em ambas as linguagens, da forma mais equivalente possível, três problemas concorrentes diferentes: multiplicação de matrizes, soma de n elementos e o problema dos filósofos. Sobre estes problemas foram realizados 64 testes em dois computadores com CPUs distintas, totalizando 128 testes.

Os resultados obtidos demonstram que, apesar de a criação de *coroutines* em Kotlin ser mais rápida que a criação de *threads* em Java, o desempenho das aplicações foi melhor em Java. Kotlin, por outro lado, se demonstrou muito mais justo na hora de escalonar tarefas para ocupar a CPU, além de permitir programação muito mais simples e parecida com a sequencial. Dado esses fatores, conclui-se que o programador deve analisar os requisitos de seu programa e, com base nas vantagens de cada API, escolher aquela que maior se adequa às suas necessidades.

2. Introdução

A linguagem Kotlin é uma linguagem de propósito geral que começou a ser desenvolvida em 2011, mas teve sua primeira versão estável lançada apenas em 2016. No entanto, em 2017 já foi adotada pelo Google como uma das linguagens oficialmente suportadas para o desenvolvimento móvel para dispositivos Android. A linguagem é compilada para a JVM e tem como uma de suas mais importantes vantagens a interoperabilidade com classes Java.

Apesar disso, também possui bibliotecas próprias que visam facilitar o desenvolvimento de aplicações. Uma dessas bibliotecas é a *kotlinx.coroutines*, cujo objetivo é fornecer um meio para o desenvolvimento de aplicações concorrentes; para tanto, permite a criação e sincronização de tarefas de forma fácil dentro do código. As *coroutines* são implementações dessas tarefas do programa concorrente e são desenvolvidas com o intuito de serem mais leves que *threads* convencionais.

Nesse trabalho esse mecanismo será avaliado de forma comparativa com o mecanismo padrão de *threads* em Java para a resolução de três aplicações concorrentes, que são: soma de N elementos, multiplicação de matrizes e o problema dos filósofos. A partir desses problemas será realizada uma análise comparativa buscando avaliar o tempo de execução e a justiça na execução das tarefas, ou seja, se cada tarefa recebe um tempo justo em relação ao tempo recebido pelas demais tarefas.

3. APIs avaliadas

3.1. Java Threads

O desenvolvimento de aplicações concorrentes em Java ocorre tipicamente com o uso de objetos da classe *Thread*, que correspondem a uma *thread* que deve ser criada e escalonada pela JVM. O código a ser executado por cada *thread* fica no método *run()* e a *thread* é iniciada quando o método *start()* é executado.

Para as implementações usadas neste trabalho, as *threads* foram implementadas como classes que estendem a classe *Thread* e sobrescrevem o método *run()* com a lógica correspondente a cada problema. A sincronização das *threads* foi realizada usando blocos *synchronized* e chamadas da função *join()* para as threads.

3.2. Kotlin Coroutines

Em Kotlin, para criar uma *coroutine* basta utilizar a diretiva launch{} e, dentro dela, escrever um bloco de código sequencial. Deve-se definir, também, o escopo em que a *coroutine* irá executar: em nossas implementações, foi utilizado o escopo global, que permite a execução concorrente das *coroutines*. Isso demonstra a facilidade de criar e programar tarefas concorrentes em Kotlin se comparado a Java: em Kotlin adicionamos apenas uma diretiva a mais ao código sequencial, que define que tal código será executado por uma *coroutine*.

A sincronização das *coroutines* é feita da mesma forma que em Java: por meio de blocos *synchronized* e pelas chamadas da função *join()*.

4. Problemas usados para teste

4.1. Soma de N elementos

O algoritmo concorrente para soma de N elementos funciona em níveis, onde a cada nível a metade das somas são executadas, no primeiro nível cada elemento é somado, de forma concorrente, com o elemento que o segue resultando em N/2 somas no primeiro nível, no nível seguinte um desses resultado é somado com o resultado seguinte e assim sucessivamente os níveis vão reduzindo em quantidade de soma pela metade até que somente uma soma reste no último nível. A figura 4.1.1 ilustra esse processo.

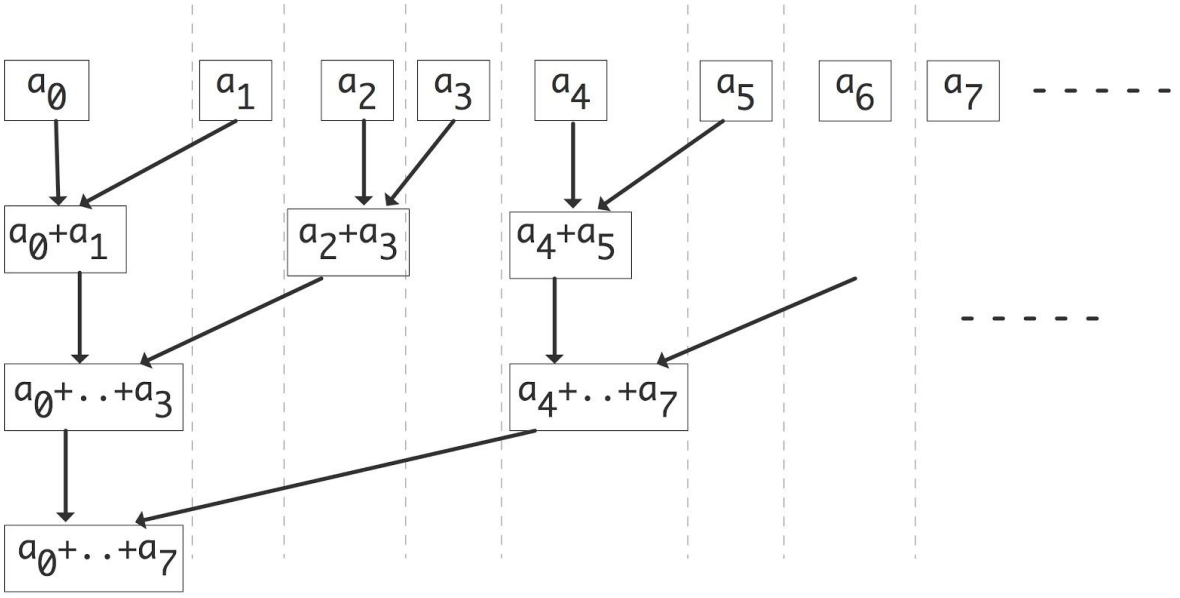

Figura 4.1.1: Ilustração do algoritmo de soma concorrente de N elementos.

Como cada tarefa envolvida nesse processo é muito simples, a criação e sincronização de tarefas tende a ser um fator determinante para o tempo de execução.

4.2. Multiplicação de matrizes

O uso de programas concorrentes para a resolução do problema de multiplicação de matrizes é bastante eficiente pois o problema permite a criação de várias tarefas que podem executar de forma paralela e, dependendo da implementação, sem a necessidade de sincronizações até o final de todas as tarefas.

As tarefas foram criadas a partir do problema dividindo as linhas da primeira matriz entre tarefas, cada tarefa é responsável somente por essas linhas e realiza todas as multiplicações que correspondem aos elementos dessas linhas com os elementos da outra matriz, salvando os resultados nas células correspondentes na matriz resultante.

Desse modo, o processamento de cada tarefa consiste em iterar por todos os elementos das suas linhas realizando as multiplicações desses elementos com elementos das colunas da segunda matriz e para cada linha e cada coluna somar o resultado dessas multiplicações que será então armazenado na matriz resultante.

As tarefas descritas são mais complexas, se comparadas com as tarefas do problema anterior e precisam de mais tempo de processamento, desse modo o tempo de execução do algoritmo deve ser determinado pela execução dessas tarefas concorrentes ao invés do tempo de sincronização e de criação das tarefas.

4.3. Problema dos filósofos

O problema dos filósofos é outro problema clássico da área de concorrência e busca explorar conceitos como o de deadlock e de efeitos que certa priorização de tarefas pode causar na execução do programa concorrente. O problema consiste em uma quantidade N de filósofos sentados em uma mesa circular e entre cada um deles existe um garfo, totalizando N garfos, de vez em quando os filósofos param de pensar e pegam o garfo que está a sua esquerda e o garfo que está a sua direita e comem. Para comer, um filósofo precisa ter os dois garfos para poder comer.

Para a abstração desse problema em código, se representa cada filósofo como uma tarefa que deve possuir o garfo à sua direita e o garfo à sua esquerda, onde cada garfo é representado como um lock para poder comer. Para a implementação o processo de comer se transformou na escrita em um arquivo que é posteriormente lido para que se possa determinar qual

tarefa possui mais acesso aos recursos compartilhados. Para evitar deadlocks adotou-se uma solução simples de fazer com que todos os filósofos com exceção de um deles pegue sempre o garfo da esquerda antes do garfo da direita, enquanto que o outro pega sempre o garfo da direita antes do garfo da esquerda.

Nesse problema o principal atributo que se busca medir não é o tempo de execução, mas se os recursos, nesse caso os garfos são compartilhados de maneira justa entre as tarefas.

5. Implementação

5.1. Multiplicação de matrizes

A implementação do problema de multiplicação de matrizes consistiu em criar tarefas responsáveis por calcular os resultados de uma quantidade q de linhas, em que q é dado pela quantidade de linhas na matriz dividido pela quantidade de tarefas que devem ser criadas. Logo, cada tarefa recebe uma quantidade uniforme de linhas.

Em ambas as versões, para realizar a implementação do problema, são criadas duas matrizes quadradas de tamanho n com valores aleatórios que serão multiplicadas. As linhas são divididas uniformemente entre p tarefas que são responsáveis por realizar as somas e multiplicações do algoritmo. É passado um índice para cada uma das threads que indica por qual porção de linhas essa tarefa é responsável. Para realizar a sincronização, executado um comando *join()* sobre cada uma das *p* tarefas.

5.2. Soma de n elementos

Para a soma de n elementos, o algoritmo é dividido em $\log_2(n)$ níveis. É criado um dado número de tarefas a cada nível, de acordo com o nível que o algoritmo está e o número máximo de tarefas que podem ser criadas, conforme o valor p definido pelo usuário; em cada nível, cada tarefa pode executar várias somas.

Cada tarefa calcula, então, o número de somas que deve executar, com base no nível atual e o número de tarefas criadas. As somas realizadas por cada tarefa são armazenadas em um vetor temporário, para evitar problemas de leitura e escrita concorrente no vetor original. Após isso, é necessário esperar que todas as tarefas de um nível terminem suas execuções antes de avançar para o próximo nível; isso é feito com a execução do comando *join()* em todas as tarefas do nível. Por fim, o vetor temporário é copiado para o vetor original, que será usado para leitura no próximo nível.

O n utilizado para realizar os testes com o algoritmo foi 33.554.432 (ou seja, 2^{25}). Os números a serem somados foram gerados aleatoriamente.

5.3. Problema dos filósofos

Para a implementação do problema dos filósofos, filósofos foram abstraídos para tarefas, garfos foram abstraídos para recursos com acesso concorrente e o ato de comer foi abstraído para a escrita de um identificador da tarefa em um arquivo. Dessa forma, a implementação consiste na criação de n recursos com acesso concorrente (garfos) e de n tarefas que tentam obter esses recursos. Para evitar *deadlocks*, uma dessas tarefas busca pegar primeiro o garfo da esquerda e depois o da direita, enquanto todas as outras buscam primeiro o da direita e depois o da esquerda.

Desse modo, o processamento de uma tarefa consiste em dois blocos *synchronized* encadeados: cada um deles busca obter o *lock* de um dos garfos. Uma vez que ambos os locks são obtidos, a tarefa escreve em um arquivo seu identificador. Todas as tarefas executam por um tempo pré-definido e após esse tempo o arquivo é lido para contar quantas vezes cada tarefa conseguiu escrever no arquivo.

6. Testes e resultados

Os testes realizados têm por objetivo a comparação das APIs (Java threads e Kotlin coroutines), considerando o tempo de execução e a justiça no escalonamento das tarefas, para tanto os testes dos programas variam parâmetros como a quantidade de tarefas criadas e o tamanho do problema (tamanho das matrizes ou quantidade de filósofos).

Ao todo foram executados 64 testes, que foram executados de forma independente em dois computadores para reduzir o viés dos resultados com um *hardware* específico. Os testes executados foram os seguintes:

- **Multiplicação de matrizes**: matrizes de 3 tamanhos (512x512, 1024x1024 e 2048x2048) e com 7 quantidades de tarefas diferentes (1, 2, 4, 8, 16, 32 e 64). Cada um desses testes foi executado em ambas as APIs, totalizando 42 testes.
- **Soma de N elementos**: para esse problema variou-se a quantidade de tarefas criadas (1, 2, 4, 8, 16, 32, 64) e, novamente, os testes foram executados de forma independente em ambas as APIs, resultando em 14 testes.
- **Problema dos filósofos**: para o problema dos filósofos variou-se a quantidade de filósofos competindo pelos garfos e avaliou-se a média de vezes que as tarefas conseguem escrever num arquivo e o desvio padrão dessas vezes. Por meio do desvio padrão se avaliou a justiça,

uma vez que quanto maior for o desvio padrão maior é a diferença entre a quantidade de vezes que as tarefas conseguem acesso aos recursos, e, portanto, menos justo é o escalonamento. Para os testes foram usadas quatro valores diferentes para a quantidade de filósofos (5, 25, 125, 625) que foram executados tanto na versão com Java threads quanto na versão com as co-rotinas de Koltin, resultando em 8 testes.

Para executar os testes foram usados os dois processadores abaixo:

- **PC1**: Processador AMD FX-6300 com três núcleos físicos e seis virtuais.
- **PC2**: Processador i7-7700k com quatro núcleos físicos e oito virtuais.

6.1. Multiplicação de matrizes

A multiplicação de matrizes de tamanho 512 por 512 finalizou de forma extremamente rápida em ambos os processadores; entretanto, mesmo nesses tempos pequenos é possível ver que a performance de Kotlin é relativamente pior que a de Java. Inicialmente não esperava-se isso, uma vez que *coroutines* são mais leves que *threads* e, portanto, sua criação é mais rápida. Entretanto, como a multiplicação de matrizes envolve muitas operações aritméticas (n^3 multiplicações, além de $n^2(n-1)$ somas) e o número máximo de tarefas que criamos foi 64, provavelmente o que dominou o tempo de execução foram as operações aritméticas. Isso pode ser visto nos dois gráficos abaixo:

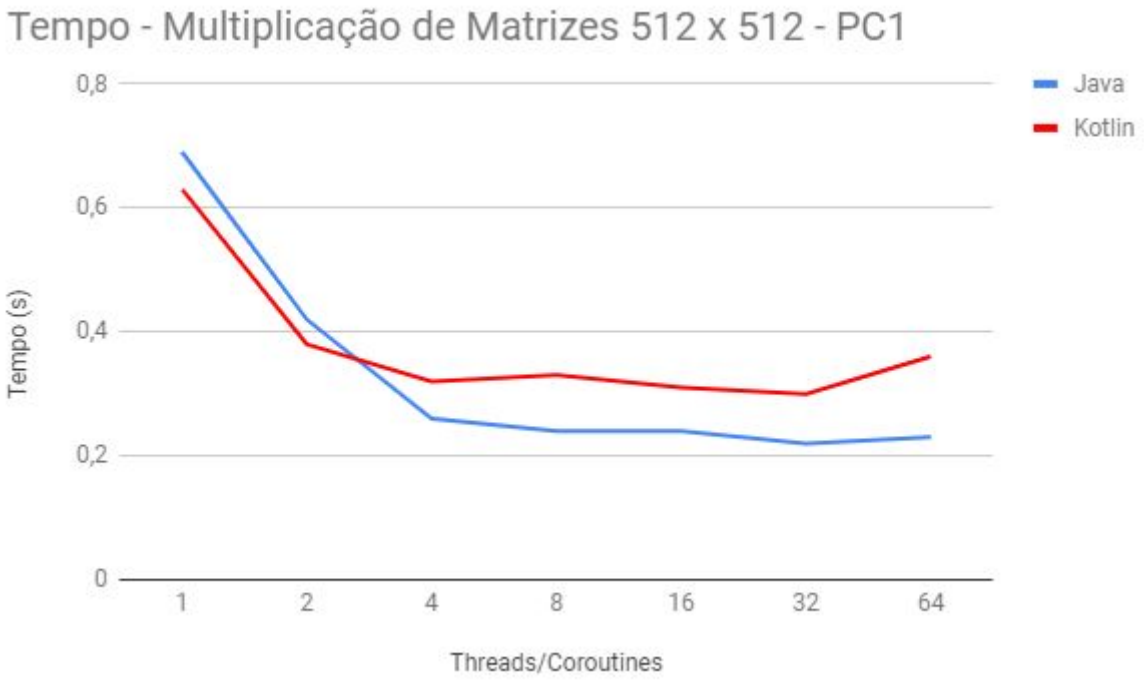

Figura 6.1.1: Multiplicação de matrizes 512x512 no PC1.

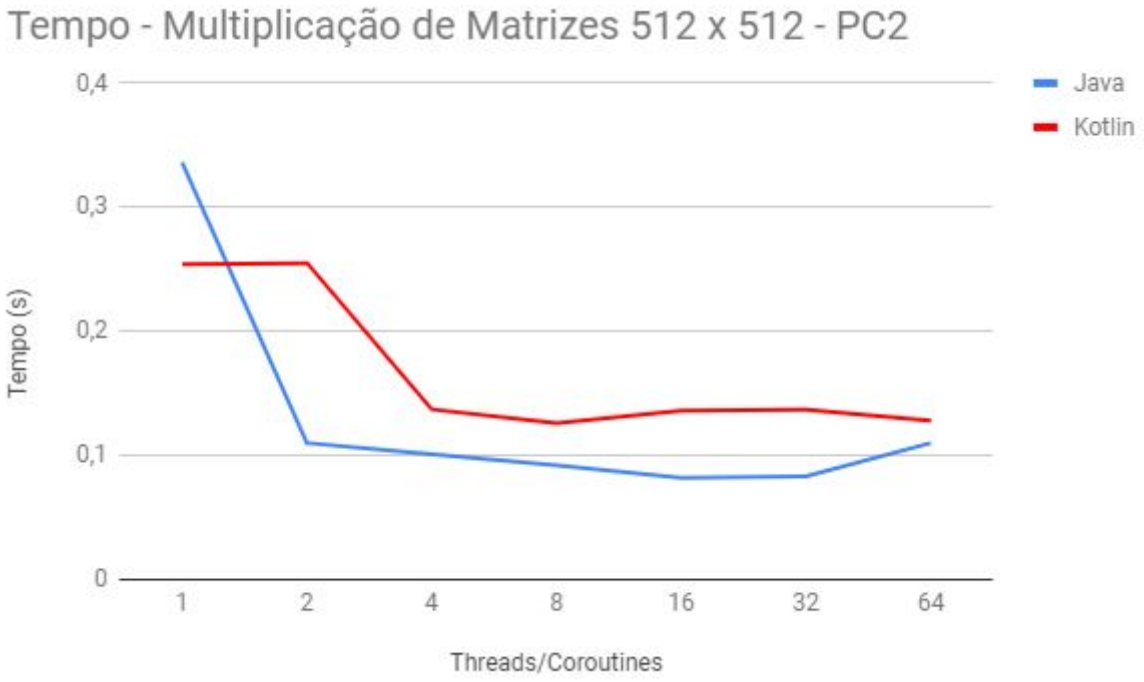

Figura 6.1.2: Multiplicação de matrizes 512x512 no PC2.

Com os maiores tempos de execução da multiplicação de matrizes 1024 por 1024, fica mais claro algo que não era tão nítido nos testes anteriores: quando a divisão de tarefas chega no limite de núcleos virtuais de um processador, a melhora de desempenho para de ocorrer. Apesar disso,

não há piora no desempenho, uma vez que não foi criado um número absurdamente grande de tarefas se comparado com o número de núcleos dos processadores. Ademais, notou-se que a implementação em Kotlin atinge o mesmo desempenho da implementação em Java, no PC2, a partir de quatro tarefas; apesar disso, com menos tarefas Java ainda se demonstrou superior.

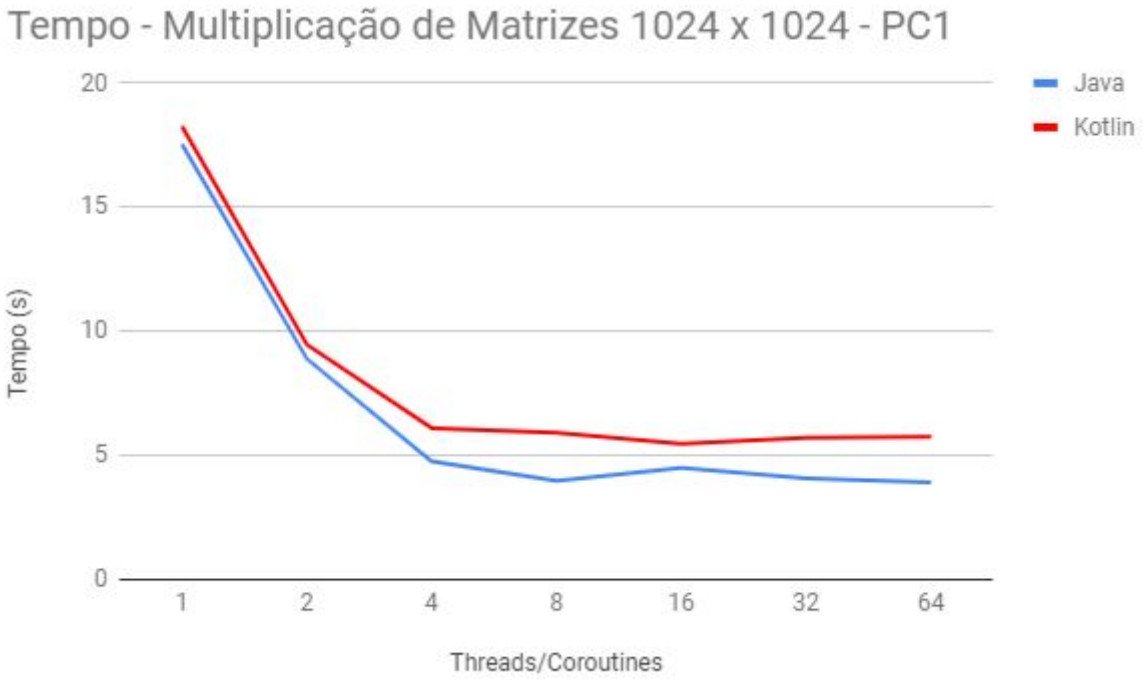

Figura 6.1.3: Multiplicação de matrizes 1024x1024 no PC1.

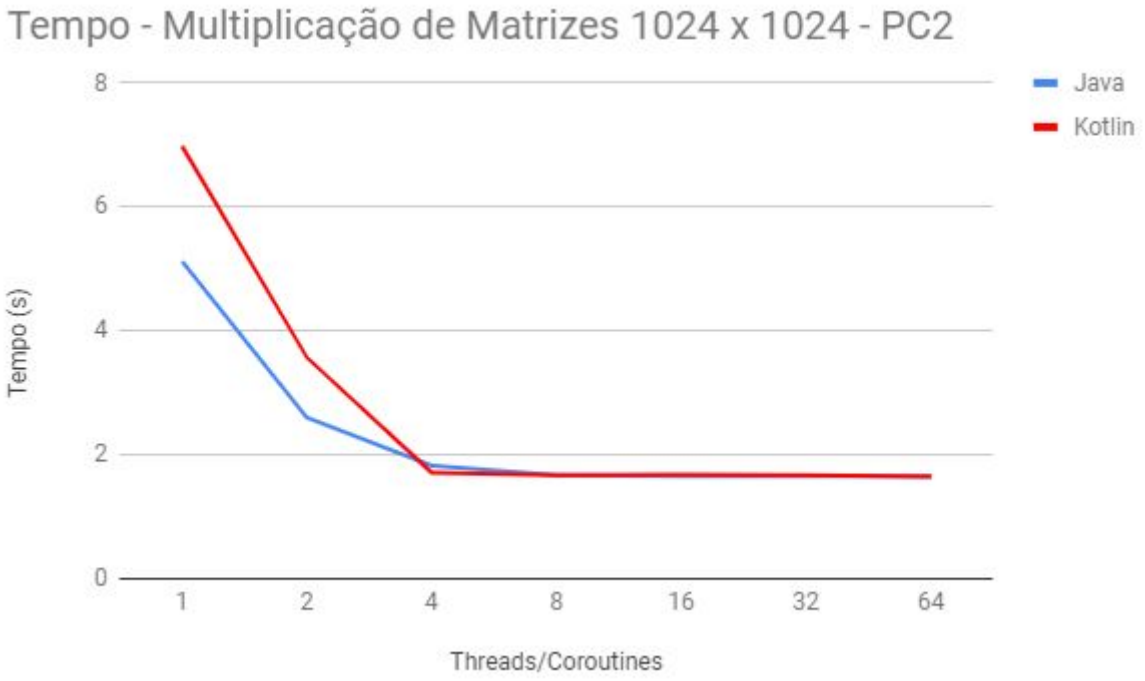

Figura 6.1.4: Multiplicação de matrizes 1024x1024 no PC2.

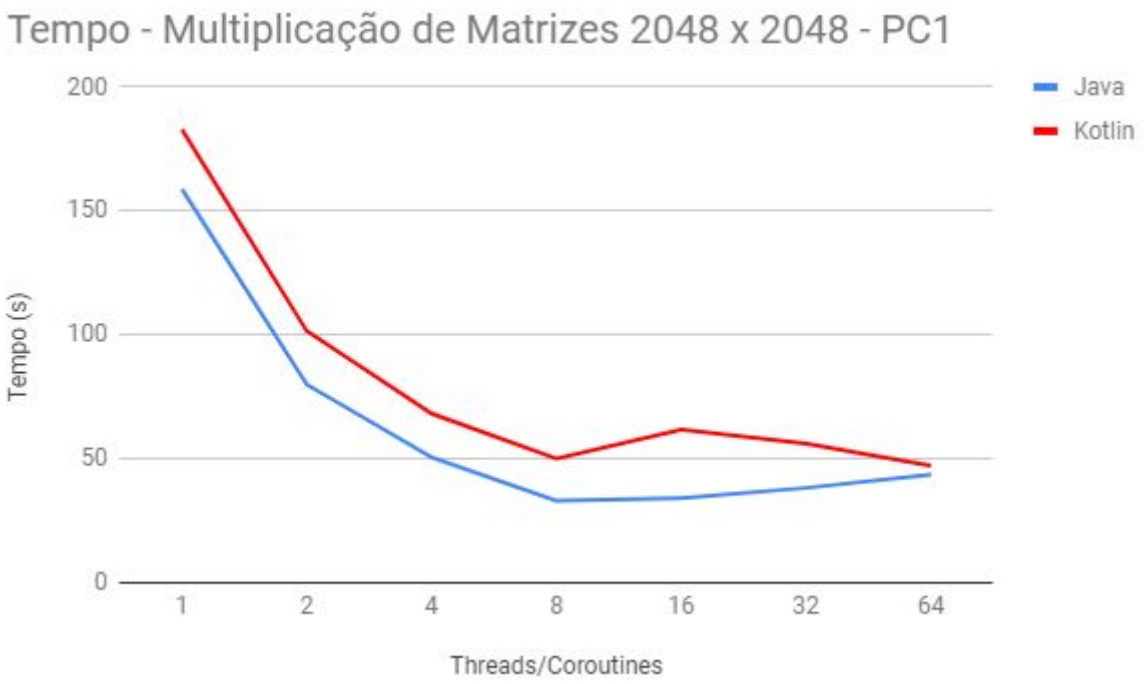

Figura 6.1.5: Multiplicação de matrizes 2048x2048 no PC1.

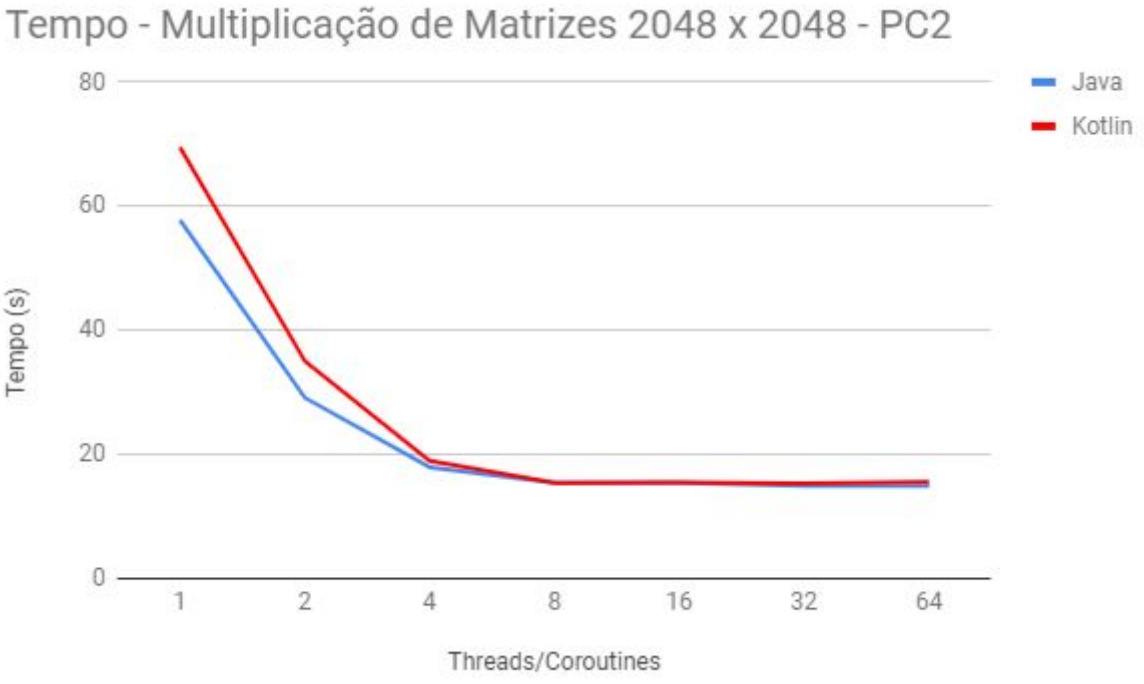

Figura 6.1.6: Multiplicação de matrizes 2048x2048 no PC2.

Por fim, a multiplicação de matrizes de tamanho 2048 por 2048 seguiu os mesmos padrões vistos até agora: Kotlin teve desempenho um pouco pior,

não há piora nem melhora de desempenho quando o número de tarefas ultrapassa o número de núcleos virtuais do processador e, para o PC2, a partir de quatro tarefas Kotlin tem um desempenho extremamente parecido com o de Java.

6.2. Soma de 33.554.432 elementos

A soma de elementos, apesar de utilizar um n muito maior do que aqueles testados com a multiplicação de matrizes, realiza bem menos operações aritméticas (apenas n-1). Com isso, pôde-se finalmente notar a diferença no tempo de criação de *threads* em Java e *coroutines* em Kotlin: como o tempo de execução não é mais dominado totalmente pelas operações aritméticas, é possível perceber que Kotlin possui um desempenho melhor, já que leva menos tempo criando as tarefas. Esse fato fica especialmente claro no PC2, a partir de 16 tarefas: o tempo de execução de Java cresce progressivamente, enquanto o de Kotlin se mantém bastante constante.

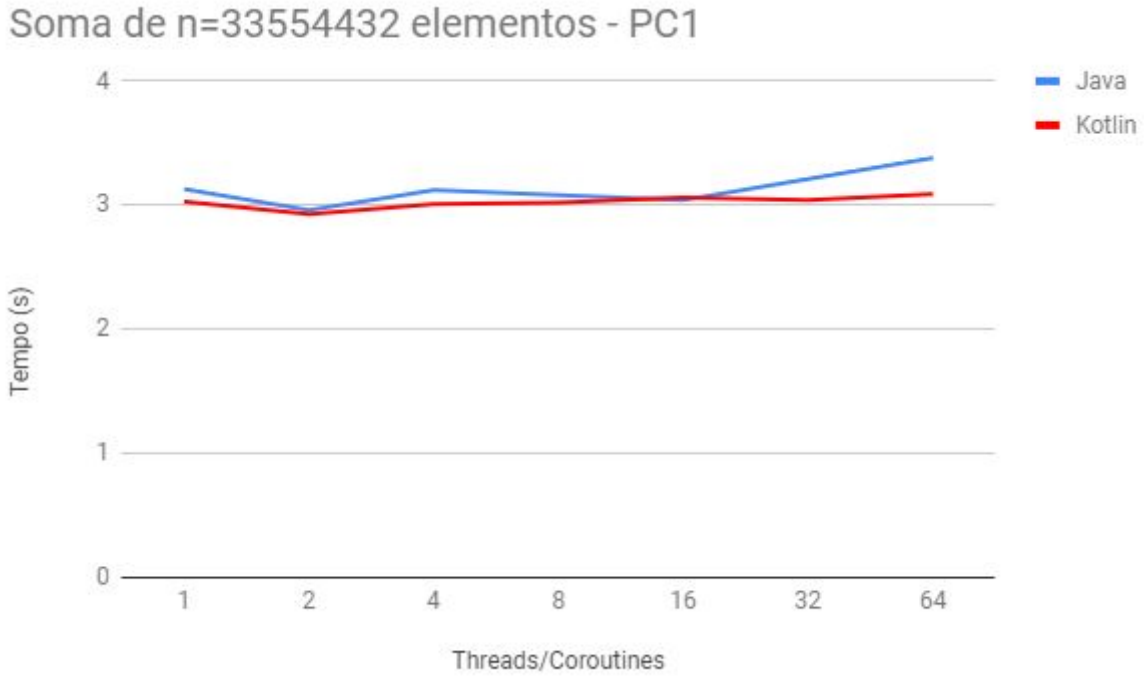

Figura 6.2.1: Soma de 33.554.432 elementos no PC1.

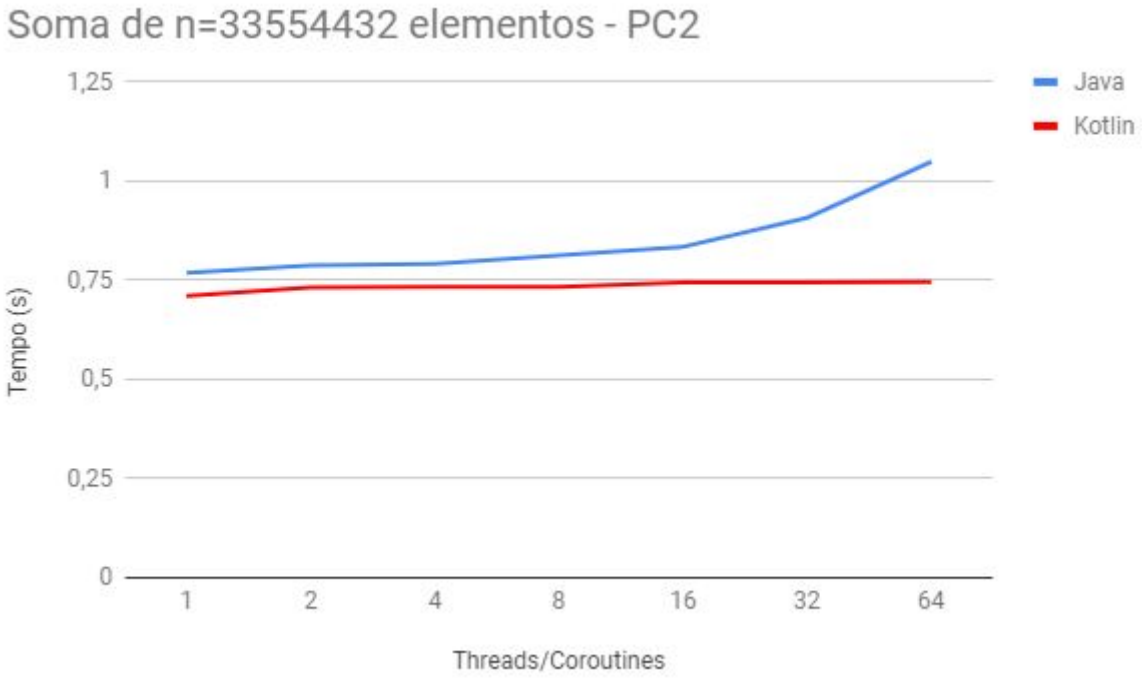

Figura 6.2.2: Soma de 33.554.432 elementos no PC1.

6.3. Problema dos filósofos

O último problema testado, por sua vez, trouxe alguns resultados interessantes. Em ambos os PCs o desvio padrão foi absurdamente maior em Java do que em Kotlin, demonstrando que Kotlin é mais justo na divisão de quem pode ocupar um núcleo da CPU. Em Java, apesar de nenhum filósofo morrer de fome, a grande maioria deles escrevia apenas uma vez no arquivo, com pouquíssimos filósofos escrevendo diversas vezes; em Kotlin essa divisão era bem mais uniforme.

Além disso, notou-se que o PC1 teve uma média de escritas muito maior que a do PC2. Apesar disso, em ambos os casos a média de escritas de Java e Kotlin num mesmo PC foi bem próxima, com exceção do caso de cinco filósofos no PC2, em que Java conseguiu uma média de escritas bem superior.

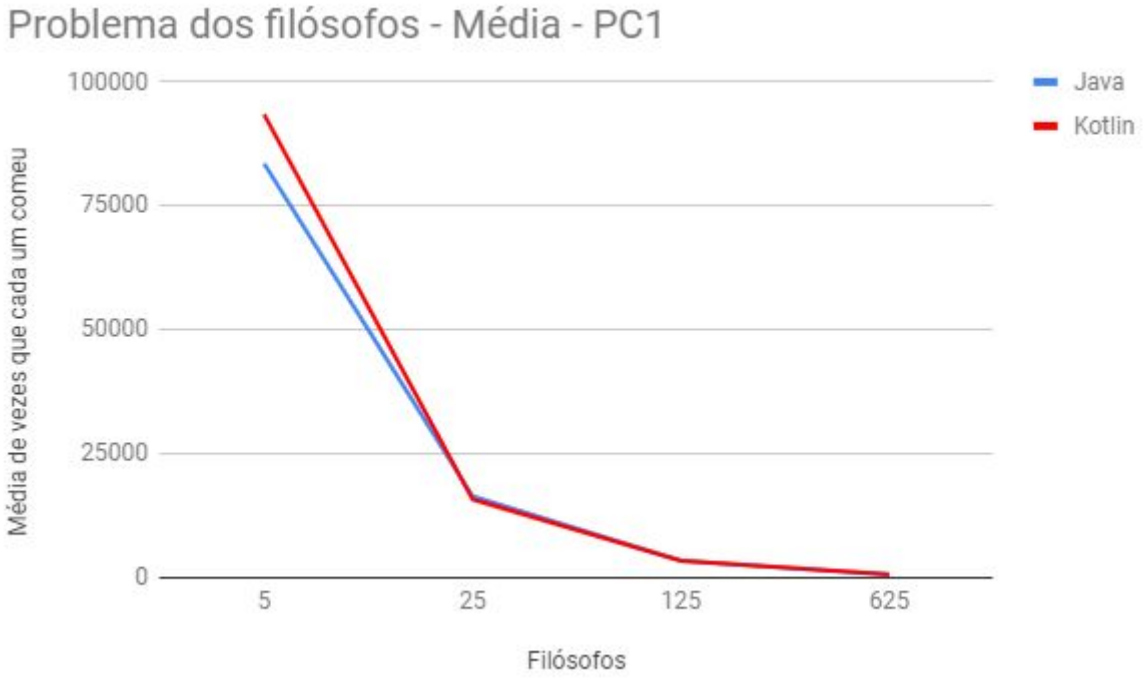

Figura 6.3.1: Média de escritas do problema dos filósofos no PC1.

Figura 6.3.2: Desvio padrão de escritas do problema dos filósofos no PC1.

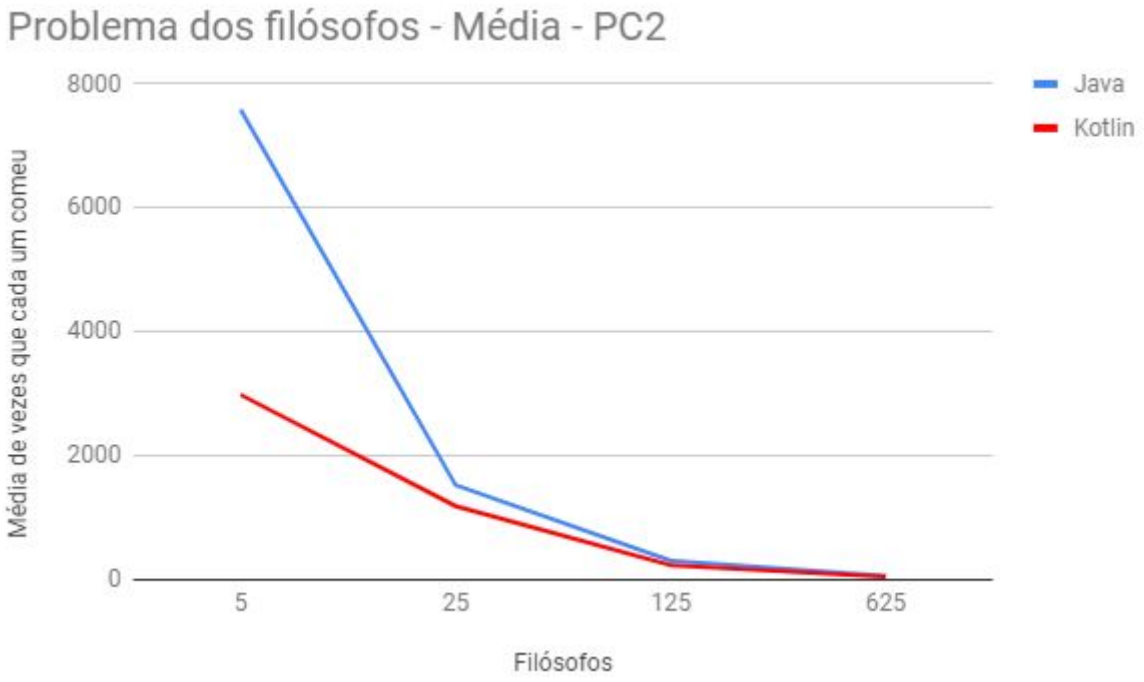

Figura 6.3.3: Média de escritas do problema dos filósofos no PC1.

Figura 6.3.4: Desvio padrão de escritas do problema dos filósofos no PC2.

7. Conclusão

Os testes realizados demonstraram que as *coroutines* de Kotlin são, de fato, mais leves e, portanto, mais rápidas de serem criadas do que as

threads de Java. Entretanto, se o programador tomar o cuidado de não criar um número enorme de tarefas, essa diferença é negligenciável, em especial se o tempo de execução das tarefas for longo o suficiente para dominar o tempo de execução do programa.

Levando em conta o fato de que a diferença no tempo de criação das tarefas é negligenciável, Java demonstrou desempenho um pouco superior ao de Kotlin, como exemplificado pelas multiplicações de matrizes.

Kotlin, por sua vez, possui a vantagem de ser extremamente mais justo que Java, como demonstrado no problema dos filósofos; além disso, a programação concorrente é mais simples, se aproximando bastante da programação sequencial.

Logo, a decisão final de quais APIs de concorrência utilizar fica a cargo do que o programador e do que ele necessita: mais desempenho, código mais simples e, portanto, de mais fácil manutenção, maior justiça na execução de tarefas e etc. Com base nesses conceitos, deve-se analisar as vantagens de cada linguagem e escolher aquela que melhor se adequa ao desejado.